Temporal Markov Decision Problems
—
Formalization and Resolution

—

THESIS

submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

delivered by the

University of Toulouse
Institut Supérieur de l'Aéronautique et de l'Espace

in the field of Artificial Intelligence.

Presented and defended by

Emmanuel Rachelson

on March 23rd, 2009.

—

JURY

Patrick Fabiani	Ingénieur de recherche — ONERA, thesis advisor.
Frédérick Garcia	Directeur de recherche — INRA, thesis advisor.
Michail G. Lagoudakis	Assistant Professor — Technical University of Crete.
Michael Littman	Professor — Rutgers University.
Rémi Munos	Directeur de recherche — INRIA.
Olivier Sigaud	Professeur — Université Paris 6.
Olivier Teytaud	Chargé de recherche — INRIA.

Abstract

This thesis addresses the question of planning under uncertainty within a time-dependent changing environment. Original motivation for this work came from the problem of building an autonomous agent able to coordinate with its uncertain environment; this environment being composed of other agents communicating their intentions or non-controllable processes for which some discrete-event model is available. We investigate several approaches for modeling continuous time-dependency in the framework of Markov Decision Processes (MDPs), leading us to a definition of Temporal Markov Decision Problems. Then our approach focuses on two separate paradigms. First, we investigate time-dependent problems as *implicit-event* processes and describe them through the formalism of Time-dependent MDPs (TMDPs). We extend the existing results concerning optimality equations and present a new Value Iteration algorithm based on piecewise polynomial function representations in order to solve a more general class of TMDPs. This paves the way to a more general discussion on parametric actions in hybrid state and action spaces MDPs with continuous time. In a second time, we investigate the option of separately modeling the concurrent contributions of exogenous events. This approach of *explicit-event* modeling leads to the use of Generalized Semi-Markov Decision Processes (GSMDP). We establish a link between the general framework of Discrete Events Systems Specification (DEVS) and the formalism of GSMDP, allowing us to build sound discrete-event compatible simulators. Then we introduce a simulation-based Policy Iteration approach for explicit-event Temporal Markov Decision Problems. This algorithmic contribution brings together results from simulation theory, forward search in MDPs, and statistical learning theory. The implicit-event approach was tested on a specific version of the Mars rover planning problem and on a drone patrol mission planning problem while the explicit-event approach was evaluated on a subway network control problem.

Keywords

Decision under uncertainty, Markov Decision Processes, hybrid time-dependent planning problems, modeling of time-dependent stochastic decision processes, control of implicit and explicit event-driven processes.

Acknowledgements

When the time comes to write the acknowledgements of a doctoral thesis, it means the manuscript is finished, or in its last stages of corrections. At least is this my case and it is with the mixed feelings of relief (this long writing is finished), pleasure (these last years have simply been extraordinary for me) and gratitude, that I start these last pages.

First of all, I would like to thank Rémi Munos and Olivier Sigaud, the two "rapporteurs" of the thesis for accepting to review this (long) document.

My next thanks go to the whole thesis jury for accepting to read the manuscript and attend the defense.

Then I would like to mention and thank the ONERA and all the personnel of the "System Control and Flight Dynamics" department for making this thesis financially and materially feasible. My gratitude goes in particular to the "Decision and Control" research team for providing a very welcoming and warm environment as well as high quality research stimuli during these last three years. There are too many people to mention here, but my gratitude goes to them all for making these years both an human and a scientific experience.

I also wish to thank the people with whom I had the occasion to work, at LAAS-CNRS and in the "Biometry and Artificial Intelligence" research team of INRA.

Similarly, as I look back on these three years, I realize I am grateful for the many rich and challenging discussions I had with members of the MDP and Reinforcement Learning communities: I found many passionate people, working in a challenging domain and willing to share about their research.

I would also like to thank Sylvie Thiébaux in particular for all the advice and support she gave me during this last year. I wish you a nice trip back to Australia and look forward to hearing from you.

As mentioned people become closer and closer to my thesis' everyday life, I would like to express my gratitude to Jean-Loup Farges. At the end of my M.Sc. thesis, I did thank you already for your "subtle and efficient support". I did not realize at that time how true it was: Jean-Loup has been a rock, present and available at all times throughout the thesis, expressing his criticism on unexpected topics which greatly contributed to the rigor and

quality of this work.

While we were sharing the same office during my M.Sc. thesis, Florent Teichteil already took an active influence on my research ideas. Thank you for all these *four* years, for your advices and your support.

Then I would like to thank Patrick Fabiani, my first advisor, who started this thesis with — I guess — a completely different research idea than what I actually began investigating, but who found an interest in my "reshaped" topic and encouraged me to go further. The incredible energy you put into more or less successful attempts at being available for me only matches the accuracy of your advices.

Frédérick Garcia has been, I suppose, the perfect thesis advisor for me. I cannot make a list of all the reasons for which I should thank you: it is a whole, which starts with your kindness and goes all the way to the passion you put into research. In hard times as well as in good ones we exchanged more than just scientific ideas.

I guess these three years would not have been the same without the rather surprising group of Ph.D. students we were. So, quite impolitely since I include myself in the group, I would like to thank us all for all the good and the bad times we had all together at work as well as outside. These occasions cover too many times and places to mention. My thesis would never have been the same without *us*.

I would also like to address a deep and grateful thank to the anonymous inventor of the "coinche" (or "belote contrée") game. I owe you countless hours of frustration, discussions and strategic pleasure[1].

There are two friends and colleagues I would like to thank in particular. Let me start with Gregory Bonnet. Thanks for surviving to three years with me in the same office, your patience is legendary my friend! Thanks also for all the insightful discussions we had: on multi-agent systems, on bad and good movies, on Noam Chomsky, on the topology of potatoes and for the unforgettable "Subotaï the magnificent".

Finally, I cannot forget to mention and thank Julien Guitton. From your very accurate and refreshing points of view on planning, to the most improbable and solid of friendships, I have thousands of reasons to thank you. Let me simply thank you for the countless hours we spent talking about nothing and everything. Please collect the prize for the ONERA "coinche" championship we just won for the two of us.

—

I dedicate this thesis to all my friends, past and present.
You know how important you are to me.

[1] I must also mention that I owe you some inspiration for my research ideas, please contact me for royalties and quotations.

Contents

Notation conventions

$*$	The convolution operator
$\|\cdot\|_{I,\infty}$	$\|g\|_{I,\infty} = \sup_{x \in I} \|g(x)\|$
δ	Process step number, similar to decision epoch number
ρ	Execution path
σ	augmented SMDP+ state, corresponding to (s, t)
μ	TMDP outcome
A	Action space
a	Action
a^δ	The action at decision epoch δ
a_n	In finite action spaces, this is the n^{th} element of A
$F(\tau \| s, a)$	SMDP duration model (cumulative distribution function)
$f(\tau \| s, a)$	SMDP duration model (probability density function)
L	Bellman operator
L^π	Policy evaluation operator
$Pr(X = x)$	Probability that random variable X is equal to x
$P(s' \| s, a)$	MDP transition model
R	MDP reward model per transition: $R(s, a, s')$
r	MDP reward model per pair state-action: $r(s, a)$
S	State space
s	State
s^δ	The random variable "state" at process's step δ
s_μ	state associated with outcome μ in a TMDP model
s_n	In finite state spaces, this is the n^{th} element of S
V^*	Optimal value function
V^{π^*}, V^π	Value function of an optimal policy / of policy π

Part I

Introduction

1

Taking good decisions: from examples to Temporal Markov Decision Problems

This introductory chapter aims at providing a general overview of the context in which we will work throughout the thesis, progressively focusing on the specific domain at hand. We start at the "human" level, considering the general question of *Decision* and its different facets, motivating the various attempts at formalizing this question and highlighting the outreach of formal approaches. Then we enumerate a number of characteristics which define different fields in the study of decision-making, in order to position our topic of interest among the different branches of *Decision Sciences*. Our interest goes to the domains of *Planning under Uncertainty* and *Reinforcement Learning* in which we finally outline the class of *Temporal Markov Decision Problems*.

1.1 The question of decision

"A human being is a deciding being."
Viktor E. Franckl — Man's Search for Meaning

Let us introduce this thesis with questions of various importances:

- Turn right or turn left? Which is the best way to go to work?
- Buy company A's shares or prefer company B's? Buy them now or later?
- Invest in the Third World's growth today or develop the inner market to participate later?
- Which dimensions should I choose for this aircraft beam?
- Should I plan a meeting with Fred before writing this report?
- Should I intervene when I see a man being attacked?
- How do I assign priorities to message transmissions in this network?
- Is this group of pixels a building? A road?

- How to optimize the maintenance of this satellite constellation?

Performing as well as possible, taking good decisions, choosing the correct (best?) option, all these constitute a common requirement and somehow a natural behavior in everyday life. This problem of taking decisions — and, preferably, taking good decisions — is, at the same time, one of the first problems to which we are confronted in our life and, paradoxically, one of the hardest dilemmas to solve, even with a lifetime of experience.

Taking a decision implies considering many aspects: ethics, physical constraints, long-term consequences, customer requirements, etc. The notions of *intelligence*, *adaptability*, *responsibility* — shared by various domains such as Philosophy, Psychology, Artificial Intelligence, Control Theory, Operation Research, etc. — all find their motivation in the question of taking the right decision and applying it. If one takes the idea of choice away, these three notions loose most of their meaning.

An old dream of Artificial Intelligence is to build an autonomous thinking machine, able to take decisions by itself. Since the early years of Computer Science and Artificial Intelligence, the definition of such an "intelligence" has shifted from the Computer Science-inspired notion of being able to solve hard computational problems, towards an idea of decisional autonomy. From a vision centered on the computer — the super-calculator, solving difficult combinatorial problems —, Artificial Intelligence has emerged with a vision focused on the *behavior* of agents which mimic, adapt or create decisions as intelligent beings.

This general consideration and large problematic spanned the — broadly-speaking — Sciences of Decision, to capture the common features of decision problems. Inside this domain, one can distinguish many fields, relative to specific problems, characteristics and constraints. Taking a relevant decision is intrinsically a different problem for a medical decision support system, a path planner, a mechanical structure optimizer, a network flow controller, a Backgammon player, etc. Formalizing the different decision problems and developing tools and ideas to solve them leads to these different fields. These fields focus either on a specific Mathematical framework, a given problematic or a family of methods and tools. From the Artificial Intelligence point of view, formalizing and solving Decision problems consists in constructing relevant Mathematical formulations (models), analysis tools and computational methods for such problems.

Deciding is often linked with the notion of separating bad solutions from good ones, and eventually choosing the "best" solution. Therefore, Decision and Optimization seem to be two close topics. But the notion of optimal decision is far from being unique. Everybody reasonably agrees that being healthy and rich is better than sick and poor, but choices are rarely that clear. How do we quantify the trade-off between the number of car accident victims and the amount of money one puts into road security policies? Is this quantification a universal value for a human life? Formalizing Decision problems is a matter of expressing the *criteria*, the *compromises*, the *context* and *constraints* in a given language, which we use to automate reasoning in order to find the good decision for these specific criteria, compromises, contexts and constraints.

While this very short introduction does not pretend to cover all of the Mathematical extensions of Decision Sciences[1], we will try, in the following paragraphs, to give a compre-

[1] Is it possible to do research in a field and pretend being able to cover its full span?

hensive view of some sub-categories related to the Mathematical analysis of Decision.

1.1.1 Formalizing Decision

Taking a piece of paper and writing down a Decision problem implies extracting *variables*, *relations* between variables and knowledge about the temporal *evolution* of these relations; it also implies defining *deciding agents* which all have different *criteria* supporting their decision. Different mathematical tools help formalizing decision problems depending on the features they present for the above *variables*, *relations*, *dynamics*, *agents* and *criteria*; they define different branches of Decision Sciences. The next paragraphs suggest a quick walk through these features, illustrating different facets of Decision Theory. Introducing these general considerations will also help in order to specifically target the class of problems addressed in this thesis.

Variables?

The very nature of the decision variables conditions the way the problem is written. Deciding the dimensions for the aircraft beam is by nature a continuous problem, thus implying real-valued variables. *Analysis* is the branch of Mathematics studying continuous spaces, the associated topology and properties.

On the other hand, finding the right number of satellites to correctly broadcast a television service is a discrete problem by nature: one cannot send half a satellite. Optimizing over ordered discrete quantities is often referred to as the field of *Combinatorial Optimization*.

Lastly, finding the right sequence of movements to solve the "towers of Hanoi" problem is a matter of searching through many combinations of unordered logical predicates which is the domain of *Logic*.

Still, many problems need a mixture of these types of variables to describe all the objects they consider. However, most of the techniques we know for solving mathematical problems are restricted to a specific class of variables. For instance, Non-Linear or Linear Programming deal with continuous quantities, Integer Linear Programming considers ordered variables, Predicate (or First-Order) Logic deals with Boolean values, etc. Algorithms designed to solve Decision problems usually build on these mathematical foundations and still today, have a hard time mixing decision variables of different natures together in the general case. Constraint Programming is maybe one of the most successful approaches on this topic. These modeling requirements orient the resolution towards specific fields and tools to find the right decision. So, the nature of decision variables defines low-level theory branches and domains dedicated to solving specific types of models.

Relations?

Once we have determined the nature of our variables, we need to describe how they interact. Classes of relations between variables spanned different research fields, each focusing on a specific feature. For example, Convex and Non-Linear Optimization deal with minimizing non-linear cost functions under continuous constraints. Thus, choosing a modeling framework (differential equations, constraint networks, etc.) is a key to the family of problems which can be represented and to the class of algorithms which can be applied to the problem

at hand. Each of these frameworks is a subcategory of the Mathematical tools for Decision.

An important question concerning the relations between decision variables is related to the uncertainty in the knowledge we have. For example, the decision of buying or selling shares in a stock exchange market is based on imprecise knowledge about the intentions of other commercial agents, the evolution of prices, etc. One usually distinguishes between deterministic problems and problems of decision under uncertainty. The latter can be modeled in different ways: probabilistic knowledge, possibilistic or fuzzy description, contingencies, etc., thus defining the corresponding number of research fields.

Dynamics?

Then comes one of the most important distinctions inside Decision Sciences. The question of knowing how relations between variables change, underlie the difference between single and sequential Decisions. If the problem is to classify a group of pixels inside an image as belonging to the same physical object, the decision is static in the sense that the choice has no immediate consequence: the decision algorithm's output is a unique decision. On the other hand, if the problem is to find the right sequence of actions in order to make tea using the basic ingredients, the solution involves a sequence of actions (boil the water — put the tea in a cup — put sugar in the cup — add the water) found by analyzing the interaction between decisions (actions) with the decision context (the environment). Fields as *Optimization* or *Statistical Learning* often deal with single decisions while domains as *Planning* or *Reinforcement Learning* explore the problem of sequential decisions.

Then one needs to distinguish between many possibilities concerning the problem's dynamics:

- Is there a known model of the decision context? When such a model is explicitly given, sequential decision-making is the field of *Planning*.
- Are the variables observable? Partially observable? For instance, doctors often have to make a diagnosis concerning organs which they do not observe directly. Modeling partial observability yields different branches of Decision; for example *Bayesian Inference* deals with the single decision case in Bayesian probabilistic settings.
- If a sequence of decisions is involved, are these decisions taken online (during the execution, with the feedback of experience) of offline (before execution starts)?
- Do we consider a continuous time? The case of single criteria, continuous time, deterministic decision problem is the field of continuous *Optimal Control*. On the other hand, modeling the system's dynamics with a discrete representation of time, with a system evolving by "steps" has been studied as *Discrete Events Dynamic Systems*.

Agents?

Considering decisions for an autonomous fire surveillance aircraft and for a team of these aircrafts are two problems which involve very different decisions. A first distinction needs to be made between the question of decision in a single-agent setup or in the larger framework of multi-agent systems. Then, even with several agents, one often distinguishes various fields such as:

- Game Theory, defining criteria for equilibrium of decisions, considering separately the questions of cooperative or adversarial games,

- Multi-agent cooperation or coordination systems,
- Meta-heuristic evolutionary methods considering large quantities of simple agents and studying the emergence of a global intelligent behavior (ant colonies, swarm methods, etc.).

Criteria?

Finally, depending on the nature of the deciding agent itself, formalizing preferences, values, or desires implies defining a criterion. Criteria can be related to the problem of *Satisfaction*, where one tries to find any decision which verifies the criterion, or to the problem of *Optimization* which ranks all solutions and searches for an optimal decision. This last category naturally leaves place for compromise by defining the possibility of finding a sub-optimal solution which is close to the optimal one and good enough for the agent. On top of these distinctions, problems do not necessarily have a single criterion, for example multi-criteria Optimization or Satisfaction try to find solutions relevant with respect to a given vector of criteria.

1.1.2 From Decision Theory to Discrete-Time Stochastic Optimal Control

Let us try to summarize the different features mentioned in the previous paragraphs in order to outline the topic of interest of these pages. We will deal with:

- Problems of sequential decision making,
- involving a single agent, which interacts with
- a Discrete Events Dynamic Systems decision context,
- described in a probabilistic framework,
- with a model of the problem given either as an explicit predictive model or as a simulator,
- and involving a single optimization criterion based on the interaction with the environment.

This problem is linked to the field of *Discrete-time, Stochastic Optimal Control*, more specifically to the domains of *Planning under Uncertainty* and *Reinforcement Learning* which we present more in detail in the next section.

1.2 Planning and Learning to act

Sequential decision models are mathematical abstractions of situations in which decisions must be made in several stages, while incurring a certain cost or receiving a certain reward at each stage. These costs or rewards correspond to the evaluation of each step's outcome: it is either a reinforcement signal provided by the environment or an immediate gain or loss evaluation. Each decision may influence the circumstances under which future decisions will be made so that the agent must balance the cost of the current decision against the expected cost of future situations.

This problem of sequential decision making under probabilistic uncertainty has been addressed from different points of view, with a common modeling basis. Since we try to use as little bibliographical citations as possible in this chapter, we simply refer the reader to the excellent textbooks of:

- [Bertsekas and Shreve, 1996] for the *Discrete-Time Stochastic Optimal Control* approach,
- [Puterman, 1994] for the *Probabilistic Planning under Uncertainty* point of view,
- and [Sutton and Barto, 1998] for an introduction to *Reinforcement Learning*.

The three disciplines mentioned above address similar problematics from different points of view. *Discrete-Time Stochastic Optimal Control* considers a "system control" approach: the decision context is viewed as a system for which we search for a (generally closed-loop) controller in order to bring the system to a certain state via a desired behavior specified by the criterion. This approach is closely related to the vocabulary of *Control Theory*. The problem solved deals with the question of determining the best controller — with respect to a given criterion — used to interact with a stochastic environment. *Probabilistic Planning under Uncertainty* is centered on the search for a sequence of decision rules and adopts the point of view of exploiting domain knowledge to build such a sequence, in an agent-centered formalism. Finally, *Reinforcement Learning* addresses the question of dynamically finding this sequence through the interaction between an agent and its environment. In all three approaches, a common modeling basis is used: Markov Decision Processes (MDPs). The underlying assumption of these fields is that the system to control / the agent's decision variables / the agent's environment can be described as an MDP[2]. This chapter tries to remain at the level of ideas so we will wait for the next chapter to introduce the formal MDP model in detail.

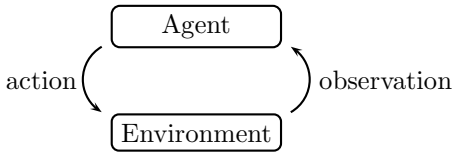


Figure 1.1: Sequential Decision framework

Sequential decisions in *Optimal Control* can be illustrated by the situation where a deciding agent acts upon its environment through the successive actions it performs (as shown on figure 1.1). In the discrete events framework, the environment evolves by discrete steps, generating a sequence of observations for the agent. These observations carry information about the state variables' evolution or the rewards and costs of the current strategy. *Planning* focuses on determining the best way to act given a model of the environment and an optimality criterion, while *Reinforcement Learning* takes the approach of dynamically improving the agent's behavior by reasoning about the experience of interacting with the environment.

[2]This model might not be known in advance to the decision-maker, especially in the case of *Reinforcement Learning*.

1.2.1 The problem of Planning

The general question of *Planning* consists in reasoning about actions' expected outcomes so as to organize them in order to fulfill some predefined objective. It is a deliberative process, requiring prior knowledge about the deciding agent's environment, which aims at finding good or optimal plans of action. [Ghallab et al., 2004] defines *Automated Planning* as the area of *Artificial Intelligence* that studies this deliberation process computationally.

Because there are various types of actions, contexts and goals, there also exist the corresponding forms of planning. These forms of planning can be seen from the applicative point of view. To cite a few examples:

- Path and motion planning imply geometric operators for organizing movements of agents.
- Economy planning: portfolio management, investment schedules.
- Urban planning: public transportation, waste management.
- Logistics: supply chains management, workflow control.
- Planning for space operations: maintenance of satellite constellations, action strategy of a Mars rover.
- Robotics planning: often combines results from motion planning with high-level mission operators. Applications include nuclear site intervention rovers, autonomous Unmanned Air Vehicle mission planning.
- Project planning: organizing the succession of projects steps given the constraints.
- Etc.

Modeling this large variety of planning problems implies defining models which present appropriate features. These features extend the ones presented in the previous section:

- Continuous vs. discrete (or boolean or hybrid) variables
- Hierarchical description of operators vs. atomic representations
- Deterministic vs. uncertain (probabilistic, possibilistic, contingent) models

On top of these domain features, tools have been developed for representing and solving planning problems, thus defining the corresponding planning disciplines. To cite a few:

- Planning-graph techniques
- Hierarchical Task Network planning
- Plan-space planning
- Constraint-based approaches
- Temporal planning
- Planning in Markov Decision Processes

- Etc.

The output of a planning algorithm is a *plan of actions*. This plan can be formulated in various forms. Classical planning algorithms generally calculate a fully ordered sequence of actions to perform. However, some representation structures can be richer than this straightforward sequence of actions. For instance, Partial Order Planners output a partially ordered set of actions or sequences of partially ordered sets of actions. Contingent Planning define conditional plans which specify alternative strategies depending on the execution outcomes. Finally, universal plans or *policies* are functions mapping execution history to actions, which are often the result of non-deterministic planning algorithms.

"Plans" are the *Planning* term for "controllers" in *Control Theory*[3]. A sequential plan is an open-loop controller, while a universal plan is a closed-loop controller, applied to the discrete events system describing the world. Open-loop control works fine when all of the following are true:

1. The model used to design the controller is a completely accurate model of the physical system.
2. The physical system's initial state can be exactly determined.
3. The physical system is deterministic.
4. There are no unmodelled disturbances.

In other words, open-loop control works fine when execution corresponds exactly to what the model predicted. These conditions hold for some of the problems studied in *Artificial Intelligence*, but they are not true for most realistic control problems. Classical planning turns towards architecture solutions, such as plan repair or re-planning, for coping with real world contingencies[4]. In the case of probabilistic systems, one often prefers to build closed-loop control policies.

Hence, the problem of *Probabilistic Planning under Uncertainty* is defined by the inference of efficient control policies, given a stochastic model of the system to control and an optimality criterion.

1.2.2 Deciding from experience: Reinforcement Learning

Reinforcement Learning searches for the exact same closed-loop policy, but adopts a learning point of view. [Sutton and Barto, 1998] introduces *Reinforcement Learning* as "learning what to do — how to map situations to actions — so as to maximize a numerical reward signal". The approach is to learn what actions provide the best reward by trying them and reinforcing the control policy. Rewards might not be immediately available after a single action, but rather accessible through a sequence of optimal actions. Hence, the two most important characteristics of *Reinforcement Learning* problems are the features of "trial-and-error" and "delayed rewards". One can notice that the core problem remains the same as the one of *Optimal Control*, but the point of view is completely different.

[3] See [Barto et al., 1995] for an comparison between Dynamic Programming, Heuristic Search and Optimal Control.

[4] Control Theory also defines the area of *robust control* which studies robustness of controllers when the model parameters vary.

It is interesting to relate *Reinforcement Learning* to the two main trends of *Machine Learning*: *Supervised* and *Unsupervised Learning*. *Supervised Learning* is learning from examples input by an external supervisor, either a teacher or a set of predefined data. *Reinforcement Learning* does not fit in *Supervised Learning* since it focuses on learning through the interaction with the system and thus does not receive samples from a teacher but from its own behavior and experience. This important distinction raises the core question of *Reinforcement Learning* methods known as the exploration vs. exploitation trade-off. To obtain a maximum reward, the agent should apply its best policy found so far and thus *exploit* its acquired knowledge. However, to discover new and better actions and situations, it has to try actions it has not tried before, thus taking the risk of earning less than what its current policy yields, by *exploring*. This exploration vs. exploitation balancing dilemma is even more problematic in the case of stochastic systems where actions need to be tried several times in the same situation to obtain a reasonable estimation of their value.

The type of problems we will address in this thesis deals with the question of building closed-loop policies for time-dependent, stochastic systems, involving uncertainty. We will consider different cases where some information is available to model the system under different forms: complete predictive model or generative model[5]. The general mathematical framework in which we will work is based on Markov Decision Processes (MDP). We will introduce the formal MDP model at the beginning of the next chapter.

1.3 Time, uncertainty and sequential decision problems

Temporal Planning is a specific branch of Classical Planning which involves dealing with durative actions and events, and focuses on the temporal extensions of planning. In *Planning under uncertainty* and more specifically in the commonly-used MDP framework, most variables are considered discrete and the sequential decision problem is supposed to take place in an stationary environment for which decisions are taken at fixed, predefined instants in time called decision epochs.

Many real world problems that involve decision under uncertainty do not satisfy the hypotheses of stationarity, fixed transition duration and fixed-time decision epochs. In order to introduce the family of time-dependent problems which we call *Temporal Markov Decision Problems*, we present here several different examples exhibiting the specific features and structure of such problems.

1.3.1 Examples

Biagent coordination

In order to cross a burning forest and to reach a specific location, a ground rover needs to plan its path and its mission. The actions the rover can perform are movements on the forest roads which can be blocked because of the fire propagation. The rover's world model is described as a navigation graph where edges are roads and vertices are crossings. Each movement's outcome is uncertain in terms of resulting state and duration because the fire and the burned terrain can lead the movement to fail.

The rover is not alone: it is teaming with a helicopter UAV (Unmanned Aerial Vehicle). The UAV can watch the forest from above but cannot go too close to zones that are still

[5] simulator

Figure 1.2: Fire fighting coordination

burning. The only goal of the UAV is to assist the rover's navigation. For this purpose, these two heterogeneous agents can communicate a limited amount of information. Since agents are heterogeneous, might receive different mission goals and need to be fully autonomous, mission planning is an individual process and necessitates some decentralized coordination as illustrated on figure 1.3.

We suppose that the communication channel between agents is used to declare intentions or exchange map information with a predefined common vocabulary. Each agent's planning procedure needs to take into account the consequences of the other agent's declared intentions. In other words, on top of the continuous time dynamics of the fire's evolution, each agent's plan affects the other's model of the world at real-valued dates. Therefore, the planning algorithm needs to cope with an observable continuous time which is a crucial variable for planning.

More specifically, when the rover declares a first navigation plan, its message modifies the UAV's reward model by adding time-dependent rewards along the rover's path because there is a higher gain in checking this portion of the road rather than another one.

Even though the planning process needs to take into account a continuous time variable, it remains a discrete event planning process: each action remains a discrete event which conditions the next step of the global process' evolution.

The subway problem

Imagine now a virtual subway network where passengers arrive at the stations to take the train following a well established distribution probability. Their destination is also known via a distribution over train stations. These distributions vary with the time of day: modeling rush hours, people going to work, leaving for lunch, etc. The problem of the subway manager is to optimize the running cost of the network over a full day by deciding when to add or remove trains from the network. Fewer trains means less exploitation cost but also imply

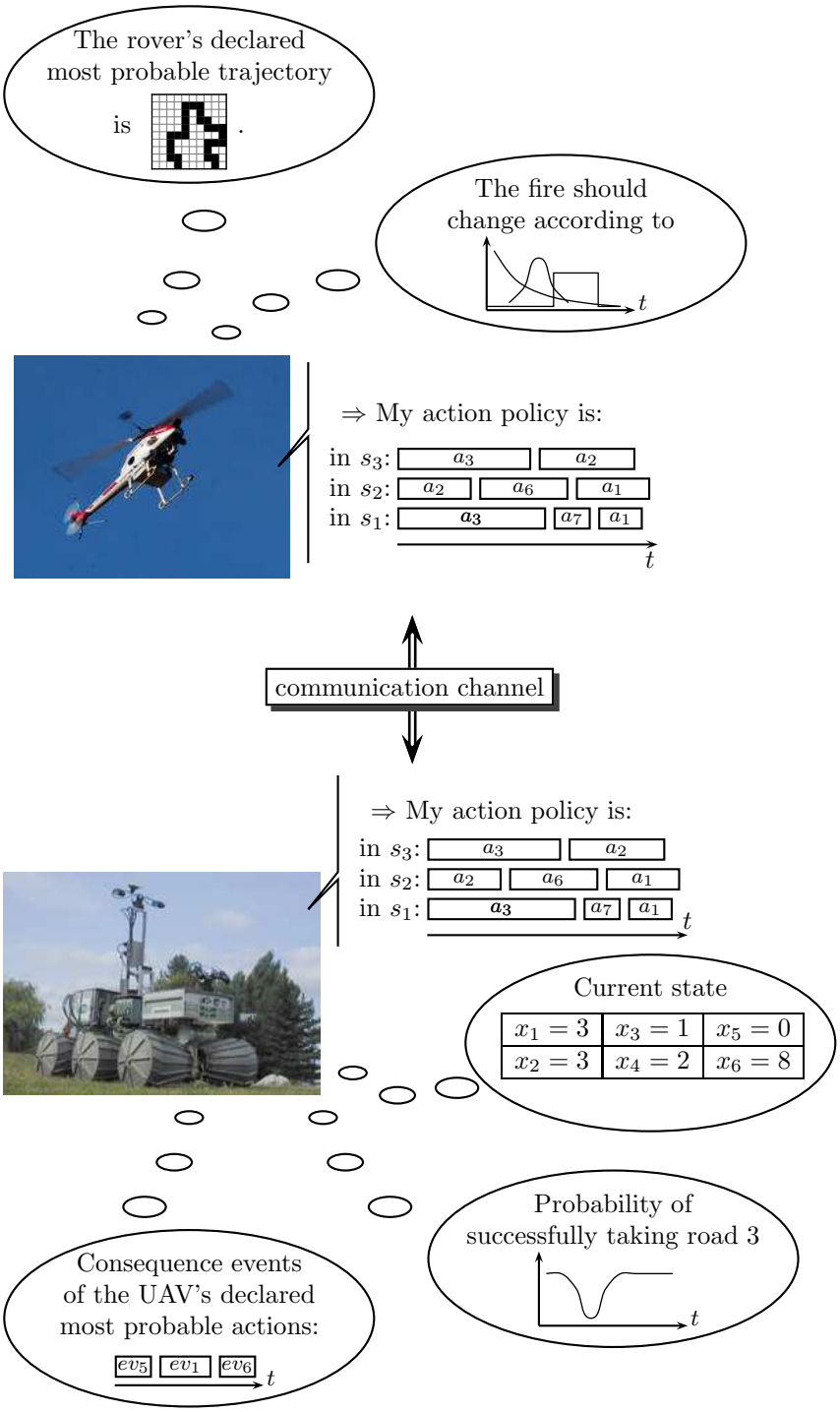
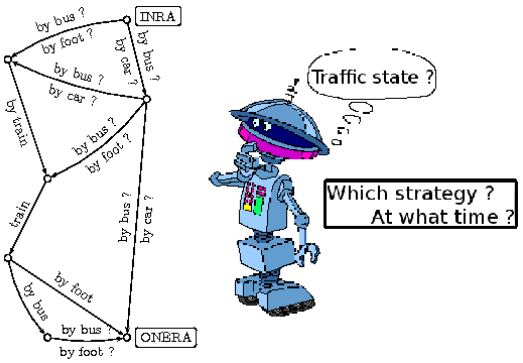


Figure 1.3: Illustrating the origins of time dependency in the coordination problem

less profit during the periods where they can be fully used.

(a) The subway network in Toulouse

(b) Airport taxiing



(c) INRA to ONERA

(d) Mars rover

Figure 1.4: Examples

Each of the manager's actions have an immediate deterministic effect but the long term consequences are stochastic, hard to predict and model because of the influence of all the concurrent stochastic events occurring in parallel (such as passengers arrivals at different stations or trains movements).

Therefore, the difficulty of this problem is twofold: on top of the large number of state variables (dimensions) which yields a large and hybrid state space, concurrency makes it is complex to write explicit transition probabilities for the overall stochastic process of the state variables. Thus, the problem itself is hard to model as a single synthetic stochastic process[6].

Airport taxiing

Today's airports are getting more and more crowded and planning the ground movement of planes is a problem which combines the critical aspects of traffic optimization and large stochastic influences from the weather, the landing conditions, airport alerts or technical failures.

[6]The reader familiar with queueing systems will find an analogy with the problem of writing the output times process' probability density function for an $M/G/n$ queue.

When a plane lands, it can leave the runway at different points. Then it needs to go through the road network which leads to the gates and terminals. Later, it needs to go through maintenance and finally picks its new crew and passengers up and leaves the airport.

Given the plane arrival and departure schedule for the day and some knowledge about the uncertainty of the problem (weather model, delay risk, etc.), we wish to compute an efficient routing strategy for the planes from the runway to the terminals and back in order to optimise the departing times.

INRA-ONERA path finding

This problem is a simplified version of the fire fighting problem. The goal here is to go as fast as possible from ONERA to INRA for a meeting, choosing at each step between taking the bus, the car or walking. Traffic, bus schedules, incident probabilities, etc. depend continuously on the time of day and so does the optimal strategy.

Mars rover

This problem is a modified version of the standard Mars rover problem in temporal planning. The agent is a ground rover which has a mission of collecting rocks and pictures from the surface of Mars. It needs to plan its mission according to the time of day (the ability to recharge), its battery and free memory level, its position and the already accomplished goals.

Depending on the time of day, the lighting changes; this affects the probability of taking a successful photo and modifies the solar panels recharge capacity.

1.3.2 Characterizing temporal problems

The examples presented in the previous section come from different areas of application but have some features in common. Their dynamics and the complexity of representing their evolution are strongly dependent on the time variable. Moreover, having an observable time variable among the state variables forbids loops (other than instantaneous): one cannot go back in time so the only possible loop is an instantaneous action which takes the process back to the same state at the same instant. Consequently, these problems present a specific loop-less structure conditioned by time.

This structure could be induced by any other non-replenishable resource. We will concentrate on time in order to build our algorithms and to work on the same family of problems — keeping in mind that this work could extend to more general setups.

Definition (Temporal Markov Decision Problems)**.** *We define* Temporal Markov Decision Problems *as decision under uncertainty problems presenting the following five main features:*

- ***Discrete event decision problems:*** *the system's evolution is described as a discrete event dynamic system. These systems have been well studied and generalized in the DEVS framework [Zeigler et al., 2000]. We will show in chapter 11 that all our models fall into this framework and we will develop the direct mapping between our most complex model and DEVS models.*
- ***Uncertainty on the actions' results:*** *as for stochastic decision processes, actions' results are uncertain and described via probability distributions over the post-action states.*

- ***Uncertainty on transition's durations:*** *contrarily to the standard MDP framework where all transitions have unit duration, we allow transitions to have stochastic, real-valued durations. The next chapter will illustrate previous models from the literature that share this feature and will highlight the differences and compatibilities with temporal Markov decision problems.*

- ***Explicit time dependency:*** *on top of having random continuous durations, time is made explicit and observable in temporal Markov decision problems. In other words, time is a state variable affected by both the uncertainty concerning the next state and the stochastic transition durations. The structure induced by including an explicit time variable in the state space will be developed in chapters 4 to 9, along with numerical methods designed to exploit this aspect. Moreover, time plays a specific role since it appears as the exponent of γ in the discounted optimization criterion. This last point will motivate the theoretical developments of chapter 8 which prove that the optimality equation can be preserved with an observable time.*

- ***Complexity from concurrency:*** *even though this aspect is hardly quantifiable, it illustrates the fact that, for some of the problems presented above (*eg. *the subway, airport or coordination problems), determining a set of state variables for which the process verifies Markov's property can be difficult. Moreover, writing the transition function over this joint state space requires a lot of engineering while the separate concurrent processes that yield the complexity of the overall one are themselves quite simple. This remark motivates the investigation of Generalized Semi-Markov Decision Processes with continuous observable time in chapters 11 and 13.*

In order to lift the ambiguity with Boyan and Littman's TMDP model, we call these problems "temporal" and not "time-dependent". The formal TMDP model is presented in the next chapter and studied in part I as a specific way of modeling temporal Markov decision problems. This distinction in vocabulary is also done to account for the genericity and additional complexity coming from all the previous features.

The next chapter focuses on presenting different approaches to dealing with time and modeling time-dependency for decision problems. These approaches come from the stochastic processes or from the MDP literature. We will try to go from simple standard Markov (Decision) Process and will progressively introduce explicit time dependency and temporal complexity.

2

Temporal Markov Decision Problems — Modeling

Planning is a branch of decision theory dealing with the selection and ordering of high-level actions which lead to a certain goal or behavior. This chapter introduces the basic notions and formalisms which will be used throughout the thesis. We start with the general framework of Markov Decision Processes (MDPs) in order to model uncertainty in action's results. Then we will highlight where the difficulties arise when one wishes to deal with observable time in MDPs. This discussion will lead to the progressive introduction of specific models from the literature in order to model continuous time dependency in planning under uncertainty.

2.1 Markov Decision Processes

The work presented in this thesis deals with the framework of Markov Decision Processes (MDPs) [Bellman, 1954; Puterman, 1994; Bertsekas, 1995]. MDPs have become a popular framework for representing decision problems where actions' resulting states are uncertain. In an MDP, each action's outcome is described through a probability distribution on the next state of the process, conditioned on the current state. This provides a straightforward way of presenting the uncertain effects of every single action on the problem. We use this section to provide a brief review on MDP basics and standard algorithms to solve them.

2.1.1 Formalism

Definition (Markov Decision Process)**.** *An MDP is a discrete time stochastic decision process described by a 4-tuple $\langle S, A, P, r \rangle$, where:*

- S *is the* state space *of the problem. States hold all the relevant information to describe the configuration, position, internal variables, etc. available to the decision-maker. S is usually a discrete countable — often finite — set of states. Extensions exist to continuous compact spaces or Borel subsets of complete, separable metric spaces for the state space.*
- A *is the set of* actions *available to the problem. Each action a has a specific stochastic influence on the problem and triggers a transition to a new state. As for S, A is usually countable and often finite, but the same extensions exist to continuous cases.*
- P *is the* transition function *of the problem. It describes the probability that action a takes the process from state s to state s'. In other words: $P(s, a, s') = Pr(s'|s, a)$. An*

important property of MDPs is that this transition function verifies Markov's property, i.e. *the probability of reaching state s' when one undertakes action a in state s only depends on a and s and not on the whole history of previous transitions. This property highlights the fact that one state holds all the information which is necessary to the agent in order to predict the current transition's outcome.*

r *is the* reward function *of the process. Whenever an action is performed and a transition is triggered, the agent receives a reward $R(s,a,s')$. Sometimes this reward function will be given as the mathematical expectation of the one-step reward: $r(s,a) = \sum_{s' \in S} P(s,a,s')R(s,a,s')$. The goal of the decision maker will be to optimize a given criterion based on the transition and reward model of the process.*

One often writes $P(s'|s,a) = P(s,a,s')$. The equations from the previous definition are recalled below.

$$P \ : \ \left\{ \begin{array}{ccc} S \times A \times S & \rightarrow & [0;1] \\ (s,a,s') & \mapsto & Pr(s'|s,a) \end{array} \right. \tag{2.1}$$

$$\forall (s,a) \in S \times A, \ \sum_{s' \in S} P(s,a,s') = 1 \tag{2.2}$$

$$R \ : \ \left\{ \begin{array}{ccc} S \times A \times S & \rightarrow & \mathbb{R} \\ (s,a,s') & \mapsto & R(s,a,s') \end{array} \right. \tag{2.3}$$

$$r \ : \ \left\{ \begin{array}{ccc} S \times A & \rightarrow & \mathbb{R} \\ (s,a) & \mapsto & \sum_{s' \in S} P(s,a,s')R(s,a,s') \end{array} \right. \tag{2.4}$$

The successive discrete times at which the agent is asked for a decision are called *decision epochs* and correspond to the *process' steps*. Figure 2.1 and 2.2 illustrate the dynamics of MDP models (for notation conventions, please refer to page xi).

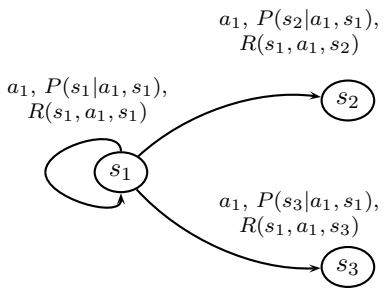


Figure 2.1: MDP transition

2.1.2 Policies, criteria and value functions

Solving an MDP problem usually consists in finding a *control policy* which optimizes a given *criterion*. A stationary Markov control policy π is defined as a mapping from states to actions specifying which action is the best to undertake in every state of the process.

$$\pi \ : \ \left\{ \begin{array}{ccc} S & \rightarrow & A \\ s & \mapsto & a \end{array} \right. \tag{2.5}$$

In the general case, policies are defined as sequences $\{\rho_0, \rho_1, \ldots\}$ of *decision rules*. Each decision rule is a mapping from previous history to actions. There is one decision rule per

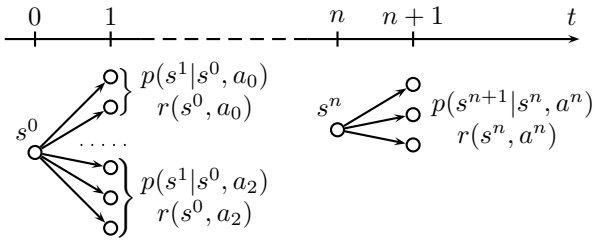


Figure 2.2: Transition and reward functions

decision epoch (thus yielding an infinite countable set of decision rules for infinite horizon problems). [Puterman, 1994] shows that for infinite horizon problems and for most common criteria, there always exists a policy which is:

- stationary: all decision rules are the same throughout the decision epochs,
- Markovian: the optimal action only depends on the current state,
- optimal: at least as good as any history-dependent policy.

This allows us to search for optimal policies in the restricted set of stationary Markov policies. To be as generic as possible, we should also mention the case of stochastic policies. A stochastic decision rule is a mapping from states to probability distributions over actions. A stochastic policy is a sequence of stochastic decision rules. [Puterman, 1994] shows that the previous results hold for deterministic and stochastic policies, namely, there exists a stationary, deterministic, Markovian, optimal policy. We won't consider the case of stochastic policies and will write $\mathcal{D}$ the set of deterministic stationary Markov policies.

Once we provide a criterion which evaluates the performance of a policy π from any initial state s, we can define the *value function* associated with the policy:

$$V^\pi \ : \ \left\{ \begin{array}{ccc} S & \rightarrow & \mathbb{R} \\ s & \mapsto & V^\pi(s) \end{array} \right. \tag{2.6}$$

This value function evaluates how well the policy performs with respect to the given criterion. We will write $\mathcal{V}$ the set of value functions. A policy π is said to be optimal if:

$$\forall s \in S, \ \forall \pi' \in \mathcal{D}, \ V^\pi(s) \geq V^{\pi'}(s) \tag{2.7}$$

Defining the criterion to optimize corresponds to defining how one evaluates the policy's quality. For instance, if we know the number T of decision epochs in advance, we might want to use the *finite horizon criterion* which is the cumulative sum of the rewards obtained when applying the policy from the current state until the horizon:

$$V(s) = E\left[\sum_{\delta=0}^{T} r_\delta | s_0 = s\right] \tag{2.8}$$

However, the optimal policy is no longer stationary for the finite horizon criterion. In many cases, the number of decision epochs is very large or unknown and while we are far from the horizon, the policy tends to be stationary. This is why we will only consider infinite horizon criteria. [Bertsekas and Tsitsiklis, 1996] introduces an elegant classification of infinite horizon problems as either *stochastic shortest path*, *discounted*, or *average cost per*

stage problems, which correspond respectively to the three following criteria.

If we know that our execution will necessarily end before an infinite number of steps (for example because there is a probability one of ending up trapped in a cost-free terminal state from any initial state), then we might want to use a *total reward criterion* which allows for unbounded horizon reasoning:

$$V(s) = E\left[\sum_{\delta=0}^{\infty} r_\delta | s_0 = s\right] \tag{2.9}$$

The total reward criterion can be seen as defining a stochastic shortest path problem in terms of rewards. In the general case though, the total reward criterion is not guaranteed to be finite. This is why the most commonly used criterion is the *discounted criterion*:

$$V(s) = E\left[\sum_{\delta=0}^{\infty} \gamma^\delta r_\delta | s_0 = s\right],\ \gamma \in [0,1) \tag{2.10}$$

The discounted criterion penalizes rewards obtained late in the future and insures convergence of the sum if the MDP's reward model is bounded. Depending on the context, the γ factor can be interpreted as a "no critical failure before the next step" probability (mission planning), an inflation loss [1] (economy) or a penalty discount.

One last criterion is sometimes used as an alternative to the discounted criterion: it is sometimes more important to have a good average behaviour over the execution rather than to optimize the (discounted) sum of rewards. For this purpose, we can define the *average criterion*:

$$V(s) = E\left[\lim_{T\to\infty} \frac{1}{T}\sum_{\delta=0}^{T} r_\delta | s_0 = s\right] \tag{2.11}$$

All these criteria (and other ones) allow the definition of the value function associated to each policy (equation 2.6). In this work, we will focus on infinite horizon MDPs with discounted or total reward criteria.

2.1.3 Policy evaluation and optimality equation

We will note V^* the *optimal* value function. More specifically, V^* is defined as:

$$\forall s \in S,\ V^*(s) = \max_{\pi\in\mathcal{D}} V^\pi(s) \tag{2.12}$$

$V^*(s)$ corresponds to the best gain one can expect with regard to a given criterion when the policy's execution starts in state s. Similarly, any policy with value function $V^\pi = V^*$ will be called *optimal* according to equation 2.7 and noted π^*. The following results hold for the discounted criterion and can apply to the total reward criterion in some specific cases (namely, when the limit in the criterion is guaranteed to exist).

One can define the policy evaluation operator L^π as:

$$L^\pi : \left\{ \begin{array}{ccc} \mathcal{V} & \to & \mathcal{V} \\ V & \mapsto & \left\{ \begin{array}{ccc} S & \to & \mathbb{R} \\ s & \mapsto & r(s,\pi(s)) + \gamma \sum_{s'\in S} P(s'|s,\pi(s))V(s') \end{array} \right. \end{array} \right. \tag{2.13}$$

[1] Actually, if we write k the inflation rate, we have $\gamma = \frac{1}{1+k}$

In other words:

$$\forall V \in \mathcal{V},\ \forall s \in S,\ L^\pi V(s) = r(s,\pi(s)) + \gamma \sum_{s' \in S} P(s'|s,\pi(s))V(s') \tag{2.14}$$

L^π is a contraction mapping with respect to the supremum norm in the set of functions from S to $\mathbb{R}$ and V^π is the unique solution to the equation $V = L^\pi V$. This provides an implicit characterization of a policy's value function. Similarly, we can define Bellman's dynamic programming operator L for MDPs as:

$$L : \left\{ \begin{array}{ccc} \mathcal{V} & \rightarrow & \mathcal{V} \\ V & \mapsto & \left\{ \begin{array}{ccc} S & \rightarrow & \mathbb{R} \\ s & \mapsto & \max\limits_{a \in A} \left\{ r(s,a) + \gamma \sum\limits_{s' \in S} P(s'|s,a)V(s') \right\} \end{array} \right. \end{array} \right. \tag{2.15}$$

$$\forall V \in \mathcal{V},\ \forall s \in S,\ LV(s) = \max_{a \in A} \left\{ r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)V(s') \right\} \tag{2.16}$$

This operator is also a contraction mapping on the Banach space of value functions (with the supremum norm) and one can prove (*eg.* in [Bellman, 1957; Puterman, 1994]) that V^* is the only solution to the equation $V = LV$:

$$V^*(s) = \max_{a \in A} \left\{ r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)V^*(s') \right\} \tag{2.17}$$

Since V^* is the optimal policy's value function, we can build π^* as the greedy policy with respect to V^* and hence derive an optimal policy from equation 2.17:

$$\pi^*(s) = \arg\max_{a \in A} \left\{ r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)V^*(s') \right\} \tag{2.18}$$

2.1.4 Q-values

An equivalent formulation of the above properties is in terms of Q-functions or Q-values. One can define the Q-value $Q^\pi(s,a)$ of performing action a in state s, given policy π, as the expected value of applying a at the first decision step and then using policy π until the horizon. For the discounted criterion, one has:

$$Q^\pi(s,a) = E\left[\sum_{\delta=0}^{\infty} \gamma^\delta r_\delta | s_0 = s, a_0 = a \right],\ \gamma \in [0,1) \tag{2.19}$$

The relationship between V^π and Q^π is given in equation 2.20.

$$\begin{aligned} V^\pi(s) &= Q^\pi(s,\pi(s)) \\ Q^\pi(s,a) &= r(s,a) + \gamma \sum P(s'|s,a)V^\pi(s') \end{aligned} \tag{2.20}$$

The optimal Q-value $Q^*(s,a)$ of action a in state s is the maximum expected cumulative reward which can be obtained over all execution paths starting with applying a in s.

The policy evaluation operator L^π and Bellman operator L also apply on Q-values and yield the policy evaluation equation 2.21 and the optimality equation 2.22, similar to equation 2.17.

$$Q^{\pi}(s,a) = r(s,a) + \gamma \sum P(s'|s,a) Q^{\pi}(s', \pi(s')) \tag{2.21}$$

$$Q^{*}(s,a) = r(s,a) + \gamma \sum P(s'|s,a) \max_{a' \in A} Q^{*}(s', a') \tag{2.22}$$

2.1.5 Optimizing policies

Based on equations 2.17 and 2.18 we can derive standard optimization methods for MDPs. We will briefly present here the Value Iteration, Policy Iteration and Linear Programming algorithms for MDPs. In the next chapters, we will provide more details as we concentrate on more specific features of these methods.

Value Iteration

The Value Iteration algorithm is directly inspired by the fact that L is a contraction mapping over $\mathcal{V}$: any sequence of value functions $(V_n)_{n \in \mathbb{N}}$ recursively defined by equation 2.23 converges to V^*.

$$V_{n+1} = LV_n \tag{2.23}$$

Calculating equation 2.23 (or the equivalent equation for a policy) is called a *Bellman backup*. This allows us to define the Value Iteration algorithm:

Algorithm 2.1: Value Iteration

$V_0 \leftarrow 0$
$n \leftarrow 0$
repeat
  for $s \in S$ **do**
    $V_{n+1}(s) = \max_{a \in A} \left(r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \cdot V_n(s') \right)$
  $n \leftarrow n + 1$
until $\|V_n - V_{n-1}\| \leq \epsilon$
for $s \in S$ **do**
  $\pi(s) \leftarrow \operatorname{argmax}_{a \in A} \left(r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \cdot V_n(s') \right)$
return V_n,π

One can see Value Iteration as the incremental propagation of the rewards to all states in the problem. Each iteration of Value Iteration has complexity in $O(|A||S|^2)$ and the optimal policy is reached in $O\left(\frac{|S||A|}{(1-\gamma)\log(1/(1-\gamma))}\right)$ iterations ([Littman et al., 1995]). The algorithm is usually stopped when $\|V_n - V_{n-1}\| \leq \epsilon$. Then one can write that $\|V_n - V^*\| \leq \frac{2\gamma}{1-\gamma}\epsilon$. In case of equivalent ϵ-optimal actions, the chosen action for π can be any of the set defined by $\operatorname{argmax}_{a \in A} \left(r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \cdot V_n(s') \right)$

Policy Iteration

The idea of Policy Iteration is to perform policy search directly in policy space. The goal here is to incrementally improve an initial policy. For this purpose, we perform one Bellman backup in every state of the problem with regard to the current policy's value function. This way, we are sure to improve the global value function (if possible) and to eventually converge to V^*.

Algorithm 2.2: Policy Iteration

$\pi_0 \in \mathcal{D}$
$n \leftarrow 0$
repeat
  Solve the system of $|S|$ equations:
  $\forall s \in S \quad V_n(s) = r(s, \pi_n(s)) + \gamma \sum_{s' \in S} P(s'|s, \pi_n(s)) V_n(s')$
  for $s \in S$ **do**
    $\pi_{n+1}(s) \leftarrow \underset{a \in A}{\operatorname{argmax}} \left(r(s,a) + \gamma \sum_{s' \in S} PS(s'|s,a) V_n(s') \right)$
  $n \leftarrow n + 1$
until $\pi_n = \pi_{n-1}$
return V_n,π_n

Each iteration of Policy Iteration has complexity in $O\left(|S|^2(|A| + |S|)\right)$ and the algorithm theoretically converges in the same number of iterations as Value Iteration. Practically, the number of iterations required before convergence is usually smaller with Policy Iteration than with Value Iteration. This makes Policy Iteration usually a good choice for MDP resolution even though the policy evaluation phase requires most of the computation time.

The policy iteration algorithm can be generalized by the Actor-Critic architecture ([Sutton and Barto, 1998]) as presented in figure 2.3. The main idea of such an architecture is the presence of two separate procedures interacting inside the algorithm:

- The *critic* performs policy evaluation. It provides an evaluation of the agent's current behaviour.
- The *actor* calculates improvements to the current policy, based on the information provided by the critic.

Policy Iteration is thus an exact, model-based, Actor-Critic algorithm. Variants of Policy Iteration, such as Asynchronous Policy Iteration, Approximate Policy Iteration or Simulation-Based Policy Iteration [Bertsekas, 1995] all fall under the Actor-Critic framework and differ in the way the actor and the critic are implemented (model-based *vs.* model-free, exact *vs.* approximate, etc.).

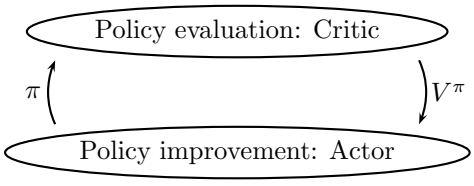


Figure 2.3: Actor-Critic architecture

Linear Programming

The last main family of algorithms for MDP optimization is based on linear programming and was introduced in [d'Epenoux, 1963]. It leaves the dynamic programming framework and solves the optimality equation by writing it as a linear program. The idea is to find the

optimal value function by remarking that if one minimises the quantity $\sum_{s \in S} V(s)$ under the optimality constraint $V \geq LV$, then the solution is necessarily equal to V^*. The associated linear program is:

$$\begin{array}{c} \min \sum_{s \in S} V(s) \\ V \geq LV \end{array} \tag{2.24}$$

Which can be written as:

$$\begin{array}{c} \min \sum_{s \in S} V(s) \\ \forall s \in S,\ \forall a \in A,\ V(s) - \gamma \sum_{s' \in S} P(s'|s,a) V(s') \geq r(s,a) \end{array} \tag{2.25}$$

This approach globally has complexity pseudo-polynomial in $|S||A|$.

Countering the *curse of dimensionality*

All standard methods dedicated to solving MDP problems suffer from Bellman's *curse of dimensionality* [Bellman, 1957]. As the number of variables in the problem increases, the size of the state space increases exponentially, thus making standard algorithms inefficient. Many techniques have been recently developed in order to tackle this problem of state space explosion. This document's purpose is not to list or explore them all, we will however mention some current trends that have been explored in MDP solving as:

- State space partitioning and exploitation of MDP decomposition as in [Hauskrecht et al., 1998; Parr, 1998; Dean and Lin, 1995; Sabbadin, 2002],
- State variable factoring as in [Boutilier et al., 2000; Hoey et al., 2000],
- Approximate Linear Programming in [Hauskrecht and Kveton, 2004; Guestrin et al., 2004; Kveton and Hauskrecht, 2006]
- Focused or heuristic search in [Barto et al., 1995; Bonet and Geffner, 2003b; McMahan et al., 2005; Smith and Simmons, 2006; Hansen and Zilberstein, 2001; Dai and Goldsmith, 2007; Teichteil-Königsbuch and Infantes, 2008],
- Value function approximation in [Lagoudakis and Parr, 2003; Munos and Moore, 2002],
- Etc.

2.2 Time and MDPs

2.2.1 Does time appear in standard MDPs?

The first question that we want to answer is: can we model all the problems of section 1.3.1 as MDPs? And if we can: how can we exploit the structure introduced by the observable continuous time variable?

In standard MDPs with no explicit time variable, one considers that P and r are stationary functions, they do not change from one decision epoch to the other. In this framework, [Puterman, 1994] showed that optimal infinite horizon policies are stationary too, so the standard behaviour of MDPs seems to be fully stationary and time-independent.

However, time appears in the discounted criterion as the exponent of γ. So, our optimization process actually takes a certain notion of future into account. This future is implicitly modeled by the sequence of decision epochs. More specifically, a reward obtained at *time* n is penalized by a factor γ^n but what this n really is, is the index of the decision epoch, not its real date. As in standard stochastic processes, we have considered so far that all transitions had unit duration. What happens if this duration is not set to one anymore?

2.2.2 From MDP to SMDP: introducing uncertain durations

In the stochastic processes framework, Markov Chains are processes that jump from the current state to the next with a probability that only depends on the current state. Since there is no notion of *decision*, *agent* or *plan* in the phenomena described by Markov Chains, they are abstract representations of the discrete stochastic evolution of a given system. Thus, it makes sense to merge the concepts of process step and process time for Markov Chains.

Actually, from a temporal point of view, standard discrete-time Markov Chains are considered to make transitions which have duration one (this way, the process time and the process steps match). Continuous Time Markov Chains include a continuous time variable, but their transition durations are governed by (memoryless) exponential distributions, as presented in [Cox and Miller, 1965].

A Markov chain with arbitrary distributions over transition durations does not retain the Markov property for the determination of the next step's date. However, if transition durations and arrival states are decoupled, most properties of Markov Processes are retained. This forms the model of semi-Markov Processes (SMPs, [Cox and Miller, 1965])[2].

In the rest of this document we will refer indifferently to *transition times* or to *sojourn times*. Indeed, the name of *sojourn time* is a more rigorous denomination of the notion at stake. This difference in vocabulary is justified by the discrete events systems paradigm: the sojourn time in a state s is the time spent in a given state before a transition occurs, but since we are considering discrete events, the evolution is discrete (the system evolves stepwise) and each transition takes the system to its new state s'; therefore, this sojourn time is also the duration of the time period between entering s and entering s', which is the transition time between s and s'.

Introducing random transition times in the MDP framework corresponds to defining semi-Markov decision processes (SMDP, [Puterman, 1994; Howard, 1963]). An SMDP's transition function is given as the probability density function $Q(\tau, s'|s, a)$ describing the probability that the next decision epoch occurs before τ time units in the future, in state s'. The Q transition function is often decoupled as $Q(\tau, s'|s, a) = P(s'|s, a)F(\tau|s, a)$ which introduces a very strong hypothesis on the model since it supposes independence between destination state s' and transition duration τ. We write $f(\tau|s, a)$ the probability density function associated to $F(\tau|s, a)$. Hence:

$$F(d\tau|s, a) = f(\tau|s, a)d\tau$$

Figure 2.4 illustrates the duration model of SMDPs. The reward model of an SMDP is built from the abstraction of a so-called *natural process* which describes the low-level

[2] A more formal definition of SMPs can be given in terms of a process $W = (X, Y)$ where X is a Markov chain and where $P(Y_n = y)$ only depends on the values of X_n and X_{n-1}. Then a process Z choosing its transition's destination state based on X and its transition duration based on Y is called a semi-Markov process. W is a Markov process while Z usually is not.

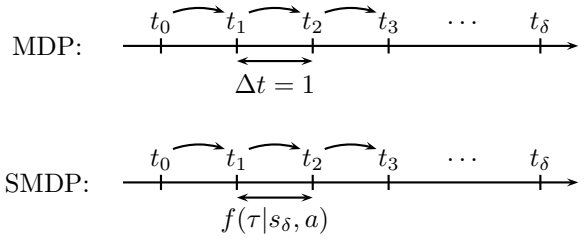


Figure 2.4: Introducing random transition times: SMDPs

continuous evolution of the system state, namely the states traversed by the process while an action completes. The natural process' state and the SMDP's state agree at decision epochs. The SMDP's reward model r is defined as in equation 2.26, where k is a lump sum reward, c is a reward rate, j are intermediate states of the natural process and p is the evolution function of the natural process.

$$r(s,a) = k(s,a) + \int_0^\infty \sum_{j \in S} \left[\int_0^u \gamma^t c(j,s,a) p(j|t,s,a) dt \right] F(du|s,a) \tag{2.26}$$

This description of the reward model allows to consider SMDPs as hierarchical abstractions of macro-actions having stochastic durations. To summarize:

Definition (Semi-Markov Decision Process)**.** *An SMDP is given by the 5-tuple* $\langle S, A, P, F, r \rangle$ *where:*

- S *and* A *are standard MDP state and action spaces,*
- $P(s'|s,a)$ *is the state transition model,*
- $F(\tau|s,a)$ *is the cumulative distribution function of the sojourn time variable* τ*,*
- $r(s,a)$ *is the reward model as described above.*

Given this model, the policy evaluation equation becomes (with $Q_\pi(d\tau, s'|s) = Q(d\tau, s'|s, \pi(s)) = P(s'|s,\pi(s)) f(\tau|s,\pi(s)) d\tau$):

$$V^\pi(s) = r_\pi(s) + \sum_{s' \in S} \int_0^\infty \gamma^\tau \cdot V^\pi(s') \cdot Q_\pi(d\tau, s'|s) \tag{2.27}$$

And if we write $m_\pi(s'|s) = \int_0^\infty \gamma^t \cdot Q_\pi(d\tau, s'|s)$, then we have:

$$V^*(s) = \max_{a \in A} \{ r(s,a) + \sum_{s' \in S} m_a(s'|s) V^*(s') \} \tag{2.28}$$

Therefore, optimizing an SMDP policy turns out to be equivalent to solving a total reward criterion optimality equation. As a matter of fact, introducing variable transition durations into the model allows to take reward rates into account. This can be useful for modeling resource consumption or continuous reward acquisition which are common characteristics of temporal problems.

However, the overall process we consider with SMDPs is still stationary and doesn't allow the representation of time-dependency in temporal Markov problems. For this purpose, the

model itself must be explicitly time-dependent and time must be included as an observable state variable. We will therefore focus on models that allow for extension of MDPs to non-stationary cases.

2.2.3 Some other models taking time partially into account

Other approaches — which do not take simultaneously into account uncertainty on actions outcomes, on transition durations and time-dependency — define different frameworks for introducing time in stochastic problems. One could mention shortest path search algorithms or Stochastic Time Dependent Network (STDN, [Wellman et al., 1995]) which define stochastic transition durations, but deterministic transition outcomes. To our knowledge, the only model that focuses on time as an independent observable state variable is the Time-dependent MDP model (TMDP, [Boyan and Littman, 2001]) which is presented in the next subsection.

2.2.4 Making time observable: the TMDP model

The TMDP model decomposes each transition resulting from the application of an action a into a set of possible *outcomes* $\{\mu\}$. Each outcome describes a resulting state and a transition duration.

Definition (Time-dependent Markov Decision Process)**.** *Formally, one can define a TMDP as:*

- S*, a discrete state space*
- A*, a discrete action space*
- M*, a discrete set of* outcomes $\mu = (s'_\mu, T_\mu, P_\mu)$*:*
  - s'_μ *is the transition's resulting state.*
  - T_μ *is a boolean indicating whether the probability density function* P_μ *concerns absolute dates or relative durations.*
  - $P_\mu(\theta)$ *is a probability density function describing the probability that the transition ends at time* $t = \theta$ *(if* $T_\mu = \text{ABS}$*) or exactly after a duration* $\tau = \theta$ *(if* $T_\mu = \text{REL}$*).*
- $L(\mu|s,t,a)$ *is a transition function giving the probability of triggering outcome* μ*.*
- $R(\mu,t,t')$ *is the reward model associated with the realization of outcome* μ*, starting at* t *and ending at* t'*.*
- $K(s,t)$ *is the reward rate of the "wait" action in state* s *at time* t*.*

The evolution of a TMDP is illustrated on figure 2.5. In state s_1 and at time t, undertaking action a_1 triggers outcome μ_1 with probability $L(\mu_1|s_1,a_1,t) = 0.2$ and outcome μ_2 with probability $L(\mu_2|s_1,a_1,t) = 0.8$. μ_2 describes the transition to s_2 and the transition absolute arrival time is given by P_{μ_2}, whereas μ_1 describes the failure in leaving s_1 (loop to s_1) with duration P_{μ_1}.

Chapters 4 to 6 will describe the TMDP model in more detail and will extend the class of problems it can represent. For now we only present a short analysis which goes slightly further than the original [Boyan and Littman, 2001] paper.

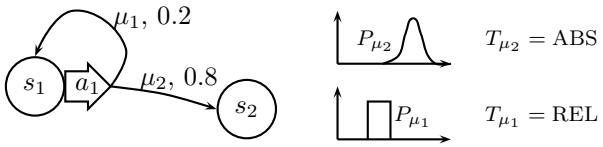


Figure 2.5: TMDP - basic elements

It is interesting to remark on figure 2.5 that TMDPs restrict somehow the user's modeling freedom by forcing the transition duration to depend on the destination state (on the outcome) while sometimes we might wish to work the other way: specifying the destination state given the transition's time-to-trigger.

Similarly, in order to insure the physical meaning and consistency of the model, it is important to add "watchdogs" to the model initially defined in [Boyan and Littman, 2001]. We need to insure that whenever one triggers μ at t, all absolute arrival times for μ are posterior to t. More formally, if we write:

$$\begin{cases} Dep_{\mu,s,a} = \{t \in \mathbb{R} / L(\mu|s,t,a) \neq 0\} \\ M_{ABS} = \{\mu \in M / T_\mu = ABS\} \\ Arr_\mu = \{t' \in \mathbb{R} / P_\mu(t') \neq 0\} \end{cases}$$, then a sound TMDP must verify:

$$\forall (\mu, s, a) \in M_{ABS} \times S \times A, \quad \forall t \in Dep_{\mu,s,a}, \quad \forall t' \in Arr_\mu, \quad t < t'.$$

Despite its simplicity and elegance, the TMDP model suffers from some inconsistencies. It defines a waiting reward rate, but does not explicitly define a "wait" action. Defining an action means being able to write the L, R and P_μ functions for it; L and R could be written for a "wait" action, but the P_μ probability density function is unknown. Instead, the model implicitly inserts waiting times before each action.

TMDP policies are defined as follows: a TMDP policy is a function mapping pairs $(s,t) \in S \times \mathbb{R}$ of initial states and dates, to pairs $(t', a) \in \mathbb{R} \times A$ indicating that the optimal strategy is to wait until time t' in order to undertake action a.

It is unclear to see how TMDPs relate to standard MDPs or SMDPs. Part of this thesis' work has concentrated on bringing the TMDP framework back into a more standard stochastic processes and MDP framework. For this purpose we introduced the SMDP+ model which we will present and compare to the TMDP model in chapter 4. More precisely, we will try to address the following questions:

- What are the mathematical obstacles of including an observable time in SMDPs?
- What is the difference between TMDPs and SMDPs with observable time?
- Is "wait" a standard MDP action ?
- What are really TMDP policies?
- Which is the criterion optimized by TMDP policies?

Based on this analysis, we will try to improve TMDP modeling and resolution through the $TMDP_{poly}$ framework, algorithm and planner in chapters 5 to 7.

These developments will also bring some more general conclusions about the problem of introducing a continuous observable time in the MDP framework in chapter 8. There, we will consider the general problem of observable continuous time in MDPs with parametric (continuous and discrete) action spaces and hybrid state spaces. We will specifically focus on establishing mathematical foundations for proving the existence of an optimality equation which includes both the MDP, TMDP and SMDP+ frameworks; we will call this generalized framework *XMDP*s.

2.2.5 Concurrency as the origin of complexity

With the TMDP model, we have introduced continuous observable time in MDPs and are now able to represent time-dependent stochastic problems of decision under uncertainty. However, it appears that when it comes to writing the transition and reward models for real-world examples in TMDP models the task can become incredibly complicated. The first reason for this difficulty is that the overall stochastic behaviour of temporal Markov decision problems often results from the concurrent influence of several separate stochastic processes (as in the subway, airport or coordination problem). On top of that, when one allows for several actions to be undertaken simultaneously, the possible branching factor in the policy search explodes. These two aspects come from the fact that we allowed concurrency in two different ways.

In the CoMDP model ([Mausam and Weld, 2005]), Mausam tackled the problem of authorizing the combination of different actions to be undertaken at the same time. However, the framework of CoMDP remained in a discrete time setup with fixed time steps. Since our focus here is on time-dependency and temporal complexity, we won't enter the CoMDP model in detail. We will remain in the framework of sequential decision theory and thus will not consider the combinatorial complexity of allowing concurrent actions to be undertaken at the same time. However, our final conclusions will show how our results extend to the case of these concurrent actions.

The complexity of our problems comes from the fact that — in the subway problem for example — different simple stochastic processes affect the same common state space. Predicting the next state of the system implies taking into account in the transition function the probability that the first event to trigger is the arrival of a passenger at station 1, or the arrival of a passenger at station 2, or a train movement between station 5 and 6, etc. Additionally to the events' concurrence — which introduce a first modeling difficulty — the individual processes are themselves time-dependent, adding to the complexity of the global process' behaviour. This simple example gives both an idea of the origin of our problem's modeling complexity and a hint as how to go around this difficulty.

Considering concurrent continuous-time stochastic processes is a framework known in the stochastic processes literature as *generalized* processes. It doesn't really make sense to consider Generalized Markov Process since they would all be synchronous and would result in a trivial global Markov Process. However, as soon as we allow for real-valued stochastic transition times, then having several concurrent processes induces a new kind of non-trivial stochastic processes. The concurrent execution of several semi-Markov processes (SMPs) affecting the same state space results in a global stochastic process called a Generalized Semi-Markov Process (GSMP). GSMPs were first introduced in [Glynn, 1989] and have

been extensively studied in the stochastic processes and discrete event systems literature (as in [Nielsen, 1998] for example).

Chapter 11 will present GSMPs more in detail and will highlight their general relation with the global discrete events systems (DEVS, [Zeigler, 1976]) theory. Formally, a GSMP (Cf. [Glynn, 1989] for further details) is described by a set S of states and a set E of events. At any time, the process is in a state s and there exists a subset E_s of events that are called *active* or *enabled*. These events represent the different concurrent processes that compete for the next transition. To each active event e, we associate a clock c_e representing the duration before this event triggers a transition as presented on figure 2.6. This duration would be the sojourn time in state s if event e was the only active event. The event e^* with the smallest clock c_{e^*} (the first to trigger) is the one that takes the process to a new state. The transition is then described by the transition model of the triggering event: the next state s' is picked according to the probability distribution $P_{e^*}(s'|s)$. In the new state s', events that are not in $E_{s'}$ are disabled (which actually implies setting their clocks to $+\infty$). For the events of $E_{s'}$, clocks are updated the following way:

- If $e \in E_s \setminus \{e^*\}$, then $c_e \leftarrow c_e - c_{e^*}$
- If $e \notin E_s$ or if $e = e^*$, pick c_e according to $F_e(\tau|s')$

The first active event to trigger then takes the process to a new state where the above operations are repeated. The framework of GSMPs could be compared with the (deterministic) framework of Timed Automata ([Alur and Dill, 1994]).

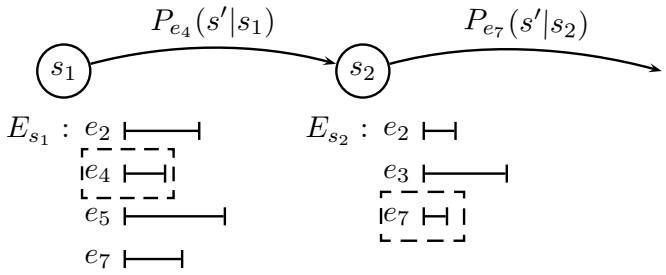


Figure 2.6: Illustration of a GSMP

One first important remark concerning GSMPs is that the overall process does not retain Markov's property anymore: knowing the current state s is not sufficient to predict the distribution on the next state of the process. [Nielsen, 1998] showed that by augmenting the state space with the events' clocks, one could retain the Semi-Markov behaviour for a GSMP.

Introducing action choice in a GSMP yields a GSMDP as defined by [Younes and Simmons, 2004]. In a GSMDP, we identify a subset A of controllable events or actions, the remaining ones are called uncontrollable or exogenous events. Actions can be enabled or disabled at will and the subset $A_s = A \cap E_s$ of activable actions is never empty since it always contains at least the "idle" action a_∞ (whose clock is always set to $+\infty$) which, in fact, does nothing and lets the first exogenous event take the process to a new state. As in the MDP case, searching for control strategies on GSMDP implies defining rewards $r(s, e)$ or $r(s, e, s')$ associated to transitions and introducing policies and criteria.

The GSMDP framework, with and without continuous observable time, will be developed in chapters 11 and 13. In chapter 13 we will especially focus on designing efficient algorithms for solving time-dependent GSMDPs.

2.2.6 Models map

Figure 2.7 summarizes the relationship between all the models presented here, from standard Markov Processes to Generalized Semi-Markov Decision Processes with continuous observable time.

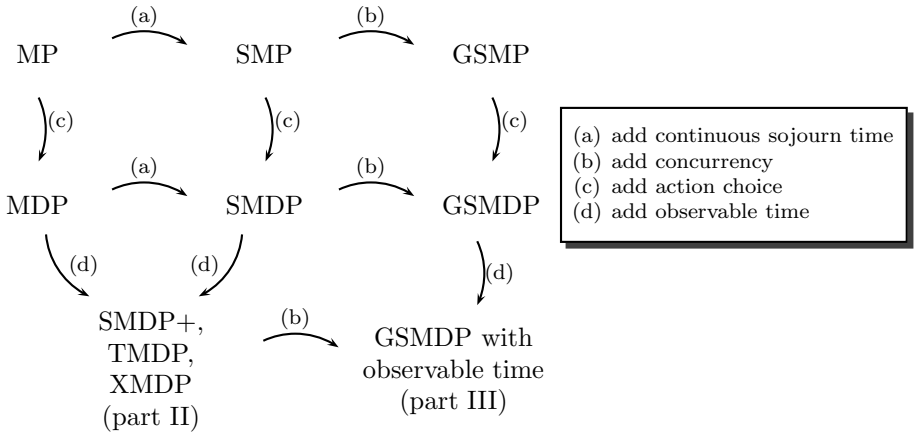


Figure 2.7: Models relational map

2.3 Similarities and differences with “classical” MDP problems

The last section presented a short walk-through about MDP models that focused on dealing with continuous time, either as a temporal extension of actions, as a way of modeling non-stationarity, or as a source of complexity. The examples presented in section 1.3.1 highlight the fact that classical problems which are rather well studied in the stationary case raise new difficulties in time-dependent frameworks.

Dealing with time as a continuous observable variable reaches out of standard frameworks and calls for specific modeling. Is time a resource? If so, is it bounded? What about the case of infinite horizon planning? But then, what is the definition of “horizon”? Is it a mission ending date or a number of actions the agent can perform? If it is not a resource, is time a state variable? If so, then we should be able to point out the effects of actions upon it (write transition models).

In the following paragraphs, we try to break this ambiguity on vocabulary concerning the time variable. This will highlight where standard MDP methods can be reused for time-dependent problems and where specific structure arises from having a continuous observable time.

2.3.1 Three different meanings for a single variable

Techniques for solving continuous-variables MDPs have been developed recently in the planning community (for example [Feng et al., 2004; Hauskrecht and Kveton, 2004; Mausam et al., 2005; Li and Littman, 2005]), but all deal with the time variable as a bounded resource. In fact, time is a strange variable with regard to our problems since it introduces an additional coupling between states and rewards by affecting the discount factor in our criterion, but also remains a state variable (an internal decision variable for the policy), and lastly time is a non controllable variable, growing as the plan executes.

Actually, separating the different notions of *time* is important to understand time dependent problems:

- We first consider the time of the underlying Markov chain. We took care of writing this variable as δ in equations 2.8 to 2.11. This time is discrete, it represents the successive *decision epochs*. At each of these decision epochs, state variables (continuous or discrete), including time, take different values according to the actions undertaken and the transition model. As was illustrated in section 2.2, in continuous-time processes, the time of the Markov chain does not necessarily agree with the physical time of the process.

- The same section 2.2 introduced the notion of *transition time*, describing the sojourn time in a given state before the discrete transition to its successor state. We will write this *sojourn time* or *sojourn duration* with the variable τ since it is a duration, as opposed to the absolute date of the current decision epoch.

- Then we need to consider the state variable t itself. This variable describes the physical time of the process — its main clock — and intervenes in the discounted criterion (as in SMDPs in equations 2.26 and 2.27). Its only dependence on the δ variable is that it never decreases as δ increases, thus yielding the structure of time-dependent problems. It is indeed a continuous state variable and therefore needs to be bounded (so that the state space remains countable). We will see in the next paragraph why this is not a hard constraint on the problem.

- Finally, we have mentioned the problem of the "wait" action which was not well-defined in the TMDP (and SMDP+) model(s). This action does not have any meaning if it is not associated with a *waiting duration* or *waiting date* which was well illustrated by the policy definition of [Boyan and Littman, 2001]. This last time variable is an action's parameter, independent from state variables and process' time. We will explore the framework of hybrid parametric actions in section 8.

Therefore, the time variable links together some non-controllable aspects (the process' time) and some controllable features (the system's state) of the problem. Action's parameters and non replenishable resources can play a similar role on the system's dynamics. Our focus here is on the time variable, so we will concentrate on this one, keeping in mind that the results we obtain for the time-dependent framework can be extended to larger setups.

2.3.2 Redefining the notion of horizon

The next question we need to answer concerning the time variable concerns the definition domain of the global process' clock: is it necessarily bounded?

When we try to build a model of decision under uncertainty with continuous observable time, we need to consider the question of the horizon. It is important to make a difference

between the succession of decision epochs — corresponding to the number of undertaken actions, *i.e.* the time of the discrete Markov chain during execution — and the "current time" variable, continuous, observable and non-decreasing. This implies making a difference between the *planning horizon* and the *temporal horizon*.

In standard finite horizon problems the number of actions to undertake is bounded. In this case, a fine discretization of time might be feasible. But most problems do not allow for finding an upper bound on the number of steps to reach the goal, and when they do, it is often too large and the problem is considered as having an infinite horizon.

Thus, our interest goes to modeling our problem with an infinite *planning horizon*. However, the knowledge about the problem's non-stationarity only extends up to a certain date in the future[3]. We call this date the *pseudo-horizon*. Beyond this date, the problem is considered as stationary (or the horizon states are supposed to be terminal states). In particular, in the cases of model learning, online planning, or plan repair, the pseudo-horizon is a moving horizon. This is why we consider infinite *planning horizon* problems with finite *temporal horizon* or finite *pseudo-horizon*.

Sometimes, when the problem is offline, the moving pseudo-horizon is fixed. Then, we can consider that planning with respect to a continuous observable time variable corresponds to planning in an infinite horizon setup with a bounded time resource than cannot be refilled. In this case, standard methods from the literature for continuous MDP solving can be applied (if they also apply to the rest of the state space). Few methods really deal with hybrid state and action spaces, therefore, the work presented on TMDP solving in the thesis' next part should be considered from the two different points of view. The first point of view concerns the problem of solving time-dependent problems. While the second one highlights the fact that the method developed here is indeed an algorithm for solving MDPs with hybrid state and action spaces.

2.3.3 Exploiting the structure of time-dependent problems

There is one last thing that needs to be mentioned about time. Even though, as we have just seen, it often is a bounded state variable, the fact that this variable is non-replenishable introduces structure in the evolution of the process. Namely, all states with t being strictly smaller than the current date are non-reachable states. Moreover, in real-life problems, instantaneous loops always come to an end and the time variable eventually grows and reaches the pseudo-horizon. This means there is a null probability of observing an infinite sequence of instantaneous transitions. In other words: executing a plan always reaches the pseudo-horizon.

[3]We do not consider periodic problems on purpose here. Namely, we suppose these problems can be dealt with as finite horizon problems.

Finally, as we have explored — without entering too much in the modeling details — the impact of making time continuous and observable in MDPs, it appears that:

- This time is (indeed) a state variable,
- but it shouldn't be confused with the process' discrete time (succession of discrete decision epochs).
- It can usually be bounded, at least as a moving horizon.
- However, it induces a specific quasi-loopless structure.
- Modeling and exploiting this structure in the framework of MDPs seems necessary to build efficient algorithms in order to generate efficient time-dependent plans or policies.

3 Thesis outline

In order to organize the successive ideas leading to our contributions and to facilitate the reader's progression across the chapters, this thesis is divided in four main parts.

Part I provided an introduction, both to the general problem of decision and to the question of introducing time in MDPs. This general introduction, in chapter 1, led to a review of models in chapter 2. These models focus on the integration of the time variable in the MDP framework. They are discussed and compared in order to highlight their specificities and to introduce the first ideas as to the mechanisms involved in their resolution. These formalisms provide the modeling basis which is reused and developed throughout the thesis.

When dealing with explicit time-dependent models, one needs to question a strong hypothesis of standard MDPs: is the model stationary anymore? More specifically, how do we model the exogenous evolution due to the environment, the system's intrinsic temporal behaviour, the opponent's or ally's actions, etc.? [Boutilier et al., 1999] makes a distinction between *implicit-event* models, where the environment's evolution and effects are factored into the representation of stochastic actions, and *explicit-event* models, where change caused by the environment is modeled separately from change caused by the agent's actions. Part II deals with *implicit-event* temporal models, trying to highlight the structure of the temporal problem and to build an adapted algorithmic solution to the resolution of the associated problem. Then, part III illustrates why such *implicit-event* models are hard to build and how one can use *explicit-event* models to learn a policy. Thus, one can summarize the question addressed by each part as:

Part I	General introduction and models
Part II	*Implicit-event* models and continuous observable time
Part III	Learning policies in *explicit-event* temporal models with hybrid state spaces
Part IV	General conclusion

In part II, our attention goes to the straightforward idea of introducing an observable, continuous time variable in an MDP model. In the literature, this approach is known as the TMDP model. We link TMDPs with SMDPs by introducing observable time in SMDPs (chapter 4). Then we improve the TMDP framework's expressiveness by extending the family of continuous functions its resolution can handle in chapter 5. We also improve the resolution scheme itself by introducing the specific $TMDP_{poly}$ algorithm in chapter 6 and evaluate this resolution in chapter 7. This work inside the TMDP framework extends to the more generic framework of time-dependent, *implicit-event*, hybrid state and action problems

for which we introduce the XMDP formalism in chapter 8. Chapter 9 introduces unfinished work presenting an alternative to the previous approaches. We keep this chapter in the thesis' corpus for three main reasons: first it provides an interesting algorithmic alternative in itself, secondly it highlights one of the weaknesses of the previous $TMDP_{poly}$ approach, and finally it introduces the first ideas underlying part III. Finally, chapter 10 summarizes our results on the question of introducing a continuous, observable time variable in *implicit-event*, time-dependent MDPs.

Part III begins with a — somehow — admission of failure: for complex domains, *implicit-event* models are generally not available. Chapter 11 explores the question of modeling temporal complexity in stochastic problems. It does so from the generic discrete events systems point of view and makes a link with the Generalized Semi-Markov Decision Processes framework, illustrating why constructing an *implicit-event* model is much harder than assembling the corresponding *explicit-event* model. Then, chapter 12 takes a brief step out of the framework of temporal problems to review the approximate and asynchronous Policy Iteration approaches in order to introduce the general idea of Real-Time Policy Iteration and to relate it as much as possible to existing approaches. Finally, chapters 13 and 14 apply the RTPI ideas to the case of temporal domains, using the simulation properties of *explicit-event* models introduced in chapter 11 and introducing specific notions related to exploration and generalization.

Finally, part IV contains a single conclusion chapter which tries to summarize the thesis' contributions.

Each part begins with a short overview, introducing the problematic at hand, summarizing the questions addressed in each chapter and presenting the organization of developed ideas. Then we introduce each chapter with a brief abstract of the problem addressed and, along the document, framed boxes try to highlight the essential results punctuating the reasoning's progression.

Part II

Planning with Continuous Observable Time in Markov Decision Processes

Overview

This part presents our contribution to model-based MDP solving when time is made continuous and observable in the decision-maker's model. This characteristic allows to consider non-stationary problems where the transition and reward functions depend explicitly on the continuous time variable.

Introducing explicit continuous time in MDP modeling raises a certain number of issues. Among these, we will look specifically at the following questions:

- How do we model the actions affecting the time variable? How should we represent the temporal consequences of actions within an MDP framework?
- Can we represent idleness in a discrete event model? Is there a difference between idleness and waiting?
- Which is the most suitable way to represent continuous evolution of the model? In practice, what kind of methods can we use and what are the appropriate representations (function classes) for these methods?
- How do we represent a policy? What kind of algorithmic precautions should we take to infer policies in practice?
- How do we make the link between policies and value functions with respect to this continuous time?
- How should we exploit this observable time to structure our policy search?

The course of our reasoning goes as follows. We start with the classical model of Semi-MDPs which includes temporal extensions of transitions and investigate what is needed to use this model in order to plan with respect to an observable time. This leads us to consider the questions of:

- Is the SMDP hypothesis of transition probability and transition duration independence still valid when one wishes to plan with respect to this observable time? How should the SMDP model be adapted to such representation constraints?
- Can we model idleness in a discrete event model? Is there a difference between idleness and waiting?

Then our attention turns to the class of problems introduced by [Boyan and Littman, 2001], known as Time-dependent MDPs (TMDPs). We try to relate the model of SMDPs with continuous observable time — which we call SMDP+ — with the TMDP model. This helps us answer the following questions:

- What criterion is really optimized with the dynamic programming equations of [Boyan and Littman, 2001]?
- Are there implicit assumptions concerning the TMDP model which need to be pointed out to improve the resolution of TMDP problems? Namely:
- Can TMDPs represent *all* time dependent problems? Including the ones where the outcome state depends on the transition duration (and not the opposite)?

- What are the assumptions behind the "dawdling" authorized by TMDPs and how do they affect the optimality equations?

This exploration of the TMDP model will highlight both its advantages and limitations. Then we focus on the TMDP resolution itself. Boyan and Littman introduced an exact resolution scheme for TMDPs. We try to find out to what extent it is possible to expand this exact resolution to a wider class of continuous temporal descriptions. This leads us to investigate the questions of:

- Given the TMDP optimality equations, can we find a class of functions which would be stable though value iterations, *ie.* for which V_{n+1} would belong to the same function space as V_n?
- What would be a reasonable set of hypotheses on the model to insure that the value function belongs to this function space?
- How would these hypotheses relate to the exact resolution framework of [Boyan and Littman, 2001]?

Finally, based on the previous analysis, we slightly extend the exact resolution framework and design an approximate algorithm which provides L_∞ bounds on the value function and exhibits good convergence properties thanks to the adaptation of the Prioritized Sweeping algorithm to TMDPs. The efficiency of this algorithm also relies a lot on the introduction of a specific piecewise polynomial framework and dedicated approximation algorithms. This allows us to answer the practical question:

- Which are the advantages and drawbacks of our $TMDP_{poly}$ algorithm which is meant to extend the standard TMDP resolution?
- More specifically: what can we expect from the "formal Bellman backups" on piecewise polynomial representations?
- And finally: how does this approach scale to temporal planning domains such as the Mars rover benchmark or the UAV coordination problem?

This exploration of the TMDP framework then leads us to a second thought about the nature of the *wait* action and the place of time in our problem. We consider the idea that *wait* is a specific continuous parametric action; this leads us to generalize the framework of TMDPs to a more general model which we call XMDP and which improves on the MDP model in two ways:

- First it considers a generalization of actions. Instead of considering raw discrete or continuous actions, it introduces structure by differentiating actions of distinct nature (*wait*, *walk*, ...) and by associating them with their respective continuous or discrete parameters. Hence, XMDPs consider *parametric actions*.
- Secondly, it provides an extension of the standard Bellman equation to the case of discounted MDPs with observable time, hence proving the soundness of a formal extension of TMDPs to hybrid state spaces, hybrid parametric action spaces and discounted criteria.

This XMDP framework thus provides a general model for implicit-event Temporal Markov Decision Problems.

This course of reasoning and the associated mathematical, modeling and algorithmic issues are linearly addressed throughout the following chapters.

- Chapter 4 establishes the link between the well-explored framework of Semi-Markov Decision Processes and TMDPs. Its goal is to point out two different features: on the one hand, we consider TMDPs under the light of temporal extensions of MDPs, showing which hypotheses are implicitly made to transform SMDPs with observable time into TMDPs. On the other hand, we try to highlight why and how is a TMDP different from a hybrid variable MDP.

- Chapter 5 focuses on the dynamic programming equations introduced in [Boyan and Littman, 2001]. It presents our attempt at finding a class of functions which is stable by the Bellman operator for TMDPs. More specifically, our contribution extends slightly the results for exact resolution presented by Boyan and Littman, highlights the difficulties and interests of using piecewise polynomial functions for TMDP solving and opens the door to the approximate resolution scheme presented in the following chapter.

- Then, in chapter 6 we present our $TMDP_{poly}$ algorithm designed to efficiently solve generalized TMDPs. It relies on the properties of exact and approximate operations on piecewise polynomial functions, makes use of convergence bounds for Approximate Value Iteration and implements an adapted version of Prioritized Sweeping for generalized TMDPs.

- Chapter 7 presents the experimental results of the $TMDP_{poly}$ planner implemented from the $TMDP_{poly}$ algorithm. Its performance and outputs are experimentally evaluated on different temporal Markov problems.

- In chapter 8 we bring mathematical foundations to an extension of TMDPs. We generalize the concept of idleness defined in TMDPs to the case of hybrid (continuous and discrete) actions. We define the XMDP framework on the basis of MDPs with observable time and hybrid states and actions. Then we introduce an extended Bellman equation for XMDPs and provide a sound set of hypotheses in order to extend the classical Bellman operator's properties. XMDPs include standard MDPs and TMDPs and provide a more general mathematical foundation to the problem of modeling and solving MDPs with observable time.

- Chapter 9 presents a possible perspective of the previous work. It introduces the idea of incrementally finding the policy's temporal bounds via the resolution of a sequence of discrete problems. Somehow in-between Value Iteration and Policy Iteration, the proposed method gives the first hints as to the model-free algorithms which will be presented in the next part of the thesis.

- Finally, chapter 10 summarizes the results and contributions seen throughout the previous chapters, highlights their strengths and weaknesses, and presents how they can contribute to more general MDP optimization methods.

4

Bridging the gap between SMDP and TMDP: the SMDP+ model

The previous part provided an introduction to models and frameworks designed to take the temporal consequences of actions into account in the MDP framework. The TMDP formalism of [Boyan and Littman, 2001] seems to be a natural way of modelling time dependency in MDPs. However, the connection with continuous-time discrete-event decision processes such as SMDPs is unclear. In this chapter, we will focus on the continuous observable time variable of the TMDP model and try to establish the link between SMDPs and TMDPs. Namely, we answer the question "are TMDPs equivalent to SMDPs with observable time?". Another important question we will try to answer regards the definition of inactivity: "How should we describe *idleness*? Can it be described within a discrete event framework? Is it equivalent to *waiting*?". We introduce the SMDP+ model for this purpose, highlight which criterion is really optimized in TMDPs in order to define policies, and clarify these questions concerning *idleness*.

4.1 Making time observable in SMDPs

The first step in introducing time in MDPs was to define Semi-MDPs (section 2.2.2) and to introduce continuous action duration. It appears natural to build on the SMDP model in order to go one step further. This step corresponds to defining a model where time intervenes not only as a random duration between decision epochs, but also as an observable continuous variable in the state space, therefore permitting the definition of non-stationary, continuous time, discrete event problems.

In the SMDP model, writing the transition model under the form of $Q(\tau, s'|s, a) = P(s'|s, a) \cdot F(\tau|s, a)$, implicitly implies that:

- The model is stationary (no dependency on t in Q),
- The transition duration τ and the post-action state s' are independent.

We introduce the SMDP+ model which extends the SMDP model with the following features:

- Explicit dependency on the current date for the transition and reward models,

- Possible dependency between post-action state and sojourn time.

The problems we wish to consider do not usually satisfy the above conditions of stationarity and independence between variables. For example, the outcome of a "take a photo" action for the Mars rover depends on the time of day (non-stationarity) and its duration depends on the success or failure of the action.

> Time-dependency is expressed through continuous evolution of the model with respect to the continuous time variable. Post-action states and action durations are often linked.

In order to overcome this modeling issue, we define an SMDP+ as a 4-tuple $\langle \Sigma, A, Q, R \rangle$:

- Σ is the augmented state space containing all $\sigma = (s, t)$ elements. This state space can be decomposed into:
  - a discrete state space $s \in S$,
  - a continuous time axis $t \in \mathbb{R}$.
- A is the discrete action space.
- $Q(\sigma'|\sigma, a)$ is the cumulative transition model. It can be written $Q(\sigma'|\sigma, a) = P(s'|s, t, a) \cdot F(t'|s, t, a, s')$. As in SMDPs, F is the duration model's cumulative distribution function. As previously and for convenience, we will write the probability density functions indifferently as $f(t'|s, t, a, s')$ or $f(\tau|s, t, a, s')$, with:

$$f(t'|s,t,a,s') = \begin{cases} 0 & \text{if } t' < t \\ f(\tau = t' - t|s,t,a,s') & \text{if } t' \geq t \end{cases}$$

- $R(\sigma', a, \sigma)$ is the reward model.

One can note that we can write either $F(t'|s, t, a, s')$ or $F(\tau|s, t, a, s')$ as long as there is no place left for ambiguity. In our notations, t' always stands for the post-action date, while $\tau = t' - t$ always describes the transition duration (or the state's sojourn time).

Using Bayes rule, we could similarly write the transition model on S as $P(s'|s, t, a, t')$ and the duration model on t' as $F(t'|s, t, a)$ and obtain $Q(\sigma'|\sigma, a) = P(s'|s, t, a, t') \cdot F(t'|s, t, a)$. This is why the SMDP+ model is defined in terms of a $Q(\sigma'|\sigma, a)$ function which — in practice — can be provided either as $P(s'|s, t, a) \cdot F(t'|s, t, a, s')$ or as $P(s'|s, t, a, t') \cdot F(t'|s, t, a)$. In our experiments, the transition duration often depends on the post-action state (for movement actions, for example) so we choose to use the $P(s'|s, t, a) \cdot F(t'|s, t, a, s')$ notation, but some examples where post-action states are more likely to depend on transition durations can be expressed using the other formulation (as for a "run to catch the bus") action.

An SMDP+ policy is defined as a function of $S \times \mathbb{R}$ into A. Evaluating an SMDP+ policy with respect to the discounted criterion of equation 4.1 yields equation 4.2.

$$V^\pi(\sigma) = E\left(\sum_{\delta=0}^{\infty} \gamma^{t_\delta} r_\delta^\pi | \sigma_0 = \sigma\right) \tag{4.1}$$

$$V^{\pi}(\sigma) = \sum_{s' \in S} \int_0^{\infty} \big(R(s', t+\tau, \pi(\sigma), \sigma) + \gamma^{\tau} V^{\pi}(\sigma')\big) \cdot f(\tau|\sigma, \pi(\sigma), s') P(s'|\sigma, \pi(\sigma)) d\tau = L^t_{\pi}(V^{\pi})(\sigma) \tag{4.2}$$

This equation is a natural extension of the standard MDP L^{π} operator to the SMDP+ case. Similarly, the optimality equation becomes equation 4.3.

$$V^*(\sigma) = \max_{a \in A} \left\{ \sum_{s' \in S} \int_0^{\infty} \big(R(s', t+\tau, a, \sigma) + \gamma^{\tau} V^*(\sigma')\big) \cdot f(\tau|\sigma, a, s') P(s'|\sigma, a) d\tau \right\} \tag{4.3}$$

$$V^*(\sigma) = LV^*(\sigma)$$

This chapter focuses on modeling and solving TMDP problems. So, for clarity, we will admit for now the intuition stating that these L^{π} and L operators really provide the value functions of π and π^*. Chapter 8 will focus on proving the mathematical foundations and correctness of equations 4.2 and 4.3 in a more general framework.

Equations 4.2 and 4.3 illustrate the tight coupling between transition dynamics and criterion whenever time is made observable: the τ duration used for the discount factor γ^{τ} is also conditioning the post-action augmented state $\sigma' = (s', t+\tau)$.

4.2 Idleness in the SMDP+ model

As we anticipated in section 2.2, as soon as we introduce continuous time, the idea of using an available "wait" action comes to mind. Hence we need to answer the question: "is there an idle action in the SMDP+ model?". If so, how do we write its transition and reward functions?

We need to consider two options: either we put an "idle" action in the action space A or we don't. The latter implies disabling the option of acting at specific times. If we do not allow idleness, then actions are executed without interruptions and we loose one of the interests of considering a continuous observable time. In the first case, we need to define the transition and reward functions associated with the "wait" action, which highlights the fact that "wait" is an abstract action which does not have a physical impact on the system as long as we don't associate it with a *duration* or an *end date*. More specifically, in TMDPs, a natural modelling of a "wait" action is chosen so as to imply a deterministic effect on the time variable. This effect itself is conditioned on the *duration* or *end date* parameter of the action. Hence, "wait" needs to be associated with a proper parameter to gain the meaning of an action operator. Then, for a "wait(t_{next})" or "wait(τ)" action, one can write the transition and reward models.

> The "wait" action's model can only be formalized with respect to some idleness *duration* or *ending date* parameters. The transition and reward model are conditioned on these parameters.

One simple remark concerning the fact that "wait" is chosen to be deterministic with respect to the time variable in TMDPs: an engineer with a good sense of humour could

decide to model the sleepy behaviour of its robot. He could then state that the decision to wait for 8 minutes might result in a different waiting duration — described for example by a Gaussian process of average 8 and standard deviation 1 — because the robot can fall asleep during idleness phases and not wake up exactly in time. This little example finds echoes in real-world problems, for example waiting before sending a request to a web service can sometimes end up in waiting for a lot longer than expected. This simple remark only highlights the fact that using a deterministic "wait" action is a deliberate choice, adapted to the problem at hand, but which can be questioned for some applications. Since our purpose here is to bridge the gap between SMDPs and TMDPs and since TMDPs consider a deterministic "wait" action, we will use deterministic idleness in SMDP+. However one should keep in mind that "wait(τ)" is not necessarily deterministic in real-world problems.

Additionally, it appears that being idle does not really correspond to "making no change" to the process, since the system might evolve by itself during idleness phases (for example the fuel resource can decrease, the exogenous processes might trigger transitions and change their state, and — of course — our observable time changes). It appears that instead of defining passive idleness, the $wait(\tau)$ (or $wait(t_{next})$) action is a particular action which we consider deterministic with respect to the time variable.

Intuitively, the notion of idleness in mission planning implicitly means "wait until it is time to undertake a new action". Thus, we can give an interpretation of the "idle" action as a "let the system change on its own until the next decision epoch". This next decision epoch occurs whenever we enter any state whose date corresponds to the end of the idleness. This notion can be illustrated in other words: since we only take decisions at decision epochs' dates, then the end of a "wait" action must match the date of the next decision epoch.

It appears that defining the "wait" transition function necessitates knowledge of the decision epoch's date. This "wait" action, applied in (s,t), takes the process to a new state s' described by the natural evolution of the process — everything happening if the agent does not interact with the world, as described by equation 4.4 — and to the date corresponding to the time of the next decision epoch as described by equation 4.5. Thus, the "wait" action's model depends on the dates of decision epochs. More specifically, the "wait" action is an instantaneous jump to the date of the next decision epoch and to a state drawn according to the undisturbed dynamics of the system $W(s'|s,t,t')$. This $W(s'|s,t,t')$ function captures all the influences of what would be the exogenous processes if we were in an explicit-event model.

$$Q(s',t'|s,t,a) = P(s'|s,t,a) \cdot F(t'|s,t,a,s')$$

$$P(s'|s,t,wait,t') = W(s'|s,t,t') \tag{4.4}$$

$$f(t'|s,t,wait) = 1_{t_{next}}(t') \text{ with } t_{next} = \min_{\delta \in \mathbb{N}} \{t_\delta | t_\delta > t\} \tag{4.5}$$

$$Q(s',t'|s,t,wait) = \int_{-\infty}^{\infty} P(s'|s,t,wait,t') \cdot f(t'|s,t,wait) dt'$$

This last paragraph illustrates the specificity of the time variable among state variables:

> Planning with respect to a continuous observable time in MDPs and allowing idleness actions does not imply knowing in advance the dates of the successive decision epochs, however, it implies considering *decision variables* which correspond to these dates — in the case of TMDPs these dates are the parameters of the deterministic "wait" action.

Moreover, from the policy representation point of view, since two successive "wait" actions yield the same result as a single longer one, we might need to factor the set of important decision epochs per state, in order to reach an efficient and compact representation of a policy. In other words, while decision epochs are defined for the whole problem, only a few pivot dates *per state* are crucial to the policy. This idea will be developed in chapter 9.

Finally, we particularize the *wait* action in the action space and write that A only contains "standard" actions, while we write $A+ = A \cup \{wait\}$ the complete action space of the SMDP+, where *wait* actually describes all possible instances of the $wait(t_{next})$ or $wait(\tau)$ actions. Based on the previous definition of idleness and on this augmented action space, we define policies over SMDP+ (and replace A by $A+$ in equation 4.3). This choice of explicitly listing *wait* as an action marks an important difference with TMDPs and helps defining policies using only the action space.

4.3 Then what is the difference between waiting and idleness?

The difference between "waiting" and "idleness" can be better explained by considering a control theory point of view. A policy is a controller over a discrete event system. Each action is an event conditioning the transition to a new state. More specifically, each action remains a discrete event and the system's evolution is made of discrete jumps from state to state. This constitutes the discrete events systems paradigm: event-driven evolution. Whenever the agent enters a new state, it immediately applies the action specified by its policy which takes it directly to the post-action state. In real-time execution, this transition might take time, but the controller is not reactivated until the agent enters the new state. This discrete events system description is to be compared to the continuous control point of view which continuously observes the state and applies the controller's command. With SMDP+ and TMDPs, we are dealing with the discrete events paradigm, so the evolution of the system cannot be continuous. Idleness would correspond to the absence of action, but the absence of action — synonym of the absence of event — in a discrete event system means that the execution is finished and that the system has reached a terminal state. No evolution is possible without events.

So the question is: should an SMDP+ policy be described by some temporal intervals specifying an action and all the other intervals returning no action, or should it specify actions in the same intervals and "wait" actions outside? In the first case, we are out of the discrete events control paradigm, in the second one, we have difficulties writing the preconditions and effects of "wait".

Modeling the continuous dynamics of the uncontrollable part of the environment — which changes on its own, independently of the actions performed — turns to defining exogenous events. The second part of the thesis will deal with such events, however, if the environment's evolution modeling is continuous and if we allow a policy to return no action, then we allow idleness and we define a hybrid controller which escapes both the discrete event and the continuous control modeling frameworks since it requires features from both of them. Such a hybrid controller continuously observes the state of the system while waiting and does nothing until it reaches a new state where its policy prescribes an action, thus switching from continuous to discrete control.

Therefore, defining *idleness* corresponds to defining the absence of action. This takes the SMDP+ problem out of the discrete event control framework and into a *hybrid control* framework. While defining *waiting* actions — associated with appropriate parameters — corresponds to defining specific actions (which might themselves rely on continuous parameters) to control the discrete events SMDP+.

We want to remain in the discrete event control formalism and therefore will describe *wait* actions in our policies. We can keep in mind the possibility of only specifying actions inside some intervals: the next paragraphs will show that in the "deterministic idleness" case, idleness and waiting are equivalent.

4.4 Defining policies

An SMDP+ policy is defined as a mapping:

$$\pi \ : \ \left\{ \begin{array}{ccc} \Sigma & \rightarrow & A+ \\ s,t & \mapsto & a \end{array} \right. \tag{4.6}$$

Applying policy π corresponds to applying action $\pi(s,t)$ in s at t. We build on the intuition that in a given state s, there exist a finite number of intervals included in the $[0;T]$ interval — where T is the pseudo-horizon — over which the policy is constant. For standard actions, this result comes from the fact that the action space A is finite. For *wait* actions, it results from the fact that consecutive *wait* actions all tend to waiting for the same date.

A policy is finally evaluated using the criterion defined in equations 4.1 and 4.2.

Finally, we recall and complete the SMDP+ definition. SMDP+ can be defined by:

- Σ: the augmented state space containing all $\sigma = (s,t)$ elements. This state space can be decomposed into:
  - a discrete state space $s \in S$,
  - a continuous time axis $t \in \mathbb{R}$.
- $A+$: the discrete action space containing standard SMDP actions and a family of explicit *wait* actions defined by their parameters.
- $Q(\sigma'|\sigma,a)$: the cumulative transition model.
- $R(\sigma',a,\sigma)$: the reward model.

4.5 Link between TMDP and SMDP+

We have extended the SMDP model to include s'/τ interdependency and explicit t dependency. This yielded the SMDP+ model which highlighted the specific properties of an *idle* action. We can now turn back to the TMDP model in order to determine whether the models are equivalent — and if not, where the difference lies.

4.5.1 TMDPs are a special case of SMDP+

We recall the TMDP definition here for convenience as it was introduced in section 2.2. TMDPs can be described as:

- S, a discrete state space
- A, a discrete action space
- M, a discrete set of *outcomes* $\mu = (s'_\mu, T_\mu, P_\mu)$ where:
  - s'_μ is the transition's resulting state.
  - T_μ is a boolean indicating whether the probability density function P_μ concerns absolute dates or durations.
  - $P_\mu(\theta)$ is a probability density function describing the probability that the transition ends at time $t = \theta$ (if $T_\mu = \text{ABS}$) or after a duration $\tau = \theta$ (if $T_\mu = \text{REL}$).
- $L(\mu|s,t,a)$ is a transition function[1] giving the probability of triggering outcome μ.
- $R(\mu,t,t')$ is the reward model associated with the realization of outcome μ, starting at t and ending at t'.
- $K(s,t)$ is the reward rate of the "wait" action in state s at time t.

It is then quite straightforward to remark that TMDP and SMDP+ dynamics are almost equivalent. For all standard actions in A, we write:

$$\begin{aligned} \text{SMDP+} &\leftrightarrow \text{TMDP} \\ P(s'_\mu|s,a,t) &= L(\mu|s,a,t) && (4.7) \\ f(t'|s,a,t,s'_\mu) &= P_\mu(t') && (4.8) \\ R(s,t,a,s'_\mu,t') &= R(\mu,t,t') && (4.9) \end{aligned}$$

This parallelism between the two formalisms illustrates their equivalence for describing standard actions' dynamics when one can write transition durations as a function of the transition outcome's state. More specifically, it shows that the TMDP framework relies on factoring transitions by actions' outcomes: one transition is first composed of the action choice, followed by the occurrence of an outcome, itself resulting in a single final state as illustrated on figure 4.1. Therefore, TMDPs describe transitions by factoring them with actions.

This last distinction illustrates the first difference between SMDP+ and TMDPs and a strong restriction of TMDPs: the latter cannot represent cases where the transition's outcome would depend on the transition duration. For example, one cannot model the discrete stochastic fuel consumption of a movement action as conditioned by the movement duration in the TMDP model. On the other hand, since SMDP+ build on the generic basis of SMDPs, they allow such action descriptions.

The main other difference lies in the definition of the "wait" action. The original TMDP of [Boyan and Littman, 2001] defines the possibility for agents' dawdling, specifying a "dawdling

[1] This L is not to be confused with the dynamic programming operator L introduced earlier. We keep this notation in order to be consistent with the notations of [Boyan and Littman, 2001] and will explicitly make the distinction between L and L when ambiguous.

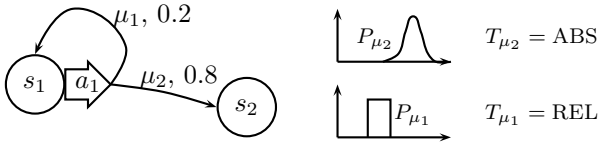


Figure 4.1: TMDP - basic elements

cost" K. However, the action space of the TMDP does not include any "wait" action, contrarily to the action space $A+$ of the similar SMDP+. Instead, Boyan and Littman introduce an extra step in the optimality equations in order to allow for some waiting between each undertaken action. As we will see through the equations of the next paragraph, on top of being deterministic with respect to the time variable, the implicit "idle" action of TMDPs has a very strong restriction, it allows no evolution of the process while waiting: waiting never changes the process' state.

Therefore, our conclusion here is that TMDPs are a special class of SMDP+ problems with an implicit "wait" action that needs to leave the process' state unchanged.

4.5.2 Dynamic programming resolution of TMDPs

We now compare the criteria defined over SMDP+ and TMDPs. We will show — as the intuition suggests – that the optimality equations introduced without mathematical justification in [Boyan and Littman, 2001] correspond to a specific total reward criterion and that the K function is indeed the reward rate of the implicit "wait" action.

In order to find policies for TMDPs using dynamic programming, [Boyan and Littman, 2001] introduce an optimality equation similar to Bellman's equation. The idea is to use Value Iteration in order to iteratively find the optimal value function. This optimality equation should reflect a certain optimality criterion, but since the "idle" action is implicit and since it participates in the optimality equation, it is not obvious to determine which criterion is really optimized. The extended Bellman equation for TMDPs introduced in [Boyan and Littman, 2001] is given by equations 4.10 to 4.13.

$$V(s,t) = \sup_{t' \geq t} \left(\int_t^{t'} K(s,\theta)d\theta + \overline{V}(s,t') \right) \tag{4.10}$$

$$\overline{V}(s,t) = \max_{a \in A} Q(s,t,a) \tag{4.11}$$

$$Q(s,t,a) = \sum_{\mu \in M} L(\mu|s,t,a) \cdot U(\mu,t) \tag{4.12}$$

$$U(\mu,t) = \begin{cases} \int_{-\infty}^{\infty} P_\mu(t')[R(\mu,t,t') + V(s'_\mu,t')]dt' & \text{if } T_\mu = \text{ABS} \\ \int_{-\infty}^{\infty} P_\mu(t'-t)[R(\mu,t,t') + V(s'_\mu,t')]dt' & \text{if } T_\mu = \text{REL} \end{cases} \tag{4.13}$$

The first equation indicates that the optimal expected value in s at time t corresponds to the maximum gain we can hope to get by waiting until t' and then applying an action. Indeed, according to the other three equations, $\overline{V}(s,t)$ represents the maximum gain we can hope for when we act immediately at t. This dynamic programming scheme over TMDPs alternates an optimization phase where one acts immediately and a calculation phase indicating how much one should wait between actions. This way, one obtains a policy defined as $\pi(s,t) = (t',a)$. This policy indicates that, in state s and at time t, the action to undertake

is twofold: "wait until time t' and then undertake action a". We can note that $t' = t$ is possible, which preserves the generic nature of this kind of policy. Therefore, a TMDP policy alternates between elements of A and "idle" phases (which can be of duration zero).

Since we now know that TMDPs are specific SMDP+ and that they simply impose some implicit, static "wait" actions between each standard action, we can define a criterion for SMDP+ and check if the optimality equations 4.10 to 4.13 match equation 4.3 for this criterion. From a common sense point of view, equations 4.10 to 4.13 optimize the total expected reward when alternating wait and action phases. We formally define a TMDP policy as in equation 4.14 and a reward model as in equation 4.15.

$$\pi \ : \ \left\{ \begin{array}{ccc} \Sigma & \rightarrow & \mathbb{R}^+ \times A \\ s,t & \mapsto & t',a \end{array} \right. \tag{4.14}$$

$$r_\delta^\pi = \int_{t_\delta}^{t'_\pi} K(s_\delta, \theta) d\theta + R(s_\delta, t_\delta, t'_\pi) \tag{4.15}$$

Then equations 4.10 to 4.13 correspond to finding the optimal expected total reward value function for the TMDP[a].

[a] The proof is not provided here since it is very similar to proving that "wait" is a parametric action in the XMDP framework which will be introduced in chapter 8. The latter proof is detailed in section 8.4.

This way, we have shown that — once the "wait" action has been made explicit — TMDP's optimization through equations 4.10 to 4.13 is equivalent to the corresponding SMDP+'s optimization with separate actions in $A+$.

One first important consequence of this result on TMDP optimization is that — as in the standard MDP case — one needs to set supplementary hypotheses on the model to guarantee the convergence of Bellman backups, because of this undiscounted criterion. Namely, one should suppose that some states are either absorbing states with null reward or terminal states, and that they are reachable from any point in the state space. With these assumptions, we can safely use this total reward criterion. In practice, these requirements are often met because we are usually in one of the two following cases (or a mix of the two):

- Bounded horizon problem: all reward functions are known until the pseudo horizon and null afterwards, which guarantees convergence of the total reward criterion and makes all actions equivalent after the pseudo-horizon. One can note that in this case, the time variable is bounded and knowledge of the non-stationarity is only necessary inside the bounds on time.
- Stochastic Shortest Path: the reward model is bounded, terminal goal states are reachable from anywhere in the state space and no action is allowed in them, yielding a zero probability of an infinite execution path and a finite total reward criterion. In this case, no hypothesis is made on the pseudo-horizon and the reward models.

Lastly, in order to model more realistic situations, one would need to describe the process' evolution during waiting phases. The general case of SMDP+ presented in equations 4.4 and 4.5 cannot be captured by the TMDP framework since TMDPs impose that the process is static during idle phases. However, it is possible to adapt [Boyan and Littman, 2001]'s

equation 4.10 slightly to match the case of deterministic evolution during idleness. For this purpose we note $s' = w(s,t,t')$ the resulting state of a "wait" action according to the deterministic probability density function $W(s'|s,t,t')$. This way — if we suppose that the system's evolution is deterministic during dawdling phases — equation 4.10 can be replaced by equation 4.16.

$$V(s,t) = \sup_{t' \geq t} \left(\int_t^{t'} K(w(s,t,\theta),\theta) d\theta + \overline{V}(w(s,t,t'),t') \right) \tag{4.16}$$

4.5.3 Policy equivalence

We have now shown that TMDPs were a specific class of SMDP+ problems for which:

- The "wait" action is not explicitly listed in the action space.
- The "wait" action is static, *ie.* waiting never changes the process' state.
- The optimality equations correspond to a total reward criterion.

The last thing to compare between TMDPs and SMDP+ deals with optimal policies. We need to verify that the execution of an SMDP+ optimal policy and the corresponding TMDP policy yield the same execution path.

An SMDP+ policy is given as: "at any time step t, there exists an optimal action to undertake, this action might be waiting, which lets the process change on its own until the next decision epoch's date". A TMDP policy, however, defines pairs of actions "wait until t' then do a". The problem of showing the equivalence of these two behaviours deals with proving that for all instants t'' between t and t', the SMDP+ policy's action remains idleness. We can turn the problem differently and show that for any date t'' between t and t', the TMDP policy's behaviour is constant and equal to "wait until t' then act". We focus on proving this second point.

In order to clarify things we can take the example presented on figure 4.2. Suppose that we are in state s, at time t_1. The SMDP+ policy specifies the explicit "wait" action as the action to perform. An outside observer anticipating on the policy can remark the next action will be a and will be started at time T (in-between, the optimal action remains "idle"). On the other hand, the TMDP policy prescribes to undertake action "wait until T_1 then execute a". By writing down the model's equivalence (equations 4.7 to 4.9) and the optimality equation (equation 4.3) we find that $T = T_1$. The main remaining question is to determine if — by picking t_2 between t and T_1 — the TMDP policy in (s,t_2) is consistent with the SMDP+ policy, *ie.* if $T_2 = T_1$ and $a' = a$.

If we prove that $T_2 = T_1$, then the $a = a'$ result is immediate. Indeed, equation 4.11 shows that the action to undertake at T_1 is uniquely defined by the function $\overline{V}(s,T_1)$. Therefore, all we need to prove is that $T_2 = T_1$.

We introduce the function $T(s,t)$ which gives the waiting end date:

$$T(s,t) : \left\{ \begin{array}{rcl} S \times \mathbb{R} & \rightarrow & \mathbb{R} \\ (s,t) & \mapsto & \underset{t' \geq t}{argsup} \left\{ \int_t^{t'} K(s,\theta) d\theta + \overline{V}(s,t') \right\} \end{array} \right. \tag{4.17}$$

Then we want to prove the following proposition:

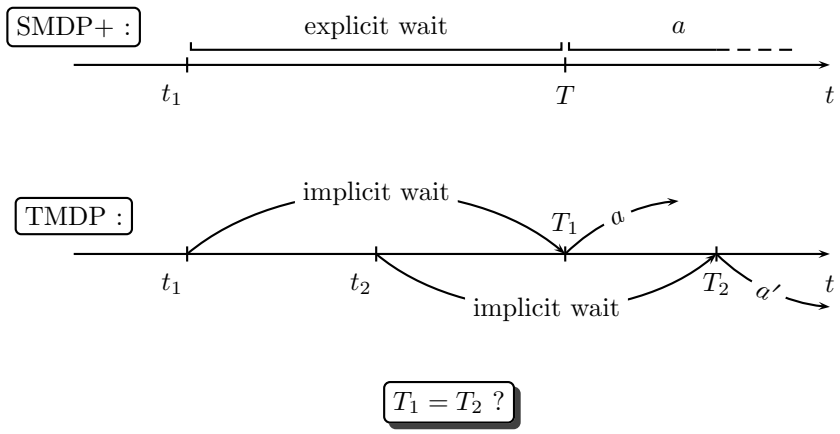


Figure 4.2: Equivalence of SMDP+ and TMDP optimal policies

Proposition. *Let $s \in S$ be a state and $t_1 \in \mathbb{R}$ a time such that $T(s,t_1) > t_1$. Let $t_2 \in \mathbb{R}$ be another time such that $t_2 \in [t_1, T(s,t_1)]$. Then we have $T(s,t_2) = T(s,t_1)$.*

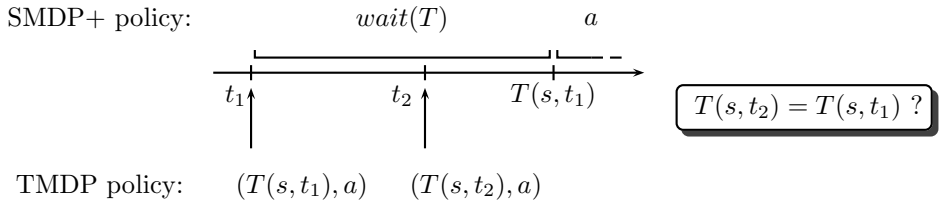


Figure 4.3: The policy equivalence problem

This proposition is illustrated on figure 4.3 and corresponds to the problem we presented on figure 4.2.

Proof. We have $\int_{t_1}^{t'} K(s,\theta)d\theta + \overline{V}(s,t') = \int_{t_1}^{t_2} K(s,\theta)d\theta + \int_{t_2}^{t'} K(s,\theta)d\theta + \overline{V}(s,t')$.
And $T(s,t_1) > t_2$ (the first sup is reached after t_2), so:
$\underset{t' \geq t_1}{argsup} \left\{ \int_{t_1}^{t'} K(s,\theta)d\theta + \overline{V}(s,t') \right\} = \underset{t' \geq t_2}{argsup} \left\{ \int_{t_1}^{t'} K(s,\theta)d\theta + \overline{V}(s,t') \right\}$.
Thus:

$$\begin{aligned} T(s,t_1) &= \underset{t' \geq t_2}{argsup} \left\{ \int_{t_1}^{t'} K(s,\theta)d\theta + \overline{V}(s,t') \right\} \\ &= \underset{t' \geq t_2}{argsup} \left\{ \int_{t_1}^{t_2} K(s,\theta)d\theta + \int_{t_2}^{t'} K(s,\theta)d\theta + \overline{V}(s,t') \right\} \end{aligned}$$

But $\int_{t_1}^{t_2} K(s,\theta)d\theta$ is constant with respect to t', it does not affect the $argsup$. Therefore:

$T(s,t_1) = \underset{t' \geq t_2}{argsup} \left\{ \int_{t_2}^{t'} K(s,\theta)d\theta + \overline{V}(s,t') \right\}$.

And finally: $T(s,t_1) = T(s,t_2)$. □

Finally we have proven that the two optimal policies defined as SMDP+ policies or TMDP policies are equivalent.

4.5.4 Generic nature of TMDP policies

Lastly, it is relevant to note that TMDP policies can capture any sequence of decisions within a TMDP model. Namely, alternating phases of idleness and phases of action does not restrict the set of policies we can consider. This comes from the following reason. If in state s, the optimal policy is to perform a_1 between t_0 and t_1 and a_2 between t_1 and t_2, then the TMDP policy can be written as:

$$\pi(s,t) = \begin{cases} (t, a_1) & \text{if } t \in [t_0, t_1[\\ (t, a_2) & \text{if } t \in [t_1, t_2[\end{cases}$$

In other words, chaining actions (a_3, a_2, a_7) during a given execution path would be strictly equivalent to the sequence $(wait(\tau = 0), a_3, wait(\tau = 0), a_2, wait(\tau = 0), a_7)$ because of the three implicit properties of the TMDP $wait$[2]:

- $wait$ is static in terms of state evolution (waiting leaves the process' state unchanged).
- $wait$ is deterministic with respect to the time variable.
- $wait(\tau = 0)$ provides a zero reward.

These three properties allow TMDP policies to be as generic as SMDP+ ones. One could actually find weaker properties for which such a genericity would be preserved. Namely, if:

- $wait(\tau = 0)$ is static,
- $wait(\tau = 0)$ provides a zero rewards,

then $wait(\tau = 0)$ is a no-op action and can be inserted infinitely often between other actions. TMDP policies break this infinite number of possible insertions by imposing a single waiting action between each other action.

4.6 Conclusion

This concludes this paragraph on the comparison between SMDP+ and TMDP. Its main purpose was to establish the link between the *ad hoc* definitions of the TMDP model and the theory of stochastic decision processes. This comparison highlighted the limits of the TMDP model and its properties. In conclusion:

[2] $wait(\tau = 0)$ is the null duration waiting in TMDPs, it is actually a shortcut for the implicit $wait(t' = t)$ in the TMDP action $\pi(s,t) = (t,a)$.

A TMDP is a total reward criterion SMDP+ where the "wait" action is implicit. This is made possible because this same "idleness" action is static in terms of the state's evolution (waiting leaves the process' state unchanged) and deterministic with respect to time. Since the *wait* action is implicit, TMDP policies are a constant alternate of standard actions and idleness phases. Such a policy's genericity is only preserved due to the fact that "wait" does not affect s and that its reward model yields reward 0 for zero-duration waiting. One could notice that such a genericity would still be preserved under the weaker condition that $wait(\tau = 0)$ leaves the process' state unchanged[a] and induces no cost or reward.

[a] this remains true as long as *wait* is the only parametric action. The general case of parametric actions will be developed in chapter 8.

This analysis raises some questions concerning the nature of this "wait" action. Namely:

- What if there are other continuous parametric actions like "wait"?
- How do we model the exogenous evolution of the world in TMDPs, how do we use the W function of SMDP+?
- Is there a more general framework — derived from MDPs — for planning with continuous time and parametric actions?
- More specifically, couldn't we write a framework with sequences of extended actions (avoiding the permanent switching of wait/action which is — somehow — a tweak in the TMDP resolution) which would be similar to the standard MDP case?

We will bring an answer to these questions in the more general framework of XMDPs, in chapter 8. In conclusion, we have shown the expressivity and the limits of the TMDP framework. In particular, its implicit *wait* action is not generic: it corresponds more to an action that would deterministically "freeze" the process state and move to an other date in time. This important feature was highlighted by the more general *wait* operator of SMDP+. For now, we will use the TMDP framework and notations and will focus on studying and improving the resolution of TMDPs.

5

Solving TMDPs via Dynamic Programming

The previous chapter connected the TMDP framework to the general case of MDPs and SMDPs. It illustrated the fact that the optimality equation on TMDPs corresponded indeed to a total reward criterion over the execution. In this chapter, we focus on this optimality equation. Our goal is to analyze why an exact resolution was possible in the case of [Boyan and Littman, 2001], how we can extend it and what computational tools we need to perform Bellman backups on TMDPs.

5.1 Optimality equations and value function properties

The optimality equations established on the total reward criterion for TMDPs in the last chapter are the basis of the dynamic programming approach to solving TMDPs. Equations 4.10 to 4.13 provide a straightforward value iteration scheme in order to find the optimal value function as presented in equations 5.1 to 5.4.

$$V_{n+1}(s,t) = \sup_{t' \geq t} \left(\int_t^{t'} K(s,\theta) d\theta + \overline{V}_n(s,t') \right) \tag{5.1}$$

$$\overline{V}_n(s,t) = \max_{a \in A} Q_n(s,t,a) \tag{5.2}$$

$$Q_n(s,t,a) = \sum_{\mu \in M} L(\mu|s,t,a) \cdot U_n(\mu,t) \tag{5.3}$$

$$U_n(\mu,t) = \begin{cases} \int_{-\infty}^{\infty} P_\mu(t')[R(\mu,t,t') + V_n(s'_\mu,t')]dt' & \text{if } T_\mu = \text{ABS} \\ \int_{-\infty}^{\infty} P_\mu(t'-t)[R(\mu,t,t') + V_n(s'_\mu,t')]dt' & \text{if } T_\mu = \text{REL} \end{cases} \tag{5.4}$$

In their 2001 paper, Boyan and Littman show that under some conditions, TMDPs can be solved exactly using Value Iteration. These conditions are:

- L and K are piecewise constant functions with respect to t.
- R can be decoupled into a sum of piecewise linear reward functions:

$$R(\mu,t,t') = r_t(\mu,t) + r_{t'}(\mu,t') + r_\tau(\mu,t'-t) \tag{5.5}$$

- P_μ is a discrete probability density function.

Even though the first two conditions are acceptable in order to model and approximate any kind of transition model and reward function, one might wish for more expressive function

shapes. On top of that, looking at discrete probability density functions for P_μ turns out to be a good approximation of the distributions but also takes the process back to the case of discretized transition durations. In this chapter, we will analyse the optimality equations presented above (equations 5.1 to 5.4) and will extend them to more general classes of functions. More specifically, we will show where the limit for exact resolution can be pushed with our approach and how we adapt the exact resolution to the approximate case by generalizing piecewise constant and linear functions to general piecewise polynomial functions.

The core question of this chapter can be stated as follows: we are looking for a value function $V(s,t)$ obeying the previous Bellman's equation. While in the discrete case, tabular representation is the simplest common basis to all representations of the value function, in the continuous case we deal with function spaces. These function spaces are generally hard to approximate and represent because of their infinite dimension. Hence, we are looking for a shape of V which is an efficient approximation and representation framework as well as an adapted formulation for the operations of equations 5.1 to 5.4. The dynamic programming approach relies on the fact that the Bellman's operator is a contraction mapping over value function space and admits a fixed point. In order to build an exact resolution scheme, it is useful to find a family of functions which would be stable by application of L. In other words, we are looking for a class of functions $\mathcal{C}$ for which:

$$\forall V \in \mathcal{C},\ LV \in \mathcal{C} \tag{5.6}$$

However, this search for an "L-stable" class of functions $\mathcal{C}$ should be done while keeping in mind the practical fact that operations on V should be easily computable. Also, we will need to compute operations between V and L[1], P_μ, K and R, which suggests that they might need to belong to the same class $\mathcal{C}$.

5.2 Piecewise polynomial functions

The choice we made in order to restrict the set of functions in which we search the elements of $\mathcal{C}$ is to look at piecewise polynomial functions. There are several reasons for that.

First of all, the initial representation of [Boyan and Littman, 2001] dealt with piecewise constant and linear functions; the transition to piecewise polynomials seems natural. The hard point will be to go from discrete probability density functions to piecewise polynomial ones and to extend the exact resolution method to this generalization.

Secondly, it is easy — in terms of computation — to approximate any distribution or function by a polynomial or a set of polynomials. The theory of splines ([Ahlberg et al., 1967]) is mainly based on this idea.

Moreover, while some phenomena might be better understood and represented with standard distributions such as Bêta, Gaussian, exponential, etc., we need to consider the question of how we will deal with the elements of $\mathcal{C}$ in our algorithms. Equation 5.4 is our main concern here, since a closer look shows that we will have to compute convolutions of P_μ and R for example. Since we are trying to extend the exact resolution of [Boyan and Littman, 2001], we need to be able to analytically compute these convolutions and other calculations. Polynomials convolution yields a new polynomial. Even though it is not a straightforward calculation by hand, it remains an easy analytical machine-calculated result in most cases.

[1] Here, the TMDP's transition model.

Convolution of piecewise polynomials follows the same rule and is a feasible solution to the problem of computing the approximate analytical result of two function's convolution while such calculations might be a lot more complicated to perform on distributions such as Gaussian, Bêta or Dirichlet which have implicitly defined cumulative distribution functions.

Third, as in [Boyan and Littman, 2001] we might want to model non-continuous functions in order to take into account drastic discontinuities in the transition, duration and reward models. This discontinuity applies to specific points in time and the evolution is continuous in-between. Using a piecewise continuous representation for the duration distributions too (instead of discrete probability density functions) might help preserve this property instead of defining more and more different intervals as the number of value iterations grows.

Finally, when we look at the problems presented in section 1.3.1, we can see that the exact shape of the distributions is not always essential to the resolution and that an approximation of their probability density functions is often an acceptable model. Additionally, these probability density functions can present local regularities and specific discontinuities which are easily modeled as splines or — more generally — piecewise polynomial functions. A simple example of this point is given on figure 5.1 where we have represented the probability of triggering the outcome "at destination" when we decide to take the train from INRA to ONERA (the other outcomes might be to end up lost in the wrong station for example)[2].

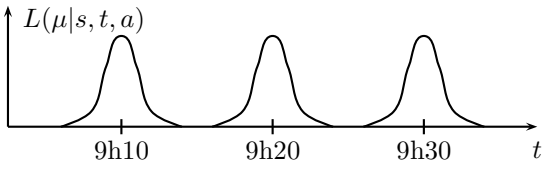


Figure 5.1: Example of $L(\mu|s,t,a)$ function

Our point here is not to indicate that piecewise polynomial representations are better than other ones — which is obviously not the case. However, the arguments above are the practical reasons which led us to choose such representations in order to extend the exact resolution of TMDPs and to build an approximate resolution scheme. From now on, we will try to find a closed-form solution to equations 5.1 to 5.4 by considering families of functions which are as general as possible. When needed, we will base our reasoning and our search for such a closed-form solution on the piecewise polynomial representation of continuous functions and probability distributions.

5.3 Finding a closed-form solution to Bellman's equation

We will now take equations 5.1 to 5.4 and try to see how V changes when we apply these equations.

[2]Note that this function is not a probability density function on the time variable, it is a probability on μ which depends on time. Thus it needs not sum to one when integrated over its definition interval (contrary to the P_μ distributions, for instance).

Let us consider equation 5.1:

$$V_{n+1}(s,t) = \underset{t' \geq t}{sup} \left(\int_t^{t'} K(s,\theta) d\theta + \overline{V}_n(s,t') \right)$$

According to Boyan and Littman's hypotheses, $K(s,\theta)$ is a piecewise constant function so $\int_t^{t'} K(s,\theta) d\theta$ is a piecewise linear function of t and t'. Let us take a simple example and observe which operations are performed to obtain $V_{n+1}(s,t)$ if we suppose $\overline{V}_n(s,t')$ known and if we write $K(s,\theta) = -k$. We have $V_{n+1}(s,t) = kt + \underset{t' \geq t}{sup} \left(-kt' + \overline{V}_n(s,t')\right)$. The time $T(s,t)$ defined by equation 4.17 corresponds to $\sup_{t' \geq t} \left(-kt' + \overline{V}_n(s,t')\right)$. Figure 5.2 illustrates how we go from any $\overline{V}_n(s,t)$ to $V_{n+1}(s,t)$. First we calculate $f(t')$, then $g(t)$ and finally $V_{n+1}(s,t)$.

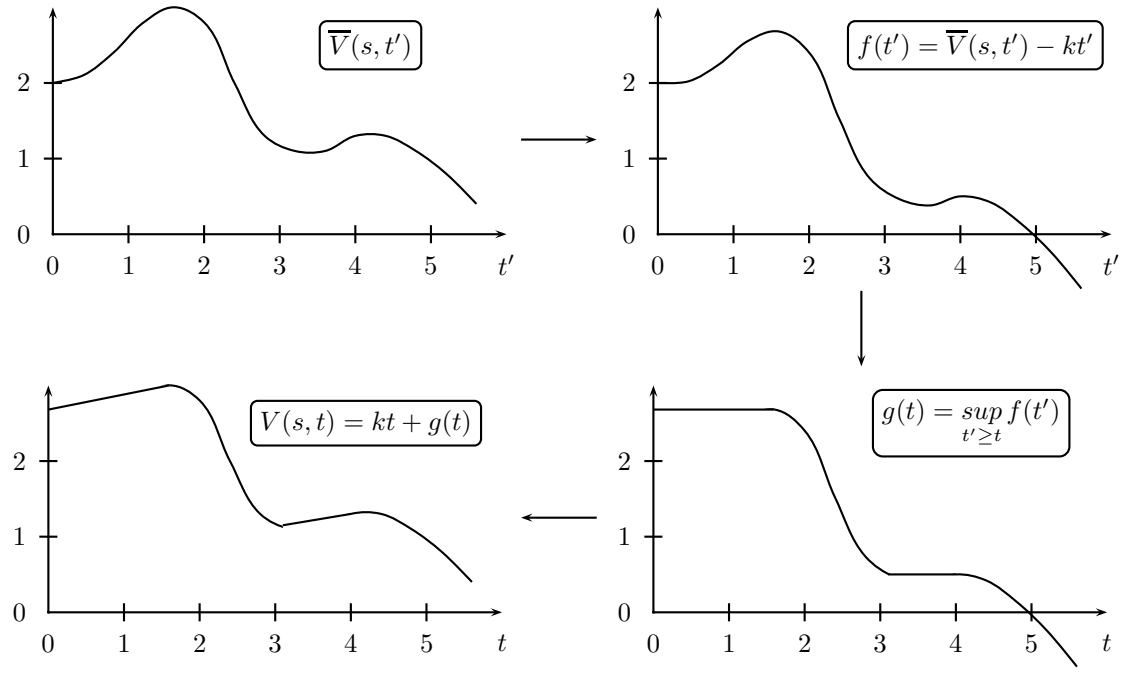


Figure 5.2: Illustrating equation 4.10

We can look at the variations of $V_{n+1}(s,t)$ with respect to t. Let t_1 and t_2 be two instants with $t_1 < t_2$. We want to compare $V_{n+1}(s,t_1)$ and $V_{n+1}(s,t_2)$. We need to distinguish two cases:

1. First case: $T(s,t_1) \geq t_2$. We saw in section 4.5.1 that in this case we have $T(s,t_1) =$

$T(s,t_2)$. Therefore:

$$\begin{aligned} V_{n+1}(s,t_1) &= \sup_{t'\geq t_1}\left(-k(t'-t_1)+\overline{V}_n(s,t')\right) \\ &= \sup_{t'\geq t_1}\left(-k(t'-t_2)+\overline{V}_n(s,t')\right)-k(t_2-t_1) \\ &= \sup_{t'\geq t_2}\left(-k(t'-t_2)+\overline{V}_n(s,t')\right)-k(t_2-t_1) \\ &= V_{n+1}(s,t_2)-k(t_2-t_1) \end{aligned}$$

$$\text{and so: } \frac{V_{n+1}(s,t_2)-V_{n+1}(s,t_1)}{t_2-t_1}=k \tag{5.7}$$

Consequently, $V_{n+1}(s,t)$ is growing with slope k between t_1 and t_2. This can be physically interpreted the following way: if we consider the process is in a state s where the optimal action is to wait in order to get a better reward later, then the expected gain found in t_1 will be lower than the one in t_2 because while the expected reward is the same in the future state, the waiting duration is greater for t_1 (and so is the waiting cost). This situation is illustrated by the linear segments in the representation of V_{n+1} in figure 5.2.

2. Second case: $T(s,t_1)<t_2$. Now there is an action to undertake in $T(s,t_1)$ and the problem considered in t_2 is totally different since we cannot plan to undertake actions in the past. Therefore we know that:

$$\begin{aligned} V_{n+1}(s,t_1) &= \sup_{t'\geq t_1}\left(-k(t'-t_1)+\overline{V}_n(s,t')\right) \\ &= \sup_{t'\in[t_1,t_2]}\left(-k(t'-t_1)+\overline{V}_n(s,t')\right) \text{ , so:} \\ V_{n+1}(s,t_1) &\geq \sup_{t'\geq t_2}\left(-k(t'-t_1)+\overline{V}_n(s,t')\right) \\ &\geq \sup_{t'\geq t_2}\left(-k(t'-t_2)+\overline{V}_n(s,t')\right)-k(t_2-t_1) \\ &\geq V_{n+1}(s,t_2)-k(t_2-t_1) \end{aligned}$$

$$\text{and so: } \frac{V_{n+1}(s,t_2)-V_{n+1}(s,t_1)}{t_2-t_1}\leq k \tag{5.8}$$

This result insures there is no waiting duration allowing for a better expected gain. If we consider an infinitely small distance between t_1 and t_2, then this result illustrates the fact that $\overline{V}_n$ doesn't grow fast enough with t' to compensate the loss due to the waiting cost rate k. Then it is better to act instantly than to wait. These are the cases where $t'=t$ illustrated on figure 5.2 by the regions where $V_{n+1}(s,t)=\overline{V}_n(s,t)$.

Finally the slope of the expected gain V_{n+1} as a function of time is bounded by the opposite of the cost rate. This is not a pessimistic conclusion, on the contrary it provides an upper bound on the value function's improvements: when one is in s at t and knows $V(s,t)$, then he knows that waiting until t' will provide future rewards of at most $-k(t'-t)$.

On top of providing a rough sketch of the algorithm we develop a little further, this analysis of $V_{n+1}(s,t)$'s variations brings up the following conclusion: on some intervals, V_{n+1} is piecewise linear (due to k being piecewise constant, this hypothesis might be relaxed to

piecewise polynomial functions later) and on the others it belongs to the same function class as $\overline{V}_n$. We write this last function class $\mathcal{D}$. If we write $\mathcal{P}_m$ the set of piecewise polynomial functions of degree m defined on $\mathbb{R}$, then in order to characterize the class $\mathcal{C}$ we can write:

$$\forall V \in \mathcal{C}, \exists (p_1, \overline{V}) \in \mathcal{P}_1 \times \mathcal{D} \ / \ V = p_1 + \overline{V} \tag{5.9}$$

We can now keep on sweeping through the optimality equation and look at equation 5.4:

$$U_n(\mu, t) = \begin{cases} \int_{-\infty}^{\infty} P_\mu(t')[R(\mu, t, t') + V_n(s'_\mu, t')]dt' & \text{if } T_\mu = \text{ABS} \\ \int_{-\infty}^{\infty} P_\mu(t' - t)[R(\mu, t, t') + V_n(s'_\mu, t')]dt' & \text{if } T_\mu = \text{REL} \end{cases}$$

By using equation 5.5's decomposition, we have:

$$U_n(\mu, t) = \begin{cases} \int_{-\infty}^{\infty} P_\mu(t')[r_t(\mu, t) + r_{t'}(\mu, t') + r_\tau(\mu, t' - t) + V_n(s'_\mu, t')]dt' & \text{if } T_\mu = \text{ABS} \\ \int_{-\infty}^{\infty} P_\mu(t' - t)[r_t(\mu, t) + r_{t'}(\mu, t') + r_\tau(\mu, t' - t) + V_n(s'_\mu, t')]dt' & \text{if } T_\mu = \text{REL} \end{cases}$$

We write $S_\mu(t') = P_\mu(-t')$ and develop the previous expressions[3]:

$$U_n(\mu, t) = \begin{cases} \left(\int_{-\infty}^{\infty} P_\mu(t')dt'\right) r_t(\mu, t) + \int_{-\infty}^{\infty} P_\mu(t') r_{t'}(\mu, t')dt' + (S_\mu * r_\tau(\mu, \cdot))(-t) + \\ \qquad \int_{-\infty}^{\infty} P_\mu(t') V_n(s'_\mu, t')dt' & \text{if } T_\mu = \text{ABS} \\ \left(\int_{-\infty}^{\infty} P_\mu(t' - t)d(t' - t)\right) r_t(\mu, t) + (S_\mu * r_{t'}(\mu, \cdot))(t) + \\ \qquad \int_{-\infty}^{\infty} P_\mu(t' - t) r_\tau(\mu, t' - t)d(t' - t) + (S_\mu * V_n(s_\mu, \cdot))(t) & \text{if } T_\mu = \text{REL} \end{cases}$$

1. For the first case (T_μ = ABS) and with the piecewise polynomial reward model hypothesis, the U function has the shape of "$\mathcal{P}_m$ + constant + $\mathcal{E}(t)$ + constant".

   The function class $\mathcal{E}$ depends on S_μ. If P_μ is piecewise polynomial then $\mathcal{E} = \mathcal{P}_m$. This is the case we will be using in the rest of this section. We will discuss the value of m a little further. For now we only write that $U \in \mathcal{P}_m$. In order to explain this choice a little better, the next paragraph briefly presents what would happen if we used other distributions than piecewise polynomial. The main result here is that the analytical calculation through value iteration method is not adapted to implicitly defined cumulative distribution function (as for Gaussian or Bêta distributions) and piecewise continuous r and V.

[3] As indicated on page xi, $*$ is the convolution operator.

In the case of $T_\mu = \text{ABS}$, and for a polynomial representation of r, calculating the two first terms of U_n implies calculating the moments of the P_μ distribution. Now if r is defined by pieces, we need to compute separately the "moments" of the P_μ distribution over the different definition intervals of r as illustrated in appendix A. From a practical point of view using Gaussian or Bêta distributions (for which there is no exact result for this calculation) prevents from performing the exact resolution of TMDPs. On the other hand, it is possible to use distributions from $\mathcal{P}_m$ or discrete distributions for this calculation. The case of discrete distributions will be considered later.

2. For the second case ($T_\mu = \text{REL}$) and with the piecewise polynomial reward model hypothesis, the U function has the shape of "$\mathcal{P}_m$ + constant + $\mathcal{E}(t)$ + $S_\mu * V$".

   The first terms in U_n's expression allow us to draw the same conclusions as above: the computation is feasible if $P_\mu \in \mathcal{P}_m$. However, the last term's calculation raises some questions. We know V is a piecewise $\mathcal{C}$ function but we do not have any result concerning its stability by convolution with an element of $\mathcal{P}_m$. For this reason, we decide to further restrict the function space in which we search for elements of $\mathcal{C}$ in order to find an L-stable family of functions.

From here on, we will look for V in the space of piecewise polynomial functions too. This allows us to keep the property of stability by convolution. Therefore we write $V \in \mathcal{P}_m$.

5.4 Bounding the polynomials' degree

We can now take a look at the degree of the elements of $\mathcal{P}_m$ and at how we perform the Bellman update. For this purpose, we need to refine the notations. Let $\mathcal{DP}_m$ be the set of piecewise polynomial probability density functions of degree lower or equal to m. We will write:

- $P_\mu \in \mathcal{DP}_A$
- $r_i \in \mathcal{P}_B$
- $L \in \mathcal{P}_C$

We extend the $\mathcal{DP}_m$ set for $m = -1$ to the set of discrete distributions. This extension is justified by the fact that the convolution of two polynomials of degree p and q yields a polynomial of degree $p+q+1$, while the convolution of a polynomial of degree p by a discrete distribution yields a polynomial of degree p as if the degree of the discrete distribution was -1. The degree of V_n is noted D_n and our goal now is to go through one Bellman backup to determine the degree D_{n+1} of V_{n+1}.

Let $d^\circ()$ be the "degree" operator over polynomials. Equation 5.4 yields:

- if $T_\mu = \text{ABS} : d^\circ(U_n) = A + B + 1$
- if $T_\mu = \text{REL} : d^\circ(U_n) = max\{A + B + 1, A + D_n + 1\}$

If we start the algorithm with a degree zero value function, then we can write — at least for the first iterations: $\forall \mu \in M, \quad U_n(\mu, \cdot) \in \mathcal{P}_{A+B+1}$[4].

Consider now equation 5.3:

$$Q_n(s,t,a) = \sum_{\mu \in M} L(\mu|s,t,a) \cdot U_n(\mu,t)$$

The result is immediate: $\forall (s,a) \in S \times A, Q_n(s,\cdot,a) \in \mathcal{P}_{A+B+C+1}$.

Equation 5.2 does not change the polynomial's degree since it builds a new polynomial by aggregating pieces of Q_n. Therefore: $d^\circ(\overline{V}) = A + B + C + 1$.

Finally, equation 5.1 closes the loop and provides the result on D_{n+1}:

$$D_{n+1} = A + B + C + 1 \tag{5.10}$$

Or with the *max* operator:

$$D_{n+1} = max\{A + B + C + 1, A + D_n + C + 1\} \tag{5.11}$$

Finally, we can draw some conclusions: if $d^\circ(V)$ is initially equal to zero, then after the first Bellman backup $D_1 = A + B + C + 1$. After the second backup, $D_2 = 2A+B+2C+2$, etc. Therefore, $d^\circ(V)$ necessarily increases with the iterations unless $A + C = -1$. The only possible case corresponding to $A + C = -1$ is found for $A = -1$ and $C = 0$. This case corresponds to the situation where:

- P_μ is a discrete probability distribution function.
- L is a piecewise constant function.
- The r_i functions are any piecewise polynomial function of degree B.

Only in this case can we conclude that V always has degree B.

5.5 Is it possible to extend the exact resolution?

The previous paragraphs established a more general result on the stability of V through the TMDP Bellman backups and provided some insight regarding the reason why [Boyan and Littman, 2001] could perform such a resolution. The next question is to try to find all cases where such an exact resolution is feasible — under the modeling hypotheses given at the beginning of section 5.1, namely piecewise polynomial functions and piecewise polynomial or discrete distributions. In the case of an ever-increasing degree of V there is no convergence of Value Iteration since the polynomials' degree keep growing. Actually, convergence is possible but the optimal value function might have an infinite degree. Thus an exact resolution necessarily implies $A + C = -1$. Then we need to check if all values of B allow for an exact calculation of V's coefficients.

For all values of B, equation 5.4 can be easily solved since we know how to calculate the coefficients of U_n without approximation[5]. The same remark also holds for equation 5.3. However, equation 5.2 implies to find — for a fixed s — the intersections of the $|A|$ curves of $(Q(s,t,a))_{a \in A}$. This corresponds to finding the roots of several polynomials of degree B. This operation is known to be feasible without approximation only in the following cases:

[4]The calculations can easily be done with the *max* operator, the reasoning is the same.

[5]for details please see appendix A

- $B = 0$, trivial case
- $B = 1$, linear case
- $B = 2$, Newton's formula
- $B = 3$, Cardan's or Sotta's Formula
- $B = 4$, Ferrari's or Descartes's formula

For $B \geq 5$ Galois proved that there was no general method to find the exact roots of a polynomial in a finite number of calculations. An interesting approximation technique used to find the smallest real root is Sturm's method[6] ([Sturm, 1835]).

Finally, equation 5.1 imposes to find the maxima of piecewise polynomial functions of degree B. This corresponds to finding the roots of polynomials of degree $B-1$; if $B < 6$ this is an exact calculation. Thus, the limiting constraint here comes from equation 5.2. Lastly:

Exact resolution of TMDPs with piecewise polynomial modeling is feasible if:

$$\begin{array}{l} P_\mu \in \mathcal{DP}_{-1} \\ r_i \in \mathcal{P}_4 \\ L \in \mathcal{P}_0 \end{array} \tag{5.12}$$

These results highlight the reason why there is little room for extension of [Boyan and Littman, 2001]'s exact resolution scheme. However, this analysis opens the door to the understanding of an approximate resolution in $\mathcal{P}_m$.

The next chapter will present how the exact resolution is calculated, followed by the approximate resolution scheme. This chapter's conclusion generalizes the results presented in [Boyan and Littman, 2001] by showing the limits of the piecewise polynomial representation framework for exact resolution and by allowing to focus on the difficulties associated with approximate resolution of piecewise polynomial TMDPs.

[6] for details please see appendix A.

6

The $TMDP_{poly}$ algorithm: solving generalized TMDPs

Bellman backups over TMDPs with piecewise polynomial transition, reward and duration functions can be performed analytically, yielding piecewise polynomial value functions. The previous chapter defined the cases when the value function's degree was stable throughout the iterations and when the calculations could be made without approximation. On this basis, we introduce the *TMDP$_{poly}$* algorithm which combines analytical computation of Bellman backups on value functions (with either exact or approximate calculations), L_∞-bounded value function approximation and prioritized dynamic programming for solving the general case of piecewise polynomial TMDPs.

In the case where the TMDP definition obeys equation 5.12 it is possible to perform analytical calculations for the successive Bellman backups of Value Iteration. The next paragraph summarizes the properties obtained from the previous chapters which are used in order to solve TMDPs. It also introduces the basis of the *TMDP$_{poly}$* algorithm which is detailed in the rest of the chapter.

6.1 Extending exact TMDP resolution: some conclusions and properties

1. *Closed-form Bellman backups*: If the reward, transition and duration functions of a TMDP model obey equation 5.12, then value iteration yields a sequence of piecewise polynomial value functions which have a stable (non-increasing) degree.

2. *Interleaving idleness and action*: TMDP resolution can interleave "wait" and "action" phases because $wait(\tau = 0)$ is an action which has no effect on the process' state and no effect on rewards.

3. *Decoupling the equations*: Interleaving these phases corresponds to alternating *wait* and other actions. The consequence on the optimality equations is a decoupling of the calculation. One can calculate first the Q-values of standard actions' (equation 5.3), find the optimal action and the associated value function $\overline{V}$ (equation 5.2) and then calculate $wait(t')$'s Q-value as in equation 6.1 and choose the best t' (equation 5.1).

$$Q(s, wait(t')) = Q_{wait}(s, t') = \int_t^{t'} K(s, \theta) d\theta + \overline{V}(s, t') \tag{6.1}$$

4. *Ordering dynamic programming passes*: As presented in section 2.3.3, making time observable in a planning problem avoids transitions that loop exactly on the same augmented state. However, loops are possible if one only takes the discrete, non-temporal part of the state space into account. The resolution scheme presented above updates the value function, in each discrete state, for all t. Therefore, taking the structure of time and causality into account in TMDP solving via dynamic programming corresponds to updating the states in a pertinent ordering. Intuitively, a good strategy would be to update first the states that are close to the "reward providing states". The idea of updating the states that have important value function change is generalized in the *prioritized sweeping* algorithm first introduced by [Moore and Atkeson, 1993].

Based on these remarks, we try to design an algorithm which implements a simple version of prioritized sweeping on the TMDP framework, first with the exact resolution hypotheses, then with an approximation scheme which allows faster convergence and easier calculations.

6.2 Exact calculation of Bellman backups

We take a model which verifies equations 5.12, namely:

- P_μ is a discrete distribution (as in figure 6.1),
- L is a piecewise constant function of t,
- The $(r_i)_{i\in\{t,t',\tau\}}$ functions are piecewise polynomial functions of degree $B \leq 4$.

More specifically, we can write:

- In the T_μ = ABS case, $P_\mu(t') = \sum_{i=1}^{M} P_{a_i} \cdot \delta_{a_i}(t')$ where δ_{a_i} is Dirac's probability distribution shifted to a_i. Hence: $S_\mu(t') = \sum_{i=1}^{P} P_{a_i} \cdot \delta_{-a_i}(t')$.
- In the T_μ = REL case, $P_\mu(t'-t) = \sum_{i=1}^{M} P_{d_i} \cdot \delta_{d_i}(t'-t)$. Therefore: $S_\mu(t'-t) = \sum_{i=1}^{M} P_{d_i} \cdot \delta_{-d_i}(t'-t)$.
- $r_\tau(\mu,\tau) = \sum_{j=0}^{B} b_j \tau^j$

Let us take the optimality equations one by one. For equation 5.4, case ABS, one has:

$$\begin{aligned}
U_n(\mu,t) &= \int_{-\infty}^{\infty} P_\mu(t') \left[r_t(\mu,t) + r_{t'}(\mu,t') + r_\tau(\mu,t'-t) + V_n(s'_\mu,t')\right] dt' \\
&= r_t(\mu,t)\left(\int_{-\infty}^{\infty} P_\mu(t')dt'\right) + \int_{-\infty}^{\infty} P_\mu(t') r_{t'}(\mu,t')dt' + \int_{-\infty}^{\infty} P_\mu(t') r_\tau(\mu,t'-t)dt' + \\
&\qquad \int_{-\infty}^{\infty} P_\mu(t') V_n(s'_\mu,t')dt' \\
&= r_t(\mu,t) + (r_{t'} * S_\mu)(0) + (r_\tau * S_\mu)(-t) + (V_n * S_\mu)(0) \\
&= r_t(\mu,t) + \sum_{i=1}^{M} P_{a_i}\left(r_{t'}(\mu,a_i) + r_\tau(\mu,a_i-t) + V_n(s'_\mu,a_i)\right)
\end{aligned}$$

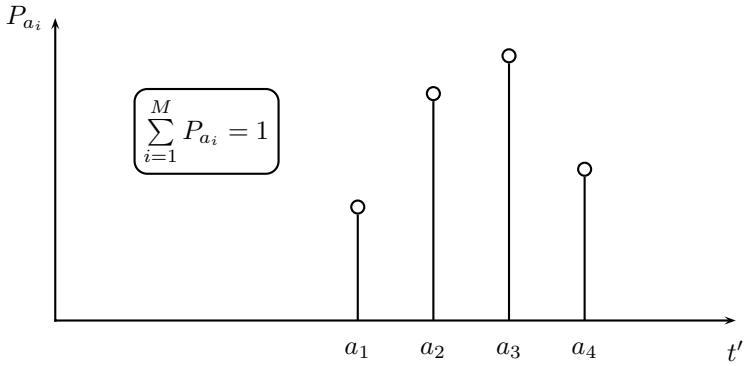


Figure 6.1: Discrete distribution example

We study separately this equation's four right-and side terms. The first term is a degree B polynomial. The second and fourth terms are constant with respect to t. The third term is calculated as follows:

$$\begin{aligned}
r_\tau(\mu, a_i - t) &= \sum_{j=0}^{B} b_j (a_i - t)^j \\
&= \sum_{j=0}^{B} b_j \sum_{k=0}^{j} C_j^k {a_i}^{j-k} (-t)^k \\
&= t^B \left[b_B C_B^B (-1)^B \right] + \\
&\quad t^{B-1}(-1)^{B-1} \left[b_B C_B^{B-1} a_i + b_{B-1} C_{B-1}^{B-1} \right] \\
&\quad t^{B-2}(-1)^{B-2} \left[b_B C_B^{B-2} {a_i}^2 + b_{B-1} C_{B-1}^{B-2} a_i + b_{B-1} C_{B-2}^{B-2} \right] \\
&\quad \vdots \\
&\quad t^l (-1)^l \left[\sum_{k=l}^{B} b_k C_k^l {a_i}^{k-l} \right] \\
&\quad \vdots
\end{aligned}$$

And so:

$$\begin{aligned}
\sum_{i=1}^{M} P_{a_i} r_\tau(\mu, a_i - t) &= \sum_{i=1}^{M} P_{a_i} \sum_{l=0}^{B} t^l \left[(-1)^l \sum_{k=l}^{B} b_k C_k^l {a_i}^{k-l} \right] \\
&= \sum_{l=0}^{B} t^l \left[\sum_{i=1}^{M} P_{a_i} (-1)^l \sum_{k=l}^{B} b_k C_k^l {a_i}^{k-l} \right]
\end{aligned}$$

So we find a polynomial of degree B and its coefficients are calculated with the previous equations (the case of piecewise polynomial calculation follows the same process). For equation 5.4, case REL, one has:

$$\begin{aligned}
U_n(\mu,t) &= \int_{-\infty}^{\infty} P_\mu(t'-t)\left[r_t(\mu,t)+r_{t'}(\mu,t')+r_\tau(\mu,t'-t)+V_n(s'_\mu,t')\right]dt' \\
&= r_t(\mu,t)\left(\int_{-\infty}^{\infty} P_\mu(t'-t)dt'\right)+\int_{-\infty}^{\infty} P_\mu(t'-t)r_{t'}(\mu,t')dt'+ \\
&\qquad \int_{-\infty}^{\infty} P_\mu(t'-t)r_\tau(\mu,t'-t)dt'+\int_{-\infty}^{\infty} P_\mu(t'-t)V_n(,s'_\mu,t')dt' \\
&= r_t(\mu,t)+(r_{t'}*S_\mu)(t)+(r_\tau*S_\mu)(0)+(V_n*S_\mu)(t) \\
&= r_t(\mu,t)+\sum_{i=1}^{M} P_{d_i}\left(r_{t'}(\mu,t+d_i)+r_\tau(\mu,d_i)+V_n(s'_\mu,t+d_i)\right)
\end{aligned}$$

The first term is a known polynomial of degree B and the third term is constant. The second and fourth terms can be found by replacing the $(b_i)_{0\leq i\leq B}$ by the coefficients of r_τ or V_n (the calculation is the same as in the previous case):

$$\sum_{i=1}^{M} P_{d_i} r_{t'}(\mu,t+d_i)=\sum_{l=0}^{B} t^l\left[\sum_{i=1}^{M} P_{d_i}\sum_{k=l}^{B} b_k C_k^l {d_i}^{k-l}\right]$$

Hence, one can easily calculate $U_n(\mu,t)$'s $B+1$ coefficients. The above calculation has been done for a single interval definition of the r_i functions; in the general case of piecewise defined functions, adapting this calculation is just a matter of shifting the definition intervals of r_i's pieces by a_i or d_i in order to find the new intervals of U_n and the process is the same.

Let us move on to equation 5.3. A first step is necessary to find all definition intervals for the Q_n functions. Let us write α_i and β_i the respective bounds of L and U_n's definition intervals. We order the α_i and β_i by increasing order. On each of these new intervals, since L is constant, Q_n is simply obtained by multiplying the coefficients of U_n by L's value. This provides us with the B coefficients of $Q_n(s,t,a)$.

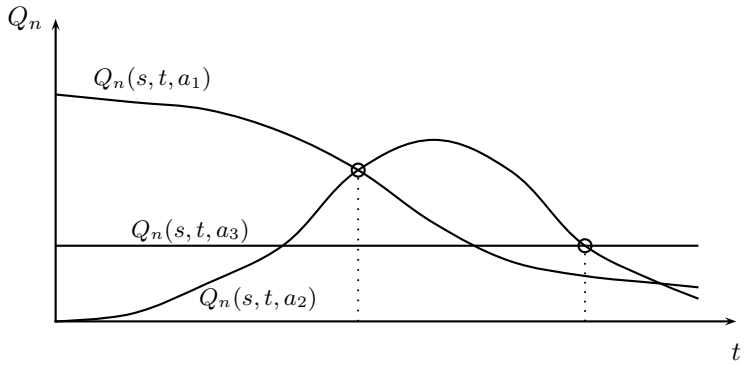


Figure 6.2: Illustrating the construction of $\overline{V}$

Let us recall equation 5.2:

$$\overline{V}_n(s,t)=\max_{a\in A} Q_n(s,t,a)$$

Solving this equation consists in searching for the intersections of polynomials. Let us consider a given s and write $a_m = \underset{a}{argmax}\, Q_n(s,0,a)$. We will iteratively find the max

function for all t as illustrated on figure 6.2 and on algorithm 6.1. We find first the first intersection of $Q_n(s,t,a_m)$ with the Q_n function of another action a. This intersection is located at the smallest root of the $Q_n(s,t,a_m) - Q_n(s,t,a)$ polynomial which is of degree $B \leq 4$.

Thus this root's calculation is an exact operation on the polynomials' coefficients. We need to make sure that the considered point is a real intersection and not a tangent point by verifying the sign change around the intersection. Now we redefine a_m, store the found Q_n in $\overline{V}_n$ and move on to the next intersection. This yields a new degree B piecewise polynomial function for $\overline{V}_n(s,t)$.

Algorithm 6.1: Assembling $\overline{V}$ from the Q functions

$a_{sup} \leftarrow \underset{a}{argmax}\, Q_n(s,0,a)$
$\overline{V}_n(s,t) \leftarrow Q_n(s,t,a_{sup})$
$t_0 \leftarrow 0$
$t_{\text{inter}} \leftarrow 0$
while $t_{\text{inter}} \neq \infty$ **do**
  $t_{\text{inter}} \leftarrow \infty$
  for $a \in A \setminus \{a_{sup}\}$ **do**
    $t_{\text{new}} \leftarrow$ first root of $Q_n(s,t,a_{sup}) - Q_n(s,t,a)$ inside the interval $[t_0, t_{\text{inter}}]$
    (if there are no roots in $]t_0, t_{\text{inter}}]$, then $t_{\text{new}} \leftarrow \infty$)
    if $t_{\text{new}} < t_{\text{inter}}$ and $Q_n(s,t,a_{sup}) - Q_n(s,t,a)$ changes sign in t_{new} **then**
      $a_n \leftarrow a$
      $t_{\text{inter}} \leftarrow t_{\text{new}}$
  $\overline{V}_n(s,t)_{|[t_0,t_{\text{inter}}]} \leftarrow Q_n(s,t,a_{sup})$
  $a_{sup} \leftarrow a_n$
  $t_0 \leftarrow t_{\text{inter}}$

Algorithm 6.1 makes the assumption that the Q functions are polynomial. In the general case of piecewise polynomial functions, we use algorithm 6.2 which takes the possible discontinuities into account.

On top of presenting the complete method for constructing $\overline{V}$ and the corresponding $\overline{\pi}$ policy, algorithm 6.2 is a good illustration of where the algorithmic difficulties of dealing with piecewise polynomial functions are.

In algorithm 6.2, we extend the notion of *dominating* action. An action is said to be dominating in s at t if its Q-value is the highest among the available actions in s. In case of equality, the dominating action is the one which has the highest first non zero derivative. If the equality persists, the actions are considered equivalent and one needs to introduce a static ordering to break the ties.

Algorithm 6.2 computes several intersections of polynomials, incrementally finding the dominating action and searching for the next intersection where another action becomes dominant. Finding intersections of piecewise polynomial functions is equivalent to finding the roots of the difference function. This difference function is called $test(t)$ in algorithm 6.2.

Algorithm 6.2: Assembling $\overline{V}$ and $\overline{\pi}$ from piecewise polynomial Q functions

$t_0 = 0$ `/* Initialization */`
$t_{\text{inter}} = 0$
a_{sup} is the dominating* action in t_0
$a_{sup\_new} = a_{sup}$
$\overline{\pi}[t_0] = a_{sup}$**
while $t_{\text{inter}} \neq \infty$ **do** `/* While intersections are found */`
  $t_{\text{inter}} \leftarrow \infty$ `/* Earliest intersection found so far after` t_0 `*/`
  for $a \in A \setminus \{a_{sup}\}$ **do**
    $t_{cand} = +\infty$ `/* Candidate for earlier intersection */`
    $test(t) = Q(s,t,a) - Q(s,t,a_{sup})$
    $t_{0\_shifted} = t_0$
    $\mathcal{I}$ = definition interval of $test$ to which t_0 belongs
    if $test(t)$ is equal to zero on $\mathcal{I}$ **then** `/* Case: equivalent actions in` t_0 `*/`
      if $\mathcal{I}$ is the last definition interval of $test(t)$ **then**
        $t_{0\_shifted} = +\infty$
      else if the next interval of $test$ starts before t_{inter} **then**
        $\mathcal{I}_{next}$ = the next interval
        $t_{0\_shifted} = \mathcal{I}_{next}$.lower_bound()
        if a dominates a_{sup} in $t_{0\_shifted}$ **then**
          $t_{cand} = t_{0\_shifted}$
          $t_{0\_shifted} = +\infty$
      else
        $t_{0\_shifted} = +\infty$
    if $t_{0\_shifted} \leq t_{\text{inter}}$
    then `/* Case: Q functions intersection */`
      t_{cand} = first point of sign change of $test(t)$ in $[t_{0\_shifted}, t_{\text{inter}}]$ ($+\infty$ if none)
    $t_{new} = +\infty$
    if $t_{cand} < t_{\text{inter}}$ **then** `/* Successful candidate found */`
      $t_{\text{inter}} = t_{cand}$
      $a_{sup\_new} = a$
    else if $t_{cand} = t_{\text{inter}}$ **then** `/* Triple Intersection */`
      This is the case of three Q functions intersecting at t_{cand} : a, a_{sup} and a_{sup_new}.
      a_{sup_new} dominates a so we only need to check if a dominates a_{sup_new}.
      if a dominates a_{sup_new} in t_{cand} **then**
        $t_{\text{inter}} = t_{cand}$
        $a_{sup\_new} = a$
  if $t_{\text{inter}} \neq +\infty$ **then**
    $\overline{\pi}[t_{\text{inter}}] = a_{sup\_new}$
    $a_{sup} = a_{sup\_new}$
    $t_0 = t_{\text{inter}}$

* dominating: highest value or, in the case of value equality, highest first non zero derivative.
** $\overline{\pi}$ is defined on the interval starting in t_0 as a_{sup}, this interval's upper bound will be provided by the next interval's lower bound.

Finally, in equation 5.1, one searches for the solution of the parametric equation:

$$f_t(t') = \int_t^{t'} K(s,\theta)d\theta + \overline{V}_n(s,t')$$

K is piecewise constant so $\int_t^{t'} K(s,\theta)d\theta$ has the form $K_1(t'-\alpha_1) + K_2(\alpha_1-\alpha_2) + \ldots + K_n(\alpha_{p-1}-t)$. Hence:

$$\frac{df_t(t')}{dt'} = K(s,t') + \frac{\overline{V}_n(s,t')}{dt'}$$

Therefore solving $\frac{df_t(t')}{dt'} = 0$ is equivalent to finding the roots of polynomials having degree $B-1$. We search for these roots on every interval defined by the intersection of the definition domains of K and $\overline{V}_n$. Since the function can present discontinuities, one has to add the boundaries of K and $\overline{V}_n$'s definition intervals. Among the points found, we search for the one that maximizes $f_t(t')$. This finally yields a function having the shape of $K_i(\alpha_i - t) + \overline{V}_n(s,t)$ or $\overline{V}_n(s,t)$ providing the B coefficients of $V_n(s,t)$.

At last, we have a complete method for exact analytical Bellman backups in the case of TMDPs which verify equation 5.12. This method uses the properties of polynomials of degree lower than 4 and of discrete distributions.

One can remark that the exact method can apply to the case of generalized degree B for the reward model, as long as "$A+C=-1$". Indeed, Sturm's method or a standard Newton-Raphson method allows us to approximate the intersections of polynomials in equation 5.2. As long as $A+C=-1$, the overall degree of V_n remains stable throughout the iterations. This is the first approximation made by the approximate *TMDP$_{poly}$* algorithm.

6.3 Prioritized sweeping

Even though the previous calculation provides the basis for an exact resolution, it might still perform a lot of unwanted calculation for the optimization of TMDP policies. The standard Value Iteration algorithm is a synchronous method which always builds a new value function with respect to the previous one. [Bertsekas and Tsitsiklis, 1996] discuss the possibility of performing the individual Bellman backups per state asynchronously, *ie.* to use the latest state evaluations found so far, during the current iteration, to update the state at hand. This idea is called *Asynchronous Value Iteration*, it was first introduced to allow parallel computation in Value Iteration and was later exploited to improve convergence speed.

But even Asynchronous Value Iteration can take time to converge if the states are updated in an inappropriate order. However, if one can find a good ordering of Bellman backups per state, then the value function might converge quite quickly. This intuition draws on the general result of Asynchronous Value Iteration, stating that ([Bertsekas and Tsitsiklis, 1996]):

> As long as every state is chosen for Bellman backups infinitely often, the overall value function converges to V^*.

This means we can update the states in the order we want: as long as we visit them infinitely often when the number of iterations tends to $+\infty$, we are guaranteed to converge

to V^*. The simplest version of asynchronous value iteration is the Gauss-Seidel method which uses the most up-to-date value function for Bellman backups but does not have a state selection strategy.

Prioritized sweeping was introduced in [Moore and Atkeson, 1993] and in [Peng and Williams, 1993] based on the following idea. During the first iteration of Value Iteration, if only one state in the state space provides a reward to the agent, then all Bellman backups in states that are not direct parents of this "goal" state will leave the value function unchanged and are unnecessary calculations. On the other hand, if the states in which we perform Bellman backups are taken in an order which moves away from the goal state, convergence of the value function will be much faster. Intuitively, this ordering operation corresponds to sweeping through states in a prioritized manner.

Unfortunately, most problems do not have unique goal states. The reward model provides a richer description than goal states alone by stating that some states with some actions provide positive or negative rewards. The strength of MDPs is to compute the best compromise in terms of expected reward. Therefore, one needs to generalize this idea of "giving the priority to certain states for Bellman backups" to the general case where no specific goals can be defined. This is the basic idea of *prioritized sweeping*.

We summarize the *prioritized sweeping* algorithm as presented by [Moore and Atkeson, 1993] in algorithm 6.3. This algorithm maintains a list of states sorted by priority score. The score of a state is directly determined by the previous iterations. Suppose state s' is updated and its value function varies by a quantity $\Delta V(s') = |V_{new}(s') - V_{old}(s')|$. Then all Q-values of transitions reaching s' with probability $P(s'|s,a)$ will be affected by this change and the order of magnitude of their own change will be in $Prio(s,a) = P(s'|s,a)\Delta V(s')$. Whenever a Q-value $Q(s,a)$ receives a $Prio(s,a)$ of more than a certain ϵ, the algorithm checks whether s's priority in the queue of states to update is higher than $Prio(s)$. If not, the priority of s is promoted to $Prio(s,a)$ and the algorithm picks the next state in the queue in order to update its value. This process is repeated as long as computation is allowed.

Algorithm 6.3: Prioritized Sweeping

Promote s_{init} to the top of the priority queue
while priority queue not empty **do**
  Remove the top state from the priority queue. Call it s'. Set $Prio(s') = 0$.
  Update $V(s') = \max_{a \in A} \left\{ r(s',a') + \gamma \sum_{s'' \in S} P(s''|s',a')V(s'') \right\}$
  Calculate Bellman error $\Delta V(s') = |V(s') - V_{old}(s')|$
  foreach $(s,a) \in predecessors(s')$ **do**
    $Prio(s,a) = P(s'|s,a)\Delta V(s')$
    if $Prio(s,a) > \epsilon$ and $Prio(s,a) > Prio(s)$ **then**
      Insert s in the priority queue with $Prio(s) = Prio(s,a)$

[Andre et al., 1998] and [Dearden, 2001] generalize this method to compact model representations as Dynamic Bayesian Networks ([Dean and Kanazawa, 1990]).

Concerning algorithm 6.3, [Moore and Atkeson, 1993] specify that prioritized sweeping is a heuristic algorithm and provide experimental arguments proving convergence and efficiency of the algorithm. Graphically, one can check that prioritized sweeping will focus on states

that need updates in order to let them converge first, before moving on to their predecessors. Prioritized Sweeping can eventually reach a given state s provided that the s_{init} used for the algorithm is reachable from s. One can note that, on top of the heuristic sweeping method, Prioritized Sweeping can make use of a carefully chosen heuristic for the initial $V(s)$. [Moore and Atkeson, 1993] use an optimistic heuristic adapted from [Kaelbling, 1990].

The version of prioritized sweeping we implemented is directly inspired from algorithm 6.3. We introduced the following modifications and improvements in order to adapt to the framework of TMDPs:

- **Bellman backups.** Since we basically want to perform Bellman backups in the discrete states (for all possible times in $[0, T]$) by applying equations 5.1 to 5.4, we need to avoid unnecessary calculations in the backup phase itself. In algorithm 6.3, the backup in state s' was only a matter of sums and multiplications of real numbers. With TMDPs, as illustrated at the beginning of this chapter, it implies polynomial convolution, root finding, multiplication, etc., hence we wish to do as little of these operations as possible. For this purpose, we slightly modify the backup phase by noting that, in algorithm 6.3, when we calculate $Prio(s, a)$, we actually calculate $Q(s, a) - Q_{old}(s, a)$. So the next Bellman backup using (s, a) actually performs this same calculation of $Q(s, a)$ a second time. Based on this remark, we decide to automatically update all Q-values for the predecessors of s' whenever $V(s')$ is updated. Then we determine the new priority of all predecessors s and move on. This way, performing a Bellman backup in s is only a matter of solving equations 5.1 and 5.2 without recalculating the $Q(s, t, a)$ — these Q functions were automatically updated *after* previous Bellman backups because they were needed to calculate the priorities anyway.

- **Priorities calculation.** Since we calculate $V(s, t)$ value functions instead of $V(s)$, we need to adapt the way $Prio(s, a)$ is calculated. Instead of using $\Delta V(s', t) = V(s', t) - V_{old}(s', t)$, we directly compute $\Delta Q(s, t, a) = Q(s, t, a) - Q_{old}(s, t, a)$ and set $Prio(s, a) = \|\Delta Q(s, t, a)\|_{t\in[0,T],\infty}$. This way, we avoid calculating an extra convolution (between P_μ and ΔV). This pushes our prioritized sweeping implementation to focus on discrete states for which the temporal part of the value function has not fully converged yet. Using an L_2 norm for instance would yield a different behaviour. An interesting option would also be to use a t-weighted or biased norm[1] such as the quantity $\|w(t) \cdot \Delta Q(s, t, a)\|$ — $w(t)$ being an increasing positive function of time —, therefore encouraging convergence in the states that have both the largest amplitude of variation between updates and the latest variation with respect to t. This might take advantage of causality properties to accelerate convergence.

- **Priority queue initialization.** As always, "there is nothing like a good initialization". In the case of prioritized sweeping, initializing the priority queue corresponds to inserting prior knowledge on the states that will yield the largest Bellman error during the first sweep. In order to initialize the algorithm, we can distinguish two classes of problems:

  - **Unstationary Stochastic Shortest Path (USSP) problems.** This problem class presents the important feature of having absorbing discrete states which correspond to the goals. Solving a USSP corresponds to finding the cost minimizing strategy from any state to a goal. For these problems, a good initialization of the priority queue is provided by performing a Bellman backup in all parent transitions of the goal states.

[1] Actually, this quantity might not be a norm anymore.

  - **General TMDPs.** This is the general class of problems without any defined goal state. In this case a simple value iteration through the whole state space might provide a good set of priorities for the states. One can also perform this iteration approximately by using $\|V(s,t)\|_{\infty,t\in[0,T]}$ instead of $V(s,t)$. It is however important to note that if rewards are distributed all over the discrete state space, then convergence will be hard to accelerate anyway since all priorities will be almost equivalent. This is the worse case for prioritized sweeping, independently from the TMDP formalism. In other words, prioritized sweeping propagates the "reward information" through the state space by focusing the propagation on crucial states. If the rewards are distributed in the discrete state space, then the information needs to be propagated from all reward-providing states and to all other states with similar probabilities and the speed-up due to prioritized sweeping is lost due to the problem's distributed structure.

- **Value function initialization.** Since TMDPs are defined with a total reward criterion, the heuristic of [Moore and Atkeson, 1993] is not usable "as is" because it requires $\gamma < 1$. One option is to adapt it by using an optimistic heuristic equal to the maximum sum of rewards available in the problem at hand (or an upper bound) for any transition which has never been tried. This sum must be finite, else it means the total reward criterion does not exist. It is generally finite because all states $(s, t = T)$ are either absorbing states or yield a null long-term reward (see the definition of the pseudo-horizon in section 2.3 and the discussion about convergence of the total reward criterion in section 4.5.2). As identified by [Sutton and Barto, 1998], defining such a heuristic corresponds to a "planning to explore" behaviour caused by the optimism of the heuristic.

- **Avoiding premature stopping.** Suppose that during the very first update of $V(s,t)$ in state s the initial guess $V_{old}(s,t)$ due to the heuristic is rather close to the updated $V(s,t)$. This can lead to $\Delta V(s) < \epsilon$ and the consequence is that all s's parent states might never enter the queue while the initial guess might be completely erroneous for them. This "good estimation bottleneck" is due to the fact that the initial heuristic is not a value function, *ie.* is not the solution of a $V = L^{\pi}V$ equation. An easy way of avoiding such problems is to perform a Bellman backup in every state of the problem whenever the priority queue becomes empty. This unprioritized sweeping through the state space yields a new full set of priorities, thus guaranteeing that all states are visited at least once and restarting the algorithm in the case of "good estimation bottlenecks". Another option is to use an easily computable initial value function really corresponding to a default policy, however, this option looses the advantage of defining a heuristic.

One can note that prioritized sweeping is an approximate value iteration scheme with the approximation error on the value function being bounded by ϵ. We can let the algorithm tend to the exact value iteration behaviour by decreasing the ϵ parameter as the number of iterations increase. This is however not always necessary since the optimal policy can be found with an inexact value function. The prioritized sweeping algorithm applied to TMDPs is presented on algorithm 6.4.

Algorithm 6.4 uses the BellmanUpdate() and BellmanBackup() routines which respectively compute the results of equations 5.4-5.3 and 5.2-5.1. The UnprioritizedVI() routine performs a standard unprioritized value iteration pass and updates both the V and Q functions and the priority queue. One can also note that we do not mention outcomes anymore and integrate them into the transitions for presentation clarity. Hence, updating a transition means updating first all its outcomes' U-functions before updating $Q(s,t,a)$ and calculating $Prio(s,a)$.

Algorithm 6.4: Prioritized Sweeping for TMDPs

```
Init: V ← h, priority_queue ← UnprioritizedVI(), continue = true.
while continue = true do
    while priority_queue ≠ ∅ do
        Remove the top state from priority_queue. Call it s'
        V(s', t).BellmanBackup()                    /* equations 5.2 and 5.1 */
        foreach (s, a) ∈ predecessors(s') do
            Q(s, t, a).BellmanUpdate()              /* equations 5.4 and 5.3 */
            Prio(s, a) = ‖Q(s, t, a) − Q_old(s, t, a)‖_{t∈[0,T],∞}
            if Prio(s, a) > ε and Prio(s, a) > Prio(s) then
                Insert s in priority_queue with Prio(s) = Prio(s, a)
    priority_queue ← UnprioritizedVI()
    if max_priority(priority_queue) < ε then
        Either take a smaller ε or set continue = false.
```

Finally, the last pass of value iteration used to avoid premature stopping is also used to compute the final policy. If *continue* is set to $false$, then the output policy is the one calculated during this last unprioritized value iteration pass.

Similarly to the calculation reduction strategy in the Bellman backups, which avoided calculating the Q-values twice, we can choose to push this idea further if we are ready to accept a little bit of approximation. During the BellmanBackup() routine, two main steps are performed. The first one deals with comparing all the Q-values in order to find the overall $\overline{V}$ function. The second step deals with finding the optimal dawdling time. For the first step, we solve:

$$\overline{V}_n(s,t) = \max_{a \in A} Q_n(s,t,a) \tag{6.2}$$

And thus we have:

$$\forall (s,a) \in S \times A,\ \overline{V}_n(s,t) \geq Q_n(s,t,a)$$

If we remember the $Prio(s,a)$ calculated during the last update of $Q(s,a)$, then we can decide to consider that the Q-values with priority less than ϵ have not changed significantly and thus write:

$$\forall (s,a) \in S \times A \text{ such that } Prio(s,a) < \epsilon,\ \overline{V}_n(s,t) \geq Q_{n+1}(s,t,a)$$

Consequently, when solving equation 6.2, instead of comparing $|A_s|$ Q-values alltogether, we only compare a set of $p+1$ functions where p is the number of transitions, starting in s, that received a priority higher than ϵ. Namely, this set of functions contains the latest $\overline{V}(s,t)$ and the Q-values which have a priority greater than ϵ. This implies accepting to have an approximation of ϵ on the Q values, hence on the value function, and finally, to obtain an ϵ-optimal policy.

6.4 Approximate TMDP optimization

The last section presented the general Prioritized Sweeping algorithm we introduced in order to solve TMDPs when the exact resolution of the optimality equations is possible. However,

the general case of piecewise polynomial representations does not allow for such a resolution because equation 5.12 is usually not verified. Moreover, in practice, even the exact resolution scheme suffers from a very quick multiplication of $V(s,t)$'s number of separate definition intervals. We introduce an intermediate approximation step in order to project the result of a Bellman backup back into a space of polynomials which have a bounded degree and a limited number of definition intervals. The challenge is to design an approximation operator which guarantees an L_∞-bounded approximation error so that we can use the results of Approximate Value Iteration. In this section, we first recall some results of Approximate Value Iteration (AVI) which help study the convergence and ϵ-optimality of our algorithm, then we present the alternatives we tested for the approximation operator.

6.4.1 Approximate Value Iteration

Let L be the standard dynamic programming operator and $\mathcal{F}(S,\mathbb{R})$ be the set of functions from S to $\mathbb{R}$. Let Ap be an approximation operator projecting a function from $\mathcal{F}(S,\mathbb{R})$ into a subspace of $\mathcal{F}(S,\mathbb{R})$. Approximate Value Iteration (AVI) is the algorithm which results from the successive application of the $\tilde{L} = Ap \circ L$ operator to an initial value function. Some general results are available for AVI, we summarize them below and prove some of them.

Let us write U_n the sequence of value functions obtained with AVI and V_n the sequence of value functions one would obtain with standard Value Iteration. We also write V^* the MDP's optimal value function; V^* is the limit of the $(V_n)_{n\in\mathbb{N}}$ sequence. Finally, we also have $V_0 = U_0$. One has:

$$U_n = \tilde{L}^n(U_0) = (Ap \circ L)^n(U_0) \tag{6.3}$$

We suppose that one can bound the approximation error in supremum norm as in equation 6.4. Calculating the supremum norm of a piecewise polynomial function over an given interval is a rather easy calculation so guaranteeing this bound won't be a problem for our algorithms.

$$\exists \epsilon \in \mathbb{R}^+ \ / \ \forall f \in \mathcal{F}(S,\mathbb{R}),\ \|Ap(f) - f\|_\infty \leq \epsilon \tag{6.4}$$

The first important result about AVI is that:

In general, Approximate Value Iteration does not converge.

However, we can prove that the value function tends to reach the neighbourhood of V^*.

[Bertsekas and Tsitsiklis, 1996] prove that as the number of iterations tend to $+\infty$, the U_n functions belong to the neighbourhood of V^*:

$$V^* - \frac{\epsilon}{1-\gamma} \leq \liminf_{n\to\infty} U_n \leq \limsup_{n\to\infty} U_n \leq V^* + \frac{\epsilon}{1-\gamma} \tag{6.5}$$

Proof. Because of equation 6.4, one has:

$$LU_0 - \epsilon \leq U_1 \leq LU_0 + \epsilon$$

So we can write that:

$$L(LU_0 - \epsilon) \leq LU_1 \leq L(LU_0 + \epsilon)$$
$$L^2U_0 - \gamma\epsilon \leq LU_1 \leq L^2U_0 + \gamma\epsilon$$

And, with equation 6.4 again:

$$LU_1 - \epsilon \leq U_2 \leq LU_1 + \epsilon$$

So we have:

$$L^2U_0 - (1+\gamma)\epsilon \leq U_2 \leq L^2U_0 + (1+\gamma)\epsilon$$

By induction, we obtain:

$$L^nU_0 - (1+\gamma+\ldots+\gamma^{n-1})\epsilon \leq U_n \leq L^nU_0 + (1+\gamma+\ldots+\gamma^{n-1})\epsilon$$

Since there is no convergence guarantee on the $(U_n)_{n\in\mathbb{N}}$ sequence one cannot write its limits, but we can still take its lim inf and lim sup, thus:

$$V^* - \frac{\epsilon}{1-\gamma} \leq \liminf_{n\to\infty} U_n \leq \limsup_{n\to\infty} U_n \leq V^* + \frac{\epsilon}{1-\gamma}$$

□

If our approximation is such that $\|Ap(f)\|_\infty \leq \|f\|_\infty$, then the previous equation turns to:

$$V^* - \frac{\epsilon}{1-\gamma} \leq \liminf_{n\to\infty} U_n \leq \limsup_{n\to\infty} U_n \leq V^* \tag{6.6}$$

Then, the interesting part is to evaluate the performance of a policy obtained with AVI.

If π_n is the greedy policy with respect to the value function U_n, then its value function V^{π_n} obeys equation 6.7.

$$\|V^* - V^{\pi_n}\|_\infty \leq \frac{2\gamma}{1-\gamma}\|V^* - U_n\|_\infty \tag{6.7}$$

And so:

$$\limsup_{n\to\infty} \|V^* - V^{\pi_n}\|_\infty \leq \frac{2\gamma\epsilon}{(1-\gamma)^2} \tag{6.8}$$

Proof. We have $LV^* = V^*$ (by definition of L and V^*), $L^{\pi_n}V^{\pi_n} = V^{\pi_n}$ (by definition of L^{π_n} and V^{π_n}) and $L^{\pi_n}U_n = LU_n$ (because π_n is greedy with respect to U_n). Equation 6.7 is a simple consequence of the inequality:

$$\begin{aligned}
\|V^* - V^{\pi_n}\|_\infty &= \|LV^* - L^{\pi_n}U_n + L^{\pi_n}U_n - L^{\pi_n}V^* + L^{\pi_n}V^* - L^{\pi_n}V^{\pi_n}\|_\infty \\
&\leq \|LV^* - L^{\pi_n}U_n\|_\infty + \|L^{\pi_n}U_n - L^{\pi_n}V^*\|_\infty + \|L^{\pi_n}V^* - L^{\pi_n}V^{\pi_n}\|_\infty \\
&\leq \gamma\|V^* - U_n\|_\infty + \gamma\|V^* - U_n\|_\infty + \gamma\|V^* - V^{\pi_n}\|_\infty
\end{aligned}$$

And so:

$$\|V^* - V^{\pi_n}\|_\infty \leq \frac{2\gamma}{1-\gamma}\|V^* - U_n\|_\infty$$

The second inequality comes from equation 6.4. □

It is also possible to derive incremental bounds on the Bellman residual by using results from [Williams and Baird, 1993]:

If π_n is the greedy policy with respect to the value function U_n, then its value function V^{π_n} obeys equation 6.9.

$$\|V^* - V^{\pi_n}\|_\infty \leq \frac{2}{1-\gamma}\|LU_n - U_n\|_\infty \tag{6.9}$$

For the case of piecewise polynomial approximation, we can easily calculate the L_∞ bounds by performing analytical calculations. However, in most approximation schemes, the approximate value function is usually obtained by minimizing an L_p-norm criterion and the previous results do not hold anymore. [Munos, 2007] extends the previous results to the case of weighted L_p-norms.

Unfortunately, TMDPs are defined with a total reward criterion so the theoretical bounds provided above cannot be used. Nevertheless, [Bertsekas and Tsitsiklis, 1996] argue that for stochastic shortest path problems and good approximation architectures, the final policy obtained by AVI is close to the optimal strategy because of the approximation's good quality. They show it is relatively easy to adapt the proof of equation 6.5 to the case of a finite number of steps, hence illustrating this intuition of convergence and ϵ-optimality.

6.4.2 Polynomial degree reduction and interval number minimization

The approximation scheme we design aims at keeping the value function in the same function space. Since equation 5.4 implies calculating convolutions of S_μ with r and with V_n, it makes sense to try to keep the degree of V_n equal to the one of r. In other words, it makes sense to use $\mathcal{P}_B$ as the projection space for our Ap operator.

On the other hand, since we have the — approximate — tools to compute roots and convolutions for polynomials of degree higher than 5, our main goal in using the Ap approximator is to keep the polynomials' degree low enough in order to avoid dealing with very high order polynomials. This means we actually don't need to perform this projection at every Bellman backup. As long as we consider it acceptable to let the polynomials' degree increase, we can perform convolutions and root searching for polynomials of degree higher than B. When the degree of our polynomials reach a certain threshold, then we can decide to use our approximation operator in order to project these polynomials back into a lower degree $\mathcal{P}_M$ space.

The intuitive idea behind the lazy approach presented in the last paragraph relies on the fact that high order polynomials can describe functions with lots of inflexions and variations while lesser degree polynomials do not have such an expressive power. Hence, when we reduce the degree of our piecewise polynomial functions, we can expect the number of definition intervals to increase. This trade-off between polynomial degree and number of definition intervals seems unavoidable and the previous method for lazy projection aims at providing a flexible approach to the approximate resolution.

Finally, the approximation problem can be stated as follows. For all function $f \in \mathcal{P}_K$, we search for a function $Ap(f) = \tilde{f}, \tilde{f} \in \mathcal{P}_M$ such that $\|f - \tilde{f}\|_\infty \leq \epsilon$. This constraint defines a set of candidate functions. Among these functions, we can define a criterion to optimize. Optimizing the $\|f - \tilde{f}\|_2$ quantity seems to be a bad idea since an obvious solution to the problem stated in equations 6.10 is found with a piecewise polynomial function having an infinite number of definition intervals.

$$\begin{aligned} &\min_{f \in \mathcal{P}_M} \|f - Ap(f)\|_2 \\ &\text{with } \|f - Ap(f)\|_\infty \leq \epsilon \end{aligned} \tag{6.10}$$

We can chose to minimise the number of intervals, considering that ϵ is small enough to insure that our approximation fits the original function. This defines the optimization problem 6.11.

$$\begin{aligned} &\min_{f \in \mathcal{P}_M} \{\text{intervals number in } f\} \\ &\text{with } \|f - Ap(f)\|_\infty \leq \epsilon \end{aligned} \tag{6.11}$$

The solution to problem 6.11 need not be unique so we might want to use a hybrid criterion in the end. We write $\mathcal{P}_{M,q}$ the subset of $\mathcal{P}_M$ containing elements having exactly q definition intervals. This yields problem 6.12.

$$\begin{aligned} &\min_{f \in \mathcal{P}_{M,q}} \|f - Ap(f)\|_2 \\ &\text{with } q = \arg\min_{p \in \mathbb{N}} \{\mathcal{P}_{M,p} \ / \ \mathcal{P}_{M,p} \cap \mathcal{S} \neq \emptyset\} \\ &\text{and } \mathcal{S} = \{f \in \mathcal{P}_M \ / \ \|f - Ap(f)\|_\infty \leq \epsilon\} \end{aligned} \tag{6.12}$$

While this last formulation seems to be an acceptable expression of our approximator's requirements, computing its optimal solution can require a lot of calculation. The only crucial rule to respect is the constraint $\|f - Ap(f)\|_\infty \leq \epsilon$. We introduce the suboptimal approximation method of algorithm 6.5 which returns a piecewise polynomial function belonging to $\mathcal{P}_{M,q'}$, with $q' \geq q$.

Algorithm 6.5: Polynomial approximation

Main loop:
input: p_{in} /* the pwp to approximate */
input: M /* the approximation's degree */
input: ϵ /* the tolerance on the L_∞ error bound */
$p_{out} = p_{in}$
$\mathcal{I} = p_{out}$.intervals() /* The set of intervals of p */
for $I \in \mathcal{I}$ **do** /* Approximating the function */
  replace $f = p_{out}$.polynomial(I) by $f' =$approx(f, M, I)
return p_{out}

approx(f, M, I):
$\mathcal{B} = \{I$.lower(), I.upper()$\}$ /* The new set of bounds inside I */
$e = \epsilon + 1$ /* the error term */
while $e > \epsilon$ **do**
  $f' =$piecewise_interpolation$(\mathcal{B}, M, f)$
  $e = \|f' - f\|_{I,\infty}$
  if $e > \epsilon$ **then** /* Check the constraint */
    $x_{worse} = \underset{x \in I}{argsup} \, |f'(x) - f(x)|$
    $\mathcal{B}$.insert(x_{worse})
return f'

Notations:
pwp: piecewise polynomial function
$\|g\|_{I,\infty} = \sup_{x \in I} |g(x)|$
piecewise_interpolation$(\mathcal{B}, M, f)$ computes the pwp interpolation of f, in $\mathcal{P}_M$, on the intervals defined by $\mathcal{B}$

This algorithm computes a piecewise polynomial approximation p_{out} of p_{in}, which has degree M and verifies $\|p_{in} - p_{out}\|_\infty \leq \epsilon$. The computation is performed by incremental cutting of each definition interval in order to simplify the portion to approximate. This way,

all local approximations eventually become bounded by ϵ. The number of intervals in p_{out} is not minimal but this algorithm remains a good compromise in terms of calculation time.

The approx method of algorithm 6.5 is illustrated on figure 6.3.

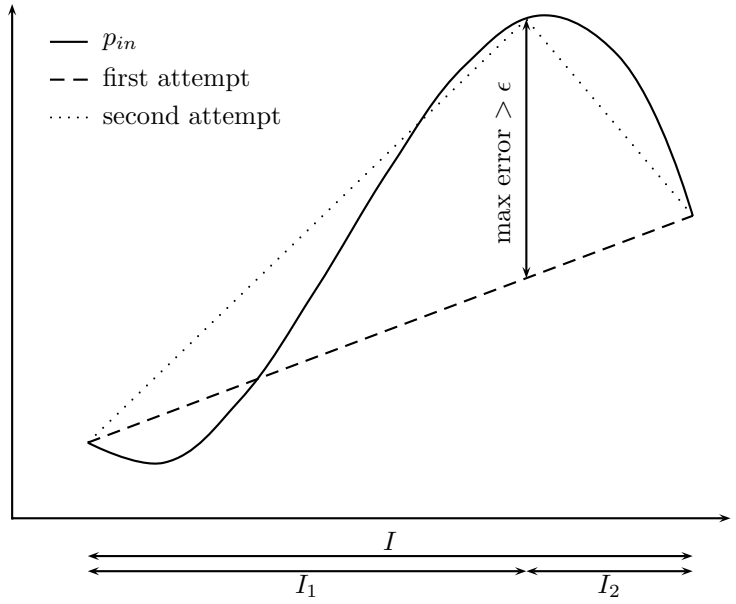


Figure 6.3: Illustrating algorithm 6.5

Three improvements to the approx method are immediately possible.

1. It is possible to incrementally check that the constraint is not violated and to refine the discretization in several points without having to go through several iterations of the "while" loop.

2. Depending on the value of the degree M, some good heuristics can be used for finding the x_{worse} points. For example, if we use polynomials of degree 3 (cubic splines for instance) then we know that these polynomials will provide good interpolation capabilities on intervals where p_{in} only has one inflexion. Finding the inflexions of p_{in} already yields a good partitioning of I for degree 3 interpolations.

3. If interpolation itself, given the set of bounds, is costless (for cubic or linear interpolation for instance), then we can further minimize the number of definition intervals by popping bounds out of the $\mathcal{B}$ set whenever we find a new cutting point which is before the bounds in $\mathcal{B}$.

4. Finally, it is possible to include a last parsing of the final interval cutting in order to merge any two posterior definition intervals over which the polynomials are close. This merging should still respect the $\|p_{in} - p_{out}\|_\infty \leq \epsilon$ constraint.

This last point appears crucial from a practical point of view. Indeed, experiments have shown that as the number of iterations grows, the number of definition intervals increases dramatically. While this might be necessary to describe the subtle variations of the value functions, it sometimes also results in very small successive intervals where the function is quasi-constant with very close successive values. On top of being unnecessarily detailed knowledge, this aspect handicaps the optimization efficiency, so it seems important for our

Algorithm 6.6: $TMDP_{poly}$ polynomial approximation

input: p_{in} /* the pwp to approximate */
input: M /* the approximation's degree */
input: ϵ /* the tolerance on the L_∞ error bound */
input: $[l, u]$ /* the approximation interval */
$p_{out} = p_{in}$
$t_0 = l$
$continue = true$
f, f', g, g_{temp} are polynomials
while $continue = true$ **do**
  $I =$ interval to which t_0 belongs
  $f = p_{out}$.polynomial(I)
  Refinement phase: /* refining the bounds to fit the function */
  if f.degree() $> M$ **then**
    Erase the interval I in p_{out}.
    $refine = true$
    $t_{up} = I$.upper()
    while $refine = true$ **do**
      $f' =$interpolation(f, M, t_0, t_{up})
      if $\|f - f'\|_{[t_0,t_{up}]} > \epsilon$ **then**
        $t_{worse} = \underset{t \in I}{argsup} \, |f'(t) - f(t)|$
        $t_{up} = t_{worse}$
      else if $\|f - f'\|_{[t_0,t_{up}]} \leq \epsilon$ and $t_{up} \neq I$.upper() **then**
        Set p_{out} to f' on $[t_0, t_{up}[$.
        $t_0 = t_{up}$
        $t_{up} = I$.upper()
      else
        $refine = false$
  Simplification phase: /* swallowing successive intervals into one */
  $I =$ interval to which t_0 belongs
  $I_{next} = I$.next_interval()
  $t_{temp} = I_{next}$.upper()
  $g = p_{out}$.polynomial(I)
  $g_{temp}(t) =$interpolation$(p_{out}, M, t_0, t_{temp})$
  while $\|p_{out} - g_{temp}\|_{[t_0,t_{up}],\infty} \leq \epsilon$ **do**
    $g = g_{temp}$
    $t_{up} = t_{temp}$
    $I_{next} = I_{next}$.next_interval()
    $t_{up} = I_{next}$.upper()
    $g_{temp}(t) =$interpolation$(p_{out}, M, t_0, t_{up})$
  Replace all polynomials of p_{out} over $[t_0, t_{up}[$ by g.
  Stopping condition:
  $t_0 = t_{up}$
  if $t_0 \geq u$ **then** $continue = false$

approximation method to be able to detect and merge all these small intervals together while still respecting the $\|p_{in} - p_{out}\|_\infty \leq \epsilon$ constraint. Once again, polynomial interpolation is particularly suited and efficient for this kind of approximation.

The final approximation method we used is presented in detail in algorithm 6.6. One can easily verify that this method provides an L_∞ bounded approximation error. In practice, this method proved to be efficient in terms of calculation time and interval number reduction.

6.5 The $TMDP_{poly}$ algorithm

Finally, we have introduced all the bricks to build the approximate $TMDP_{poly}$ algorithm. Let us summarize them here.

- **Exact calculation of Q_{n+1}.** Using the exact convolution method presented in appendix A we perform analytical computation of U_{n+1} and Q_{n+1} from V_n, using equations 5.3 and 5.4.
- **Polynomial degree reduction.** When the polynomial's degree become larger than a certain threshold we apply algorithm 6.6 to return in $\mathcal{P}_B$ space with as little definition intervals as possible. This degree reduction guarantees that the approximation lies within an ϵ bound of the original function. This bound is calculated with respect to the supremum norm, thus guaranteeing the AVI properties presented in section 6.4.
- **Approximate calculation of V_{n+1}.** Using equations 5.1 and 5.2 we compute V_{n+1} from Q_{n+1}. This calculation is approximate because of the root finding phase when searching for the maximum over all Q_{n+1}.
- **Prioritized sweeping.** We reuse the same prioritized sweeping method as was presented in algorithm 6.4 but now the BellmanBackup and BellmanUpdate routines make use of the three previous points.

The algorithm presentation itself does not differ from algorithm 6.4 so we refer the reader to previous paragraphs for details.

7

Implementation and experimental evaluation of the $TMDP_{poly}$ algorithm

This chapter presents two instances of problems solved using the $TMDP_{poly}$ planner implemented from the $TMDP_{poly}$ algorithm. The first problem is an adaptation from the standard Mars rover benchmark, with action result uncertainty and a continuous time resource, as presented in [Bresina et al., 2002]. We derive several variants on the same problem. The second illustrates a different interpretation of TMDPs: it is a surveillance mission planning problem where the *wait* action is no longer a passive action, on the contrary, it is the only action providing rewards. This second example generalizes TMDPs to the case of other continuous actions than *wait* and presents a different point of view on the possible applications of TMDPs.

7.1 Implementation choices

The $TMDP_{poly}$ planner has been implemented in C++ as a general library of functions permitting the definition, modification and optimization of TMDP problems. Details on the $TMDP_{poly}$ library's current implementation are available at `http://emmanuel.rachelson.free.fr/en/software/`.

Three layers of functionalities have been developed independently to build the $TMDP_{poly}$ planner:

- First, the *POLYTOOLS* library has been developed. Its goal is to allow the definition of polynomial and piecewise polynomial functions and to provide built-in methods for all operations on these polynomials. These operations range from simple addition or multiplication to complex operations such as:
  - Root finding (exact and approximate methods)
  - Variations analysis
  - Convolution of piecewise polynomial functions
  - The approximation scheme presented at the end of the previous chapter

  The *POLYTOOLS* library provides a simple and rich interface which manages the low-level memory allocation and the complete operations for piecewise polynomial functions calculus. The algorithmic part of *POLYTOOLS* is presented in appendix A and the

latest implementation itself is available at `http://emmanuel.rachelson.free.fr/en/software`.

- Then, using the *POLYTOOLS* layer, the $TMDP_{poly}$ library itself defines an interface to declare generalized TMDP problems and provides optimization functions that implement both Value Iteration and the $TMDP_{poly}$ algorithm to solve them. In order to allow the use of discrete distributions as well as piecewise polynomial ones, the $TMDP_{poly}$ library also defines a "discrete pdf" class with compatibility operators for operations with polynomials (for convolution for instance).
- Finally, for the case of the gridworld UAV patrol problem example, the provided graphical interface encapsulates some functions of the $TMDP_{poly}$ library to build a visualization interface for the optimization operations and result.

Compared to the algorithms presented in the previous chapter, the $TMDP_{poly}$ library provides an exact implementation with the following choices:

- Approximation frequency. Since the approximation algorithm includes the interval reduction method, it is applied to $V(s)$ every time a state is updated.
- Approximation degree. In order to use the simplicity of linear regression, the approximation method always projects the value function onto the space of piecewise linear functions. It is a deliberate choice made for simplicity; using cubic splines (or even other splines) might provide better interpolation capacities with even less definition intervals but it has not been tested yet.

The following sections present the results obtained on different examples. All experiments were ran on the same computer. This computer's configuration is briefly given in the next table:

Processor	AMD Athlon 3200+ (single core, 1.8 GHz)
Memory	1019 MB
OS	GNU/Linux Ubuntu version 8.04
C/C++ compiler	gcc 4.2.4

7.2 Simple examples and results with the $TMDP_{poly}$ planner

7.2.1 Two simple test examples: the three states problem

The goal of these two problems is only to illustrate how the basic functions of $TMDP_{poly}$ work and to introduce the metrics we define for evaluating the $TMDP_{poly}$ results.

Problem 1 is a loopless three states problem illustrated on figure 7.1. State s_3 is an absorbing state and each action has a single outcome, so actions and outcomes can be directly identified ($a_i \leftrightarrow \mu_i$) and actions considered deterministic with respect to the discrete part of the state. This is expressed by:

$$\forall i \text{ such that } a_i \text{ is applicable in } s, L(\mu_i|s,t,a_i) = 1$$

All outcomes have parameter $T_\mu = REL$, and the duration distributions are defined as:

- $P_{\mu_1}(\tau) = \delta_1(\tau)$

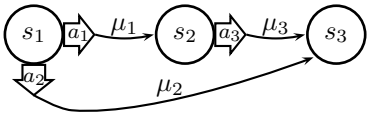


Figure 7.1: 3 states problem - 1st version

- $P_{\mu_2}(\tau) = \delta_3(\tau)$
- $P_{\mu_3}(\tau) = \delta_1(\tau)$

So all outcomes have deterministic durations. Finally, the reward models are given by:

$$r_t(\mu_2, t) = 2 \cdot 1_{[45,75]}(t)$$
$$r_{t'}(\mu_3, t') = 1$$

All other rewards are equal to zero.

Finally this is a very simple deterministic problem with time-dependent rewards. We can set the pseudo-horizon to 100 for example but it is not a crucial variable since we have an absorbing terminal state (thus making the total reward criterion converge anyway).

Problem 2 is similar to problem 1 but the loopless structure is broken by a fourth action that allows returning to s_1 from s_3 as illustrated on figure 7.2.

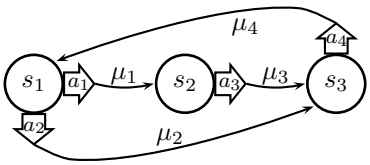


Figure 7.2: 3 states problem - 2nd version

The duration of μ_4 is given by: $P_{\mu_4}(\tau) = \delta_{30}(\tau)$. So μ_4 is a rather long transition compared to the other outcomes.

The reward models are given by:

$$r_t(\mu_2, t) = 4 \cdot 1_{[50,75]}(t)$$
$$r_{t'}(\mu_3, t') = 1_{[0,100]}(t')$$
$$r_t(\mu_4, t) = -2 \cdot 1_{[0,100]}(t)$$

So there is a penalty in undertaking action a_4 (corresponding to outcome μ_4) which can only be compensated by the rewards of the other outcomes if they are available.

7.2.2 Optimisation results

For the first version of the three states problem, when the priority list is initialized with s_3, the algorithm converges in 4 iterations, updating the states in the order: s_3, s_1, s_2, s_1. More specifically, the update priorities were:

$$\begin{aligned} s_3 &: \infty \\ s_1 &: 2 \\ s_2 &: 1 \\ s_1 &: 1 \end{aligned}$$

According to algorithm 6.4, after the first update, the value function of s_3 is set to the constant zero and the parent U and Q-functions are updated. These U functions correspond to μ_2 and μ_3 thus letting the priorities 2 and 1 be assigned to s_1 and s_2 respectively. Then the second update, occurring in s_1, only updates s_1's value function since there are no parent transitions. The highest (and only) priority of the priority queue at this step is then s_2. Its value function update propagates the priority 1 to s_1 which is updated finally. During this final update, it appears that during $[0, 45]$ it is better to wait for the reward of μ_2 than to try μ_1. The final value functions are presented on figure 7.3.

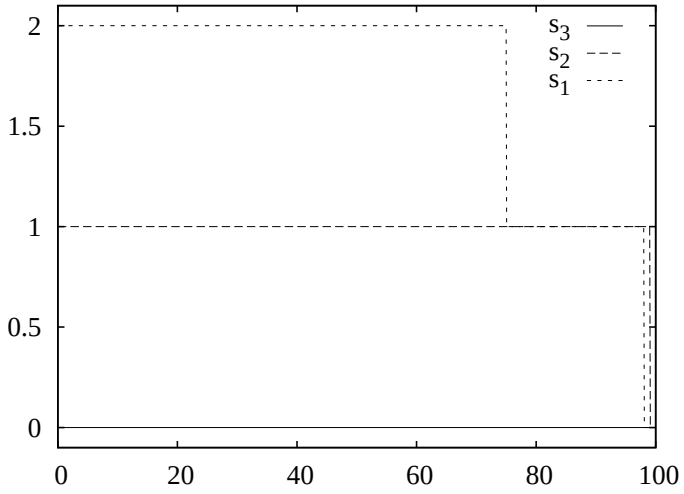


Figure 7.3: Final value functions for the three states problem, first version

The associated policy found is:

$$\begin{aligned} s_1 &: [0; 45] \rightarrow \text{wait} \\ &\quad [45; 75] \rightarrow \text{down} \\ &\quad [75; 100] \rightarrow \text{right} \\ s_2 &: [0; 100] \rightarrow \text{right} \\ s_3 &: [0; 100] \rightarrow \text{wait} \end{aligned}$$

The overall calculation time was smaller than 10^{-2} seconds.

Similarly, the second version of the three states problem converges in 5 iterations when initialized with a null value function and an initial sweeping through the transitions to get the initial priorities. The initial priorities are given by:

$$\begin{aligned} s_1 &: 4 \\ s_3 &: 2 \\ s_2 &: 1 \end{aligned}$$

And the *a posteriori* state ordering followed for optimization was:

$$
\begin{array}{ll}
s_1: & 4 \\
s_3: & 4 \\
s_2: & 2 \\
s_1: & 3 \\
s_3: & 1
\end{array}
$$

This ordering is given in figure 7.4 as an example of what we shall use as a performance profile in the next sections.

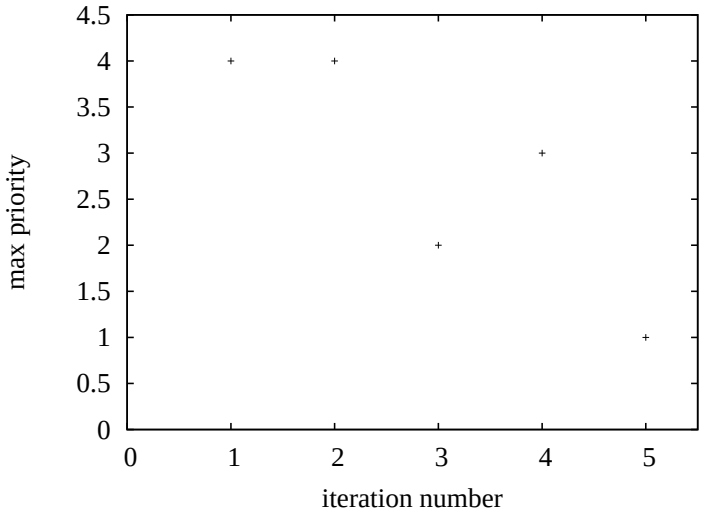


Figure 7.4: Evolution of the maximum priorities for the three states problem, second version

This ordering illustrates the fact that priorities are not necessarily always decreasing. One could expect them to follow a global decreasing trend: since they are related to Bellman's error, they should converge to zero, but this convergence is not monotonous.

If we try to follow the update mechanism to understand how priorities can sometimes increase, we can notice the following sequence of events. For simplicity of notation, we write Q_i the Q function associated with the action triggering outcome μ_i. This is possible since there is only one outcome per action in this toy example. During the first sweep meant to retrieve the initial priorities, s_2 receives priority 1 because Q_3 becomes equal to $1_{[0;99]}(t)$. But as s_3 is updated (second update), Q_3 becomes equal to $3 \cdot 1_{[0;44]}(t) + 1_{[44,99]}(t)$. The $\|\Delta Q_3\|_\infty$ is equal to 2, thus assigning a new priority of 2 to s_2. However, the cumulative variation since the last value function update in s_2, in L_∞ norm, is equal to 3: the algorithm is blind to such a change because Q_3 has increased in two steps. Hence, when we update s_2, its value function becomes equal to Q_3, and Q_1 jumps from zero to $3 \cdot 1_{[0;43]}(t) + 1_{43;98}(t)$, yielding a $\|\Delta Q_1\|_\infty$ of 3 and finally assigning priority 3 to s_3, which corresponds in an increase in the maximum priority of the queue.

Our update mechanism just keeps track of the largest variation of Q in a single update, not of the cumulative variation, thus yielding priorities sometimes smaller than the actual value function change they induce. When the value function is updated, this creates a priority higher than the previous ones. This is the first reason for non-monotonicity of the priorities:

The priority mechanism keeps track of the largest $\|\Delta Q\|_\infty$ in one iteration and not the cumulative $\|\Delta Q\|_\infty$ since the last update. Therefore, priorities sometimes overestimate or underestimate the impact of value functions updates. This induces the non-monotonous behavior of the priorities. It could be compensated by keeping track of the Q functions just after a value function update.

Since priority propagation is a local mechanism, states are often updated soon after receiving their high priorities. Thus, this breaking of monotonicity does not appear often because there are few transition that receive their $\|\Delta Q_1\|_\infty$ in more than one time. More importantly, this also illustrates why the priorities quickly drop again after these value functions updates. This will be particularly visible in the rover and UAV examples.

The final value functions are presented on figure 7.5.

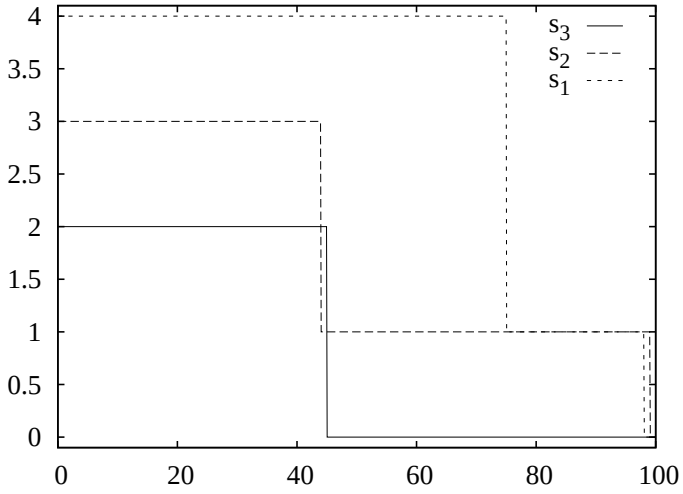


Figure 7.5: Final value functions for the three states problem, second version

The associated policy found is:

$$\begin{array}{ll} s_1: & [0;50] \rightarrow \text{wait} \\ & [50;75] \rightarrow \text{down} \\ & [75;100] \rightarrow \text{right} \\ s_2: & [0;100] \rightarrow \text{right} \\ s_3: & [0;45] \rightarrow \text{up} \\ & [45,100] \text{ wait} \end{array}$$

The overall calculation time was also smaller than 10^{-2} seconds.

If one changes slightly the problem in order to make the reward of μ_1 accessible twice by taking the loop through s_3, then the policy changes accordingly. For example if we write:

$$r_t(\mu_2, t) = 4 \cdot 1_{[30,75]}(t)$$

Then the optimization finishes in 7 iterations, still in a computing time smaller than 10^{-2} seconds. The prioritized sweeping through the state space was:

$$
\begin{aligned}
s_1 &: \ 4 \\
s_3 &: \ 4 \\
s_2 &: \ 2 \\
s_1 &: \ 3 \\
s_3 &: \ 2 \\
s_2 &: \ 2 \\
s_1 &: \ 2
\end{aligned}
$$

The final value function is given on figure 7.6

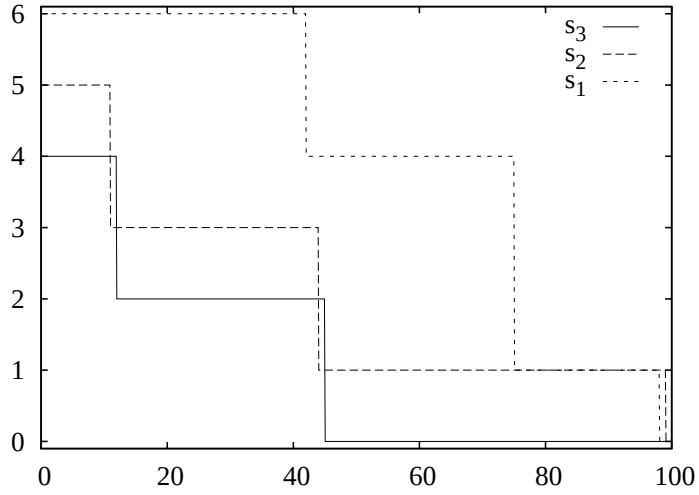


Figure 7.6: Final value functions for the three states problem, second version modified

The associated policy is:

$$
\begin{aligned}
s_1 &: \ [0;30] \rightarrow \text{wait} \\
&\quad\ [30;75] \rightarrow \text{down} \\
&\quad\ [75;100] \rightarrow \text{right} \\
s_2 &: \ [0;100] \rightarrow \text{right} \\
s_3 &: \ [0;45] \rightarrow \text{up} \\
&\quad\ [45,100] \text{ wait}
\end{aligned}
$$

These examples' purpose was only to illustrate the general behaviour of the $TMDP_{poly}$ planner. The next sections analyze its performance on larger problems.

7.2.3 Metrics

The two following sections present the Mars rover and the UAV patrol domain. We use these examples to illustrate the behavior of the $TMDP_{poly}$ planner. More specifically, we evaluate:

- The performance graph: since priorities are related to Bellman error, we use them as an approximate measure of performance of the global policy through the iterations. This measure is only reliable if we are certain of the priority queue's initialization. This is the case since we use the automatic initialization procedure of section 6.3 for the rover case and the list of rewards for the UAV.

- The expected reward: the problems at hand are not supposed to have a starting state in particular. Since plotting all the state value functions might be a little cumbersome and not relevant, we try to underline the final value function obtained for states that seem relevant. In the UAV case, the graphical interface illustrates well the associated policy, even though nothing replaces direct user evaluation by "playing" with the interface.

- Complexity: we plot the evolution of individual state updates duration in order to see how the calculation time evolves as the functions become more complex. However, this metric is implementation and machine dependent. So we also relate it to the number of state updates before the priority queue becomes empty.

7.3 The Mars rover problem

7.3.1 Problem definition

Overview

The rover problem is inspired by and adapted from the International Planning Competition "rover" domain and by the original Mars rover problem statement of [Bresina et al., 2002].

This domain describes the problem of mission planning for a rover over a full day on Mars. The rover's mission is to collect two rock samples from different sites and to take a photo of a distant object. Available actions deal with recharging the batteries, taking the photo, collecting the samples and moving from site to site. One can make the problem more complex by adding possible transmissions with a remote station, on-board analysis actions, memory management, etc. We will keep this first simple description of the problem for our experiments since it seems rich enough to describe an interesting problem.

Previous work on the problem of planning the operations of the Mars rover tackled different aspects of the problem stated in [Bresina et al., 2002]. The complete rover domain, as presented by [Bresina et al., 2002], involves dealing with contingencies, probabilities, continuous variables, continuous time, concurrent actions, etc. [Bresina et al., 2002] lists a number of algorithms, planners and approaches for this domain, highlighting their strengths and weaknesses. Later work by [Mausam and Weld, 2007] addresses the question of dealing with concurrent actions, synchronized on a discretized time, with duration uncertainties. [Feng et al., 2004; Li and Littman, 2005] attacked the problem from the fully continuous point of view, representing value functions as *kd-trees*. HAO* [Benazera et al., 2005] also attacked the rover problem by addressing the question of hybrid state spaces and heuristic search and pruning. While our algorithm is not designed to compete with the previous approaches as a matter of performance, it provides a different alternative which could be combined, for example, with the heuristic approach of HAO*, or with the action elimination scheme of [Mausam and Weld, 2007] for dealing with larger action spaces.

Figure 7.7 illustrates the mission planning problem. The rover can navigate between nodes labeled 1 to 6 which correspond to values of p, the position variable. Each movement action has a certain success or failure probability: these actions can end up in the destination of in the initial position. Similarly, movement durations and energy consumption are uncertain. The labels attached to the edges of the navigation graph correspond to the average travel duration for a successful move along the edge. The filled nodes correspond to sample sites: sample 1 is available at position 5 and sample 2 at position 2. The dark gray areas are obstacles to both navigation and vision while the light gray area is an obstacle to navigation

only. Consequently, the photo can be taken from any of the nodes numbered 3 to 6. However, this picture has different probabilities of being successful depending on the shooting site. The rover has the on-board ability to roughly analyse the image in order to determine whether it is good or not. So whenever the picture is taken, it can result in either a good image or a bad one but there is no notion of ranking among images. Consequently, whenever a good image has been shot, it is kept without further questioning. The preferred shooting site is position 6.

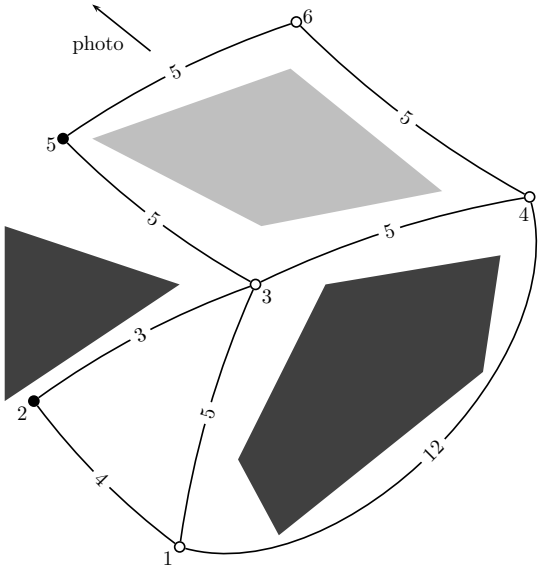


Figure 7.7: Mars rover problem — mission presentation

We consider a day of length 70 time units and we suppose the goal is to finish the mission before nightfall but this constraint is flexible and the mission does not really have to be completed in one day. After 70 time units, night falls and the rover switches to energy saving. We consider $e = 0$ to be the lowest energy level corresponding to surviving during one night. Hence, the mission can be restarted everyday from any state of the problem which implies we are interested in the policy in every possible starting state.

Finally, depending on the time of day, lighting changes which affects the recharge ability of the rover and the photography's success probability.

The state variables we consider are summarized in table 7.1. They yield a hybrid state space containing 1968 discrete states and one continuous variable. It can be interesting to compare with a discrete problem generated using a unit discretization of time[1]: this fully discrete problem has 139728 states. Some current algorithms for MDPs can deal with such state space sizes — especially heuristic search algorithms and algorithms making use of factored representations — but simple algorithms as Value Iteration over standard tabular representations of this size take a long time converging.

One could object to this argument that, with a unit discretization of time, the resolution takes exactly 71 value iterations because the problem is a finite horizon MDP with uncertain

[1]as presented in the next paragraphs, a unit discretization of time is the least necessary to roughly approximate the time-dependency introduced by the L and P_μ functions.

durations. Indeed, the comparison with the 139728 states problem is more valid for the case of general continuous variables. Anyway, 71 iterations times 1968 states corresponds to 139728 state *value* updates while we will see a little further that our prioritized sweeping method finishes in about 35000 state *function* updates which might be an interesting trade-off between calculation complexity and having continuous dynamics representations.

Our implementation of the $TMDP_{poly}$ algorithm is rather straightforward and leaves a lot of space to heuristic search improvements and better representations of the state space's discrete part, so it makes sense comparing the performance of Value Iteration or standard Prioritized Sweeping on these large discrete problems and the performance of $TMDP_{poly}$ on the hybrid one.

Variable	Description	Domain
t	time	$[0, 70]$
e	energy	$\{0, 1, \ldots, 39, 40\}$
p	position	$\{1, 2, 3, 4, 5, 6\}$
im_1	image 1 taken	$\{0, 1\}$
sa_1	sample 1 collected	$\{0, 1\}$
sa_2	sample 2 collected	$\{0, 1\}$

Table 7.1: Mars rover problem — state variables

The action space of the rover is described on table 7.2. The number of actions defined in this table is 23 (if we don't count the continuous wait action). However this number does not really mean much since only few of these actions are available in each state. Therefore, it is better to count the minimum and maximum number of actions available per state in the problem to get an idea of the problem's difficulty.

- Example of states that have the most available actions: $p = 3; 6 \leq e < 40; im_1 = 0$ $\leftrightarrow \{move(3,1), move(3,2), move(3,4), move(3,5), take\_picture(3), recharge, wait\}$
- Example of states that have the least available actions: $e = 0$ $\leftrightarrow \{recharge, wait\}$

$move(p_1, p_2)$	movement from p_1 to p_2
$take\_picture(p)$	takes the photo from position p
$sample\_rock(p)$	collects a rock sample from position p
$recharge$	fully charges the rover's battery
$wait(\tau)$	waits for a future date $t' = t + \tau$

Table 7.2: Mars rover problem — action space

Movement actions

Each movement action can result in six different outcomes:

- μ_1 — movement success and short duration
- μ_2 — movement failure and short duration

- μ_3 — movement success and average duration
- μ_4 — movement failure and average duration
- μ_5 — movement success and long duration
- μ_6 — movement failure and long duration

One has, independently of the current state, time and destination state:

$$\begin{aligned} L(\mu_1) &= 0.6 \\ L(\mu_2) &= 0.05 \\ L(\mu_3) &= 0.15 \\ L(\mu_4) &= 0.025 \\ L(\mu_5) &= 0.15 \\ L(\mu_6) &= 0.025 \end{aligned}$$

Destination state of outcome μ_1 corresponds to the target position with an energy decrease corresponding to a short duration movement. The destination states of the other outcomes can be described similarly.

The duration probability density functions have been implemented in five different versions, all bringing different complexity to the problem. These distribution are chosen so as to match the average and standard deviation of a Gaussian distribution on movement durations.

1. The first one uses piecewise polynomial probability density functions. More specifically, cubic splines, used to interpolate Gaussian distributions. An example of such a distribution is plotted on figure 7.8. Some additional details on calculation of the associated splines are given with the battery charge action description.
2. The second one only uses discrete distributions.
3. The third one uses quadratic splines yielding similar distributions to the ones of the first version.
4. The fourth one uses piecewise linear functions corresponding to applying algorithm 6.6 to the first version's distributions.
5. The fifth one uses only piecewise linear distributions, mainly "triangular" distributions.

We give an example of the two first versions on outcome μ_3 of action $move(1,4)$ applied in position $p = 1$. The piecewise polynomial version is plotted on figure 7.8.

$$\begin{aligned} P_{\mu_3}(\tau) =& 1_{[11,12]}(\tau) \cdot \left(-2\tau^3 + 69\tau^2 - 792\tau + 3025\right) + \\ & 1_{[12,13]}(\tau) \cdot \left(2\tau^3 - 75\tau^2 + 936\tau - 3887\right) \end{aligned}$$

$$P_{\mu_3}(\tau) = 0.25 \cdot \delta_{11.5}(\tau) + 0.5 \cdot \delta_{12}(\tau) + 0.25 \cdot \delta_{12.5}(\tau)$$

No reward is associated with movement actions.

There is an important caveat to mention here. *POLYTOOLS* is a rather complex set of operations trying to combine knowledge about formal calculus, algorithmic efficiency and numerical calculus stability. For example, the sequence of polynomials built for Sturm's method

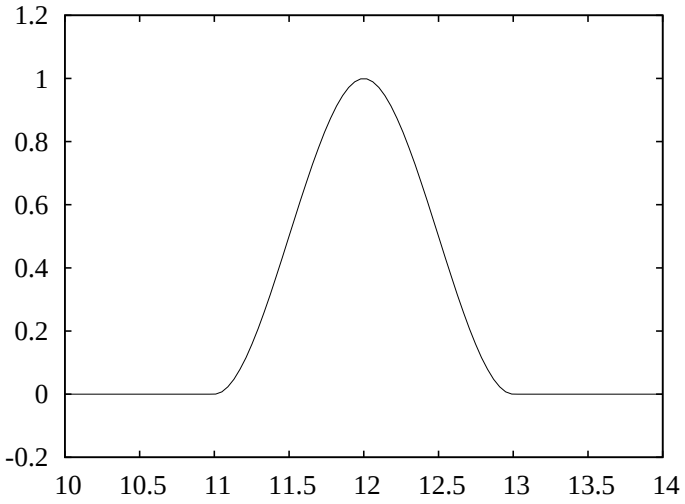


Figure 7.8: Duration probability of μ_3

(see appendix A for details) implies performing an exact Euclidean division of polynomials which is feasible in theory and easy to implement but which can imply a lot of numerical instability for ill-conditioned polynomials[2]. There are many examples of such technical difficulties which are completely unrelated to the planning problem but constitute a major obstacle to testing the $TMDP_{poly}$ planner for higher order polynomials. Because of these technical problems, only versions 2, 4 and 5 of the rover problem were actually solved using our implementation. The other versions are readily available but *POLYTOOLS* still needs some improvements and some fixing before they can be solved. This has another drawback: it was not possible to evaluate the trade-off between polynomial degree and number of pieces in the piecewise polynomial description because *POLYTOOLS* still has trouble with higher degree polynomials. However, the simple comparison between discrete density functions and piecewise linear ones already allows to draw some conclusions regarding the complexity on the operations involved and the advantages/drawbacks of such modeling features.

Taking the picture

This action is only available in positions 3 to 6, when the energy resource is sufficient and if a successful photo has not already been stored in memory. It can result in two different outcomes, either the picture is good or it has to be re-shot. The probabilities of a successful picture depend on the shooting location and on the time of day. They are illustrated on figure 7.9.

In all cases, the energy decrease is 1. Similarly, the transition duration is deterministic and has duration 1.

Finally, the reward for taking a good photo depends on the outcome's end date and on

[2]polynomials having a very small coefficient of high degree and a very large constant coefficient

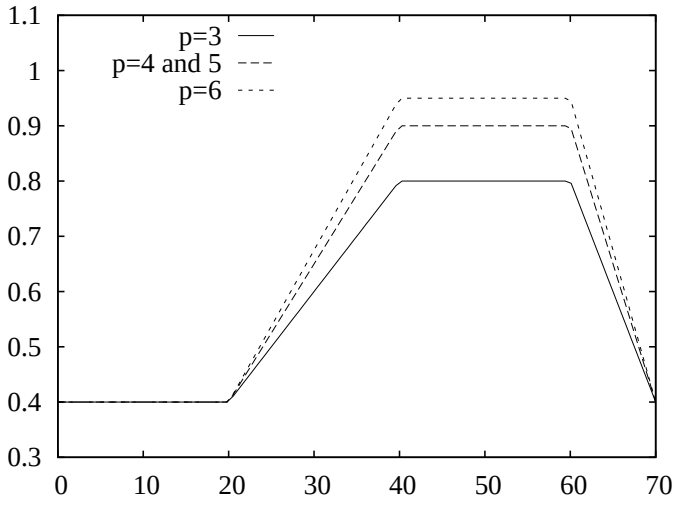


Figure 7.9: Probability of successful photo — $L(\mu_{success}|s, t, \texttt{take\_picture})$

the shooting site:

$$r_{t'}(\mu_{success}, p, t) = \begin{cases} 4 & \text{if } p = 3 \\ 5 & \text{if } p = 4 \\ 4 & \text{if } p = 5 \\ 7 & \text{if } p = 6 \end{cases}$$

Collecting the samples

Similarly to the picture action, this action is only available in positions 2 and 5, if the energy level is high enough and if the sample corresponding to the current position has not been collected yet. This action can result in a success or failure outcome; failure corresponding to a failure in grabbing the right sample and storing it. The probability of successfully collecting the sample is 0.7, regardless of the sampling site, the current state or the time of day.

Sampling duration can vary according to several possibilities in the grabbing scenario. It results in the following duration distributions:

$$P_{\mu_{success}}(\tau) = 0.2 \cdot \delta_3(\tau) + 0.6 \cdot \delta_4(\tau) + 0.2 \cdot \delta_5(\tau)$$
$$P_{\mu_{failure}}(\tau) = 0.5 \cdot \delta_2(\tau) + 0.5 \cdot \delta_3(\tau)$$

The reward for collecting sample 1 is 5, and the reward for sample 2 is 3.

Charging the batteries

Charging the batteries is an all-or-nothing action which performs a full battery charge, regardless of the initial energy level. However, the recharge duration depends on this initial level and on the lighting (directly linked with the time of day). There are two recharging speeds corresponding to two different outcomes: μ_1 corresponds to slow charging and μ_2 to fast charging. If the recharge action is undertaken between time 30 and 65, the μ_2 outcome is triggered, else μ_1 determines the recharge duration.

The final discrete state of a recharge action corresponds to setting e to its maximum value. The average durations of outcomes μ_1 and μ_2 are given by the following equations:

$$dur(\mu_1) = \begin{cases} 1 & \text{if } (e_{max} - e)/2.9 < 1 \\ (e_{max} - e)/2.9 & \text{else} \end{cases}$$
$$dur(\mu_2) = \begin{cases} 1 & \text{if } (e_{max} - e)/4.7 < 1 \\ (e_{max} - e)/4.7 & \text{else} \end{cases}$$

And we use a "deviation" parameter w:

$$w(\mu_1) = 1$$
$$w(\mu_2) = \begin{cases} 1 & \text{if } dur(\mu_2 \leq 8) \\ 2 & \text{else} \end{cases}$$

Similarly to the case of movement actions, we implemented two versions of the recharge action, the first one uses piecewise polynomial distributions, the second one uses discrete distributions. In the first case, the duration distribution function is — similarly to figure 7.8 — the cubic spline interpolation going through the points $(dur(\mu) - w(\mu), 0)$, $(dur(\mu), 1/w(\mu))$, $(dur(\mu) + w(\mu), 0)$ with slope zero at each interval's end[3].

In the discrete distributions case, the distribution was given as:

$$P_\mu(\tau) = 0.25 \cdot \delta_{dur(\mu)-w(\mu)}(\tau) + 0.5 \cdot \delta_{dur(\mu)}(\tau) + 0.25 \cdot \delta_{dur(\mu)+w(\mu)}(\tau)$$

There is no reward associated with the recharge action.

7.3.2 Optimization results

We present first the optimization results on the problem with only discrete probability density functions (version 2 of the movement and recharge actions). For this problem, we used a threshold on the priorities of 0.1, a precision on the t bounds of 10^{-3} and an approximation tolerance of 0.05. The rover problem took 38017 iterations to converge, corresponding to an average running time of 1690 seconds.

The algorithm was initialized with a null value function in all states. The initial priorities were obtained by performing a first pass of Value Iteration in the whole state space. The initial priority queue size was 868.

The evolution of the maximum priorities is shown on figure 7.10[4]. As expected, this evolution is not monotonous, but since it is closely linked with the Bellman error, it is globally decreasing. After 38017 iterations no update priority is above 0.1 and the priority list becomes empty.

Even though the decrease of priorities is not monotonous, one can easily see that, after the 10000th iteration, the few points which have high priorities are quickly solved and the moving average of priorities closely follows the lower bound on priorities of figure 7.10. This

[3] An interesting property of these distributions is that we don't need any scaling other than the division by $w(\mu)$ to guarantee they sum to one between $-\infty$ and $+\infty$.

[4] It is important to note that figure 7.10 represents only the evolution of the *maximum* priority along the iterations. So each point corresponds to a different abscissa (there is no range of priorities plotted here, only the largest one). This graph looks very dense because the 38017 points are plotted.

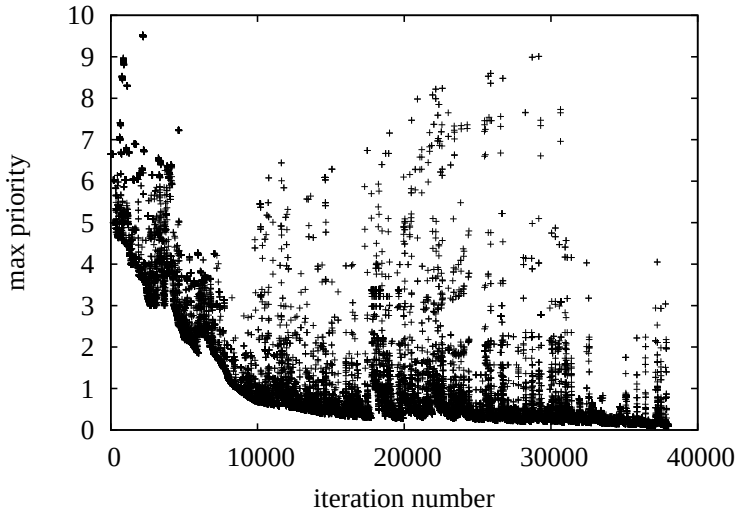


Figure 7.10: Evolution of the maximum priorities for the Mars rover problem

justifies in practice the use of this moving average curve as a performance profile.

Although the decrease of Bellman error is much faster than in the case of Value Iteration, the curve is not as steep as one could expect. Reaching a close-to-optimal value function in 10000 state visits is acceptable for a 1968 discrete states problem with an additional continuous variable, but the decrease in Bellman error seems a little slow. One can explain this "pathology" by the discretization of the energy variable. By discretizing, we create a highly connected problem with many states having an almost equivalent value function. Even though these value functions are very similar, the algorithm needs to update each of the corresponding states individually, thus linearly increasing the number of state visits needed to reduce the maximum priority's value.

In the case at hand (discrete distributions), the piecewise polynomial degree is stable so the approximation phase is *a priori* not necessary. However, successive convolutions with discrete density functions and addition, intersection, etc. yield a piecewise polynomial function with a lot of definition intervals over which the function itself is almost constant. Therefore, it makes sense to apply the approximation algorithm in order to reduce this number of intervals while conserving an L_∞-error bounded approximate function. This actually dramatically increases the algorithm's performance and avoids many numerical instabilities such as intervals having null width or the number of intervals exploding.

Figure 7.11 shows the evolution of each iteration's duration. These times were measured by steps of 10^{-2} seconds in order not to slow down too much the execution of the algorithm. Times returned as equal to zero are actually between zero and 0.01 seconds, but too small to be measured.

The next figures show the optimal policy and value function obtained in some specific states:

- $p = 1$, $e = 40$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$: figure 7.12.
- $p = 3$, $e = 20$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$: figure 7.13.

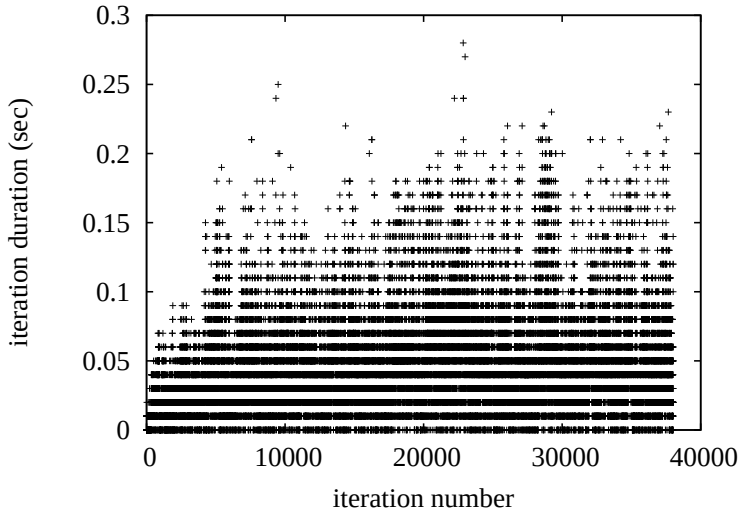


Figure 7.11: Evolution of individual iteration durations for the Mars rover problem

- $p = 2$, $e = 20$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$: figure 7.14.
- $p = 5$, $e = 30$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$: figure 7.15.

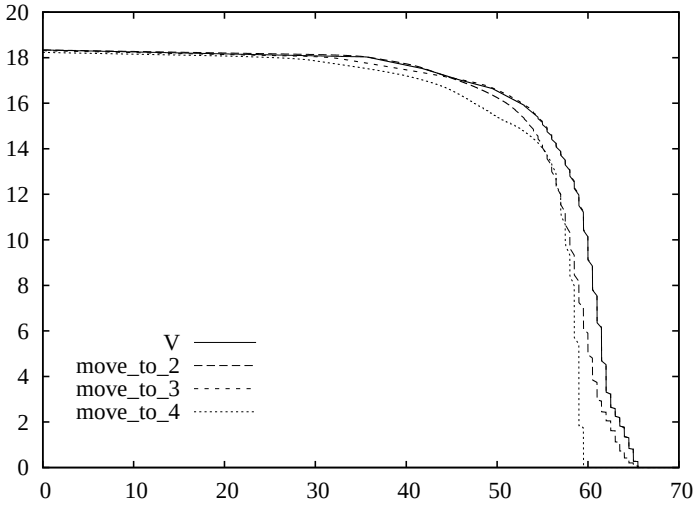


Figure 7.12: State $p = 1$, $e = 40$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$ — Value function

The *recharge* action is very problematic to our algorithm: it almost acts as a *wait* action but often provides better results since charging durations are not prohibitively long and they usually lead to higher gain states. Therefore, several successive optimizations often introduce intermediate charge actions inside other actions. This is not visible on the policy of the $e = 40$ states, but becomes obvious with, for example, the $e = 20$ state whose value function is represented on figure 7.13.

The policy in state $p = 1$, $e = 40$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$ is:

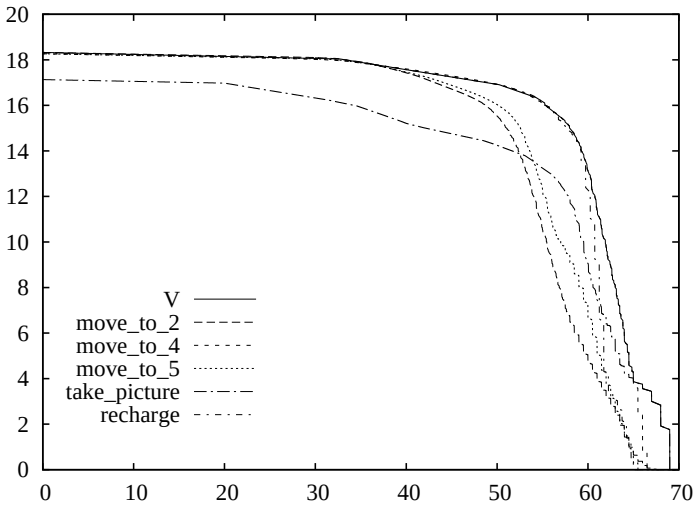


Figure 7.13: State $p = 3$, $e = 20$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$ — Value function

$$\begin{array}{ll} [0; 44.6159] : & \text{move\_to\_2} \\ [44.6159; 70] : & \text{move\_to\_3} \end{array}$$

While the policy in state $p = 3$, $e = 20$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$ is:

$$\begin{array}{ll} [0; 30.7689] : & \text{move\_to\_2} \\ [30.7689; 39.809] : & \text{recharge} \\ [39.809; 49.6915] : & \text{move\_to\_4} \\ [49.6915; 49.7044] : & \text{recharge} \\ [49.7044; 49.8936] : & \text{move\_to\_4} \\ [49.8936; 50.0645] : & \text{recharge} \\ [50.0645; 53.0532] : & \text{move\_to\_4} \\ [53.0532; 55.2447] : & \text{recharge} \\ [55.2447; 55.266] : & \text{move\_to\_4} \\ [55.266; 55.2925] : & \text{recharge} \\ [55.2925; 55.3298] : & \text{move\_to\_4} \\ [55.3298; 55.3772] : & \text{recharge} \\ [55.3772; 55.4149] : & \text{move\_to\_4} \\ [55.4149; 55.7447] : & \text{recharge} \\ [55.7447; 56.1915] : & \text{move\_to\_4} \\ [56.1915; 56.2447] : & \text{recharge} \\ [56.2447; 58.617] : & \text{move\_to\_4} \\ [58.617; 58.6592] : & \text{recharge} \\ [58.6592; 58.6809] : & \text{move\_to\_4} \\ [58.6809; 58.7447] : & \text{recharge} \\ [58.7447; 59.117] : & \text{move\_to\_4} \\ [59.117; 59.2447] : & \text{recharge} \\ [59.2447; 65] : & \text{move\_to\_4} \\ [65; 70] : & \text{take\_picture} \end{array}$$

Once again, such a policy doesn't mean "start recharging at time 30.7689 and stop at time 39.809", instead it means "if the policy is asked for an action to perform at any time between 30.7689 and 39.809, trigger the battery recharge action" this action might take us

to a completely different state at a time independent of the values 30.7689 and 39.809.

It is interesting to note that the rover found it more interesting to go to $p = 2$ early in the "morning", when the lighting is still bad and not appropriate for a picture. This behavior is consistent with the problem specification of [Bresina et al., 2002] and the expected optimal plan. We can follow this action and go check in $p = 2$ what the policy is. For this, we choose the energy level $e = 20$. The value function of state $p = 2$, $e = 20$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$ is plotted on figure 7.14.

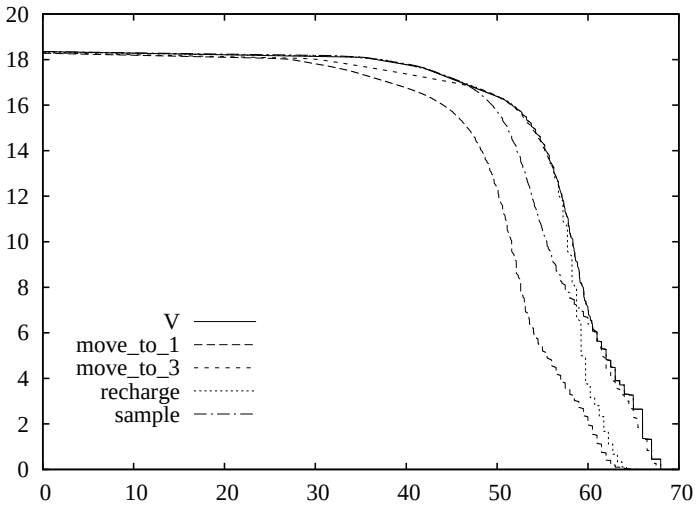


Figure 7.14: State $p = 2$, $e = 20$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$ — Value function

The associated policy is:

$[0; 31.1855]$:	sample
$[31.1855; 33.0164]$:	recharge
$[33.0164; 36.1848]$:	sample
$[36.1848; 38.6721]$:	recharge
$[38.6721; 42.8409]$:	sample
$[42.8409; 54.2447]$:	recharge
$[54.2447; 54.3191]$:	move_to_3
$[54.3191; 54.7447]$:	recharge
$[54.7447; 54.8936]$:	move_to_3
$[54.8936; 55.2447]$:	recharge
$[55.2447; 55.5]$:	move_to_3
$[55.5; 55.7447]$:	recharge
$[55.7447; 56.0426]$:	move_to_3
$[56.0426; 56.2447]$:	recharge
$[56.2447; 56.6277]$:	move_to_3
$[56.6277; 56.7447]$:	recharge
$[56.7447; 60.766]$:	move_to_3
$[60.766; 66]$:	sample
$[66; 66.4311]$:	move_to_3
$[66.4311; 70]$:	sample

While the *recharge*/act interleaving is still present, we can see that the strategy is consistent: in the morning the rover finds it more interesting to immediately perform the sampling operation while in the afternoon, it is better to move to the shooting sites in order to get the best picture possible. This is illustrated again in our last sample state: $p = 5$, $e = 30$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$ whose value function is plotted on figure 7.15

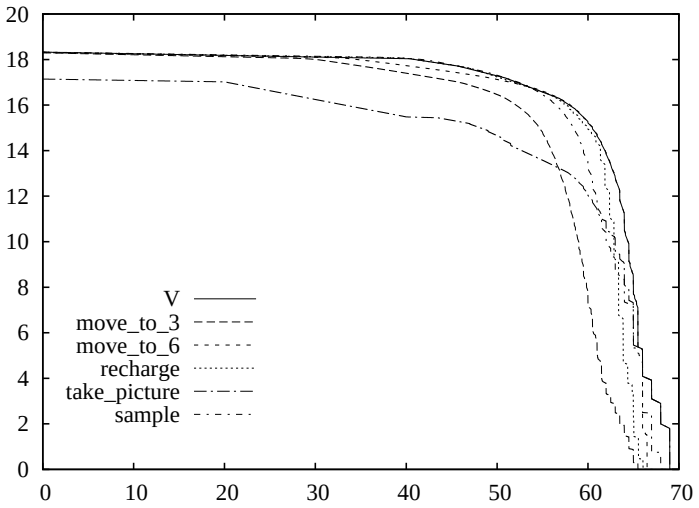


Figure 7.15: State $p = 5$, $e = 30$, $im_1 = 0$, $sa_1 = 0$, $sa_2 = 0$ — Value function

In this state, the choice is crucial, the rover can either sample the rock, take the picture, or move to $p = 6$ to get a better shooting position. The policy found is:

$[0; 38.1281]$:	sample
$[38.1281; 40.4101]$:	recharge
$[40.4101; 47.3729]$:	sample
$[47.3729; 49.2064]$:	recharge
$[49.2064; 53.2271]$:	sample
$[53.2271; 65.5]$:	move_to_6
$[65.5; 70]$:	take_picture

The *recharge*/act interleaving appears sometimes between other actions than *recharge* as for the latest part of the policy in $p = 2$, $e = 20$, etc. Even though there is no final explanation to this behavior, it can be due to two main different reasons:

- First, $TMDP_{poly}$ introduces a static ordering over actions[5]. Therefore, whenever there is a tie between actions, the same one is always chosen first. Since *recharge* might take the process to states with equivalent rewards, this can introduce equivalent actions.
- Secondly, when the actions' Q functions are very close (but not equal), the approximation scheme can sometimes slightly alter the monotonicity of a Q function and locally break the dominance of an action over another. This only occurs when the two Q functions are already very close in the first place and might result in this interleaving. Therefore, one can expect such an interleaving to have little impact on the global behavior: the value function still tends to the optimal value function.

[5]This ordering is due to the way we store the transitions: they are sorted by action name for efficient lookup. Therefore, the static ordering only depends on the alphabetical order.

It is interesting to notice that as one sweeps through the different energy levels for the same value of p, the policy bounds change but the structure itself remains. This feature is illustrated on figures 7.16(a) and 7.16(b). This illustrates the strong similarity between energy and time in this case and between *wait* and *recharge*. It also points out the limits of the discretization approach which arbitrarily separates continuous domains into discrete levels and makes the problem more complex in discretized form than in continuous formulation. Solving the problem with two continuous actions *wait* and *recharge* would be an interesting challenge which is beyond the scope of this chapter and will be discussed in chapter 8.

Similarly, figure 7.16(b) illustrates the interests and limits of a kd-tree representation for the policy or value function. As the number of variables grows, the structure of the policy becomes harder to capture using sets of hypercubes. Thus, in higher dimensional state spaces the approaches of [Feng et al., 2004], [Li and Littman, 2005] or [Benazera et al., 2005] might capture less easily the variations of the policy and value function.

Finally, one can remark that *wait* rarely appears in the policy. Actually, it does appear before some movement actions. This can be explained by the fact that it is more risky to start recharging at the time in question than to wait before moving, taking a picture (for example) and recharging afterwards. Since recharging times are not too long, in most cases, it is preferable to recharge first and then to act. *wait* can also be found just before a recharge action because of the recharging mode switching.

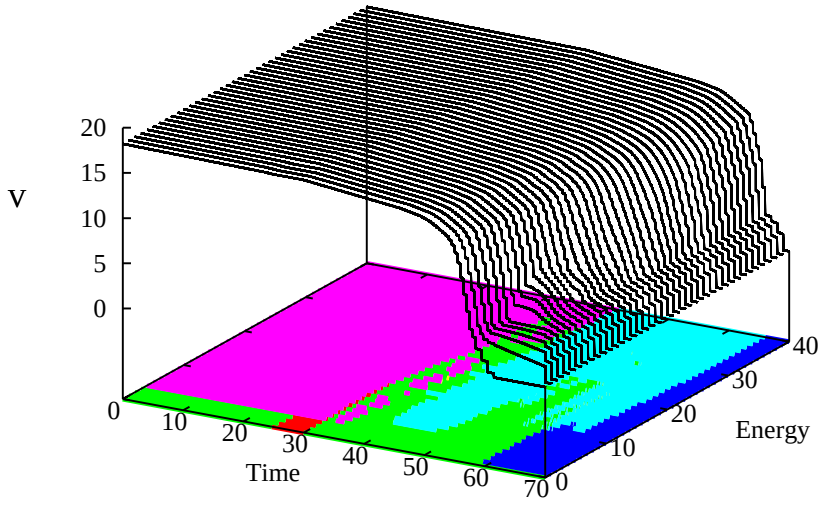
For version 5 of the rover problem, the final policy and value function is comparable to the ones presented above. The computation time and number of iterations needed to reach an empty priority queue are given in table 7.3 for each version. As for the discrete distributions case, we used a threshold on priorities of 0.1, a precision on t of 10^{-3} and an approximation tolerance of 0.05.

Problem version	Iterations before convergence	Average running time
version 2	38017	1690 seconds
version 5	53976	9155 seconds

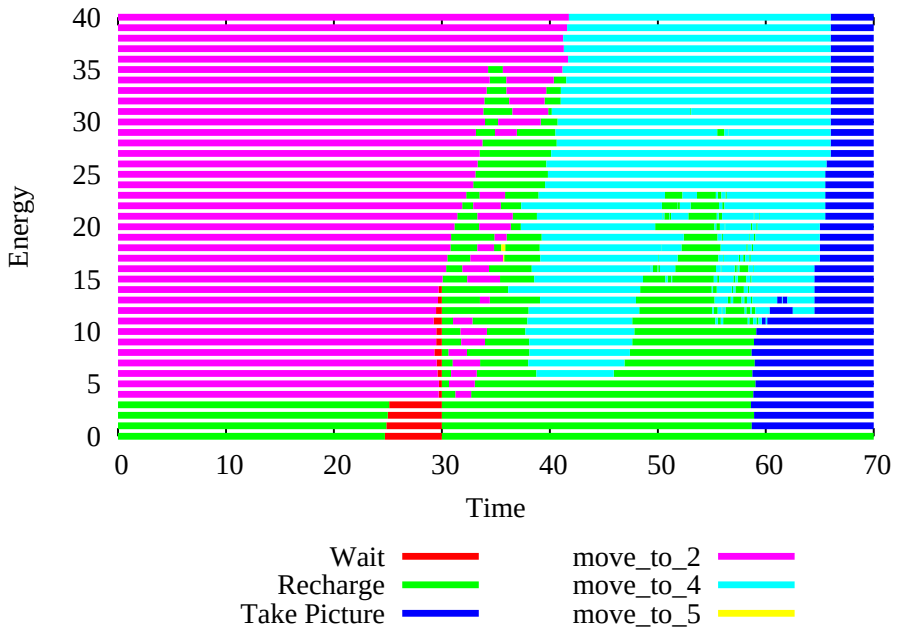
Table 7.3: Rover problem — optimization time

Table 7.3 illustrates the calculation overhead introduced by the piecewise polynomial function calculations, compared to the case of discrete distributions. For an increase in the number of iterations of a factor 1.4, the calculation time has been multiplied by 5.4. This underlines the fact that while the complexity of the resolution does not change in terms or state updates, the calculation times on piecewise polynomial representations still need a lot of attention and improvement.

Version 4 suffered from another technical difficulty due to the L_∞ norm. For continuous functions of t, it makes sense to use a L_∞ norm in order to derive bounds for optimality. It implies having a precision on V of ϵ. It implies exactly the same thing for piecewise continuous functions but the discontinuity points become very hard to solve since the L_∞ measure has repercussions on the precision on t for these discontinuity points. This feature makes the problem of real number rounding even more problematic and quickly introduces instability in the prioritized sweeping ordering. Thus, the optimization of version 4 of the rover problem did not converge with our current implementation. It was stopped after 20



(a) Value function and policy in $p = 3$ when no goals have been completed yet



(b) Policy in $p = 3$ when no goals have been completed yet — 2D view

Figure 7.16: Structured policy in $p = 3$ for the rover problem

hours of computing and about 110000 state updates, while presenting a pseudo-oscillating behavior on the priorities which we empirically attribute to the approximation phase and more specifically to the difficulties of calculating the L_∞ bounds at discontinuity points. There might be other reasons which we would not have identified for such a non-convergent behaviour, such as implementation mistakes in the first place, or approximation tolerance propagation which induces repetitive Bellman errors of more than ϵ in some states.

Figures 7.17 and 7.18 show the evolution of the priorities and individual iteration times when solving version 5 or the Mars rover problem.

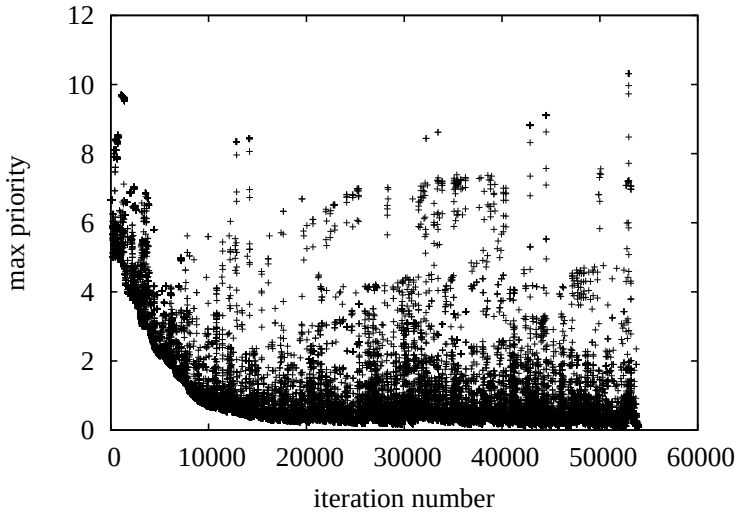


Figure 7.17: Evolution of the maximum priorities for the Mars rover problem, version 5

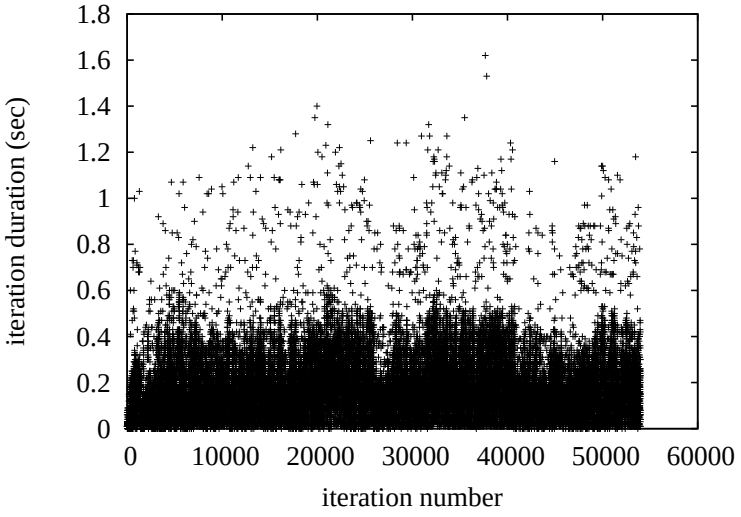


Figure 7.18: Evolution of individual iteration durations for the Mars rover problem, version 5

7.4 The UAV patrol problem

7.4.1 Problem definition

The second main example we will present here highlights an interesting use of TMDPs. In all the previous examples, the *wait* action was mainly used to "freeze" the agent's discrete state while letting the time variable grow in order to catch any good future reward available in the current state. The UAV patrol problem is different in the sense that it does not define a *wait* action, but a *patrol* action which is strictly equivalent to *wait* in terms of TMDP description. $patrol(\tau)$ is both a continuous action and — contrarily to other *wait* actions which usually provide costs — the only action providing rewards. This example illustrates the fact that we can replace *wait* by another continuous action and optimize a strategy on a hybrid action space.

Let us now imagine an unmanned air vehicle (UAV) having a mission defined in terms of patrolling over certain areas of a map. More specifically, let us imagine a map with four areas of interest where the UAV has to observe a certain phenomenon. The human agent specifying the mission indicates during which time intervals the UAV should watch each zone and assigns different importances to zones in case of scheduling conflicts. For example, one could say:

> "Set importance 2 on position p_1 between $t = 0$ and $t = 25$,
> then set importance 2 on the same position p_1 between $t = 60$ and $t = 70$,
> also set importance 5 on position p_2 between $t = 45$ and $t = 50$,
> assign importance 2 on position p_3 between $t = 20$ and $t = 50$
> and finally set importance 3 on position p_4 between $t = 45$ and $t = 70$."

Now let us suppose that the UAV's navigation map is described as a grid of positions $p = (x, y)$ as in figure 7.19. This grid represents the navigation environment of the UAV and the reward rates associated to each of the patrol zones. The UAV is given a meteorological model indicating how the wind is supposed to blow during the mission and has some probabilistic knowledge about the results of its atomic movement actions depending on the wind.

The planning problem corresponds to finding the optimal policy of movement between positions and local patrolling as a function of the current position and the current time. Thus, the action space can be written as in table 7.4 and the state space contains the variables presented in table 7.5.

$patrol(\tau)$	continuous action indicating to patrol the current position for τ time units
N, S, E, W	discrete movement actions taking the UAV to a nearby position

Table 7.4: Patrol problem — action space

We use this paragraph to shortly present the simple wind model we used. Between $t = 8$ and $t = 30$, the wind blows from East to West, and between $t = 60$ and $t = 80$, from North to South. At all other times, there is no wind. When the wind blows, this changes the probabilities of making a successful move and the transition durations. Without entering

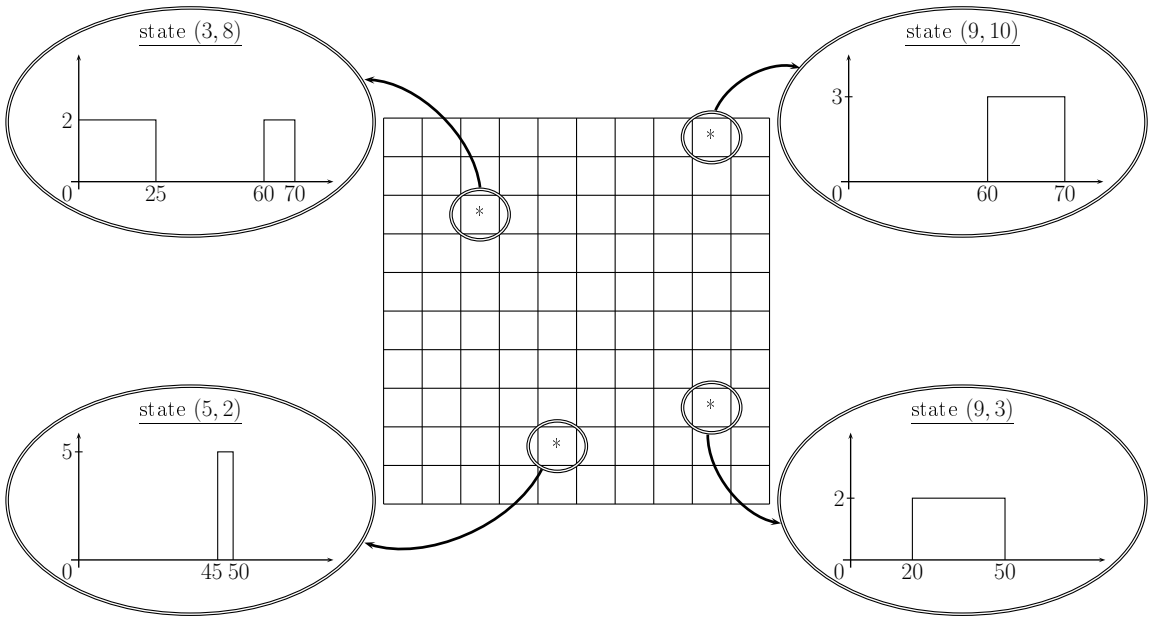


Figure 7.19: UAV patrol problem — Reward rates

t	the current time, continuous variable taking its values in $[0, 100]$
x	discrete latitude of the UAV, taking its values in $\{1, \ldots, 10\}$
y	discrete longitude of the UAV, taking its values in $\{1, \ldots, 10\}$

Table 7.5: Patrol problem — state space

the modeling details, the wind has the influence of "pushing" the UAV in a specific direction which shortens or lengthens the movement durations and can result in off-course final transition states.

Therefore, the UAV patrol problem is a grid world navigation problem with stochastic movement actions, stochastic continuous transition durations, hybrid state and action spaces with the TMDP hypothesis on the continuous action.

7.4.2 Optimization results

Because the discrete state space represents only the geographical position of the UAV, this problem is easy to represent graphically. As in the rover case, we designed several versions of the patrol problem. The first version uses only discrete probability density functions, the second one uses piecewise linear density functions.

Table 7.6 summarizes the optimization results for the two versions of the patrol problem. In both cases, the threshold on priorities was set to 0.1, the approximation L_∞ bound was equal to 0.05 and the precision on t for the approximate polynomial calculations was 10^{-3}.

Problem	Iterations before convergence	Average running time
version 1	531	13.90 seconds
version 2	824	740.17 seconds

Table 7.6: Patrol problem — optimization time

As in the Mars rover case, the figures of table 7.6 illustrate the fact that piecewise polynomial operations (such as convolution, etc.) still need a lot of optimizing. For an increase of a factor 1.55 in the number of iterations, the calculation time has been multiplied by 53.25.

One can also compare this number of 531 state visits with the number of state updates performed in the Value Iteration-like algorithm of [Boyan and Littman, 2001]. With the latter algorithm, the value function converges to an ϵ-optimal value function after 330 passes through the state space, corresponding to 33000 state updates. Therefore, performing asynchronous dynamic programming with priorities reduced the number of state visits by a factor 62.

Figures 7.20 to 7.23 present the evolution of priorities and calculation times for the two versions of the patrol problem.

The increase of priorities around iteration 120 is due to the same phenomenon as illustrated on the three states problem, in section 7.2.

In order to illustrate the evolution of V on a single state, we have selected state $(7, 7)$ on the first version of the patrol problem. This state is only updated five times during the whole process. Since there are 531 updates and 100 states, this number of updates is representative of what happens in average over the whole state space. Figures 7.24 to 7.28 show the evolution of the value function and of the policy. One can tell the "update story" of this state:

- State $(7, 7)$ is updated for the first time during the 40th iteration because it previously had a high priority of 74.98, directly inherited from the propagation of the reward for

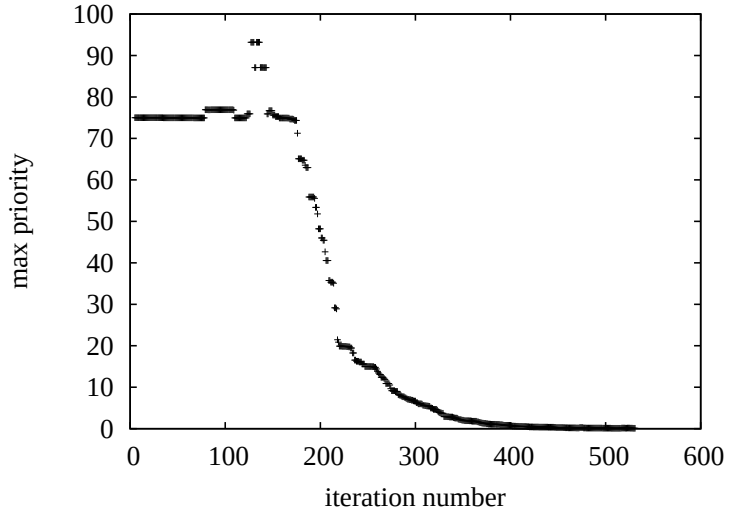


Figure 7.20: UAV patrol problem — Priorities evolution, first version

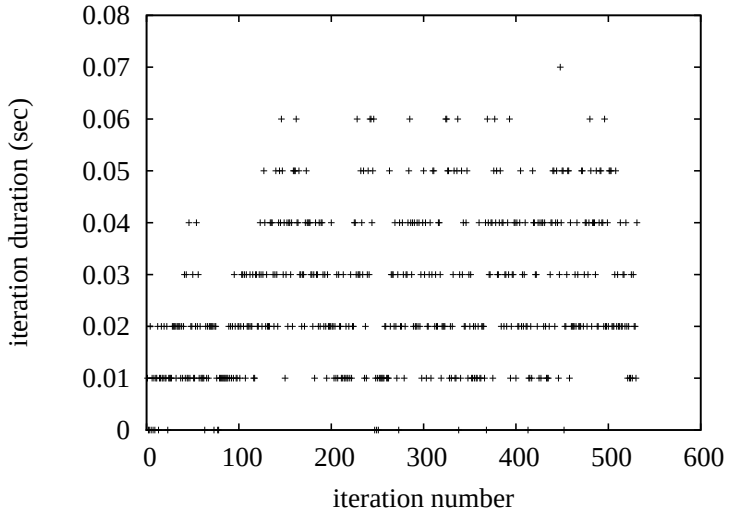


Figure 7.21: UAV patrol problem — Update durations, first version

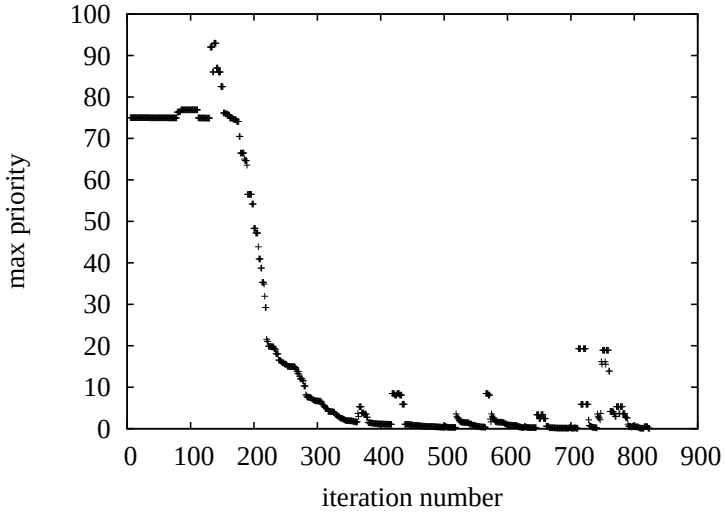


Figure 7.22: UAV patrol problem — Priorities evolution, second version

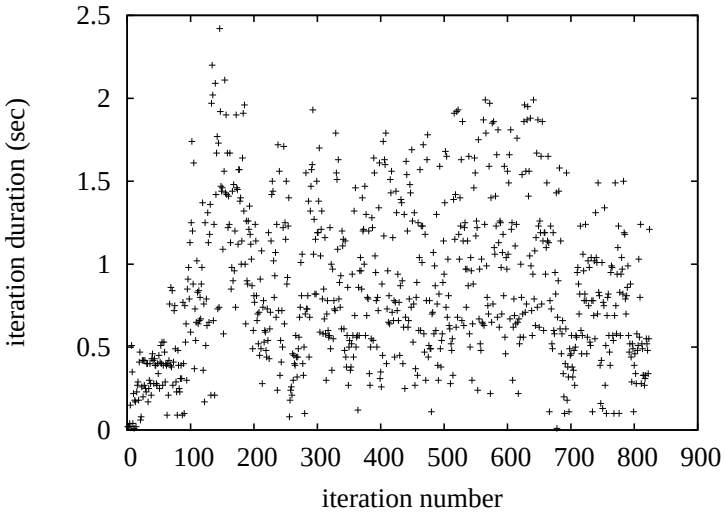


Figure 7.23: UAV patrol problem — Update durations, second version

the patrol zone situated in $(9, 10)$ (figure 7.24).

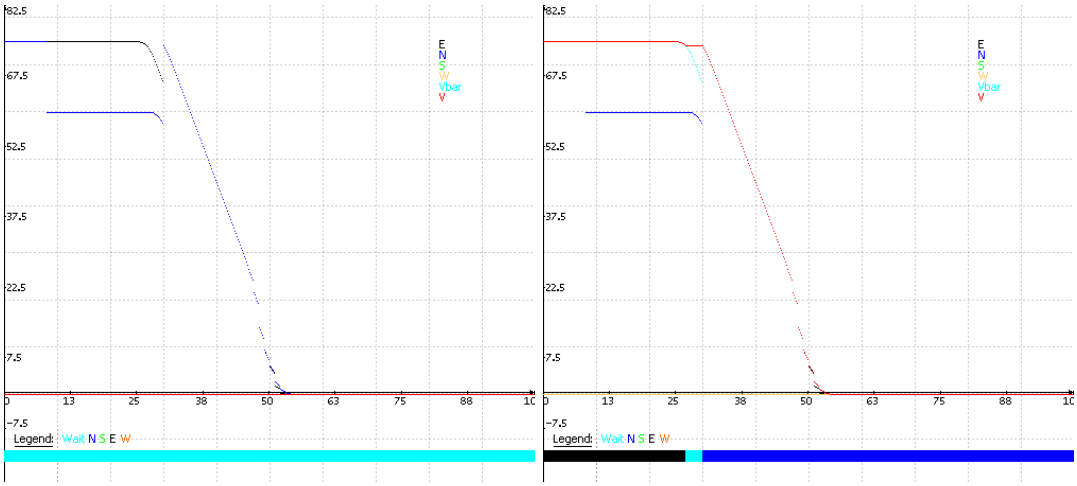
- At iteration 43, one of its neighbors is updated and it receives priority 14.99.
- At iteration 57, again one of its neighbors is updated and it receives a higher priority of 74.96, thus pushing it almost at the top of the priority list.
- It is then updated for a second time at iteration 66 (figure 7.25).
- Almost immediately after its update, at iteration 68, it receives priority 14.96. These quick priority changes come from the fact that $TMDP_{poly}$ focuses on the states which have the largest variations to let them converge first. Since $(7, 7)$ is one of the central states in the map, we can expect the policy to be a delicate compromise between directions and $TMDP_{poly}$ will focus on it in order to let it converge early in the optimization process.
- The priorities propagate the change information to the rest of the state space and nothing happens before iteration 225 when a neighbor is updated again, hence providing $(7, 7)$ with priority 16.26.
- It is updated for the third time during update 237 (figure 7.26) and keeps its priority of zero until update number 275 where it receives priority 6.01.
- This priority lets it be updated for the fourth time at update 304 (figure 7.27).
- Its priority is finally set to 0.56 at update 333.
- The final update occurs at iteration 408 (figure 7.28).
- After this iteration no priority of more than 0.1 is assigned to state $(7, 7)$ and the value function and policy do not change anymore.

$TMDP_{poly}$ uses the alphabetical static ordering on actions to break any ties. Since actions "West" and "North" appear to be equivalent several times during the updates, the chosen action is always "North", leaving some patches of "West" in the policy when the latter is strictly dominant (at iteration 304 for instance).

Based on the $TMDP_{poly}$ planner, we built a graphical demonstration interface for the patrol problem. As illustrated on figure 7.29, this interface allows to change the optimization parameters, perform step-by-step prioritized sweeping, run and pause the optimization process and save the result to text files or images.

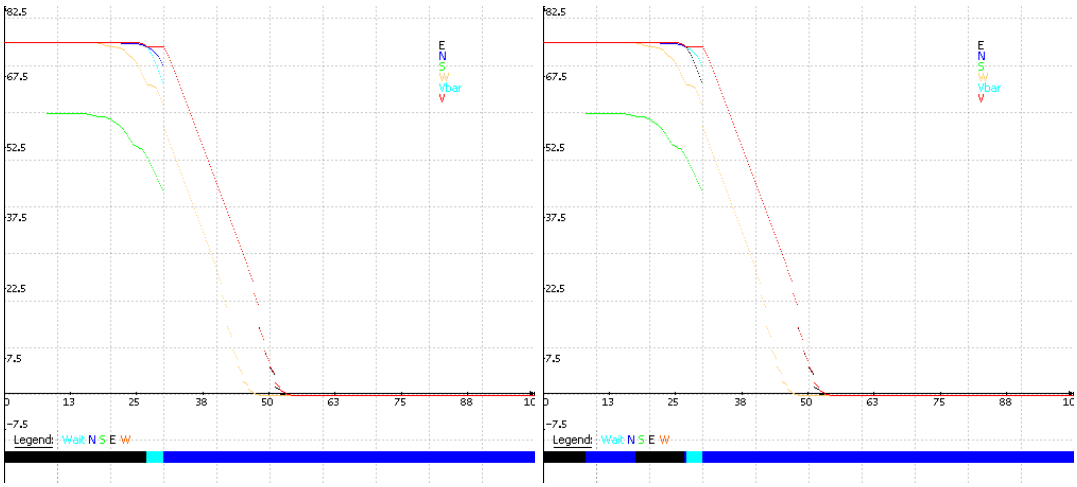
In the "grid" window of the interface, the red square indicates the first state in the current priority queue. For instance, on figure 7.29, one can see in window "TMDPpoly" that 124 states have been updated so far and that the current highest priority is 75.93. This priority is the one of state $(4, 6)$ where the red cursor is positioned. The blue square in the "grid" window is positioned by the user. It is used to select a certain discrete state and to display its current V, $\overline{V}$ and Q functions as well as its current policy in the windows in the middle.

The numbers displayed on the grid represent the current priority queue. This priority queue is initialized with the four patrol zones and quickly spreads by local propagation of the priorities.



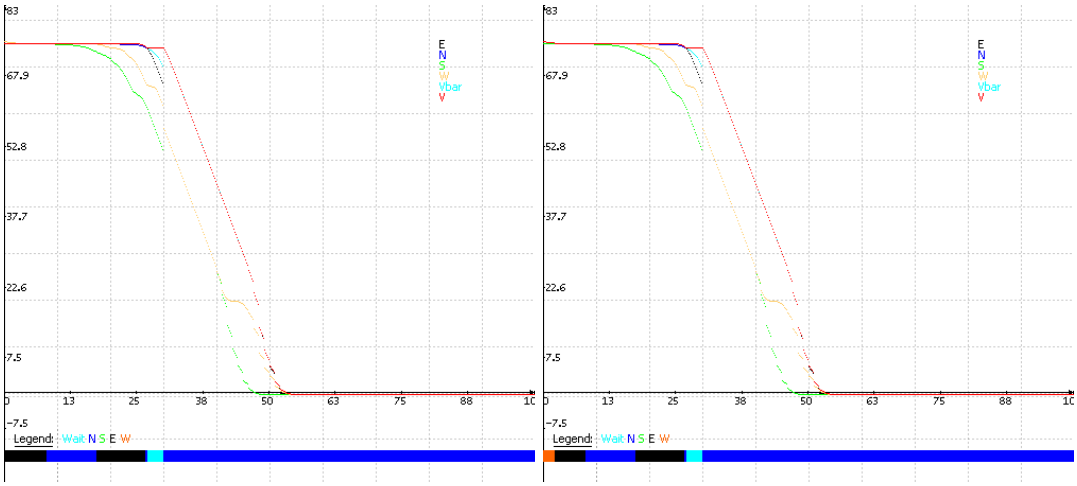
(a) Before (b) After

Figure 7.24: UAV patrol problem — state $(7, 7)$, iterations 40 and 41



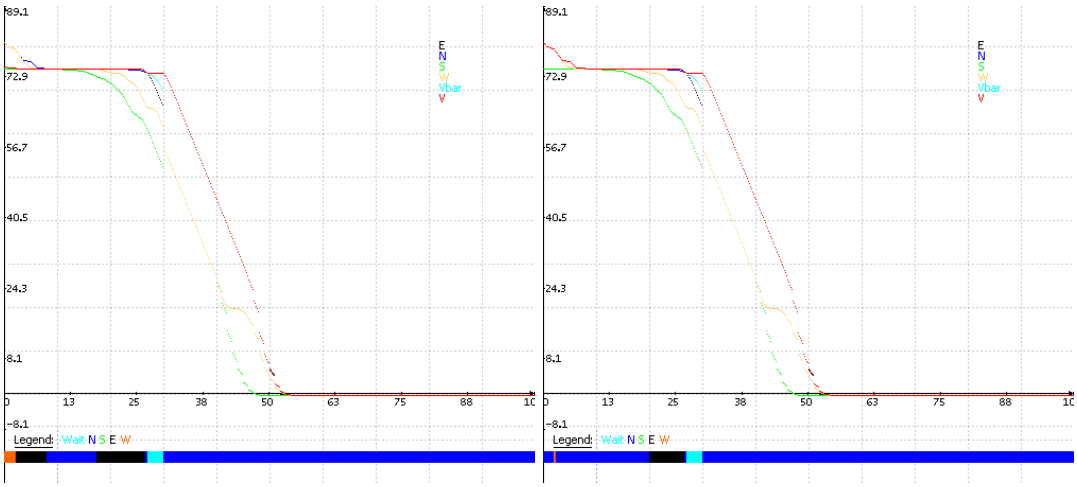
(a) Before (b) After

Figure 7.25: UAV patrol problem — state $(7, 7)$, iterations 66 and 67



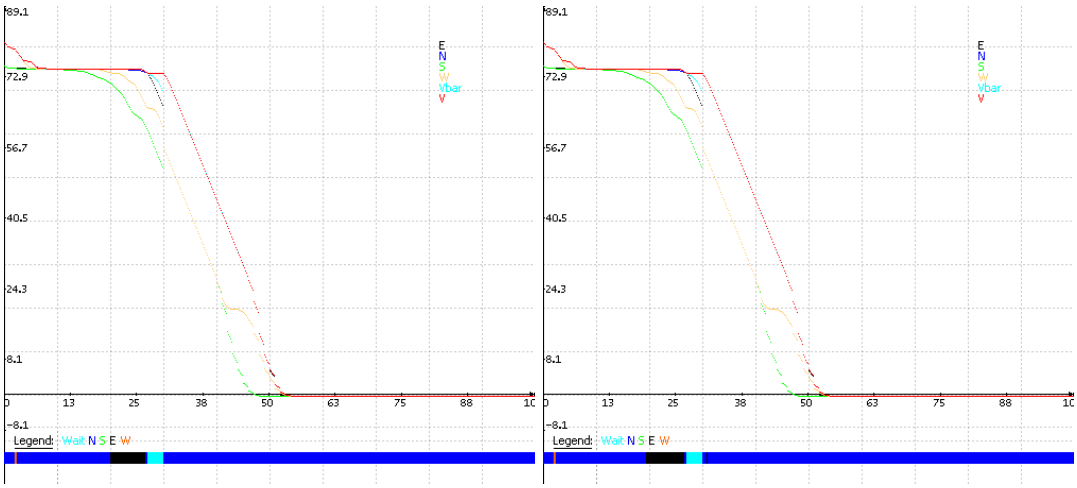
(a) Before (b) After

Figure 7.26: UAV patrol problem — state $(7, 7)$, iterations 237 and 238



(a) Before (b) After

Figure 7.27: UAV patrol problem — state $(7, 7)$, iterations 304 and 305



(a) Before (b) After

Figure 7.28: UAV patrol problem — state $(7, 7)$, iterations 408 and 409

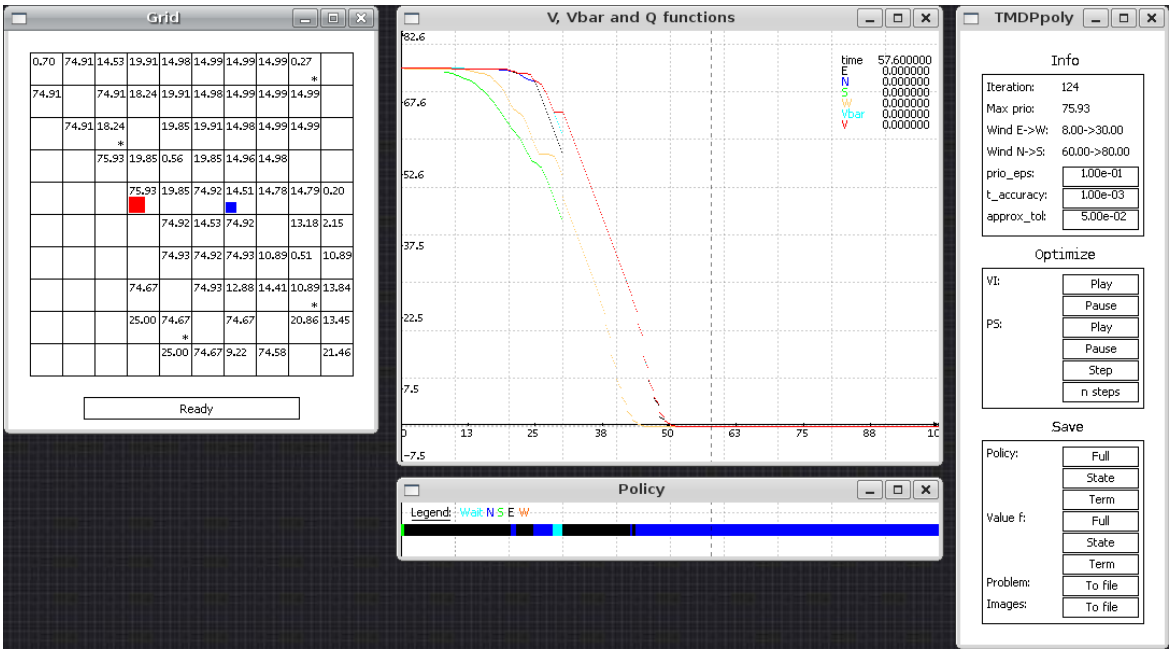


Figure 7.29: UAV patrol problem — graphical interface

Often, when clicking in the grid on a certain discrete state, one can notice that some Q-functions are actually higher than the current V or $\overline{V}$ functions. This is normal since, as explained in algorithm 6.4 and in section 6.3, Q functions are updated *after* updating the V functions, in order to propagate the priorities to parent states. Therefore, some states can have Q functions higher than their V functions just because their neighbors have been updated. In these cases, the states in question necessarily have a non-zero priority[6].

In the end, the UAV patrol problem illustrate an interesting alternative use of TMDPs by making the *wait* (*patrol*) action the only reward-providing action. It opens the door to the general specification of hybrid state and action problems as long as they verify the TMDP hypotheses.

7.5 Conclusion

Finally, this chapter illustrates how the $TMDP_{poly}$ algorithm works and where are its algorithmic advantages and drawbacks. It results in a formal method for computing the time-dependent optimal policy for temporal Markov decision problems, formulated as TMDPs. By pointing out the TMDP limitations, we were able to extend them, both in terms of representation capability (continuous distributions) and in terms of resolution method (the $TMDP_{poly}$ algorithm in itself). A next step in extending the TMDP resolution framework would be to integrate the use of the W function, specifying the system's dynamics during waiting phases. Using this function might however bring the problem back to a more general setup: if the undisturbed system's evolution is stochastic, then *wait* will have to be redefined and the difference with other possible continuous actions will be reduced. Another step would be to introduce the biases on priorities which we presented in section 6.3 in order to exploit even more the causality property associated with the time variable. This indeed corresponds to extending the priorities definition to states (s, t) (instead of states s currently). Such an improvement is expected to improve even more $TMDP_{poly}$'s efficiency since it will directly exploit the loop-free structure of temporal Markov decision problems.

This chapter also brings multiple perspectives. First, it introduces a fully implemented method for performing what we could name "formal Bellman backups" on a hybrid state space. This method is directly applied to the TMDP case and depends a lot on the TMDP hypotheses. It provides a practical, polynomial-based, formal calculus alternative to Monte-Carlo sampling methods which are the current common way of tackling hybrid state and action problems.

Chapter 8 will generalize the current TMDP framework to a more general class of hybrid problems, thus underlining how this current implementation can be reused for more general cases. Chapter 10 will try to highlight how this method of formal Bellman backups can be extended to these more general cases and will discuss where the difficulties lie.

Secondly, we used the $TMDP_{poly}$ algorithm as defined in the previous chapters to solve hybrid state and action problems such as the Mars rover and the UAV patrol problems. While this implementation is not able to scale to very large state spaces yet, it already provides a reasonable basis for solving this class of continuous time problems and extends immediately to the case of a single continuous state variable and a single continuous action as in the patrol problem case. Improving this method with heuristic guidance, structured representations of

[6]Even though, at the end of the algorithm, these priorities can be considered null because they are below the priority threshold. In this case, the slight variation of Q (amplitude < 0.01 is not visible on the graph).

the discrete part of the state space and better low-level function manipulation operators are some of the keys needed to scale up to larger domains. While these issues will be discussed in chapter 10, they are independent of the basis of the *TMDP$_{poly}$* method which already provides results on time-dependent problems as the Mars rover problem.

Then, one of the main practical conclusions from our experiments is that improving the efficiency of the *POLYTOOLS* implementation yields a dramatic improvement of the overall planner's efficiency. This is quite natural since the whole architecture is built above the *POLYTOOLS* implementation. Therefore, it would be very interesting to:

- improve *POLYTOOLS* 's implementation and efficiency in the first place, but also to
- test the *TMDP$_{poly}$* planner with different degrees for interpolation, in particular cubic splines which are not functional today because of technical implementation reasons; this will allow us to
- evaluate the degree/pieces compromise[7].

Therefore, improvement of the *POLYTOOLS* / *TMDP$_{poly}$* framework is still necessary to help understanding the advantages and drawbacks of our method and extend them to more general cases.

One important conclusion which does not appear visibly in the previous results is the huge impact of algorithm 6.6 on the optimization process. Without this algorithm, both the degree of polynomials and the number of definition intervals explode and the optimization gets stuck in very long, sometimes unpredictable, calculations for nothing. Even in the discrete distributions case, algorithm 6.6 decreases dramatically the computational time while conserving the global efficiency of the method and the L_∞ bounds on the value function.

From the algorithmic point of view, the *causality* feature of temporal Markov problems has not been used to its full possibilities. Even though this is encouraging with respect to the adaptability of our method to another continuous variable which would not have such properties[8], it is a point on which improvement of the *TMDP$_{poly}$* algorithm is possible. For example, focusing on the latest time intervals of the problem first might accelerate convergence since we work with backward propagation. Letting the latest times converge first can actually insure that full parts of the time-dependent value functions have converged and need not be further revised. Thus it would exploit more the oriented nature of the time variable. As mentioned in section 6.3 this could be done by biasing the way we calculate the priorities. It could also take advantage of partial calculation of the $V(s,t)$ functions: during the first state updates, only the "latest" part of the function is important, then, when it has converged, one can focus on "earlier" parts.

Finally, we can conclude that the main obstacle to the *TMDP$_{poly}$* implementation and experiments was spawned by the very nature of piecewise polynomial functions formal calculus. While this obstacle has been at least partially overcome, there still remains a lot of possible improvements for this work. These improvements can especially reduce the gap between the computational times associated to piecewise linear versions of our problems and the discrete distribution versions. Since the number of iterations needed before convergence

[7] Compromise between polynomial's degree and number of definition intervals in the piecewise polynomial functions.

[8] Namely, causality implies no current event will have repercussions in the past and thus no change in the policy at t will have impact on the value function of the current policy at posterior times.

is comparable in both cases, using piecewise continuous distributions will become competitive with discrete ones when the related operations will have been improved regarding calculation time. Still, by only looking at the number of iterations before convergence, we can deduce that there was very little additional complexity associated with using these piecewise continuous distributions at the planner's level. Moreover, low-level numerical problems such as the ones mentioned earlier — related to the precision of L_∞ bounds and the failures of root finding methods — illustrate the main obstacles associated to dealing with piecewise continuous functions in general and piecewise polynomial ones in our particular case. Some problems intrinsically have such piecewise continuous distributions and it might be rather cumbersome to approximate them with discrete ones. The $TMDP_{poly}$ planner with improved piecewise polynomial functions handling might open the door to directly dealing with such problems.

8

Generalizing MDPs to continuous observable time: the XMDP framework

Including time as a continuous observable variable in the MDP state space naturally leads to considering the continuous $wait(\tau)$ action on top of all other previous discrete actions. More generally, including continuous variables in the state space often calls for continuous or hybrid (continuous and discrete) actions. We have seen in chapter 2 that the time variable played a particular role with respect to the discounted criterion. In this chapter, we build on the standard MDP framework in order to extend it to continuous time and resources and to the corresponding parametric actions. We aim at providing a framework and a sound set of hypothesis under which a classical Bellman equation holds in the discounted case.

8.1 Hindsight on the TMDP model: what is the "wait" action?

The *wait* action defined in the previous chapter and in [Boyan and Littman, 2001] in order to allow for inactivity is defined as an overlay over the standard Bellman equations. Chapters 4 and 5 showed that it was actually equivalent to letting the system be idle for a while if the expected gain was better at a latter time. These sections actually highlighted the fact that in TMDPs, *wait* indeed was an action, but with some specificities, namely:

- no effect on the discrete part of the state space,
- deterministic with respect to its effects on the time variable,
- no reward when used with zero duration.

It actually seems that there is not one *wait* action but a whole continuum of these actions since equation 4.10 optimizes the waiting time. In other words, the action space is hybrid: it can be described by a set $A \cup \mathbb{R}^+$. All actions chosen inside this action space are either a waiting duration or a discrete action to undertake.

With this representation of the action space, *wait* does not differ anymore from any other continuous action. However, representing the action space as a $A \cup \mathbb{R}^+$ set becomes more complicated as the number of possible continuous actions grows. This is because the $\cup$ operator does not capture the action space's structure: there are several different high-level actions which differ strongly by nature - *eg.* $goforward$, *pickup*, *wait* - and each of these

actions corresponds to a continuous action subspace - *eg.* $goforward(l)$, $pickup(), wait(\tau)$.

MDPs are often defined with a finite action space, summarizing all the high-level actions an agent can undertake. But sometimes, even high-level actions need to be continuous: "invest an amount X of money", "go forward L meters", "inject A centiliters of drug 1", etc. are examples of such actions. In standard models, discretizing the action space implies associating a unique, fixed value to the parameters of each action and hence, restricting the agent's possibilities. Figure 8.1 illustrates the problem of discretizing the action space: an optimal policy in a discretized action space might miss a very reward obtained with an intermediate action parameter.

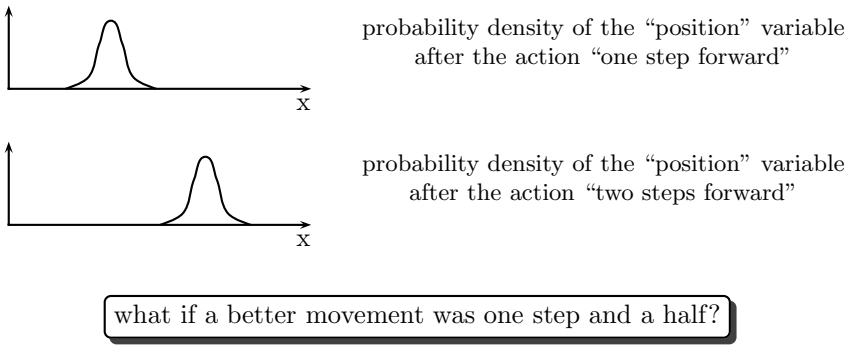


Figure 8.1: The problem of action discretization

This reasoning brings up the notion of *parametric action* which captures both the structural part of the action space (different high-level actions with different properties and meaning) and the parametric part (continuous or discrete parameters for each action).

Therefore, one needs to model the *wait* action of TMDPs as a parametric action — as the intuition indicated. Moreover, it seems that — even though *wait* plays a specific role with respect to the discounted criterion — MDPs could be extended to deal with parametric action spaces. This representation would allow for easy decoupling (when possible) of the discrete and the continuous part of the optimization process for the choice of the best action.

Continuous and hybrid state spaces have been addressed in the MDP literature from different points of view. Partially Observable MDP (POMDP, [Kaelbling et al., 1998]) describe a continuous bounded belief space over possible states. On the other hand, efficient partitioning of a continuous state space using kd-trees and dynamic programming was developed in [Feng et al., 2004] or [Li and Littman, 2005]. Approximating the continuous transition functions by phase-type distributions allowing for model simplification, was presented in [Marecki et al., 2006]. Recent contributions of [Hauskrecht and Kveton, 2006] and [Guestrin et al., 2004] use Approximate Linear Programming to solve MDPs defined on hybrid state spaces. However, all these approaches keep a discrete action space. Simple continuous action spaces were presented in the examples of [Puterman, 1994] and [Bertsekas, 1995]. They revisited problems introduced through the formalism of Controlled Markov Chains [Altman and Shwartz, 1993; Altman, 1999]. Recent advances in reinforcement learning and approximate dynamic programming such as [Hasselt and Wiering, 2007] also tackle the problem of solving decision problems with continuous or hybrid action variables.

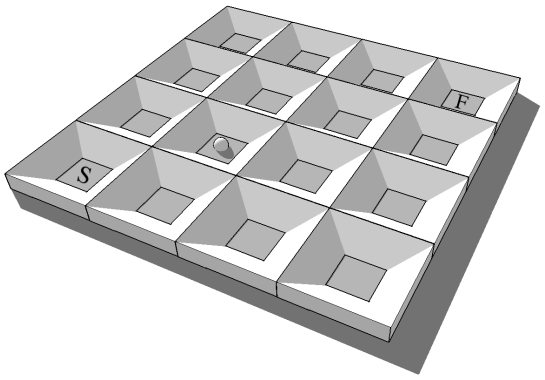


Figure 8.2: Illustrative example

The goal of the next paragraph is to generalize the Bellman equation for MDPs to the case of hybrid state and action spaces with observable time. As the proofs of section 8.3 will show, introducing an observable time is the main difficulty on the way to proving that the adapted Bellman equation for this broader class of problems is still valid. The results presented in sections 8.2 and 8.3 were first introduced in [Rachelson et al., 2008a].

8.2 A model with hybrid state and action spaces and with observable continuous time

8.2.1 Model definition

In order to illustrate the following definitions on a simple example, we propose the game presented in figure 8.2. In this game, the goal is to bring the ball from the start box to the finish box. Unfortunately, the problem depends on a continuous time variable because the boxes' floors retract at known dates and because actions durations are uncertain and real-valued. At each decision epoch, the player has five possible actions: he can either push the ball in one of the four directions or he can wait for a certain duration in order to reach a better configuration. Finally the "push" actions are uncertain and the ball can end up in the wrong box. This problem has a hybrid state space composed of discrete variables - the ball's position - and continuous ones - the current date[1]. It also has four non-parametric actions - the "push" actions - and one parametric action - the "wait" action. We are therefore trying to find a policy on a stochastic process with continuous and discrete variables and parametric actions (with real valued parameters). Keeping this example in mind, we introduce the notion of parametric MDP:

Definition (XMDP)**.** *A parametric action MDP, or XMDP, is a tuple* $\langle S, A(X), p, r \rangle$ *where:*

- S *is a Borel state space which can describe continuous or discrete state variables including the process' time.*
- A *is an action space describing a finite set of actions* $a_i(x)$ *where* x *is a vector of parameters taking its values in* X*. Therefore, the action space of our problem is a hybrid action space, factored by the different actions an agent can undertake. In practice, each action only depends on a subset of variables from* X*.*

[1]This illustrative example can actually be written as a TMDP. We draw from the TMDP experience to generalize to the broader framework of hybrid, parametric actions and hybrid states.

p is a probability density transition function $p(s'|s, a(x))$.

r is a reward function $r(s, a(x))$.

8.2.2 Emphasizing the place of time

As in the MDP case, we consider the set T of timed decision epochs. In order to deal with the more general case, we will consider a real-valued time variable t and — as previously — will write the state (s, t) in order to emphasize the specificity of this variable in the discounted case.

So it is important to note that — since we consider t as a state variable — we can split the random variable describing the current state into a vector with at least two components. To clarify this, let us reuse a previous SMDP+ notation and call σ any state in the state space. The first component of the σ vector is the time corresponding to the current state t_σ. The second component is the classical state value s_σ, aggregating any other state variable values which might be defined for the problem at hand.

This leads us to correct our notations: we call S the set of values taken by the vector of state variables but time. Hence, our state space will be written as the Borel algebra on the $S \times \mathbb{R}$ topological space and we will use pairs (s, t) to describe one state. To simplify the notations, we will refer to this state space as $S \times \mathbb{R}$.

Note that for discrete variables, the $p()$ function of the XMDP is a discrete probability distribution function and that writing integrals over $p()$ is equivalent to writing a sum over the discrete variables.

Concerning the time variable, if it is discrete and bounded, optimizing a policy for the XMDP corresponds to optimizing a finite horizon criterion. In the case of continuous observable time, section 8.4 will show that the TMDP and SMDP+ are subproblems of the XMDP framework. In order to summarize:

> XMDPs are a generalization of MDPs where hybrid action spaces are represented through the abstraction of parametric actions. Moreover, XMDPs include the process' time as a state variable, thus suppressing the distinction between stationary and non-stationary policies for MDPs.

Lastly, as previously, we will write δ the number of the current decision epoch, and, consequently, t_δ the time at which decision epoch δ occurs.

8.2.3 Reward model

One can comment that writing the reward model as $r((s, t), a(x))$ might not span the full genericity of the models we wish to represent. For instance, we could wish to write some $r((s, t), a(x), (s', t'))$ reward model in order to specify the reward of each transition.

In the following developments, we will nevertheless restrict ourselves to the $r((s, t), a(x))$ expected reward model for simplicity. Even though we did not write the proofs for the general case of $r((s, t), a(x), (s', t'))$, making a parallel with the standard MDP case — where

one can similarly establish equations for $r(s,a)$ or $r(s,a,s')$ — seems possible.

One interesting option for decomposing the reward model is the one inspired by SMDPs. A transition reward $r(s,a)$ in an SMDP is given as a lump sum reward $k(s,a)$ and a set of reward rates $c(j,s,a)$ where the j states correspond to all the intermediate states encountered during the transition and before the next decision epoch (for details on SMDPs reward models and underlying stochastic processes, one can refer to [Puterman, 1994], page 533). The reward model $r(s,a)$ then corresponds to the lump sum reward plus the discounted integral over possible durations of the reward rates in each state j visited by the underlying process.

The interesting point in such a reward model is that it separates the lump sum reward from the reward rates. For an XMDP, such a model would imply an important property for instantaneous transitions: for such transitions, the reward acquired is only the lump sum reward.

The TMDP type of reward model is then the extension to (s,a,s') transitions of such an SMDP reward model with lump sum reward in t, duration reward through a reward rate and lump sum reward in t'.

Even though these modelling options might be appealing because of their practical implications, they remain a special case of an $r((s,t),a(x))$ (or $r((s,t),a(x),(s',t'))$) reward model and we won't adopt them for further reasoning.

8.2.4 Policies and criterion

We define a deterministic Markovian *decision rule* at decision epoch δ as the mapping from states to actions:

$$d_\delta : \left\{ \begin{array}{ccc} S \times \mathbb{R} & \rightarrow & A(X) \\ s,t & \mapsto & a(x) \end{array} \right. \tag{8.1}$$

$d_\delta(s,t)$ specifies the parametric action to undertake in state (s,t) at decision epoch δ. A *policy* is defined as a set of decision rules (one for each δ) and we consider, as in [Puterman, 1994], the set $\mathcal{D}$ of stationary (with respect to δ) markovian deterministic policies.

Definition (Stationary, deterministic, Markovian policy)**.** *Such a control policy π is given by a single decision rule, applicable at each decision epoch. Hence we identify this decision rule and the policy:*

$$\pi : \left\{ \begin{array}{ccc} S \times \mathbb{R} & \rightarrow & A(X) \\ s,t & \mapsto & a(x) \end{array} \right. \tag{8.2}$$

For simplicity and without justification as to the relevance of this approach, we choose to search for policies in $\mathcal{D}$. Analysis of stochastic or non-Markovian policies for instance is beyond the scope of this chapter's results.

In order to evaluate policies in $\mathcal{D}$ for our problem, we need to define a criterion. Given the strong similarity of XMDPs and SMDPs, the discounted criterion we define follows the example of [Howard, 1963] and integrates the expected reward over all possible transition durations. We introduce the discounted criterion for XMDPs as the expected sum of the

successive discounted rewards, with respect to the application of policy π starting in state (s,t):

$$V_\gamma^\pi(s,t) = E^\pi_{(s_0=s,t_0=t)} \left\{ \sum_{\delta=0}^{\infty} \gamma^{t_\delta - t_0} r_\pi(s_\delta, t_\delta) \right\} \tag{8.3}$$

In order to make sure this series has a finite limit, our model introduces three more hypothesis:

- $|r((s,t),a(x))|$ is bounded by M,
- $\forall \delta \in T, \quad t_{\delta+1} - t_\delta \geq \alpha > 0$, where α is the smallest possible duration of an action ,
- $\gamma < 1$.

The discount factor γ^t insures the convergence of the series. Physically, it can be seen as a probability of still being functional after time t. With these hypothesis, one can write:

Lemma 1. *$\forall (s,t) \in S \times \mathbb{R}$:*

$$|V_\gamma^\pi(s,t)| < \frac{M}{1-\gamma^\alpha} \tag{8.4}$$

Proof. Since the transition durations are lower-bounded by α, one can write, for all $\pi \in \mathcal{D}$:

$$\begin{aligned} t_\delta &\leq t_0 + \delta\alpha \\ \gamma^{t_\delta} &\leq \gamma^{t_0+\delta\alpha} \\ \gamma^{t_\delta - t_0} &\leq \gamma^{\delta\alpha} \end{aligned}$$

And since $|r|$ is bounded by M we have:

$$\forall \pi \in \mathcal{D}, -\gamma^{\delta\alpha} M \leq \gamma^{t_\delta - t_0} r_\pi(s_\delta, t_\delta) \leq \gamma^{\delta\alpha} M$$

And finally:

$$\forall \pi \in \mathcal{D}, -\sum_{\delta=0}^{\infty} \gamma^{\delta\alpha} M \leq \sum_{\delta=0}^{\infty} \gamma^{t_\delta - t} r_\pi(s_\delta, t_\delta) \leq \sum_{\delta=0}^{\infty} \gamma^{\delta\alpha} M$$

Which finally provides the result of the lemma. □

We will further restrict the first assumption by saying that we only consider positive reward models, thus we write: $0 \leq r((s,t),a(x)) \leq M$. In the following discussion, we will highlight why and where this assumption is necessary.

The rather strong assumption of a lower bound on the duration of transitions will be important for the proof in section 8.3.2. We will see in section 8.3.3 how we can try to lift this hypothesis (and why it is not trivial).

However, it seems important to avoid a — somehow intuitive — misconception: this assumption is a constraint on the transition model and not on the policy search space. The mentioned misconception would be to believe that the constraint $t' - t \geq \alpha$ corresponds to forbidding any $wait(\tau)$ with $\tau < \alpha$ action. While this seems intuitively the case, it is not what is implied by this assumption: any $wait(\tau)$ action is possible (with respect to this lower bound only — we will see a little further that we might have to restrict the $A(X)$ space as well for other reasons) but their result will always yield a transition with $t' - t \geq \alpha$.

We will admit that the set $\mathcal{V}$ of value functions (functions from $S \times \mathbb{R}$ to $\mathbb{R}$) is a complete metrizable space for the supremum norm $\|V\|_\infty = \sup_{(s,t)\in S\times\mathbb{R}} V(s,t)$.

The optimal value function for Markovian, deterministic policies, given the discounted criterion is defined as $V^* = \sup_{\pi\in\mathcal{D}} V_\gamma^\pi$.

An optimal policy in $\mathcal{D}$, if it exists, is then defined as a policy π^* which verifies $V_\gamma^{\pi^*} = \sup_{\pi\in\mathcal{D}} V_\gamma^\pi$. Guaranteeing the existence of such a policy whose value actually reaches the sup requires three hypothesis which will be developed in section 8.3.4:

- The r and p models are upper semi-continuous with respect to the parameters x.
- The reward model is positive.
- $A(X)$ is a compact action subset of a topological space describing a finite set of actions $a_i(x)$ where x is a vector of parameters taking its values in X. Since we consider a finite set of a_i high-level actions, this hypothesis can be rephrased for $A_i(X)$: $A_i(X)$ is homeomorphic to a compact subset of a topological space describing the set of parameters admissible for the finite set of abstract actions a_i.

These three hypothesis guarantee that the value function will be upper semi-continuous and that there exists a vector x of finite parameters that reaches the sup of the value function. Such a proof was immediate in the classical MDP model because the action space was countable, the upper semi-continuity and compacity arguments allow the extension to the parametric action case.

We avoided including these additional hypothesis directly in the previous model definition because one can prove the existence of an optimal value function without it. However, this hypothesis is a sufficient condition to guarantee the existence of a policy whose value is equal to the optimal value function. So, we will specifically mention when we make these assumptions in the following proofs.

From here on we will omit the γ index on V.

8.2.5 Summarizing the XMDP's hypothesis

Based on the previous short discussion, we can summarize the assumptions upon which the XMDP framework is built. An XMDP problem is given by the $S \times \mathbb{R}$ and $A(X)$ sets, the p and r functions, the γ value and the set $\mathcal{D}$ of policies, following the assumptions:

Assumption 1 (State space structure)**.** *The state space is the Borel algebra defined on a topological space $S \times \mathbb{R}$. In order to simplify the notations, we will write $(s,t) \in S \times \mathbb{R}$ the elements of the state space.*

Assumption 2 (Action space structure)**.** *$A(x)$ is the union of a finite collection of sets $A_i(X)$, where each $A_i(X)$ is homeomorphic to a compact subset of a topological space X. Each element of $A(X)$ is noted $a_i(x)$ where a_i refers to the set $A_i(X)$ to which the element originally belongs and x is the corresponding element in $A_i(X)$. It represents the action set.*

Note that since the compacity assumption is not always necessary for all of the following proofs, we will specifically mention when we use or discard it.

Assumption 3 (Transition model vs. actions)**.** *p is a Markovian probability density function $p((s',t')|(s,t),a(x))$ which is upper semi-continuous with respect to x. It describes the transition model, from (s,t) to (s',t'), given action $a(x)$.*

Assumption 4 (Reward model vs. actions)**.** *r is a real-valued function $r((s,t),a(x))$ which is upper semi-continuous with respect to x. It is the reward model of undertaking action $a(x)$ in state (s,t).*

Assumption 5 (Bounded rewards)**.** *The reward model is bounded and positive:*

$$\exists M \in \mathbb{R}^+ / \forall (s,t,a(x)) \in S \times \mathbb{R} \times A(X), 0 \leq r((s,t),a(x)) \leq M$$

Assumption 6 (Time advance)**.** $\forall \left(a_n(x_n)\right)_{n\in\mathbb{N}} \in A(X)^{\mathbb{N}}, \forall \delta \in \mathbb{N}, \quad t_{\delta+1} - t_\delta \geq \alpha > 0$

On this basis, we look for a way of characterizing the optimal policy and value function. Namely, we wish to characterize the value of policies (section 8.3.1), to prove the existence of an optimality equation for V^* (section 8.3.2) and to relate this optimal value function to an optimal policy (section 8.3.4).

8.3 Extended Bellman equation

We introduce the policy evaluation operator L^π. Then we redefine the Bellman operator L for XMDPs and we prove that V^* is the unique solution to $V = LV$. Dealing with random decision times and parametric actions invalidates the proof of [Puterman, 1994], we adapt it and emphasize the differences in section 8.3.2.

8.3.1 Policy evaluation

The policy evaluation operator L^π provides the expected value function associated to applying π for the first step of execution and then receiving a reward corresponding to V.

Definition (L^π operator)**.** *The policy evaluation operator L^π maps any element V of $\mathcal{V}$ to the value function:*

$$L^\pi V(s,t) = r(s,t,\pi(s,t)) + \int\limits_{\substack{t'\in\mathbb{R}\\ s'\in S}} \gamma^{t'-t} p(s',t'|s,t,\pi(s,t)) V(s',t') ds' dt' \tag{8.5}$$

We note that for non-parametric actions and discrete state spaces, $p()$ is a discrete probability density function, the integrals turn to sums and the L^π operator above turns to the classical L^π operator for standard MDPs. This operator represents the one-step gain if we apply π and then get V. We now prove that this operator can be used to evaluate policies.

Proposition (Policy evaluation)**.** *Let π be a policy in $\mathcal{D}$. Then $V = V^\pi$ is the only solution of $L^\pi V = V$.*

Proof. In the following proofs $E^\pi_{a,b,c}$ denotes the expectation with respect to π, knowing the values of the random variables a, b and c. Namely, $E^\pi_{a,b,c}(f(a,b,c,d,e))$ is the expectation calculated with respect to d and e, and is therefore a function of a, b and c.

Our starting point is $(s_0, t_0) = (s, t)$:

$$\begin{aligned} V^\pi(s,t) &= E^\pi_{s_0,t_0}\left\{\sum_{\delta=0}^{\infty}\gamma^{t_\delta - t} r_\pi(s_\delta, t_\delta)\right\} \\ &= r_\pi(s,t) + E^\pi_{s_0,t_0}\left\{\sum_{\delta=1}^{\infty}\gamma^{t_\delta - t} r_\pi(s_\delta, t_\delta)\right\} \\ &= r_\pi(s,t) + E^\pi_{s_0,t_0}\left\{E^\pi_{\substack{s_0,t_0\\s_1,t_1}}\left(\sum_{\delta=1}^{\infty}\gamma^{t_\delta - t} r_\pi(s_\delta, t_\delta)\right)\right\} \end{aligned}$$

The inner mathematical expectation deals with random variables $(s_i, t_i)_{i=2\ldots\infty}$, the outer one deals with the remaining variables (s_1, t_1). We expand the outer expected value with $(s_1, t_1) = (s', t')$:

$$V^\pi(s,t) = r_\pi(s,t) + \int_{\substack{t'\in\mathbb{R}\\s'\in S}} E^\pi_{\substack{s_0,t_0\\s_1,t_1}}\left(\sum_{\delta=1}^{\infty}\gamma^{t_\delta - t} r_\pi(s_\delta, t_\delta)\right)\cdot p_\pi(s', t'|s, t)ds'dt'$$

$$V^\pi(s,t) = r_\pi(s,t) + \int_{\substack{t'\in\mathbb{R}\\s'\in S}} \gamma^{t'-t} p_\pi(s', t'|s, t)\cdot E^\pi_{\substack{s_0,t_0\\s_1,t_1}}\left(\sum_{\delta=1}^{\infty}\gamma^{t_\delta - t'} r_\pi(s_\delta, t_\delta)\right)ds'dt'$$

The expression inside the $E^\pi_{s_0,t_0,s_1,t_1}()$ deals with random variables (s_i, t_i) for $i \geq 2$. Because of the Markov property on the $p()$ probabilities, this expectation only depends on the (s_1, t_1) variables and thus:

$$E^\pi_{\substack{s_0,t_0\\s_1,t_1}}\left(\sum_{\delta=1}^{\infty}\gamma^{t_\delta - t} r_\pi(s_\delta, t_\delta)\right) = V^\pi(s', t')$$

And we have:

$$V^\pi(s,t) = L^\pi V^\pi(s,t) \tag{8.6}$$

The solution exists and is unique because L^π is a contraction mapping on $\mathcal{V}$ and we can use the Banach fixed point theorem (the proof of L^π being a contraction mapping is similar to the one we give for the L operator in the next section). □

The value function V^π corresponding to the expected value of applying policy π given an XMDP and the discounted criterion of equation 8.3 is the only solution of:

$$V^\pi(s,t) = r(s,t,\pi(s,t)) + \int_{\substack{t'\in\mathbb{R}\\s'\in S}} \gamma^{t'-t} p(s', t'|s, t, \pi(s,t))V^\pi(s', t')ds'dt' \tag{8.7}$$

In order to ease the notations, we introduce an operator which provides the expected reward of performing $a(x)$ in a given state (s, t) and then receiving value V:

Definition ($L^{a(x)}$ operator)**.** *The action evaluation operator $L^{a(x)}$ maps any element V of $\mathcal{V}$ to the value function:*

$$L^{a(x)}V(s,t) = r(s,t,a(x)) + \int_{\substack{t'\in\mathbb{R}\\s'\in S}} \gamma^{t'-t} p(s', t'|s, t, a(x))V(s', t')ds'dt' \tag{8.8}$$

One can note that this operator is consistent with the previous notation since $L^\pi V(s,t) = L^{a(x)=\pi(s,t)}V(s,t)$

8.3.2 Bellman operator

Introducing the L^π operator is the first step towards defining the dynamic programming operator L. This operator provides the value function corresponding to maximizing the reward of the first action in all states (hence finding a one-step optimizing policy) and then receiving V.

Definition (L operator)**.** *The Bellman dynamic programming operator L maps any element V of $\mathcal{V}$ to the value function LV.*
In function notation:

$$LV = \sup_{\pi \in \mathcal{D}} \{L^\pi V\} \tag{8.9}$$

And at the state level:

$$LV(s,t) = \sup_{a(x) \in A(X)} L^{a(x)} V(s,t)$$

$$LV(s,t) = \sup_{a(x) \in A(X)} \left\{ r(s,t,a(x)) + \int_{\substack{t' \in \mathbb{R} \\ s' \in S}} \gamma^{t'-t} p(s',t'|s,t,a(x)) V(s',t') ds' dt' \right\} \tag{8.10}$$

This operator represents the one-step look-ahead action optimization, in every state of the process, with respect to a value function V. We now prove that L defines the optimality equation allowing to characterize optimal policies for the discounted criterion (equation 8.3).

Proposition (Bellman equation)**.** *For an XMDP with a discounted criterion, the optimal value function is the unique solution of the Bellman equation $V = LV$.*

Proof. The proofs adapts [Puterman, 1994] to the XMDP hypotheses stated above. However, for this specific proof, we will not make use of some of the previous assumptions. The assumptions we lift for this proof are: the positive reward model assumption, the compact action space and the semi-continuous reward and transition models. These assumptions will prove necessary in section 8.3.4 to prove the existence of an optimal policy but are not necessary to prove the existence of an optimal value function. Our reasoning takes three steps:

1. We first prove that if $V \geq LV$ then $V \geq V^*$,
2. Then, we similarly prove that if $V \leq LV$ then $V \leq V^*$,
3. Lastly, we prove that there exists a unique solution to $V = LV$.

Suppose that we have a V such that $V \geq LV$. Therefore, with π a policy in $\mathcal{D}$, we have: $V \geq \sup_{\pi' \in \mathcal{D}} \left\{ L^{\pi'} V \right\} \geq L^\pi V$. Since L^π is positive[2], we have, recursively:

$$V \geq L^\pi V \geq L^\pi L^\pi V \ldots \geq L^{\pi(n+1)} V$$

We want to find a $N \in \mathbb{N}$ such that $\forall n \geq N, \quad L^{\pi(n+1)} V - V \geq 0$.

[2]Reminder: The concept of spectrum of an operator is the generalization of eigenvalues for matrices to infinite dimensional spaces. A positive operator has all its spectrum values positive.

$L^{\pi(n+1)}V$ corresponds to the reward we get for applying policy π for $n+1$ steps and then getting reward V.

$$L^{\pi(n+1)}V = r_\pi(s_0,t_0) + E^\pi_{s_0,t_0}\Bigg(\gamma^{t_1-t_0}r_\pi(s_1,t_1) \; + E^\pi_{s_1,t_1}\Bigg(\gamma^{t_2-t_0}r_\pi(s_2,t_2) + E^\pi_{s_2,t_2}\Bigg(\ldots + E^\pi_{s_{n-1},t_{n-1}}\Bigg(\gamma^{t_n-t_0}r_\pi(s_n,t_n) \; + E^\pi_{s_n,t_n}\Bigg(\gamma^{t_{n+1}-t_0}V(s_{n+1},t_{n+1})\Bigg)\Bigg)\ldots\Bigg)\Bigg)\Bigg)$$

$$V^\pi = r_\pi(s_0,t_0) + E^\pi_{s_0,t_0}\Bigg(\gamma^{t_1-t_0}r_\pi(s_1,t_1) \; + E^\pi_{s_1,t_1}\Bigg(\gamma^{t_2-t_0}r_\pi(s_2,t_2) + E^\pi_{s_2,t_2}\Bigg(\ldots + E^\pi_{s_{n-1},t_{n-1}}\Bigg(\gamma^{t_n-t_0}r_\pi(s_n,t_n) \; + E^\pi_{s_n,t_n}\Bigg(\sum_{\delta=n+1}^{\infty}\gamma^{t_\delta-t_0}r_\pi(s_\delta,t_\delta)\Bigg)\Bigg)\ldots\Bigg)\Bigg)\Bigg)$$

When writing $L^{\pi(n+1)}V - V^\pi$ one can merge the two expressions above in one big expectation over all random variables $(s_i,t_i)_{i=0\ldots\infty}$. Then all the first terms cancel each other and we can write:

$$L^{\pi(n+1)}V - V^\pi = E^\pi_{\substack{(s_i,t_i)\\ i=0\ldots n}}\Bigg(\gamma^{t_{n+1}-t_0}V(s_{n+1},t_{n+1}) \; - \sum_{\delta=n+1}^{\infty}\gamma^{t_\delta-t_0}r_\pi(s_\delta,t_\delta)\Bigg)$$

and thus:

$$L^{\pi(n+1)}V - V^\pi = E^\pi_{\substack{(s_i,t_i)\\ i=0\ldots n}}\Bigg(\gamma^{t_{n+1}-t_0}V(s_{n+1},t_{n+1})\Bigg) \; - E^\pi_{\substack{(s_i,t_i)\\ i=0\ldots n}}\Bigg(\sum_{\delta=n+1}^{\infty}\gamma^{t_\delta-t_0}r_\pi(s_\delta,t_\delta)\Bigg)$$

We write: $L^{\pi(n+1)}V - V^\pi = q_n - r_n$.

Since $t_{n+1}-t_n \geq \alpha > 0$, $(t_n)_{n\in\mathbb{N}}$ is a divergent sequence. So this last assumption (without the lower bound α) would be enough to insure that $\gamma^{t_{n+1}}$ tends to zero.

Lemma 1 indicates that V is bounded by $\|V\|$, γ^{-t_0} is a constant and $\left(\gamma^{t_{n+1}}\right)_{n\in\mathbb{N}}$ is a sequence converging to zero so we have:

$$\gamma^{t_{n+1}-t_0}V(s_{n+1},t_{n+1}) \; \leq \; \gamma^{t_{n+1}}\gamma^{-t_0}\|V\|$$

And we can write $\lim\limits_{n\to\infty} q_n = 0$.

On the other hand, r_n is the remainder of a convergent series (the criterion). Thus we have: $\lim\limits_{n\to\infty} r_n = 0$.

So $\lim\limits_{n\to\infty} L^{\pi(n+1)}V - V^\pi = 0$.

We had $V \geq L^{\pi(n+1)}V$, so $V - V^\pi \geq L^{\pi(n+1)}V - V^\pi$. The left hand side expression doesn't depend on n and since the right hand side expression's limit is zero, we can write: $V - V^\pi \geq 0$.

Since this is true for any $\pi \in \mathcal{D}$, it remains true for the superior bound of the value functions over all $\pi \in \mathcal{D}$:

$$V \geq LV \Rightarrow V \geq V^*$$

We can follow a similar reasoning for $V \leq LV$. If $V \leq LV$, then there exists $\pi \in \mathcal{D}$ such that $V \leq L^{\pi}V \leq LV$. Therefore $V \leq L^{\pi(n+1)}V$ and $V - V^{\pi} \leq L^{\pi(n+1)}V - V^{\pi}$. With the same argument as previously, we obtain $V - V^{\pi} \leq 0$. Since $V^{\pi} \leq V^{*}$, we have:

$$V \leq LV \Rightarrow V \leq V^{*}$$

The two previous assertions show that if a solution to $V = LV$ exists, then this solution is equal to V^{*}.

In order to finish proving the proposition, we need to prove that there always is a solution to $V = LV$.

According to [Bertsekas and Shreve, 1996] $\mathcal{V}$ is a metrizable space, complete for the supremum norm $\|V\|_{\infty} = \sup_{(s,t)\in S\times\mathbb{R}} V(s,t)$. If we show that L is a contraction mapping in $\mathcal{V}$, then we will be able to apply Banach fixed point theorem.

Let us fix $(s,t) \in S \times \mathbb{R}$ and let U and V be two elements of $\mathcal{V}$ with $LV(s,t) \geq LU(s,t)$ (we suppose $LV(s,t) \geq LU(s,t)$ only to simplify the writing and will perform the same proof with $LV(s,t) \leq LU(s,t)$ later).

We have:

$$|LV(s,t) - LU(s,t)| = LV(s,t) - LU(s,t)$$

Since we have withdrawn the assumption that $A(X)$ is compact and that the reward model is positive, we cannot guarantee the existence of an $a(x)$ which reaches this sup for V. However, there exists a sequence $(a_n(x_n))_{n\in\mathbb{N}}$ of elements of $A(X)$ such that:

$$\lim_{n\to\infty} L^{(a_n(x_n))}V(s,t) = LV(s,t)$$

Let us consider the sequence $\left(L^{(a_n(x_n))}U(s,t)\right)_{n\in\mathbb{N}}$. This real-valued sequence is bounded since value functions are bounded. Consequently, we can make use of Bolzano-Weierstrass' theorem and extract a convergent sequence from $\left(L^{(a_n(x_n))}U(s,t)\right)_{n\in\mathbb{N}}$.

Let us call j the subscripts of this extracted sequence and $(a_j(x_j))_{j\in\mathbb{N}}$ be the sequence of actions obtained by extracting elements of $(a_n(x_n))_{n\in\mathbb{N}}$ correspondingly.

We can now insure the limits exist and write:

$$\lim_{j\to\infty} L^{(a_j(x_j))}V(s,t) = LV(s,t)$$
$$\lim_{j\to\infty} L^{(a_j(x_j))}U(s,t) \leq LU(s,t)$$

So:

$$\begin{aligned} LV(s,t) - LU(s,t) &\leq \lim_{j\to\infty} L^{(a_j(x_j))}V(s,t) - \lim_{j\to\infty} L^{(a_j(x_j))}U(s,t) \\ &\leq \lim_{j\to\infty} \left[L^{(a_j(x_j))}V(s,t) - L^{(a_j(x_j))}U(s,t)\right] \end{aligned}$$

And:

$$LV(s,t) - LU(s,t) \leq \lim_{j\to\infty} \left[r(s,t,a_j(x_j)) \; + \int\limits_{\substack{t'\in\mathbb{R}\\ s'\in S}} \gamma^{t'-t} p_{a_j(x_j)}(s',t'|s,t) V(s',t') ds'dt' \; - \right.$$

$$\left. r(s,t,a_j(x_j)) \; - \int\limits_{\substack{t'\in\mathbb{R}\\ s'\in S}} \gamma^{t'-t} p_{a_j(x_j)}(s',t'|s,t) U(s',t') ds'dt' \right]$$

Which yields:

$$LV(s,t) - LU(s,t) \leq \lim_{j\to\infty} \int\limits_{\substack{t'\in\mathbb{R}\\ s'\in S}} \gamma^{t'-t} p_{a_j(x_j)}(s',t'|s,t) \cdot \big(V(s',t') - U(s',t')\big)\, ds'dt'$$

But concerning the term inside the limits, we have: $\begin{cases} V(s',t') - U(s',t') \leq \|V-U\| \\ t'-t \geq \alpha > 0 \\ \int_{\substack{t'\in\mathbb{R}\\ s'\in S}} p(s',t'|s,t,a_j(x_j)) \leq 1 \\ \gamma < 1 \end{cases}$,

so we can write:

$$\int\limits_{\substack{t'\in\mathbb{R}\\ s'\in S}} \gamma^{t'-t} p_{a_j(x_j)}(s',t'|s,t) \cdot \big(V(s',t') - U(s',t')\big)\, ds'dt' \leq \|V-U\| \cdot \gamma^{\alpha}$$

Each of the terms in the $\left(\int\limits_{\substack{t'\in\mathbb{R}\\ s'\in S}} \gamma^{t'-t} p_{a_j(x_j)}(s',t'|s,t) \cdot \big(V(s',t') - U(s',t')\big)\, ds'dt' \right)_{j\in\mathbb{N}}$ sequence is lower or equal to $\|V-U\| \cdot \gamma^{\alpha}$, so this remains valid for their limit and we can finally write:

$$LV(s,t) - LU(s,t) \leq \|V-U\| \cdot \gamma^{\alpha}$$

The same argument for all $(s,t) \in S \times \mathbb{R}$ such that $LV(s,t) \leq LU(s,t)$ yields:

$$\|LV - LU\| \leq \|V-U\| \cdot \gamma^{\alpha}$$

Since $\gamma^{\alpha} < 1$, this proves L is a contraction mapping on $\mathcal{V}$. Banach fixed point theorem then tells us that there exists a fixed point $V' \in \mathcal{V}$ to the L operator such that $V' = LV'$.

The previous results allow us to conclude that under the general hypothesis mentioned above, the equation $LV = V$ has a unique solution and this solution is equal to V^*, the optimal value function with respect to the discounted criterion.

$$LV = V \Rightarrow V = V^* \tag{8.11}$$

□

Given an XMDP and the discounted criterion of equation 8.3, the optimal policy's value function V^* is the only solution, for all $(s,t) \in S \times \mathbb{R}$, of:

$$V^*(s,t) = \sup_{a(x) \in A(X)} \left\{ r(s,t,a(x)) + \int_{\substack{t' \in \mathbb{R} \\ s' \in S}} \gamma^{t'-t} p(s',t'|s,t,a(x)) V^*(s',t') ds' dt' \right\} \tag{8.12}$$

8.3.3 Lifting some of the previous assumptions

We wrote earlier that some of the assumptions listed in section 8.2.5 were quite strong and that it might be desirable to try to remove them. These assumptions are the positivity of the reward model, the compacity of the action space and especially the lower bound α on the transition durations.

Let us start with this last assumption. From the execution point of view, we wish to avoid any sequence of decisions which might have non-zero duration but a finite cumulative duration. In other words, in practice, we want our system's time to move forward for all control policies and to avoid being "trapped" before a certain horizon. For example, we want to avoid any sequence of decision epoch's dates of the type $t_n = \frac{t_{max}+t_{n-1}}{2}$. This could be obtained by a policy like $\pi(t) = wait(\frac{t_{max}-t}{2})$ in a simple XMDP with deterministic *wait* for instance. Therefore we wish to guarantee we will not be facing such situations which lead to some kind of temporal "Zeno's paradox".

For this purpose, an assumption of the form $\lim_{\delta \to \infty} t_\delta = \infty$ seems sufficient and not as restrictive as imposing a strictly positive lower bound. However, if this softer hypothesis guarantees that the execution moves forward in time, it does not allow to maintain the previous proof. Before explaining where the problem is in the proof, let us formalize this assumption a little more to illustrate why it does not really provide more genericity than the "α" lower bound.

A first way of writing this assumption formally is to consider that it affects the XMDP model itself. This way, the assumption would be written:

Assumption 7. *We suppose that the transition model p of the XMDP is such that:*

$$\forall\, (a_n(x_n)) \in A(X)^{\mathbb{N}}, \lim_{\delta \to \infty} t_\delta = \infty$$

This puts hard constraints on the transition model since, for instance, to satisfy this assumption, the behavior of a sequence of actions generated by the previous example's policy $\pi(t) = wait(\frac{t_{max}-t}{2})$, would conflict with the intuitive transition model of *wait*.

Another possibility to formulate this assumption is to restrict the search space even more and say that:

Assumption 8. *Given the XMDP's transition and reward models, we look for the optimal policy among the set of policies:*

$$\left\{ \pi \in \mathcal{D} | \lim_{\delta \to \infty} t_\delta = \infty \right\}$$

Here again, this assumption puts many constraints on the search space and conflicts with the intuition that the model itself should exhibit a "natural" behavior, independently of the policy search space.

An interesting way of lifting this assumption builds on the idea that some actions might not be available in every state of the process. The subsets of actions available in state (s, t) can be written $A_{s,t}$. This adds an extra structuring element in the XMDP model and would allow to introduce a more natural assumption such as:

Assumption 9. *Given the XMDP's transition and reward models, we suppose that the available action sets $A_{s,t}$ insure that:*

$$\forall\,(a_n(x_n)) \in A(X)^{\mathbb{N}}, \lim_{\delta\to\infty} t_\delta = \infty$$

In more practical words, this last assumption expresses the fact that whenever the process enters a sequence of actions whose cumulative duration would lead to a "contracting time", it will eventually reach one of the states having an $A_{s,t}$ such that the process leaps forward in time and exits such a "dead-end" sequence.

These first attempts at removing this lower bound on the transition durations are all destined to insure that our model describes a divergent sequence of decision epochs (one could try to consider an example where a contracting time has physical meaning but such an application requiring a convergent time might not be very common). However there is another way of looking at this problem, which is to consider that we imposed this lower bound to guarantee that our criterion exists and to make the previous proofs.

Here again, having a simply divergent set of decision epoch's dates is not sufficient. One could build the counter-example of a situation where the sequence of actions applied is $a_n(x_n) = wait(\ln(1 + \frac{1}{n}))$ in an XMDP with a deterministic *wait* action a discount criterion γ of 0.9 and a reward model of $+1$ at each step. The sequence of decision epochs induced is $t_n = \ln(n + 1)$ which tends to ∞ while the expected reward of such a sequence of actions tends to ∞ as well.

Moreover, during the previous proof, we took care of mentioning where the $\lim_{\delta\to\infty} t_n = \infty$ hypothesis was sufficient in place of the $t_{\delta+1} - t_\delta \geq \alpha$ assumption. However, at the end of the proof, in order to show that L is a contraction mapping, we needed to bound $\gamma^{t'-t}$ by a factor strictly smaller than 1. This required bounding the difference $t' - t$. One could try to lift the "α" hypothesis by writing that a finite number — at most K — of several consecutive transitions will have a cumulative duration of more than alpha, thus allowing a finite number (less than K) of transition durations to be less than alpha and rewriting the proof to show that L^K (instead of L) is a contraction mapping.

This previous idea needs also to be related to the question of convergence of a Bellman equation in the MDP case with a total reward criterion since our discount factor tends to one when the transition duration tends to zero. In this case, the contraction property of L is more complicated to establish: while a single application of L might not be a contraction mapping, a repeated application (as L^K for instance) might recover this property.

This parallel actually provides two new ways of lifting the lower bound on transition durations. The first one is the K idea mentioned above: it states that for any policy π and any value α, there exists $K \in \mathbb{N}$ such that $\forall\delta \in \mathbb{N}, t_{\delta+K} - t_\delta > \alpha$:

Assumption 10. *We suppose that the transition model and the $A_{s,t}$ set (if they exist) insure that:*

$$\forall \pi \in \mathcal{D}, \forall \alpha \in \mathbb{R}^+, \exists K \in \mathbb{N} \ / \ \forall \delta \in \mathbb{N}, t_{\delta+K} - t_\delta > \alpha$$

This assumption lifts the α bound for individual transitions but keeps it for a sequence of K transitions.

Note that while the previous assumptions were designed for any sequence of actions $(a_n(x_n)) \in A(X)^{\mathbb{N}}$, this last one stands only for the sequences of actions induced by the application of π. While this seems to be a lighter assumption, it remains very difficult to verify in practice.

The last possibility we consider is a completely different approach which is based on a restriction of the reward model. If the reward obtained as the transition duration tends to zero, also tends to zero with the same convergence rate, we might be able to guarantee again the convergence of the criterion. This is relatively easy to guarantee if one looks at an SMDP-like reward model since it corresponds to having a zero lump sum reward and only a reward rate. However, this hypothesis restricts us to only a fraction of the models which could be described otherwise by XMDPs. We call this assumption the "no lump sum reward" assumption:

Assumption 11. *The reward model has only reward rates and no lump sum rewards. Moreover, the reward $r((s,t),a(x))$ associated with a transition is a discounted reward as in the SMDP case:*

$$r(s,a) = 0 + \int_0^\infty \sum_{j \in S} \left[\int_0^u \gamma^t c(j,s,a) p(j|t,s,a) dt \right] F(du|s,a) \tag{8.13}$$

This equation is to be related with equation 2.26 with no lump sum reward.

Hence, our conclusion for this "lower bound on durations assumption" is that there exist ways of lifting it in some specific cases which can occur quite often in practice. However, establishing the proof for a general setting of XMDPs without the lower bound assumption of transition duration (or without adding another hypothesis on the reward model) seems more difficult.

The next section discusses the origin of the "reward model positivity", "upper semi-continuity of the reward and transition models" and "compacity of the action space" assumptions.

8.3.4 Existence of an optimal policy

In order to illustrate why the upper semi-continuity and positivity hypotheses are necessary, we can try to prove the following lemma which makes use of these assumptions.

Lemma 2. $\forall \pi \in \mathcal{D}, a(x) \in A(X),\ L^{a(x)}V^\pi$ *is upper semi-continuous in x.*

Proof. Because the reward model is positive, $V^\pi(s',t')$ is necessarily positive.
Since $\gamma^{t'-t}$ is also positive, one can write:
For all $(s,t,s',t') \in S \times \mathbb{R} \times S \times \mathbb{R}$, the function

$$g_{(s,t,s',t')}(a(x)) = \gamma^{t'-t} p(s',t'|s,t,a(x)) V^\pi(s',t')$$

is upper semi-continuous in x and so is its sum with respect to s' and t'. Since $r(s,t,a(x))$ is also upper semi-continuous in x, the $L^{a(x)}V^{\pi}(s,t)$ function of x is upper semi-continuous with respect to x. □

Then, if one wishes to find the $\sup_{a(x)\in A(X)} L^{a(x)}V^{\pi}(s,t)$ and if this sup corresponds to a discontinuity point of $L^{a(x)}V^{\pi}(s,t)$ with respect to $a(x)$, *inside* $A(X)$, then the upper semi-continuity property of lemma 2 insures that there is an $a(x)$ which reaches this sup.

However, this sup value can correspond to an upper bound on the reachable values at the border of the $A(X)$ action space. The assumption of compacity for $A(X)$ (which is, indeed, an assumption of compacity of each $A_i(X)$) is then a sufficient condition to guarantee that there is always an $a(x)$ which reaches this sup.

Consequently, the three assumptions of "reward model positivity", "upper semi-continuity of the reward and transition models" and "compacity of the action space" constitute a sufficient set of assumptions to guarantee that this sup is a max.

Since $V^* = \sup_{\pi\in\mathcal{D}} V^{\pi}$, V^* is also positive. This last result insures that for any state (s,t), there exists an optimal action $a^*(x^*)$ defined by:

$$a^*(x^*) = \underset{a(x)\in A(\mathbb{R})}{argsup} \left\{ r(s,t,a(x)) \; + \int_{\substack{t'\in\mathbb{R}\\ s'\in S}} \gamma^{t'-t} p_{a(x)}(s',t'|s,t) V^*(s',t') ds'dt' \right\} \tag{8.14}$$

Finally, we were able to prove the existence of an optimal value function without these assumptions but still require them to guarantee the existence of an optimal policy.

8.3.5 Parametric formulation of Dynamic Programming

One can rewrite the previous Bellman equation in the following way, making it more suitable for dynamic programming algorithms such as value or policy iteration:

$$LV(s,t) = \max_{a\in A} \sup_{x\in X} \left\{ r(s,t,a(x)) \; + \int_{\substack{t'\in\mathbb{R}\\ s'\in S}} \gamma^{t'-t} p(s',t'|s,t,a(x)) V(s',t') ds'dt' \right\} \tag{8.15}$$

Using this formulation, we alternate:

1. an optimization on x of each action's value, providing the optimal parameter's value per action,
2. a choice among the (discrete) set of possible actions (with their optimal parameters).

For a brief example giving the flavor of the next section, we can imagine a problem with a single continuous time variable factoring a discrete state space and a single continuous duration parameter τ affecting only the "wait" action. This is a generalized TMDP setup (TMDP without the hypothesis on *wait*). Then equation 8.15 can be straightforwardly implemented as a two-step value iteration algorithm. The first step calculates the optimal value

of τ for any action that depends on it. The second step is a maximization step over all actions with their optimal parameter. This naive example shows the difficulties we can expect form designing algorithms to solve XMDPs. These difficulties deal with representing the continuous functions of the model's dynamics, solving the integrals in the Bellman equation and representing the continuous part of the policy. These problems have been encountered more generally when dealing with continuous variables in MDPs and various solutions for representing / approximating value functions have been proposed in [Boyan and Littman, 2001; Liu and Koenig, 2006; Li and Littman, 2005; Marecki et al., 2006; Hauskrecht and Kveton, 2006].

One can notice that if the state space is discrete, all probability density functions are discrete and integrals turn to sums. If the parameter space is discrete as well, by re-indexing the actions in the action space, the sup operator turns to a max and the above Bellman equation (equation 8.11) is the standard dynamic programming equation characterizing the solutions of classical MDPs. Therefore we can conclude that the XMDP model and its optimality equation includes and generalizes previous results for standard MDPs.

8.4 Back to the TMDP framework

We have introduced the XMDP framework on the intuition that hybrid actions could be efficiently represented via parametric objects. This raised the problem of a continuous time variable in the optimality equation and thus called for an extension of Bellman's equation. This framework was inspired by the TMDP's *wait* action. We now need to check that the TMDP problem can be written in the framework of parametric actions. Let us start by identifying the elements of both formalisms:

- The state is composed of the $(s,t) \in S \times \mathbb{R}$ variables as defined in the TMDP definition of sections 2.2 or 4.5.1. Let us redefine the notations and write s_d the discrete part of the TMDP state space (S_d being the set of discrete states) and s_t the time variable (S_t being its definition set), thus we write $s = (s_d, s_t)$ and S the complete state space.

- The action space includes all discrete actions which are independent of the parameter's vector. For each discrete action a_i, we actually have $a_i(x) = a_i$. We add a single parametric action to this action set: the $wait(\tau)$ action, with τ the duration parameter. The parameter space is therefore composed of a single variable $x = \tau$ and $X = \mathbb{R}^+$.

- For discrete actions, we have:

$$p(s'|s, a(\tau)) = \sum_{\mu_{s'_d}} L(\mu_{s'_d}|s_d, a, s_t) \cdot P_{\mu_{s'_d}}(s'_t - s_t)$$

  $\mu_{s'_d}$ being the set of outcomes μ reachable from state s and reaching s'_d. Here, the ABS and REL cases only have a modeling importance: in both cases it is the transition duration that is defined for a given outcome. If needed, one could replace $P_{\mu_{s'_d}}(s'_t - s_t)$ by $P_{\mu_{s'_d}}(s'_t)$ in order to get back to the ABS case. Since the *wait* action is considered deterministic and stationary, one can write:

$$P(s'|s, wait(\tau)) = \begin{cases} 1 & \text{if } s'_d = s_d \text{ and } s'_t = s_t. \\ 0 & \text{else} \end{cases}$$

  In other words:

$$p(s'|s, wait(\tau)) = \delta_{(s_d, s_t+\tau)}(s')$$

- Finally, the reward model is defined as previously for discrete actions:

$$r(s, a(\tau)) = r_t(\mu_{s'_d}, s_t) + \int_{s' \in S} p(s'|s, a(\tau)) \left[r_{t'}(\mu_{s'_d}, s'_t) + r_\tau(\mu_{s'_d}, s'_t - s_t) \right] ds' \quad (8.16)$$

And for the parametric *wait* action, one has: $r_\tau(\mu_{s'_d}, y) = \int_{s_t}^{s_t+y} K(s_d, \theta) d\theta$, $r_t(\mu_{s'_d}, s_t) = 0$ and $r_{t'}(\mu_{s'_d}, s'_t) = 0$, and thus:

$$r(s, wait(\tau)) = \int_{s_t}^{s_t+\tau} K(s_d, \theta) d\theta$$

One could note that a natural partition in three parts of the state space arises: first the set of discrete variables S_d which yields discrete probability distributions in the transition model, then the set of continuous variables S_c which is empty for TMDP problems and finally the temporal variable which takes its values from the set S_t and which feeds the special $\tau(s'_t - s_t)$. This natural decomposition results from the abstraction of the three different aspects of XMDP problems: discrete, continuous and temporal.

It is also important to note that, if one tries to relate the TMDP hypotheses to the discussion in section 8.3.3, TMDPs fall under the "infinitesimal lump sum reward for infinitesimal duration transitions" case for which we admit that the Bellman equation still holds.

Similarly, one can notice that the parameter space X is not upper bounded which can be a problem in order to find an optimal policy. In practice, this has little impact because the resolution scheme of TMDPs is a value iteration algorithm, looking for the optimal value function before inferring the optimal policy. For this optimal policy, one can consider that we are looking at the extended real number line for the parameter set: $X = \overline{\mathbb{R}}$, hence allowing a waiting time until $t = \infty$[3].

The TMDP model can thus be mapped into the framework of parametric actions. The main problem to address is to identify the optimality equations of the TMDP and XMDP models. We wish to solve equation 8.15 with the TMDP hypotheses:

$$\begin{aligned}
V^*(s) &= \max_{a \in A} \left\{ \sup_{\tau \in \mathbb{R}^+} \left(r(s, a(\tau)) + \int_{s' \in S} \gamma^{s'_t - s_t} V^*(s') p(s'|s, a(\tau)) ds' \right) \right\} \\
&= \max_{a \in A} \left\{ \sup_{\tau \in \mathbb{R}^+} \left(r(s, a(\tau)) + \int_{s' \in S} \gamma^{s'_t - s_t} V^*(s') \sum_{\mu_{s'_d}} L(\mu_{s'_d}|s_d, a, s_t) \cdot P_{\mu_{s'_d}}(s'_t - s_t) ds' \right) \right\} \\
&= \max_{a \in A} \left\{ \sup_{\tau \in \mathbb{R}^+} \left(r(s, a(\tau)) + \iint_{\substack{s'_d \in S_d \\ s'_t \in S_t}} \gamma^{s'_t - s_t} V^*(s') \sum_{\mu_{s'_d}} L(\mu_{s'_d}|s_d, a, s_t) \cdot P_{\mu_{s'_d}}(s'_t - s_t) ds'_d ds'_t \right) \right\} \\
&= \max_{a \in A} \left\{ \sup_{\tau \in \mathbb{R}^+} \left(r(s, a(\tau)) + \sum_{s'_d \in S_d} L(\mu_{s'_d}|s_d, a, s_t) \int_{s'_t \in S_t} \gamma^{s'_t - s_t} V^*(s') \cdot P_{\mu_{s'_d}}(s'_t - s_t) ds'_t \right) \right\}
\end{aligned}$$

[3]This actually provides a formal explanation to the fact that TMDP policies are equal to *wait* after the pseudo-horizon.

In the TMDP framework, we had $\gamma = 1$; we also admit the extension of the previous optimality equation to the $\gamma = 1$ case. So[4]:

$$V^*(s) = \max_{a \in A} \left\{ \sup_{\tau \in \mathbb{R}^+} \left(r(s, a(\tau)) + \sum_{s'_d \in S_d} L(\mu_{s'_d} | s_d, a, s_t) \int_{s'_t \in S_t} P_{\mu_{s'_d}}(s'_t - s_t) V^*(s') ds'_t \right) \right\}$$

One can separate the *wait* action from the discrete actions. Let us write $A^- = A \setminus \{wait\}$:

$$V^*(s) = max \left\{ \max_{a \in A^-} \left\{ \sup_{\tau \in \mathbb{R}^+} \left(r(s, a(\tau)) + \sum_{s'_d \in S_d} L(\mu_{s'_d} | s_d, a, s_t) \int_{s'_t \in S_t} P_{\mu_{s'_d}}(s'_t - s_t) V^*(s') ds'_t \right) \right\}, \right.$$
$$\left. \sup_{\tau \in \mathbb{R}^+} \left(r(s, wait(\tau)) + \sum_{s'_d \in S_d} L(\mu_{s'_d} | s_d, wait(\tau), s_t) \int_{s'_t \in S_t} P_{\mu_{s'_d}}(s'_t - s_t) V^*(s') ds'_t \right) \right\}$$

So:

$$V^*(s) = max \left\{ \max_{a \in A^-} \left\{ \sup_{\tau \in \mathbb{R}^+} \left(r(s, a(\tau)) + \sum_{s'_d \in S_d} L(\mu_{s'_d} | s_d, a, s_t) \int_{s'_t \in S_t} P_{\mu_{s'_d}}(s'_t - s_t) V^*(s') ds'_t \right) \right\}, \right.$$
$$\left. \sup_{\tau \in \mathbb{R}^+} \Big(r(s, wait(\tau)) + V^*(s_d, s_t + \tau) \Big) \right\}$$

The static nature of *wait* (no change of the discrete part of the state during waiting) implies that one cannot have two successive *wait* actions resulting from the optimality equation. We prove this by contradiction by considering the quantity $\sup_{\tau \in \mathbb{R}^+}(\ldots)$. We show that in a given state (s, t), if the policy specifies two successive actions $wait(\tau_1)$ and $wait(\tau_2)$ with τ_1 and τ_2 non null, then there exists an action $wait(\tau_1 + \tau_2)$ with an expected reward higher than $wait(\tau_1)$ in (s, t). Therefore $wait(\tau_1)$ does not correspond to $\underset{\tau \in \mathbb{R}^+}{argsup}(\ldots)$.

On top of that, the reward associated with a zero duration waiting is null ($r(s, wait(0)) = 0$). Hence, we can consider the execution of the policy as a sequence of actions alternating *wait* and discrete actions. This specificity of TMDPs allows to write:

$$V^*(s) = \sup_{\tau \in \mathbb{R}^+} \left(r(s, wait(\tau)) + \max_{a \in A \setminus \{wait\}} \left\{ r(s, a(\tau)) + \sum_{s'_d \in S_d} L(\mu_{s'_d} | s_d, a, s_t) \cdot \right. \right.$$
$$\left. \left. \int_{s'_t \in S_t} P_{\mu_{s'_d}}(s'_t - s_t) V^*(s') ds'_t \right\} \right) \quad (8.17)$$

But the reward model of equation 8.16 can be decomposed into:

[4] As mentioned earlier, we consider implicit the difference between REL and ABS cases. One could, if needed, replace $P_{\mu_{s'_d}}(s'_t - s_t)$ by $P_{\mu_{s'_d}}(s'_t)$.

$$
\begin{aligned}
r(s, a(\tau)) &= r_t(\mu_{s'_d}, s_t) + \int_{s' \in S} p(s'|s, a(\tau)) \left[r_{t'}(\mu_{s'_d}, s'_t) + r_\tau(\mu_{s'_d}, s'_t - s_t) \right] ds' \\
&= \int_{s' \in S} p(s'|s, a(\tau)) \left[r_t(\mu_{s'_d}, s_t) + r_{t'}(\mu_{s'_d}, s'_t) + r_\tau(\mu_{s'_d}, s'_t - s_t) \right] ds' \\
&= \sum_{s'_d \in S_d} L(\mu_{s'_d}|s_d, a, s_t) \int_{s'_t \in S_t} P_{\mu_{s'_d}}(s'_t - s_t) \left[r_t(\mu_{s'_d}, s_t) + r_{t'}(\mu_{s'_d}, s'_t) + r_\tau(\mu_{s'_d}, s'_t - s_t) \right] ds'_t
\end{aligned}
$$

Hence, equation 8.17 can be written:

$$
\begin{aligned}
V^*(s) = \sup_{\tau \in \mathbb{R}^+} \Bigg(& r(s, wait(\tau)) + \max_{a \in A \setminus \{wait\}} \Bigg\{ \sum_{s'_d \in S_d} L(\mu_{s'_d}|s_d, a, s_t) \cdot \\
& \int_{s'_t \in S_t} P_{\mu_{s'_d}}(s'_t - s_t) \left[r_t(\mu_{s'_d}, s_t) + r_{t'}(\mu_{s'_d}, s'_t) + r_\tau(\mu_{s'_d}, s'_t - s_t) + V^*(s') \right] ds'_t \Bigg\} \Bigg)
\end{aligned} \tag{8.18}
$$

Equation 8.18 corresponds exactly to equations 4.10 to 4.13. Consequently, the policy expressed with the parametric actions formalism is the same as the one calculated within the TMDP framework. This provides an answer to the initial question:

> The TMDP problem is an undiscounted non-stationary parametric action problem and its dynamic programming resolution is equivalent to solving equation 8.15.

In the method presented by [Boyan and Littman, 2001] and improved in chapter 6 the parametric aspect is hidden by the inclusion in TMDP policies of pairs of *wait* and discrete actions. This is made possible because $wait(0)$ has no effect on the state and provides no reward. Separating these pairs of actions brings the resolution back to the general framework of parametric actions and continuous time which we captured in the XMDP framework.

It is now rather easy to provide a discounted Bellman equation for TMDP problems. If $\gamma < 1$, then equation 8.18 can be written:

$$
\begin{aligned}
V^*(s) = \sup_{\tau \in \mathbb{R}^+} \Bigg(& r(s, wait, \tau) + \gamma^\tau \max_{a \in A \setminus \{wait\}} \Bigg\{ \sum_{s'_d \in S_d} L(\mu_{s'_d}|s_d, a, s_t) \cdot \\
& \int_{s'_t \in S_t} \gamma^{s'_t - s_t} P_{\mu_{s'_d}}(s'_t - s_t) \left[r_t(\mu_{s'_d}, s_t) + r_{t'}(\mu_{s'_d}, s'_t) + r_\tau(\mu_{s'_d}, s'_t - s_t) + V^*(s') \right] ds'_t \Bigg\} \Bigg)
\end{aligned} \tag{8.19}
$$

This equation is the discounted dynamic programming equation for TMDPs.

8.5 Conclusion on the XMDP framework

Our goal when introducing the XMDP framework was not to design a new method for solving time-dependent or hybrid state and/or action spaces MDPs. On the contrary, we wanted

to provide a sound framework with clear hypotheses, easily captured by intuition, which generalized MDPs to these hybrid spaces and which explicitly included the time variable.

What we have in the end is a model, similar in some ways to the Borel model for MDPs as presented in [Puterman, 1994] and rarely used as such, which includes observable time and makes the link with the successive decision epochs of the process to control. We can summarize these results:

An XMDP is a 4-tuple $\langle S, A(X), p, r \rangle$ describing a temporal decision process, defined over hybrid state and action variables and over a continuous observable time.

When the assumptions of section 8.2.5 are verified, one can guarantee the validity of a policy evaluation equation $V^\pi = L^\pi V^\pi$ and an optimality equation $V^* = LV^*$ for a discounted criterion.

These equations and assumptions provide the existence and characterization of an optimal policy π^*.

9

Perspectives: evolutive partitioning of time

The main practical drawback from the analytical calculation of V^* in the $TMDP_{poly}$ case — aside from technical computational difficulties — comes from the very large number of separate definition intervals in the value function. When comparing this number to the number of definition intervals needed to describe the policy, one could try to imagine another approach which would not necessitate such a fine representation of the value function. The method we develop in this chapter rests on the very simple idea that the crucial problem is to identify the bounds of the policy's temporal definition intervals. However, finding these bounds and finding the optimal action to perform in-between belong to the same optimization process. We try to iteratively find the values of all decision variables (bounds and actions) by solving a sequence of discrete problems generated by incremental evolution of the local temporal bounds.

This chapter is a "perspectives" chapter since it describes unfinished work and makes the link between different ideas. The ideas presented here are related both to the same problematic as the previous chapters on solving time-dependent MDPs, but also introduce the first ideas of approximate policy iteration for complex temporal problems which will be developed in the second part of the thesis. Quite ironically, the idea of evolutive temporal bounds for SMDP+ came chronologically very early in the thesis and spawned a lot of the research developments. Even though this idea did not result in a proper implementation, it provides a nice abstraction and overview of the problem of finding the bounds of decision intervals. It also introduces the idea of direct policy search which is at the core of the thesis' second part.

9.1 Definitions and general idea

We work with the general case of discounted SMDP+ as presented in chapter 4. We shortly recall the SMDP+ definition and the optimality equation derived from the proofs of chapter 8. An SMDP+ is given by the 4-tuple $\langle \Sigma, A+, Q, R \rangle$ where:

- Σ is the augmented state space containing all $\sigma = (s, t)$ elements. This state space can be decomposed into:
  - a discrete state space $s \in S$,
  - a continuous time axis $\mathbb{R}$.

- $A+$ is the action space which can be decomposed into
  - A, the discrete action space,
  - *wait* the parametric action representing *idleness*.
- $Q(\sigma'|\sigma, a)$ is the cumulative transition model. It can be written $Q(\sigma'|\sigma, a) = P(s'|s, t, a) \cdot F(t'|s, t, a, s')$. As previously and for convenience, we will write the probability density functions indifferently as $f(t'|s, t, a, s')$ or $f(\tau|s, t, a, s')$, with:

$$f(t'|s,t,a,s') = \begin{cases} 0 & \text{if } t' < t \\ f(\tau = t' - t|s,t,a,s') & \text{if } t' \geq t \end{cases}$$

- $R(\sigma', a, \sigma)$ is the reward model. It can be reformulated as:

$$r(\sigma, a) = \sum_{s' \in S} P(s'|s,t,a) \int_{-\infty}^{\infty} f(t'|s,t,a,s') R(\sigma', a, \sigma) dt' \tag{9.1}$$

The value of policy π is the cumulative sum of all successive rewards, each being discounted by a γ^t factor corresponding to the reward's date. Policy π's value function obeys equation 9.2.

$$V^\pi(\sigma) = \sum_{s' \in S} \int_0^\infty \left(R(s', t+\tau, \pi(\sigma), \sigma) + \gamma^\tau V^\pi(\sigma')\right) \cdot f(\tau|\sigma, \pi(\sigma), s') P(s'|\sigma, \pi(\sigma)) d\tau = L_\pi^t(V^\pi)(\sigma) \tag{9.2}$$

The optimal policy's value function V^* obeys equation 9.3.

$$V^*(\sigma) = \max_{a \in A+} \left\{ \sum_{s' \in S} \int_0^\infty \left(R(s', t+\tau, a, \sigma) + \gamma^\tau V^*(\sigma')\right) \cdot f(\tau|\sigma, a, s') P(s'|\sigma, a) d\tau \right\} \tag{9.3}$$

$$V^*(\sigma) = LV^*(\sigma)$$

A policy defined for SMDP+ models is a mapping from Σ to $A+$ specifying which action to undertake in discrete state s at time t. If the action is *wait* then it corresponds to let the system evolve by itself until we reach a new pair (s, t) where another action is necessary. Section 4.5.1 proved that — because *wait* is supposed deterministic with regard to time and does not change the state — this policy was indeed equivalent to a continuous TMDP policy. However, as soon as the effects of "wait" become stochastic with respect to the state, this equivalence does not hold anymore.

Hence, in a given discrete state s, one can describe the policy $\pi(s, t)$ as "for all dates between t_0 and t_1, the best action to undertake is a_4, however if t is between t_1 and t_2 it is better to remain idle, and between t_2 and t_3 the best action is a_1". Starting with this simple description, we try to find the values of t_0, t_1, etc. as well as the optimal actions on these intervals at the same time. This approach was first suggested in [Rachelson et al., 2006].

We can rephrase this goal by saying that we look for the most efficient partitioning of the time resource *per state*. For this purpose, we define the notion of *decision interval*. A decision interval in a given discrete state is a temporal interval over which the policy is constant.

One can relate the notion of decision interval to the Borel model of MDPs as presented in [Puterman, 1994] and, more practically, to the continuous variables partitioning of [Feng

et al., 2004], [Li and Littman, 2005] or [Benazera et al., 2005]. Similarly to these approaches, decision intervals can be easily factored and represented as *kd-tries* for example. However, we are not directly looking for an incremental refinement of our discretization as in [Munos and Moore, 2002], but for an incremental evolution of the decision intervals set of bounds (in number of bounds and in value). The central idea of the method we introduce in the next section is to populate, correct and reduce the set of bounds per state as needed to define and improve the policy.

9.2 Evolution of decision intervals and actions by solving a sequence of discrete problems

9.2.1 Algorithm overview

Let us introduce the set $\mathcal{T}_s$ of decision intervals in s, relative to the last policy defined. With this discrete set of intervals, we can consider the discrete abstract state space:

$$\tilde{\Sigma} = \{(s,T)\,/s \in S, T \in \mathcal{T}_s\}$$

Suppose we start with an initial guess of the time partitions per state, *ie.* we have an initial $\mathcal{T} = \{\mathcal{T}_s, s \in S\}$. Then our algorithm proceeds in four steps:

- **Discretization.** First of all, it computes the transition and reward models of the $\tilde{M}$ discrete MDP, having state space $\tilde{\Sigma}$ and approximating the behaviour of the hybrid SMDP+ problem M.
- **Action optimization.** Then we compute the optimal policy with respect to the $\tilde{M}$ problem. Let $\tilde{\pi}$ be this policy. We merge any two consecutive intervals of $\mathcal{T}_s$ over which the optimal action remains the same.
- **Policy evaluation.** Third we evaluate $\tilde{\pi}$'s value with respect to the continuous model by defining the corresponding SMDP+ policy π.
- **Decision interval's evolution.** Finally we use this value function to perform a single Bellman backup providing the date where we could bring the best improvement to the policy's value. We use this date to populate the sets $\mathcal{T}_s$.

9.2.2 The method in detail

We can now consider these four phases in detail.

First step and initialization: generating $\tilde{M}$. We build the discrete MDP problem $\tilde{M}$ with:

- the state space $\tilde{\Sigma}$,
- the action space $A+$,
- the transition function $\tilde{Q}(\tilde{\sigma}'|\tilde{\sigma}, a)$,
- the reward model $\tilde{r}(\tilde{\sigma}, a)$.

The transition model $\tilde{Q}(\tilde{\sigma}'|\tilde{\sigma}, a)$ describes the probability that action a, undertaken in s_σ, during T_σ, takes the process to state $s_{\sigma'}$ at a date belonging to $T_{\sigma'}$. Similarly $\tilde{r}$ represents the average reward obtained when applying a and going from (s_σ, T_σ) to $(s_{\sigma'}, T_{\sigma'})$.

More precisely, if t_{low} and t_{up} represent respectively the lower and upper bounds of interval T_σ, we can choose to calculate $\tilde{Q}$ as the average over the T_σ interval of the probability of reaching the $T_{\sigma'}$ interval:

$$\begin{aligned}\tilde{Q}(\sigma', a, \sigma) &= \frac{1}{t_{up} - t_{low}} \int_{t_{low}}^{t_{up}} Pr(t' \in T_{\sigma'}, s' = s_{\sigma'}|a, s_\sigma, t_\sigma)dt \\ &= \frac{1}{t_{up} - t_{low}} \int_{t_{low}}^{t_{up}} \left(P(s'|s, t, a) \int_{t'_{low}}^{t'_{up}} f(t'|s, t, a, s')dt' \right) dt\end{aligned}$$

And if we write the cumulative distribution function F:

$$F(v|s, t, a, s') = Pr(t' \leq v|s, t, a, s') = \int_{-\infty}^{v} f(t'|s, t, a, s')dt'$$

Then we have:

$$\tilde{Q}(\sigma', a, \sigma) = \frac{1}{t_{up} - t_{low}} \int_{t_{low}}^{t_{up}} P(s'|s, t, a) \left[F(t'_{up}|s, t, a, s') - F(t'_{low}|s, t, a, s') \right] dt \tag{9.4}$$

Similarly, we chose to write $\tilde{r}$ as the average over the T_σ interval of the rewards obtained during the transitions (σ, a, σ').

$$\tilde{r}((s_\sigma, T_\sigma), a) = \frac{1}{t_{up} - t_{low}} \int_{t_{low}}^{t_{up}} r((s_\sigma, t), a)dt \tag{9.5}$$

The choice of taking the average over the T_σ interval is arbitrary and questionable. One could choose, for example, to use the best reward obtained over the interval in order to build an optimistic reward model instead.

The transition model of *wait* takes the process to a new state described by the system's dynamics $P(s'|s, t, wait) = W(s'|s, t)$ and to the first date of the next decision interval in s'.

Evaluating the discrete $\tilde{Q}$ and $\tilde{r}$ functions can be done easily through analytical calculation as previously if possible. Else, it can be approximated via Monte-Carlo sampling or continuous functions discretization.

It is important to note that $\tilde{M}$ is an *approximation* and an *abstraction* of M. It is an approximation is the sense that it approximates the transition and reward models over the decision intervals by taking the average values. It also is an abstraction because it does not respect the causality principle anymore. In $\tilde{M}$, it is possible to reach a temporal interval beginning before the current date, and from this interval, to reach another prior interval which would entirely lie before the initial current date. Therefore, $\tilde{M}$ can be seen as an approximate optimistic problem where causality can be violated and where reachability is considered from a very optimistic point of view.

We provide no theoretical justification of the soundness of such an approximation and abstraction. Instead, we rely on the idea that one does not need to evaluate exactly the transition dynamics and the rewards to build a rough plan of action. This $\tilde{M}$ problem can thus be seen as a — rather drastic — variation of the "optimism in the face of uncertainty"

philosophy developed in [Kaelbling, 1990].

Second step: searching for the optimal action. The second step consists in solving the Bellman optimality equation corresponding to problem $\tilde{M}$. We suppose there is a "black box" discrete MDP solver available and we can feed the $\tilde{M}$ problem to this solver. This optimization provides us with a $\tilde{\pi}$ policy defined on $\tilde{\Sigma}$.

It can happen that the policy defined on two consecutive decision intervals of the same state ends up in pointing to the same action after the optimization process. In this case, we merge the two decision intervals into one in order to keep the number of bounds low and the representation as compact as possible. No new introduction of bounds is possible at this step since we are only optimizing the discrete problem $\tilde{M}$.

Third step: Evaluating $\tilde{\pi}$ on the real system. One can see $\tilde{\pi}$ as an approximation of an optimal policy for M. It is not exactly a policy obtained through approximate dynamic programming since it results from the "black box" solver used in step 2 — which might be either an exact or an approximate solver, but its generation relies on an approximation of the model which yields an exact or approximate value function on this approximate model, which in turn provides us with $\tilde{\pi}$. Consequently, the $\tilde{\pi}$ policy leaves room for improvement with respect to the continuous initial problem because the problem solved was a discrete approximation of this initial problem. The goal of step 3 is to let the $\mathcal{T}$ discretization evolve in order to let the next step's $\tilde{\pi}$ be better than the current one, with respect to the continuous temporal problem.

This leaves us with two separate problems:

- Suppose we have found the optimal policy π^* for M then we have a partitioning set $\mathcal{T}^*$ used for this policy's description and we can build the associated $\tilde{M}^*$ problem. Then, to guarantee the soundness of our algorithm, we need to insure that the optimal policy $\tilde{\pi}$ found after after optimization on the $\tilde{M}^*$ problem is identical to π^*.
- Secondly, the evaluation method of $\tilde{\pi}$ with respect to the continuous problem must be good enough so as to eventually find the points in time where the policy can be improved.

The first problem corresponds to proving that the overall approximation and optimization scheme has a fixed point in π^*. Ideally, one should also prove it is a contraction mapping in order to insure convergence. As for many approximate dynamic programming algorithms, proving such a property is often very hard or impossible. For an example illustrating this difficulty, see the discussion on approximate value iteration of section 6.4. However, proving the stability (or bounding the variations) of π^* through the model approximation and optimization steps provides a good criterion to evaluate the consistency of the approximation method for generating $\tilde{M}$.

Similarly, the evaluation of $V^{\tilde{\pi}}$ can be done via several different methods. If exact computation with the continuous functions of M is feasible, one could try a $TMDP_{poly}$ -like evaluation. Approaches such as Approximate Linear Programming (least-square minimization of a vector of weights on feature functions) as in [Guestrin et al., 2004] or Monte-Carlo approaches are also possible. Depending on the nature of the continuous problem at hand, one could choose an option or another, the goal remains to obtain an evaluation of $\tilde{\pi}$'s quality on the real continuous problem, *ie.* to solve equation 9.2 for $\tilde{\pi}$.

Fourth step: populating the decision intervals sets. Once we have the evaluation $V^{\tilde{\pi}}$, we need to answer the question "where should I introduce a new bound in order to improve my policy's quality?". Answering this question actually means inferring that by performing another action than the one specified by $\tilde{\pi}$, one improves the expected gain of an execution. This idea is very close to the improvement step of Policy Iteration. Here, one could consider that the decision variables are the decision intervals' bounds and that we search for new values of these bounds which will improve the efficiency of our policy. Hence, we need to find where we can potentially improve the policy's quality.

Evaluating such an improvement can be done by trying to find the best action to undertake in the current state before applying $\tilde{\pi}$ for the rest of the execution. It corresponds to calculating the one-step lookahead best action by performing one Bellman backup. Therefore, we are looking, *per state*, for the greatest value of the Bellman error as a function of t.

We recall the definition of the Bellman error as presented in [Bertsekas and Tsitsiklis, 1996]. Let π be a policy defined on the state space of a discrete MDP. Let V^{π} be π's value function. The Bellman error in state s is the value of the best improvement possible with a one-step dynamic programming optimization of the policy:

$$BE(V^{\pi}(s)) = \max_{a \in A} \left(r(s,a) + \gamma \sum_{s' \in S} P(s,a,s') V^{\pi}(s') \right) - V^{\pi}(s) \tag{9.6}$$

We define the *Bellman t-error*, in discrete state s, as the function of time representing the gain obtained by optimizing the first action of an execution path, before applying the current policy (or before receiving the value specified by the value function of the policy). In a given discrete state s, the Bellman t-error with respect to value function V is given by:

$$BE_s(t) = \max_{a \in A} \left(r(s,a,t) + \sum_{s' \in S} \int_{-\infty}^{\infty} \gamma^{t'-t} V(s',t') P(s'|s,a,t) f(t'|s,t,a,s') dt' \right) - V(s,t) \tag{9.7}$$

Finding and maximizing $BE_s(t)$ can either make use of analytical calculation if it is possible (in the *TMDP$_{poly}$* case, finding the supremum of a piecewise polynomial function is an easy calculation). One can also make use of other optimization techniques such as local convex optimization (gradient descent, Newton methods, evolutionary algorithms) depending on how much information we can extract from $V^{\tilde{\pi}}$ (values, gradients, Hessian matrices, etc.).

Let us consider the question of finding the largest Bellman error more precisely. For notation convenience, we introduce the L_a operator for standard MDPs:

$$L_a(V)(s) = r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V(s) \tag{9.8}$$

One can then write: $\forall s \in S, LV(s) = \max_{a \in A} L_a V(s)$.

Similarly, for SMDP+, we write:

$$L_a^t(V)(s,t) = r(s,a,t) + \sum_{s' \in S} \int_{-\infty}^{\infty} \gamma^{(t'-t)} V(s',t') P(s'|s,t,a) f(t'|s,t,a,s') dt' \tag{9.9}$$

Consequently, we can write:

$$BE_s(t) = \max_{a \in A} \left\{ L_a^t \left(V^{\pi} \right)(s,t) \right\} - V^{\pi}(s,t) \tag{9.10}$$

We are looking for $\sup_{t\in\mathbb{R}} BE_s(t)$ but:

$$\begin{aligned}\sup_{t\in\mathbb{R}} BE_s(t) &= \sup_{t\in\mathbb{R}} \max_{a\in A} \left\{ L_a^t(V^\pi)(s,t) - V^\pi(s,t) \right\} \\ &= \max_{a\in A} \sup_{t\in\mathbb{R}} \left\{ L_a^t(V^\pi)(s,t) - V^\pi(s,t) \right\}\end{aligned}$$

So we are left with $|S| \cdot |A|$ maximization problems where we want to solve:

$$\sup_{t\in\mathbb{R}} \left\{ L_a^t(V^\pi)(s,t) - V^\pi(s,t) \right\} \tag{9.11}$$
$$t \in [0,T]$$

Then, depending on the shape of M's functions and of V^π, we can try to apply different optimization techniques. Gradient descent might generally be sufficient to find the possible sup values.

9.2.3 Related work and conclusion

As mentioned earlier, this method differs from the algorithms presented in [Feng et al., 2004], [Li and Littman, 2005] and [Benazera et al., 2005] (HAO*) because it does not search for a local refining of a continuous variable's partitioning, but for the smallest set of bounds needed to define the policy on this variable.

Earlier work on this problem and on the problem of incremental discretization of continuous variables was proposed in [Munos and Moore, 2002] and [Munos and Moore, 2000]. The method proposed in the previous paragraph builds on the same idea to concentrate accuracy where it is needed. However, the main difference lies in the fact that our method sacrifices two aspects to obtain as little bounds as possible: *optimality* and *causality*. Optimality is lost because we pop some bounds out of the bounds' list when two consecutive actions are equal, thus implying a worse approximation in the discretized model than if we had not removed these bounds. Causality in the discretized problem is lost because of the approximation method, as explained at step 1 of the previous algorithm.

Lastly, one can push the comparison with policy iteration a little further. If one considers the $(\tilde{t}, \pi(s,\tilde{T}))$ decision variables, the previous algorithm can be seen as a policy iteration algorithm where the evaluation phase is an approximate evaluation of the policy using an optimistic model obtained by discretization of the continuous problem and where the optimization phase results from the discrete optimization for the actions and from the continuous approximate optimization for the bounds' evolution.

Finally:

The method presented in this chapter separates the decision intervals' bounds optimization and the action selection procedure. It relies on an incremental method, similar to the philosophy of policy iteration, to improve the bounds' number and values and on a discrete MDP resolution scheme to preserve the coupling between these bounds and the optimized actions. This method could be implemented using different tools for MDP optimization, model discretization and convex optimizations, providing a family of variants based on the same principle of incrementally finding the right intervals for policy definition.

10

Conclusion

This chapter summarizes the results obtained in the previous chapters. We also discuss the possibility of adapting the $TMDP_{poly}$ method and tools to the more general case of XMDPs with hybrid state and action spaces, highlighting where the advantages and difficulties are. Finally we conclude on this first part of the thesis and explain how it leads to the second part.

10.1 "Take-away" messages

This first part of the thesis focused on the problem of introducing a continuous time variable in the MDP framework. This raised questions concerning the link with the discounted criterion, the resolution algorithm and the formal representation framework of temporal Markov decision problems. Here is a short summary of the conclusions drawn from the previous chapters:

- Considering a continuous observable time variable implies looking at a hybrid state space MDP. Furthermore, having an observable time directly affects the definition of the discounted criterion.
- Introducing continuous variables such as time often calls for the introduction of continuous actions such as *wait*. This yields a hybrid action space MDP with hybrid state space and observable time in the discounted criterion.
- The XMDP framework captures these characteristics and establishes an optimality equations for the policies one could define on such problems. This XMDP framework includes standard MDPs, SMDP+ and TMDPs.
- In practice, when time is the only continuous variable and *wait* the only continuous action, some extra hypotheses can be made. Namely, *wait* is often deterministic with respect to the states variables and the reward for a zero duration waiting is zero. This falls into the framework of SMDP+. Sometimes *wait* might even have no impact on the discrete part of the state space. This is the standard TMDP framework which we slightly extended to deterministic effect on the state variables through the use of a W function describing the deterministic evolution of the system while waiting.
- The optimality equations presented in [Boyan and Littman, 2001] for the TMDP framework correspond to a total reward criterion for the equivalent XMDP.

- Trying to extend the exact resolution scheme of TMDPs to the case of piecewise polynomial functions is quickly refrained by the properties of formal calculations on such representations. Namely, this exact resolution scheme could not be extended further than discrete probability density functions, piecewise constant transition probabilities and piecewise polynomial reward functions of degree lower than 5.

- The analysis of the TMDP optimality equations provided a more global approximate resolution method for the case of piecewise polynomial functions, based on:

  - Exact and approximate formal calculations on piecewise polynomial functions.
  - Prioritized sweeping adapted to TMDPs.
  - Approximate value iteration.

  These features spawned the *TMDP*$_{poly}$ algorithm and planner.

- The main drawback of value iteration methods for temporal Markov decision problems comes from the difficulty to define precisely the value functions. In the case of piecewise polynomial functions it is expressed through the number of definition intervals needed to accurately describe the value functions. We provided a first attempt at simplifying this representation by taking a short look into evolutive partitioning of time. This resulted in a "policy iteration"-like method which contains the first ideas about the model-free reinforcement learning methods of the thesis' next part.

10.2 Perspectives

On top of the perspectives concerning the evolutive discretization of time presented in chapter 9, it is interesting to take a look at how the developments made for the *TMDP*$_{poly}$ algorithm and planner can apply to a more general class of MDP problems with hybrid action and state spaces. More specifically, the question one could ask is: how could we adapt the *TMDP*$_{poly}$ algorithm to XMDPs?

Even though most of the bricks seem available, building the house is not straightforward. There are several reasons for that. First of all, few methods have been developed in the literature to perform formal Bellman backups as we have done for piecewise polynomial TMDPs. Generally, the option taken is either to solve a linear program trying to fit V_{n+1} onto a good set of feature functions, as in ALP or LSPI. The initial Neuro-Dynamic Programming approach uses neural networks as a common representation of value functions, other recent approaches deal with different kinds of estimators or regression operators, but — to our knowledge — in most cases, the problem turns out to be similar to a supervised learning (or fitting) problem. The search for an L-stable family of functions over which one could perform formal Bellman backups is a rather hard problem and has provided few results until now.

When one deals with several continuous variables on top of the discrete ones, finding a good representation framework seems even harder. The case of piecewise constant or linear function representations has been used in [Feng et al., 2004], [Li and Littman, 2005] and [Benazera et al., 2005]. We can also mention an interesting alternative method using phase-type distributions presented in [Marecki et al., 2006]. An *XMDP*$_{poly}$ implementation could probably make good use of piecewise linear or constant functions, associated to discrete states, as in [Benazera et al., 2005].

However, the main difficulty comes with the definition of continuous and hybrid actions. With TMDPs, the optimization benefited a lot of the deterministic behaviour of *wait* and of the fact that it was the only continuous action. This allowed the decoupling of equations as illustrated by the proof (from XMDPs back to TMDPs) of section 8.4. In the general case, one should solve the parametric formulation of dynamic programming presented in equation 8.15. In other words, one should separately find the optimal action parameters before comparing the actions together.

Here again, a good representation of the value functions might facilitate the search for these optimal parameters. Action elimination procedures, as presented in [Puterman, 1994] or used in [Mausam and Weld, 2006], can also reduce the amount of computation needed to consider these hybrid actions.

Even though these approaches are not directly related to model-based MDP optimization, one should also mention the recent work in Reinforcement Learning of [Hasselt and Wiering, 2007] or [Antos et al., 2007] on the topic of optimizing MDPs with continuous action spaces.

Finally, what makes most current MDP planners efficient are their search strategies. Even though prioritized sweeping is an efficient way of ordering Bellman backups for complete resolution of MDP problems, heuristic search provides an important efficiency gain for the partial resolution of focused problems. Thus, depending on the kind of XMDP problems one wishes to solve, an XMDP planner might not make use of dynamic programming steps in the same fashion as the $TMDP_{poly}$ planner.

To summarize these ideas, XMDPs and $TMDP_{poly}$ open the door to a more general class of methods for MDPs with hybrid state and action spaces, but:

- Finding a good representation for the value function will remain a hard problem for which the piecewise polynomial representation clearly has limitations.
- Formal Bellman backups might be useful to solve the parametric formulation of dynamic programming (equation 8.15).
- These problems will still suffer from a somehow extended *curse of dimensionality*. This implies action selection, heuristic and focused search and approximate methods will be a critical issue.

10.3 Opening

By considering how this work could apply or be adapted to the generalized case of XMDPs we already took a step back from the academic problem of TMDP optimization. Let us take a second step away and consider the way our problem was stated in the first place.

Since the beginning of this first part, we considered that a model was explicitly available to us and that we could approximate it, using some specific tools as, for example, discrete or piecewise polynomial distributions or piecewise polynomial functions. However, in practice, the difficulties encountered when dealing with the problems we tried to solve do not only concern the design of efficient algorithms to solve them; they also leave a lot of work on the task of writing them down in the first place.

As for most classical MDP problems, writing the transition matrices or the transition probability density functions is a task which requires a lot of engineering. Many of the systems we want to control are not easily described through an explicit probability distribution simply because finding the exact shape or values of such a distribution is a hard task in the first place. Let us take the subway example of chapter 2 for instance, where the state variables would be the number of passengers at each station and in the trains, the current position of each train and the time variable. Given a current state, the probability distribution on the next state requires a lot of engineering to be explicitly computed, while writing a simulator for such a problem is a much simpler task.

This simple example illustrates the reason why we need to focus on representing and formalizing complex stochastic temporal processes, in order to efficiently capture their behaviour and to design sound simulation systems.

Optimizing control policies without using an explicit model of the process, by exploiting reward signals provided by the environment, is the field of study of Reinforcement Learning. The second part of the thesis focuses on representing the complexity of temporal Markov decision problems in order to build a *generative model* of the process. This generative model is then used in conjunction with a simulation-based, approximate policy iteration method, designed to exploit the observable time variable.

Part III

Controlling Time-dependent Stochastic Systems with Concurrent Exogenous Events

Overview

There is a number of ways to introduce this third part's contents. The point of view we develop here follows the global orientation of the thesis on temporal problems. At the end of this overview, we will propose an alternative way of introducing and reading the following chapters.

While the previous part focused on the inference of the optimal value function, given a formal model of the process, and based on an analysis of the adapted Bellman operator for time-dependent problems, this second part starts with the following admission of weakness: such a formal model is often not available.

Physically, if one can suppose there exist underlying probability distributions describing the behaviour of our systems, these distributions are often hard to obtain exactly. In a sense, part II used this argument to justify our polynomial approximations. However, approximating a model implies having an explicit prior knowledge about its behavior. Real world problems are often too complex to have an explicit — even approximate — *implicit-event*, *predictive model* of the system.

This leads us to consider the question of planning without such an explicit *predictive model*. For many real world problems, it is remarkably easier to build simulators than to describe the system through synthetic, explicit equations. In other words, while *predictive models* are hard to build, *generative models* are often available. The question of searching for a policy through the interaction with a simulator is the field of Reinforcement Learning.

Temporal problems of decision under uncertainty fall into this category of complex planning domains for which a synthetic representation of the problem is often not available as a single, explicit, stochastic decision process. We resolve to — at least in a first time — sacrifice the idea of finding an optimal policy; instead, we rather search for improvements of an initial behavior by locally improving the agent's policy in the most likely current situations. Based on these ideas, the following chapters introduce a number of contributions related to very different domains:

- Chapter 11 explores the question of analyzing the temporal problems' complexity. It illustrates the crucial contribution of *concurrency* to this complexity. It also establishes a link between concurrent stochastic processes modeling and the discrete event systems specification theory (DEVS, [Zeigler et al., 2000]). It finally extends this analysis to decision processes. This provides a complete study of the temporal decision processes' structure as well as an elegant method for compactly specifying temporal Markov decision problems, based on the GSMDP model of [Younes and Simmons, 2004].

- Chapter 12 then focuses on the question of locally improving the agent's behavior. This question is strongly related to Policy Iteration approaches. We provide a complete review of Policy Iteration, approximate Policy Iteration and asynchronous dynamic programming in order to naturally introduce the idea of Real Time Policy Iteration (RTPI) and relate it as much as possible to its parent ideas.

- In chapter 13 and 14 we introduce the Approximate Temporal Policy Iteration (ATPI) algorithm. This algorithm is a specialization of RTPI to the case of temporal problems with continuous action spaces. It brings together the simulation basis introduced in

chapter 11, the algorithmic method of chapter 12 and tools borrowed from the field of Statistical Learning which are needed to adapt to the features of continuous state spaces. In particular, the improved ATPI algorithm of chapter 14 introduces a notion of *confidence* in the policy and value function, related to the problem of approaching the sufficient statistics for the $V^\pi(s)$ and $\pi(s)$ variables. This notion of confidence is related to the exploration/exploitation compromise of Reinforcement Learning. It can also be exploited as a new attempt to bring together heuristic search and exploration in continuous state spaces.

Now that this first introduction has situated this second part's contribution inside the topic of temporal Markov decision problems, there is a second point of view which is important to consider. This point of view is directly related to the problem of searching for a policy in a continuous, unstructured, high-dimensional state space.

Contrarily to the case of finite discrete state spaces, this family of problems regains one fundamental property of real world continuous problems: there is a zero probability of visiting the same state twice. Therefore, all methods based on rollout sampling for evaluation, need — at least to some extent — to rely on a supplementary notion of *local generalization*. This property of generalization — which applies both to the policy and the value function — is an important feature of learning systems in general, which is somehow hidden in standard, discrete state Reinforcement Learning and which is central in Statistical Learning theory.

Moreover, even though the state space might be unstructured, the problem itself often exhibits a specific organization: some states can be grouped together, some regions can be seen as similar from the point of view of the optimal policy, the value function, or the transition function. Discovering the structure of the problem, of the optimal value function, or of the policy is a an important key to efficient reasoning in large, unstructured problems. While Reinforcement Learning specializes in the online, dynamic improvement of an agent's behavior, Statistical Learning focuses on static structure analysis and abstraction. Hence, establishing the link between Reinforcement Learning and Statistical Learning appears crucial to build intelligent learning agents.

While the question of bridging the gap between the dynamic structure of Reinforcement Learning and the static problems of Statistical Learning is beyond the scope of this thesis, the problem appears fundamental for an autonomous agent operating in a continuous or hybrid environment. In that matter, this thesis third part and the ATPI algorithm constitute an attempt to build such a bridge in the case of temporal Markov decision problems.

11

Concurrency: an origin for complexity

Many temporal problems present a complex structure. Writing models or simulators for such problems quickly becomes a huge engineering task, sometimes as difficult as solving the decision problem itself. This chapter focuses on what appears to be one of the main reasons for complexity of temporal problems: concurrency of local phenomena. Efficient and compact representation of stochastic processes' concurrency seems to be a key to tackling large, complex temporal problems. The framework of Generalized Semi-Markov Processes (GMSPs) elegantly captures the complexity of the global temporal process. After exploring the properties of GMSPs, we investigate more precisely the question of modeling such stochastic processes in the unified DEVS framework. Then we introduce action choice in GSMPs in order to model the full problem of decision making under uncertainty with concurrent exogenous events and observable continuous time.

11.1 The complexity of writing the model for stochastic temporal problems

Let us turn back to three of the examples presented in chapter 2, namely, the subway, the airport and the coordination problems. All these three problems present some features in common:

- They take place in a strongly time-dependent, stochastic environment.
- The decision problem has a limited number of initial states.
- Part of environment's evolution is controllable through the agent's actions but part of it is not.

Modeling and analyzing the environment's non-controllable behaviour already raises questions: does this process retain Markov's property? Do we have to go through intensive analysis of each of the process's variables to get an idea of the global process' evolution? How can we simply and compactly capture this behaviour?

One first remark concerning the time dependency of these problems is that time plays a central role because it is one of the crucial, non-replenishable resources upon which the processes depend. It could indeed be replaced by another equivalent variable. However we

will keep using time as our "red line variable" for clarity.

The direct dependency of the process on time has been studied in the thesis' first part and it is not this explicit time dependency that is hard to model here. Indeed, when the process depends *only* on time, then the overall problem turns to solving a hybrid variable MDP problem as in the previous part. So the complexity of the examples' behaviour does not uniquely come from the time-dependency of the processes at hand.

What makes it hard to predict the next state of the process for the above examples is the fact that these processes result from the local interaction of heterogeneous phenomena. If we take the airport example: the probability that the process' next state corresponds to the arrival of plane p at terminal t is given by the probability of a movement's success, provided that there are no airports alarms triggering before this arrival, that the weather does not change before this arrival, that another plane does not reach another terminal before this arrival, etc. This simple example illustrates the fact that the complexity of writing the transition model of such a discrete event process comes from one simple statement:

> The overall process' complexity results from the concurrent interaction of local processes. These local processes are often simple time-dependent processes but they are strongly coupled together via the values of the common state variables.

So what makes our problems hard to model is not really their time dependency; it is the fact that they are the resulting process of multiple small processes, all coupled through the state space. This coupling is strong in the sense that each individual process affects most variables of the state space and, conversely, all processes outcomes depend on the current global state. Hence, we need to make a distinction here with the *weakly*-coupled MDP framework of [Dean and Lin, 1995; Meuleau et al., 1998; Bernstein and Zilberstein, 2001] which has laid the basis of decomposition algorithms for MDPs. Such a decomposition is not possible here because of the *strong* coupling between concurrent processes through the state space.

Consequently, if we can find a modeling framework that captures both the local processes and their coupling via the state space, then we will be able to compactly represent the behaviour of the global system.

11.2 Generalized Semi-Markov Processes

In the stochastic processes literature, the resulting process of several concurrent temporal processes is called a *generalized process*. The main example of such a process is the framework of Generalized Semi-Markov Processes introduced by [Glynn, 1989]. These processes were briefly introduced in chapter 2 and we provide more details here.

A GSMP represents the concurrent execution of several semi-Markov processes (SMPs). All these processes have stochastic transition destinations and stochastic sojourn times. Moreover, there is a strong coupling between the processes because they all affect the same random variables. Consequently, the overall process is a discrete event process resulting from the successive triggering of transitions in the different individual SMPs.

Formally, a GSMP is described by a set S of states and a set E of events. Each event can be described as an independent semi-Markov process over the random variables of the state space. At any time, the process is in a state s and there exists a subset E_s of events that

are called *active* or *enabled*. These events represent the different concurrent processes that compete for the next transition. To each active event e, we associate a clock c_e representing the duration before this event triggers a transition as presented on figure 11.1. This duration would be the sojourn time in state s if event e was the only active event and thus corresponds to the associated SMP's sojourn time in state s. The event e^* with the smallest clock c_{e^*} (the first to trigger) is the one that takes the process to a new state. The transition is then described by the transition model of the triggering event: the next state s' is picked according to the probability distribution $P_{e^*}(s'|s)$. In the new state s', events that are not in $E_{s'}$ are disabled (which actually implies setting their clocks to $+\infty$). For the events of $E_{s'}$, clocks are updated the following way:

- If $e \in E_s \setminus \{e^*\}$, then $c_e \leftarrow c_e - c_{e^*}$
- If $e \notin E_s$ or if $e = e^*$, pick c_e according to $F_e(\tau|s')$

The first active event to trigger then takes the process to a new state where the above operations are repeated.

Definition (Generalized Semi-Markov Process, [Glynn, 1989])**.** *A GSMP is given by the 4-tuple $\langle S, E, F, P\rangle$, where:*

- *S is the set of possible values for the process' state.*
- *E is the set of events describing the process. This set can be reduced to a subset E_s of active events in each state s.*
- *F is the cumulative distribution function giving the duration before an event triggers. The duration τ before e triggers is drawn according to $F(\tau|s, e) = F_e(\tau|s)$.*
- *P is the transition function of the process. When event e triggers, the new state s' of the process is drawn according to $P(s'|s, e) = P_e(s'|s)$.*

The framework of GSMPs could be compared with the (deterministic) framework of Timed Automata introduced in [Alur and Dill, 1994] which uses a similar description of the temporal behaviour of a system involving concurrency.

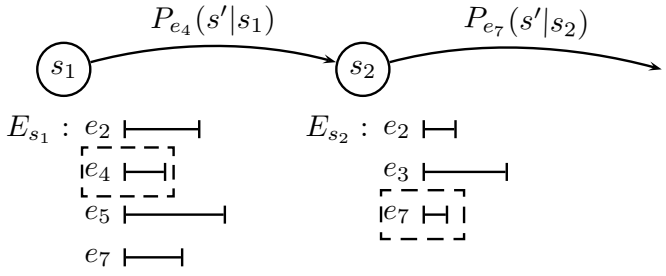


Figure 11.1: Illustration of a GSMP

A GSMP is an event-driven stochastic process, summarizing the concurrent effects of several semi-Markov processes on a common state space.

One can notice that — as in the SMP case — one can let the transition model depend on the clock c_{e^*}, thus yielding a $P_{e^*}(s'|s, c_{e^*})$ transition function.

Since GMSPs represent the overall process by factoring it through its separate concurrent events, it provides a much simpler description than a monolithic model of the global process. In fact, each of the individual SMPs constituting the GMSP might have rather simple transition and duration probability functions and thus can be easy to model. Writing the corresponding GSMP avoids the heavy task of explicitly integrating all these concurrent processes into one large, explicit stochastic process. The drawback of this situation is that we do not have an explicit formalization of the overall process anymore but rather a compact description of its dynamics.

Consequently — as exposed by Glynn in his introductory paper in 1989 — GMSPs provide both a "precise language for describing discrete event systems, and a mathematical setting within which to analyze discrete event processes"; the core idea being to capture the essential dynamical structure of a (stochastic) discrete event system. The analysis of GMSPs clarifies the connections between continuous variable dynamic systems and discrete event dynamic systems by considering GSMPs as event-driven stochastic processes.

The specialization of GSMPs to time-homogeneous sojourn times yields the time-homogeneous GSMP setting which can be reduced and analyzed as a continuous time Markov chain and thus as a standard Markov process through the operation of uniformization. This raises a similar question for the general case of GSMPs: does the stochastic process corresponding to the evolution of the SMPs' common state space random variables s still retain Markov's property?

As for SMPs, the answer is no. It is rather straightforward to provide a physical explanation to this: when considering the above process, defined on the common state space random variables s — which we will call the *natural process* — from an external point of view, an observer does not have enough information to predict which event will trigger next, and hence, which is the probability distribution on the next state of the process. This also implies the GSMP does not even retain the semi-Markov behaviour of the underlying SMPs.

In his 1998 paper, Nilsen presents an implementation of a GSMP modeling and simulation tool (GMSim). In order to build the simulator's underlying process, Nilsen uses the *supplementary variable technique* (presented, for instance, in [Cox and Miller, 1965]) in order to insure the semi-Markov behaviour of the global process, namely, to be able to predict the future state by only looking at the current state. However, as expected from a collection of SMPs, the sojourn times remain time inhomogeneous. The supplementary variable technique is used to construct an *augmented state* containing both the state of the natural process (natural state) and the active events' clocks. With this information, it is possible to write the probability distribution on the next augmented state of the process without information about its past history. We leave the notation details to [Nielsen, 1998] and simply conclude that:

> The stochastic process described by the *natural state* variables of a GSMP does not retain the semi-Markov behaviour of the individual underlying semi-Markov processes. By including the events' clocks in an *augmented state*, we are able to build a process over the random variables (s, c) which regains this semi-Markov behaviour.

This last conclusion raises an important question concerning the systems we wish to control: is the augmented state observable to an external decision-maker? In other words: will it be possible to re-use the results known for Markov or semi-Markov decision processes in order to control GSMPs with action choice? Will we have a guarantee of an optimal Markov policy? We leave this question for the next chapter but already underline the fact that in most practical cases, only the natural state of the process is observable and the clocks are generally unknown. A simple example to illustrate this fact is the "roads crossing" example: suppose the agent is a car driver arriving at a multiple crossing with traffic lights, he can observe the current natural state of the process (the lights) but cannot predict which one will turn green first because he cannot observe the individual processes' clocks.

Figure 11.2: Hard to predict which will turn green first.
(Picture credit: `http://www.greenwichup.org.uk`)

Finally, GMSPs seem to be an elegant, compact and efficient way of describing the complexity of temporal stochastic processes, especially if we include time as an observable continuous state variable. We will focus on this last point in the next chapter also, as we will introduce time and action choice alltogether in the problem. For now, we remain in the discrete event dynamic systems modeling problematic and try to make a link between GSMPs and the general DEVS modeling framework.

11.3 DEVS modeling

11.3.1 Five levels of Discrete Events Systems Specification

In [Zeigler, 1976], B. P. Zeigler proposes to describe the notion of *system* through a formal specification. This formal specification depends on the level of refinement in the system's description. He describes five levels of description going from uninterpreted input-output system specification to a complete specification of the system dynamics through the notions of internal state, internal transition function, external transition function and model coupling. These levels of specification rely heavily on the notion of discrete event system. He applies the fifth (and most detailed) level of specification to isolate the core components of a discrete event system and design the Discrete EVent system Specification (DEVS) framework

which is built to capture the modeling of any discrete event system.

Discrete event systems find an expression through different formalisms such as finite automata, Petri nets, state charts, cellular automata or stochastic processes for instance. Many of these formalisms have been studied and mapped to the DEVS framework. This makes DEVS more than a high-level description of the behaviour of discrete event systems: it captures the notion of consistent multi-modeling and of models integration. Consequently, it provides a sound theoretical basis for the study of discrete event systems modeling, coupling and simulation.

We use this section to introduce the basic notions of DEVS modeling in order to write GSMPs as DEVS models in the next section. This presentation is a pragmatic view of DEVS modeling, for a more formal presentation and a more detailed description, please see [Zeigler, 1976] and [Zeigler et al., 2000].

11.3.2 Atomic models

DEVS models are composed of atomic models describing independent processes, eventually coupled together through a hierarchical notion of coupled models. The idea of DEVS modeling is to capture the basic elements of a discrete event system's behaviour through the minimal concepts of evolution functions and variables. The atomic DEVS model builds on the idea of the black box having input and output ports.

Definition (Atomic DEVS model, [Zeigler, 1976])**.** *An atomic DEVS model is described by the 8-tuple* $\langle X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta \rangle$*:*

- X*, a set of input ports and their associated value domains,*
- Y*, a set of output ports and values,*
- S*, a set of internal states,*
- $\delta_{ext} : S \times X \rightarrow S$*, an external transition function, describing the evolution of the model's internal state when an external event occurs on one of the input ports,*
- $\delta_{int} : S \rightarrow S$*, an internal transition function, describing the natural evolution of the model's internal state,*
- $\delta_{con} : S \times X \rightarrow S$*, a transition conflict function, specifying the behaviour in case of a conflict between an internal and an external event (usually chooses to use* δ_{int} *or* δ_{ext}*),*
- $\lambda : S \rightarrow Y$*, an output function, updating the values on the output ports,*
- $ta : S \rightarrow \mathbb{R}^+$*, a "time advance" function, used to schedule the time of the next transition to a new internal state.*

A port-based DEVS model can be represented as on figure 11.3.

It is important to lift the terminology ambiguity between DEVS external and internal events and GSMP events. DEVS internal events correspond to changes inside an atomic model. DEVS external events can be seen as messages, travelling between DEVS models.

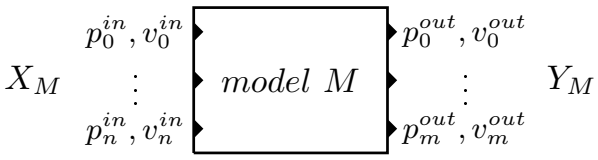


Figure 11.3: DEVS atomic model with ports

They are events that are emitted towards the other models, thus allowing model coupling. Hence, these events correspond to changes on the input ports of connected models. GSMP events are somehow a miscalling: they point to distinct processes triggering the discrete events that condition the evolution of the global system.

The temporal execution of a DEVS model can be described as follows. Initially, model M is in an internal state $s \in S_M$. The ta function is called to determine how long the system should remain in this state. If no external event arrives on an input port, at time $ta(s)$, the δ_{int} function is called and the system evolves to state $s' = \delta_{int}(s)$. Then the output function λ is called and the output ports Y are set to the value of $\lambda(s)$. ta is called again to find the next undisturbed transition date $ta(s')$. If an exogenous event with a vector v of values occurs in X_M before time $ta(s')$, then the model's state changes according to δ_{ext}. The next state is $s'' = \delta_{ext}(s', v)$. The output function is not called since there was no internal transition and the ta function is immediately called to get the new undisturbed transition time for the model $ta(s'')$. If no other exogenous event has occurred at $ta(s'')$, then, as previously, δ_{int} determines the next step of the process, λ sets the output ports' values and ta is called again.

Since DEVS models are event-driven models, they do not rely on a notion of *synchronization* on a common time. Therefore, there is no notion of time step and the models evolutions are asynchronous (they are simply coupled through the emission and reception of events). However, in order to define a sound behaviour, one has to plan the possibility of an external event arriving exactly at the transition time specified by ta. In this case, the δ_{con} function resolves the conflict: $\delta_{con}(s, v)$ determines the new state (usually by choosing to call either δ_{int} or δ_{ext})[1]. Similarly to δ_{ext}, when δ_{con} is called, the output function is not called (this behaviour can be regained by introducing intermediate volatile states).

This describes the individual behaviour of an atomic DEVS model. But the DEVS framework is also meant to authorize the parallel execution and interaction of several different models. This is where the multimodeling problematic arises: sometimes one needs to represent the discrete event process resulting from different processes where each can be specified in a different formalism (for instance, one could be described as a Petri net, another as a discrete time differential equation and a third as a cellular automata). Interfacing all these models in the DEVS framework corresponds to introducing the notion of coupling between models.

[1]It is important to note that defining the δ_{con} function as a function of the internal state s and the input port values v allows to consider several external events and one internal event occurring at the same time, thus guaranteeing the behaviour's consistency even with multiple concurrent DEVS events. The same remark holds for δ_{ext}.

11.3.3 Coupled models

A coupled DEVS model defines how individual models are coupled together in order to form a high-level macro-model. This macro-model can itself be part of a coupled model, etc.

In a coupled model (also called "network of models"), there is no additional notion of state than the abstract aggregate state of all individual models and these states remain private to each model. We provide a general definition of a coupled DEVS model. For a more formal definition, see [Zeigler et al., 2000].

Definition (Coupled DEVS model, [Zeigler et al., 2000])**.** *A coupled DEVS model is given by:*

- *a set of models,*
- *a set of input ports collecting incoming external events,*
- *a set of output ports emitting events,*
- *a set of connections between the coupled model's input and output ports and the individual model's input and output ports.*

Graphically, one can represent a coupled model as on figure 11.4.

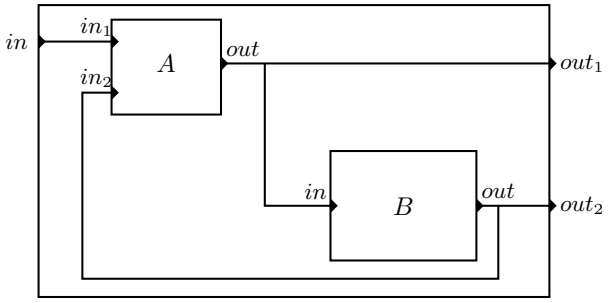


Figure 11.4: Coupled DEVS model

A lot of extensions to DEVS modeling have been developed during the last thirty years as, for instance, Cell-DEVS, which establishes a direct mapping from cellular automata to DEVS models, or DS-DEVS, allowing dynamic structure change in coupled DEVS models.

One can make a strong parallel between DEVS modeling and discrete-time asynchronous multi-agent modeling since an atomic DEVS model can be seen as an autonomous entity, owning a private state and sharing information with other models through events in a network of local connections between models. This actually highlights the main feature that will be problematic in the next section: a model's state is internal to the model and cannot be shared without emitting events, even for coupled models.

11.3.4 Abstract graphical representation

We adopt a graphical convention to represent the internal dynamics of a DEVS model. In this convention, a model is a box with its name in the upper left corner. The internal state — or only the abstract, relevant part of this state — is represented as circular nodes. The *ta* function in a given abstract state is shown in the lower half of the node, while the upper half displays the node's name. Internal transitions are shown as solid arcs between nodes. External transitions are represented using dashed arcs. Finally, the output function associated to an internal transition is shown using a solid arrow starting on the transition's arc. All the atomic DEVS models represented in the rest of the thesis follow this convention, as, for example, on figure 11.5(b).

11.4 GSMPs and DEVS models

The DEVS methodology has become a rather important modeling and simulation formalism partly because of its generality and simplicity. The last decades saw its expansion in a number of different directions. However, there are few results, both on the formalism side and on the implementation side, for the extension to stochastic processes. Some work has been done by [Melamed, 1976] and [Ahn and Kim, 1993] about mapping Markov chains to the DEVS framework, and [Joslyn, 1996] provides a nice analysis of the different aspects of qualitative DEVS models (deterministic, stochastic, possibilistic, fuzzy, ...) and their link with finite automata.

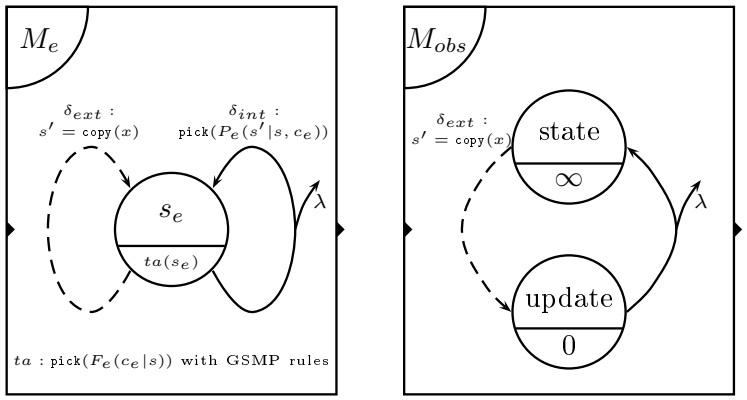
In this last section, we focus on trying to extract the discrete event system characteristics of GSMPs in order to map them to DEVS models.

The first intuition on the link between GSMPs and DEVS models is to map each concurrent entity in the GSMP framework to its DEVS counterpart. Namely, the idea would be to map each of the individual semi-Markov processes to a separate DEVS model. Since each of these SMPs captures the dynamics of one event, the global DEVS model turns out to be a DEVS representation with one atomic model per event. However, one crucial difference kicks in at this point. While all the SMPs of a GSMP depend on the same shared random variables, independent DEVS models have independent state spaces. Therefore, writing a GSMP as a collection of DEVS models, each representing an event, implies performing a *synchronization* operation between models on all the variables representing the state space.

The first architecture one can consider in order to represent GSMPs as DEVS models necessitates to define an *observer* which synchronizes the state among events. The idea of such an observer is to act as the "real world", being affected by the happening of events and back-propagating its state to the events' models.

In this architecture, each event holds a copy of the process's state variables s, or — in a more memory-efficient version — only the variables on which it depends and the variables it affects. Then, each event can be represented graphically as a model with a single abstract internal state. To avoid confusion, we will write s the natural state of the GSMP and s_e the internal state of model M_e, corresponding to event e. The state variables s_e of model M_e correspond to the ones of s (or only the fraction of variables related to e) plus the current clock of event e. The *ta* function of such a model corresponds to picking a sojourn time according to $F_e(s)$ and c_e. If c_e is equal to zero, the time is picked according to $F_e(s)$. If

c_e is not equal to zero it means another event has been triggered and the observer has sent an update message concerning s, the new c_e is updated accordingly. The internal transition function corresponds to picking the next values of the state variables according to $P_e(s'|s, c_e)$. The external transition function updates the internal state according to any incoming event from the observer. Finally, the output function generates a DEVS event containing the information about the new state[2]. The event model is represented on figure 11.5(a).



(a) Graphical model of the GSMP event in DEVS

(b) Graphical model of the GSMP state observer in DEVS

Figure 11.5: DEVS atomic models for GSMPs

Similarly, one needs to define the observer model. This model holds an internal state describing the real state s of the global process. Graphically, one can represent this model with an abstract two states model as represented on figure 11.5(b). The first abstract state called "state" is an idle position, in this state $ta = \infty$ and the model waits for an incoming exogenous DEVS event. The other "update" state is a volatile state (its ta is always equal to zero) which serves to send the DEVS event corresponding to the update of s to all M_e models. Consequently, the M_{obs} model only changes the s state upon reception of DEVS events sent by the M_e models: its external transition function consists in copying the values of the state variables received into the internal state. When in abstract state "update", the M_{obs} model instantaneously returns to "state" and uses λ to emit the current process' state s. The global architecture is summarized on figure 11.6.

Even though this representation is a sound mapping from GSMPs to DEVS models, it provides a rather bad simulator. The reason for that is the amount of communication needed between models for state synchronization; directly inherited from the complexity of strongly coupled processes. Highly communicating DEVS models yield rather inefficient simulators because they loose the distributed nature of DEVS models.

It is somehow possible to simplify the representation presented on figure 11.6. The first simplification we can introduce to reduce the communication load between models is — as mentioned in the above paragraphs — to only send to a model the set of variables it depends on and to only receive the set of variables it affects. This idea is rather close to the description of transition functions as dynamic Bayesian network introduced by [Dean and Kanazawa,

[2]This actually implies introducing an extra intermediate volatile state since λ is a function of s_e and not s'_e. But we do not mention it for clarity purposes.

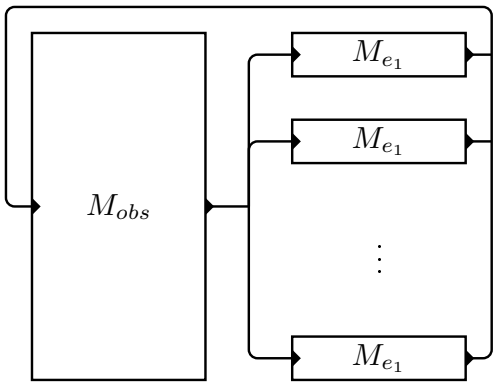


Figure 11.6: Coupled DEVS model for GSMPs

1990] for compact representation of transition models. But this simple optimization does not significantly reduce the communication load.

A second simplification consists in getting rid of the observer and directly connecting the output ports of the M_e models to the input ports of other M_e models. In this case, the state is not centralized is an observer anymore but it is still consistent as long as the connection graph remains equivalent to the version with the observer. The global state of the process then needs to be collected from the different models. However, this last simplification implies building a complex connection graph by analyzing processes' effects and dependencies.

Finally, it appears that:

Because GSMPs represent processes that are coupled through a common state space, writing a distributed, coupled DEVS model equivalent to a GSMP necessarily implies redundancy in the storage of state variables and yields a coarse communication network (connection graph) between models.

This analysis of GSMPs underline why their global behaviour appears complex, while their atomic elements remain simple. This lays the foundations for sound design of GSMP simulation engines and their coupling with other discrete event formalisms.

Consequently, if one wishes to implement a GSMP extension to a DEVS simulation engine, he has the choice of either building the safe coupled model presented above, or designing an atomic model which fully implements the GSMP behaviour. The *Virtual Laboratory Environment* platform (VLE, [Quesnel et al., 2007]) is a software and an Application Programming Interface (API) which supports multimodeling and simulation by implementing the DEVS abstract simulator. VLE is oriented toward the integration of heterogeneous formalisms as those presented earlier. Furthermore, VLE is able to integrate specific models developed in most popular programming languages into one single multimodel. We designed and implemented the GSMP extension to the VLE multimodeling and simulation platform using the option of designing an atomic model interface for GSMPs. This extension has also a stochastic decision process version, implementing the concepts of GSMDPs which will be presented in the next section, thus making a first attempt at coupling results from the field of discrete event simulation and the theory of simulation-based decision optimization.

11.5 MDPs, continuous time and concurrency

The first part of this chapter focused on modeling and simulating concurrent stochastic processes. This analysis needs to be considered under the light of decision theoretic planning: our goal is to design sound simulations of such processes in order to evaluate the choices of different actions in the systems they represent. In this section, we introduce the possibility of partially controlling the global process through action choice in GSMPs, building the Generalized Semi-Markov Decision Processes (GSMDP) framework. We highlight the main difficulty when dealing with GSMDPs: since the natural process does not retain Markov's property, there is no guarantee of optimality on Markovian policies. We discuss how to deal with this last point. We finally make time observable to the decision maker for time-dependent problems and, as in the previous part of the thesis, consider the impact such a choice on the problem definition.

11.5.1 Generalized Semi-Markov Decision Processes

Introducing action choice in GSMPs was first proposed by [Younes and Simmons, 2004] in the Generalized Semi-Markov Decision Process (GSMDP) framework. Moving from GSMPs to GSMDPs consists in separating the events into two categories: controllable and non-controllable. We identify a subset A of *controllable events* or *actions*. The remaining events are called *non-controllable* or *exogenous events*. Actions are events that can be activated or deactivated at will and the subset $A_s = A \cap E_s$ of activable actions in state s is never empty since it always contains at least the a_∞ idle action.

Definition (GSMDP, [Younes and Simmons, 2004])**.** *A GSMDP is a GSMP where some events are defined as* controllable*. At each decision epoch, a controller agent can activate or deactivate these events at will. Similarly to a a GSMP, a GSMDP is given by:*

- *its state space S,*
- *its event space E among which one distinguishes between uncontrollable and controllable events. The controllable events are called* actions *and their subset is noted A,*
- *its duration F and transition P functions,*
- *and finally a reward model given per event, specifying a lump sum reward k_e and a reward rate c_e, similarly to the SMDP case.*

This new definition of idleness is both consistent with the analysis developed in chapter 4 and with the intuitive physical meaning of performing no action. The a_∞ action always has its clock set to $+\infty$ and thus is never the first event to trigger change in the global process. Therefore, it really corresponds to letting the non-controllable, exogenous events naturally take the process to a new state. Because its clock is always set to $+\infty$, there is no need to define the transition model of action a_∞. By convention, we will write that this transition model is deterministic and does not change the state.

A GSMDP's execution follows the same rules as a GSMP. At any step of the process, all active events plus the current chosen action a have an associated clock value. The smallest clock determines the triggering event, which takes the process to state s'. In s', the decision-maker has the following choices:

- If $a \in A_{s'}$ he can choose to leave a active. In this case, the action continues concurrently with all exogenous events. It is treated as any other active event, its clock being decremented by the previous c_{e^*}.

- He can also choose to change the current action to a'. He has to do so if $a \notin A_{s'}$, but he can also choose to change actions. In this case, a is deactivated and a new clock is drawn for a'.

To summarize, at each state change, the decision-maker is asked to choose an action in A_s. Once this action is chosen, it is dealt with just as any other active event.

GSMDPs are GSMPs where one distinguishes between controllable events (actions) and non-controllable (exogenous) events, leaving the choice of activating the actions to the decision-maker.

One can notice that only one action can be active at a time. Two important comments need to be made on this point. First, this does not prevent parallel action triggering. Of course, one could relax the previous hypothesis of only one active action at a time and allow for n active actions at a time (n might be unbounded). But without relaxing this hypothesis, concurrent actions are still possible. For example, we could allow for *activation* of only one action at a time but this activation would have a clock reading of zero which would authorize the decision-maker to chain other action activations right after as long as he wishes. When he does not want to activate anymore actions, then he activates a_∞ and the process' time increases again. The main problem in this case lies in the number of combinations of activated actions / states. For a complete study of discrete-time steps, concurrent actions in the MDP case, we refer the reader to [Mausam and Weld, 2005; Mausam, 2007; Mausam and Weld, 2007].

The second comment underlines the fact that authorizing only one action *activation* at a time actually corresponds to saying that the agent focuses only on one thing at a time. But this does not prevent parallel *execution*: actions can trigger non-controllable events representing the action's effects or the action's execution which are considered exogenous because the agent is not *monitoring* them anymore (because it is focusing on the new action activation) and cannot stop them unless we introduce specific stopping actions (which takes us back to the first comment).

Consequently, the "one active action at a time" hypothesis is not a strong restriction on the model. It could nevertheless be relaxed but would yield a more complex problem because of the combinatorial growth of the action space. We will keep the above hypothesis and declare that only one action is active at a time[3].

Finally, as we briefly made the parallel between GSMPs and finite timed automata, we can make a similar link between their decision counterparts: GSMDPs and Timed Game Automata [Bouyer et al., 2004].

[3]Future work on dealing with concurrent events *and* actions in GSMDPs is an interesting line of research since the problem happens quite often in real life. Merging the results of [Mausam, 2007] on action pruning and heuristic search with the GSMDP formulation and optimization algorithms is — in my opinion — a promising approach.

11.5.2 Controlling GSMDPs

As in the MDP case, searching for control strategies on GSMDP implies defining rewards $r(s, e)$ or $r(s, e, s')$ associated to transitions and introducing policies and criteria.

The same characteristics arise with GSMDP than in the GMSP case: the transition function for the global semi-Markov process does not retain Markov's property without augmenting the state space. In other words, only the augmented process is Markovian, the natural process is not. In the classical MDP framework, one can make use of the transition function's Markov's property to prove that there exists a Markovian policy (depending only on the current state) which is at least as good as any history-dependent policy (Cf. [Puterman, 1994]). In the GSMDP case however, this is no longer possible because the natural process is not Markovian.

In order to define criteria and to find optimal policies, we need — in the general case — to allow the policy to depend on the whole *execution path* of the process. [Younes and Simmons, 2004] define execution paths for a GSMDPs. An execution path of length n from natural state s_0 to state s_n is a sequence $\rho = (s_0, t_0, e_0, s_1, \ldots, s_{n-1}, t_{n-1}, e_{n-1}, s_n)$ where t_i is the sojourn time in state s_i before event e_i triggers. As in [Younes and Simmons, 2004], we define the discounted value of ρ by:

$$V_\gamma^\pi(\rho) = \sum_{i=0}^{n-1} \gamma^{T_i} \left(\gamma^{t_i} k(s_i, e_i, s_{i+1}) + \int_0^{t_i} \gamma^t c(s_i, e_i) dt \right) \tag{11.1}$$

where k and c are traditional SMDP lump sum reward and reward rate functions[4], and $T_i = \sum_{j=0}^{i-1} t_j$.

One can then define the expected value of policy π in state s as the expectation over all execution paths starting in s:

$$V_\gamma^\pi(s) = E_s^\pi \left[V_\gamma^\pi(\rho) \right] \tag{11.2}$$

This provides a criterion for evaluating policies. The goal is now to find policies that maximize this criterion. The main problem here is that it is hard to search the space of history-dependent policies. So the simplest solution would be to make our process Markovian again by using the supplementary variable technique and then to search for an optimal policy in the space of Markovian policies defined over the augmented state space.

Using the supplementary variable technique consists in augmenting the natural state space with just enough variables so that the distribution over future augmented states only depends on the current values of these variables. As in [Nielsen, 1998], we can augment the natural state s of the process with all the clock readings and show that this operation brings Markov behavior back to the GSMDP process. We will note this augmented state space (s, c) for convenience.

Unfortunately, as foreseen at the end of the previous section, it is unrealistic to define policies over this augmented state space since clock readings contain information about the *future* of the system. From here, several options are possible:

[4] see equation 2.26 for details on k and c.

- One could decide to sacrifice optimality and to search for "good" policies among a restricted set of policies, for example the policies defined on the current natural state only.

- One could also search for representation hypothesis that simplify the GSMDP model and that make natural state Markovian again.

- One could compute optimal policies on the augmented state space (s, c) and then derive a policy on observable variables only.

- Finally, one could look for a set of *observable* variables which retain Markov's property for the process, for example the set composed of the natural state of the process s, the duration for which each active event e_i has been active τ_i and its activation state s_i. We will note this augmented state (s, τ, s_a).

This series of options leads to consider the question of observability in the stochastic process. Namely, it is important to know which variables are observable and what prior knowledge we have concerning the relationship between observations and the real state of the process. The field of Partially Observable MDPs (POMDPs, [Kaelbling et al., 1998]) studies this question in detail. We won't enter the question of partial observability in MDPs here and will consider the natural state to be fully observable.

The optimization approach taken by [Younes and Simmons, 2004] is based on the second option listed above. Namely, the authors approximate any distribution on time by a chain of exponential distributions called a *phase-type distribution*. These distributions, as presented in [Neuts, 1981], allow to fit any number of known moments of a prior distribution using only chains of exponential distributions. Once this approximation is made, they introduce extra states for the intermediate steps (the *phases*) corresponding to the inner states of the chains. Using the memoryless property of exponential distributions, this brings the GSMDP back to a time-homogeneous continuous time MDP (CTMDP). Then, they perform uniformization, as in [Puterman, 1994], in order to transform the CTMDP into a standard MDP which can be solved using standard discrete MDP methods.

This approach necessitates to be able to approximate the duration functions with phase-type distributions. On top of that, it is limited to discrete natural states. We wish to avoid — as much as possible — making hypothesis on the model itself and on the underlying distributions. Moreover, many of the variables we will consider are continuous variables or a mix of continuous and discrete variables. We also need, for the problems at hand, to consider methods for policy search which can handle state spaces with large dimensions. Consequently, in the list above, we will look for a solution corresponding either to option 1, 3 or 4.

Because of the non-Markov behaviour of GSMDPs' natural process, there is no guarantee that there exists an optimal policy verifying Markov's property. On the other hand, searching for a policy in the space of history-dependent policies is not an acceptable solution. Hence, one needs to make a choice between:

- Sacrificing optimality by restricting the policy search space.
- Finding the correct minimal number of observable variables to add to the natural process in order to regain Markov behaviour.
- Constructing policies on non-observable variables and then use a priori knowledge to derive policies on observable variables.

11.5.3 Introducing continuous observable time in GSMDPs

Finally we need one more brick to build the modeling part of concurrent, time-dependent stochastic decision processes. We need to reintroduce continuous observable time into the process.

This operation is quite straightforward. Suppose that our GSMDP has a state space including the continuous observable time variable. Then the GSMDP events affect this time variable as any other variable and time increments correspond to the clock readings of the triggering events.

The only problem lies in the definition of the criterion. As with TMDPs, we should check that Bellman's equation is still valid. But we know that the augmented state's stochastic process retains Markov property (as for GSMPs). So when we include continuous observable time, the global process turns to an XMDP and the results of chapter 8 apply. Consequently, optimizing a policy on the augmented state process of a GSMDP turns to optimizing a policy on an XMDP. Finally, we can safely use the standard Bellman equation since GSMDPs with observable time are simply equivalent to XMDPs with a hidden part of the state.

In general, as for a GSMP, the augmented state space is not observable and we know that natural process does not retain Markov's property. Therefore, we are left with the options listed in the previous paragraph.

11.6 Conclusion

All this chapter has been devoted to capturing the complexity of temporal stochastic decision processes. We put a special emphasis on concurrency because it appears to be a major modeling obstacle for temporal systems. Recent articles in the planning community, such as [Cushing et al., 2007] for instance, point out similar conclusions as to the importance of concurrency and the notion of decision time for discrete event systems.

Finally, we conclude that GSMDPs with continuous observable time are an elegant and efficient framework for capturing compactly the dynamics of problems such as the coordination, the subway or the airport management problems. These problems present other characteristics which make the idea of building a full, explicit model, unpractical: they have many variables, inducing a high-dimensional state space, some of these variables are discrete, others are continuous, these problems have a finite (eventually sliding) horizon. Furthermore, the overall behaviour of the stochastic processes described by GSMPs or GSMDPs does not

retain Markov's property.

Concerning this last point, we make the hypothesis that knowing the exact values of the clocks is not crucial to building good policy. This hypothesis comes from the fact that if the general values of clocks are small compared to the horizon, then the absolute time-dependency will have more influence on the optimal policy than the values of clocks. Therefore, if our hypothesis is relevant, we can try to find policies defined over the natural state space even if the associated process is not Markovian anymore.

The difficulty of writing a synthetic model for such processes leads to another conclusion. Standard inference of a policy from the explicit model becomes very hard in the cases at hand because of the difficulty to obtain this explicit model and then the difficulty to extract information from it. Therefore, instead of relying on standard MDP techniques, we turn to Reinforcement Learning techniques and choose to use our GSMDP representation as a *generative model* to simulate and evaluate policies in order to learn a good controller.

This approach is one of the core ideas of the next chapters.

12

Real-Time Policy Iteration

The idea of performing direct policy search, using the sound simulation basis introduced in the previous chapter for Monte-Carlo policy evaluation, is closely related to the algorithm of Policy Iteration. We leave the GSMDP framework for a while as we introduce an algorithm which holds the essential properties of the method presented in the next chapters. As seen through the previous example of Prioritized Sweeping, Dynamic Programming becomes really efficient when cleverly guided in its exploration of the search space. Real-Time Dynamic Programming (RTDP, [Barto et al., 1995]) is one of the most efficient family of algorithms designed to solve MDPs. It relies on heuristic guidance and on asynchronous updates of the value function. After reviewing the basis of Asynchronous Dynamic Programming, we introduce a Policy Iteration version of RTDP which we call Real-Time Policy Iteration (RTPI). While RTDP and its variants are all based on Asynchronous Value Iteration, RTPI searches for a solution directly in policy space.

It can seem odd to abruptly introduce such a chapter. There is indeed no continuity between the previous chapter's ideas and this one. Actually, from a chronological point of view, this chapter should come last in the thesis since it is the abstraction and the generalization of the Approximate Temporal Policy Iteration (ATPI) algorithm which will be introduced later in the document. However, ATPI is designed for Temporal Markov Decision Problems and will result from putting together a whole set of ideas, methodologies and algorithms. Among these ideas, one will find the one of incrementally building local policies, with an optimization procedure guided by greedy simulations and updates. This idea is independent of the planning domain at hand (temporal problems) and the global presentation of the thesis becomes easier when separate ideas are presented separately. Furthermore, the idea of Real-Time Policy Iteration goes beyond the scope of the thesis and deserves an independent chapter alone. Therefore, this chapter will introduce the general idea of the Real-Time Policy Iteration algorithm in the general MDP case. This algorithm is inspired by a single intuition:

In an incremental, direct policy search algorithm, given an initial state or a set of possible initial states, to obtain efficient and quick policy improvements, the relevant states for policy update are the ones we are likely to encounter during execution of the current policy, *ie.* the ones visited by policy simulation.

This idea is very close to the idea behind the Asynchronous Value Iteration RTDP algorithm, thus providing the name of RTPI.

The progression of ideas in this chapter is as follows. We start by reviewing the idea of asynchronous Bellman backups in section 12.1 and focus more specifically on Policy Iteration. This leads us to discuss the question of approximation in Policy Iteration in section 12.2. Then we get back to our main point: we review the greedy exploration algorithm of RTDP in section 12.3 and show how we can build such a simulation-based algorithm with the RTPI method of section 12.4.

12.1 Asynchronous Dynamic Programming

For brevity, we do not recall in detail the Value Iteration and Policy Iteration algorithms which were presented in chapter 2. The goal of this first section is to introduce the notion of *Asynchronous Dynamic Programming*, to provide a first illustration before focusing on Asynchronous Policy Iteration and finally to show which questions it raises.

12.1.1 Origins of Asynchronous Dynamic Programming

The bottomline idea of Asynchronous Dynamic Programming is the fact that ([Bertsekas and Tsitsiklis, 1996]):

> As long as every state is chosen for Bellman backups infinitely often, the overall value function converges to V^*.

These Bellman backups can be performed in value function space (Asynchronous Value Iteration) or in policy space (Asynchronous Policy Iteration). The initial idea for asynchronous dynamic programming underlined the possibility to use parallel computation to reduce calculation time. Illustrating the local aspect of Bellman backups and the independence of convergence properties on state ordering was crucial for this purpose. Later, the idea of using specific ordering for state updates spawned a whole family of algorithms.

Section 6.3 already introduced asynchronous value iteration as a specific asynchronous dynamic programming algorithm. This fundamental result underlines the fact that the ordering of Bellman backups has little importance as long as they cover the whole state space infinitely often (as the number of iterations tends to $+\infty$). In other words, since we know that standard value or policy iteration converges within an ϵ bound of the optimum in a certain number of state updates with a given ordering of these updates, asynchronous dynamic programming expresses the fact that this number of updates can be reduced by finding the right ordering on states.

Prioritized Sweeping is a *backward asynchronous dynamic programming* method. It propagates the change information from children states to their parents, defining priorities based on the amplitude of this change. Therefore, it focuses the optimization on back-propagation of rewards to the whole state space. Reversely, one could take the problem differently, by working from an initial state and trying to focus on states that are likely to lead to the goal. This other approach is related to *forward dynamic programming search*[1] and we will focus

[1] One could build an adapted version of Prioritized Sweeping which heuristically updates states according to their distance to the start states. This implies biasing the priorities, using a balance between real priorities and a notion of distance to the initial state. To the best of our knowledge, this option has not been explored in the framework of Prioritized Sweeping. Similar approaches of backward-forward heuristic search for MDPs were investigated in [Teichteil-Königsbuch and Infantes, 2008] for example and these algorithms benefit largely of their heuristic guidance. Their analysis is beyond the scope of this chapter.

on this feature in section 12.3.

12.1.2 Asynchronous Policy Iteration

Reinforcement Learning borrowed many of its initial intuitions to the learning behavior of living species, trying to mimic the learning process induced by sequences of trial and errors. In particular, we, as humans, usually keep track of our behavior (policy) instead of its expected reward (value function), even though we have a rough idea of what our behavior is worth. Therefore, we perform some sort of direct policy reinforcement, changing our behavior when it seems that a new action can improve our "expected gain". This idea underlies the approach of Policy Iteration which we review throughout this chapter.

We first recall the basics of the Policy Iteration algorithm and highlight its main drawback: the evaluation phase. We postpone the discussion considering the set of approximate evaluation methods for the policy, defining variants of Approximate Policy Iteration to section 12.2. Instead, we first focus on the specification of an Asynchronous Policy Iteration algorithm.

Let us start with a reminder on Policy Iteration.

Reminder on Policy Iteration

Policy Iteration (Cf. [Bertsekas and Tsitsiklis, 1996; Puterman, 1994]) is a dynamic programming method which operates directly in policy space. It can be summarized by saying that if one has a current policy π_n and is able to exactly evaluate the expected gain V^{π_n} of this policy in every state of the process, then performing a Bellman backup in state s with respect to V^{π_n} corresponds to finding a better or equivalent action a in s than the one specified by $\pi_n(s)$. Therefore, replacing $\pi_n(s)$ by a yields a new policy π_{n+1} which has a better or equivalent expected reward. Consequently, Policy Iteration jumps from policy to policy in the policy space and from value functions to better value functions in the value function space.

We recall below the Policy Iteration algorithm as presented in algorithm 2.2. It alternates two phases: the policy evaluation phase and the improvement phase. The evaluation phase consists in evaluating exactly the policy's value function V^{π_n} (without any optimization). The improvement phase sweeps through the state space, updating the action in every single state by performing a Bellman backup based on V^{π_n}, and builds the new policy π_{n+1}.

Evaluating the policy can be done in a number of ways. For example, one can use the Value Iteration algorithm without the maximization step (namely, using the L^π operator instead of L) in order to build a sequence of functions converging to V^π. Organizing the Bellman backups and focusing on relevant states was actually the first idea behind the prioritized sweeping method of [Moore and Atkeson, 1993] since it was first introduced for Markov prediction problems and later extended to Markov decision tasks. Another option consists in performing explicit matrix inversion in order to solve the linear system of equations $V^\pi = L^\pi V^\pi$. This kind of resolution can exploit the fact that transition matrices are generally rather sparse, thus allowing significant improvements in matrix inversion.

Algorithm 12.1: Policy Iteration

$\pi_0 \in \mathcal{D}$
$n \leftarrow 0$
repeat
  Solve the system of $|S|$ equations:
  $\forall s \in S \quad V_n(s) = r(s, \pi_n(s)) + \gamma \sum_{s' \in S} p(s'|s, \pi_n(s)) V_n(s')$
  for $s \in S$ **do**
    $\pi_{n+1}(s) \leftarrow \underset{a \in A}{\operatorname{argmax}} \left(r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V_n(s') \right)$
  $n \leftarrow n + 1$
until $\pi_n = \pi_{n-1}$
return V_n,π_n

However, the evaluation phase usually remains the bottleneck for most Policy Iteration methods.

Therefore, one often uses Approximate Policy Iteration which allows for faster approximate evaluation and despite the lack of theoretical guarantees. We discuss this question in section 12.2.

Asynchronous Policy Iteration

Dynamic Programming is not limited to Value Iteration. Building an Asynchronous Policy Iteration algorithm seems a little more complicated in the first place because of the two distinct phases of the standard Policy Iteration Algorithm. We use this section to review the idea of asynchronism in Policy Iteration.

In the following paragraphs, we will make the — rather drastic — assumption that there exists a black box which quickly evaluates the policy's value function. This assumption is made for clarity of presentation and we will discuss it along with the Approximate Policy Iteration methods in the next sections.

The algorithm of Modified Policy Iteration provides a smooth transition from Policy Iteration to Asynchronous Policy Iteration. It builds on the idea that one can use other value functions than the evaluation of π_n to find π_{n+1} as long as the value function used respects some properties. Namely, the evaluation phase of Modified Policy Iteration at iteration n consists in performing m_n times the $V_{k+1} = L^{\pi_{n+1}} V_k$ operation in order to approach $V^{\pi_{n+1}}$. [Puterman, 1994] shows that Modified Policy Iteration converges for any non-zero value of m_n.

Because of the alternance of evaluation / improvement phases, Policy Iteration seems to be a Synchronous Dynamic Programming method by nature. Introducing asynchronism in Policy Iteration implies allowing these two phases to mix, therefore performing partial evaluation and partial improvements of the policy. The case of Modified Policy Iteration is a good illustration of the fact that Asynchronous Policy Iteration necessarily relies on some sort of approximation in the policy's value function. In the case of Modified Policy Iteration, this approximation has no impact on optimality while with less conservative approximation methods it might necessitate the error bounds of Approximate Policy Iteration.

As for Value Iteration, Policy Iteration can be made more efficient when the local Bellman backups are performed asynchronously and in a relevant order.

[Bertsekas and Tsitsiklis, 1996] lay the basis of Asynchronous Policy Iteration. At iteration n, we select a subset S_n of S and perform a policy Bellman backup on all $s \in S_n$. This yields policy π_{n+1} with:

$$\pi_{n+1}(s) = \begin{cases} \underset{a \in A}{\operatorname{argmax}}\, r(s,a) + \gamma \sum_{s \in S} P(s'|s,a) V^{\pi_n}(s') & \text{if } s \in S_n \\ \pi_n(s) & \text{if } s \notin S_n \end{cases} \tag{12.1}$$

One can also similarly perform a certain number of $V_{k+1} = L^{\pi_n} V_k$ operations to update the value function on the states of S_n.

Hence, if one alternates one policy update and one value function update then the latter is equivalent to a Value Iteration update over the S_n states. Similarly, if the number of value function updates is unbounded, we obtain the standard Policy Iteration method. Finally, if we alternate one policy update and m_n value function updates, we obtain the Modified Policy Iteration algorithm.

[Bertsekas and Tsitsiklis, 1996] prove that if the initial policy and value function verify $V_0 \leq L^{\pi_0} V_0$ and if value function and policy updates are performed infinitely often in all states as n tends to $+\infty$, then V_n converges to V^* and the policy converges to an optimal policy.

Finally, one can see Asynchronous Policy Iteration as an elegant way of formulating both Value and Policy Iteration algorithms. It also naturally introduces the use of approximate value functions for V^π and helps distinguishing between "conservative" methods (Modified Policy Iteration, matrix inversion) and "less conservative" methods (approximate policy evaluation) to analyze convergence and optimality. Asynchronous Policy Iteration can be similarly presented from the point of view of actor-critic architectures.

12.2 Approximation for Policy Iteration

We have mentioned several times the possibility — and sometimes the need — for approximate policy evaluation methods. This section discusses the drastic assumption we made earlier about the existence of an evaluation black box and presents the different architectures of *Approximate Policy Iteration*.

12.2.1 Why Policy Iteration?

Let us start with a common sense question: why would one prefer a Policy Iteration method to a Value Iteration one? There is no particular reason for the choice of Policy Iteration against Value Iteration in general. Experience shows that exact Policy Iteration might converge in less iterations but more time (because of the evaluation phases) than Value Iteration, but this rule of thumb does not always apply and the time taken by the evaluation phase quickly becomes prohibitive.

Value Iteration methods have also received more attention in the Planning community because of their efficient representation of reward-to-go functions and the ease of manipulation of value functions. These value functions often have good properties, such as convexity,

monotonous evolution across the iterations, etc. On top of that, using value functions as a unified way of storing information facilitates the construction of asynchronous methods for value function optimization and allows to use results from heuristic search.

However, in order to make problems tractable, one often turns towards approximation schemes. Part II is a good illustration of how value functions can be more complex objects than policies. [Anderson, 2000] analyses why approximating a policy can be easier than approximating a value function. By comparing Q-learning (Cf. [Watkins, 1989]) and the direct gradient algorithm of [Baxter and Bartlett, 1999], both based on neural networks approximators, Anderson presents an example where Q-learning oscillates between the optimal policy and a suboptimal policy, while the direct policy search method converges to the optimal policy. As mentioned in the conclusion of the above paper, such an illustration does not support any general conclusion about the relative merits of policy-only versus value functions methods.

However, it suggests that it might be relevant to examine the complexity of approximating value functions or policies for the problem at hand in order to choose the way we represent the agent's strategy.

Since value functions often hold more information than policies, they might be harder to approximate with good enough granularity. Dedicating the resources of a function approximator to representing relevant and irrelevant value function variations can be useless with respect to the final policy and can lead to non-convergence of algorithms or degradation of good policies.

Policy Iteration is basically an algorithm which explicitly stores the policy. However, it is often used in conjunction with value function storage, in order to facilitate policy evaluation. This feature of being both a policy and value function based algorithm yields Policy Iteration's robustness (the ability to actually find a good policy) but also its long execution time because of the alternance of updates on the policy and value function. While the optimization is performed directly on the policy, it needs to be propagated to the value function during the evaluation phase, yielding the drawbacks of Policy Iteration methods. Section 12.1.2's analysis of Asynchronous Policy Iteration underlined even more the coupling between value function and policy.

For the arguments presented above and similarly to the first ideas of chapter 9, our approach has turned towards approximate Policy Iteration methods and towards direct improvement of the decision variables.

12.2.2 Convergence of Approximate Policy Iteration

As mentioned earlier, exact Policy Iteration converges in practice in less iterations than Value Iteration but usually takes more time because of the evaluation phase's computational cost. Thus, as for Value Iteration, it is common to use approximation schemes for this evaluation phase. This approximation's goal is to reduce the complexity of a policy's evaluation while still trying to fit its value function as closely as possible.

Similarly to the Value Iteration case, one has a few results for Approximate Policy Iteration. The first of these results being that, for the same reason as presented in section 6.4:

Approximate Policy Iteration usually does not converge.

Depending on the approximation's quality, the first iterations yield a close-to-optimal policy which then oscillates around the optimal policy. The previous section's argument was that this policy might oscillate "less" than the approximate value function itself and therefore is more robust to approximation.

In the case of discounted problems, [Bertsekas and Tsitsiklis, 1996] show that if we write the approximation error (the critic's error) as:

$$\exists \epsilon \in \mathbb{R}^+ \ / \ \forall f \in \mathcal{F}(S, \mathbb{R}), \ \|Ap(f) - f\|_\infty \leq \epsilon \tag{12.2}$$

then we can write equation 12.3.

For discounted problems, one can bound the optimality loss due to approximation by:

$$\limsup_{k \to \infty} \|V^* - V^{\pi_k}\| \leq \frac{2\gamma\epsilon}{(1-\gamma)^2} \tag{12.3}$$

We can even be a little more precise and write that:

$$\limsup_{k \to \infty} \|V^* - V^{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \left(\sup_{j \leq k} \|Ap(V^{\pi_j}) - V^{\pi_j}\|_\infty \right) \tag{12.4}$$

In the case of undiscounted Stochastic Shortest Path problems, a similar bound exists, provided that ϵ is small enough. To establish this bound, one needs to introduce the ρ_π quantity defined in equation 12.5. This ρ_π is the maximum probability that the process is in a non-goal state s after $|S|$ steps of applying π, starting in a non-goal state.

$$\rho_\pi = \max_{s \notin GoalStates} Pr(s^{|S|} \notin GoalStates | s^0 = s, \pi) \tag{12.5}$$

If we consider the sequence of policies generated by the Approximate Policy Iteration algorithm, we can introduce ρ_k:

$$\rho_k = \sup_{j \leq k} \rho_{\pi_j} \tag{12.6}$$

And similarly, for all *proper* policies, *ie.* for all policies such that $\rho_\pi < 1$ (policies that eventually lead to the goal with probability one), one can define ρ:

$$\rho = \sup_{\pi \in ProperPolicies} \rho_\pi \tag{12.7}$$

Since, for ϵ small enough, all policies are proper (Cf. [Bertsekas and Tsitsiklis, 1996]), one can write:

$$\limsup_{k \to \infty} \|V^* - V^{\pi_k}\|_\infty \leq \frac{2|S| \left(1 - \rho + |S|\right) \epsilon}{(1-\rho)^2} \tag{12.8}$$

The results from [Munos, 2003] generalize these results to the case of weighted quadratic norms. This is of crucial interest since many approximation techniques for value functions solve a regression problem defined in terms of L_2 norms[2].

[2] A good counter-example is provided in [Guestrin et al., 2001], where an L_∞ norm is used.

12.2.3 Approximation methods

Linear approximation architectures

A first set of Approximate Policy Iteration methods can be grouped under the name of "feature-based approximations" or, most commonly "linear approximation architectures". Even though all regression methods are more or less related to feature-based representations, this specific category uses a predefined finite set of feature functions. The idea is to represent the value function as a linear combination of features and thus to project the value functions (or the Q-functions) onto the subspace spanned by the features as illustrated by equation 12.9.

$$V^{\pi}(s) = \sum_{i=1}^{k} w_i^{\pi} \phi_i(s) \tag{12.9}$$

The fixed degree polynomial approximations of part II fall into this category but not the piecewise polynomial approximation since one cannot exhibit a finite basis for the space of bounded degree piecewise polynomial functions. The linear approximation architecture has been used for instance in the Least-Squares Temporal Difference Learning (LSTD, [Bradtke and Barto, 1996]) algorithm for prediction tasks. This same method inspired the evaluation phase of the Least-Squares Policy Iteration (LSPI, [Lagoudakis and Parr, 2003]) algorithm. An other example of a direct linear approximation architecture is the approach of Approximate Linear Programming (ALP, see [Hauskrecht and Kveton, 2004] for example) which can be used for policy optimization or simply policy evaluation. These approaches provide a robust evaluation phase and help build efficient Policy Iteration algorithms, both from the model-based (planning) and the model-free (learning) point of view. Their main drawback lies in feature selection as pointed out by [Kveton and Hauskrecht, 2006]. The two next families of algorithms try to overcome this difficulty.

Simulation-based methods

We distinguish a second family of methods which we could call "Monte-Carlo methods" or "simulation-based methods" in the sense that they do not rely on a value function approximation architecture but on direct simulation and sampling to obtain an evaluation of the considered random variables. It is important to note that the families of algorithms we distinguish are closely related to each other: our point is not to categorize and separate algorithms but to provide a structured review of existing approximation methods. For instance, the LSTDQ evaluation in LSPI relies on the reusability of samples generated from the exploration versus exploitation trade-off. Monte-Carlo methods make extensive use of generative models, *ie.* suppose that generating samples and experience can be done at a very low cost.

Simulation-based approaches are quite close to the online approaches of RTDP or LAO*. The algorithm of [Kearns et al., 2002] for instance explores the reachable states from a current state s by simulating N times each action and repeating until a certain depth H. This recursively defines value functions for horizon 1 to horizon H. Then, the best action found in s is returned. This method is however quickly handicapped by the complexity of breadth-first search and it is hard to reach sufficiently large values of H and N to guarantee optimality and convergence. A more focused alternative is what [Bertsekas and Tsitsiklis, 1996] presents as simulation-based policy evaluation. It consists in calculating all Q^{π}-values

of actions starting in s by simulating a followed by the current policy until a certain horizon and then returning the action corresponding to the best Q-value. This was exploited in the Rollout method of [Tesauro and Galerpin, 1997]. These results were also analyzed in [Péret and Garcia, 2003; Péret and Garcia, 2004; Péret, 2004] and can be reused for any evaluation phase or for simulation-based Value Iteration.

The question of simulation and sampling for multistage adaptive algorithms was introduced also in [Bertsekas and Tsitsiklis, 1996]. The idea is — similarly to [Kearns et al., 2002] — to solve an m-stage look-ahead problem and to use the result in the online setting, *ie.* to only apply the action found for the current state s. Recent work of [Chang et al., 2007] shades a different light on this topic and makes the link with population-based evolutionary approaches.

Structured representation methods

The last category of methods we distinguish is related to the idea of compactly representing the value function through an efficient approximation architecture. Again this approach is related to the previous families of algorithms since it builds on the same idea as the linear approximation architecture but tries to build structured representations which do not depend on features (or automatically learn the features) and that adapt to samples.

Among such structured representations, one can mention methods based on trees and especially randomized trees as in [Ernst et al., 2005]. [Whiteson and Stone, 2006] explores the use of evolutionary functions for Reinforcement Learning. Finally, [Ormoneit and Sen, 2002] adapts the Bellman equation to approximate kernel-based representations and shows how regression methods applied to Reinforcement Learning are subject to estimation biases.

> All these methods provide a panel of approximation and regression techniques to evaluate policy's values. The choice among these techniques (and eventually others) depends on the application at hand and is closely related to the question of dynamically learning the structure of the problem, the value function and the policy.

Such techniques can be used independently or in conjunction with Approximate Policy Iteration. All use the bounds defined in [Bertsekas and Tsitsiklis, 1996] and [Munos, 2003], proving that Approximate Policy Iteration is a sound direct-policy search method.

12.3 Heuristic forward search for Asynchronous Value Iteration

In section 12.1.1, we mentioned the possibility to build an Asynchronous Value Iteration algorithm by focusing the value functions updates on the states which are likely to lead to the goal. This *forward search* approach starts from the initial state and tries to work its way towards the goal, relying on an initial heuristic guidance.

12.3.1 Real-Time Dynamic Programming

Real-Time Dynamic Programming (RTDP) was introduced by [Barto et al., 1995], based on this last idea of forward search in the state space. It is comparable to [Korf, 1990]'s LRTA*

algorithm.

RTDP is a Value Iteration algorithm which tries to optimize a policy for an MDP given an initial state and a heuristic value function.

RTDP starts with an initial state and a heuristic function which is used to initialize the value function. The repeated operation of RTDP can be summarized as:

- In state s, perform one Bellman backup with respect to the current value function of children states.
- Update $V(s)$ and apply the best action found to reach s'.
- Repeat in s'.

More specifically: the class of problems RTDP was first designed for are the undiscounted stochastic shortest path problems. The algorithm itself is defined in terms of trials. An RTDP run is composed of a sequence of trials, each starting in the initial state s_0 and ending in a goal state. An RTDP trial is the result of the above operations as presented on algorithm 12.2. Namely:

In each state s, one performs a Bellman backup, updating the Q-values, choosing the best action a to perform and updating the value function in s. Then, the next state to update is picked according to the distribution $P(s'|s,a)$. Whenever a goal state is encountered, a new trial is started.

Algorithm 12.2: Real Time Dynamic Programming

RTDP(state s, value function h)
$V(s) \leftarrow h(s)$
repeat RTDPtrial(s) **until** convergence of the value function

RTDPtrial(state s)
while $s \notin GoalStates$ **do**
  $a \leftarrow \underset{a \in A}{\operatorname{argmax}}\, r(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot s'.\text{value}$
  $s.\text{value} \leftarrow s.\text{Qvalue}(a)$
  $s \leftarrow \text{pickNextState}(s,a)$

[Barto et al., 1995] call *relevant states* the states reachable from s_0 by an optimal policy. [Bonet and Geffner, 2003b] restrict this definition to the states reachable from s_0 with the unique optimal policy; this policy being defined by adding a static ordering on actions in order to bring out a single policy out of the set of optimal policies. If the goal is reachable from every state of the process and if the heuristic used is admissible (ie. is an upper bound of the optimal value function), then one has the following results:

- $V_n(s)$ is a monotonous, decreasing sequence.
- RTDP trials terminate in a finite number of steps (Cf. [Bertsekas and Tsitsiklis, 1996]).
- $V_n(s)$ eventually converges to $V^*(s)$ in all relevant states.

The first and third above properties illustrate the Value Iteration oriented nature of RTDP.

This notion of *relevant state* is crucial to improving the effectiveness of dynamic programming methods in practice. Many problems described as MDPs present very large state spaces, suffering from Bellman's *curse of dimensionality*, while in the end, the execution of an optimal policy, given an initial state, only visits a small subset of the state space. Therefore:

> For problems where the initial state is known, finding these relevant states and organizing dynamic programming passes so that these states are updated often is a crucial step towards convergence speed-up.

In other words, finding the *relevant states* in forward search dynamic programming is similar to finding to highest priority state in backward search algorithms such as prioritized sweeping. It corresponds in the end to letting the optimization adapt to the structure of the problem and to prior heuristic knowledge about the domain if such knowledge is available. This is basically what RTDP does by letting the best action found so far in state s guide the choice of the next state s' to update.

RTDP suffers from the problem of asymptotic convergence. The number of trials needed to reach V^* is not bounded. For example, unprobable states are rarely visited because RTDP focuses on states that are likely to be encountered. Therefore, termination of RTDP is usually given in terms of finding an ϵ-optimal policy over the relevant states or in the initial state. This termination usually uses the criterion of having a Bellman residual smaller than ϵ.

12.3.2 Labeled RTDP: asynchronous backward-forward Dynamic Programming

As [Bonet and Geffner, 2003b] and [Bonet and Geffner, 2003a] point out, RTDP has a very good anytime behaviour, it quickly finds good policies, provided that it was initialized with a good heuristic. However, the smooth improvement of this policy and the final convergence are slow. This is a consequence of RTDP's exploration strategy: greedy simulation. Greedy simulation focuses on states which are likely to be encountered, therefore quickly yielding a good policy. But the finer improvement of this policy implies considering states that are less likely to be visited. Hence, RTDP is handicapped by the same feature that gave it its good anytime behaviour.

In order to improve the exploration strategy, several options are possible. Q-learning or TD-learning for example (Cf. [Watkins, 1989; Watkins and Dayan, 1992; Sutton, 1995; Sutton and Barto, 1998]) introduce noise in action choice. Labeled RTDP (lRTDP), introduced by [Bonet and Geffner, 2003b], take a different approach: they keep track of the states having converged by labeling them and letting RTDP focus on the rest of the state space. A state is said to have converged, or to be solved, whenever the associated Bellman residual is smaller than a given ϵ. Since the most likely states will be found and updated often very early in an RTDP run, it leaves all the further computing resources available for the convergence of the other, less probable states.

Labeled RTDP works on the same idea as RTDP. An lRTDP run includes repeating lRTDP trials until the initial state is labeled as having converged as shown on algorithm 12.3. An lRTDP trial is — similarly to the RTDP case — a trajectory in the state space

where successive states are updated and where the transition from a state to another is conditioned by the updated greedy action.

lRTDP trials are not only stopped when a goal state is reached: they can also be stopped earlier if a state labeled as having converged is encountered. This allows to avoid spending time updating states for which an ϵ-optimal value function has already been found.

In a sense, it is very similar to putting priorities on the states as in Prioritized Sweeping.

At the end of each trial, the stack of visited states is unpiled and the `CheckSolved` procedure is called for each of them. The idea of this procedure is to construct the *greedy envelope* for a given state, up to a certain depth corresponding to the first states having a Bellman residual larger than ϵ. This greedy envelope in state s is the set of all reachable states from s with the current greedy policy. These states are the nodes of what [Bonet and Geffner, 2003b] call the greedy graph. `CheckSolved` does not really build the full greedy envelope: instead it performs a Depth First Search in the greedy graph in order to find all the fringe states having a residual greater than ϵ. Whenever such a state is found, its siblings are not checked (the node is not expanded) and itself is put in the "closed" stack, thus avoiding the complete exploration of the greedy graph. These states are the fringe states which are not solved yet in the reachable graph from s. If all states in the greedy envelope have a Bellman residual of less than ϵ, they are labeled accordingly as solved and `CheckSolved` returns true. Else, the procedure updates the value function in each of the states in the "closed" stack and `CheckSolved` returns false.

Since `CheckSolved` is called in reverse order of visit (it is called on the last visited state first) it means it tries to label these last states as solved first. If it does not succeed, at least it performs an additional update in the unsolved fringe states of "closed" before returning false.

Therefore, `CheckSolved` acts as a backward propagation method trying to let the states that are closer to the goals converge first. Thus, with the trials acting as a forward search propagation and the `CheckSolved` procedure focusing on backward dynamic programming updates, lRTDP is a complete forward-backward algorithm. On top of that, the labeling procedure avoids spending extra time on solved states, thus allowing the algorithm to focus on less probable states and to converge in a bounded number of trials.

As soon as a state returns false when `CheckSolved` is called on it, the sequence of `CheckSolved` calls is stopped and a new trial is entered.

12.3.3 Related approaches and extensions

lRTDP is by many ways comparable to the LAO* algorithm of [Hansen and Zilberstein, 2001]. Recent variants of RTDP include the HDP algorithm of [Bonet and Geffner, 2003a], Focused Dynamic Programming of [Ferguson and Stentz, 2004], Bounded RTDP of [McMahan et al., 2005] and Focused RTDP of [Smith and Simmons, 2006]. It is interesting to note that planners built on the heuristic search ideas of RTDP and on reachability analysis with structured (forward-backward) update propagation provide most of the state-of-the-art MDP planners, as for example [Teichteil-Königsbuch and Infantes, 2008] (winner of the 2008 International Planning Competition, probabilistic track).

Algorithm 12.3: Labeled Real Time Dynamic Programming

lRTDP(state s, value function h, float ϵ)
$V(s) \leftarrow h(s)$
repeat lRTDPtrial(s, ϵ) **until** s.solved

lRTDPtrial(state s, float ϵ)
$visited = \emptyset$
while $\neg s$.solved **do** `/* Play a trajectory */`
  $visited$.push(s)
  if $s \in GoalStates$ **then** break
  $a \leftarrow \underset{a \in A}{\text{argmax}}\, r(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot s'$.value
  s.value $\leftarrow$ s.Qvalue(a)
  $s \leftarrow$ pickNextState(s, a)
while $\neg visited$.empty() **do** `/* Unpile the` $visited$ `stack */`
  $s = visited$.pop()
  if $\neg$CheckSolved(s, ϵ) **then** break

CheckSolved(state s, float ϵ)
$solved = true$
$open = \emptyset$
$closed = \emptyset$
if $\neg s$.solved **then** $open$.push(s)
while $open \neq \emptyset$ **do** `/* Build the` $closed$ `stack */`
  $s = open$.pop()
  $closed$.push(s)
  if s.residual $> \epsilon$ **then** $solved = false$; `/* Found an unsolved state */`
  else
    $a \leftarrow \underset{a \in A}{\text{argmax}}\, r(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot s'$.value `/* Expand state */`
    foreach s' such that $P(s'|s,a) > 0$ **do**
      if $\neg s'.solved \wedge s' \notin \{open \cup closed\}$ **then** $open$.push(s')
if $solved = true$ **then** `/* All the greedy graph is solved */`
  foreach $s' \in closed$ **do** s'.solved$= true$
else
  while $closed \neq \emptyset$ **do** `/* Update unsolved states */`
    $s = closed$.pop()
    $a \leftarrow \underset{a \in A}{\text{argmax}}\, r(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot s'$.value
    s.value $\leftarrow$ s.Qvalue(a)
return $solved$

12.4 Real Time Policy Iteration

Having swept through the Policy Iteration family of methods in sections 12.1 and 12.2, we can now try to adapt the greedy simulation guidance of RTDP to a Policy Iteration framework which would retain the base properties of Asynchronous Policy Iteration and make use of the Approximate Policy Iteration results.

12.4.1 Using greedy simulation to select S_n

The crucial steps of Asynchronous Policy Iteration are the choice of the S_n subset and the decision to switch from policy improvement to policy evaluation. If we completely rely on a policy evaluation black box, then the choice of S_n alone becomes predominant. Seen from this point of view, one can see RTDP as an Asynchronous Value or Policy[3] Iteration method where S_n is chosen by simulating the greedy policy obtained so far.

> Applying the greedy policy simulation idea to the selection of the S_n subset for undiscounted stochastic shortest path problems provides the first idea of the Real-Time Policy Iteration algorithm presented in algorithm 12.4.

Algorithm 12.4: Real-Time Policy Iteration

RTPI(state s, policy π)
repeat RTPItrial(s) **until** convergence of the policy

RTPItrial(state s)
while $s \notin GoalStates$ **do**
  s.updateQvalues()
  $\pi(s) \leftarrow \underset{a \in A}{\text{argmax}}\, s.\text{Qvalue}(a)$
  $s \leftarrow \text{pickNextState}(s, \pi(s))$

This RTPI algorithm relies on greedy simulation to select the S_n subset but neglects the problem of policy evaluation. Actually, it presents the problem slightly differently: instead of requiring the expected current value of policy π, RTPI looks for the expected Q-value of action a, in state s, with policy π. This leaves us with a number of possibilities for this evaluation black box. This decoupling between policy improvement and policy evaluation is to be related to the actor-critic architecture [Sutton and Barto, 1998]: the actor of RTPI uses greedy simulation to select the states to update, while the critic can be implemented in an independent way.

The straightforward calculation of $Q^\pi(s,a)$ can be done through direct calculation of V^π at each step, using matrix inversion or prioritized sweeping for example[4]. In this case, the updateQvalues() function first updates V^π and then calculates the Q-values:

$$Q^\pi(s,a) = r(s,a) + \sum_{s' \in S} P(s'|s,a) V^\pi(s') \tag{12.10}$$

[3] Asynchronous Value Iteration is a special case of Asynchronous Policy Iteration which alternates single passes of policy update and value function update on S_n as explained in section 12.1.2.

[4] The latter should probably be preferred since — for discounted criteria — local changes in the policy have a local impact on the value function which locally propagates to other states. The extent of this propagation depends partly on the value of γ.

This way of computing Q^π is consistent with the initial idea of [Bertsekas and Tsitsiklis, 1996]: one could have two independent real-time threads in the optimization program, one for the actor, one for the critic. The actor performs greedy exploration with respect to the latest V^π available and asks regularly for specific values of V^π to calculate the Q^π values. In the meanwhile, the critic permanently updates its evaluation of V^π, based on the latest policy available. This way, there is no fixed number of iterations of each procedure but simply an interleaving of the actor and the critic execution which uses the most up-to-date information for the Asynchronous Policy Iteration scheme of RTPI.

One can also choose to only update the value function once at the beginning of each trial instead of updating it at each state update. In this case, RTPI uses an approximate value function in the same way as Modified Policy Iteration. This is particularly pertinent for time-dependent problems as the next section will illustrate.

Then, different approximation schemes for policy evaluation can be used. Projecting the value function on a subspace of *features* and performing linear regression among these features borrows to the evaluation phase of Approximate Linear Programming (ALP, [Guestrin et al., 2004; Kveton and Hauskrecht, 2006; Hauskrecht and Kveton, 2006]). The idea of Least Squares Policy Iteration (LSPI, [Lagoudakis and Parr, 2003]) is similar and uses LSTDQ as the approximation phase. Similarly, by using a generative model one can perform Monte-Carlo evaluation to directly obtain the $Q^\pi(s, a)$ values, as in Simulation-based Policy Iteration of [Bertsekas and Tsitsiklis, 1996].

Finally, heuristic guidance for RTPI can be provided by either a value function (in this case the considered policy is greedy with respect to this value function) or by an initial policy. Different features for value function approximation and heuristic evaluation and guidance can be combined to build an efficient evaluation phase. For example one could combine Monte-Carlo sampling with UCB-based bounds (Cf. [Auer et al., 2002; Kocsis and Szepesvari, 2006; Coquelin and Munos, 2007]) to perform both efficient greedy simulation guidance and admissible policy evaluation.

12.4.2 Evaluating π, the specific case of time-dependent problems

The specific case of time-dependent problems can take advantage of its structure for RTPI runs. Having an explicit continuous time included in the state space implies respecting *causality* in the transition functions. Therefore, the only transition allowed are transitions to states which have the same current time or a posterior time. Loops in the state space are possible but since, in practice, an infinite number of instantaneous transitions has probability zero, the possibility of an infinite loop is excluded for time-dependent problems.

This has important consequences on the resolution. The previous case of TMDPs illustrated that since we have a limited knowledge of time-dependency and since time is explicitly included in the state space, it makes sense (under some hypotheses) to consider total reward criteria.

Consequently, since we cannot loop to the past — we only make transitions to the future and the present, since we have total reward criterion and since we are performing forward search exploration of the state space, we can deduce that updating the policy in state s during a trial will not change the policy's value function in the next state s'. The only exception to this rule is for instantaneous transitions.

Thus, we can consider the value function calculated at the beginning of each trial as valid all along the run, even when the policy is updated.

12.5 Conclusion

Presenting the idea of RTPI in a separate chapter was important because its application reaches beyond the scope of temporal Markov decision problems. We tried to present RTPI as an extension of RTDP, Asynchronous DP and existing algorithms, shading a different light on the question of "how can we perform efficient asynchronous Policy Iteration?".

No experiments where directly performed on this idea, however, the ATPI algorithm presented in the next chapter is an instance of an RTPI algorithm. Therefore, this chapter serves both as a general introduction to RTPI and as first element in constructing the ATPI algorithm.

We now turn back to the framework of temporal Markov decision problems. Studying and improving RTPI (through labelling schemes, action elimination, heuristic function discovery, adaptation to hybrid spaces, generalization schemes, etc.) is beyond the scope of this thesis but is a very strong topic of interest in future research.

13

Simulation-based local incremental policy search for observable time GSMDPs: the ATPI algorithm

Solving high-dimensional temporal Markov decision problems presents many difficulties. The first difficulty dealt with writing the problem down in the first place. Chapter 11 provided some hindsight on the inherent structure underlying the complexity of such temporal processes. This highlighted the fact that these processes where not necessarily Markovian, often had a large number of variables and were easy to model if one broke them into atomic elements; but the coupling between concurrent events remained the core reason of the global process' complexity. Consequently, these complex processes are simple to simulate and to capture into a *generative model* but hard to integrate into a single global implicit-event *predictive model*.
Based on the GSMDP formulation with observable continuous time and on the idea of Real-Time Policy Iteration, we design an algorithm based on Approximate Policy Iteration to construct policies for such problems. This algorithm relies on simulation-based exploration and evaluation and on statistical learning regression techniques to construct the value functions. We present initial results on this ATPI algorithm and illustrate its main weakness, thus motivating the improved version of the following chapter.

13.1 General idea

Temporal Markov decision problems modeled as GSMDPs represent a class of problems which is both hard to model and to solve. First, they are hard to model because of the non-Markov behaviour of the global *natural process*. Then, even if the process itself retained Markov property, these problems would be hard to represent as an explicit "$p(s'|s)$" process because of the complexity resulting from the concurrent interaction of their local coupled temporal processes. Lastly, they are hard to model, because they often involve hybrid state spaces, mixing continuous variables such as time or energy with discrete ones as subway station number or passenger count and boolean ones as mission definition flags.

This modeling complexity is found again when one tries to solve problems defined as GSMDPs with continuous observable time. This chapter introduces our contribution to the problem of solving high-dimensional, hybrid, stochastic temporal problems. We design an algorithm based on Policy Iteration which uses greedy simulation for exploration of the state

space. This algorithm also builds on the ideas of statistical learning to extract as much information as possible from simulations. Finally it exploits the presence of an observable continuous time in the flavor presented at the end of the previous chapter.

The problems we wish to represent as GSMDPs and for which we want to find good policies involve a large number of variables. If one artificially includes the GSMDP clocks into the state space to make the process Markovian, this number of variables increases even more. On top of these features, integrating the GSMDP process into a single, explicitly defined, stochastic process is a complicated task while simulating a pair "GSMDP + policy" seems more tractable. Therefore, we turn to simulation-based evaluation, exploration and learning in order to locally and incrementally improve policies for the temporal Markov problems at hand.

The general idea of the Approximate Temporal Policy Iteration (ATPI) algorithm we introduce in this chapter can be summarized as:

> For observable time GSMDPs for which the initial state is known, we perform local improvements of the policy in the states visited by the greedy simulation of the best policy found so far. Since our problem uses a total reward criterion, every run through the state space and until the horizon provides a realization of the "reward-to-go" random variable in each of the visited states. We use statistical learning tools to generalize this information in order to build an approximate value function for the current policy. Then this generalized value function is used to perform local Bellman backups at the next trial.

In other words, we will:

- perform partial exploration of the state space, guided by greedy policy simulation,
- collect rewards, as samples in the state space of the random variables $R^\pi(s)$ (reward-to-go),
- use these samples to construct a regression of the last run's policy's value function,
- use this value function to improve the policy during the next trial.

Additionally, in order to gather enough samples to build a relevant regression for the value function, we run the exploration trials several times with respect to the same value function, thus obtaining multiple evaluations of the greedy policy and directly building the new policy's value function.

13.2 Approximate Temporal Policy Iteration

13.2.1 Algorithm overview

The initial version of the Online Approximate Temporal Policy Iteration (online-ATPI) algorithm was introduced in [Rachelson et al., 2008b]. Algorithm 13.1 summarizes the essential steps of online-ATPI which we develop below.

The main loop of online-ATPI performs trials in order to build a training set. These trials are sample paths, through the state space, guided by the execution of the greedy policy with respect to $\tilde{V}$. In other words, if one starts with a policy π_n and its associated

Algorithm 13.1: Online-ATPI

main:
input: π_0 or $\tilde{V}_0$, s_0
repeat
  $TrainingSet \leftarrow \emptyset$
  for $i = 1$ to N_{sim} **do**
    $\{(s, v)\} \leftarrow \texttt{simulate}(\tilde{V}, s_0)$
    $TrainingSet \leftarrow TrainingSet \cup \{(s, v)\}$
  $\tilde{V} \leftarrow \texttt{TrainApproximator}(TrainingSet)$
until termination

$\texttt{simulate}(\tilde{V}, s_0)$:
$ExecutionPath \leftarrow \emptyset$
$s \leftarrow s_0$
while horizon not reached **do**
  $a \leftarrow \texttt{ComputePolicy}(s, \tilde{V})$
  $(s', r) \leftarrow \texttt{GSMDPstep}(s, a)$
  $ExecutionPath \leftarrow ExecutionPath \cup (s', r)$
convert execution path to value function $\{(s, v)\}$
return $\{(s, v)\}$

$\texttt{ComputePolicy}(s, \tilde{V})$:
for $a \in A$ **do**
  $\tilde{Q}(s, a) = 0$ **for** $j = 1$ to $N_{samples}$ **do**
    $(s', r) \leftarrow \texttt{GSMDPstep}(s, a)$
    $\tilde{Q}(s, a) \leftarrow \tilde{Q}(s, a) + r + \gamma^{t'-t}\tilde{V}(s')$
  $\tilde{Q}(s, a) \leftarrow \frac{1}{N_{samples}}\tilde{Q}(s, a)$
return $\underset{a \in A}{\text{argmax}}\, \tilde{Q}(s, a)$

approximate evaluation $\tilde{V}_n$, then the action applied in each state s is the action $\pi_{n+1}(s) = \underset{a \in A}{\text{argmax}} \left[r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \tilde{V}_n(s') \right]$. This way, the values put in the $TrainingSet$ are samples of $R^{\pi_{n+1}}$.

Once a set of trials has been completed, the training set is passed to a statistical regression method in order to build an interpolating function which has the properties of:

- compactness: it stores the information in a memory-saving fashion,
- accessibility: it can easily answer value requests,
- generalization: it presents local smoothness, generalizing the values to their neighborhood in the state space.

The choice of this regression method is important for the performance of the algorithm and will be discussed with the results.

13.2.2 Greedy simulation for exploration

The function used to build the $TrainingSet$ is the `simulate`$(\tilde{V}, s_0)$ function. This method starts from state s_0 and simulates the optimal greedy action at each step by calling the `ComputePolicy`$(s, \tilde{V})$ and `GSMDPstep`(s,a) procedures. After collecting all the samples (s', r) from the execution path, the `simulate`$(\tilde{V}, s_0)$ procedure performs a cumulative sum, starting from the horizon and moving backwards in time, in order to build the $\{(s,v)\}$ set. This set contains realizations of the random variable $R^{\pi_{n+1}}(s)$ which is the reward-to-go variable. This cumulative sum's calculation is straightforward in the case of undiscounted criterion. For a discounted criterion, this sum uses equation 11.1. One has:

$$V^{\pi_{n+1}}(s) = E(R^{\pi_{n+1}}(s))$$

So, as the value of N_{sim} tends to $+\infty$ the average value of state s in the $TrainingSet$ tends to $V^{\pi_{n+1}}(s)$.

> ATPI performs sampling in the state space along paths defined by the greedy policy's simulation. These trajectories provide a set of samples corresponding to the reward-to-go random variable in each state for the greedy policy.

Similarly to the RTDP case, one major advantage of performing policy-driven simulation is that the policy guides the exploration of the state space to the states most likely to be visited. Thus we refine the training set over the *relevant states*, having the largest probability of being reached by the policy. This provides us with a second advantage: this rollout technique is adapted to sampling in large dimension state spaces without suffering from the *curse of dimensionality*.

13.2.3 Simulation-based policy evaluation

The `GSMDPstep`(s, a) procedure follows the discrete events systems paradigm applied to GSMDPs. It activates (or maintains) the a event as a concurrent event to all the current GSMDP's exogenous active events and triggers the first transition (not necessarily caused by a), taking the process to a new state. Therefore, this `GSMDPstep`(s, a) function really corresponds to performing one simulation step of the GSMDP controlled by the current policy. This consists in:

- activating a (plus deactivating any other previous action, or maintaining a if it was previously active),
- triggering the event with the smallest clock.

The choice of the optimal greedy action in the `simulate`$(\tilde{V}, s_0)$ procedure is made by calling the `ComputePolicy`$(s, \tilde{V})$ function. We push the logic of Monte-Carlo sampling in the same way that [Bertsekas and Tsitsiklis, 1996] did and consider that since we do not have a *predictive model* of our system, even the one-step greedy action needs to be simulated in order to obtain its Q-value. Consequently, we simulate each of the available actions $N_{samples}$ times in order to obtain its Q-value and use these approximate Q-values to select the best action as shown in the last part of algorithm 13.1.

> Generating the samples for $\tilde{V}$ and for $\tilde{Q}$-values are two separate processes. Q-values are obtained through the use of $\tilde{V}$ and of one-step simulation. Whereas the samples for the next $\tilde{V}$ are generated by collecting successive rewards in the global greedy simulation process.

It is important to separate the set of samples generated for the $TrainingSet$ and the Q-values computed for action selection. The $\{(s, v)\}$ values come from the cumulative rewards really obtained during the last run. Whereas the $\tilde{Q}(s, a)$ are obtained by simulating a single step in the GSMDP and using function $\tilde{V}_n$. These $\tilde{Q}(s, a)$ thus correspond to approximate values for $Q^{\pi_n}(s, a)$ while the $\{(s, v)\}$ values correspond to sampled values for $V^{\pi_{n+1}}(s)$. Having separated these two sampling processes, the important point here is to note that the value of $\tilde{V}_n$ helps choosing the greedy action but never affects the value of samples used for $\tilde{V}_{n+1}$.

> The data fed to the regression method in order to compute $\tilde{V}_{n+1}$ is independent of previous approximation errors for $\tilde{V}_n$ since it results from real experience of interaction between the greedy policy and the simulator.

Since we are using sampling and regression, our approach falls into the category of "less conservative" approximate Policy Iteration methods presented in chapter 12 and thus, there is no theoretical guarantee of termination in terms of optimality. However, in practice, we will see that we can track the expected value of the initial state and stop the algorithm when we are satisfied with the value obtained. Hence, we do not provide a termination condition for the algorithm.

13.2.4 Value function regression

Once simulation has provided the set of samples in the space of valued trajectories, we want to use it as a training set for a regression method that will generalize it to the entire continuous state space. Several approaches to regression-based reinforcement learning have been proposed in the machine learning community: methods based on neural networks (starting with [Bertsekas and Tsitsiklis, 1996]), trees [Ernst et al., 2005], evolutionary functions [Whiteson and Stone, 2006], kernel methods [Ormoneit and Sen, 2002], etc. We chose to focus — in a first time — on support vector machines (SVM) because of their ability to handle the large dimension spaces over which our samples are defined.

SVM belong to the family of kernel methods and can be used for both regression (SVR) and classification (SVC). Training a standard SVR over a given set of samples corresponds to looking for a hyperplane interpolating the samples in a higher dimensional space called *feature space*. Practically, SVMs take advantage of the *kernel trick* which avoids expressing the feature space explicitly. Namely, a *kernel* is the result of a dot product in the feature space. For a detailed presentation on support vector regression, we refer the reader to [Vapnik et al., 1996] or [Smola and Schölkopf, 1998]. We also provide a short overview of SVR in appendix B.

The important feature of using regressors for the value function estimation, compared to the simulation-based Policy Iteration methods of [Bertsekas and Tsitsiklis, 1996], [Kearns et al., 2002] or [Tesauro and Galerpin, 1997] lies in two facts. Firstly, we deal with continuous state spaces, thus, for continuous probability distributions, there is a null probability to visit the same state twice when simulating our policy. Consequently, we are only interested in the information carried by our samples if we are able to generalize this information, at least locally. But state space continuity is a constraint of a specific version of the exploration problem. Similar problems occur with high-dimensional discrete state spaces. In fact, we suppose there is an underlying structure in the value function which needs to be inferred from our sampling. More specifically, finding this structure corresponds to finding which states have similar values than the samples and how the value function can be represented compactly. This notion of generalization is independent of the state space continuity, this last argument only reinforces the need for regressors. Consequently:

> We use Support Vector Regression in order to interpolate the value function between our valued trajectory samples. By doing so, our goal is to build a statistically sound value function, defined over the large dimension state spaces of GSMDPs and expressing in a compact form the local properties of the value function.

13.2.5 Online policy instantiation: Policy Iteration without policy storage

For the large state spaces of our continuous-time GSMDPs, even with a reasonably good policy evaluation, it might be very long or impracticable to compute the one-step improvement of the policy. Indeed, most of the time, computing a complete policy is irrelevant since most of this policy will never be used for the simulation-based evaluation step. Instead, it might be easier to compute *online* the best greedy action in the current state, with respect to the value function. This feature characterizes ATPI as an RTPI algorithm.

In a standard Policy Iteration method, the optimization step consists in solving equation 13.1 in every state of the process. In the case of Asynchronous Policy Iteration, it corre-

sponds to solving this equation in every state of the S_n subset, which, for ATPI, is defined incrementally by greedy simulation.

$$\pi_{n+1}(s) \leftarrow \arg\max_{a \in A} \tilde{Q}_{n+1}(s,a) \tag{13.1}$$
$$\text{with: } \tilde{Q}_{n+1}(s,a) = r(s,a) + \sum_{s' \in S} P(s'|s,a)\tilde{V}_n(s,a)$$

Since the states of S_n are generated "on the fly" by the simulation process, they are not known in advance and one cannot optimize the policy over these states as a "batch" improvement. Instead, at the end of the evaluation phase, the value function $\tilde{V}_n$ is stored and no policy is computed from it. A new improvement phase is immediately entered and whenever the policy π_{n+1} is asked for the action to perform in the current state s, it performs *online* the estimation of all Q-values for state s and then chooses the best action to undertake. We call this online instantiation of the policy "online approximate policy iteration", thus the "online" prefix in the algorithm name.

An important property of online policy instantiation is that, in the end, the policy is never explicitly stored. It is defined on the fly in each visited state and evaluated after a set of trials when enough samples have been collected to infer the $\tilde{V}_{n+1}$ value function.

For continuous state spaces, computing exactly π_{n+1} implies being able to compute integrals over P and $\tilde{V}_n$. Since we wish not make hypothesis on our model, we reuse the simulation engine with the GSMDPstep(s,a) function in order to sample the value of $Q(s,a)$ from the process by performing one-step simulations as presented in section 13.2.3.

13.2.6 What about Markov's property?

We have seen in chapter 11 that the stochastic process of the *natural state* of a GSMDP controlled by a given policy did not retain Markov's property. In the previous paragraphs however, the policy and value functions we defined only use state variables belonging to this natural state (since the events' clocks are often not observable during execution). This can seem paradoxical since we only know that there exists a deterministic Markovian optimal policy for Markovian process.

As discussed in section 11.5.2, we can take several options concerning the optimality of a policy defined only on the natural state of the process.

In this first version, we chose to make the assumption that the optimal policy does not depend on the clock values, or at least that the clocks' influence on the optimal policy are negligible.

This assumption is justified by the fact that we expect event clocks to have approximately comparable values and that simulating several times the global process' behavior between time zero and the horizon performs some averaging on the local behavior induced by the clocks.

Therefore, we abandon even more the idea of finding an optimal policy by only considering the natural state variables, but hope this same optimal policy does not depend too much

on the clocks' values.

There is however, an important distinction to be made between:

- The *observable* variables upon which we build the policy and the value function. These variables are the variables one can really measure during execution and input to a controller to decide which action to undertake.
- The *Markovian* variables which are purely internal to the simulator. Among these variables, one can find the process' clocks. Without these variables, one could not simulate the process since they are necessary to predict the next simulation step. Note that this set of variables is not necessarily unique and equivalent simulators can be designed with more or less efficient sets of Markovian variables.

The Markovian variables are never accessible to the policy or the value function. However, they are stored inside the simulation engine and the complete state of the simulator is always defined by these hidden Markovian variables. This has two important consequences.

The first consequence is that knowing the observable state s_{obs} is never sufficient to predict the next observable state of the process. Indeed, to predict this next state, one would need the complete internal state of the simulator. This brings the conclusion that our simulator needs to have one important property: one must be able to make a duplicate of it, in order to run simulations initialized from the current state, without affecting the central process itself. This duplicate should conserve the current state of the "real" process. This is necessary for the two calls made to `GSMDPstep`(s,a) in algorithm 13.1. During the first call (in `simulate`$(\tilde{V}, s_0)$), the step is taken inside the "real" process in order to let it advance towards the horizon. In the second call (in `ComputePolicy`$(s, \tilde{V})$), a duplicate of the process is made first because the simulation ran from the current state does not concern the evolution of the "real" process: it serves to calculate the Q-value of the current action considered. In other words, in order to call the `GSMDPstep`(s,a) procedure, one actually:

- makes a copy of the current process in observable state s,
- activates event a,
- asks the simulator to go one step forward.

This is consistent with the online-ATPI algorithm presented in algorithm 13.1 since this `GSMDPstep`(s,a) procedure is only called *online*, *ie.* in the current state of the simulator. Thus, one does not need to explicitly observe the full state of the process, the only requirement is that the simulator be "copy-able".

> Because online-ATPI performs online estimation and improvement based on simulation, it is not necessary to observe the *Markovian* state to simulate the process. However, a weaker condition must still be satisfied: one must be able to "clone" the simulator in its current state in order to run simulations without affecting the global process' state.

The second consequence we develop here derives naturally from the observations made above. We know there is an underlying Markov process which is only partially observable and drives the global behaviour of the observable variables, hence, it makes sense to look for

the hidden Markov model of these hidden variables and to derive a policy on these variables instead of the observable variables. The field of studying Hidden Markov Models [Rabiner, 1989; Cappé et al., 2005] provides mathematical foundations and tools for such problems. However, our focus goes to assembling the simulation-based method of ATPI with all its components and we will keep the previous assumption of low impact of the clocks on the policy in the current case. This remains an open area of research and interest which goes beyond the scope of this thesis and should be addressed in future work.

13.2.7 Continuous or hybrid state variables?

Finally, it is interesting to remark that we apparently only deal with continuous variables in the ATPI algorithm, while we stated that the problem at hand presented a hybrid state space.

Indeed, we assimilate every variable to a continuous (or a set of continuous) variable(s), using the following conversion rule from discrete to continuous:

- If the variable's discrete values are ordered, then we make a direct mapping to the corresponding continuous variable. Typical examples are the remaining fuel level or the number of passengers in a station. For the latter, we "artificially" introduce the possibility of 3.74 passengers by considering the variable as continuous but we should keep in mind that the interpolating value function will only be asked for the value at integer points of this variable since these points come from simulation.
- Else, a variable with d unordered values is mapped to the higher dimensional space of 2^p with $p \leq d$, in order to preserve a notion of equivalent distance between values. For example, mission advancement flags will be represented by as many boolean variables as needed, all mapped to the $[0, 1]$ interval.

The idea behind such a transformation from hybrid towards continuous is to preserve the notion of distance. One can define a distance of 3 between "42 passengers" and "45 passengers", similarly to a distance of 7.2 between time 16.5 and time 23.7. However, one cannot define an ordering or a distance between "recharge mode 1", "recharge mode 2" and "recharge mode 3". For such boolean variables, the only distance we define is a distance of 1 between "recharge mode 1" and "not recharge mode 1", thus projecting this initial variable having 3 unordered values into the $[0, 1]^3$ space.

All the samples collected during simulation are samples defined for the exact values of the discrete variables (no values appear in-between since these values are sample states from the simulator). Similarly, the inferred value function is always asked for values in real states since these states are sampled from the simulator as well. Consequently, we train the regressor with only the discrete values for previously discrete variables and we query the regressed function only in these discrete values again.

> This way, our option of only considering continuous variables corresponds to the idea of introducing "virtual" values between discrete ones, which are used uniquely for the purpose of building the regression. These virtual continuous variables maintain a notion of distance between states which respects the initial distances between hybrid states.

In other words, we transform the initial hybrid state space into a continuous metric space. This increases even more the dimension of the process' state space. We rely on simulation-based exploration to counter the *curse of dimensionality* and on SVR to handle the large

vectors of sampled state variables.

Conclusion

Finally, ATPI is an approximate Policy Iteration method, alternating phases of partial, approximate evaluation of the current policy — through the use of simulation-based sampling and support vector regression — and phases of local policy improvement in a subset of states chosen by greedy simulation.
Because of the observable time variable's presence which avoids loops, the policy needs not be stored after each action improvement in order to build the $V^{\pi_{n+1}}$ value function.
When applying ATPI to GSMDPs, we chose to focus on the natural state variables even though the associated process does not retain Markov's property.

13.3 First results with ATPI on the subway problem

We implemented and ran ATPI on an instance of the subway problem. This section illustrates the first results obtained. In a first part, we introduce the problem's modeling, then we present various optimization results and finish by explaining why ATPI is an incomplete optimization method, justifying the improvement of chapter 14.

13.3.1 The subway problem

Consider the problem of organizing a subway network from the daily point of view of the network manager. This manager only has a few actions available: he can only decide to add trains on the subway lines if there are still available trains in the garage or he can decide to remove trains from the lines when they reach the station just before the terminus. His goal is to balance the cost of running the network by the benefit from ticket sales. In order to make his decision, he has a simulator of the network. This simulator is a discrete-event system described as a GSMDP. The natural state of this GSMDP concerns the number of passengers present in each station and in each train, the current position of each train in the network and the time of day. This GSMDP is driven by events of different nature which can be divided into three categories: passengers arrival in the stations, train movements, manager's actions.

In the simple version of the problem we will consider here, the network has a single subway line, six stations and four trains. This already yields a rather complex problem with strongly coupled events. Adding more lines and trains is mainly a matter of adding more variables and events. The subway problem's state space is summarized in table 13.1.

Since there are 6 stations and 4 trains, these variables yield a state space of dimension 21.

The process defined over this state space is driven by the set of events described in table 13.2. Most of the events listed in table 13.2 have deterministic effects on all the variables but t. The most complex event to describe is mt_j. This event summarizes three sequential phases in a single event. First the train moves to station $p_j + 1$, then a certain number of passengers leave the train and lastly, passengers from the station board the train. These three phases occur without any possible interruption so we decided to model them into a

variable name	value domain	variable description
ns_i	$\{0,\ldots,50\}$	Number of passengers waiting for a train in station i ($i \in \{0,\ldots,5\}$). The maximum capacity of a station is 50.
nt_j	$\{0,\ldots,50\}$	Number of passengers onboard train j ($j \in \{1,\ldots,4\}$). The maximum capacity of a train is 50.
p_j	$\{0,\ldots,5\}$	Current position (station number) of train j.
f_i	$\{0,1\}$	These boolean variables indicate whether station i is "free" or not. Even though these variables are redundant with the p_j variables, they facilitate the process coordination.
t	$[0,1440]$	Time of day, in minutes

Table 13.1: Subway problem — state space

single event but it would still be possible to break them in three different consecutive events of the GSMDP. The movement phase is deterministic with respect to the post-action value of p_j. The number of passengers going down is simulated from a normal law based on the percentage $\rho(i,t)$ of passengers wishing to go to station i at time t. Drawing a percentage value from this model provides the fraction of passengers leaving the train. Once the travellers have left the train, the ones waiting in the station can board, as long as the train is not full. If the train becomes full, the remaining passengers stay in the station and wait for the next train.

Event durations are provided with the f function of each event and — depending on the event — can be deterministic of stochastic. To keep this description short and simple we will only state that the specification of the GSMDP follows common-sense rules: movement actions are only possible if the next station is free, a train sent to the garage lets its passengers leave before entering the garage, etc. The events listed in table 13.2 constitute a set of 19 coupled events (27 if one splits the movement events into their three components).

For the reward model, we introduce three parameters allowing to define how "economically hard" the problem is. These parameters are summarized in table 13.3, they are the train cost rate, the lump sum cost of starting a train and the ticket price.

The initial values presented in table 13.3 correspond to a rather hard economical problem in the sense that if, for example, one sets the four trains to run all day long, then the overall cost is $4 \times 1440 \times 2 = 11520$ which implies that the subway needs to transport $11520/1.5 = 7680$ passengers to be profitable with this strategy.

13.3.2 Optimization results

The version of ATPI we implemented makes use of the LIBSVM C++ library of [Chang and Lin, 2001]. We used the standard ϵ-insensitive SVR where the only parameter to tune is the SVR covariance matrix. LIBSVM uses as the defaut an isotropic covariance matrix of $\sigma^2 I$ for which we chose $\sigma^2 = 20$ after having scaled all our values between zero and one.

event name	controllable?	event description
ap_i	no	Arrival of passenger in station i. This increases ns_i by one, except if the station is already saturated.
mt_j	no	Moves train j. Increases p_j by one (modulo 6), sets f_{p_j} to zero and f_{p_j+1} to one. Also changes the number of passengers inside the train and in station $p_j + 1$ by letting the passengers in and out of the train.
ad_j	yes	Puts train j online from the garage. Train j starts running at station 1. This action is only available is train j indeed is in the garage and station 1 is free.
re_j	yes	Sends train j to the garage. This action is only available if train j is in station 0.
a_∞	yes	The no-op action, lets the first exogenous event to trigger take the process to a new state.

Table 13.2: Subway problem — event list

variable name	value used	variable description
c	-2	Cost of running a single train, per time unit.
d	-2	Lump sum cost of moving a train out of the garage.
tc	1.5	Ticket cost, this is the reward obtained by the manager every time a passenger *leaves* the subway network.

Table 13.3: Subway problem — reward parameters

Simulation of trials and outputs of rewards were performed by interfacing ATPI with the VLE simulation engine of [Quesnel et al., 2007] for which we had developed the GSMP and GSMDP simulation extensions. The experiments were ran on a 1.7GHz single core processor with 884 MB of RAM.

The parameters used for ATPI were chosen so as to perform $N_{samples} = 15$ one step lookahead state samples per action and $N_{sim} = 20$ trials before recomputing the SVR value function.

Figures 13.1 to 13.3 present the results of running ATPI when the value function is initialized with the SVR obtained by running a first exploratory set of trials without optimization, thus providing an estimate of the initial policy.

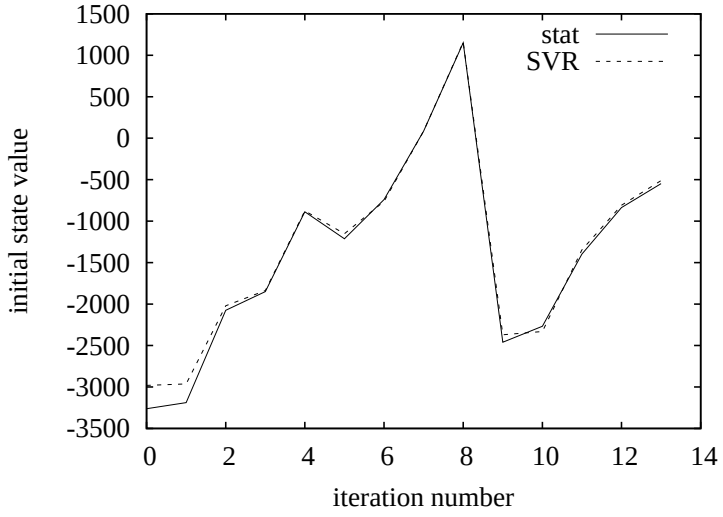


Figure 13.1: Subway optimization — Policy quality

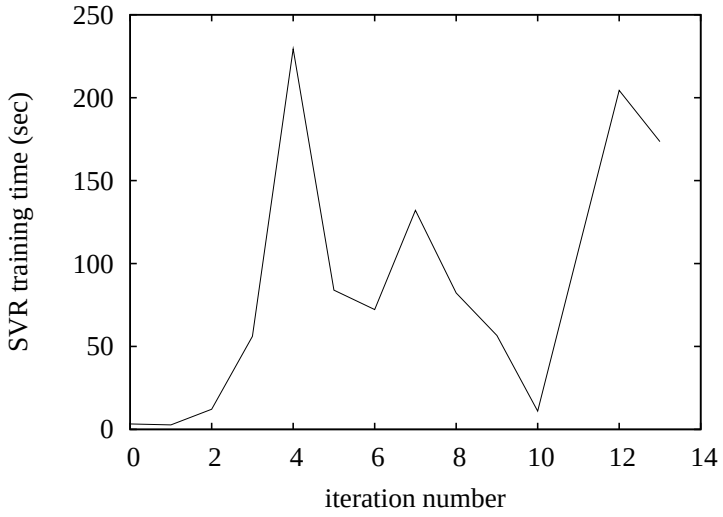


Figure 13.2: Subway optimization — SVR training time

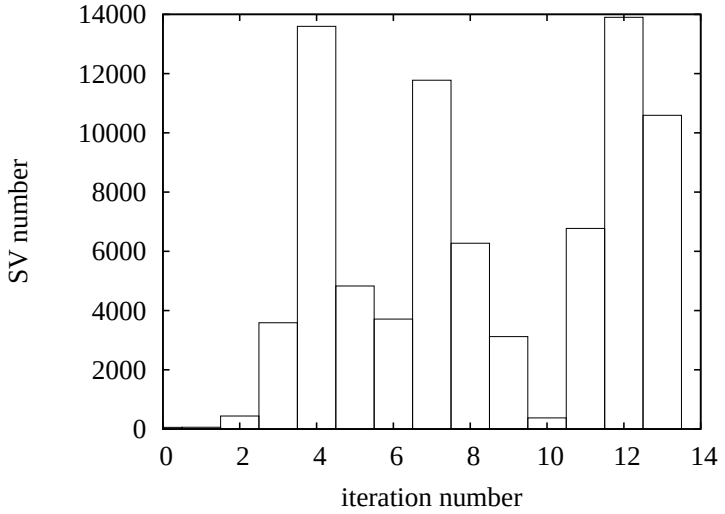


Figure 13.3: Subway optimization — Number of support vectors

13.3.3 Discussion

What happened during optimization?

Figure 13.3 presents the number of support vectors needed to build the regression of the value function. As the policy becomes more complex, its value function also changes and has more variations. Thus, the number of support vector increases and evaluating the SVR takes more time. Even though this rule of thumb is a quick description of what really happens during support vector regression, it seems to hold for some of the value functions used.

Having more complex SVR as value functions results in longer trial times: the more support vectors, the longer the SVR evaluation in a given state, and thus the longer the simulation time.

As the number of iteration grows, the value function presents more variations, which implies that building the regression becomes more and more complex. This complexity is balanced by the local regularity of the samples' values. This is illustrated by the beginning of the curve in figure 13.2. The sudden drop in calculation time as well as the number of support vectors in the regression presented on the two corresponding figures will be explained a little further.

Figure 13.1 is maybe the most important figure in the results. It presents the value of the initial state (time zero, stations empty, trains in the garage) given the policy, as the number of iterations grow. The solid curve is the simple average of the values obtained by simulation, the dotted curve is the result given by the SVR regression when asked for the value of the initial state.

First, it is interesting to note that — as expected from the previous rough analysis — the initial policy of letting all four trains run all day long results in a somewhat economical disaster: the subway manager looses approximately 3000 currency units per day.

However, the first iterations show an interesting increase in policy value. We need to

remember that the values plotted on figure 13.1 are the average of the cumulative sum of rewards really obtained during simulation. Therefore, this means that, for example, the strategy used by ATPI during iteration 4 provided an expected average reward of -880. In other words, simulating policy π_4, N_{sim} times, resulted in an average value of -880.

Consequently, on the previous example, when looking at the first iterations, one could consider that the monotonous evolution of the policy's value function during Policy Iteration is preserved and thus that ATPI does not suffer too much from the approximation due to the SVR in its approximate Policy Iteration scheme.

The small decrease in policy value at iteration 5 and the huge one at iteration 9 constitute a problem though: we will analyze this feature a little further.

A good policy after iteration eight

At iteration 7, the subway management strategy reaches an economical balance and at iteration 8, it seems that the subway becomes profitable, even with the price constraints given in table 13.3. It is encouraging concerning the ability of ATPI to find a good policy in a relatively short number of iterations.

However, this also depends a lot on the initial policy and on the states explored by this initial policy. The first simulations guided the exploration towards parts of the state space which allowed policy search to find better execution paths. With another policy initialization, improvements can be much slower and ATPI might not find a good policy at all.

This pathology is actually a special case of what causes the sudden drop in value at iteration 9. We seemed to have found a good π_8 and suddenly, π_9 seems to perform terribly and provides an average reward in the initial state of -1900. The bias and approximation error of SVR cannot be held responsible for such a policy decrease: approximate Policy Iteration does not converge but usually oscillates around policies which are not too far from optimality.

In the next paragraphs, we first discuss the choice of SVR for value function regression and then focus specifically on the value function decrease of iteration 9.

Why using SVR maybe was not the best idea

Our initial reason for choosing SVR as our value function approximator was based on two pragmatic statements:

- we need a regressor which can handle large dimension continuous sample spaces,
- support vector theory provides a well-understood framework and efficient implementations of classifiers and regressors.

The initial results of ATPI seem to comfort us in this conclusion. However, there are arguments which go against the use of ϵ-insensitive SVR and SVR in general.

First of all, ϵ-insensitive SVR constitute a biased estimator. We want to obtain a regression computing the average of the $R^\pi(s)$ variables, given the noisy input of pairs of state samples and values. ϵ-insensitive SVR tries to fit all samples inside the insensitivity tube, thus biasing the estimator: the value output is closer to a median than to an average. This

feature gives SVR its robustness but introduces a bias in the regressor's output.

Parsimony, *ie.* the low number of support vectors in the final regression, actually comes from the value of ϵ and the loss function. One can see on figure 13.3 that this number of support vectors sometimes becomes really important and that this parsimony is lost. Also, reducing the value of ϵ implies augmenting the number of *outliers* and thus the number of support vectors. Hence, the SVR formulation seems not to be appropriate for our problem.

Several options are possible from here. If we wish to keep the ability of kernel methods to handle large dimensions, we need to reformulate the regression problem. The kernel ridge regression or the kernelized LASSO (Least Absolute Shrinkage and Selection Operator) formulation of [Tibshirani, 1996; Roth, 2004; Wang et al., 2007] makes use of the kernel trick but writes the regression problem as[1]:

$$\min_{w,b} \|w\|_1 + C \sum_i \left(y_i - w^T \phi(x_i) - b\right)^2$$

While the SVR formulation was:

$$\min_{w,b} \|w\|_2 + C \sum_i l\left(y_i - w^T \phi(x_i) - b\right)$$

In other words, the LASSO formulation uses a L_1 regularization term with an L_2 penalization term while the ϵ-insensitive SVR used an L_2 regularization penalization with an ϵ-insensitive L_1 loss function l. We refer the reader to the above references for details on the generalized LASSO method and the associated search for efficient kernels. The L_2 loss function avoids the previous bias which was cause by the l function. Additionally, generalized LASSO regression provide very sparse regressors. We simply conclude that the LASSO formulation is a much better expression of our need for a regressor, as long as samples are provided as a batch set of data.

Another option is to consider that each sample only affects the regressor locally. Thus, one could give up the idea of global parsimony to focus on the approaches of *Local Learning* (as in [Atkeson et al., 1997]). The recent Locally Weighted Projection Regression algorithm of [Vijayakumar et al., 2005] represents the state-of-the-art of local learning for regression and brings together the results of locally weighted learning, Gaussian models and Partial Least Squares regression.

Lastly, we can consider the option of storing all the samples obtained without post-processing and relying on the ideas of Parzen windowing in order to build value functions: the value in a given state being the weighted average of all neighbors. The weights correspond to some kernel function.

Even though this last idea looks like brute-force processing and might be problematic for very large databases, we will see that it nevertheless constitutes an interesting and efficient basis for regression.

These two last approaches will be developed along with the improved ATPI algorithm in the next chapter.

[1] with the notation conventions of appendix B

We chose SVR in the first place because of their ability to handle large dimension state spaces. However, the bias in the SVR regressor makes it unappropriate for value function regression. Moreover, if one wishes to remain in the kernel methods family of regressors, the k-LASSO method seems more parsimonious and accurate. Finally, and because of the local incremental impact of our samples in the value function, we will explore the options of Locally Weighted Learning in the next chapter and will also evaluate an efficient of raw storage of samples.

This formulation of ATPI is incomplete

Finally, we need to provide an explanation for the small decrease in value function at iteration 5 and, more importantly, for the drastic loss at iteration 9. This unwanted behavior of ATPI comes from the problem of exploration for policy evaluation: in regions which have not been explored by the last iteration's trials, using the regressor is a very risky operation.

Indeed, in these regions, no samples have been given to the regressor and thus the estimation can be completely erroneous. We can graphically illustrate this behavior on figure 13.4.

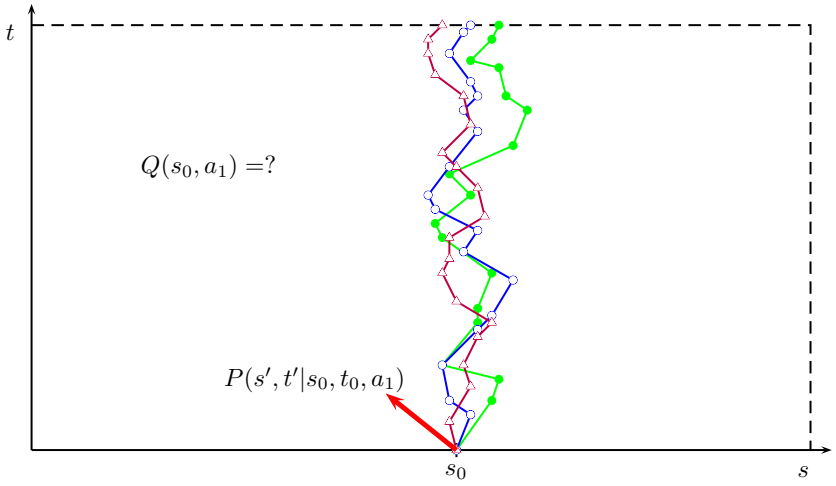


Figure 13.4: The exploration for evaluation pathology

In figure 13.4, we consider a 2-dimensional augmented state space with one variable in s plus the observable time variable. This way, the state space can be enclosed in the rectangle area. Trajectories resulting from the trials are sets of samples going from the "bottom" line at $t = 0$ to the "top" line at $t = T$. Three trajectories have been plotted in order to represent the trials at iteration n. One first important thing to notice is that these trials cross only a very small part of the state space and thus only evaluate the policy in these states. Thus, the implicit policy π_n, which is only instantiated online in visited states as explained earlier, is also only evaluated in the same visited states.

When one enters iteration $n+1$ and tries to compute the greedy policy in state (s_0, t_0), he has to do with respect to the V^{π_n} value function. This value function is the one computed from the previous iteration's regression. Suppose now we try some action a_1, in (s_0, t_0), which is different of $\pi_n(s_0, t_0)$. This action might take us to a state rather far from the

states explored in the last trial. This raises the question: is computing $Q(a_1, s_0, t_0)$ using the regression from the last iteration still relevant?

Indeed it is not. As long as the policy explores a lot of states or as long as the rewards obtained through exploration yield low cumulative rewards, the greedy policy is "tempted" to explore because the bad cumulative rewards can be directly compensated by immediate positive rewards. Thus, as long as the SVR underestimates the value of the last run's policy in the unexplored regions, the exploration might be drawn to them and eventually yield interesting trajectories.

However, when good trajectories have been found, the average cumulative reward is high. The next step's exploration then uses a value function that might overestimate the policy's value in non-explored areas. For states far from the last iteration's trials, the regressor approximately outputs the average of rewards. In this case, it can draw the exploration phase out of the good regions found at the previous iteration and towards overestimated regions. When these overestimated regions reveal themselves as not providing any large reward, one obtains the behavior witnessed between iteration 8 and 9 in figure 13.4. The V^{π_8} value function overestimates the expected gain in unexplored regions, attracts the exploration for π_9 out of the trajectories visited for π_8, the new trajectories yield lower rewards than expected and the value function drops drastically. More generally, the exploration is drawn out of the reward-providing regions towards worse regions and the cumulative reward of trials decreases from one iteration to the other.

Actually, this problem works both ways: it is particularly visible when the value of the initial state decreases, but the problem is the same for underestimating the value function in unexplored regions. When a regressor underestimates the true value of a policy, it can lead the exploration to miss some good regions early in the iterations while a good policy might have been found if the regressor did not underestimate the expected gain.

As stated in the last paragraphs, this problem seems to be a dead-end: we are looking for a representative value function but we are not ready to perform sampling everywhere. We propose to solve this dilemma by introducing a notion of *confidence* in the value function. This is the basis idea of the improved ATPI algorithm presented in the next chapter.

To conclude on the incompleteness of this first ATPI version:

> This first version of ATPI — which we call *naive* ATPI because it naively believes that samples can yield information about far unexplored areas — suffers from the problem of evaluating the confidence we have in the regressed value function.

13.4 Conclusion

In this chapter, we presented the ATPI algorithm. This algorithm relies on the following distinct features:

1. It performs forward, simulation-based search, by using greedy policy simulation to guide its exploration of the state space. Such an exploration behavior, related to policy iteration, can be seen as an "improve the policy only in the situations we are likely to encounter" optimization strategy. Hence, ATPI is an RTPI algorithm.

2. ATPI considers the state space to be a metric space and generalizes discrete and boolean variables to continuous ones. Consequently, it defines its policies and value functions over this extended continuous state space. Used in conjunction with simulation-based exploration and policy iteration, this features leads to evaluating the value function and policy only in "natural" values of the variables even though they are considered continuous. Similarly, it results in sampling trajectories only in these "natural" values.

3. Moreover, since time is included in the set of state variables, the specific oriented structure of temporal problems allows the forward search optimization scheme to store only execution paths without remembering the optimized policies. Namely, knowing explicitly policy π_n is not necessary to perform iteration $n + 1$.

4. In order to overcome the problem of the infinite continuous state space, we consider a generalization scheme based on support vector regression in order to infer the global value function from the trajectories obtained during the trials.

In the end, ATPI can be related to an LAO* algorithm (see [Hansen and Zilberstein, 2001]) applied to temporal, continuous state space problems. However, as shown in the previous chapter, ATPI is more closely related to RTPI since it is a policy-centered method which does not need a heuristic value function in the first place.

The first results on ATPI showed both encouraging results and important weaknesses in the algorithm. ATPI, when initialized with a policy that leads to exploration, found a good policy for the subway problem after only 8 iterations.

However, we face a drawback of partial exploration coupled with generalization. This drawback deals with the definition of a notion of confidence — similar to the notion of statistical relevance — and the naive version of ATPI cannot overcome the problem of over or underestimating the value function and knowing when to trust it or not.

The following chapter defines a more general framework by trying to extract the core components of a discrete events, controllable, temporal system and introduces the *improved ATPI* algorithm in order to build policies for such systems.

14

The improved ATPI algorithm

This chapter presents the result of our work on controlling explicit-event temporal problems. It draws from the results of all the previous chapters in part III and introduces an abstract description of the systems we want to control. This description is based on the discrete events systems paradigm and captures the class of problems which we call Discrete Events Controllable Temporal Systems (DECTS). Based on this abstract formulation and on ATPI's previous conclusions, we introduce the *improved ATPI* algorithm, which corrects the confidence problem of ATPI. This algorithm synthesizes all the previous contributions into a single framework. We present several examples of *improved ATPI* implementations and discuss the first results obtained on the subway problem.

14.1 Defining discrete events, controllable, temporal systems

14.1.1 Core properties of DECTS

The subway or airport problems cannot be modeled directly as Markov Decision Problems. This is due to the presence of internal, non-observable dynamics. The *natural* process resulting from these hidden dynamics is not an MDP anymore and needs specific modeling attention, for example as a GSMDP. Nevertheless, the previous chapter intended to design the ATPI algorithm to control such non-Markovian decision problems, by underlining which hypothesis needed to be made along the reasoning.

In this first section, we try to capture the essential properties of the systems we wish to control using ATPI-like algorithms. We call such problems *Discrete Events, Controllable, Temporal Systems* or DECTS in short.

Our introduction of a DECTS follows the idea of DEVS discrete events models in the sense that a discrete event system is given as a black box, with internal and external dynamics (the equivalent of the DEVS δ_{int} and δ_{ext}), temporal behavior (ta), outputs (λ), etc. In the Reinforcement Learning vocabulary, such a system would be associated with a *generative model*, *ie.* a black box which outputs a state value and a reward, given an action as input.

It is however important to make a distinction between what we could call Markovian generative models and non-Markovian ones. Markovian generative models take a pair (s, a) as input and return a pair (s', r) as output, they are simulators which do not depend at all

on hidden internal variables. If there are any internal variables in such a model, they are fully observable and thus participate in the s input mentioned above.

$$\text{Markovian model's transition function: } T(s,a) = (s',r)$$

On the contrary, a non-Markovian generative model takes an action a as input and outputs a pair (s',r) which does not only depend on a. Such models can have a hidden, non-Markovian internal state which influences the output. Simulators in general belong to this class of generative models. The engineer designing the model designs the internal dynamics but these dynamics and internal variables are not necessarily observable to the decision maker and the resulting behavior does not retain Markov's property because the observable variables do not uniquely define an internal state.

$$\text{non-Markovian model's transition function: } T(a) \equiv T_{internal}(a, s_{internal}) = (s',r)$$

Hence, with this kind of models, applying action a in observable state s does not always result in the same distribution over (s',r). The natural state process of GSMDPs is an example of a non-Markovian generative model. Controlling such models implies either making hypothesis about the importance of hidden variables or having some knowledge about the link between observations and internal state. In the latter case, these problems can be addressed — at the price of a higher complexity — as Partially Observable MDPs (POMDPS, [Kaelbling et al., 1998]).

So the first feature of a DECTS is to be a discrete events system, with a controllable aspect of inputting decisions and observing results and rewards, provided as a generative model which does not necessarily retain Markov's property.

Hence, a DECTS defines a transition function or *step function* providing the output observation and reward as a function of the input decision and the internal state. From the GSMDP point of view, this *step* function triggers the next GSMDP event and moves on to the next state. From a purely DEVS point of view, this *step* function is the δ_{ext} function when the input is received on an input port corresponding to actions.

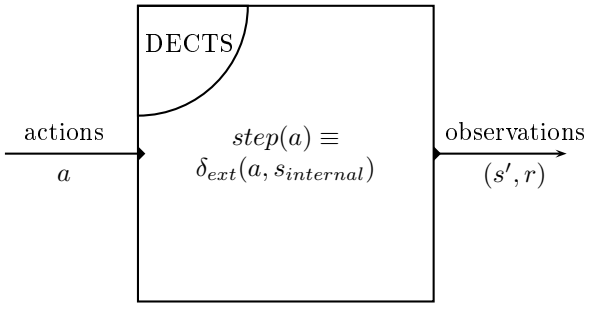


Figure 14.1: Schematic representation of a DECTS as a DEVS model

The internal dynamics remain a discrete event system, specified in any discrete event formalism. The internally triggered transitions of the DECTS depend on the internal state, itself being affected by the history of action inputs. This feature makes DECTS a *controllable*

discrete events system.

This notion of Controllability is both to relate and to distinguish from the general notion of Controllability in *Control Systems Theory*. A DECTS is said to be controllable in the sense that an external decider can act upon the evolution of the DECTS's process through discrete actions and thus try to control the future state of the system. However, general controllability for a control system starting in s_0 and aiming at s_f refers to the existence of a sequence of inputs which leads from s_0 to s_f. Such a notion is not guaranteed in the case of DECTS [1].

Finally, a DECTS is a *temporal* system in two separate senses:

- Each internal, discrete event transition has a temporal extension, characterizing the way the discrete event system evolves in time. Similarly to DEVS models, time is the common synchronizing feature between independent models. Hence, DECTS are event-driven temporal models.
- The internal transition function of a DECTS can depend on an explicit time. This simple feature, which particularizes time among other variables, focuses the class of physical problems we can address with DECTS to sequential decision tasks with temporal extensions. It is important to note that these tasks can present non-stationarity features or not, as well as observable time or not (in which case we could talk about *time-dependent* DECTS).

A DECTS is a *discrete events* model, evolving through discrete steps. It is also a *controllable* model, defining possible action inputs which condition the internal dynamics. It *needs not be a Markovian model* and can have a hidden internal state. Finally, it is a *temporal* system, where transitions have temporal extensions and where the internal dynamics can depend on an explicit time variable.

We define a particular class of DECTS which we call *reproducible* DECTS. A standard DECTS, from an object representation point of view, needs only define its initialization and *step* functions. From the DEVS point of view, such a model simply defines the δ_{ext}, δ_{int}, δ_{con}, ta, λ, etc. functions with a specific emphasis on the observable time variable if it is present and on the action input ports. However, for *reproducible* DECTS, we introduce an additional hypothesis. We suppose the DECTS defines a *clone* function. This clone function creates a copy of the system, including its internal state, still without letting the decision-maker access this internal state.

Reproducible DECTS are a specific compromise between Markovian and non-Markovian generative models. They never allow the deciding agent to access the internal state of the process, but define a weak notion of reproducibility, allowing the user to clone the current process in order to make copies of it as a black box.

By putting a name on such models and making this explicit distinction, we broaden the class of systems we will deal with and propose an alternative to the MDP / POMDP / non-Markov processes usual categories. *Reproducible* DECTS are particularly interesting in the case of simulation based-approaches since they allow to reproduce experiments with a

[1] However, one could remark that such a notion becomes a lot weaker in the case of stochastic systems and is related to the existence of *proper* policies.

guarantee on the initial state, even though this state might not be observable.

In particular, the GSMDP description we gave in the previous chapter is a particular case of *reproducible* DECTS.

> *Reproducible* DECTS are a specific class of DECTS with non-observable internal state which define a weaker notion than observability. Namely, these models make the hypothesis that the user has the possibility to *clone* a model in its current internal state (without necessarily observing this state).

From now on, our goal will be to construct efficient controllers for *reproducible* DECTS decision models.

14.1.2 Controlling DECTS and modeling a learner

Now that we have extracted the essential properties of the systems we want to control, defined the class of DECTS which extends controllable Markovian problems and distinguished the specific case of *reproducible* DECTS models, we wish to define the core properties of a learning algorithm for DECTS. Our idea is to extend the context of actor / critic architectures to the case of simulation-based approaches.

In classical actor / critic approaches, the operator (the actor) interacts with a main experiment, which can be repeated if needed. The actor controls the experiment while the critic evaluates the actor's behavior. Depending on the problem's hypothesis, the actor has the ability to reset the process to a given state or not. The results of interacting with this experiment provide the reinforcement signal for the learner. However, in simulation-based approaches, the agent can generate side experiments dedicated to obtaining information about the system (Q-values for instance), these experiments consist in temporary simulations which do not affect the global experiment. The idea of simulation-based approaches is that both the global experiment and the side ones have the same nature but are independent: they all are simulations which do not interact. Consequently, it must be possible to represent them in the same modeling framework.

Hence the goal of this section is to represent the interaction of the learner's black box with the simulation DECTS model inside a common modeling framework.

We define a DECTS learner as a black box with inputs regarding experience from interaction with the DECTS and outputs which contain optimization instructions. More specifically, the learner receives experimental results from any started experiment: either the main simulation of a trial or an evaluation simulation. As for outputs, the learner generates control commands directed to a given simulation or instructions to start a new experiment.

From a DEVS point of view, this corresponds to creating *recursive simulations* through the use of *dynamic models*. Dynamic models are coupled models where an *executive model* creates and links other DEVS models on the fly: this model can create, delete and link new models in the DEVS coupled graph. The DS-DEVS extension of [Barros, 1997] defines such dynamic structure systems. However, the dynamically created DS-DEVS models share a common simulation time, while our experiments are "virtual" with respect to the trial time

and to the optimization time. Hence, we need to be able to define *recursive simulations*. Recursive simulations, as presented in [Gilmer Jr. and Sullivan, 2005], consist in starting these virtual simulations as virtual experiments embedded in a main course of action (see the above reference for a precise definition of the term "course of action"). Even though recursive simulations have received little attention in the DEVS community so far, they can be easily modeled as dynamically created models where the internal state is the "child" simulation itself. These models have a *ta* function always returning zero (they do not change the global simulation time) and output the simulation results before being suppressed from the graph of models.

Figure 14.2 illustrates the representation of a DECTS learner as a DEVS model using dynamic recursive simulations creation.

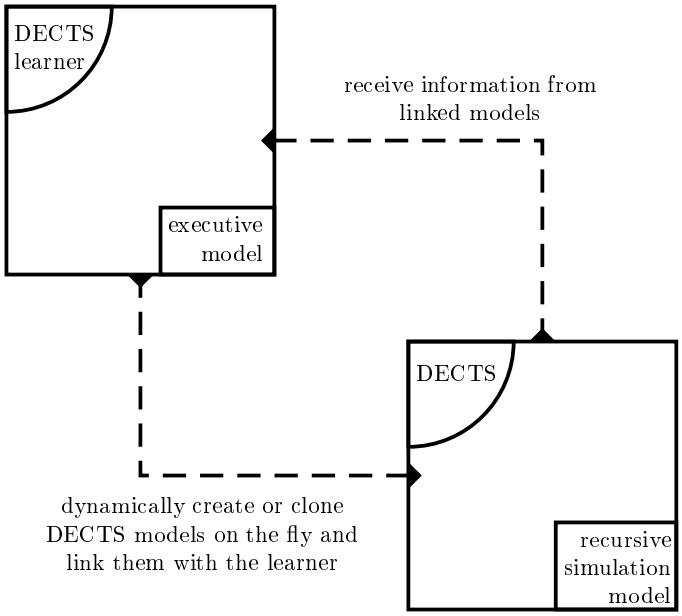


Figure 14.2: Modeling a DECTS learner inside the discrete events framework

Inside the learner's model, different *decision objects* constitute the internal learner's state. These objects can be a value function, a policy, a set of decision rules or any internal variable needed by the system. These *decision objects* form the internal state of the executive model for the learner. Combined with the learner's dynamics (expressed as the separate steps of the learning algorithm), they provide the complete δ_{int} and δ_{ext} functions for the learner's model.

In the case of the previous naive ATPI algorithm, these objects were the value function and the set of samples from the last iteration.

Since the learner implements the learning algorithm and since this algorithm is a sequence of instructions, the learner itself can be written as a discrete events system[2]. We provide the example of the naive ATPI learner in figure 14.3.

[2]Similarly to any instance of a Turing machine.

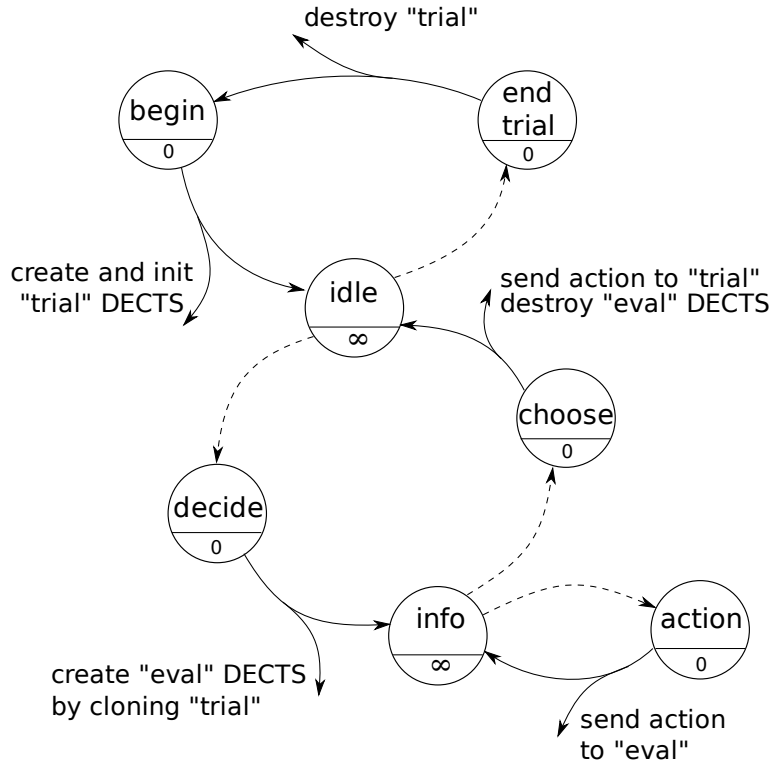


Figure 14.3: The DECTS learner of *naive* ATPI

Figure 14.3 follows the intuitive representation of DEVS models. Circles represent abstract internal states with their name and the associated *ta* function's value, solid arrows represent internal transitions and dashed arrows represent external transitions. Since the λ output function is only called for internal transitions, the forked solid arrows represent the output value.

The learner begins in state "begin", with an initial policy and the knowledge of the DECTS's initial state. Since the learner is an executive model, it has the ability to control the graph of coupled models and can create new models and link them together. This is what the first transition to "idle" does. It creates the "trial" DECTS which is the main experiment with which the interaction will take place. This DECTS is a GSMDP. Whenever this DECTS has been initialized, it sends a first request to the learner for an action since, in GSMDPs, decisions are possible at every state transition.

The learner then enters the forward search loop. The request for an action triggers an external transition taking the learner to state "decide". For this purpose, the algorithm instantly creates all the "eval" simulations by cloning the "trial" DECTS in its current state. Upon action requests, the learner sends the possible actions to the "eval" models and retrieves the resulting state and rewards in order to evaluate the Q-values of these actions. When all "eval" models have returned their next state and reward, the external transition to "choose" is triggered. In this state, the learner computes the optimal action by using its internal SVR and the results from the "eval" models. Finally, it deletes all "eval" models and sends the computed best action to "trial".

This process goes on until the horizon is reached and "trial" sends an interrupt event.

Then, the learner makes the transition from "idle" to "end trial". In this state, it computes the $V^{\pi_{n+1}}$ value function and returns to "begin" by deleting the "trial" model.

A DECTS learner is a *discrete event model* describing the optimization procedure. This model is an *executive model* in a DS-DEVS representation. It drives the global optimization process by *dynamically* creating DECTS as *recursive simulation models*. The internal state of the DECTS learner is composed of initial knowledge about the problem and of so-called *controller objects* which are the internal tools used to compute decisions.

14.1.3 Why DECTS?

The purpose of providing a DECTS formulation and the associated learner description is twofold. First of all, the idea is to present clearly the essential properties of the systems we wish to control and of the learning methods we apply to them. This presentation points out the specificities of DECTS by setting them in the general framework of discrete events processes.

Hence, DECTS do not constitute a new formalism, but rather an extension to DEVS models trying to lay a bridge between discrete events simulation specifications and sequential learning optimization processes.

In particular, one important characteristic of DECTS models — for both the controlled process and the learner — is that such a representation formalizes the optimization process in the same language as the controlled system itself. This provides a first attempt at representing an optimization process inside the framework of discrete event systems and opens the door to interfacing heterogeneous optimization and simulation models in the framework of discrete events modeling.

14.2 Revisiting the idea of ATPI

Having defined the family of systems we wish to control and pointed out their specification, properties and limits, we now turn back to the ATPI algorithm and try to overcome the flaws presented at the end of chapter 13. For this purpose, we investigate the question of *confidence* which was raised in section 13.3.3. We perform this investigation in the framework of reproducible DECTS by remarking that GSMDPs are indeed a specific class of reproducible DECTS.

14.2.1 The initial ATPI intuition: simulating to explore and evaluate

Let us restart from the basics of ATPI and work our way to the *improved ATPI* algorithm by highlighting where the mistakes were previously made and how we compensate for them.

First of all, applying ATPI to a temporal control problem implies having some initial information. The available information about the optimization problem is that:

- We know the initial state[3].

[3]Or a distribution over the possible initial states, but we won't discuss this case.

- Our problem has a very large state space.
- There exists a quantified notion of similarity between states, allowing to defined the *neighborhood* of a state with respect to a given *distance metric*.
- We have a generative model which we represent as a DECTS.

Thus, we want to exploit our ability to simulate in order to explore and evaluate. More specifically:

- Exploration: we take the option of letting the greedy policy simulation guide the selection of the set of states upon which we will perform optimization.
- Evaluation: we rely on Monte-Carlo sampling to retrieve expected gain information from the trajectory space.

Since our model is a reproducible DECTS with observable time, the presence of an explicit time variable guarantees two important properties:

1. Simulations have a finite number of steps.
2. There is a zero probability of an infinite loop because such a loop would imply remaining at the same time indefinitely which — we suppose — is physically impossible.

Finally, the approach of Monte-Carlo sampling is not useful in continuous state spaces if we are not able to generalize the obtained samples to their neighbor states. This is particularly important in continuous state spaces since in this case, there is a zero probability of visiting the same state twice[4].

14.2.2 The need for generalization

Because of this last argument, we need a generalization method in order to identify states which are similar to the ones previously visited and to infer these states' relevant information (value function for example) from the experience collected with their neighbors. Naive ATPI used generalization for the value function only, since it took the option of never explicitly storing the policy. But this is where this approach showed its main weakness: we need to define *where* in the state space we can trust this generalization.

14.2.3 The problem of confidence

Trusting the generalization — the regressor in the naive ATPI case — in order to use it implies building a notion of *confidence*. This confidence is a measure indicating whether the output of the regressor is reliable or not. Graphically, as illustrated previously on figure 13.4, this notion of confidence is linked with the density of samples collected to build the regressor and to the consistency of the information these samples provide.

[4]The same argument actually holds for discrete large state spaces too, since the main problem behind such a generalization is the identification of *similarities* between encountered situations.

More precisely, whenever we evaluate the regressor in s, we need to know if the previously collected experience is sufficient in order to make a prediction. In other words, we need to determine (or arbitrarily decide) whether the samples we have collected constitute a sufficient statistics of the statistical parameter $V^{\pi_n}(s)$. Even more specifically, we need to characterize a threshold on the *minimal* sufficient statistics for this variable.

In order to approximate a measure of the samples' statistical sufficiency, we choose to approach the density of these samples or — more formally — the probability density of the underlying process which drove us to pick these specific samples during the last trials.

In the end, from a pragmatic and technical point of view, we need to estimate a probability density function for the process underlying the Monte-Carlo sampling operation. The associated difficulty is that this probability distribution is defined over a high-dimensional metric space (the state space). For this purpose, we can use several tools from the literature. To list a few, we can mention One-Class SVM (OC-SVM, [Schölkopf et al., 2001]), Gaussian Processes (see [Chen et al., 2006] for example) or Parzen windowing (see [Parzen, 1962]).

Once this density estimation has been constructed, we can use it as a *confidence function* in order to decide when to trust the value function regression or not.

14.2.4 Using the confidence function to improve ATPI

Now that we have defined a confidence function attached to the last iteration's regression, we need to adapt the algorithm to use it. When we try to evaluate $Q^{\pi_n}(s, a)$ values, the straightforward use of this confidence function is to do the following. As in the naive ATPI case, we still sample the next state using a clone DECTS of the current "trial" DECTS. If the sampled state corresponds to a state for which the confidence function returns true, then we consider we can safely use the regressor.

However, if the confidence function returns false, it means we haven't had enough information around the sampled state to make any prediction as to the value of policy π_n in this state. In this case, we need to acquire the needed information and hence, we need to simulate π_n from this state until we reach a state in which we trust the regressor or until we reach the horizon.

But if we want to simulate, we need a control policy for the DECTS. This means we have stored the policy from the previous iteration. Although naive ATPI avoided explicit instantiation and storage of the policy, this seems now necessary to compensate for the confidence problem. In the end, it means our DECTS learner has at least three different internal objects: a value function describing the last policy's value, the associated confidence function and the last up-to-date policy.

14.2.5 Storing policies for ATPI

Computing and storing full policies is a heavy handicap for MDP algorithms. Computing full policies seems a very unfavorable compromise since most of these policies' actions will never be used before being replaced by improved actions. Moreover, storing the policy in a memory-saving fashion might become quickly problematic. Consequently, we wish to keep the online policy instantiation feature of ATPI while introducing a way to store previous

improvements to the policy.

ATPI is initialized with a policy covering the whole state space. This policy does not have to be an explicit, fully-instantiated representation of the $s \mapsto a$ function but it needs to provide a way of computing the action anywhere in the state space. It can consist in hand-made decision rules, heuristic functions, etc. The same way RTDP is initialized with a heuristic value function, ATPI starts with an initial policy which can be described in a very synthetic manner.

Along the iterations, this initial policy is "patched". Namely, it is locally replaced by the optimized actions in the states which have been visited by the trials. Consequently, each iteration of ATPI adds a new patch upon the last policy. In other words, we define incremental, partial policies which come correcting the last global policy.

Since we take this option of "patching" the policy, we are confronted with two problems. The first problem deals with storing a pile of partial policies along the iterations. This problem is purely practical: it is a matter of compactly storing each partial policy and storing a pile of these compact representations. It might become problematic if the number of patches increases a lot, but we expect this number to remain low because it corresponds to the policy iteration's iteration number before convergence.

Along the iterations, patches are piled upon the initial policy. Whenever the policy is asked for an action, the stack of patches is unpiled and the first applicable patch serves to compute the action returned.

This brings the second problem which is the same as the value function confidence problem: since the optimized actions are computed online in the sample states visited by the trials, we need to generalize these actions to the neighbor states and thus we need to classify which states map to which action.

Hence, the problem of storing policies actually hides a problem of classifying actions over states and generalizing local sampled information to local continuous values. But it also deals with evaluating the "support states" of each patch, which corresponds in the end to defining a confidence function for each patch, similarly to the confidence function for the value function.

14.2.6 A full statistical learning problem

In the end, the question of generalization in our approach leaves us with a full statistical learning problem:

We need to infer a value function from a set of samples and to build a regressor as well as a density estimator from these samples. Moreover, since we decided to simulate π_n in the states for which we are not confident, we will receive new samples for V^{π_n} during iteration $n+1$ and thus we need to incrementally integrate these samples into the generalized value function.

Hence, estimating policy π's value function is an *online incremental regression* problem.

Similarly, the value function's confidence function needs to be updated at the same time as the value function is updated.

> Estimating the confidence function for V^{π_n} is an *online incremental density estimation* problem.

Then, we need to generalize and compactly store the action experience from iteration n in a "patch" for the global policy obtained so far. Since we will never need to simulate the policy in a state where the value function's confidence function returns true, the classification scheme for the policy needs not be online. At the end of each iteration, we need to build a classifier indicating the latest improved actions, corresponding to the last set of trials.

> Constructing the generalization for the policy is an *offline* (between iterations) *batch classification* problem.

But to efficiently patch the global policy with the latest classifier, we need to define where this classifier applies, and thus we have the same kind of confidence estimation function to build for the policy than we did for the value function.

> Our incremental partial policy construction via the "patching" method implies an *offline probability density estimation* problem.

Finally, we are left with a complete statistical learning problem which comes directly from the very nature of the process we want to control: large state spaces and continuous variables.

14.3 The *improved ATPI* algorithm

14.3.1 Algorithm overview

The improved online-ATPI (iATPI) algorithm relies on the construction of a regressor, a classifier and the corresponding confidence functions. We will write V_n the regressor for the samples obtained at iteration n, π_n the corresponding classifier and C_{V_n} and C_{π_n} the associated confidence functions. Accordingly, π_0 designates the initial (eventually implicit) policy.

The improved ATPI algorithm is presented in algorithm 14.1[5].

For clarity, in this algorithm, we separate the training sets for the value function and for the policy into two distinct databases: $actionDB$ holds pairs of states and actions while $valueDB$ contains pairs of states and samples of the value function. These two databases are built using the successive *execution paths* issued from the different simulations and trials. Execution paths are noted σ.

The `trainRegressor` and `trainClassifier` procedures actually also build the associated confidence functions. The `convertExecutionPathToValueFunction` procedure performs the

[5]This presentation differs slightly in the notations from our initial introduction of iATPI in [Rachelson et al., 2008c].

backward cumulative sum of rewards in a given execution path in order to build a set of value function samples. The other procedures have effects corresponding to their names.

As explained in the previous sections, iATPI works the same way as ATPI, by performing forward search in the state space, guided by greedy policy simulation and using local generalization. The confidence problem is addressed through the introduction of the C_{V_n} and C_{π_n} functions which indicate when it is absolutely necessary to gather more simulation samples in order to choose between actions. In the end, this implies generating more simulations but it also limits the computational impact of these simulations by stopping them as soon as a new confidence region is entered. This is the result of the "while" condition in the `simulateWithStop` procedure: the simulations are stopped as soon as a state for which $C_{V_n}(s) =$ `true` is encountered.

It is interesting to note that the values of N_{sim} and N_a can be set by hand but can also be automatically tuned by using a statistical likelihood test on a set of samples for the value function and the Q-values. We will present this option a little further.

14.3.2 Writing the algorithm in the framework of DECTS

Figure 14.4 shows the improved ATPI algorithm as a DECTS controller. This learner's internal objects are the regressor, classifier and confidence function defined above. Graphically, the model presented in figure 14.4 is the same as the one for naive ATPI. The main difference lies in the bottom right transitions which actually define the soundness of $Q(s, a)$'s evaluation: these transitions are triggered differently with iATPI than with ATPI. With naive ATPI, the transition from "info" to "action" and back was triggered only once per "eval" model, regardless of the final state of the "eval" model. On the contrary, with *iATPI*, the "eval" model continues its simulation — and thus its action requests — as long as it does not reach a confidence state. Hence, this "info" – "action" loop can be triggered more than once per "eval" model with *iATPI*.

More specifically, during an *iATPI* run, when in state "info", the learner waits for action queries from the "eval" models. Any such query takes the learner to state "action", where it computes the action to send to the "eval" model which requested it, based on the current model's observation. Then the learner instantly returns to state "info", waiting for other requests. The first action sent to an "eval" model is the a test action for $Q(s, a)$. Then actions are sent using the latest policy π_n. This process continues as long as the state of the "eval" models correspond to states of unconfidence. As soon as a state for which $C_{V_n}(s) =$ `true` is entered, the transition from "info" does not go to "action" but to "choose". This stops the evaluation simulations in states for which we are confident and thus corrects the problem raised by naive ATPI.

Then the process goes on as for naive ATPI: the value function is updated and the Q-values are used to select the action sent to the "trial" DECTS process.

The interaction between the ATPI algorithm implemented as a DECTS learner and the controlled DECTS itself is illustrated in figure 14.5. This interaction follows the ideas of DS-DEVS dynamic models creation and linking, the recursive simulation design scheme and the discussion of section 14.1.2.

Algorithm 14.1: Improved online-ATPI: *iATPI*

main:
input: π_0, s_0
repeat
  $valueDB$.reset() /* new trial */
  $actionDB$.reset()
  for $i = 1$ to N_{sim} **do**
    σ.reset()
    $trialProcess$.init(s_0)
    while horizon not reached **do**
      a =bestAction(s)
      $trialProcess$.activateEvent(a)
      $(s', r) \leftarrow trialProcess$.step()
      σ.add(s, a, r)
    $valueDB$.convertExecutionPathToValueFunction(σ)
    $actionDB$.add(σ)
  $V_n, C_{V_n} \leftarrow$ trainRegressor($valueDB$) /* decision objects */
  $\pi_n, C_{\pi_n} \leftarrow$ trainClassifier($actionDB$)
until termination

bestAction(s):
for $a \in A_s$ **do**
  $\tilde{Q}(s, a) = 0$
  for $j = 1$ to N_a **do**
    $\tilde{Q}(s, a) = \tilde{Q}(s, a)+$ simulateWithStop(s, a) /* test the actions */
  $\tilde{Q}(s, a) = \frac{\tilde{Q}(s,a)}{N_a}$
return $\arg\max_{a \in A} \tilde{Q}(s, a)$

simulateWithStop(s):
$\sigma_{eval} \leftarrow \emptyset$
$evalProcess = trialProcess$.clone()
$evalProcess$.activateEvent(a)
$(s', r) \leftarrow evalProcess$.step()
$Q \leftarrow r$
$s \leftarrow s'$
while horizon not reached and $C_{V_n}(s) =$ false **do** /* explore until confident */
  $a = \pi_n(s)$
  $evalProcess$.activateEvent(a)
  $(s', r) \leftarrow evalProcess$.step()
  $Q \leftarrow Q + r$
  $s \leftarrow s'$
  σ_{eval}.add(s, r)
$Q = Q + V_n(s)$
$valueDB$.convertExecutionPathToValueFunction(σ_{eval})
$V_n, C_{V_n} \leftarrow$ reTrainRegressor($valueDB$) /* update value function */
return Q

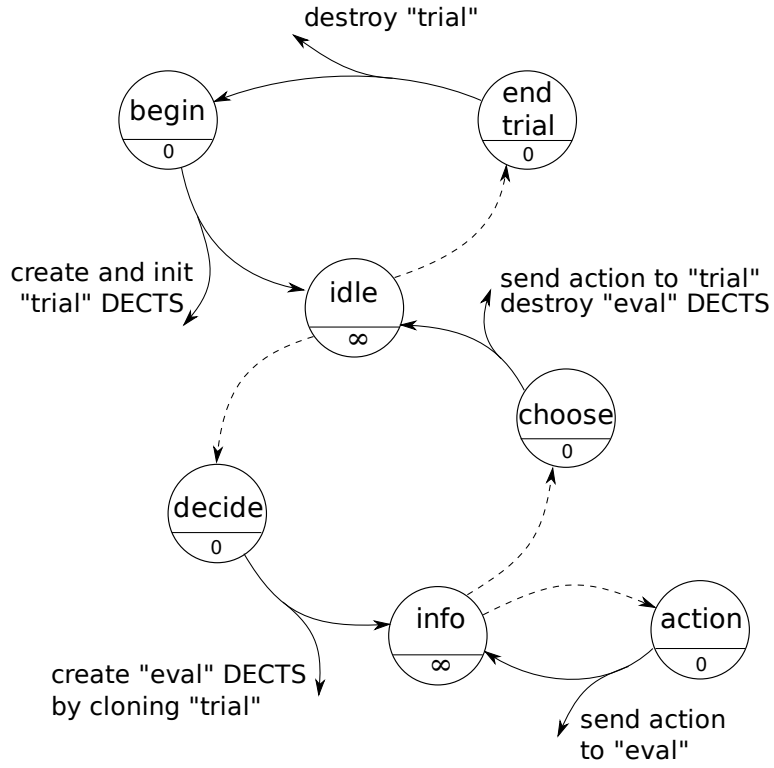


Figure 14.4: The DECTS learner of *improved ATPI*

14.4 First experience with *iATPI* in practice — difficulties and initial results

14.4.1 Statistical Learning tools

The general algorithm presented in the previous section can be implemented using various tools from Statistical Learning theory. The following paragraphs explain the different choices we have explored for improved ATPI.

Regression

For the regression part, the previous chapter explained why SVR were not the best choice. The qualities we expect from our regression method are the following:

1. It should be unbiased since we need to evaluate the average of the samples.
2. It should be able to perform incremental learning in order to allow the addition of new samples on the fly (during the `simulateWithStop` procedure).
3. It should be implemented in a compact, memory-saving fashion, in order to allow for easy evaluation in a given state and low memory storage cost.

Few regressors actually meet these three requirements. To our knowledge, the LWPR method of [Vijayakumar et al., 2005] corresponds to these needs. LWPR is actually a very interesting choice since it defines *receptive fields* which correspond to a good estimation of the value function's confidence function.

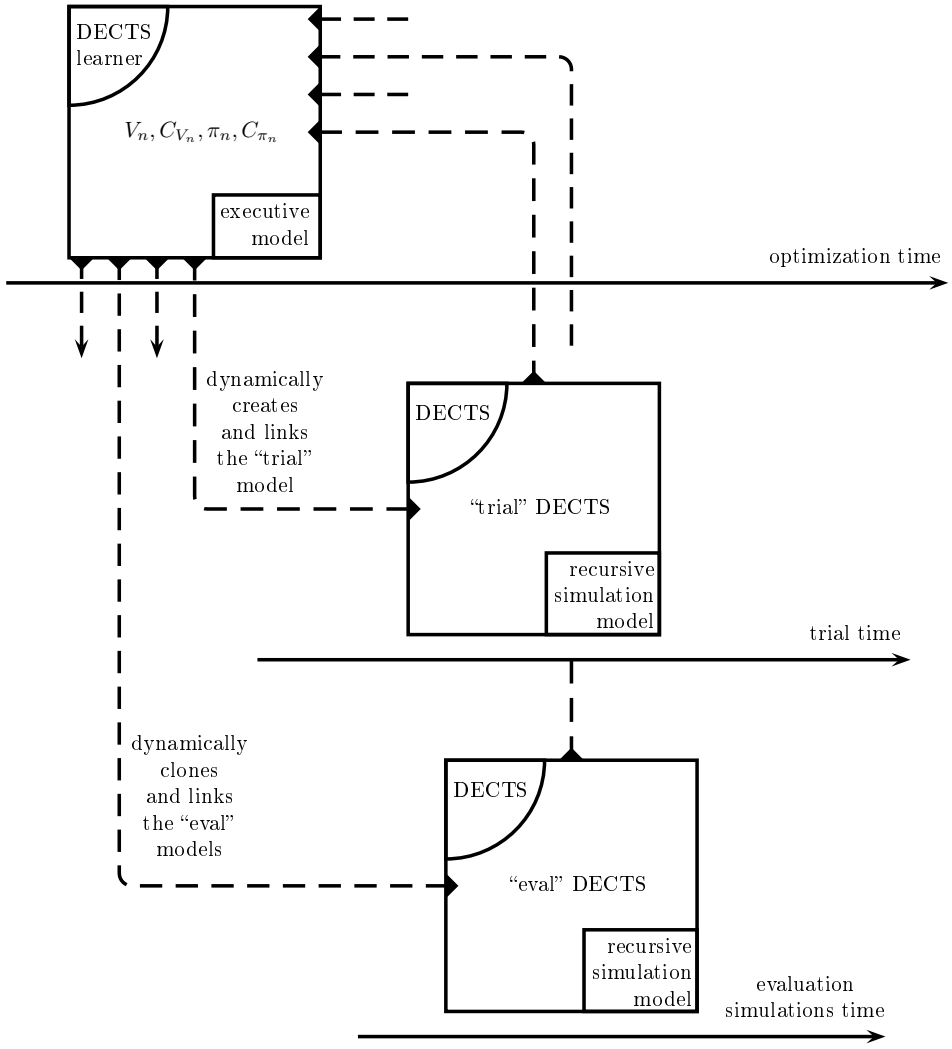


Figure 14.5: Illustrating the virtual different time references of the *iATPI learner*

However, our experience with LWPR showed that it needed to see a lot a points before converging to a good value function. The alternative to LWPR we have used for comparison consists in storing all samples in a relational database. This way, we used efficient look-up strategies in this database in order to find the neighbors of a given state and to perform local averaging via Parzen regression.

Density estimation

For the case of the full database storage, Parzen windowing provides a straightforward implementation to density estimation.

Similarly, for the LWPR case, we can use the maximum activation level of receptive field as a confidence measure for the regressor.

In the general case, the technique of One-Class SVM (OC-SVM, presented in [Schölkopf et al., 2001] for example) can provide good density estimation results but requires some fine tuning of the kernel's parameters.

We used these three techniques in order to compare the different versions of iATPI.

Classification

Support Vector Machines have proven themselves particularly efficient in classification problems where one needs non-linear separators between arbitrary regions in large dimension spaces. For this reason, we turned towards the tools of Multi-Class SVM (MC-SVM) which are a collection of standard two-class SVM classifiers performing a majority vote to discriminate between several different classes. A discussion on MC-SVM is provided in the LIBSVM documentation in [Chang and Lin, 2001].

Similarly to the previous cases, in the case of raw database storage, the policy can be determined by a majority vote among neighbor states. This actually provides the possibility of defining stochastic policies[6].

Building versions of improved ATPI without a regressor

It is interesting to note that one can build versions of improved ATPI in the absence of some of the previous tools. For example, if one wishes not to store the value function and its associated confidence function, then by default the confidence function for V returns false and the algorithm turns out to be very close to the flavor of simulation-based Policy Iteration presented in [Bertsekas and Tsitsiklis, 1996]. This approach implicitly throws away samples after seeing them and does not remember any information as to the value function. We call this "memory-less" version of iATPI "Monte-Carlo iATPI" since it fully relies on Monte-Carlo sampling in order to gather information about the value function, and completely forgets the samples after seeing them.

[6]However we did not explore this feature.

Similarly, one could use static regressors (unable to perform incremental online learning) and thus use static value and confidence functions, compensating the unability to incrementally enrich the regressor by using simulation more often. If one wishes to adapt the work done on the previous chapter's ATPI implementation, this might be the simplest way to do — even though it does not exploit the full potential of improved ATPI.

Summary of the different versions of improved ATPI

Table 14.1 summarizes some possible options for iATPI as the cross product of the options presented in the previous paragraph. These are the ones we have implemented and tried, many more are possible. In the fourth column, "RF" stands for the LWPR "receptive fields" which are Gaussian kernels defining the activation of local linear models in LWPR (see [Vijayakumar et al., 2005] for details). In some cases, we gave names to the different versions of iATPI, these names are indicated in table 14.1.

π_n + C_{π_n} \ V_n + C_{V_n}	None + $s \mapsto$ false	SVR + OC-SVM	LWPR + RF activ$^{\text{n}}$ level	Parzen regr. + window.
MC-SVM + OC-SVM	Monte-Carlo iATPI	"non-naive" iATPI	compact iATPI	—
local vote + Parzen w.	—	—	—	full storage iATPI

Table 14.1: iATPI versions

It is interesting to note that only the 4 rightmost versions of iATPI are fully incremental versions, taking advantage of the samples for V^{π_n} collected during iteration $n+1$.

Looking at statistical learning tools for reinforcement learning under the light of what [Anderson, 2000] illustrate — namely that policy-based methods might be more robust to approximation than value function ones, because they generally build both the robust object of a policy and the efficient estimation of a value function — provides an interesting hindsight for iATPI. Value estimation errors are not propagated through the iterations in the iATPI approach since only the samples issued from trials experience are used to build the value function estimation. Instead, approximation errors can lead to exploration difficulties but this pathology is present for both value and policy based algorithms. Hence, the conjunction of statistical learning tools with policy-centered methods seems to be a promising way of tackling problems which present large and/or continuous state spaces and for which approximation is often necessary.

14.4.2 Subsampling for *iATPI*

When running the previous versions of *iATPI* on the subway problem, we were confronted with a new difficulty. The numerous clocks lead our discrete event system to have very short time intervals between transitions and hence, very long trajectories. For instance, over a half-day, the subway problem has trajectories of about 4000 points. These 4000 points correspond to all intermediate states encountered by the process before reaching the temporal

horizon. The problem associated with this large number of points per trajectory is that we actually try to optimize the policy in each of these points. This might not be necessary because of the following simple reason.

Imagine the process is in a given state where the learner finds an improved action, say "remove train 3". This action is activated and the next step takes the process to a new state. Imagine now this last transition does not correspond to the removal of the train but to the arrival of a passenger at station 5. This has no impact on the previous decision we took and might need to recompute the same "remove train 3" action in this new state, even though it was already activated (but not triggered). Similarly, it might be cumbersome to compute better actions in all the sample states of a trajectory. The following paragraphs rephrase this idea and extend it.

As an optional feature, one can further exploit the presence of the observable time variable in the model. On top of having a notion of distance in the state space — allowing us to define similarity between states — we also know that transitions leap forward in time by increments corresponding to the clocks' differences.

Thus, if one knows that these clocks' difference values will be small compared to the problem's temporal horizon, it might be useful to consider that two consecutive states with similar times will have similar optimal actions. This is particularly enforced by the fact that most of our system's transitions are not due to the action events but to exogenous uncontrollable events.

Consequently, in order to improve optimization efficiency and speed, one could decide to only optimize actions once in a while (every 10 transitions for instance or when some specific states are encountered) instead of doing so at each state change.

We call this feature "optimization subsampling" since it subsamples the states to optimize among the sample states of an execution path.

Actually, incremental regression for the value function should leverage the need for subsampling. When in state s^n, finding the optimal action implies using the regressor and / or simulating new trajectories. The incremental regression process integrates the new samples used to compute the best action in s^n into the value function before moving to s^{n+1}. If state s^{n+1} is indeed close to s^n, then — because of these last samples — the confidence function should return true and finding the best action in s^{n+1} is only a matter of sampling the following states when testing actions. This illustrates why incremental regression should improve the performance of policy optimization. However, the process cloning operation, the next state sampling and the regressor's interrogation are still non-trivial operations, slowing the optimization process. Hence, we keep the subsampling optional feature anyway in order to facilitate policy search.

14.4.3 An example of implementation using LWPR and MC-SVM

Implementation overview

Before presenting this implementation, we provide a short reminder on LWPR and MC-SVM.

The Locally Weighted Projection Regression algorithm of [Vijayakumar et al., 2005] is

a local learning regression algorithm. The regression is computed as a combination of local linear models. The weight associated with each linear model is the level of activation of a Gaussian model. Each of the Gaussian models is called a *receptive field*. These receptive fields define a notion of locality and incrementally adapt their shape to new samples, in the flavor of Principal Component Analysis. The linear models are updated using Partial Least Squares incremental regression. This allows to project samples onto a lower dimensional space in order to compute the local linear regression. More specifically, when a new point is added, if it activates some receptive fields, it triggers their update (the update of the Gaussian model's parameters and of the linear model). Else, if it does not activate any receptive field, a new one is created, centered on this sample. This procedure makes LWPR a local regression method and a fully incremental method. We refer the reader to [Vijayakumar et al., 2005] for a more detailed and more rigorous presentation of LWPR's properties.

Multi-Class Support Vector Machines (MC-SVM) are an extension to standard Support Vector Classifiers (SVC). Instead of separating two distinct classes of object using non-linear classifiers, MC-SVM need to classify a certain number of different classes of objects (as actions, in our case). This is done by defining a number of SVC separating two classes at a time, then performing a vote between these SVC in order to decide to which class a point belongs. For a more detailed discussion on MC-SVM, see [Chang and Lin, 2001].

The "compact *iATPI*" version of *iATPI* presented in table 14.1 uses the tools that best fit our requirements for V_n, π_n, C_{V_n} and C_{π_n} representations:

- V_n is implemented using LWPR. This representation is the only one allowing for actual incremental regression, using the sample points and discarding them after. It is important to note that some kernel regression methods actually perform incremental regression. However, an essential difference with LWPR is that most of these methods actually keep track of *all* the sample points seen so far and incrementally extract a subset of relevant points. For example, most Support Vector algorithms performing incremental regression move points in and out of the support vectors set as new points are added to the training set. On the contrary, once LWPR has seen a point, it can be discarded without further storage. This yields a weakness and a strength of LWPR: first it needs a lot a points to converge to a relevant value function and secondly, it is a truly online incremental regression method.

- Another interest of using a local learning method as LWPR is that it automatically provides a measure of confidence and locality. We used the maximum activation level of LWPR's receptive fields as a measure of confidence for C_{V_n}. This measure describes how "close" one is to the nearest receptive field. The notion of distance being defined by the covariance matrix of each receptive field.

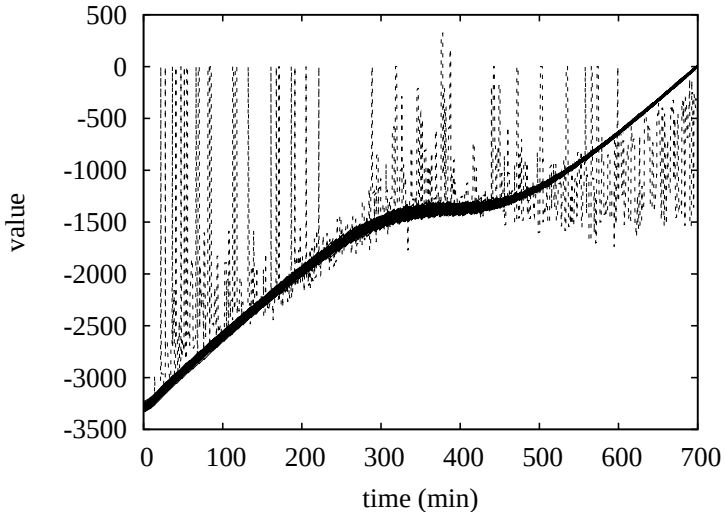
- π_n is implemented using MC-SVM. This representation actually proved itself to be quite reliable.

- Finally C_{π_n} was implemented using One-Class SVM (OC-SVM, [Schölkopf et al., 2001]) in order to estimate the probability density function of the distribution underlying the set of points used to train the classifier.

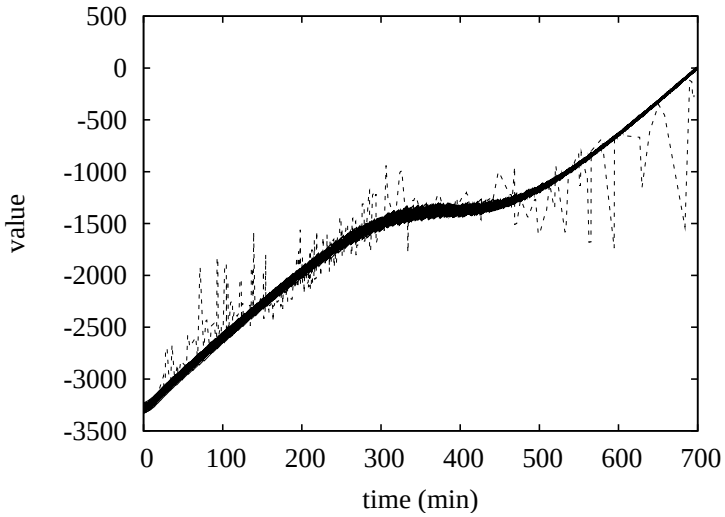
Filtering with the confidence function can be efficient

The "compact *iATPI*" version suffered from several drawbacks. The first drawback is illustrated on the two graphs of figure 14.6. These graphs were obtained by running 50 trajectories

with the initial policy π_0, incrementally inferring the value function with LWPR, and then running an additional trajectory. On this trajectory, instead of using the points to train LWPR, we simply observe the value prediction made by our regressed value function. The solid lines correspond to the 50 training trajectories and the dashed curve is the prediction curve along the trajectory.



(a) Raw evaluation



(b) Filtered evaluation

Figure 14.6: The interest of using confidence for regression

It is important to note that these trajectory were ran across the 21-dimensional state space of the subway problem. We plot their values against time in order to make them readable but nothing guarantees that these trajectories will actually be close in the state space.

Figure 14.6(a) presents the regression's estimated values in all encountered states during the fifty first run. The large noise which seems to regularly send the value function to zero illustrates the absence of receptive fields in the states encountered by this last simulation. This confirms the fact that — even though the value evolution with respect to time was

similar for the 50 previous simulations — the trajectories can be quite far from each other. Hence, this is another illustration of what caused the weakness of naive ATPI.

On the other hand, figure 14.6(b) shows only the points returned by the regressor with a confidence value above the confidence threshold. This operation filters the points which are too far from any receptive field and thus justify both the use of the value function in confidence points and the necessity of new simulations in non-confidence ones.

Nevertheless, an important caveat remains here. Even the filtered value function provides a noisy evaluation which is not fully satisfying. This is probably due to the fact that we did not provide enough points to LWPR for training. With this problem we reach one of the limits of such an implementation: even if simulating a GSMDP remains easier than explicitly calculating its transition and reward model as an MDP, this operation remains rather complex and takes time. Hence, we cannot fully consider that simulation is "free" and there is a compromise to make between the number of simulations and the accuracy of our regressors. In order to illustrate this compromise, it is interesting to note that fifty simulation correspond to providing approximately 200000 points to LWPR for training. But in dimension 21, this might still not be sufficient.

The compromise between simulation and regression

This is further illustrated by the behavior shown on figure 14.7.

Similarly to the previous figures, figure 14.7(a) represents the fifty trajectories obtained by simulating the policy obtained after four iterations of the algorithm. Behind the fifty solid lines corresponding to these trajectories we plotted the fifty-first evaluation trial as a dashed line. This evaluation trajectory is reproduced on figure 14.7(b). Finally, figure 14.7(c) shows the value function filtered by the confidence function.

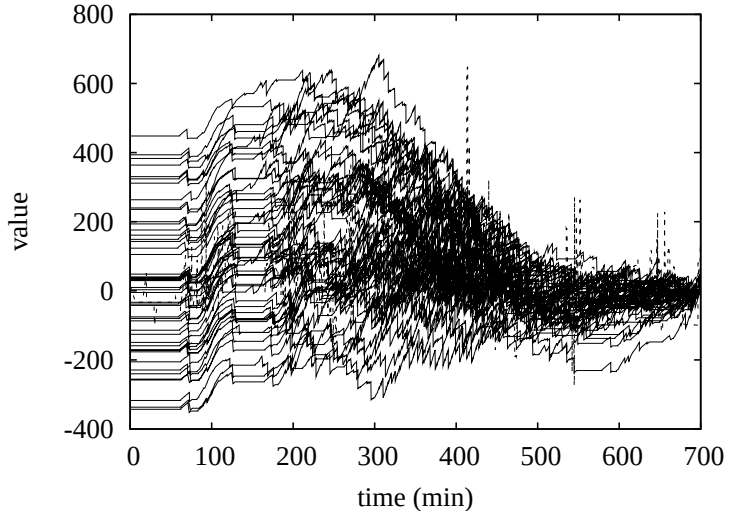
This time, the $R^\pi(s_0)$ random variable has a much larger standard deviation and fifty samples might mot be enough to obtain a reliable estimate. Moreover, training LWPR with noisy samples requires providing even more samples and the regression process might still be unreliable here.

It is reassuring to note that only few points come through the filtering process, as shown on figure 14.7(c). It means our confidence measure might be reliable. In particular, it is important to see that no confidence point was found after time 450 — probably because the trajectories diverge too much after this time. However, even though the value function estimation's of figure 14.7(c) strongly reduced the estimation variance, there remains some questions concerning the accuracy of such an evaluation, especially because of the lack of samples given to LWPR.

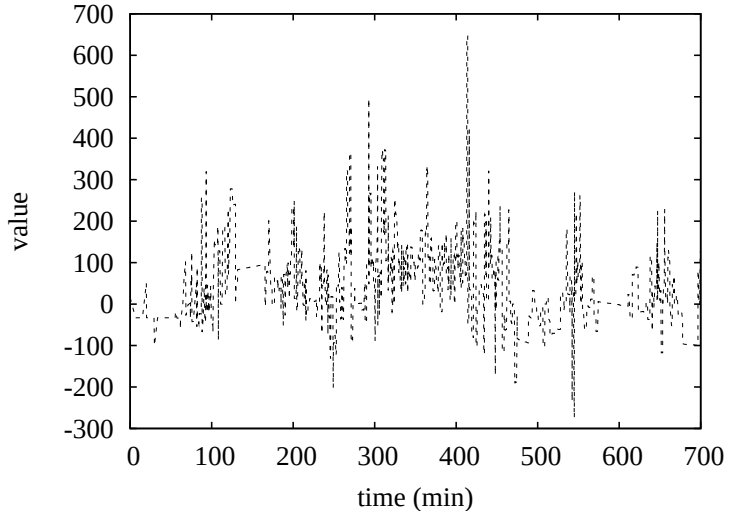
Even non-parametric methods require some tuning

Finally, it appeared that as the problem became more and more complex, the training and evaluation of the LWPR regression took more and more time and eventually handicapped the algorithm optimization time itself.

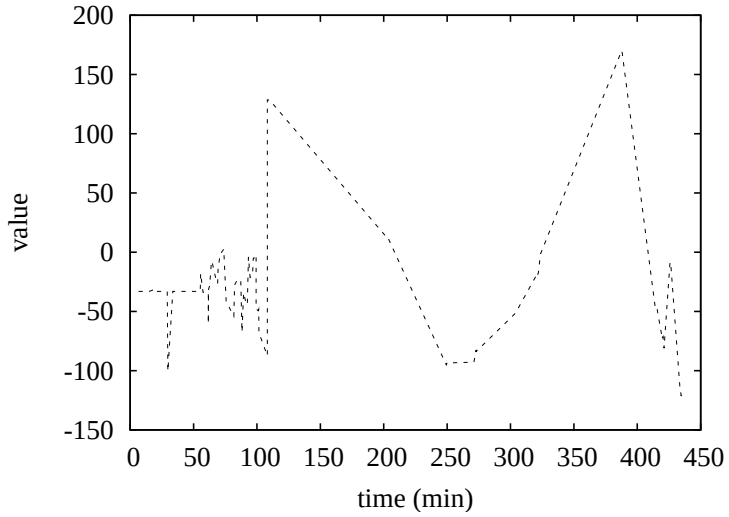
Our guess to explain this behavior was that the number of receptive fields and the associated linear models increased to a point where updating the models took a lot of time. We



(a) Fifty trajectories



(b) Raw evaluation



(c) Filtered evaluation

Figure 14.7: High variance estimation

tried to verify this hypothesis by using different initializations for LWPR's receptive fields. LWPR is initialized with a given covariance matrix defining the default shape of a receptive field. This covariance matrix is updated as new samples are processed in order to adapt to the principal directions of the regression. This usually avoids overfitting and the local explosion in the number of models. However, a bad initialization of this covariance matrix can lead to many more receptive fields and to different accuracies.

We tried different initializations for this covariance matrix, using isotropic matrices rI corresponding to a certain radius r. It is important to note that the "size" of a receptive field grows as the radius diminishes, since this covariance matrix serves to define a dot product for the activation levels of receptive fields. Using a small radius for the covariance matrix leads to less receptive fields, more overlapping and less precision. However, as the radius grows, after a certain threshold, we can except the receptive fields to be very small and thus lead to poor generalization properties.

Consequently, in order for LWPR to perform an efficient regression, this initial covariance matrix is very important. The following figures will illustrate several consequences of using different radii. As previously, we will generate a training set with fifty trajectories and will train LWPR with these trajectories. Then we will test the obtained regressor to measure its accuracy. This accuracy can be measured in terms of the mean squared error over trajectories or the maximum of these squared errors. We will also measure the training time and will relate it to the number of receptive fields. Finally, we will illustrate again the interest of confidence filtering on the mean squared errors.

Figures 14.8 to 14.11 illustrate these experiments. We tested several ranges of radii and the most interesting part happens for $r \in [0, 50]$.

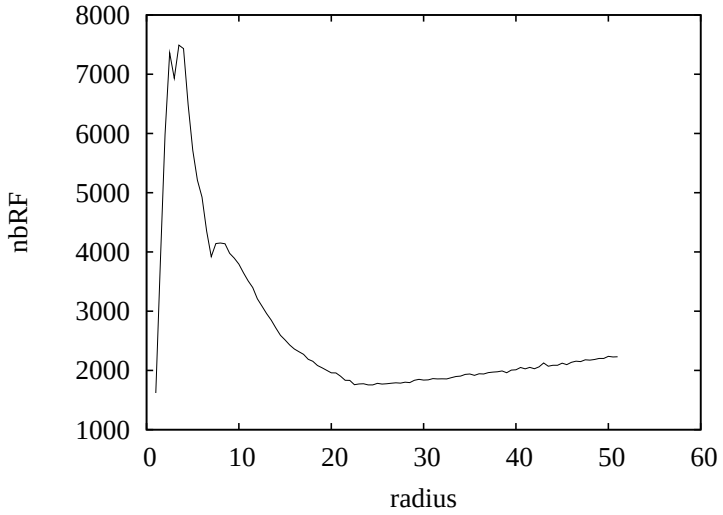


Figure 14.8: Number of receptive fields as a function of r

Figures 14.8 and 14.9 illustrate the linear relationship between the number of receptive fields and the training time. This validates our initial hypothesis. Now, the goal would be to find the best value of r for initialization.

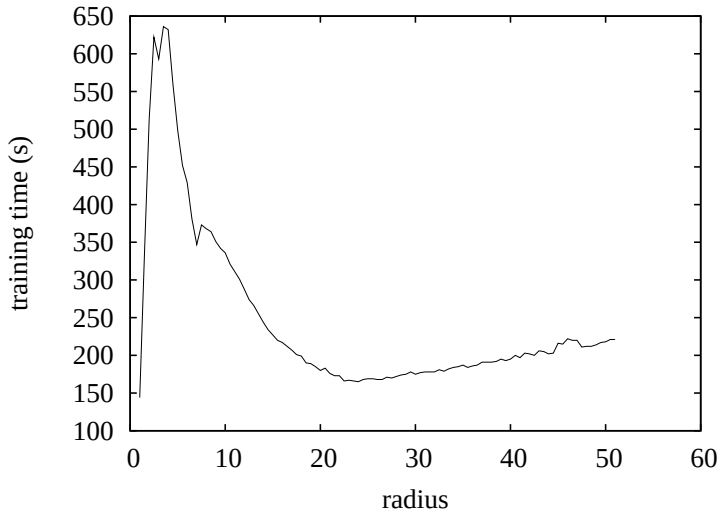


Figure 14.9: Training time as a function of r

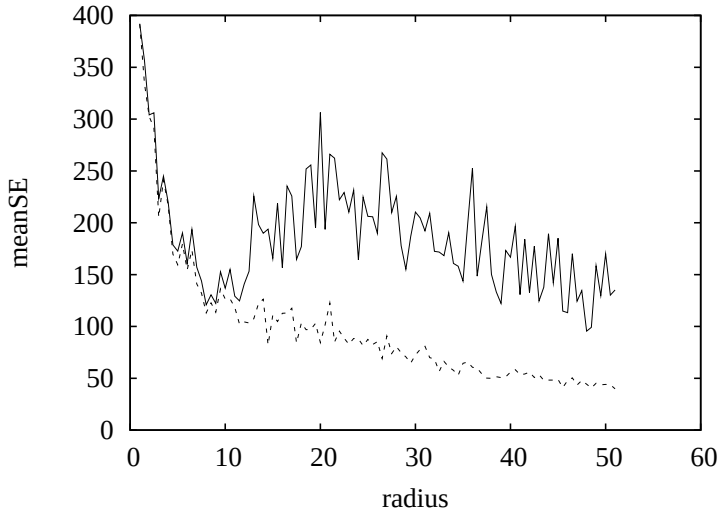


Figure 14.10: Mean Squared Error as a function of r

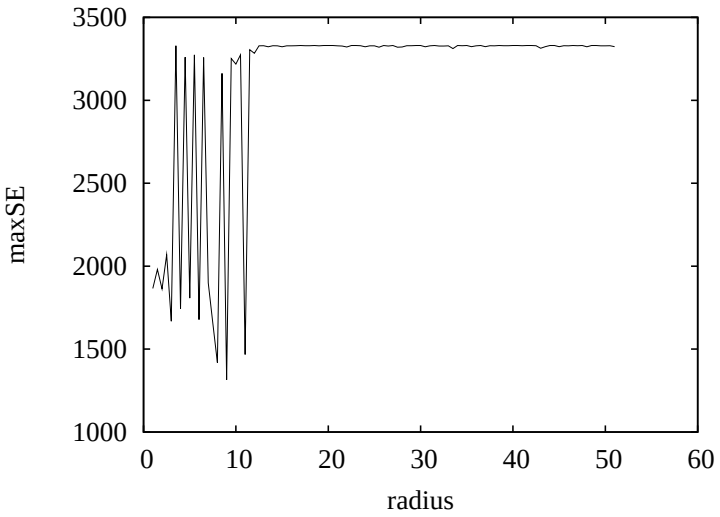


Figure 14.11: Max Squared Error as a function of r

In order to build a computationally efficient regression, one could use the value $r = 23$, corresponding to the minimum number of receptive fields. However, it is necessary to check whether this value of r yields a relevant value function.

The maximum squared error shown in figure 14.11 illustrates again the fact that as soon as a point is taken outside of a confidence region, its value should not be trusted because the associated error can be very high. The "low" values at the beginning of this curve actually correspond to the situation where only a few very large receptive fields are set over the whole state space and the regression is then almost a linear regression. The error is then the one of this linear regression. As the receptive fields become smaller, some parts of the state space become uncovered and the evaluation there returns zero, thus yielding the even larger maximum squared errors encountered for larger values of r.

Finally, figure 14.10 illustrates two important features. First, the solid line presents the evolution of the mean squared error. This evolution shows a minimum in the error for $r = 10$. Hence, it could imply that a compromise needs to be made between the number of receptive fields (optimal for $r = 23$) and the average error of the regressor (optimal for $r = 10$).

But the dashed line avoids making such a compromise. This dashed line shows the mean squared error only for points corresponding to confidence regions. As one could expect, this curve is below the solid curve since we have better knowledge of the value function in such regions. The second important thing to notice is that the mean squared error for such a filtered regression keeps decreasing as r increases. Hence, $r = 23$ seems to be a good value for isotropic initialization of the receptive field's covariance matrices.

Conclusion

These initial results concerning the "compact *iATPI*" implementation raise new questions and allow us to draw some conclusions:

- LWPR has all the desired theoretical properties required for our incremental regression

problem.

- Using the confidence filtering technique improved drastically the quality of the value function estimation.
- However, LWPR needs a lot of samples to provide a good estimation of the value function and this can become problematic, especially if $R^\pi(s)$ has a large variance.
- Moreover, good optimization behavior requires some fine tuning of LWPR's initial parameters. We illustrated this on isotropic initial covariance matrices but anisotropic ones could yield even better results[7]

For all these reasons, this attempt at building an efficient "compact *iATPI*" version resulted in incomplete results. Some of them reproduce the behavior of naive ATPI while others fail to improve the policy. More time would be necessary to analyze the essential reasons of this behavior and to reach a functional version of compact *iATPI*.

14.4.4 Full storage *iATPI*

The database trick

In order to build a comparison line between algorithms, we tried to implement a tabular representation of values and policies for *iATPI*. Tabular look-up quickly becomes intractable for our long trajectories yielding thousands of points. However, there remains the possibility to exploit the fact that we defined a distance over our state space and, more specifically, that we could *sort* all variables' values. Hence, we defined a *relational database* for the value function and for the policy. In these databases, elements can be sorted separately by each variable. This reduces the complexity of searching for the neighbors of a given state.

The value database is composed of pairs $(s, v) \in S \times \mathbb{R}$ and one can sort the database using any of the variables in s. Hence, with m being the number of variables in s, the worse case complexity of finding an element in the value database is m times the worse case complexity of finding an element in a hash table. Since, the average lookup, insertion and deletion times in a hash table is constant ($O(1)$ in the number of elements), the average time of looking up an element in the database is constant as well. Similarly, the policy database is composed of a multi-sorted set of pairs $(s, a) \in S \times A$.

Such a representation is more efficient than a brute-force tabular representation and trades time complexity against space complexity since it maintains m lists of references on the stored items.

Online local evaluation

A strong interest of the database representation is that there is no computation time associated to *building* the regressor since it is only composed of the set of collected samples.

When the learner needs to ask the value function for a state's value, the regressor computes online an average by looking up in the database the neighbors corresponding to the question's state. The efficient search in the database allows to compute such an average quite easily and to define a notion of confidence based on the samples' density around the

[7] For example by "stretching" the initial shape of the receptive fields along the axis of the temporal variable.

requested state.

In order to use our database search ability to its maximum, we decided to implement the following regression scheme. When one wishes to know the value of state s, two operations are performed:

1. Find all the neighbors of s within a given hypercube. Start the search with the temporal variable to prune the search space.

2. Compute the average value of all states s_i in this hypercube using a vector of weights $w_i = w(s_i)$.

Note that the notion of locality is defined using a weighted L_1 norm. However, the weights w_i were implemented using an L_2 norm in the state space. The above operation corresponds to performing a Parzen regression using a translation-invariant, non-linear kernel. This kernel is equal to zero for states s_i such that $\|s - s_i\|_1 > d_{max}$, where d_{max} is the kernel's outreach and where the L_1 norm is a weighted norm. In all states s_i for which $\|s - s_i\|_1 \leq d_{max}$, the kernel is proportional to $\|s - s_i\|_2$.

The same ideas drive the evaluation of the policy database: in the averaging operation, the above "+" operator is replaced by a weighted majority vote. Such a classifier is very similar to a k-nearest neighbors classifier.

The memory problem

Although we have not encountered this problem yet in our experiments, it appears obvious that storing all points works only up to a certain limit. When the database's size exceeds the available memory, such an option is not feasible again and purely online incremental methods such as LWPR become the only option since they store information in a more compact fashion and do not need to keep track of previously seen points.

Automatic confidence calculation — adjusting N_{sim} and N_a

We wrote earlier that our confidence function needed to be based on the density of samples. While this seems to be a good approximation when all random variables have the same variance, it appears to be a poorer method when the variables start becoming really different.

This was illustrated by the high variance example in the LWPR experiments. We would like to have a better confidence measure which allows us to have exactly the right number of samples in the different parts of the state space.

Defining such a measure implies finding bounds for the sampling process. The following paragraphs summarize the problem in terms of a stochastic process and presents the approach we chose.

Let us take the example of finding the right number of samples for a $Q(s, a)$ estimator (suppose we are in state s and we are testing action a). If we do not have any samples yet, we need to simulate in order to gather the information necessary to the evaluation of $Q(s, a)$. Suppose now that we already have a set of n samples $\{q_i\}_{1 \leq i \leq n}$ corresponding to previous experiments and our problem is to determine whether they constitute a sufficient statistics for evaluating the expected value for $Q(s, a)$. Let us formalize the problem the following way.

$Q(s,a)$ is the average of a random variable which we write Q. The q_i are a *sample path* of the stochastic process defined by drawing n occurrences of Q. Let us call Q_i the random variable corresponding to the i^{th} draw. If we form the set of random variables:

$$M_n = \frac{\sum_{i=1}^{n} Q_i}{n}$$

Then the *central limit* theorem tells us that the probability law of M_n tends to a Gaussian density of average $Q(s,a)$ and of standard deviation $\frac{\sigma^Q}{\sqrt{n}}$ where σ^Q is the standard deviation of the random variable Q.

In other words, we want to find the average $Q(s,a)$ of Q and we know that — as n tends to $+\infty$ — M_n is drawn using a Gaussian distribution, centered on $Q(s,a)$ and having a decreasing standard deviation. Hence, we are going to sample new states (new q_n) until this standard deviation becomes small enough for us to use the m_n realizations of M_n to evaluate $Q(s,a)$. Note that in practice, m_n corresponds to $\tilde{Q}_n(s,a)$.

The problem is that one cannot directly access the parameters of M_n's law. Hence, we proceed differently. We make the approximation that after a certain number of samples, M_n is indeed drawn using a normal law with parameters $\left(Q(s,a), \frac{\sigma^Q}{\sqrt{n}}\right)$. This is only an approximation since the central limit theorem only indicates the asymptotical behavior of M_n, when n tends to $+\infty$. In practice, this behavior is often verified after a relatively small number of samples[8].

We build the σ_n^Q value which corresponds to the standard deviation's unbiased estimator for the first n draws of Q_n[9]:

$$\sigma_n^Q = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n} q_i^2 - \frac{n}{n-1} m_n^2}$$

Then we have the standard deviation of M_n's approximate probability density function:

$$\sigma_n^M = \frac{\sigma_n^Q}{\sqrt{n}}$$

Finally, we can decide to stop the sampling when σ_n^M becomes smaller than a certain threshold k. Whenever this happens, the last m_n is used for $Q(s,a)$ and all sampled points are added to the database.

We can avoid recalculating these m_n and σ_n values from scratch each time we get a new q_n. For this purpose, we can use the following incremental expressions (and the intermediate variable ζ_n) to get m_n and σ_n from m_{n-1}, σ_{n-1} and q_n:

[8] An empirical threshold on this number of samples is often expressed as "with the central limit theorem, the infinite often starts at six".

[9] One could also use the $\sigma_n^Q = \sqrt{\frac{1}{n}\left[\sum_{i=1}^{n} q_i^2\right] - m_n^2}$ estimator without much difference in the results, but the latter is biased by a factor $\sqrt{\frac{n-1}{n}}$.

$$m_n = m_{n-1} + \frac{1}{n}(q_n - m_{n-1})$$
$$\zeta_n = \zeta_{n-1} + q_n^2$$
$$\sigma_n = \sqrt{\frac{1}{n-1}\zeta_n - \frac{n}{n-1}m_n}$$

Suppose such an approximation of M_n's behavior holds, ie. suppose M_n is indeed drawn according to a $\mathcal{N}(Q(s,a), \sigma_n^M)$ distribution and let us call H_0 this null hypothesis. Then the probability that m_n was drawn with an error ϵ from $Q(s,a)$ can be written:

$$Pr(|Q(s,a) - M_n| > \epsilon | H_0) = 1 - \int_{Q(s,a)-\epsilon}^{Q(s,a)+\epsilon} \frac{1}{\sigma_n^M \sqrt{2\pi}} e^{\frac{1}{2}\left(\frac{x-Q(s,a)}{\sigma_n^M}\right)} dx$$

In other words, the probability of correctly picking m_n with at most an error ϵ from $Q(s,a)$ is given by:

$$Pr(|Q(s,a) - M_n| < \epsilon | H_0) = \text{erf}\left(\frac{\epsilon}{\sigma_n^M \sqrt{2}}\right)$$

This provides — with the H_0 hypothesis — a PAC-style guarantee on the evaluation of $Q(s,a)$.

Let H_0 be the assumption stating that "the asymptotical behavior of $\tilde{Q}_n(s,a)$ towards a distribution $\mathcal{N}(Q(s,a), \sigma)$ is a good approximation of $\tilde{Q}_n(s,a)$'s law, early in the sampling process" (ie. for small n). With this assumption, we can guarantee that $\tilde{Q}_n(s,a)$ is an estimate of $Q(s,a)$ with an error smaller than ϵ and with probability greater than $\text{erf}\left(\frac{\epsilon}{\sigma_n^M \sqrt{2}}\right)$. In other words, whenever σ_n^M becomes smaller than $\frac{\epsilon}{\text{erf}^{-1}(p)\sqrt{2}}$, we can guarantee to have m_n within ϵ of $Q(s,a)$ with probability at least p.

Such a bound can be compared with the Hoeffding bound stating that if Q_n takes its values within an interval of length d, then $Pr(|Q(s,a) - M_n| \leq \epsilon) \leq 2\exp(\frac{-2n\epsilon^2}{d^2})$. A more thorough analysis of the asymptotic behavior of additive functions in Markov chains can be found in [Maxwell and Woodroofe, 2000] or [Dedecker, 2008] with similar considerations and bounds.

The consequence of such an adaptive confidence measure is to refine the sampling in regions of large variance while avoiding oversampling in other regions. This procedure serves to automatically stop the sampling for N_a, hence it corresponds to an adaptive and automatic tuning of the N_a variable. Similarly, with such a procedure, N_{sim} does not need to be entered by hand: a set of trials is stopped whenever the new value of the initial state has a "σ_n^M" lower than the threshold given by the result above.

We gave the previous bounds for the evaluation of $Q(s,a)$ quantities. However, similar bounds can be derived for the simple estimation of V^π. During evaluations, the same procedure can be used to enrich the database if the samples already collected result in a too large σ_n^M. More specifically, during an evaluation, we want to stop the rollout as soon as possible if we encounter a state in which we can trust the value function. So whenever we reach s, we can use our Parzen kernel to compute an estimated $\tilde{V^\pi}(s)$ and the associated σ_n^M from the existing neighboring entries in the database. If this weighted σ_n^M is below the admissibility threshold, we can stop the current simulation in s, else we need to continue sampling.

This last motivation of early stopping in the rollouts comes from a simple lesson from experience. While we have a rather efficient generative model for our GSMDP (in terms of computation time), simulation is not as costless as if we were simulating a standard MDP, mainly because of the computation occurring behind the scenes to maintain the clocks' values. Hence, simulation still comes at a cost which we might want to balance by performing early rollout termination.

Conclusion

Consequently, the study of the database version of *iATPI* shows that one only needs three parameters to build both the regressor and the classifier:

- A translation-invariant Parzen "window" defining the points which will be considered to infer the value of a given state. This window is given as the W operator:

$$W : \left\{ \begin{array}{lcl} S & \rightarrow & \mathcal{F}(S,\{0,1\}) \\ s & \mapsto & \left\{ \begin{array}{lcl} S & \rightarrow & \{0,1\} \\ s_i & \mapsto & \left\{ \begin{array}{ll} 1 & \text{if } \|s_i - s\|_1 \leq d_{max} \\ 0 & \text{else} \end{array} \right. \end{array} \right. \end{array} \right.$$

  Such an operator is completely defined by the d_{max} parameter and the chosen L_1 weighted norm.

- A "weight" operator w corresponding to the density of the Parzen kernel. Similarly to the window operator, the weight operator is given as:

$$w : \left\{ \begin{array}{lcl} S & \rightarrow & \mathcal{F}(S,\mathbb{R}^+) \\ s & \mapsto & \left\{ \begin{array}{lcl} S & \rightarrow & \mathbb{R}^+ \\ s_i & \mapsto & \|s_i - s\|_2 \end{array} \right. \end{array} \right.$$

  This operator is parameterized by the chosen weighted L_2 norm. The complete corresponding Parzen kernel is defined by the normalization of the product $W(s,\cdot)w(s,\cdot)$.

- A confidence threshold k defining a limit on the σ_n^M variables. While this limit is not reached, sampling continues. This k value can be inferred from an estimation error ϵ and a correct estimation probability p, using the PAC-style bounds given previously.

14.5 Conclusion

The results from the different versions of *iATPI* are still too incomplete to draw final conclusions as to the efficiency of the method and to its interests and weaknesses. However, the initial experiments presented above showed two complementary things. First, that the question of using statistical learning tools was rooted in various problematics. Namely:

- Compactly representing and storing acquired experience.
- Generalizing experience to unexplored states in a statistically relevant fashion.

These problems are crucial to efficient learning in large state spaces, and even more important in the case of continuous or hybrid state spaces.

Despite recent attempts at characterizing sound statistical learning systems for reinforcement learning (*eg.* [Ormoneit and Sen, 2002; Farahmand et al., 2008; Dimitrikakis and Lagoudakis, 2008]), this question remains an open area of research to which we hope to contribute with the first results of *iATPI*.

Secondly, these first results showed important features for a good implementation of *iATPI*. It provided some insight at how to characterize the algorithm's soundness and efficiency and hence, at how to improve its performance.

The *iATPI* algorithm relies on efficient generalization, based on a sampling process. Its first results pointed out possible weaknesses of Statistical Learning methods concerning the unexplored parts of the state space and provided some insight as to the statistical relevance of sampling.

Finally, the optimization of policies for Temporal Markov Decision Problems through the *iATPI* algorithm builds on both the statistical learning objects and the presence of the time variable. *iATPI* is our contribution to solving *external-event* temporal problems using these statistical learning tools and exploiting the presence of time. Time is used in the following features:

- Finite trials. The bounded temporal horizon guarantees the termination of trials. This corresponds to defining absorbing terminal states in a classical MDP problem but in the case of temporal problems, there might be a very large subset of these states since the process might stop, at the temporal horizon, in any state of the state space.
- Structured dynamics. The time variable conditions the evolution of the problem more than any other one. On this topic, as mentioned earlier, time could be replaced by an other crucial variable. However, in practice, many time-dependent problems can benefit of special temporal modeling. Hence, time remains the structuring parameter of the problems dynamics.
- Structured policies. On top of using the statistical learning tools to extract the inherent policy structure. Including time in the state space helped recovering — at least part of — the behavior of a Markovian problem. In other words, observing time compensates partially for the non-observability of event clocks. Hence, time is a crucial structuring variable for the found policies.

Finally, our contribution to the resolution of Temporal Markov Decision Problems also consists in the definition of the DECTS framework and its associated "learner" model which builds a bridge between the reinforcement learning optimization community and the DEVS simulation community.

DECTS are an abstraction of the problems we want to capture as Temporal Markov Decision Problems and a more general framework than MDPs. They constitute a first step in the process of defining a common representation for the sequential decision optimization processes and the simulation ones.

15

Conclusion

This chapter summarizes the progression of ideas followed throughout part III. It gathers our contributions in a synthetic view and opens areas of perspective work.

15.1 Summary

Part III's main focus has been centered on the question of solving *explicit-event* Temporal Markov Decision Problems. Starting with the statement that such processes were often too complex too represent as an MDP, we turned towards the general setting of DEVS theory to represent the generative models of our systems. This brought our first contributions: we expressed our decision problem as a continuous time, hybrid GSMDP, analyzed the complexity of the process and its non-Markovian behavior and finally bridged the gap from GSM(D)P to DEVS models. This provided us with a general description of the system to control, compatible for interaction with other discrete event systems.

Then we introduced the idea of greedy simulation for forward search in Asynchronous Policy Iteration, yielding the RTPI algorithm. This algorithm is strongly inspired by the ideas of Asynchronous Dynamic Programming of [Bertsekas and Tsitsiklis, 1996] and the greedy exploration of [Barto et al., 1995] and [Bonet and Geffner, 2003b]. This idea is our second contribution and comes from the statement that, for the Temporal Markov Decision Problems we had, we knew the initial state, had a way to simulate a policy and wished to limit the exploration of the state space to the most likely states.

These two first bricks of simulation and of algorithmic procedure then lead to define the ATPI algorithm which makes use of the observable time of GSMDPs to avoid storing a policy during an RTPI algorithm and to facilitate Bellman backups. The *naive* ATPI algorithm is our third contribution and the first step towards its improved version. We tested ATPI on the subway problem, showing at the same time very promising results and crucial weaknesses due to the memoryless structure of the algorithm itself.

The initial results on naive ATPI drove us to several conclusions. First of all, we took some distance with the — already quite general — framework of GSMDPs and tried to abstract the core properties of the systems we wished to control. This lead us to define, as our fourth contribution, Discrete Events Controllable Temporal Systems which are DEVS-compatible discrete events systems, capturing the event-driven properties of GSMDPs, their

non-Markovian behavior and the general class of temporal discrete event control problems. One of the most important points in this contribution was to represent the simulated system and the optimization process (the DECTS learner) in the same discrete events formalism, using dynamic models and recursive simulation principles and introducing the idea of *decision objects*. This allows to consider an optimization process as a discrete event system and to hierarchically build optimization procedures.

Finally, building on the DECTS description, we rebuilt the ATPI algorithm, taking care to avoid the previously identified flaws. This lead to our last contribution, the *iATPI* algorithm. *iATPI* brings together results from simulation theory, the algorithmic procedure of RTPI and tools borrowed to the field of Statistical Learning. It aims at solving high-dimension, continuous state space Temporal Markov Decision Problems, involving the following features:

- Simulation-based exploration (greedy search),
- Simulation-based evaluation (Monte-Carlo evaluation),
- Local generalization of experience in the state space (Smoothness of the regressor / classifier for the *decision objects* we consider).

Even though there was not enough time to extensively test *iATPI*, preliminary results opened many areas of improvement. Namely, it provided hints concerning the statistical relevance of the *decision objects*; it also defined a notion of confidence, which serves in guiding the exploration for optimization; it finally showed the necessary properties of appropriate tools for regression, classification and density estimation used in conjunction with *iATPI*.

15.2 Perspectives

Quite obviously, the first step in future work consists in testing extensively the different implementations of *iATPI*. Testing should involve different versions regarding the statistical learning tools used, but also dealing with very different benchmarks as the subway problem, the airport problem, the Mars rover or the bi-agent coordination problem.

Building an efficient version of *iATPI* might imply exploring or defining other tools, in order to build relevant value functions and policies. Further investigation and refinement of the LWPR method (to compensate for its current slow learning feature or to adapt it to classification problems in the flavor of Gaussian Processes for instance) is a possible area of improvement.

Another interesting perspectives takes place within the DECTS framework. Having written the simulated system and the learner in the same description language allows us to consider the whole as a new discrete event system and to hierarchically build other optimization systems. Among other things, this opens a door towards modular discrete optimization systems implementation. More importantly, this opens a new perspective to verification, validation and testing in a unified framework of discrete event systems.

Part IV

Conclusion

General conclusion

From the initial problematic of deciding, under uncertainty constraints, in the context of non-stationary problems, we have explored two distinct modeling fields, both corresponding to extensions of classical Markov Decision Processes.

We first made the assumption that an "all-integrated" stochastic model of the process to control was readily available under the form of a TMDP. These TMDPs captured the notion of *implicit-event* decision models in the presence of unbounded, continuous time and explicit time-dependency. From this modeling framework, we focused on providing a sound basis to its optimality equation, on improving its resolution scheme and extending its expressiveness. This lead to two more general modeling frameworks: SMDP+ and — more importantly — XMDP, which highlighted the interests and limitations of TMDP, generalized its optimality equation and opened the door to the generalization of its resolution scheme.

But *implicit-event* decision models are often not available as such and it is much easier to describe a temporal decision problem through *explicit-event* decision models. The drawback of such models is that we are no longer able to optimize policies for them with the general guarantees we had with MDPs. Hence, we focused our efforts on comprehending and formalizing these *explicit-event* models. This study brought us from the field of Discrete Events Systems specification to the techniques of Statistical Learning, all serving the same purpose: building a sound generic algorithm for explicit-events Temporal Markov Decision Problems.

The specific fields and contributions presented in parts II and III could be summarized the following way:

- Part II focused on formalization of implicit-event decision models, extending the well-known TMDP model to a more general framework of optimization (piecewise polynomial representations) and introducing the general case of observable continuous time with the XMDP formalization. Efforts in the resolution of such temporal problems were directed at backward induction algorithms, performing a value iteration-like optimization in a prioritized way.

- Part III was oriented towards modeling of explicit event systems where inclusion of the observable time variable partially compensated for the loss of Markov's property. Algorithmic contributions focused on forward search methods, using the generalizing properties of Statistical Learning techniques to deal with high-dimensional hybrid state

spaces.

We won't recall further here the contributions brought by the study of these two distinct fields and will refer the reader to the conclusion chapters of their respective parts (chapters 10 and 15) for these contributions. Instead, it seems interesting to take a last look at the very nature of the temporal variable, in order to draw some more general conclusions.

The discussion of section 2.3 — which underlined the differences and similarities between standard MDPs and Temporal Markov Decision Problems — brought up three different notions of time in the modeling context of stochastic processes: the process' time (the successive decision epochs numbering), the transition or sojourn time (the temporal extension of transitions) and the physical time or clock (the measure the agent has by looking at its watch). Since we remained in the discrete event framework throughout the thesis, our focus was set on the two last notions, their relationships and dependencies. In the end, it appears that:

- physical time *can* be dealt with like any other non-replenishable variable. The generalization from TMDPs to XMDPs is probably the best illustration we could have found of such a property. Even if they define an unbounded time *a priori*, TMDPs rely on the implicit assumption that the knowledge about non-stationarity is finite and corresponds to a certain time interval. Hence, one could turn TMDPs back to the case of continuous variable MDPs. However,

- physical time *should not* be dealt with like any other continuous variable in a Temporal Markov Decision Problem. The first reason for this is the fact that time is the structuring variable of the process: it conditions the transition functions, the reward model and — consequently — the optimal policy. Giving time a special place in the optimization process — as the TMDP and XMDP frameworks do — helps structuring the optimization itself by taking advantage of the *causality principle*[1]. Furthermore, the introduction of an observable continuous physical time variable in the GSMDP framework — when the initial state is known — partially compensated for the loss of Markov behavior due to the non-observability of the internal dynamics deciding the sojourn times.

From the practical point of view, both the $TMDP_{poly}$ and the *iATPI* algorithms — which form the main algorithmic contribution of the thesis — can still benefit from improvements, better understanding and better implementations. However, they both constitute a new contribution to their respective fields and to the question of temporal planning and learning under uncertainty.

Finally, this question of dealing with time is far from being closed. Different communities adopted different formalisms to deal with it: temporal logic, temporal planning, TMDPs, differential propagation equations, ... all constitute distinct attempts at capturing the time-dependency of decision problems and their inherent structure. We presented in this thesis a contribution to the specific framework of decision under uncertainty, trying, at a modest level, to lay bridges between communities (planning under uncertainty and discrete events modeling for instance). However, both in our fields and in the general case, time retains its puzzling aura, confirming that its study still necessitates more ... time.

[1] direct consequence of the fact that time is a non-replenishable variable.

Appendix

A

Computing complex operations on piecewise polynomial functions

A.1 Basics

Real-valued coefficient polynomial functions are commonly used as an approximation framework for continuous functions, for interpolation purposes or for compact representation of basis functions. They constitute an easy-to-use representation because they allow for both low memory cost storage of the function in a compact form (coefficients storage) and practical operations between functions.

The first application of polynomial functions to interpolation consists in Lagrange polynomials. But one could also mention Bezier curves and, of course, the well developed theory of spline functions (see [Ahlberg et al., 1967]).

Polynomials offer the possibility to turn complex functional operations into coefficient manipulation. This makes the framework of polynomial interpolation particularly attractive.

However, an important misconception needs to be lifted here: polynomial functions — especially Lagrange polynomials — are not always the "simplest" fitting curve. Indeed, as the number of fitted points increases, performing exact interpolation implies using high degree polynomials and thus introducing the corresponding inflexions and variations in the fitting curve. Splines correct this problem by introducing discontinuities in the function or in its derivatives.

This appendix presents some of the problems associated with computing operations at the functional level on *piecewise* polynomial functions. Due to lack of time, this presentation might be incomplete. All the algorithms mentioned here and other ones were implemented in version 0.3 the *POLYTOOLS* library. This library is available at `http://emmanuel.rachelson.free.fr`.

A.2 Common dangers of coefficient manipulation

Imagine trying to compute the difference of two polynomial functions, the first one being $p_1(x) = \frac{1}{3}x + 1$ and the second one being $p_2(x) = 0.333333333333334x + 1$. Depending on how arithmetic operations are implemented on the specific machine used and on machine precision, the resulting $p_3 = p_1 - p_2$ polynomial might result in being 0 or ϵx where ϵ is a very small number.

While this is not a surprise when one is used to finite machine precision calculation, it holds a danger for polynomial handling. When we wants operations to be automated, we cannot afford to have such a random behavior in calculation results. We take the "graphical" point of view and state that the two above possible polynomials for p_3 are equal.

Another way to present this problem is simply to state that finite machine precision leads to inexact results even when the formulas are exact. Hence, one should be careful about the operations performed. For example, when looking for maxima of the $f(x) = -x^4$ polynomial, if the root finding algorithm used on the derivative returns an approximate result x 0, then using the first non-zero derivative to determine whether the optimum found is a maximum or a minimum is usually not a robust method. This is because it implies testing whether a real value is exactly equal to zero or not and this specific testing is very error-prone.

Consequently, we take the option to state that a polynomial function can only be numerically defined with respect to a maximum range of its argument x and with a threshold parameter on the coefficients in order to avoid having ill-conditioned polynomials[1]. See for example what happens if you take the remainder of the Euclidean division of a polynomial by a second one that has a very small highest order coefficient: the resulting polynomial has a huge highest order coefficient. This is particularly unwished and induces numerical errors, it is even critical in the application of the high-level Sturm's theorem.

Consequently, to insure numerical stability of polynomial handling, we adopt the following finite precision criterion.

A polynomial function is numerically defined by the triplet $\langle \mathbf{a}, M, \epsilon \rangle$:

- $\mathbf{a}$ is the vector of coefficients. It contains $n + 1$ real values (n being the polynomial's degree).
- M is the *definition span*, it is the largest value of x, regardless of sign, which will ever be presented to the polynomial function.
- ϵ is the precision threshold under which one considers a polynomial evaluation to be null.

We introduce the following simplification scheme for numerical calculation stability. Whenever a coefficient a_i in $\mathbf{a}$, corresponding to the degree i term verifies $a_i M^i < \epsilon$, this coefficient is set to zero.

A.3 Usual operations: polynomial arithmetic, evaluation and root finding

Due to lack of time, we only provide a short review of these methods which have been more widely developed in specifically dedicated books (see for example [Press et al., 2007]).

Addition and subtraction of polynomial functions is rather straight forward, given the previous simplification scheme.

[1] By ill-conditioned polynomials, we mean polynomials which would have a very large ratio of $\frac{A}{a}$ where A is the largest coefficient and a the smallest.

Multiplication of polynomials corresponds to the convolution of their coefficient vectors.

Implementing a polynomial division scheme (Euclid division) can be done with minimal complexity as presented in [Press et al., 2007], given the above simplification scheme.

Evaluation follows Horner's method which relies on factored polynomial form:

$$p(x) = (((a_n x + a_{n-1})\, x + a_{n-2})\, x + \ldots)\, x + a_0$$

Still with the same simplification scheme, derivation and integration of polynomial functions is an easy task. One should note however that these operation are no more guaranteed to be commutative for ill-conditioned polynomials because of the simplification scheme.

Finally, for root finding, exact formulas exist for polynomials up to degree four (included). Namely:

- degree 0, trivial case
- degree 1, linear case
- degree 2, Newton's formula
- degree 3, Cardan's or Sotta's Formula
- degree 4, Ferrari's or Descartes's formula

Still these formulas should be handled keeping the finite precision problem, especially when testing real numbers for equality to zero.

For polynomials of degree equal or greater than five, one often uses approximate methods. For this purpose, Sturm's theorem — presented in [Sturm, 1835] — provides a very nice dichotomy algorithm for isolating roots of a polynomial function. We recall this theorem below, writing $\mathrm{rem}(p_1, p_2)$ the remainder of the division of p_1 by p_2.

Let p be a polynomial function. Let $\{p_n\}_{n\in[0,N]}$ be the finite sequence of polynomials defined by:

$$\begin{aligned} p_0 &= p \\ p_1 &= p' \\ p_i &= \mathrm{rem}(p_{i-2}, p_{i-1}) \\ 0 &= \mathrm{rem}(p_{N-1}, p_N) \end{aligned}$$

Finally, let a and b be two real numbers such that $a < b$, $p(a) \neq 0$ and $p(b) \neq 0$ and let $\sigma(a)$ (resp. $\sigma(b)$) be the number of sign changes in the $(p_0(a), \ldots, p_N(a))$ sequence where zeros are not counted as sign changes.
Then the number of real, distinct roots of p in the $[a, b]$ interval is $\sigma(a) - \sigma(b)$.

Sturm's theorem provides a nice theoretical way of bracketing roots. However, it is subject to the same numerical difficulties mentioned earlier, especially when some $p_i(a)$ values become close to zero. This can lead to an erroneous number of expected roots in a given

interval.

Sturm's theorem is often used in conjunction with a Newton-Raphson gradient descent algorithm in order to refine the value of the found root.

Our *POLYTOOLS* library offers implementations of all these algorithms.

The piecewise polynomial case derives directly from the polynomial function case.

A.4 Convolutions

We dedicate a specific section to the problem of computing the convolution of two piecewise polynomial functions for two reasons:

- This operation is a non-trivial operations, imply many intermediate steps.
- It is one of the core functionalities needed for the $TMDP_{poly}$ planner, which justified the creation of a specific *POLYTOOLS* library.

The goal here is to calculate the function:

$$h(t) = \int_{-\infty}^{\infty} f(x)g(t-x)dx$$

Where f is a piecewise polynomial function of degree A and g is a piecewise polynomial function of degree B.

A.4.1 Preliminary: convolution of a piecewise polynomial function with any probability distribution function

This preliminary paragraph shows what would happen if f or g was not a piecewise polynomial function.

A.4.2 Problem introduction

Let us introduce the problematic of this section by starting with the simple case of solving equation A.4 in the case of any function f (not necessarily polynomial), and a polynomial function g (defined in one piece). We will write:

$$g(x) = \sum_{j=0}^{B} b_j x^j$$

Thus:

$$\begin{aligned}
g(t-x) &= \sum_{j=0}^{B} b_j (t-x)^j \\
&= \sum_{j=0}^{B} b_j \sum_{k=0}^{j} C_j^k t^k (-x)^{j-k} \\
&= \sum_{j=0}^{B} \sum_{k=0}^{j} b_j C_j^k t^k (-x)^{j-k} \\
&= \quad b_0 \\
&\quad + b_1 (t-x) \\
&\quad + b_2 (t - 2tx + x) \\
&\quad \vdots \\
&\quad + b_B \sum_{k=0}^{B} t^k C_B^k (-x)^k \\
&= \quad t^B \left(b_B C_B^B \right) \\
&\quad + t^{B-1} \left(b_B C_B^{B-1} (-x) + b_{B-1} C_{B-1}^{B-1} \right) \\
&\quad + t^{B-2} \left(b_B C_B^{B-2} (-x)^2 + b_{B-1} C_{B-1}^{B-2} (-x) + b_{B-2} C_{B-2}^{B-2} \right) \\
&\quad \vdots \\
&\quad + t^1 \left(b_B C_B^{B-(B-1)} (-x)^{B-1} + \ldots + b_1 C_1^1 \right) \\
&\quad + t^0 \left(b_B C_B^{B-B} (-x)^B + \ldots + b_0 C_0^{B-B} \right) \\
&= \sum_{j=0}^{B} t^j \sum_{i=j}^{B} b_i C_i^j (-x)^{i-j} \\
&= \sum_{j=0}^{B} t^j \sum_{m=0}^{B-j} (-1)^m b_{j+m} C_{j+m}^j x^m
\end{aligned}$$

So we have:

$$h(t) = \int_{-\infty}^{\infty} f(x) \left(\sum_{j=0}^{B} t^j \sum_{m=0}^{B-j} (-1)^m b_{j+m} C_{j+m}^j x^m \right) dx$$

Hence:

$$h(t) = \sum_{j=0}^{B} t^j \left(\sum_{m=0}^{B-j} (-1)^m b_{j+m} C_{j+m}^j \int_{-\infty}^{\infty} x^m f(x) dx \right)$$

The quantity $\int_{-\infty}^{\infty} x^k f(x) dx$ (provided that this quantity exists) is the mth moment of a random variable governed by a probability density function f. We will write it m_k and thus:

$$h(t) = \sum_{j=0}^{B} t^j \left(\sum_{k=0}^{B-j} (-1)^k b_{j+k} C_{j+k}^j m_k \right)$$

Therefore, one can conclude that in order to compute h for any f and for a polynomial g of degree B, one needs to be able to compute the B first moments of f. Let us now turn to piecewise polynomial functions. We will write β_i the set of bounds limiting the definition intervals of g. $\mathcal{B}$ is the number of definition intervals of g.

If $g(x)$ is piecewise defined over the successive intervals $[\beta_1, \beta_2]$, $[\beta_2, \beta_3]$, ..., $[\beta_{\mathcal{B}}, \beta_{\mathcal{B}+1}]$, with $\beta_1 < \ldots < \beta_{\mathcal{B}+1}$, then the function $g_t(x) = g(t-x)$ is defined over the intervals $[t-\beta_{\mathcal{B}+1}, t-\beta_{\mathcal{B}}]$, ..., $[t-\beta_3, t-\beta_2]$, $[t-\beta_2, t-\beta_1]$. We write $g_{i,t}$ the restriction of g_t to the interval $[t-\beta_{i+1}, t-\beta_i]$. Therefore, calculating h turns to:

$$h(t) = \sum_{i=1}^{\mathcal{B}} \int_{t-\beta_{i+1}}^{t-\beta_i} f(x) g_{i,t}(x) dx$$

We have seen previously that on each $[t-\beta_{i+1}, t-\beta_i]$ interval, $g_t(x)$ could be written as:

$$g_{i,t}(x) = \sum_{j=0}^{B} t^j \sum_{m=0}^{B-j} (-1)^m b_{i,j+m} C_{j+m}^j x^m$$

So, the calculation of $h(t)$ can be written:

$$h(t) = \sum_{i=1}^{\mathcal{B}} \int_{t-\beta_{i+1}}^{t-\beta_i} f(x) \sum_{j=0}^{B} t^j \sum_{m=0}^{B-j} (-1)^m b_{i,j+m} C_{j+m}^j x^m dx$$

So it turns out that calculating h for piecewise polynomial g functions means being able to compute the quantity $\int_{t-\beta_{i+1}}^{t-\beta_i} x^k f(x) dx$. While calculating f's moments was a standard operation for most probability density functions, when the integral's bounds are not infinite anymore, the calculation becomes more complex and there is no standard method. This introductory analysis illustrates why we chose piecewise polynomial probability density functions in the $TMDP_{poly}$ algorithm.

Now we can turn to our first objective in this section, namely calculate $h(t)$ for piecewise polynomial f and g functions. We will first show that h is piecewise polynomial too and will analyze its definition intervals. This will break the calculation into integral calculations over standard polynomials.

A.4.3 Breaking the problem into pieces

Let us call $\{\alpha_i\}_{1 \leq i \leq \mathcal{A}+1}$ and $\{\beta_j\}_{1 \leq j \leq \mathcal{B}+1}$ the bounds of f and g's definition intervals respectively. The question in calculating h is to define intervals over which the definition of f and g is constant in order to perform the integration.

Suppose we have found such intervals. Their bounds are either elements of $\{\alpha_i\}_{1 \leq i \leq \mathcal{A}+1}$ or of $\{\beta_j\}_{1 \leq j \leq \mathcal{B}+1}$. Over one of these intervals, the $f(x)g(t-x)$ product can be written as a polynomial in t where the coefficients depend on x. More specifically, the coefficients of this $(f(x)g_t(x))(t)$ polynomial are themselves polynomials in x. The interval's bounds being linear in t, we can deduce that over each of these intervals, $h(t)$ is a polynomial function and thus that h is piecewise polynomial.

Now we need to find these intervals, depending on the value of t. Figure A.1 clarifies the problem.

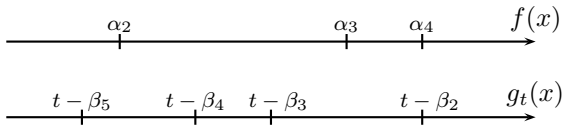


Figure A.1: Example of definition intervals for a given t

It appears that for a very small t (close to $-\infty$), the list of bounds defining intervals over which both f and g have a constant definition is given by:

$$t-\beta_{\mathcal{B}+1}, t-\beta_{\mathcal{B}}, \ldots, t-\beta_2, \alpha_2, \ldots, \alpha_{\mathcal{A}+1}$$

We have dropped the β_1 and α_1 values on purpose to avoid inserting infinite valued bounds into the list. Then, as t increases, $t-\beta_2$ becomes larger than α_2 and the bounds permute. This happens for $t=\gamma_2=\alpha_1+\beta_1$. And the process goes on in the same manner, t grows and as soon as one of the bounds switches position with an other, a new threshold γ on t is defined, and the list of bounds is updated with the new list of bounds which have been reordered. This provides an efficient manner of computing $f(x)g_t(x)$'s definition intervals — they are defined by the γ_k — and at the same time to calculate the list of bounds used for the calculation of h on each of the t intervals defined by the γ_k.

Finally, for $t\in[\gamma_k,\gamma_{k+1}]$, we have a list of bounds — written either as α_i or as $t-\beta_j$ — defining ordered intervals over which f and g_t have a constant definition. The $h(t)$ polynomial over $[\gamma_k,\gamma_{k+1}]$ is given by the sum over all of these intervals of the quantities:

$$\int_{bound_{[\gamma_k,\gamma_{k+1}],l}}^{bound_{[\gamma_k,\gamma_{k+1}],l+1}} f(x)g(t-x)dx$$

Between these bounds, f and g are simple polynomial functions. So our global problem of computing $h(t)$ transforms into lots of small problems of computing quantities like:

$$\int_{\gamma}^{\delta} f(x)g(t-x)dx$$
$$\int_{\gamma}^{t-\delta} f(x)g(t-x)dx$$
$$\int_{t-\gamma}^{t-\delta} f(x)g(t-x)dx$$

Where γ and δ are real numbers. The next paragraphs explain how to perform such a calculation.

A.4.4 Preliminary calculations

Let f and g be written as:

$$\begin{aligned} f(x) &= \sum_{i=0}^{A} a_i x^i \\ g(x) &= \sum_{j=0}^{B} b_j x^j \end{aligned}$$

We want to calculate:

$$S(t) = \int_{bound_1}^{bound_2} f(x)g(t-x)dx = \int_{bound_1}^{bound_2} \left(\sum_{i=0}^{A} a_i x^i\right)\left(\sum_{j=0}^{B} b_j (t-x)^j\right) dx$$

Where $bound_1$ and $bound_2$ are either real valued bounds as in γ or "shifting bounds" as in $t-\delta$. Before we compute the integral itself, we can try to simplify the expression under the integral sign. As calculated in the previous paragraph, one can write:

$$g(t-x) = \sum_{j=0}^{B} t^j \sum_{i=j}^{B} b_i C_i^j (-x)^{i-j}$$

Let bb_j be the polynomial:

$$bb_j(x) = \sum_{i=j}^{B} b_i C_i^j (-x)^{i-j}$$

Introducing $m = i - j$, one can rewrite bb_j as:

$$bb_j(x) = \sum_{m=0}^{B-j} (-1)^m b_{j+m} C_{j+m}^j x^m$$

Let $b\vec{b}_j$ be the vector containing the $B - j + 1$ coefficients of bb_j:

$$b\vec{b}_j = \begin{bmatrix} b_j C_j^j \\ -b_{j+1} C_{j+1}^j \\ \vdots \\ (-1)^{B-j} b_B C_B^j \end{bmatrix}$$

And let $\vec{a}$ be the vector containing f's coefficients:

$$\vec{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_A \end{bmatrix}$$

One has:

$$\begin{aligned} f(x)g(t-x) &= \left(\sum_{i=0}^{A} a_i x^i\right)\left(\sum_{j=0}^{B} b_j (t-x)^j\right) \\ &= \left(\sum_{i=0}^{A} a_i x^i\right)\left(\sum_{j=0}^{B} bb_j(x) t^j\right) \end{aligned}$$

We can introduce vector $\vec{c_j}$ which is the discrete convolution of $\vec{a}$ and $b\vec{b}_j$:

$$\vec{c_j} = \vec{a} * b\vec{b}_j$$

Vector $\vec{c_j}$ has $A + B - j + 1$ coefficients, and the associated polynomial is:

$$c_j(x) = f(x) \cdot bb_j(x)$$

Hence we have:

$$f(x)g(t-x) = \sum_{j=0}^{B} c_j(x)t^j$$

And if we call $\vec{d_j}$ the vector of coefficients corresponding to the primitive polynomial[2] $d_j(x)$ of polynomial $c_j(x)$:

$$\vec{d_j} = \begin{bmatrix} 0 \\ c_0 \\ \frac{c_1}{2} \\ \vdots \\ \frac{c_{A+B-j}}{A+B-j+1} \end{bmatrix}$$

Then we have:

$$\begin{aligned} \int_{bound_1}^{bound_2} f(x)g(t-x)dx &= \int_{bound_1}^{bound_2} \sum_{j=0}^{B} c_j(x)t^j dx \\ &= \sum_{j=0}^{B} t^j \int_{bound_1}^{bound_2} c_j(x)dx \\ &= \sum_{j=0}^{B} t^j \left[d_j(bound_2) - d_j(bound_1)\right] \end{aligned}$$

From this point, we need to distinguish cases depending on the nature of $bound_1$ and $bound_2$

A.4.5 Calculating $\int_{\gamma}^{\delta} f(x)g(t-x)dx$

This version is rather simple. Since one has the d_j polynomials from the a_i and b_j coefficients, one can write:

$$\int_{\gamma}^{\delta} f(x)g(t-x)dx = \sum_{j=0}^{B} t^j \left[d_j(\delta) - d_j(\gamma)\right]$$

A.4.6 Calculating $\int_{\gamma}^{t-\delta} f(x)g(t-x)dx$

This calculation is a little more tricky since one of the bounds reintroduces t into d_j. We have:

$$\int_{\gamma}^{t-\delta} f(x)g(t-x)dx = \sum_{j=0}^{B} t^j \left[d_j(t-\delta) - d_j(\gamma)\right]$$

Let us split the problem again and write:

$$R(t) = \sum_{j=0}^{B} t^j d_j(\gamma)$$

$$Q(t) = \sum_{j=0}^{B} t^j d_j(t-\delta)$$

[2]Since one can take any primitive polynomial, we choose the one with a null constant term.

Calculating $R(t)$ is rather straightforward. The main problem comes from $Q(t)$. Calculating $d_j(t-\delta)$ is similar to the previous calculation of $g(t-x)$. The indexes get more complicated but the operation is the same, for each d_j one obtains a family of $d_{j,k}$ polynomials. For simplicity, we will write $D_j = A + B - j + 1$ and[3]:

$$d_j(t-\delta) = \sum_{k=0}^{D_j} d_{j,k}(t-\delta)^k = \sum_{k=0}^{D_j} t^k dd_{j,k}(\delta)$$

So one finally has:

$$Q(t) = \sum_{j=0}^{B} \sum_{k=0}^{D_j} t^{j+k} dd_{j,k}(\delta)$$

This provides an incremental way of building Q's coefficients. One first builds the $q_0 \dots q_{A+B+1}$ coefficients by initializing them to $q_k = dd_{0,k}(\delta)$. Then at the second pass, j is incremented to one and one updates all $q_1 \dots q_{A+B+1}$ coefficients. q_0 is not changed anymore since j is equal to one. The process goes on, updating the higher order coefficients of Q until we reach $j = B$ for the last pass, which updates all $q_B \dots q_{A+B+1}$ coefficients.

Finally, the result is found with the difference between Q and R:

$$\int_{\gamma}^{t-\delta} f(x)g(t-x)dx = Q(t) - R(t)$$

It is interesting to note that, by substituting x by $t - y$ one can write:

$$\int_{\gamma}^{t-\delta} f(x)g(t-x)dx = \int_{\delta}^{t-\gamma} g(y)f(t-y)dy$$

But the bb_j, c_j, d_j, Q and R polynomials yielded by the left hand side and the right hand side terms of this equation are completely different in nature. For example, there are B c_j and d_j polynomials respectively for the left hand side calculation, while the right hand side provides A polynomials for c_j and d_j. Similarly, the Q and R are different but since the degree of Q is necessarily higher than the degree of R, the highest order coefficients of Q must be the same in both cases. This remark does not provide any implementation improvement (except for debugging) but illustrates an interesting property of these polynomial's operations by establishing an equality between the $Q - R$ difference in the two cases.

A.4.7 Calculating $\int_{t-\gamma}^{t-\delta} f(x)g(t-x)dx$

For this last integral, one can remark that the variable replacement $y = t - x$ yields:

$$\int_{t-\gamma}^{t-\delta} f(x)g(t-x)dx = -\int_{\gamma}^{\delta} f(t-y)g(y)dy$$

If $t-\gamma$ and $t-\delta$ were actually ordered with $t-\gamma < t-\delta$, then $\delta < \gamma$ and it makes sense to permute the bounds and to get rid of the minus sign:

$$\int_{t-\gamma}^{t-\delta} f(x)g(t-x)dx = \int_{\delta}^{\gamma} g(y)f(t-y)dy$$

Which takes us back to the case of constant bounds.

[3]The reader interested in comparing with the POLYTOOLS-0.3 implementation will find that $d_j(t-\delta)$ has been renamed to $p_j(t)$. All other variables are named consistently with the variables used in these paragraphs.

A.5 Common difficulties

The programmer trying to implement an efficient version of a formal calculus library often encounters a series of problems which are mainly due to the very difference in nature between the abstract objects considered by the mathematical reasoning and the physical representations of numbers on a computer.

The excellent book of [Press et al., 2007] provides some hindsight on these problems. We will simply illustrate it with two distinct examples.

A.5.1 The case of Sturm's theorem

The application of Sturm's theorem to find the number of real roots of a polynomial in a given interval implies building Sturm's sequence. This sequence is obtained by successive division of polynomials. However, during this division operation, due to numerical limited precision, some coefficients which should become zero take very small values. At the next division, they yield other coefficients with very large values, and so on. This become problematic when the coefficients in question are the ones of highest degree.

An easy way to go around this problem, as mentioned in introduction, is to say that a polynomial function f is given by its $n+1$ coefficients a_i and its *span* M. M is the absolute value of the largest scalar which will ever be given to the polynomial for evaluation; it is an upper bound on the absolute values in the definition domain. Then, one can say that whenever $a_i \cdot M^i$ is below a certain threshold ϵ, a_i is supposed equal to zero.

While this is fine for simple evaluation of polynomials, it can completely change the result of Sturm's theorem. Setting arbitrarily some coefficients at zero can change Sturm's sequence or can shift the values obtained for the number of bounds inside a given interval.

Our experience with such problems lead us to results stating that some polynomials had -2 roots in a given interval. This first example illustrates one of the *caveat* one should keep in mind while working with formal coefficient calculus. Future versions of *POLYTOOLS* aim at improving the treatment of such problems.

A.5.2 Finding extrema

Finding the maximum of a polynomial corresponds to finding the corresponding root in its derivative. This is particularly important whenever one wishes to calculate the $g(t) = \sup_{t' \geq t} f(t')$ function in the piecewise polynomial case.

The variable precision schemes of computer languages can result in some inconsistencies here. Without entering too much into the details, some operations *require* some successive "`==0.`" tests (for details, see the above function in *POLYTOOLS*) which, depending on the way they are implemented, can sometimes return `true` or `false` while they should always return `true`.

Such inaccuracies result in completely erroneous output polynomials and can completely change the face a result. The conclusion of this short *caveat* paragraph is that one should always be very careful with both the implementation of such functions and the use of the provided tools in *POLYTOOLS* .

B

Short reminder of Support Vector Regression

This appendix presents a quick review of the notions at stake in Support Vector Regression (SVR). It provides a entry point for basic theory and properties of SVR methods and of some statistical learning methods used throughout the thesis. Instead of inserting them inside the thesis, we chose to make further references to this appendix or to their respective authors in order to ease the document's reading. Our goal here is not to provide an extensive and rigorous presentation of SVR techniques. For this purpose, we refer the reader to the quoted authors who generally provide excellent comprehensive textbooks or references. Hence, this appendix should only be seen as a quick reminder as well as an introductory basis for more general questions about kernel methods for regression, classification and density estimation which we won't develop here.

B.1 Least-Squares Linear Regression

We first recall the idea of least squares linear regression:

Suppose we have a set of l samples $\{(x_i, y_i)\}_{i=1..l}$, with $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$. We wish to find the hyperplane of $\mathbb{R}^{n+1}$ which fits best the $\{(x_i, y_i)\}_{i=1..l}$ set given a penalty term given as the Euclidean distance between the hyperplane and the points.

If a is the hyperplane's orthogonal unit vector, then the distance between a point x and the hyperplane is given as $x^T a$, so our goal is to minimize the quantity $\sum_{i=1}^{l}(x_i^T a - y_i)^2$.

If one writes X the matrix of all x_{ij} coordinates and y the vector of all y_i, then this problem turns to a quadratic programming problem which has solution:

$$a = (X^T X)^{-1} X^T y$$

B.2 ϵ-insensitive Support Vector Regression

The idea of SVR can be presented in a very similar concise manner. Suppose we have a set of l samples $\{(x_i, y_i)\}_{i=1..l}$, with $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$. We wish to find a function interpolating these points. Linear regression would provide us with the best interpolating hyperplane in $\mathbb{R}^n$, but the samples do not necessarily follow a linear evolution. Projecting the samples into a feature space is costly since feature spaces have higher dimension than n and because we need to chose the features themselves and the dimension of the feature space. However, if we were able to project our samples in a very large or infinite dimension feature space, then

the linear regression method would be very appropriate.

Thus, the idea of SVR is to find an interpolating function having expression:

$$f(x) = \langle w|\phi(x)\rangle + b \tag{B.1}$$

Where $\phi(x)$ is the projection of x into feature space, w is a vector of weights and b is the linear regression offset.

Then, the standard SVR problem is expressed as a compromise between *flatness* of the regressed function and the regression error. Flatness can be interpreted as insensitivity to noise and is measured by the norm of the w vector. In classical ϵ-sensitive SVR, the regression error is a non-linear soft margin term. This error is equal to zero if $|y_i - f(x_i)| < \epsilon$. This corresponds to the fact that we consider null the regression errors as long as the y_i points are within a tube of radius ϵ around the $f(x)$ curve in feature space. In order to formalize this ϵ-insensitivity, we can introduce slack variables ξ_i and ξ_i^*, quantifying the distance between the insensitivity tube and the points. ξ_i represents the distance "above' the tube and ξ_i^* the distance "below" as presented on figure B.1.

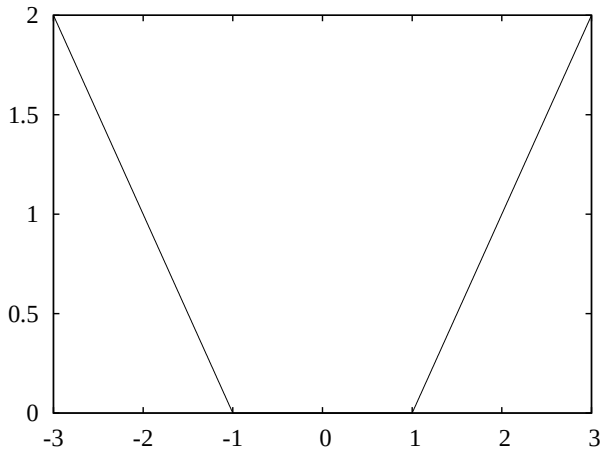


Figure B.1: Soft margin cost

Hence, the SVR problem can be expressed by using the trade-off constant C and the slack variables ξ_i and ξ_i^*:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{l}(\xi_i + \xi_i^*) \\ \text{subject to} & \left\{\begin{array}{lcl} y_i - \langle w, \phi(x_i)\rangle - b & \leq & \epsilon + \xi_i \\ \langle w, \phi(x_i)\rangle + b - y_i & \leq & \epsilon + \xi_i^* \\ \xi_i, \xi_i^* & \geq & 0 \end{array}\right. \end{array} \tag{B.2}$$

This non-linear optimization problem can be solved using its dual formulation. For this

purpose, we form the corresponding Lagrange function:

$$\begin{aligned} L =& \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{l}(\xi_i + \xi_i^*) - \sum_{i=1}^{l}\alpha_i\left(\epsilon + \xi_i - y_i + \langle w, \phi(x_i)\rangle + b\right) \\ & - \sum_{i=1}^{l}\alpha_i^*\left(\epsilon + \xi_i^* + y_i - \langle w, \phi(x_i)\rangle - b\right) \\ & - \sum_{i=1}^{l}\eta_i\xi_i + \eta_i^*\xi_i^* \end{aligned} \tag{B.3}$$

The optimal solution of the problem stated in equation B.2 is a saddle point of the above Lagrange function with respect to the primal and dual variables. Consequently, the derivative of L with respect to the primal variables (w, b, ξ_i, ξ_i^*) is equal to zero at the optimality point:

$$\begin{aligned} \partial_b L = & \quad \sum_{i=1}^{l}(\alpha_i^* - \alpha_i) & = 0 \\ \partial_w L = & \quad w - \sum_{i=1}^{l}(\alpha_i - \alpha_i^*)\,\phi(x) & = 0 \\ \partial_{\xi_i^{(*)}} L = & \quad C - \alpha_i^{(*)} - \eta_i^{(*)} & = 0 \end{aligned}$$

By replacing in equation B.3, one obtains the dual optimization problem:

$$\begin{aligned} &\text{maximize} \quad -\tfrac{1}{2}\sum_{i,j=1}^{l}(\alpha_i - \alpha_i^*)(\alpha_i - \alpha_i^*)\langle\phi(x_i), \phi(x_j)\rangle - \epsilon\sum_{i=1}^{l}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{l} y_i(\alpha_i - \alpha_i^*) \\ &\text{subject to} \quad \begin{cases} \sum_{i=1}^{l}(\alpha_i - \alpha_i^*) = 0 \\ \alpha_i, \alpha_i^* \in [0, C] \end{cases} \end{aligned} \tag{B.4}$$

So if we are able to express the dot product in feature space, directly as a function of x_i and x_j, we do not need to explicitly define the feature projection ϕ. In other words, if we can find a function $k(x_i, x_j)$ such that $k(x_i, x_j) = \langle\phi(x_i), \phi(x_j)\rangle$ then the problem can be written:

$$\begin{aligned} &\text{maximize} \quad -\tfrac{1}{2}\sum_{i,j=1}^{l}(\alpha_i - \alpha_i^*)(\alpha_i - \alpha_i^*)\,k(x_i, x_j) - \epsilon\sum_{i=1}^{l}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{l} y_i(\alpha_i - \alpha_i^*) \\ &\text{subject to} \quad \begin{cases} \sum_{i=1}^{l}(\alpha_i - \alpha_i^*) = 0 \\ \alpha_i, \alpha_i^* \in [0, C] \end{cases} \end{aligned} \tag{B.5}$$

And finally:

$$f(x) = \sum_{i=1}^{l}(\alpha_i - \alpha_i^*)\,k(x_i, x) + b \tag{B.6}$$

If one writes the Karush-Kuhn-Tucker conditions:

$$\begin{aligned} \alpha_i\left(\epsilon + \xi_i - y_i + \langle w, x_i\rangle + b\right) = 0 \\ \alpha_i^*\left(\epsilon + \xi_i^* - y_i + \langle w, x_i\rangle + b\right) = 0 \end{aligned}$$

It appears that for all points *inside* the ϵ-insensitivity tube, the α_i, α_i^* vanish. Because of the same equations, one also has $\alpha_i\alpha_i^* = 0$ for all l samples. Thus, only a few α_i are non-zero and the final solution is *sparse*.

The points corresponding to non-zero $\alpha_i^{(*)}$ are called *Support Vectors*, they are the only ones participating in the optimal solution.

B.3 Variations on the theme of kernel-based regression

Most of the effort in SVR has been dedicated to designing efficient kernels in order to represent expressive feature spaces. To cite only a few, one can mention:

- the polynomial kernel $k(x_i, x_j) = (\langle x_i, x_j \rangle + c)^p$, used for instance in optical character recognition,
- the sigmoid kernel $k(x_i, x_j) = \tanh(c + d\langle x_i, x_j \rangle)$, similar to the activation function of common neural networks,
- the Gaussian kernel $k(x_i, x_j) = e^{-\frac{\|x_1 - x_2\|^2}{2\sigma^2}}$, which is quite widespread and has the important property of being translation invariant.

Variations on SVR formulation yield the Least-Squares SVR formulation which reduces the insensitivity tube to a null width and minimizes an L_2 error term in the objective function (instead of the L_1 term expressed in terms of slack variables in equation B.2). This turns the resolution into a linear problem. However, the solution of LS-SVR is not very sparse. Generally speaking, *sparsity* comes from the norm used for w and from the *loss functions* used. These loss functions express the weight we put on *outlier* points, *ie.* points that do not fit well our regression. The linear ϵ-sensitive formulation is the loss function of the standard SVR presented above, LS-SVR use L_2 loss functions. Other functions corresponding to other expected densities of samples (noise) have been explored such as Gaussian, Laplacian or Huber's robust loss functions.

An interesting alternative to SVR in kernel-based regression is the LASSO formulation which yields interestingly sparse representations. This formulation is based on an L_1 regularization term ($\|w\|_1$) and an L_2 loss function.

List of Figures

List of Algorithms

Bibliography

Ahlberg, J. H., Nielson, E. N., and Walsh, J. L. (1967). *The Theory of Spline Functions and Their Applications*. Academic Press, New York.

Ahn, M. S. and Kim, T. G. (1993). Analysis on Steady State Behavior of DEVS Models. In *International Conference on AI, Simulation and Planning in High Autonomy Systems*.

Altman, E. (1999). *Constrained Markov Decision Processes*. Chapman & Hall/CRC, London.

Altman, E. and Shwartz, A. (1993). Time-Sharing Policies for Controlled Markov Chains. *Operations Research*, 41(6):1116–1124.

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

Anderson, C. W. (2000). Approximating a Policy can be easier than Approximating a Value Function. Technical Report TR-CS-00-101, Colorado State University.

Andre, D., Friedman, N., and Parr, R. (1998). Generalized Prioritized Sweeping. In *Neural Information Processing Systems*, pages 1001–1007.

Antos, A., Munos, R., and Szepesvari, C. (2007). Fitted Q-iteration in continuous action-space MDPs. In *Neural Information Processing Systems*.

Atkeson, C., Moore, A., and Schaal, S. (1997). Locally Weighted Learning. *Artificial Intelligence*, 11(4):76–113.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning Journal*, 47(2–3):235–256.

Barros, F. J. (1997). Modelling Formalisms for Dynamic Structure Systems. *ACM Transactions on Modelling and Computer Simulation*, 7:501–515.

Barto, A. G., Bradtke, S. J., and Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138.

Baxter, J. and Bartlett, P. (1999). Direct gradient-based reinforcement learning: I. Gradient estimation algorithms. Technical report, Computer Science Laboratory, Australian National University.

Bellman, R. E. (1954). The Theory of Dynamic Programming. *Bulletin of the American Mathematical Society*, 60:503–516.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press, Princeton, New Jersey.

Benazera, E., Brafman, R., Meuleau, N., Mausam, and Hansen, E. A. (2005). An AO* Algorithm for Planning with Continuous Resources. In *Workshop on Planning under Uncertainty for Autonomous Systems, at ICAPS*.

Bernstein, D. S. and Zilberstein, S. (2001). Reinforcement Learning for Weakly-Coupled MDPs and an Application to Planetary Rover Control. In *Conference on Uncertainty in Artificial Intelligence*.

Bertsekas, D. P. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific.

Bertsekas, D. P. and Shreve, S. E. (1996). *Stochastic Optimal Control: The Discrete-Time Case*. Athena Scientific. Originally published in 1978.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bonet, B. and Geffner, H. (2003a). Faster Heuristic Search Algorithms for Planning with Uncertainty and Full Feedback. In *International Joint Conference on Artificial Intelligence*.

Bonet, B. and Geffner, H. (2003b). Labeled RTDP: Improving the convergence of real-time dynamic programming. In *International Conference on Automated Planning and Scheduling*, pages 12–21.

Boutilier, C., Dean, T., and Hanks, S. (1999). Decision-theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research*, 11:1–94.

Boutilier, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic Dynamic Programming with Factored Representations. *Artificial Intelligence*, 121(1–2):49–107.

Bouyer, P., Cassez, F., Fleury, E., and Larsen, K. G. (2004). Optimal Strategies in Priced Game Automata. In *Foundations of Software Technology and Theoretical Computer Science*.

Boyan, J. A. and Littman, M. L. (2001). Exact Solutions to Time Dependent MDPs. *Advances in Neural Information Processing Systems*, 13:1026–1032.

Bradtke, S. J. and Barto, A. G. (1996). Linear Least-Squares Algorithms for Temporal Difference Learning. *Machine Learning*, 22(2):33–57.

Bresina, J., Dearden, R., Meuleau, N., Ramakrishnan, S., and Washington, R. (2002). Planning under Continuous Time and Resource Uncertainty: a Challenge for AI. In *Conference on Uncertainty in Artificial Intelligence*.

Cappé, O., Moulines, E., and Rydén, T. (2005). *Inference in Hidden Markov Models*. Springer.

Chang, C.-C. and Lin, C.-J. (2001). *LIBSVM: a library for support vector machines*. Software available at `http://www.csie.ntu.edu./tw/~cjlin/libsvm`.

Chang, H. S., Fu, M. C., Hu, J., and Marcus, S. I. (2007). *Simulation-based Algorithms for Markov Decision Processes*. Communications and Control Engineering. Springer-Verlag London.

Chen, T., Morris, J., and Martin, E. (2006). Probability Density Estimation via Infinite Gaussian Mixture Model: Application to Statistical Process Monitoring. *Journal of the Royal Statistical Society (series C)*, 55(1):699–715.

Coquelin, P.-A. and Munos, R. (2007). Bandit Algorithm for Tree Search. In *Conference on Uncertainty in Artificial Intelligence*.

Cox, D. R. and Miller, H. D. (1965). *The Theory of Stochastic Processes*. John Wiley & Sons, Inc.

Cushing, W., Kambhampati, S., Mausam, and Weld, D. S. (2007). When is Temporal Planning Really Temporal? In *International Conference on Automated Planning and Scheduling*.

Dai, P. and Goldsmith, J. (2007). Multi-Threaded BLAO* Algorithm. In *FLAIRS Conference*, pages 56–61.

Dean, T. L. and Kanazawa, K. (1990). A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150.

Dean, T. L. and Lin, S.-H. (1995). Decomposition Techniques for Planning in Stochastic Domains. In *International Joint Conference on Artificial Intelligence*.

Dearden, R. (2001). Structured Prioritized Sweeping. In *International Conference on Machine learning*.

Dedecker, J. (2008). Inégalités de Hoeffding et théorème limite central pour les fonctions peu régulières de chaînes de markov non irréductibles. *Numéro spécial des Annales de l'ISUP*, 52:39–46.

d'Epenoux, F. (1963). A Probabilistic Production and Inventory System. *Management Science*, 10(1):98–108.

Dimitrikakis, C. and Lagoudakis, M. (2008). Algorithms and Bounds for Sampling-based Approximate Policy Iteration. In *European Workshop on Reinforcement Learning*.

Ernst, D., Geurts, P., and Wehenkel, L. (2005). Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6:503–556.

Farahmand, A., Ghavamzadeh, M., Szepesvári, C., and Mannor, S. (2008). Regularized Policy Iteration. In *Neural Information Processing Systems*.

Feng, Z., Dearden, R., Meuleau, N., and Washington, R. (2004). Dynamic Programming for Structured Continuous Markov Decision Problems. In *Conference on Uncertainty in Artificial Intelligence*.

Ferguson, D. and Stentz, A. (2004). Focussed Dynamic Programming: Extensive Comparative Results. Technical Report CMU-RI-TR-04-13, Robotics Insitute, Carnegie Mellon University.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Gilmer Jr., J. B. and Sullivan, F. J. (2005). Issues in Event Analysis for Recursive Simulation. In *Winter Simulation Conference*.

Glynn, P. (1989). A GSMP Formalism for Discrete Event Systems. *Proc. of the IEEE*, 77.

Guestrin, C., Hauskrecht, M., and Kveton, B. (2004). Solving Factored MDPs with Continuous and Discrete Variables. In *Conference on Uncertainty in Artificial Intelligence*.

Guestrin, C., Koller, D., and Parr, R. (2001). Max-norm Projections for Factored MDPs. In *International Joint Conference on Artificial Intelligence*, pages 673–682.

Hansen, E. A. and Zilberstein, S. (2001). LAO*: a heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2).

Hasselt, H. and Wiering, M. A. (2007). Reinforcement Learning in Continuous Action Spaces. In *IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*.

Hauskrecht, M. and Kveton, B. (2004). Linear program approximations for factored continuous-state Markov decision processes. *Advances in Neural Information Processing Systems*, 16:895–902.

Hauskrecht, M. and Kveton, B. (2006). Approximate Linear Programming for Solving Hybrid Factored MDPs. In *International Symposium on Artificial Intelligence and Mathematics*.

Hauskrecht, M., Meuleau, N., Kaelbling, L. P., Dean, T. L., and Boutilier, C. (1998). Hierarchical Solution of Markov Decision Processes using Macro-actions. In *Conference on Uncertainty in Artificial Intelligence*, pages 220–229.

Hoey, J., St. Aubin, R., Hu, A., and Boutilier, C. (2000). Optimal and Approximate Stochastic Planning using Decision Diagrams. Technical Report TR-2000-05, University of British Columbia - Vancouver, BC, Canada.

Howard, R. A. (1963). Semi-Markovian Decision Processes. In *34th Session of the International Statistical Institute*.

Joslyn, C. (1996). The Process Theoretical Approach to Qualitative DEVS. In *International Conference on AI, Simulation and Planning in High Autonomy Systems*.

Kaelbling, L. P. (1990). *Learning In Embedded Systems*. PhD thesis, Stanford University, Department of Computer Science.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101:99–134.

Kearns, M. J., Mansour, Y., and Ng, A. Y. (2002). A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes. *Machine Learning*, 49:193–208.

Kocsis, L. and Szepesvari, C. (2006). Bandit Based Monte-Carlo Planning. In *European Conference on Machine Learning*.

Korf, R. E. (1990). Real-Time Heuristic Search. *Artificial Intelligence*, 42:189–211.

Kveton, B. and Hauskrecht, M. (2006). Learning Basis Functions in Hybrid Domains. In *AAAI Conference on Artificial Intelligence*.

Lagoudakis, M. and Parr, R. (2003). Least-Squares Policy Iteration. *Journal of Machine Learning Research*, 4:1107–1149.

Li, L. and Littman, M. L. (2005). Lazy Approximation for Solving Continuous Finite-Horizon MDPs. In *National Conference on Artificial Intelligence*.

Littman, M. L., Dean, T. L., and Kaelbling, L. P. (1995). On the Complexity of Solving Markov Decision Problems. In *Conference on Uncertainty in Artificial Intelligence*, volume 11, pages 394–402.

Liu, Y. and Koenig, S. (2006). Functional Value Iteration for Decision-Theoretic Planning with General Utility Functions. In *National Conference on Artificial Intelligence*.

Marecki, J., Topol, Z., and Tambe, M. (2006). A Fast Analytical Algorithm for Markov Decision Process with Continuous State Spaces. In *International Conference on Autonomous Agents and Multi-Agent Systems*, pages 2536–2541.

Mausam (2007). *Stochastic Planning with Concurrent, Durative Actions*. PhD thesis, University of Washington.

Mausam, Benazera, E., Brafman, R., Meuleau, N., and Hansen, E. A. (2005). Planning with continuous resources in stochastic domains. In *International Joint Conference on Artificial Intelligence*, pages 1244–1251.

Mausam and Weld, D. S. (2005). Concurrent Probabilistic Temporal Planning. In *International Conference on Automated Planning and Scheduling*.

Mausam and Weld, D. S. (2006). Probabilistic Temporal Planning with Uncertain Durations. In *National Conference on Articifial Intelligence*.

Mausam and Weld, D. S. (2007). Planning with Durative Actions in Stochastic Domains. *Journal of Artificial Intelligence Research*, 31:33–82.

Maxwell, M. and Woodroofe, M. (2000). Central Limit Theorems for Additive Functionals of Markov Chains. *Annals of Probability*, 28(2):713–724.

McMahan, H. B., Likhachev, M., and Gordon, G. J. (2005). Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *International Conference on Machine learning*, pages 569–576.

Melamed, B. (1976). *Analysis and Simplification of Discrete Event Systems and Jackson Queuing Networks*. PhD thesis, University of Michigan.

Meuleau, N., Hauskrecht, M., Kim, K.-E., Peshkin, L., Kaelbling, L. P., Dean, T., and Boutilier, C. (1998). Solving Very Large Weakly Coupled Markov Decision Processes. In *AAAI Conference on Artificial Intelligence*.

Moore, A. W. and Atkeson, C. G. (1993). Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time. *Machine Learning Journal*, 13(1):103–105.

Munos, R. (2003). Error Bounds for Approximate Policy Iteration. In *International Conference on Machine Learning*.

Munos, R. (2007). Performance Bounds for Approximate Value Iteration. *SIAM Journal on Control and Optimization*, 46(2):541–561.

Munos, R. and Moore, A. W. (2000). Rates of Convergence for Variable Resolution Schemes in Optimal Control. In *International Conference on Machine Learning*.

Munos, R. and Moore, A. W. (2002). Variable Resolution Discretization in Optimal Control. *Machine Learning Journal*, 49(2-3):291–323.

Neuts, M. R. (1981). *Matrix-geometric solutions in stochastic models: an algorithmic approach*. The John Hopkins University Press, Baltimore.

Nielsen, F. (1998). GMSim: a tool for compositionnal GSMP modeling. In *Winter Simulation Conference*.

Ormoneit, D. and Sen, S. (2002). Kernel-Based Reinforcement Learning. *Machine Learning Journal*, 49:161–178.

Parr, R. (1998). Flexible Decomposition Algorithms for Weakly Coupled Markov Decision Problems. In *Conference on Uncertainty in Artificial Intelligence*.

Parzen, E. (1962). On the Estimation of a Probability Density Function and the Mode. *Annals of Mathematics and Statistics*, 33:1065–1076.

Peng, J. and Williams, R. J. (1993). Efficient Learning and Planning Within the Dyna Framework. *Adaptive Behaviour*, 1(4):437–454.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press.

Puterman, M. L. (1994). *Markov Decision Processes*. John Wiley & Sons, Inc.

Péret, L. (2004). *Recherche en ligne pour les Processus Décisionnels de Markov : application à la maintenance d'une constellation de satellites*. PhD thesis, Institut National Polytechnique de Toulouse.

Péret, L. and Garcia, F. (2003). On-line Search for Solving Large Markov Decision Processes. In *European Workshop on Reinforcement Learning*.

Péret, L. and Garcia, F. (2004). On-line Search for Solving Markov Decision Processes via Heuristic Sampling. In *European Conference on Artificial Intelligence*.

Quesnel, G., Duboz, R., Ramat, E., and Traore, M. K. (2007). VLE - A Multi-Modeling and Simulation Environment. In *Moving Towards the Unified Simulation Approach, Proc. of the 2007 Summer Simulation Conf.*, pages 367–374.

Rabiner, L. R. (1989). A tutorial on Hidden Markov Models and selected applications in speech recognition. In *Proceedings of the IEEE*, pages 257–286.

Rachelson, E., Fabiani, P., Farges, J.-L., Teichteil, F., and Garcia, F. (2006). Une approche du traitement du temps dans le cadre MDP : trois méthodes de découpage de la droite temporelle. In *Journées Françaises Planification Décision Apprentissage*. F. Garcia, G. Verfaillie editors.

Rachelson, E., Garcia, F., and Fabiani, P. (2008a). Extending the Bellman Equation for MDP to Continuous Actions and Continuous Time in the Discounted Case. In *International Symposium on Artificial Intelligence and Mathematics*.

Rachelson, E., Quesnel, G., Garcia, F., and Fabiani, P. (2008b). A Simulation-based Approach for Solving Generalized Semi-Markov Decision Processes. In *European Conference on Artificial Intelligence*.

Rachelson, E., Quesnel, G., Garcia, F., and Fabiani, P. (2008c). Approximate Policy Iteration for Generalized Semi-Markov Decision Processes: an Improved Algorithm. In *European Workshop on Reinforcement Learning*.

Roth, V. (2004). The Generalized LASSO. *IEEE Transations on Neural Networks*, 15(1).

Sabbadin, R. (2002). Graph partitioning techniques for Markov Decision Processes decomposition. In *European Conference on Artificial Intelligence*, pages 670–674.

Schölkopf, B., Platt, J. C., Shawe-Taylor, J., Smola, A., and Williamson, R. (2001). Estimating the Support of a High-Dimensional Distribution. *Neural Computation*, 13(1):1443–1471.

Smith, T. and Simmons, R. G. (2006). Focused Real-Time Dynamic Programming for MDPs: Squeezing more out of a Heuristic. In *AAAI Conference on Artificial Intelligence*.

Smola, A. and Schölkopf, B. (1998). A Tutorial on Support Vector Regression. Technical Report NC-TR-98-030, Royal Holloway College, University of London, NeuroCOLT Technical Report.

Sturm, C. (1835). *Mémoire sur la résolution des équations numériques*. Ins. France Sc. Math. Phys., t. 6.

Sutton, R. S. (1995). TD Models: Modeling the World at a Mixture of Time Scales. In *International Conference on Machine Learning*, pages 531–539.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA.

Teichteil-Königsbuch, F. and Infantes, G. (2008). Tr-FSP: Forward Stochastic Planning using Probabilistic Reachability. In *International Symposium on Search Techniques in Artificial Intelligence and Robotics*.

Tesauro, G. and Galerpin, G. R. (1997). On-line Policy Improvement using Monte-Carlo Search. *Advances in Neural Information Processing Systems*, pages 1068–1072.

Tibshirani, R. (1996). Regression Shrinkage and Selection via the Lasso. *Journal of the royal Statistical Society: series B*, 58(1):267–288.

Vapnik, V., Golowich, S., and Smola, A. (1996). Support Vector Method for Function Approximation, Regression Estimation and Signal Processing. *Advances in Neural Information Processing Systems*, 9:281–287.

Vijayakumar, S., D'Souza, A., and Schaal, S. (2005). Incremental Online Learning in High Dimensions. *Neural Computation*, 17:2602–2634.

Wang, G., Yeung, D.-Y., and Lochovsky, F. H. (2007). The Kernel Path in Kernelized LASSO. In *AISTATS*.

Watkins, C. J. C. (1989). *Learning from Delayed Rewards*. PhD thesis, Cambridge University.

Watkins, C. J. C. and Dayan, P. (1992). Q-learning. *Machine Learning*.

Wellman, M., Ford, M., and Larson, K. (1995). Path Planning under Time-Dependent Uncertainty. In *Conference on Uncertainty in Artificial Intelligence*, pages 532–539.

Whiteson, S. and Stone, P. (2006). Evolutionary Function Approximation for Reinforcement Learning. *Journal of Machine Learning Research*, 7:877–917.

Williams, R. J. and Baird, L. C. (1993). Tight Performance Bounds on Greedy Policies Based on Imperfect Value Functions. Technical Report NU-CCS-93-14, College of Computer Science, Northeastern University, Boston, Massachussetts.

Younes, H. L. S. and Simmons, R. G. (2004). Solving Generalized Semi-Markov Decision Processes using Continuous Phase-Type Distributions. In *AAAI Conference on Artificial Intelligence*.

Zeigler, B. P. (1976). *Theory of Modeling and Simulation*. Wiley Interscience.

Zeigler, B. P., Kim, D., and Praehofer, H. (2000). *Theory of modeling and simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press.