

Université Pierre et Marie Curie - Paris 6

PhD Thesis

Department of Computer Science
Institut des Systèmes Intelligents et de Robotique

Hierarchical & Factored Reinforcement Learning

by

Olga KOZLOVA

Committee

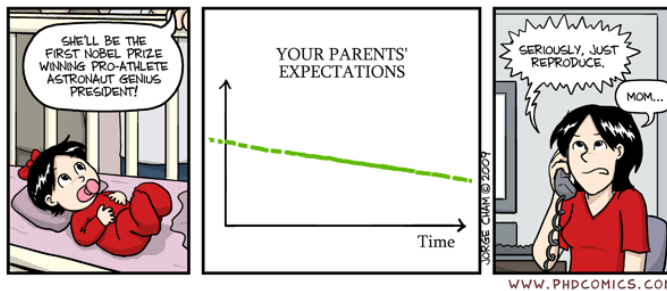
Philippe Bidaud (Chair)	Professor, Université Pierre et Marie Curie, Paris
Alain Dutech (Reporter)	Research fellow INRIA, Vandoeuvre les Nancy
Christophe Meyer (Co-Supervisor)	PhD, THALES - Defence & Security , Palaiseau
Abdel-Ilah Mouaddib (Reporter)	Professor, Université de Caen Basse-Normandie, Caen
Rémi Munos (Examiner)	Senior Researcher, INRIA, Lille
Olivier Sigaud (Supervisor)	Professor, Université Pierre et Marie Curie, Paris

Paris, May 2010

ISIR
Institut des Systèmes Intelligents et de Robotique
Université Pierre et Marie CURIE
Pyramide - T55/65
CC 173 - 4 Place Jussieu
75005 Paris

Thales D3S/Simulation
1, rue Général de Gaulle - Osny
BP 226
95523 Cergy Pontoise Cedex

*To my parents,
Elena & Alexey Kozlov*



Abstract MadLibs!!

This paper presents a _____ method for _____
(synonym for new) (sciencey verb)
the _____. Using _____, the
(noun few people have heard of) (something you didn't invent)
_____ was measured to be _____ +/- _____
(property) (number) (number)
_____. Results show _____ agreement with
(units) (sexy adjective)
theoretical predictions and significant improvement over
previous efforts by _____, et al. The work presented
(Loser)
here has profound implications for future studies of
_____ and may one day help solve the problem of
(buzzword)

(supreme sociological concern)

Keywords: _____, _____, _____
(buzzword) (buzzword) (buzzword)

Abstract

This thesis is accomplished in the context of the industrial simulation domain that addresses the problems of modelling of human behavior in military training and civil security simulations.

The aim of this work is to solve large stochastic and sequential decision making problems in the Markov Decision Process (MDP) framework using Reinforcement Learning methods for learning and planning under uncertainty.

The Factored Markov Decision Process (FMDP) framework is a standard representation for sequential decision problems under uncertainty where the state is represented as a collection of random variables. Factored Reinforcement Learning (FRL) is an Model-based Reinforcement Learning approach to FMDPs where the transition and reward functions of the problem are learned under a factored form. As a first contribution of this thesis, we show how to model in a theoretically well-founded way the problems where some combinations of state variable values may not occur, giving rise to what we call *impossible states*. Furthermore, we propose a new heuristics that considers as impossible the states that have not been seen so far. We derive an algorithm whose improvement in performance with respect to the standard approach is illustrated through benchmark experiments on MAZE6 and BLOCKS WORLD problems.

Besides, following the example of FMDPs, a Hierarchical MDP (HMDP) is based on the idea of factorization, but brings that idea on a new level. From *state factorization* of FMDPs, HMDP can make profit of *task factorization*, where a set of similar situations (defined by their goals) are represented by a partially defined set of independent subtasks. In other words, it is possible to simplify a problem by splitting it into smaller problems that are easier to solve individually, but also reuse the subtasks in order to speed up the global search of a solution. This kind of architecture can be efficiently represented using the options framework by including temporally extended courses of actions.

The second contribution of this thesis introduces TeXDYNA, an algorithm designed to solve large MDPs with unknown structure by integrating hierarchical abstraction techniques of Hierarchical Reinforcement Learning (HRL) and factorization techniques

of FRL. TeXDYNA performs incremental hierarchical decomposition of the FMDP, based on the automatic discovery of subtasks directly from the internal structure of the problem.

We evaluate TeXDYNA on two benchmark problems, namely TAXI and LIGHT BOX, and we show that combining contextual information abstraction through the FMDP framework and hierarchy building through the HMDP framework results in very efficient compaction of the structures to be learned, faster computation and improved convergence speed. Furthermore, we appraise the potential and limitations of TeXDYNA on a toy application more representative of the industrial simulation domain.

Keywords: factored markov decision processes, reinforcement learning, options, hierarchical decomposition, structured dynamic programming, impossible states

Acknowledgments

I thank all those who supported me,
and also those who didn't.

Contents

Abstract	5
Acknowledgments	7
Contents	9
List of Figures	13
List of Tables	17
List of Algorithms	19
1 Introduction	21
1.1 The industrial simulation domain	22
1.2 Reinforcement Learning	24
1.3 Contributions	26
1.4 Outline	27
I Background	29
2 Markov Decision Processes	31
2.1 Definitions	32
2.2 Algorithms	34
2.2.1 Policy Iteration	34
2.2.2 Value Iteration	35
2.3 Discussion & Summary	36
3 Reinforcement Learning	37
3.1 Algorithms	37
3.1.1 TD - LEARNING	38
3.1.2 Q-LEARNING & SARSA	38

3.1.3	Actor-Critic	39
3.1.4	DYNA-Q	39
3.1.5	Prioritized sweeping	41
3.2	Discussion & Summary	42
4	Factored Reinforcement Learning	43
4.1	FMDPs	43
4.2	Structured Dynamic Programming	45
4.2.1	Tree manipulations	46
4.2.2	Decision-Theoretic Regression	47
4.2.3	SPUDD	52
4.3	Factored Reinforcement Learning algorithms	52
4.3.1	SDYNA and SPITI	52
4.3.2	Learning Factored Model	53
4.3.3	Incremental Planning	54
4.4	Discussion & Summary	55
5	Hierarchical Reinforcement Learning	57
5.1	HMDPs	58
5.1.1	SMDPs & the Options framework	59
5.1.2	Hierarchical Reinforcement Learning algorithms	60
5.1.3	Discovering the hierarchy	61
5.2	Combining HRL and FRL	61
5.2.1	HEXQ	62
5.2.2	VISA	62
5.2.3	Incremental-VISA	63
5.3	Discussion & Summary	64
II	Contribution	65
6	Dealing with Impossible States: IMPSPITI	67
6.1	Impossible states in FMDPs	68
6.2	The BLOCKS WORLD problem	69
6.3	Modeling impossible states	71
6.3.1	Impact on regression	72
6.3.2	Impact on other tree operations	75
6.3.3	IMPSVI	76
6.4	IMPSPITI	76
6.5	Experimental study	78
6.5.1	The MAZE6 problem	78
6.5.2	Performance and convergence speed of IMPSVI	79
6.5.3	Performance and convergence speed of IMPSPITI	80

6.5.4	Comparison with tabular algorithms	84
6.6	Discussion	87
6.6.1	Limitations	88
6.6.2	Extensions: Structured Prioritized Sweeping	88
6.7	Conclusion	90
7	TeXDYNA : hierarchical decomposition in FRL	93
7.1	The LIGHT BOX problem	95
7.2	TeXDYNA: global view	96
7.3	Learning: Discovering options	96
7.3.1	Introducing options	98
7.3.2	Building the hierarchy of options	101
7.3.3	Incremental update	104
7.4	Planning: FRL over options	105
7.4.1	SPITI with options	106
7.4.2	Acting: choosing options	106
7.4.3	Hierarchical policy	108
7.5	Experimental study	109
7.5.1	The TAXI problem	110
7.5.2	Performance and convergence speed	111
7.5.3	Transition function representation	116
7.5.4	Localization of the transition function	117
7.5.5	State space exploration	119
7.6	Discussion	120
7.6.1	Related work: Incremental-VISA	121
7.6.2	Limitations	121
7.7	Conclusion	122
8	Applications	125
8.1	Application problem definition	126
8.2	TeXDYNA and IMPTeXDYNA implementation	128
8.3	Experimental results	129
8.3.1	Transition function	130
8.3.2	Hierarchical policy	131
8.3.3	Performance	134
8.3.4	Adaptation to changing environment	135
8.3.5	Problem size scaling	136
8.4	Discussion	137
8.4.1	Contributions	137
8.4.2	Limitations	138
8.5	Conclusion	138

9 Discussion & Conclusion	139
9.1 Learning the problem structure	140
9.2 IMPSVI, IMPSPITI & Impossible states	141
9.3 TeXDYNA	142
9.4 Application to the industrial simulation domain	142
9.5 Perspectives	143
Appendices	147
Résumé en français	147
Résumé du Résumé	147
Introduction	149
Simulation industrielle	149
Apprentissage par Renforcement	150
Contributions	151
IMPSVI, IMPSPITI & Gestion des états impossibles	152
TeXDYNA : Décomposition hiérarchique dans FRL.	154
Application au domaine de simulation	157
Discussion & Conclusion	158
IMPSVI, IMPSPITI & Etats impossibles	158
TeXDYNA	159
Application dans le domaine de simulation	160
Perspectives	160
Bibliography	161

List of Figures

2.1	Markov Decision Process.	32
2.2	Policy Evaluation/Improvement	35
3.1	The DYNA process.	40
4.1	Transition function representation within the FMDP framework. (A) Dynamic Bayesian Network representing the dependencies between the variables. (B) Tabular representation of the Conditional Probability Distribution. (C) Decision Tree representation of the Conditional Probability Distribution.	44
4.2	Reward function representation within the FMDP framework. (A) Dynamic Bayesian Network representing the dependencies of the reward from the variables. (B) Tabular representation of the reward values. (C) Decision Tree representation of the reward values.	45
4.3	A tree simplified by removal of incompatible test $V(x_1x_2\bar{x}_1) = 9$ and redundant nodes $V(\bar{x}x_2) = V(\bar{x}\bar{x}_2) = 6$	46
4.4	Appending trees using maximization as combination operator.	46
4.5	Merging trees $Tree[Q_a^V]$ to obtain $Tree[\pi]$ using maximization as combination operator and simplification of incompatible and redundant nodes.	47
4.6	A toy example: the tree representation of a reward function and of the transition functions of binary variables X_1 and X_2 given an unique action a_0 . Notations are similar to the ones in [Boutilier et al., 2000], but in the boolean case we note x_i for $X_i = true$ and \bar{x}_i for $X_i = false$	48
4.7	$P(X' X, a_0)$ computed by the <i>PRegress</i> operator from the example of Figure 4.6.	49
4.8	Regression with structured representations using the values computed in Table 4.1. We consider the first iteration where $Tree[V] = Tree[R]$. Notice that transitions to \bar{x}_1x_2 and $\bar{x}_1\bar{x}_2$ are summed to \bar{x}_1	49
4.9	SPITI algorithm.	53

5.1	Hierarchical MDP.	58
6.1	Example of synchronic arcs	69
6.2	A BLOCKS WORLD scenario, from a random initial position (a) to the goal position (d).	70
6.3	Variable combinations giving rise to impossible states in the BLOCKS WORLD problem.	71
6.4	DBN representation for problems with independent variables and impossible states. K and K' stand for constraints stating if a particular combination of values is possible. Variables in gray are observed. K' and relevant X_j are used to predict X'_i for all i . K' does not necessarily depend on all X'_i	72
6.5	Removing impossible state \bar{x}_1x_2 probabilities in $P(X' X, a_0)$ computed by the <i>Pregress</i> operator.	74
6.6	Combining two trees can generate leaves about impossible states (here, \bar{x}_1x_2 is impossible).	75
6.7	A global view of the IMPSPITI algorithm.	77
6.8	Visited states representation in a tree form. Here, states x_1x_2 , $x_1\bar{x}_2$ and $\bar{x}_1\bar{x}_2$ have been visited.	77
6.9	The MAZE6 problem.	79
6.10	Convergence on the MAZE6 problem in number of steps needed to perform an episode as a function of the number of episodes.	81
6.11	BLOCKS WORLD problem: value function size as a function of the size of the problem. Labels on points indicate this value / the total number of states considered. Note the log scale for <i>Binary</i>	82
6.12	BLOCKS WORLD problem (size 4-3-4): performance along episodes. We run 70 episodes with <i>Blocks</i> to wait for convergence.	83
6.13	The D-MAZE problem.	85
6.14	Convergence over episodes on MAZE6 with DYNA-Q and IMPSPITI.	85
6.15	Convergence over episodes on D-MAZE with DYNA-Q and IMPSPITI.	86
6.16	Action DBN, Conditional Probability Trees, Value and Reward function example.	89
6.17	Binary 2-2-2 BLOCKS WORLD problem: example of possible states.	89
6.18	Binary 2-2-2 BLOCKS WORLD problem: value tree and transition trees.	90
7.1	(a) The LIGHT BOX problem: number and color of “lights” with their dependencies. (b) Internal dependencies of the LIGHT BOX problem.	95
7.2	TeXDYNA global view.	97
7.3	Transition trees for the variables “16” and “11” in the LIGHT BOX problem.	99
7.4	Transition trees for the variable “16” and actions <i>toggle16</i> and <i>toggle2</i> in the LIGHT BOX problem.	100
7.5	Example of the options discovered in the LIGHT BOX problem.	103

7.6	Example of incomplete transition tree with erroneous dependencies (irrelevant variable “7”, missing variables “8” and “5”).	105
7.7	Local policy of the options <i>toggle16</i> and <i>toggle11</i>	109
7.8	The TAXI problem.	110
7.9	Example of options discovered in the TAXI problem. The option <i>PickUp</i> changes the value of variable <i>Passenger</i> from <i>No</i> (not in the taxi) to <i>Yes</i> (in the taxi). Its exit context contains 2 variables: <i>Taxi Location</i> and <i>Passenger Location</i> . It has 4 sub-options.	112
7.10	Convergence over episodes on the stochastic TAXI problem.	112
7.11	Convergence over episodes on the LIGHT BOX problem.	114
7.12	Policy size on the LIGHT BOX problem (IMPSPITI refers to the record of visited states).	115
7.13	Convergence over episodes on the TAXI problem using a transition function representation with one tree per variable per action - $Tree[P_a(x' s)]$ or one tree per variable where the actions are attributes - $Tree[P(x' s)]$	116
7.14	Convergence over episodes on the LIGHT BOX problem using transition function representation with one tree per variable per action - $Tree[P_a(x' s)]$ or one tree per variable where the actions are attributes - $Tree[P(x' s)]$	116
7.15	Convergence over episodes on the TAXI problem using local or global transition functions.	118
7.16	Number of visited states over episodes in the TAXI problem.	119
7.17	Convergence over episodes on the TAXI problem.	119
8.1	Terrorist vs. Guard scenario.	126
8.2	Terrorist(T) vs. Guard(G) scenario. Schematic representation.	128
8.3	Terrorist vs. Guard scenario. Transition function for variable “FIRE”.	130
8.4	Terrorist vs. Guard scenario. Reward function example fore extinguishing fire related subspace.	131
8.5	Terrorist vs. Guard scenario. Reward function example fore taking a gun related subspace.	131
8.6	Terrorist vs. Guard scenario. Hierarchy of options.	132
8.7	Examples of the policies of the options.	132
8.8	Performance in 4 and 6 halls problems.	134
8.9	Terrorist vs. Guard scenario. Convergence in number of steps when the reward function is modified at episodes 250 and 600.	135
8.10	Policy size on 4 and 6 halls problem.	137
1	Quelques combinaisons de variables qui représentent les états impossibles dans le problème BLOCKS WORLD.	152
2	Problème BLOCKS WORLD (taille 4-3-4) : performance sur épisodes.	153

3	Le problème de LIGHT BOX : numéros et couleurs des “lumières” avec leurs dépendances.	155
4	Exemple des options découverts dans le problème de LIGHT BOX.	155
5	Convergence en nombre d’épisodes dans le problème LIGHT BOX.	156
6	Scénario Terrorist vs. Guard. Hiérarchie des options.	157
7	Scénario Terrorist vs. Guard. Convergence en nombre de pas de temps quand la fonction de récompense est modifiée à l’épisode 250 et 600.	158

List of Tables

4.1	Probabilities for joint variables, resulting from Figure 4.7.	49
6.1	BLOCKS WORLD representations of the situation given in Figure 6.2 (G stands for gripper).	70
6.2	Probabilities for joint variables, resulting from Figure 6.5.	74
6.3	Transition probabilities for joint variables resulting from Figure 6.5, given that $\bar{x}_1'x_2'$ is impossible.	75
6.4	MAZE6 performance.	79
6.5	The BLOCKS WORLD problem: rate of impossible states and time to perform one step (in seconds). A “-” indicates that the value could not be obtained after three days of computation. IMPSVI uses trees to represent impossible states (the time with the <i>ad hoc</i> rules is indicated between parentheses).	80
6.6	MAZE6 performance (cost in memory and time).	81
6.7	The BLOCKS WORLD problem: rate of impossible states and time to perform one step (in seconds). A “-” indicates that the value could not be obtained after three days of computation. IMPSPITI uses trees to represent impossible states (the time with the <i>ad hoc</i> rules is indicated between parentheses).	84
6.8	MAZE6 and D-MAZE performance (cost in memory and time) with DYNA-Q (in number of Q-functions) and IMPSPITI (in number of leaves in policy tree).	86
7.1	State variables of the TAXI problem.	111
7.2	Actions of the TAXI problem.	111
7.3	Performance on the stochastic TAXI problem.	113
7.4	Options of the TAXI problem (P - Passenger, TL - Taxi Location, PL - Passenger Location, PD - Passenger Destination).	113
7.5	Performance on the LIGHT BOX problem (Policy and value function size in total number of nodes in decision trees).	114
7.6	Performance on the TAXI problem with 2 transition function representations.	117

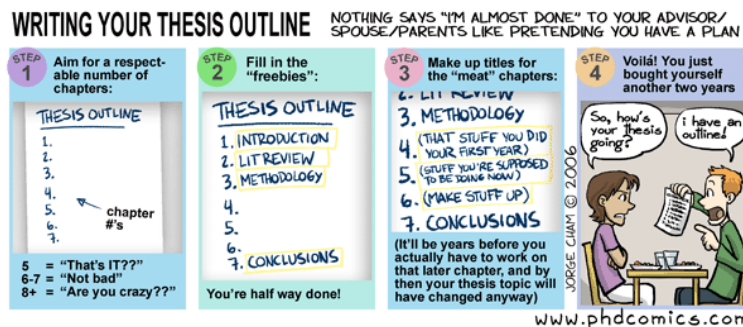
7.7	Performance on the LIGHT BOX problem with 2 transition function representations.	117
7.8	Performance on the TAXI problem using local or global transition function.	118
8.1	State variables of Terrorist vs. Guard scenario with 4 halls.	127
8.2	Actions of Terrorist vs. Guard scenario.	127
8.3	State variable value changes for the action <i>Move West</i> of Terrorist vs. Guard scenario with 4 halls.	129
8.4	Evolution of the option “Shoot Terrorist in Hall 4” during the discovery process.	133
8.5	Evolution of the option “Extinguish fire in Hall 4” during the discovery process.	133
8.6	Terrorist vs. Guard scenario. Performance in time (sec/step). The global time is given in parenthesis.	136
1	Le problème de BLOCKS WORLD : le taux d'états impossibles et le temps nécessaire pour accomplir chaque pas (en secondes). “-” signifie que nous n'étions pas en mesure d'obtenir les résultats après trois jours de calcul. IMPSPITI utilise les arbres pour représenter les états impossibles. Les résultats sont donnés pour le cas où les états visités sont enregistrés dans un arbre de décision (le temps entre parenthèses représente les résultats avec les règles <i>ad hoc</i> pour définir les états impossibles).	153
2	Performance sur le problème LIGHT BOX (la taille des fonctions de politique et de valeur en nombre total des nœuds dans les arbres de transition).	156

List of Algorithms

2.1	Greedy	34
2.2	Policy Iteration	34
2.3	Value Iteration	35
3.1	TD - LEARNING	38
3.2	Q-LEARNING	39
3.3	DYNA-Q	40
3.4	Prioritized sweeping	41
4.1	PRegress	48
4.2	Regress	50
4.3	Structured Value Iteration (SVI)	50
4.4	Greedy	51
4.5	Structured Successive Approximation (SSA)	51
4.6	Structured Policy iteration (SPI)	51
4.7	UpdateFMDP	53
4.8	LearnTree	54
4.9	Incremental SVI (IncSVI)	55
5.1	Hierarchical Semi-Markov Q-LEARNING (HSMQ)	61
5.2	VISA	63
6.1	IMPSVI	76
7.1	TeXDYNA	96
7.2	Update options (model of transitions: one tree per variable)	100
7.3	Update options (model of transitions: one tree per variable per action)	101
7.4	SPITI with options	107
7.5	Choose Option	107
.1	TeXDYNA	154

Chapter 1

Introduction



This thesis was funded by CIFRE convention 1032/2006 with THALES company. The thesis was accomplished in THALES Simulation department and conjointly in the Synthetic Environments & Simulation team in ThereSIS lab (Thales European Research center for E-Gov & Secured Information Systems). These departments are working on simulation of human behavior in military training and civil security simulations, proposing software solutions for project testing, training and control.

The main challenges of such applications are building realistic and adaptable artificial entities, on the one hand, and providing high-level behavior and structured representation of the solution understandable by a human operator, on the other hand.

Consequently, the applicative goal of this work is to test the methods that can be efficiently used for modeling human behavior in such simulations.

1.1 The industrial simulation domain

The industrial Simulation domain, particularly Synthetic Environments also called “Serious Games” are used to simulate training or testing environments, that are cheaper and easier to realize on artificial support including software simulation than in real conditions. The simulation techniques are used to deal with a large scope of problems involving planning and learning in stochastic environments, such as: simulating adaptive agents behaving in a dynamic environment, solving optimization problems or even discovering the internal structure of an unknown environment. For instance, manufacturing systems that optimize scheduled task sequences or simulating the behavior of NPCs (non-player characters) in computer games evolving in a stochastic environment are examples of simulation where solving these problems is mandatory. Among other things, the success of the “Serious Games” approach comes from the fact that using existing software development infrastructure, the developers can create simulations at a fraction of the cost of traditional expenses. Used for training, advertising, forecasting, or education, the Synthetic Environments domain is in full expansion. Besides, while traditional simulators usually cost millions of dollars not only to develop, but also to deploy, and generally require specialized hardware, the cost of support for “Serious Games” is much lower.

Basically, the Synthetic Environment & Simulation staff in THALES develops new types of realistic simulators for civil and military applications such as Computed Generated Forces or Operation Control Center. That is simulation of the physical and behavioral environment that represents real life tasks that can be tested or learned by the users of the system. In particular, such systems can compute complex and adaptive behaviors as well as low level navigation and environment interactions. With such an engine, it is possible to populate complex critical infrastructure, for instance a subway station, an airport or a whole city, with a crowd of persons exhibiting context specific decisions and optimizing their own motivations according to their roles as passengers, policemen, firemen, soldiers or terrorists. Each agent in such simulation is guided by a decision mechanism more or less complicated according to the complexity of the expected behavior.

Modeling human agents behavior

In this work, we are focusing on the simulation of the behavior of human agents. By behavior, in this case, we mean a sequence of actions that have an influence on the environment or the internal state of the agent.

Modeling human agents behavior raises some constraints. First, this behavior must seem natural and realistic to the user. Thus, the resulting behavior (or policy) and parameters must have some legibility to be usable by an uninformed user. For instance, when

solving tasks that have critical issues that need to be checked by a human, providing a clear solution in a context of user friendly systems is an advantage. Then, it requires a high computation performance, since many simulations are real time systems that run a large number of agents simultaneously. Finally, such behavior can be hard coded, derived from a rule base or learned from scratch. When a learning method is used, there are two strategies: *online* or *offline* learning. While in offline learning the entire sequence of inputs, representing the trajectory of the agent in the environment, is given in advance and the behavior sequence is computed once and for all, the online algorithms must process each input in turn, without detailed knowledge of future inputs. Online learning methods improve their response to the environment along their interaction with it giving rise to an additional behavior validation problem. Thus, the choice of the method depends on the application context. For instance, the behavior of the agents that represent soldiers, who follow a strict military doctrine, would be better predefined, hard coded in the program or learned offline and validated before their actual use in a simulation. By contrast, the behavior of the agents representing civilians, militias, terrorists may have learned components in order to be less predictable and adaptable to the changing situation. In this thesis we propose techniques that can potentially be used in both contexts, but our target is rather the offline use since human validation may play an important role in the simulation domain.

In this perspective, the behavior of such agents is modeled as an action selection mechanism (ASM), that is a computational mechanism that implements the process of choosing, at each moment in time, the most appropriate action with respect to the current and sometimes expected, situation of the environment given as inputs. This system should implement the following requirements adapted from [Tyrrell, 1993] :

- **Adaptability:** the need to be able to adapt the behavior to changing situation, that is, learning new responses to the evolving dynamic environment, or in other words, creating adaptable agents by opposition to the deterministic ones in static environments;
- **Compromise:** the need to be able to choose actions that, while not the best choice for any one sub-problem alone, are best when all sub-problems are considered simultaneously;
- **Persistence:** the need to have a tendency to persist with an action overtime in order to avoid dithering among activities and, to continue the current sequence once started, rather than beginning a new sequence for a different goal;
- **Opportunism:** the need to incorporate information about availability. This should allow the agent to interrupt other activities to take advantage of infrequently-available opportunities if they should suddenly arise;

- Interaction: the need to take into account the behavior of other agents, that is exhibit realist crowd movements but also respond to stress and aggressiveness;
- Quick response time: the need to compute the output of the ASM in real time.

Several efficient approaches to such kind of simulations were developed over last decades. On the one hand, there are systems like *Cathexis* [Velásquez, 1998], *OCC* [Ortony et al., 1988], *Sloman Architecture* [Sloman, 2001] or *Emile* [Gratch and Marsella, 2001] that do not learn behavior, but implement an emotional representation of the situation of the entities and then use it for action selection. On the other hand, learning can be implemented by using some rules systems like SOAR [Laird et al., 1987] in MRE (Mission Rehearsal Exercise) [Traum et al., 2004]. Some of these techniques were tested in the previous work [Kozlova, 2006] on the example of SOAR architecture. Our aim in the current work is to study non-emotional learning strategies. In fact, our goal is to provide the mechanisms of action selection that can address three of the requirements presented here above, namely: adaptability, compromise and persistence in the context of complex simulation problems.

1.2 Reinforcement Learning

In a way, learning consists in acquiring knowledge about what is coming next. The ability to learn is possessed by (some?) PhD students, (some?) humans, animals and some machines. More generally, a learning subject gets new knowledge, skills or behaviors, preferences or values. Learning is a process where a learning agent interacts with a dynamic (i.e. evolving in time) environment.

In AI Machine Learning field [Russell and Norvig, 2003] addresses the problem of automated knowledge acquisition in its own way. Here, learning means automatically recognizing complex patterns and making intelligent decisions based on some data. Unlike humans, who have to make complex decisions based on incomplete, muddled and often very sparse data, machines mostly deal with well defined and somehow organized data. Basically, these data are provided by the internal status of the learning system and the dynamic environment of the learning agent. These data can contain variables representing some characteristics of the system or of the environment and also an outcome that qualifies the interaction between the learning agent and its environment.

The application field that delimits this work is concerned by the simulation of adaptive agents behaving in a dynamic environment. The representation of such problems can be modeled as a discrete (a sequences of separate time steps) and online (the solution is computed while exploring the environment) process that involves a set of actions and states. Each state represents some situation at a given moment in time. The agent moves from one state to another by taking actions in its environment and receives rewards for

its accomplishments. In this case, the problem to solve is to find a sequence of actions representing a trajectory through a solution space that optimizes the expected outcome. The major difficulties of such problems are the following points:

- The size of the environment - that is the expression the well known “curse of dimensionality” problem where the space of the solution grows exponentially in relation to the size of the problem; the definition of states, actions and rewards play an important role.
- The complexity of the final solution - that is, on the one hand, providing the solution that can be understood by the user and, on the other hand, the adequateness of the solutions to the complexity of the problem.

One of the most popular solutions to such problems of modeling the behavior of the agents in the complex simulated environments is the Reinforcement Learning (RL) approach [Sigaud, 2004]. RL methods, based on the animal learning theories [Thorndike, 1911], and further developed in [Sutton and Barto, 1998], are trial and error methods to compute optimal solutions for stochastic sequential decision problems. Then, such problems can be modeled as a Markov Decision Process (MDP) [Puterman, 1994], a standard mathematical framework for learning and planning under uncertainty. Although there are solutions proved to be sound and efficient, the difficulties arise when the problem is large and/or has a complex structure (e.g. contradictory goals).

In order to address the difficulty with the size of the problem, Factored-MDPs (FMDPs) [Boutilier et al., 1995] allow the representation of large-scale problems by exploiting their structure. These solutions factorize (by aggregating similar states and representing sets of states) the state space in order to reduce it. But, in practice, a perfect knowledge of the state transitions in the problem is rarely available. This case is addressed by Factored Reinforcement Learning (FRL), an approach to FMDPs where the transition and reward functions of the problem are learned using Supervised Learning methods.

Another approach to the large problems difficulty is scaling over the problem size using divide and conquer methods such as Hierarchical Reinforcement Learning (HRL), based on Hierarchical-MDPs (HMDPs). HMDPs simplify the problem by decomposing the overall task into a hierarchal set of sub-tasks that are easier to solve individually. In some cases, one can use hierarchical decompositions designed by a human operator. But in many situations such as large, complex or unusual problems, providing the correct decomposition is difficult if not impossible. Therefore, automatically finding a hierarchical decomposition is one the most challenging problems in the domain of HRL.

1.3 Contributions

The contributions of this thesis can be considered in two dimensions: theoretical and applicative. From the theoretical point of view, this thesis proposes a new algorithm to solve hierarchical and factored RL problems when the structure is unknown. As to the practical applications, the goal of this work is to test how HRL and FMDP methods can be efficiently combined to address the long standing goal of modeling human behavior in Synthetic Environment simulations.

In brief, in this thesis we study the solutions to large and complex stochastic sequential decision problems. Our main contributions are the following:

Dealing with impossible states. Basically, an FMDP is a standard representation for large sequential decision problems under uncertainty. In this framework, the variables are considered independent from one another. However, in some cases, dependencies between state variables at a given time step can appear giving rise to more complex solution algorithms. We address a particular case of this problem where some combinations of state variable values may not occur, giving rise to what we call impossible states. We propose a new class of algorithms (IMPSVI and its FRL version, IMPSPITI) that adapts existing FRL algorithms in a theoretically well-founded way in order to address a wider class of problems that contains such kind of additional constraints.

Hierarchical decomposition in FRL. We propose the TeXDYNA framework that automatically discovers the hierarchical decomposition of a sequential decision problem by combining the abstraction techniques of HMDPs with an FRL method. The first contribution comes from the fact that the hierarchical decomposition is derived directly from the trees representing the transition function of the problem, taking advantage of the internal structure of the task. To the best of our knowledge, at this moment in time, there is no hierarchical FRL (HFRL) system that implements sub-task discovery based on the decision trees structure. Hence, the central contribution of this thesis comes from the fact that the discovery of the hierarchical decomposition and the learning of the model of the FMDP are simultaneous and performed online.

Application to the simulation domain. To satisfy the industrial requirements of this work, all solutions proposed in this thesis are based on a framework that returns transition and reward functions of the problem as well as a solution policy in a decision tree form which is understandable even by an uninformed user. We show on a simplified simulation problem the kind of application where HRL and FRL techniques like TeXDYNA are useful. We also show the limitation of our approach when facing more realistic simulation problems and indicate possible directions for addressing industrial applications in the future.

1.4 Outline

The remainder of this thesis is organized as follows:

In **Part I (Chapters 2, 3, 4 and 5)** we give a brief review of the MDPs, FMDPs, HMDPs and RL approach, as well as the state of the art of the related algorithms.

In **Part II (Chapters 6, 7 and 8)** we present new algorithms proposed in this thesis and their applications. More precisely:

In **Chapter 6** we present IMPSVI and IMPSPITI, new FRL algorithms that deal with impossible states.

In **Chapter 7** we introduce the TeXDYNA framework that performs automated discovery of options in FRL.

In **Chapter 8** we give an example of practical application of this work to a simplistic instance of the industrial simulation domain and discuss future directions of the applicative effort.

Finally, **Chapter 9** summarizes the main contributions of this thesis and discusses future work and open problems.

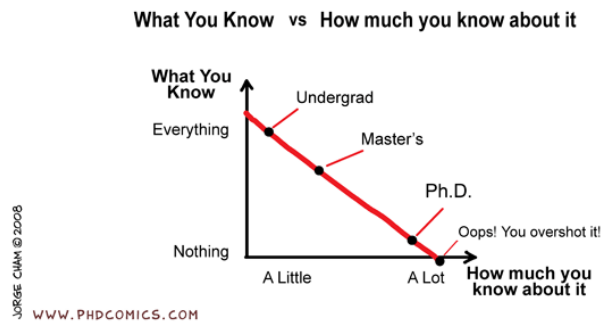
To end-up this overview, remembering Umberto Eco's "Anti-library", we have to mention the "non-contributions" or the subjects that are interesting or related to the topic, but not treated in this thesis. For instance: we do not study partially observable MDPs, do not use emotional or motivational learning strategies, do not look at the continuous variables and actions possibilities, do not check the multiagent applications either, and, certainly, so many more not less interesting directions.

Part I

Background

Chapter 2

Markov Decision Processes



Markov Decision Processes (MDPs), named after Andrei Andreyevich Markov (1856 - 1922), is an extension of Markov chains with the addition of actions and rewards. An MDP is a mathematical model for the random evolution of a memoryless system. Such a system respects the *Markov property*, that is a given future state depends only on the present state and where knowing more about the past does not bring any further information.



A. A. Markov

An MDP, as a discrete time stochastic control process, provides a mathematical framework for modeling decision making in situations where the outcome is partially controlled by the agent. One of the first solutions methods for MDPs, Dynamic Programming (DP) algorithms - simplifies a complicated problem by breaking it down into simpler subproblems in a recursive manner. These algorithms were proposed in the 1950s by [Bellman, 1957] and further improved and detailed by [Howard, 1971], [Puterman, 1994] and [Bertsekas, 1995].

2.1 Definitions

Formally an MDP is defined by a tuple $\langle S, A, R, P \rangle$, where

- S is a finite set of states;
- A is a finite set of actions;
- $R : S \times A \rightarrow \mathbf{R}$ is the immediate reward function $R(s, a)$;
- $P : S \times A \times S \rightarrow [0, 1]$ is the transition probability function $P(s'|s, a)$.

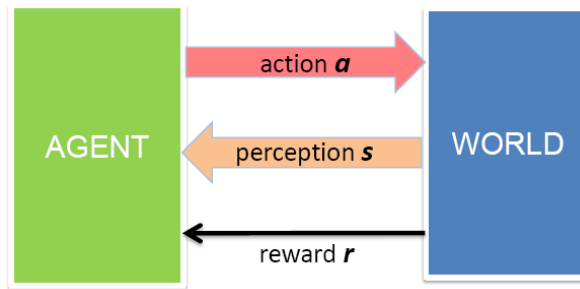


Figure 2.1: Markov Decision Process.

An MDP is called finite if its state and action spaces are both finite. A state s is called terminal (or absorbing) if the process never leaves it once entered. An MDP is called episodic if it is finite-horizon, that is time step $t \in \mathbb{N}$, has terminal states and the process restarts from the beginning once a terminal state was reached [Szepesvári, 2009].

Generally, an MDP (Figure 2.1) represents an agent which, during its evolution through the environment, takes an action $a \in A$ in state $s \in S$ and proceeds to the state $s' \in S$ with probability $P(s'|s, a)$ receiving a reward $R(s, a)$. A policy $\pi : S \times A \rightarrow [0, 1]$ defines the probability $\pi(s, a)$ that the agent takes the action a in state s .

The goal of planning or learning is to find a policy π that maximizes the outcome, i.e. cumulative reward of the agent. Since there is a reward given for every state/action pair, it is possible to associate a value to each state that represents the expected value of the next cumulative reward derived from the value of the next state. In other words, it is possible to define $V_\pi(s)$ as a value obtained in a state s equal to the cumulative expected reward received in the state s following the policy π . That is

$$V_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t))\right] \quad (2.1)$$

Therefore, given a policy π , the value of the state s is defined by the Bellman equation:

$$V_{\pi}(s) = R_{\pi}(s) + \gamma \sum_{s' \in S} P_{\pi}(s'|s, a) V_{\pi}(s') \quad (2.2)$$

with $R_{\pi}(s) = R(s_t, \pi(s_t))$ and $\gamma \in]0, 1]$ the discount factor.

A policy π is optimal if, $\forall s \in S : V_{\pi}(s) \geq V_{\pi'}(s)$. Since there may be more than one optimal policy, all the optimal policies are noted π^* . The value function of any optimal policy is called the optimal value function and is noted V^* . In other words, the optimal value function maximizes the value of all states $s \in S$ with respect to the policy π and defines the solution of the Bellman optimality equation:

$$V^*(s) = \max_{\pi} [V_{\pi}(s)] \quad (2.3)$$

The action-value function $Q_{\pi}(s, a)$ is the expected return starting in a state s , taking action a and following the policy π and is defined by the Bellman equation:

$$Q^{V_{\pi}}(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V_{\pi}(s') \quad (2.4)$$

The optimal action-value function $Q^*(s, a)$ is defined by:

$$Q^*(s, a) = \max_{\pi} [Q_{\pi}(s, a)] \quad (2.5)$$

or in terms of optimal value function V^* :

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a) V^*(s') \quad (2.6)$$

Hence, the optimal value function is defined by:

$$V^*(s) = \max_a [Q^*(s, a)] \quad (2.7)$$

and the optimal policy as:

$$\pi^*(s) = \operatorname{argmax}_a [Q^*(s, a)] \quad (2.8)$$

If the value function of the problem is known, it is possible to define a greedy policy of Equation 2.8 using Algorithm 2.1.

Algorithm 2.1: Greedy

input : $V(s)$
output: $\pi(s)$

- 1 **forall** the $a \in A$ **do**
- 2 $Q(s, a) \leftarrow R(s, a) + \gamma \sum_{s' \in S} P(s'|s, a)V(s')$
- 3 $\pi(s) = \operatorname{argmax}_a Q(s, a)$
- 4 **return** $\pi(s)$

2.2 Algorithms

The standard DP algorithms are methods to compute the optimal policy of the finite-horizon MDP where the transition and reward functions are known.

2.2.1 Policy Iteration

The Policy Iteration algorithm is given in Algorithm 2.2 [Howard, 1971]. It starts with an initial policy that is evaluated by a policy evaluation process. The policy evaluation process is performed by solving a system of linear equations where the reward $R(s, a)$ and transition probability $P(s'|s, a)$ are given and the value function $V_\pi(s)$ is unknown. That evaluation can also be done by an iterative computation of the value function for that policy, where the value function of a policy is the expected infinite discounted reward that will be gained, at each state, by executing that policy. Then, when the value of each state under the current policy is known, the algorithm improves the current policy by choosing greedily an action in each state with the best value $V_\pi(s')$.

Algorithm 2.2: Policy Iteration

input : \emptyset
output: $V^*(s), \pi^*(s)$

- 1 Initialize $V_\pi(s)$ and $\pi(s)$ arbitrarily
- 2 **Policy Evaluation**
- 3 Compute the value function $V(s)$ of policy π by solving the linear equations
- 4 $V_\pi(s) = R(s, a) + \gamma \sum_{s'} P(s'|s, a)V_\pi(s')$
- 5 **Policy Improvement**
- 6 $\text{policy-stable} \leftarrow \text{true}$
- 7 **forall** the $s \in S$ **do**
- 8 $b \leftarrow \pi(s)$
- 9 $\pi(s) \leftarrow \text{Greedy}(s)$
- 10 **if** $b \neq \pi(s)$ **then** $\text{policy-stable} \leftarrow \text{false}$
- 11 **if** policy-stable **then** stop **else** go to step 2
- 12 **return** $V_\pi(s)$ and $\pi(s)$

This step is guaranteed to strictly improve the performance of the policy. When no more improvements are possible, then the policy is optimal. The iterations of the algorithm produce a sequence of monotonically improving policies and value functions as shown in Figure 2.2.



Figure 2.2: Policy Evaluation/Improvement

Since there is a finite number of distinct policies, and the sequence of policies improves at each step, this algorithm terminates in at most an exponential number of iterations. Depending on the problem size, the algorithm can result in an important computational cost. Indeed, at each iteration, the algorithm has to run through the entire states space S as many times as necessary until the convergence of the $V_{\pi}(s)$ and then improve the policy $\pi(s)$ for each state $s \in S$.

2.2.2 Value Iteration

The Value Iteration algorithm, given in Algorithm 2.3 [Bellman, 1957] does not maintain a set of policies to evaluate. Instead, the value of $\pi(s)$ is calculated once it is needed usually in the end when $V(s)$ has converged. In fact, Value Iteration is obtained simply by turning the value function optimality equation (Equation 2.7) into an update rule.

Algorithm 2.3: Value Iteration

input : \emptyset
output: $V^*(s), \pi^*(s)$

- 1 Initialize $V(s)$ arbitrarily
- 2 **repeat**
- 3 $v \leftarrow V(s)$
- 4 **forall the** $s \in S$ **do**
- 5 **forall the** $a \in A$ **do**
- 6 $Q(s, a) \leftarrow R(s, a) + \gamma \sum_{s'} P(s'|s, a)V(s')$
- 7 $V(s) \leftarrow \max_a Q(s, a)$
- 8 **until** $|v - V(s)| < \epsilon$
- 9 $\pi(s) \leftarrow Greedy(s)$
- 10 **return** $V(s)$ and $\pi(s)$

In a same way as Policy Iteration, Value Iteration algorithm requires complete sweeps over the state space at each computation cycle.

2.3 Discussion & Summary

MDPs provide a mathematical framework for modeling decision-making and are useful for studying a wide range of optimization problems solved via Dynamic Programming (DP) and Reinforcement Learning (RL).

However, the algorithms presented in this chapter are subject to the “curse of dimensionality” [Bellman, 1961] that refers to the problem caused by the exponential increase in volume associated with adding extra dimensions to a state space. For instance, a problem represented by 10 binary variables has a 2^{10} possible states, the problem with 11 binary variables has a 2^{11} possible states, etc. Each time an additional dimension is introduced, the size of the problem increases by an order of magnitude. That is the reason why exact algorithms that consider the entire state-action space of the problem cannot address large scale problems.

Chapter 3

Reinforcement Learning



Inspired by the related psychological theory [Thorndike, 1911], Reinforcement Learning (RL) is a trial-and-error learning process driven by interacting with an environment. The RL agent does not have any prior knowledge about the environment and learns from the consequences of its actions by receiving a reinforcement signal. In this case, the optimal value function and/or the optimal policy have to be learned by an adaptive process. The goal of the RL algorithms is to select actions that maximize the expected cumulative reward of the agent.

3.1 Algorithms

Basically, the algorithms can be split in two classes: model-free and model-based algorithms [Kaelbling et al., 1996]. The model-free algorithms attempt to learn a policy without learning a model of the environment. Temporal Difference Learning (TD - LEARNING) like Q-LEARNING or SARSA and ACTOR-CRITIC are the examples of model-free algorithms.

In contrast with model-free algorithms, model-based algorithms build an estimated model of the transition function $P(s'|s, a)$, which describes the transition probability of going from state s to s' when performing action a and of the reward function $r(s, a)$ i.e. receiving the reward r in a state s after executing the action a . The model-based set of algorithms are improved represented by DYNA-like methods [Sutton, 1991], Prioritized Sweeping (PS) [Moore and Atkeson, 1993] and Real Time Dynamic Programming (RTDP) [Barto et al., 1993].

3.1.1 TD - LEARNING

Algorithm 3.1: TD - LEARNING

initialization: $\forall s \in S$ define $V(s)$ arbitrarily,
 π the policy to be evaluated,
 learning rate α

```

1 foreach episode do
2    $s \leftarrow$  initial state
3   while  $s$  is not terminal do
4     foreach time step in episode do
5       choose action  $a$  given by  $\pi$  for  $s$ 
6       take action  $a$ , observe next state  $s'$  and reward  $r$ 
7        $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$ 
8        $s \leftarrow s'$ 
  
```

TD - LEARNING [Sutton and Barto, 1998] is a combination of Monte Carlo ideas (repeated random sampling) and DP ideas. The goal of the TD approach is to learn to predict the value of a given state based on what happens in the next state by bootstrapping, that is updating values based on the learned values, without waiting for a final outcome. While TD(0) is one-step backup algorithm (see Algorithm 3.1, it is possible to perform the averaging of the n-step backups by applying TD(λ), where the λ parameter defines the importance of the rewards from the distant states, with $\lambda = 1$ producing parallel learning to Monte Carlo RL algorithms. Algorithm 3.1 gives the TD(0) procedure.

3.1.2 Q-LEARNING & SARSA

Q-LEARNING [Watkins and Dayan, 1992] (Algorithm 3.2) is a unifying TD control algorithm which simultaneously optimize the value function and the policy. Q-LEARNING is often used in an off-policy manner, learning about the greedy policy while the data is generated by a different policy that ensures exploration.

Algorithm 3.2: Q-LEARNING

initialization: $\forall s \in S, \forall a \in A$ define $Q(s, a)$ arbitrarily,
learning rate α
 $s \leftarrow$ initial state

1 **foreach** *time step* **do**
2 choose action a using exploration policy derived from Q
3 execute action a , observe next state s' and reward r
4 $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
5 $s \leftarrow s'$

SARSA (State-Action-Reward-State-Action) is an on-policy TD algorithm that learns the action-value function. The name reflects the fact that the main function for updating the Q-value depends on the current state of the agent s , the action the agent chooses a , the reward r the agent gets for choosing this action, the state s' where the agent will be in after taking that action, and finally the next action a' the agent will choose in its new state, hence (s, a, r, s', a') . As in all on-policy methods, the Q-function are continually estimated for the behavior policy using Equation (3.1), and the current policy is updated greedily.

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \quad (3.1)$$

see line 4 in Algorithm 3.2.

3.1.3 Actor-Critic

ACTOR-CRITIC methods [Barto et al., 1983] are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. The policy structure is known as the actor, because it is used to select actions, and the estimated value function is known as the critic, because it criticizes the actions made by the actor. Learning is always on-policy: the critic must learn about and critique whatever policy is currently being followed by the actor. The critique takes the form of a TD error.

3.1.4 DYNA-Q

The model-free approaches, presented so far, do not exploit the knowledge about the environment gathered during exploration. [Sutton, 1991] proposed a DYNA architecture that exploits that experience to learn the model of the transition and reward functions and then use it to directly improve the value function and policy using one of the TD - LEARNING algorithms. In this approach, the planning is achieved by applying

Reinforcement Learning methods to the simulated experiences just as if they really happened. The relationships between experience, model, values, and policy are summarized in Figure 3.1.

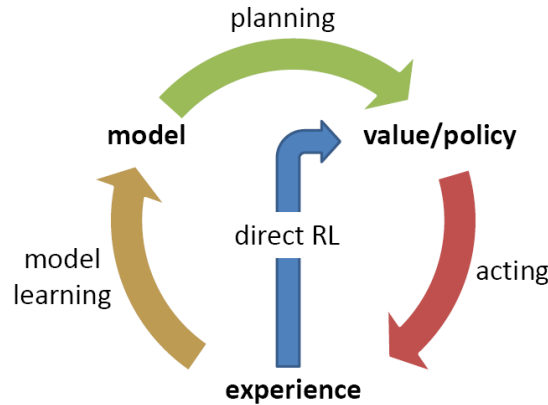


Figure 3.1: The DYNA process.

Algorithm 3.3: DYNA-Q

initialization: $\forall s \in S, \forall a \in A$ define $Q(s, a)$ arbitrarily,
 $Model(s, a)$ world model

- 1 **foreach** *time step* **do**
- 2 **Decision:**
- 3 $s \leftarrow$ current state
- 4 choose action a using the exploration policy derived from Q
- 5 execute action a , observe next state s' and reward r
- 6 **Learning:**
- 7 $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
- 8 update $Model(s, a)$ from the observation $\langle s', a, r \rangle$
- 9 **Planning:**
- 10 **for** N *times* **do**
- 11 $s \leftarrow$ a randomly chosen observed state
- 12 $a \leftarrow$ a randomly chosen action already executed in a state s
- 13 predict s' and r using $Model(s, a)$
- 14 $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

Algorithm 3.3 presents the version of DYNA using Q-LEARNING for policy learning called DYNA-Q. DYNA-Q follows the process shown in Figure 3.1: acting, model-learning, planning and direct RL. The model-learning method is table-based and assumes the world is deterministic. After each transition, the model records in its table the entry for the prediction that will follow. The planning method is the random-sample one-step

tabular Q-LEARNING method where the sample size depends on the parameter N , that is a number of planning steps usually fixed between $N = 0$ for a nonplanning agent and $N = 50$. Finally the direct RL method is one-step tabular Q-LEARNING.

The DYNA algorithm can be generalized to the stochastic case by modifying the Q-function update [Peng and Williams, 1993]. In this case, Q-functions must be weighted by the probability of the corresponding transition model $P(st|s, a)$, that is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \times P(st|s, a)[r + \gamma \max_{a'} Q(st, a') - Q(s, a)]$$

3.1.5 Prioritized sweeping

Algorithms like DYNA suffer from being relatively undirected, since they update states chosen randomly in a set of visited states. It is particularly true when the goal has just been reached or when the agent is stuck in a dead-end continuing to update random state-action pairs, rather than concentrating on the “interesting” parts of the state space. The prioritized sweeping method [Moore and Atkeson, 1993] and a similar Queue-Dyna method [Peng and Williams, 1993] deal with these problems.

Algorithm 3.4: Prioritized sweeping

initialization: $\forall s \in S, \forall a \in A$ define $Q(s, a)$, $Model(s, a)$ and $PQueue = \emptyset$

```

1 foreach time step do
2    $s \leftarrow$  current state
3   choose action  $a$  using exploration policy derived from  $Q$ 
4   execute action  $a$ , observe next state  $st$  and reward  $r$ 
5    $Model(s, a) \leftarrow st, r$ 
6    $p \leftarrow |r + \gamma \max_{a'} Q(st, a') - Q(s, a)|$ 
7   if  $p > \theta$  then insert  $s, a$  into  $PQueue$  with priority  $p$ 
8   for  $N$  times, while  $PQueue \neq \emptyset$  do
9      $s, a \leftarrow first(PQueue)$ 
10     $st, r \leftarrow Model(s, a)$ 
11     $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(st, a') - Q(s, a)]$ 
12    foreach  $\bar{s}, \bar{a}$  predicted to lead to  $s$  do
13       $\bar{r} \leftarrow$  predicted reward
14       $p \leftarrow |\bar{r} + \gamma \max_{a'} Q(s, a) - Q(\bar{s}, \bar{a})|$ 
15      if  $p > \theta$  then insert  $\bar{s}, \bar{a}$  into  $PQueue$  with priority  $p$ 

```

The Prioritized sweeping algorithm, given in Algorithm 3.4, is similar to DYNA, except for the fact that the updates are no longer chosen at random but relatively to the states with the largest Bellman error (computed in line 14 of the Algorithm 3.4), that is the states that bring more information about the changes in the environment. Therefore, the algorithm stores additional information about the states remembering the predecessors, i.e. the states from which there is a non-zero probability to reach the given state.

3.2 Discussion & Summary

In this chapter, we presented the basics on RL as well as the major RL algorithms namely TD - LEARNING, Q-LEARNING, SARSA, DYNA and Prioritized sweeping.

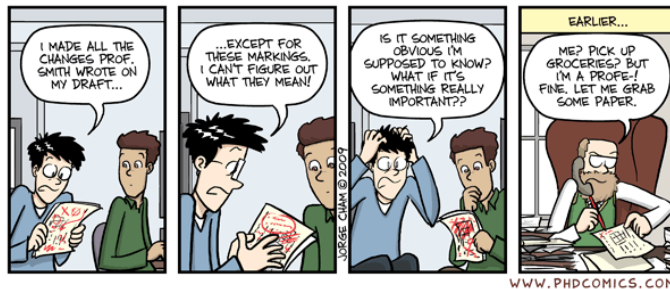
According to R. Sutton, Reinforcement Learning theory is based on four key ideas:

1. Time/life/interaction - it involves an interaction with an external world over and in time, that is the interaction with the environment during a period of time separated in atomic steps;
2. Reward/value/verification - a formulation of short and long-term goals that enables the agent to tell for itself when it is right or wrong;
3. Sampling - that is trial-and-error learning as a solution to the “curse of dimensionality” where just a part of the entire state-action space is used to build an approximate solution;
4. Bootstrapping - that is solving problems using recursive solutions involving the methods of Bellman equation solution, temporal-difference learning and other approximate solutions.

The major challenges of RL domain are the “curse of dimensionality”, temporal credit assignment problem, state-action space tiling, non-stationary environments and exploration-exploitation dilemma.

Chapter 4

Factored Reinforcement Learning



In this chapter we first present Factored Markov Decision Process (FMDP) and then Factored Reinforcement Learning (FRL) approach. The idea of factorization comes from the necessity to represent large-scale problems that cannot be solved with the standard MDP representations. FMDPs [Boutilier et al., 1995] exploit the structure of the problem to represent large MDPs compactly when the state of the problem can be decomposed into a set of random variables. In the remainder, we present the definition of FMDPs and compact representation techniques, as well as the FRL algorithms used in this framework.

4.1 FMDPS

In the FMDP framework, the state space of the problem is represented as a collection of random variables $X = \{X_1, \dots, X_n\}$. A state is then defined by a vector $x = (x_1, \dots, x_n)$ with $\forall i, x_i \in \text{Dom}(X_i)$. Then for each action a , the model

of transitions is defined by a separate Dynamic Bayesian Network (DBN) model [Dean and Kanazawa, 1989].

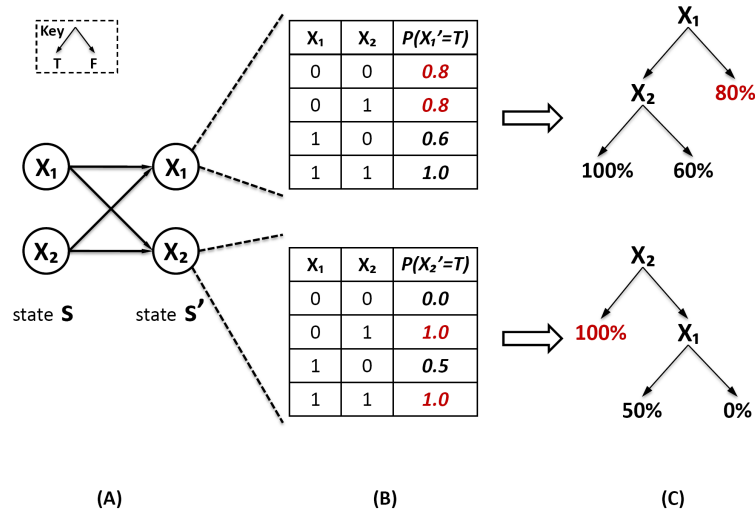


Figure 4.1: Transition function representation within the FMDP framework. (A) Dynamic Bayesian Network representing the dependencies between the variables. (B) Tabular representation of the Conditional Probability Distribution. (C) Decision Tree representation of the Conditional Probability Distribution.

The DBN model for each action, noted G_a is a two-layer directed acyclic graph whose nodes are $\{X_1, \dots, X_n, X'_1, \dots, X'_n\}$ with X_i a variable at state s and X'_i the same variable at the next state s' . The parents of X'_i are noted $\text{Parents}(X'_i)$. The model of transitions is quantified by Conditional Probability Distributions (CPDs), noted $\forall a \in A$, $P(X'_i | \text{Parents}(X'_i), a)$, associated to each node $X'_i \in G_a$. The CPD can be represented in a tabular, decision tree or rules form.

Figure 4.1 shows the representation of a transition function with the CPD expressed in a tabular and corresponding decision tree form and Figure 4.2 shows the representation of a reward function. A tree representing $P(X'_i | \text{Parents}(X'_i), a)$ has three functional elements:

- nodes - the test on the variable X_i ;
- branches - the set of nodes i.e. set of tested variables with their values representing a state $s \in S$ or a set of states S_i having the same value;
- leaves - the distribution of probabilities $P(X'_i | S_i, a)$ over the set of states S_i represented by the branch leading to this leaf.

As exemplified in Figures 4.1 and 4.2, the tree representation is more compact than the tabular one since the states that have the same values are aggregated in the same leaf

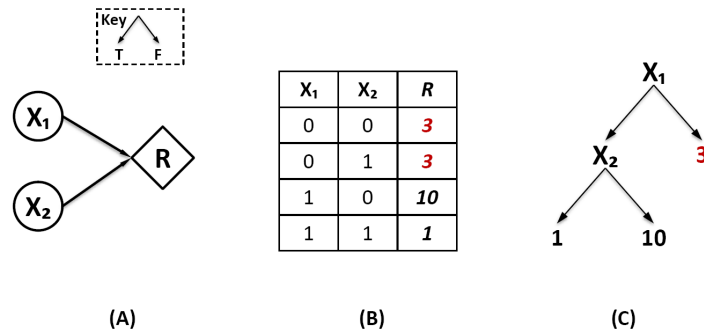


Figure 4.2: Reward function representation within the FMDP framework. (A) Dynamic Bayesian Network representing the dependencies of the reward from the variables. (B) Tabular representation of the reward values. (C) Decision Tree representation of the reward values.

while the tests on variables are simplified. For instance, the probability $P(x_2 = 1)$ of the future value of the variable X_2 in the state st given the current state $\{x_1 = 0, x_2 = 1\}$ is equal to the same probability given the state $\{x_1 = 1, x_2 = 1\}$. Hence, it needs one leaf to be represented instead of two table entries, and the node testing the variable X_1 is simplified since its values are irrelevant. The same example holds for the reward function representation where the states given by $\{x_1 = 0, x_2 = 0\}$ and $\{x_1 = 0, x_2 = 1\}$ have the same reward value.

4.2 Structured Dynamic Programming

FMDPs can be solved by Structured Dynamic Programming (SDP) algorithms [Boutilier et al., 1995, Boutilier et al., 2000] or Linear Programming algorithms [Guestrin et al., 2002, Guestrin et al., 2003]. Here we focus on SDP algorithms.

Standard SDP algorithms such as Structured Value Iteration (SVI) and Structured Policy Iteration (SPI) use decision trees as factored representation. Thus, SVI and SPI can be seen as an efficient way to perform the Bellman-backup operation (Equation 4.1) on trees, expressed as follows:

$$Q^V(s, a) = R(s, a) + \gamma \sum_{st} P(st|s, a) V(st) \quad (4.1)$$

↓

$$Tree[Q_a^V] = Tree[R_a] + \gamma Tree[P_a^V V] \quad (4.2)$$

Therefore, the value and policy functions are represented in a decision tree form noted $Tree[V]$ and $Tree[\pi]$.

4.2.1 Tree manipulations

In order to be able to compute $Tree[Q_a^V]$ according to Equation (4.2), SVI and SPI use the following standard operations on decision trees:

- **Tree Simplification:** the process of removing any redundant or incompatible interior nodes in a tree (i.e., meaningless splits) as illustrated in Figure 4.3.

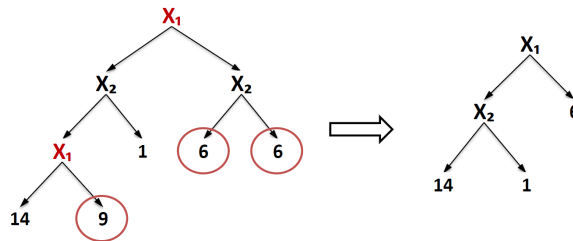


Figure 4.3: A tree simplified by removal of incompatible test $V(x_1x_2\bar{x}_1) = 9$ and redundant nodes $V(\bar{x}x_2) = V(\bar{x}\bar{x}_2) = 6$.

- **Appending Trees:** the process of extending a tree with the structure of another tree, where the new leaves are labeled using one of the three possible combination operations: sum, maximization or union. It is possible to append a tree to a particular leaf of to the entire tree, that is appending a tree to each leaf of the initial tree followed by simplification process. Figure 4.4 gives an example of the appending trees operation, where the values of the leaves are appended using a maximization operator.

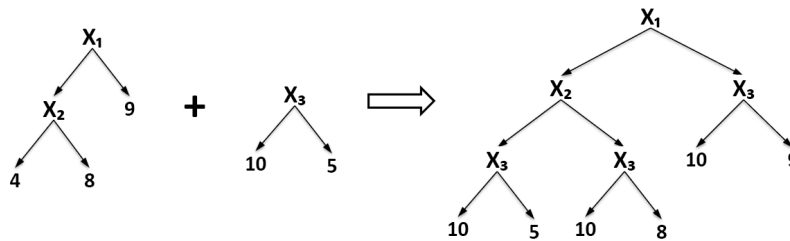


Figure 4.4: Appending trees using maximization as combination operator.

- **Merging Trees:** the process of producing a single tree from the set of trees. This can be accomplished by repeated appending of trees to the merge of the earlier trees. Figure 4.5 gives an example of the merging trees operation, where the values of the leaves are appended using a maximization operator and redundant nodes are simplified. The policy tree where $Tree[\pi]$ is a result of merging $Tree[Q_a^V]$ using maximization operator over the distribution of values over actions contained in leaves of $Tree[Q_a^V]$. The leaves of $Tree[\pi]$ are label with the actions that have the maximum value.

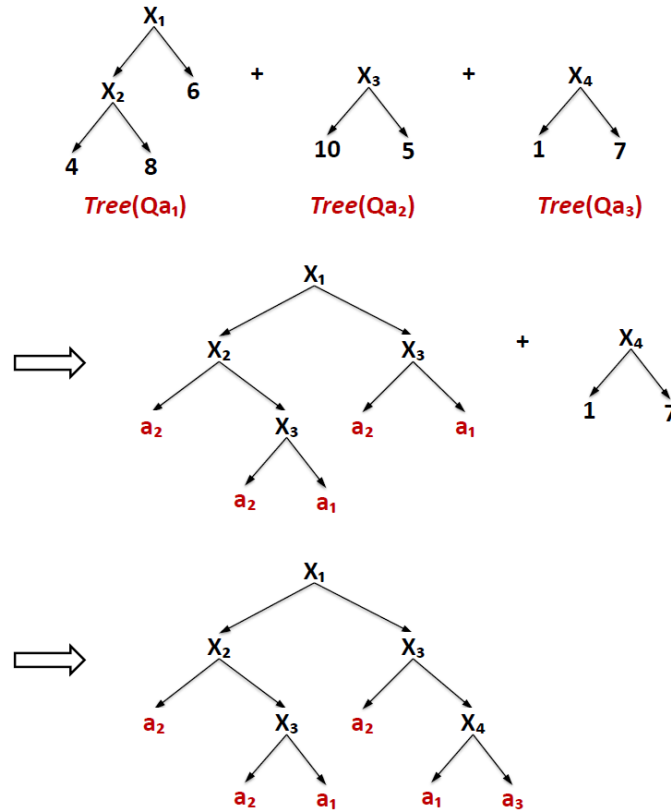


Figure 4.5: Merging trees $Tree[Q_a^V]$ to obtain $Tree[\pi]$ using maximization as combination operator and simplification of incompatible and redundant nodes.

4.2.2 Decision-Theoretic Regression

SVI and SPI [Boutilier et al., 2000] are structured version of Value and Policy Iteration algorithms given in Section 2.2.2 and 2.2.1. They need to redefine the computation operators on values to manipulate decision trees. This process is called *Decision-Theoretic Regression* as a generalization of goal regression - a standard process of backpropagation of the goal state values.

Thus, the Bellman-backup operator given in Equation 4.2 is performed by the $Regress(Tree[V], a)$ operation detailed in Algorithm 4.2. Inside $Regress$, $PRegress$, given in Algorithm 4.1, computes $Tree[P_a^V]$ using the structure of $Tree[P]$ and $Tree[V]$.

Algorithm 4.1: PRegress

input : $Tree[V]$, action a
output: $Tree[P_a^V]$

- 1 **if** $Tree[V]$ contains a single leaf **then**
 - └ **return** an empty tree $PTree(Q_a^V)$
- 2 $X \leftarrow$ the variable labeling the root of $Tree[V]$
 $Tree[P_a^V] \leftarrow Tree[P_a](X'|x)$ transition function of the variable X for action a (with leaves labeled by distributions over $val(X)$)
- 3 **foreach** $x_i \in val(X)$ **do**
 - (a) Let $Tree[V_{x_i}]$ be the subtree in $Tree[V]$ attached to the root X by arc x_i
 - (b) $Tree[P_a^{V_{x_i}}] \leftarrow PRegress(Tree[V_{x_i}], a)$
- 4 **foreach** leaf $l \in Tree[P_a^V]$, labeled with the probability P^l **do**
 - (a) $Tree_l = Merge(\{Tree[P_a^{V_{x_i}}] : \forall x_i, P^l(x_i) > 0\})$, using union as combination operator
 - (b) Append $Tree_l$ to the leaf l , using union as combination operator
- 5 **return** $Tree[P_a^V]$

Consider the toy example whose reward and transition functions for an unique action a_0 are given in Figure 4.6.

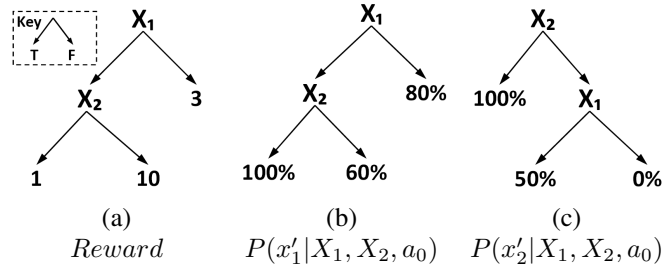


Figure 4.6: A toy example: the tree representation of a reward function and of the transition functions of binary variables X_1 and X_2 given an unique action a_0 . Notations are similar to the ones in [Boutilier et al., 2000], but in the boolean case we note x_i for $X_i = true$ and \bar{x}_i for $X_i = false$.

From the transition trees, $PRegress$ first computes the $Tree[P_{a_0}](X'|x)$ shown in Figure 4.7. This tree represents the individual probabilities of each variable value at $t + 1$ given the variable values at t . For instance, the rightmost branch of the tree reads as follows: if X_1 and X_2 were false, the probability that X'_1 and X'_2 are true are 80%

and 0% respectively. Note that the probability of one of the values can be omitted in the representation and inferred from the other probabilities. Furthermore, $PRegress$ only computes the combinations of values that are necessary to perform regression from the current value function.

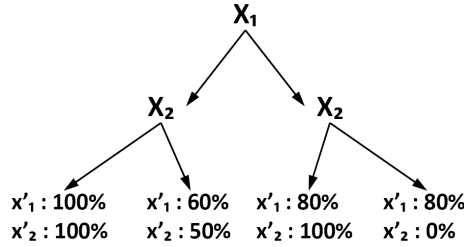


Figure 4.7: $P(X'|X, a_0)$ computed by the $PRegress$ operator from the example of Figure 4.6.

$P(X' X, a_0)$	$x'_1 x'_2$	$x'_1 \bar{x}'_2$	$\bar{x}'_1 x'_2$	$\bar{x}'_1 \bar{x}'_2$
$x_1 x_2$	100%	0%	0%	0%
$x_1 \bar{x}_2$	30%	30%	20%	20%
$\bar{x}_1 x_2$	80%	0%	20%	0%
$\bar{x}_1 \bar{x}_2$	0%	80%	0%	20%

Table 4.1: Probabilities for joint variables, resulting from Figure 4.7.

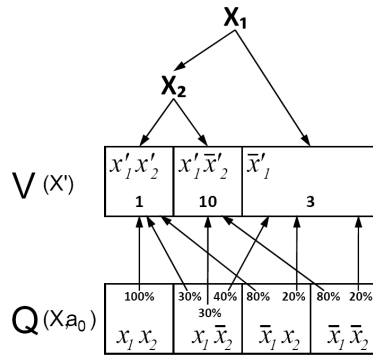


Figure 4.8: Regression with structured representations using the values computed in Table 4.1. We consider the first iteration where $Tree[V] = Tree[R]$. Notice that transitions to $\bar{x}_1 x_2$ and $\bar{x}_1 \bar{x}_2$ are summed to \bar{x}_1 .

Given the tree represented in Figure 4.7 and considering that the variables at $t + 1$ are independent conditionally to those at t , $PRegress$ computes the joint probabilities as a product, as shown in Table 4.1. Note that the table representation is not computed explicitly in the algorithm: in the more general case with any number of variables and enumerated values, this calculation is implemented by expanding a tree of all variable

values combinations and computing probabilities at the leaves as a product on individual probabilities.

The last step of *PRegress* computes the union of $Tree[P_a^{V_{x_i}}]$ for each leaf with $P(X' = x_i) > 0$ using the structure shown in Figure 4.8 and returns the transition probability tree $Tree[P_a^V]$.

Algorithm 4.2: Regress

input : $Tree[V]$, action a
output: $Tree[Q_a^V]$

- 1 $Tree[P_a^V] \leftarrow PRegress(Tree[V], a)$
- 2 Construct $FVTree[Q_a^V]$ as follows.
 - foreach** branch $b \in Tree[P_a^V]$ (with leaf l_b) **do**
 - (a) Let P^b be the joint distribution obtained from the product of the individual variable distributions labeling l_b
 - (b) Compute $v_b = \sum_{b' \in Tree[V]} P^b(b')V(b')$ where:
 - b' are branches in $Tree[V]$,
 - P^b is the probability of the conditions labeling that branch as given by the distribution P^b ,
 - $V(b')$ is the value labeling the leaf l_b in $Tree[V]$
 - (c) re-label leaf l_b with v_b
- 3 $Tree[Q_a^V] \leftarrow \gamma \cdot Tree[Q_a^V]$, (multiply each leaf label by the discount factor γ)
- 4 $Tree[Q_a^V] \leftarrow Append(Tree[R_a], Tree[Q_a^V])$, using addition as combination operator
- 5 **return** $Tree[Q_a^V]$

Then *Regress* computes $Tree[Q_a^V]$ according to Equation (4.2) by performing the product with γ and the sum with $Tree[R_a]$.

Algorithm 4.3: Structured Value Iteration (SVI)

input : $Tree[R]$
output: $Tree[V^*]$ and $Tree[\pi^*]$

- 1 $Tree[V^0] \leftarrow Tree[R]$
- 2 **repeat**
 - foreach** action a **do**
 - (a) $Tree[Q_a^{V^k}] \leftarrow Regress(Tree[V], a)$
 - (b) $Tree[V^{k+1}] \leftarrow Merge(Tree[Q_a^{V^k}])$, using maximization as a combination operator
- until** termination criterion
- 3 $Tree[\pi] \leftarrow Greedy(Tree[V_\pi])$
- 4 **return** $Tree[V^*]$ and $Tree[\pi^*]$

On top of *Regress*, SVI (Algorithm 4.3) and SPI (Algorithm 4.6) behave differently. In SVI, the value function $Tree[V]$ is computed by merging the set of action-value functions $Tree[Q_a^V]$ using maximization as combination function. It is shown in [Boutilier et al., 2000] that, given a perfect knowledge of the transition and reward

functions and starting with $Tree[V_0] = Tree[R]$, $Tree[V]$ converges in a finite number of time steps to the optimal value function $Tree[V^*]$. Then one can extract $Tree[\pi^*]$ from $Tree[V^*]$ using a tree-based greedy operator given in Algorithm 4.4.

Algorithm 4.4: Greedy

input : $Tree[V]$
output: $Tree[\pi]$

- 1 **foreach** action a **do**
- 2 $Tree[Q_a^V] \leftarrow Regress(Tree[V], a)$
- 3
- 4 $Tree[\pi] \leftarrow Merge(Tree[Q_a^V])$
- 5 **return** $Tree[\pi]$

Algorithm 4.5: Structured Successive Approximation (SSA)

input : $Tree[R], Tree[\pi]$
output: $Tree[V_\pi]$

- 1 $Tree[V_\pi^0] \leftarrow Tree[R]$
- 2 **repeat**
- 3 $Tree[V_\pi^{k+1}] \leftarrow Regress(Tree[V_\pi^k], a)$
- 4 **until** termination
- 5
- 6 **return** $Tree[V_\pi] \leftarrow Tree[V_\pi^n]$

In SPI, the process is slightly more complex. Policies $Tree[\pi]$ and value functions $Tree[V^\pi]$ are computed iteratively until convergence using the SSA operation given in Algorithm 4.5, making profit of the structure of $Tree[\pi]$ to optimize the computation of $Tree[V^\pi]$.

Algorithm 4.6: Structured Policy iteration (SPI)

input : $Tree[R]$
output: $Tree[V^*]$ and $Tree[\pi^*]$

- 1 $Tree[\pi'] \leftarrow Tree[\pi]$
- 2 **repeat**
 - (a) $Tree[\pi'] \leftarrow Tree[\pi]$
 - (b) $Tree[V_\pi] \leftarrow SSA(\pi)$
 - (c) $Tree[\pi'] \leftarrow Greedy(Tree[V_\pi])$
- until** $\pi' = \pi$
- 3 $Tree[\pi^*] \leftarrow Tree[\pi], Tree[V^*] \leftarrow Tree[V_\pi]$
- 4 **return** $Tree[V^*]$ and $Tree[\pi^*]$

4.2.3 SPUDD

SPUDD [Hoey et al., 1999] stands for Stochastic Planning Using Decision Diagrams. It uses the Algebraic Decision Diagrams (ADDs) instead of decision tree to represent the transition function of FMDPs. This results in a more compact representation and consequently it accelerates the computations. Then, in the same way as SVI, SPUDD is based on the Value Iteration algorithm, adapting it to ADDs where all the states variables are binary. The algorithm was improved and simplified in the further work of [St-Aubin et al., 2000].

4.3 Factored Reinforcement Learning algorithms

The algorithms presented until here require the perfect knowledge of the reward and transition functions. But in practice, this knowledge is rarely available. In this section we present algorithms that address this issue.

Factored Reinforcement Learning (FRL) is a Model-based Reinforcement Learning approach to FMDPs where the transition and reward functions of the problem are learned. First, it uses the methods of Supervised Learning to construct the factored representation of the environment. In parallel, planning algorithms can use this representation to build the policy.

4.3.1 SDYNA and SPITI

An implementation of FRL is expressed in the SDYNA framework [Degris et al., 2006a, Degris et al., 2006b] as a structured version of the DYNA architecture [Sutton, 1991] (presented in Section 3.1). SDYNA integrates incremental planning algorithms based on FMDPs with supervised learning techniques building structured representations of the problem. The inner loop of SDYNA is decomposed into three phases:

- *Acting*: choosing an action according to the current policy, including some exploration;
- *Learning*: updating the model of the transition and reward functions of the FMDP from $\langle X, a, X', R \rangle$ observations;
- *Planning*: updating the value function $Tree[V]$ and policy $Tree[\pi]$ using one sweep of SDP algorithms.

SPITI (Figure 4.9) is a particular instance of SDYNA using ϵ -greedy as exploration method, the *Incremental Tree Induction* (ITI) algorithm [Utgoff et al., 1997] to learn the model of transitions and reward functions as a collection of decision trees, and

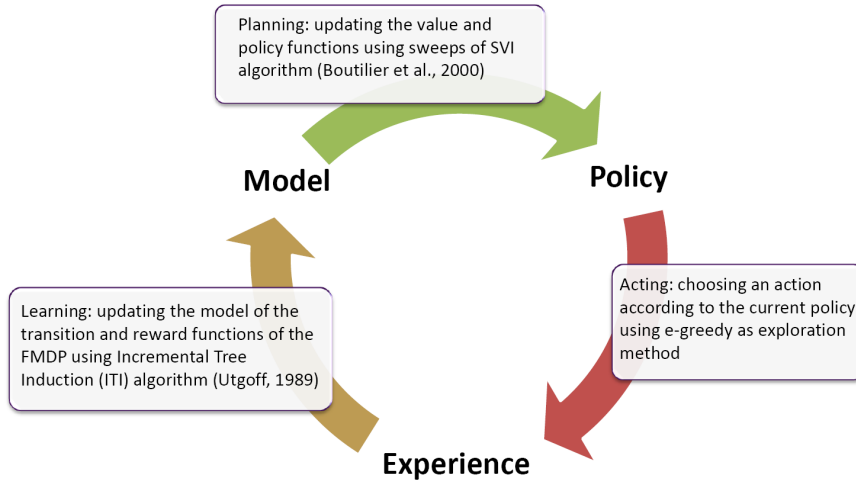


Figure 4.9: SPITI algorithm.

the inner loop of SVI as planning method. An algorithmic description is given in [Degris et al., 2006b]. We give a detailed account of the learning and planning steps hereafter.

4.3.2 Learning Factored Model

During the learning phase of the SPITI algorithm, the decision trees are built from the flow of examples, extracted from the current state of the environment. Each tree $Tree[P]$ quantifying the probabilities of transitions for a given variable X is built directly without building the corresponding DBN for each action. Thus it is possible to build one tree per variable and per action $Tree[P(X'|X, a)]$ or one tree per variable $Tree[P(X'|X)]$ where actions are represented as variables in decision nodes.

Algorithm 4.7: UpdateFMDP

input: FMDP $\mathcal{F} = \{\forall X_i \in X : Tree[P(X_i'|s)]\}, \forall R_i \in R : Tree[R_i]$

- 1 **foreach** $X_i \in X$ **do**
 - 2 example $e = \langle class \varsigma = st[X_i], attribute \alpha = \{s, a\} \rangle$
 - 3 $LearnTree(Tree[P(X_i'|s)], e)$
 - 4 **foreach** $R_i \in R$ **do**
 - 5 example $e = \langle class \varsigma = r[R_i], attribute \alpha = \{s, a\} \rangle$
 - 6 $LearnTree(Tree[R_i], e)$
-

The incremental learning procedure is given in Algorithm 4.7. It uses a classification algorithm to learn a function from a set of examples $\langle \mathcal{A}, \sigma \parallel$ with \mathcal{A} a set of attributes and

σ the class of the example. Consequently, from the observation of the agent $\langle s, a, s', r \rangle$ with $s = (x_1, \dots, x_n)$ and $s' = (x_1', \dots, x_n')$, the algorithm forms the examples $\langle \sigma = r, \mathcal{A} = (x_1, \dots, x_n, a) \rangle$ to learn the reward function and $\langle \sigma = x_i', \mathcal{A} = (x_1, \dots, x_n, a) \rangle$ to learn the transition function. The examples, formed at each time step, compose a stream that enables incremental learning.

Algorithm 4.8: LearnTree

```

input: information measure  $\chi^2$ , example  $e = \langle \varsigma, \alpha \rangle$ 
1 Let  $k$  be the current node
2  $\xi_k$  the examples in node  $k$ 
3 add  $\langle \varsigma, \alpha \rangle$  to  $\xi_k$ 
4 if  $k$  is Pure then
5   | return
6 if  $IsDiffSig(\chi^2, \forall v \in Dom(V_k), \xi_k)$  is False then
7   | transform  $k$  in a leaf
8 else
9   | attribute  $v = BestTest(\chi^2, \forall v \in V)$ 
10  | if  $v$  is null then
11  |   | return
12  | else if  $k$  do not test  $v$  then
13  |   | transform  $k$  in a decision node testing  $v$ 
14  |   | foreach  $\langle \varsigma_j, \alpha_j \rangle \in \xi_k$  do
15  |   |   | LearnTree( $\langle \varsigma_j, \alpha_j \rangle, k_{\alpha_j[v]}$ ), with  $k_{\alpha_j[v]}$  the child node of  $k$ 
16  |   |   |   | corresponding to a branch  $\alpha_j[v]$ 
17  |   | else
18  |   |   | LearnTree( $\langle \varsigma, \alpha \rangle, k_{\alpha[v]}$ ), with  $k_{\alpha[v]}$  the child node of  $k$  corresponding to a
19  |   |   |   | branch  $\alpha[v]$ 

```

The *UpdateFMDP* algorithm calls the *LearnTree* algorithm built upon the ITI algorithm. The procedure is detailed in Algorithm 4.8. The examples are classified in tree nodes corresponding to the attributes and class values. To determine the best test to install at a decision node, the algorithm uses an information-theoretic metric, namely χ^2 as suggested by [Quinlan, 1986]. In addition, in SDYNA it is possible to use other classification algorithms, such as ID3 and ID4 [Quinlan, 1986], C4.5 [Quinlan, 1993] or regression trees [Breiman et al., 1984].

4.3.3 Incremental Planning

The planning algorithm in SPITI is a modified incremental version of the SVI algorithm. In fact, using SVI without modifications in SPITI is not practical because it is not relevant to wait until convergence before building the policy, while the model is incomplete. That is why, at each time step, the IncSVI algorithm (Algorithm 4.9) performs only one

iteration of SVI and updates the corresponding ϵ – greedy policy used in the acting phase.

Algorithm 4.9: Incremental SVI (IncSVI)

input : $Tree[R], Tree[P]$
output: $Tree[V^*]$ and $Tree[\pi^*]$

- 1 **if** $Tree[R]$ changed during the last learning step **then** $Tree[V^0] \leftarrow Tree[R]$
- 2 **foreach** action a **do**
 - (a) $Tree[Q_a^{V^k}] \leftarrow Regress(Tree[V], a)$
 - (b) $Tree[V^{k+1}] \leftarrow Merge(Tree[Q_a^{V^k}])$, using maximization as a combination operator
- 3 $Tree[\pi^k] \leftarrow Greedy(Tree[V^k])$
- 4 **return** $Tree[V^*]$ and $Tree[\pi^*]$

As highlighted in [Degrís et al., 2006b], $Tree[V]$ converges in a finite number of time steps to the optimal value function $Tree[V^*]$ if the transition model and the reward function learned by the agent are stationary. However, as long as the model of the environment is incomplete, $Tree[V^*]$ may significantly differ from the optimal value function of the problem to solve.

4.4 Discussion & Summary

In this section we presented the Factored Markov Decision Processes (FMDP) and the solutions that can be used to solve this kind of RL problems. Structured Dynamic Programming algorithms such as Structured Value Iteration (SVI) and Structured Policy Iteration (SPI) use decision trees as factored representation. In the second part of the chapter, we presented Factored Reinforcement Learning (FRL) algorithms and particularly SDYNA where the transition and reward functions are learned from experience and then used in the incremental planning algorithm.

FMDPs address the “curse of dimensionality” problem by exploiting the similarities in the state space regions and internal dependencies between variables and the reward. This way, the irrelevant variables are ruled out while states are aggregated into sets or partitions. However, this approach does not scale well on some problems where the state space is very diversified i.e. it is difficult to aggregate states into sets using some similarities; or the inter-variable dependencies are very dense i.e. each state has to be represented by the entire scope of the variables.

In the next chapter, we present another approach to solving the “curse of dimensionality” based on the hierarchical decomposition of MDPs.

Chapter 5

Hierarchical Reinforcement Learning



At first glance, the idea of using hierarchies in artificial decision making seems obvious since humans naturally use hierarchical representations to act in the world. Besides, as mentioned in the previous chapter, many problems cannot be directly addressed by standard MDP and FMDP approaches because of large state space or weak factorization rate problem. Hence, it seems natural to try to use a hierarchical approach to decompose the problem into smaller pieces that are easier to solve individually. Indeed, many problems have a hierarchical structure where some subroutines can be reused in various conditions and/or some tasks have to be accomplished before the others. Therefore, automatically finding a hierarchical decomposition as a human would do, would be very helpful. In some cases, one can use hierarchical decompositions designed by a human operator. But in many situations such as large, complex or unusual problems, providing the correct decomposition is difficult if not impossible. That is why the automatic hierarchical task decomposition is one of the most challenging problems in the domain of RL.

In this section, we will first give some basic definitions for Hierarchical Learning in MDPs, followed by the state of the art algorithms.

5.1 HMDPS

Following the example of FMDPs, an HMDP is based on the idea of factorization, but brings that idea on a new level. From *state factorization* of the FMDPs, HMDP can look for *task factorization*, where a set of similar situations (defined by their goals) are represented by a partially defined set of independent sub-tasks. In other words, it is possible to simplify a problem by splitting it into smaller problems that are easier to solve individually, but also reuse the sub-tasks in order to speed up the global solution.

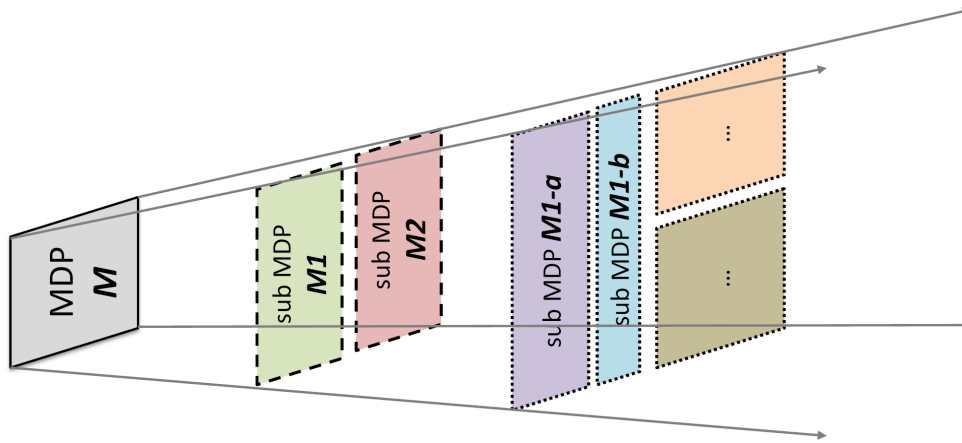


Figure 5.1: Hierarchical MDP.

In this respect, an HMDP \mathcal{M} is defined by a set of states, a set of actions and some reward and transition functions $\langle S, A, R, P \rangle$ exactly like an MDP, but it can additionally be decomposed into sub-MDPs \mathcal{M}_1 and \mathcal{M}_2 , that is $\mathcal{M} = \{\mathcal{M}_1, \mathcal{M}_2\}$ with $\mathcal{M}_1 = \langle S_1, A_1, R_1, P_1 \rangle$ where $S_1 \in S$, $A_1 \in A$ and R_1, P_1 reward and transition functions local to \mathcal{M}_1 . Similarly, \mathcal{M}_1 can be decomposed into sub-MDPs $\mathcal{M}_1 = \{\mathcal{M}_{1a}, \mathcal{M}_{1b}\}$ and so on. Figure 5.1 schematically represents the hierarchical decomposition. Note that this partition into sub-tasks is rarely as strict as shown in the figure, but may constitute several overlapping sub-spaces.

While this representation is visually similar to the Hierarchical Task Network (HTN) [Tate, 1977], where the planning problem is represented as a network of sub-tasks, the HMDP formalism is used to deal with the “curse of dimensionality” problem in the sequential decision problems that includes learning approaches.

The approaches to Hierarchical Learning imply the introduction of abstraction and hierarchical structure in the learning algorithms. That can be done in two ways:

- State/goal abstraction: that is ignoring differences between states based on the context by decomposing the state space into subsets and splitting the overall task into a set of sub-tasks;
- Temporal abstraction: that is grouping sequences or sets of actions together to form temporally extended actions with partial policies as with well-defined termination conditions.

If we take an example of a young lady who lives in Riga and wants to go to college in Paris, her project would be a complex sequence of actions. In order to succeed, she can first use state/goal abstraction to decompose her project into sub-tasks like “travel from Riga to Paris”, “register at university”, “convince parents” etc. Second, she can use temporal abstraction in order to simplify complex action sequences. For instance the travel from Riga to Paris is composed of the number of atomic actions like buying tickets, taking the bus to the airport, walking to the bus, packing luggage etc. All these actions can be represented by one macro-action “Go to Paris”. This way, the task can be structured and sub-tasks organized in a way that optimize the efforts and the expenses.

5.1.1 SMDPs & the Options framework

The notion of abstraction first was introduced with the Semi-MDP (SMDP) formalism [Korf, 1985a, Puterman, 1994]. The SMDP framework is a generalization over MDPs to the case where the number of time steps between one decision and the next is a random variable. SMDPs can be considered as the formal basis for Hierarchical Reinforcement Learning (HRL) algorithms. Several HRL approaches have been developed based on SMDPs.

- [Korf, 1985b] and [Laird et al., 1986] introduce macro-operators as a sequence of operators or actions that can be invoked by their name as if it were a primitive operator or action.
- Hierarchical abstract machines (HAM) [Parr and Russell, 1998] and programmable HAM [Andre and Russell, 2002] explores Hierarchical Learning with partially specified policies represented as a set of finite state machines defined by some constraints over the actions.
- MAXQ [Dietterich, 1998, Dietterich, 2000] is a standard framework for solving SMDPs when the transition and reward functions are known. MAXQ uses the MAXQ-Q algorithm based on HSMQ, that implements value function decomposition and state abstraction, given a handcrafted task hierarchy.

The one we focus on is the *options* framework [Sutton et al., 1999], [Precup, 2000]. The notion of an option can be introduced as a generalization of primitive actions

including temporally extended courses of actions. Even if options are added to the primitive actions set, resulting in a bigger core MDP representation, options provide a decomposition of the original task into subtasks, leading to a simplification of the global problem. Options can also facilitate the transfer of learned local policies to related tasks. As exemplified in a previous section, the temporal abstraction paradigm can be used to “shrink” a complex sequence of actions to one option “Go to Paris”, that can be reused independently of the person who goes to Paris, of the airline company and of the season.

An option o is a tuple $\langle \mathcal{I}, \pi, \beta \rangle$, where $\mathcal{I} \subseteq S$ is an initiation set, that is a subset of states in which it is possible to execute o , $\pi : S \times O \rightarrow [0, 1]$ is a policy executed in o , and $\beta : S \rightarrow [0, 1]$ is a termination condition function, that is the probability of terminating the option in each state. A primitive action $a \in A$ of the original MDP is also an option, called one-step option, with $\mathcal{I} = \emptyset$ and $\beta(s) = 1$. If the option is executed, then sub-options are selected according to π until the option terminates in state s' with probability $\beta(s')$. When the option terminates, the agent can select another option. Therefore the SMDP model is represented by a hierarchy of options, in which options on one level select their actions among options on a lower level.

In this perspective, an option o can be viewed as a subtask given by the option SMDP $\mathcal{M}_o = \langle S_o, O_o, \Psi, T_o, R_o \rangle$ where $S_o \in S$ is the option state set, O_o is the set of sub-options that o selects from, Ψ is the set of admissible state-option pairs, i.e. a set of pairs including states determined by the initiation sets of options in O_o , T_o is a transition probability function and R_o is the option reward function [Ravindran, 2004].

5.1.2 Hierarchical Reinforcement Learning algorithms

Over the last two decades, some major progress have been made in the Hierarchical Reinforcement Learning (HRL) domain, allowing the adaptation of the classical RL algorithms and creating new efficient approaches to solve hierarchical tasks. A non-exhaustive list of methods that can be found in literature is given below. First, we give the basic Q-LEARNING algorithms for the hierarchical approach, followed by a brief state of the art on the Hierarchical Learning algorithms, ending with a more detailed description of the methods that are directly linked to this thesis.

Q-LEARNING is one of the first RL algorithms adapted to hierarchical problems [Bradtke and Duff, 1995, Mahadevan et al., 1997]. The Hierarchical Semi-Markov Q-LEARNING (HSMQ), in its more or less modified form, is used by the such well known HRL methods like MAXQ or VISA presented here below. The pseudo-code of HSMQ is given in Algorithm 5.1. HSMQ introduces the idea of recursive policy amelioration by going down from the more abstract subroutines down to the primitive actions.

Another Q-LEARNING adaptation is developed by [Dayan and Hinton, 1993] as a feudal Q-LEARNING by recursively partitioning the state space and the time scale from one level to the next. In a similar spirit, indirect approaches like DYNA have been adapted

Algorithm 5.1: Hierarchical Semi-Markov Q-LEARNING (HSMQ)

```

Function: HSMQ
input   : state  $s$ , subtask  $p$ 
Init    :  $total\ reward=0$ 

1 while  $p$  is not terminated do
2   | Choose action  $a = \pi_x(s)$  according to the current policy  $\pi_x$ 
3   | Execute  $a$ 
4   | if  $a$  is primitive then
5   |   | Observe one-step reward  $r$ 
6   | else
7   |   |  $r=HSMQ(s, a)$ , which invokes subroutine  $a$  and returns the  $total\ reward$  received
7   |   | while  $a$  executed
8   |   |  $total\ reward = total\ reward+r$ 
9   |   | Observe resulting state  $s'$ 
10  |   | Update  $Q(p, s, a) = (1 - \alpha)Q(p, s, a) + \alpha[r + max_{a'} Q(p, s', a')]$ 
return :  $total\ reward$ 

```

as hierarchical-DYNA in [Singh, 1993]. Finally, [Ghavamzadeh, 2005] extends HRL methods to the multi-agent RL framework with Cooperative HRL- a hierarchical multi-agent RL algorithm.

5.1.3 Discovering the hierarchy

Among methods that perform autonomous hierarchical decomposition, some look for bottleneck states, the states that connect two or more strongly-connected regions. [McGovern and Barto, 2001] propose to discover such states by applying diverse density to successful trajectories, [Simsek et al., 2005] propose to discover the relative novelty metric that is based on visitation frequencies, while [Ravindran, 2004] and [Wolfe and Barto, 2006] develop state abstraction methods based on MDP homomorphisms. [Konidaris and Barto, 2009] introduce skill chaining, a skill discovery method for continuous domains that produces chains of skills leading to a salient event. Finally, [Zang et al., 2009] proposes another automated task decomposition method that uses a set of near-optimal trajectories to discover options and incorporates them into the learning process.

5.2 Combining HRL and FRL

FMDP and HMDP methods can be combined in order to decompose automatically the overall task into a hierarchy of factored subtasks. Autonomous subgoal discovery or “learning the hierarchy” in FMDPs consists in identifying subgoal states, generating temporally extended actions that take the agent to these states, and discovering

the hierarchical ranking between these temporally extended actions. The following algorithms are designed to discover the hierarchical structure of FMDPs.

5.2.1 HEXQ

HEXQ [Hengst, 2002] performs hierarchical decomposition by discovering subgoals corresponding to exit conditions in the state space. More generally, it is a hierarchical and model-free RL algorithm which automatically attempts to decompose and solve an FMDP. HEXQ discovers state abstractions and temporal abstractions by finding and exploiting repeatable substructures in the environment. To perform the hierarchical decomposition of an FMDP, HEXQ determines an ordering on the state variables based on the frequency with which the value of each variable changes. The state variable whose value changes the most frequently becomes the lowest variable in the ordering. For each state variable in the ordering, HEXQ identifies “exits”, that are state-action pairs $\langle s_e, a \rangle$ meaning that taking action a from state s_e causes an unpredictable transition where the action a : (i) causes the value of the next state variable in the ordering to change or (ii) causes the task termination. These exits identify the boundaries between Markov regions representing subtasks of the overall MDP. HEXQ defines a sub-MDP over each region with an exit as a transition to an absorbing state. The solution to this sub-MDP is a policy over the region that drives the agent out of an exit starting from any entry. Sub-MDP policies in HEXQ are learned on-line and the exit policies just learned become abstract actions at the next level. As the abstract actions can take varying primitive time durations to get executed, the task is represented as an SMDP that has less variables than the original MDP and only uses abstract actions.

5.2.2 VISA

Variable Influence Structure Analysis (VISA), is another algorithm that dynamically performs hierarchical decomposition of FMDPs [Jonsson and Barto, 2006]. In contrast with HEXQ, VISA determines causal relationships between state variables by building the state-variable graph of influence using a given model of an FMDP. This graph contains one node per state variable plus one node corresponding to the reward. A directed edge between two state variables or one variable and the reward node indicates that there is a relationship between them, i.e., that there is an edge between these nodes in the corresponding DBN for at least one action. To introduce options, VISA uses a formalism similar to the one used in the HEXQ algorithm but instead of frequency ordering, VISA is based on the state variable graph of influence, to represent variable relationships. To learn the policy of options, VISA uses model-free RL algorithms which do not require an explicit knowledge of the transition probabilities. A sketch of the VISA algorithm is given in Algorithm 5.2.

Algorithm 5.2: VISA

```

input: DBN model of the FMDP with set of state variables  $S$ 
1 construct the causal graph of the task
2 compute the strongly connected components of the causal graph
3 perform a topological sort of the strongly connected components
4 foreach strongly connected component  $SC \in S$  in topological order do
5   identify exits that cause the values of state variables in  $SC$  to change
6   while the number of exits exceeds a threshold do
7     merge  $SC$  with a parent strongly connected component
8     label the new strongly connected component  $SC$  and recompute the exits
9   foreach exit  $\langle c; a \rangle$  of the strongly connected component  $SC$  do
10    perform any possible exit transformations
11    compute the set  $Z$  of influencing state variables
12    construct an initiation set  $I$ 
13    construct a termination function  $\beta$  using the context  $c$ 
14    construct a policy tree by merging transition graphs of parent components
15    let  $S_o$  be the leaves of the policy tree
16    let  $O_o$  be the set of options that changes values of state variables in  $Z$ 
17    let  $\Psi$  be defined by the initiation sets of options in  $O_o$ 
18    let  $T_o$  be undefined
19    define  $R_o$  as  $-1$  everywhere except when the context  $c$  is unreachable
20    construct the option SMDP  $\mathcal{M}_o = \langle S_o, O_o, \Psi, T_o, R_o \rangle$ 
21    construct an exit option  $o = \langle I; \pi; \beta \rangle$ , where  $\pi =$  optimal policy of  $\mathcal{M}_o$ 
22  construct the transition graph of the strongly connected component  $SC$ 
23 construct a task option corresponding to the original task
24 use reinforcement learning techniques to learn the policy of each option

```

5.2.3 Incremental-VISA

In parallel with the work presented here, Incremental-VISA [Vigorito and Barto, 2008a, Vigorito and Barto, 2008b] adapts VISA to the case where the factored structure of the problem is not known in advance. The approach implements an agent that can learn incrementally and autonomously both the causal structure of the environment and useful skills that exploit this structure. First, it uses DBN structure learning techniques to learn the environment structure and then, SDP algorithms like SVI to build hierarchical policies online. The authors propose an active learning scheme to improve the efficiency with which this structure is acquired that bootstraps on existing structural and procedural knowledge. At this moment in time, this approach is limited to the deterministic case. Despite the fact that the incrementality property is the major benefit of this algorithm, it needs to rebuild the complete variable influence graph each time one of the DBNs changes and consequently to re-check the soundness of each option. In order to avoid introducing irrelevant or incomplete options into the planning procedure, Incremental-

VISA maintains a set of controllable variables, defining the reachability of each value of this variable. In this manner, an option is introduced only if the corresponding variable is completely reachable. As a result, this approach is not convenient for problems with a lot of variables having a lot of values. A more detailed discussion of Incremental-VISA is presented in Section 7.6.1.

5.3 Discussion & Summary

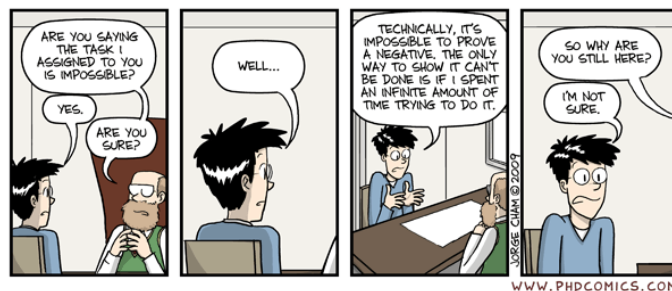
In this chapter, we presented hierarchical methods to solve MDPs and FMDPs. In this respect, an HMDP is a generalization over MDPs based on introduction of a hierarchical structure. The HMDP framework relies on the theory of SMDPs and can then exploit HRL methods to compute a policy. In brief, the HMDP theory addresses the “curse of dimensionality” problem through the hierarchical structure of complex decision problems. First, it gives the possibility to simplify the problem by decomposing the overall task into smaller sub-problems (e.g. options), that can be then reused in various conditions. Consequently, it can provide an important speed up of learning and planning. Some algorithms use handcrafted hierarchical decomposition while others perform autonomous subgoal discovery of the problem represented as an FMDP. Since transition functions and hierarchical decomposition are rarely available for the complex, large and real world problems, one of the most challenging research directions is the autonomous hierarchical decomposition of problems with an unknown structure. Another challenging research direction is to combine the advantages of the FMDP and of the HMDP formalisms into an HFMDP representation, that is hierarchically ordered set of imbricated sub-FMDPs.

Part II

Contribution

Chapter 6

Dealing with Impossible States: IMPSPITI



In Chapter 4 we explained how the FMDPs can be used to solve large discrete problems if their states can be represented by a set of independent variables. FMDPs exploit the internal structure of the problem in order to represent it compactly. In this representation the combinatorial of all the independent variables with all their values represents the maximal imaginable size of the problem. The efficiency of this approach comes from the fact that a large part of this information does not appear directly in the representation of the problem. It is hidden in the compact representation or simply does not exist.

Nevertheless, this second observation can generate a new problem called the problem of *impossible states*. In fact, when a combination of variables does not exist in the problem, nothing prevents its representation. These impossible states involve unnecessary computations and additional memory occupation. We can imagine several solutions to deal with this problem. First, do not use the FMDPs. But this approach does not solve the problem; it avoids it and the question of solving large problems using structured versions of RL methods is left opened. Second, it is always conceivable to use other Markov models, such as POMDPs, but, depending of the problem to solve, giving

rise to more complex and memory expensive algorithms. Third, we can use the existing FRL methods and just ignore those impossible combinations or remove them from the policy and value functions after the calculations are done. These methods are still sound but not efficient in problems with a lot of impossible states because they need to perform unnecessary computations and demand an extensive memory use. In return, our fourth idea is that we can deal with these impossible states by modifying FRL algorithms in order to prevent the appearance of the impossible combinations and consequently, improve the efficiency of the algorithm when facing problems that contains impossible states.

In this chapter, we present a new FRL algorithm that addresses a wider sub-class of FMDP problems where particular combinations of values of variables may not occur, giving rise to impossible states. We argue that this situation often happens in practice and we show through benchmark experiments that modifying standard algorithms to deal with such impossible states is more efficient than just ignoring this phenomenon.

More precisely, we show how the presence of impossible states can be modeled and how the SDP algorithms must be modified as a consequence. Then, we present the IMPSPITI algorithm adapted to the case where the structure and parameters of the FMDP are learned from experience. Finally, we examine through benchmark experiments the benefits that can result from our method when impossible states are present, and we conclude with a discussion on the benefits of our approach depending on the rate of impossible states, the limitations and the possibility to extend this work in different directions. In particular, we show that considering as impossible a state that has never been seen so far is an efficient heuristic to learn quickly in FMDPs.

This chapter corresponds to the following publications: [Sigaud et al., 2009] and [Kozlova et al., 2009b].

6.1 Impossible states in FMDPs

On the one hand, the authors of [Boutilier et al., 2000] propose to model the case where the variables are not independent by adding synchronic arcs between variables at $t + 1$, that is dependencies between post action variables. An example of synchronic arcs is given in Figure 6.1.

In this case, the solution requires the recording of the joint probabilities of all groups of variables that are connected by such synchronic arcs. These joint probabilities are then used to extend the resulting trees with these correlations. This results in more complex, slower and more memory-intensive algorithms, but that can deal with a much wider class of problems. A more detailed study of that case is presented in [Boutilier, 1997].

On the other hand, while using an FMDP representation of the problem, if no precaution is taken and, nothing forbids the representations of the corresponding impossible states

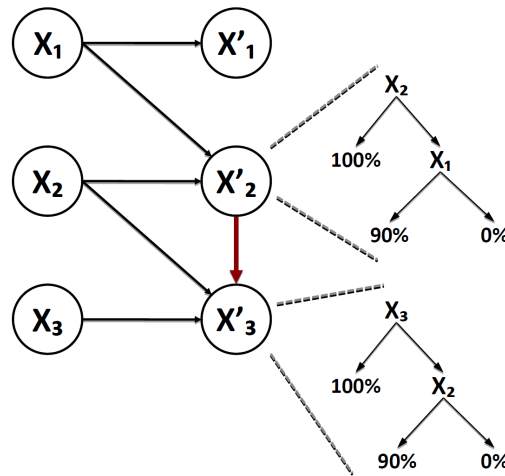


Figure 6.1: Example of synchronic arcs

if impossible combinations of variables values exist. In this chapter, we address the class of problems that is intermediate between the “no synchronic arcs” and the “any synchronic arcs” classes. It corresponds to problems where the variables at $t + 1$ behave as if they were independent, but some combinations of values for some variables do not occur in practice, which contradicts the independence assumption. We show how such a situation can be modeled without using the general class of problems with synchronic arcs in the DBNs.

6.2 The BLOCKS WORLD problem

We use the BLOCKS WORLD problem as an illustration throughout this chapter. In the version of BLOCKS WORLD, introduced in [Butz et al., 2002] (see Figure 6.2), b blocks are distributed over a given number s of stacks. At the beginning of each episode, the blocks are distributed randomly among “legal” locations, i.e. put on the table or on another block. The agent can manipulate the stacks by the means of a gripper that can either grip or release a block on a certain stack. Additionally, the problem contains a goal state, which consists in putting a particular number $y \leq b$ of blocks on the left handside stack. The stacks are not limited in height. Figure 6.2(d) shows the goal in the problem with $b = 4, s = 3, y = 3$.

The complexity of BLOCKS WORLD highly depends on the representation chosen to encode the states. We developed three such representations.

The first is the one used in [Butz et al., 2002]. We call it *Binary* representation. The agent perceives the current blocks distribution coding each stack with b binary variables. One additional variable indicates if the gripper is currently holding a block. In this

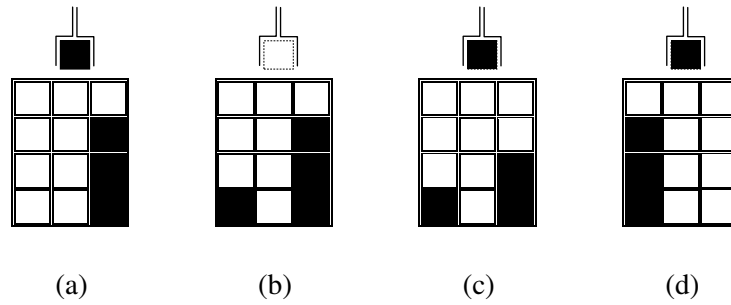


Figure 6.2: A BLOCKS WORLD scenario, from a random initial position (a) to the goal position (d).

(A)	(B)	(C)	(D)
BINARY			
0000,0000,1110,1	1000,0000,1110,0	1000,0000,1100,1	1110,0000,0000,1
STACKS			
STACK ₁ =0	STACK ₁ =1	STACK ₁ =1	STACK ₁ =3
STACK ₂ =0	STACK ₂ =0	STACK ₂ =0	STACK ₂ =0
STACK ₃ =3	STACK ₃ =3	STACK ₃ =2	STACK ₃ =0
G =TRUE	G =FALSE	G =TRUE	G =TRUE
BLOCKS			
BLOCK ₁ =S ₃	BLOCK ₁ =S ₃	BLOCK ₁ =S ₃	BLOCK ₁ =G
BLOCK ₂ =S ₃	BLOCK ₂ =S ₃	BLOCK ₂ =S ₃	BLOCK ₂ =S ₁
BLOCK ₃ =S ₃	BLOCK ₃ =S ₃	BLOCK ₃ =G	BLOCK ₃ =S ₁
BLOCK ₄ =G	BLOCK ₄ =S ₁	BLOCK ₄ =S ₁	BLOCK ₄ =S ₁

Table 6.1: BLOCKS WORLD representations of the situation given in Figure 6.2 (G stands for gripper).

representation, many arbitrary combinations of variable values correspond to states that do not occur in practice. Indeed, all states where a block is lying neither on top of another block nor on the table are impossible. The more empty cells in the problem, the more such impossible states.

In the second representation (called *Stacks*), there is one variable per stack giving the number of blocks it contains, and one additional variable indicating if the gripper is holding a block. The impossible states are the states where the total number of blocks is not b . Thus, in that case, an *ad hoc* way to decide if a state is possible consists in simply summing the represented blocks and comparing to b .

Finally, in the third representation (called *Blocks*), there are b variables whose values are given by the gripper or stack where the corresponding block is currently placed. The only impossible states are the ones where several blocks are in the gripper, which gives a straightforward *ad hoc* rule to decide if a state is possible. The major drawback of this representation is that, blocks being identical, there are many ways to represent the same state of the problem. For instance, $\{\text{block}_1=s_1, \text{block}_2=s_2, \dots\}$ and $\{\text{block}_1=s_2, \text{block}_2=s_1, \dots\}$ represent the same configuration and there are 12 different ways to represent Figure 6.2(c). This results in a greater number of possible states than necessary, thus in larger value and policy trees.

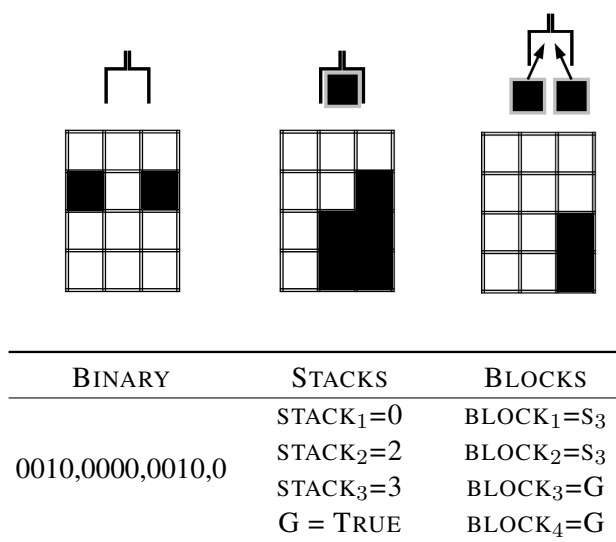


Figure 6.3: Variable combinations giving rise to impossible states in the BLOCKS WORLD problem.

Table 6.1 shows an example of these three representations with $b = 4$, $s = 3$, $y = 3$ and Figure 6.3 illustrates the impossible states that can be represented in BLOCKS WORLD problem such as the irrelevant number of blocks or incorrect position of blocks.

6.3 Modeling impossible states

The class of problems we want to address can be modeled with the kind of DBNs shown in Figure 6.4, given that there is one such DBN for each action.

In this representation, there are no synchronic arcs between variables at $t + 1$, but there are some constraints K on whether some combinations of variable values are possible or not. K (resp. K') stands for the knowledge of impossible states x (resp. x'). The values of K and K' are either *true* or *false* depending on the possibility of the corresponding states.

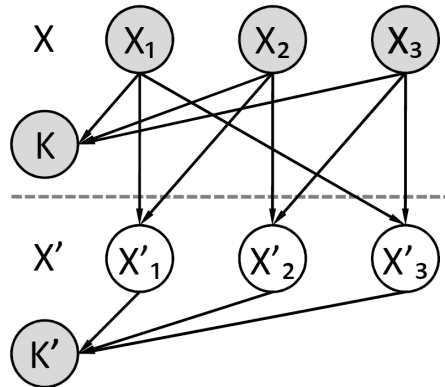


Figure 6.4: DBN representation for problems with independent variables and impossible states. K and K' stand for constraints stating if a particular combination of values is possible. Variables in gray are observed. K' and relevant X_j are used to predict X'_i for all i . K' does not necessarily depend on all X'_i .

As we illustrate in Section 6.5.3, without using such constraints, the $Tree[V]$ and $Tree[Q_a^V]$ structures may represent many states that do not occur in practice. Dealing with the constraints explicitly is a way to avoid the computational and memory overhead resulting from this useless information by filtering out all impossible states in the data structures.

We show below that this filtering can be performed safely just by discarding the impossible states and normalizing again the probabilities when it is necessary.

6.3.1 Impact on regression

Let us show that the values computed by *Regress* taking the constraints into account, i.e. $\sum_{x'} P(X'|x, a, k') \cdot V(x')$, are equal to the values one would obtain without taking these constraints into account, just leaving the impossible states away and normalizing again the probabilities.

First, with the representation above, the probability distribution of X' for a particular action a can be computed given the values of the variables x_i and the value k' of the common constraint K' . This probability distribution of X' knowing the value of x and k' is proportional to the product of the probability distribution of k' knowing the distribution of X' and values of x and the probability distribution of X' knowing x . Indeed, the distribution of X' can be expressed as

$$P(X'|x, k') \propto P(k'|X', x)P(X'|x).$$

Furthermore we have $P(K'|X', X) = P(K'|X')$, since $K' \perp\!\!\!\perp X|X'$.

Thus, we have:

$$P(X'|x, k') \propto P(k'|X')P(X'|x) \quad (6.1)$$

and $P(k'|X') = 0$ or 1 depending on the constraint.

If the variables are independent, we have

$$P(X'|x, a, k') = \prod_i P(X'_i|x, a) = \prod_i P(X'_i|\text{parent}(X'_i), a)$$

thus we are in the standard context where the proof of convergence given in [Boutilier et al., 2000] applies.

Now, if we consider impossible states, from (6.1) we have

$$\sum_{x'} P(X'|x, a, k') = \sum_{x'} N_{x,a} P(k'|X') P(X'|X, a) \quad (6.2)$$

where $N_{x,a}$ is a normalization factor such that

$$\forall x, \forall a, \sum_{x'} N_{x,a} P(k'|X') P(X'|X, a) = 1.$$

Then,

$$\sum_{x'} P(X'|x, a, k') = \sum_{x'} N_{x,a} [P(k'|X') \prod_i P(X'_i|\text{parent}(X'_i), a)]$$

and $P(k'|X') = 1$, thus

$$\sum_{x'} P(X'|x, a, k') = \sum_{x'} N_{x,a} \prod_i P(X'_i|\text{parent}(X'_i), a).$$

In equation (6.2), there are two categories of terms. If X' corresponds to impossible states, we have $P(k'|X') = 0$ and the corresponding term is removed. Otherwise, the state variables in X' are independent thus

$$P(X'|x, a, k') = \prod_i P(X'_i|\text{parent}(X'_i), a) \quad (6.3)$$

and $P(k'|X') = 1$.

Thus (6.2) can be simplified as

$$\sum_{x'} N_{x,a} P(k'|X') \prod_i P(X'_i|\text{parent}(X'_i), a)$$

where only the existing states remain. We are back to the situation where we consider only possible states with independent variables and the proof from [Boutilier et al., 2000] applies again.

From the result above, it turns out that, if we take the constraints into account, the values can be computed in *PRegress* as in the case without dependencies, just discarding the impossible states and normalizing again so that the sum of probabilities over all remaining states is 1.

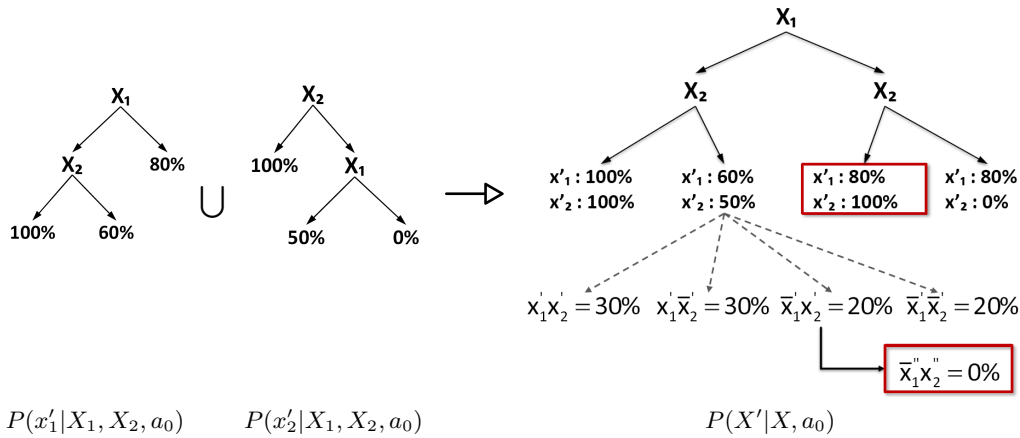


Figure 6.5: Removing impossible state \bar{x}_1x_2 probabilities in $P(X'|X, a_0)$ computed by the *PRegress* operator.

Note that this way to remove impossible states in the computation of *PRegress* is the only one which results in the possibility to renormalize. Otherwise, if, for instance, we remove the leaves corresponding to impossible states in $Tree[Q_a^V]$ or $Tree[V]$, we cannot perform the normalization anymore since the probability information is lost in these trees. Furthermore, in addition to being theoretically well-founded, this way of filtering out impossible states at the heart of the *PRegress* operator is much more efficient than filtering later on, since this other solution would result in expanding the number of leaves in the value tree before reducing it, which is exactly what we want to prevent.

$P(X' X, a_0)$	$x'_1x'_2$	$x'_1\bar{x}'_2$	$\bar{x}'_1x'_2$	$\bar{x}'_1\bar{x}'_2$
x_1x_2	100%	0%	0%	0%
$x_1\bar{x}_2$	30%	30%	20%	20%
\bar{x}_1x_2	80%	0%	20%	0%
$\bar{x}_1\bar{x}_2$	0%	80%	0%	20%

Table 6.2: Probabilities for joint variables, resulting from Figure 6.5.

To illustrate our approach, let us consider again the example given in the previous section. Now, assume that we have some information that the state \bar{x}_1x_2 is impossible (see boxed leaves in Figure 6.5). As a consequence, $P(\bar{x}_1'x_2')$ is null and the probability of any state at $t + 1$ given \bar{x}_1x_2 is pointless. Thus, the corresponding probabilities in Table 6.2 must be filtered out and the remaining values must be renormalized as shown in Table 6.3. Here again, Table 6.3 is not computed explicitly, it is represented as a tree as in the standard case, adding in the algorithm the filtering out of the branches corresponding to impossible states and finally normalizing again the values at the leaves.

$P(X' X, a_0, k')$	$x'_1x'_2$	$x'_1\bar{x}'_2$	$\bar{x}'_1\bar{x}'_2$
x_1x_2	100%	0%	0%
$x_1\bar{x}_2$	37.5%	37.5%	25%
$\bar{x}_1\bar{x}_2$	0%	80%	20%

Table 6.3: Transition probabilities for joint variables resulting from Figure 6.5, given that $\bar{x}_1'x_2'$ is impossible.

6.3.2 Impact on other tree operations

After modifying *PRegress* as presented above, the trees corresponding to Equation (6.2) are free from impossible states for all actions.

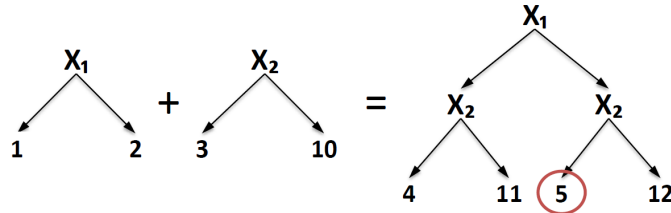


Figure 6.6: Combining two trees can generate leaves about impossible states (here, \bar{x}_1x_2 is impossible).

But then, performing a Bellman-backup according to Equation (4.2) and the other operations required to run SVI or SPI implies some operations over the resulting trees. As exemplified in Figure 6.6, despite the filtering performed in *PRegress*, simple operations on several value-related trees ($Tree[R]$, $Tree[V]$, $Tree[Q_a^V]$) can generate leaves representing impossible states if these trees do not share the same structure. Indeed, whereas generalization over identical values can “hide” the expression of impossible states in the source trees, the operations can make these states appear in the resulting trees. As a result, we must filter impossible states out in such operations.

6.3.3 IMPSVI

So far, we have described modifications that apply to SDP algorithms. To evaluate their impact, we modify SVI and develop a new algorithm called IMPSVI, that incorporates the following changes:

- in *PRegress*, when computing the joint probabilities over variable values, the branches in the tree corresponding to impossible states are discarded and the probabilities are normalized again;
- in the sum $Tree[R_a] + \gamma Tree[P_a^V V]$ and the maximization $Tree[V] = \operatorname{argmax}_a Tree[Q_a^V]$, the branches of the resulting trees corresponding to impossible states are discarded;

Algorithm 6.1 schematically describes IMPSVI.

Algorithm 6.1: IMPSVI

input : FMDP $\mathcal{F}\{Tree[P_a](x'|x)\}, Tree[V_t]$
output: $Tree[V^*]$ and $Tree[\pi^*]$

- 1 $Tree[V_0] \leftarrow Tree[R]$
- 2 **repeat**
 - foreach** action a **do**
 - (a) $Tree[Q_a^{V_t}] \leftarrow \text{addTrees}[Tree[R_a], \gamma.PRegress[Tree[V_t], a, N_{x,a}]$
discarding impossible states in $\text{addTrees}()$
 - (b) $Tree[V_{t+1}] \leftarrow \text{Merge}_a(Tree[Q_a^{V_t}])$, using maximization as a combination operator and discarding impossible states
- until** *termination criterion*
- 3 $Tree[\pi] \leftarrow \text{Greedy}(Tree[V_\pi])$
- 4 **return** $Tree[V^*]$ and $Tree[\pi^*]$

6.4 IMPSPITI

To evaluate the impact of these modifications in the context of FRL, we design a new system inspired from SPITI, called IMPSPITI (Figure 6.7), that incorporates IMPSVI in its *Planning* phase. The model of transition and reward functions being learned from experience, they cannot contain impossible states, thus the *Learning* phase does not need to be modified.

However, we need a function to decide that a state is impossible. Since this function is called at three places in each step of the central iteration, it may result in a significant time overhead. In the experimental section, we will compare two approaches. One consists in using problem specific expert rules. The second is more general, it consists

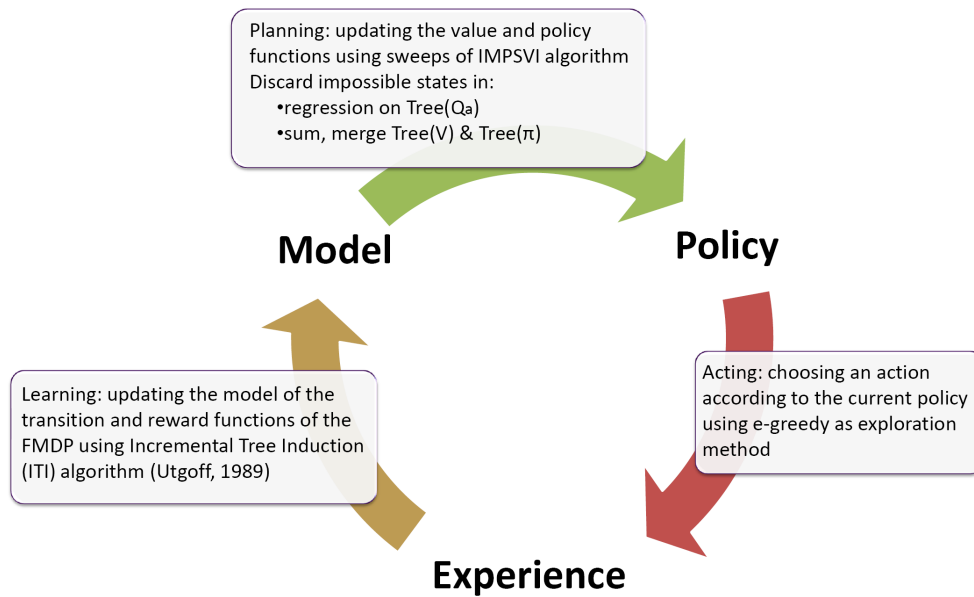


Figure 6.7: A global view of the IMPSPITI algorithm.

in building a tree where the states already visited by the agent are stored as possible as shown in Figure 6.8. This function can call upon a rule or a list of visited or non-visited states. In the worst case, this representation would boil down to the complete enumeration of states, but this is also true for the trees manipulated in standard SDP algorithms (see [Boutilier et al., 2000] for a discussion). In most cases of interest, however, this representation will benefit from factorization over states.

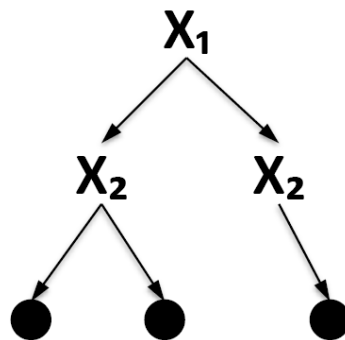


Figure 6.8: Visited states representation in a tree form. Here, states x_1x_2 , $x_1\bar{x}_2$ and $\bar{x}_1\bar{x}_2$ have been visited.

This function gives the list of the states that have been visited, by contrast with the list of states that may be constructed by the combinatorial combination of state variables. We use the heuristic of considering all states that have not been visited yet as impossible. In this case, the exploration rate is maintained by the ϵ -greedy exploration policy and unknown states are not included in the value and policy functions.

6.5 Experimental study

To illustrate the benefits of our approach, we perform experiments on two benchmarks: MAZE6 and BLOCKS WORLD problems.

The results presented here below are averaged over 20 runs of 100 episodes where each episode is limited to 50 steps. The curves are smoothed by computing the moving average weighted over ten neighboring values. The algorithms use the following parameters: $N = 300$ in DYNA-Q and $\epsilon = 0.1$ in ϵ -greedy. The ITI algorithm uses $\chi^2 = 30$ in the stochastic problems and $\chi^2 = 0$ in deterministic ones. The algorithms are coded in C# and run on Intel Core2Duo 1.80GHz processor with 2Go RAM.

We start by evaluating the performance and convergence speed of the algorithms followed by the comparison with tabular algorithms.

6.5.1 The MAZE6 problem

MAZE6 environments are standard benchmark problems in the Learning Classifier Systems (LCSS) literature, LCSS being a heuristic approach to FRL (see [Sigaud and Wilson, 2007]). The maze environments are also among the worst cases for FMDP algorithms given their poor structure. We choose it in order to test the robustness of our algorithms. Mazes are represented by a two-dimensional grid. Each cell can be occupied by an obstacle, denoted as variable value by a '1', a reward, denoted by a 'R', or can be empty, denoted by a '0'. The agent perceives the eight adjacent cells starting with the cell to the north and coding clockwise. Figure 6.9 shows MAZE6, one of such mazes, designed so that Markov property holds.

For example, an agent located in the cell below the reward perceives 'R1110011' whereas an agent located as shown in Figure 6.9 perceives '00110101'. Although there are only 37 actual states within the problem, the combinatorial representation results in $3^8 = 6561$ states. The agent can perform eight actions, the movements to adjacent cells. If a movement leads to a cell containing an obstacle, the action has no effect and there is no penalty. In the stochastic case, the chosen action may result in a move corresponding to an immediately adjacent action, with probability 10%. Once the reward position is reached, the environment provides a reward of 10 and the episode ends. In that case, the agent starts again in a randomly chosen empty cell.

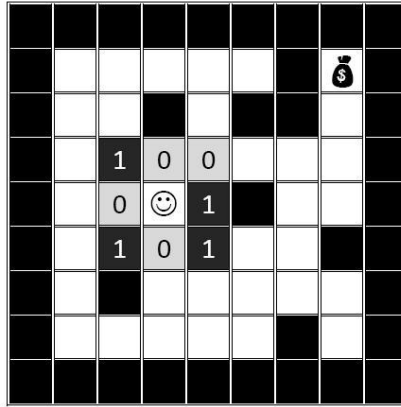


Figure 6.9: The MAZE6 problem.

6.5.2 Performance and convergence speed of IMPSVI

First, we evaluate the performance of IMPSVI, that is computing policy when the transition function is given. In all problems listed below, the policy obtained after convergence performs similarly with SVI and IMPSVI.

Table 6.4 recaps the performance on MAZE6 of SVI and IMPSVI respectively. IMPSVI clearly outperforms SVI: it converges faster in time and in number of iterations, but also requires less memory to represent value and policy functions.

	STEPS UNTIL CONVERGENCE	VALUE SIZE	POLICY SIZE	TIME (SEC) PER STEP
SVI	95.1 ± 1.2	242	240	1.9 ± 0.04
IMPSVI	14.4 ± 1.2	37	37	1.07 ± 0.09

Table 6.4: MAZE6 performance.

To evaluate IMPSVI in the BLOCKS WORLD benchmark, we tried five different sizes with the three representations described above. In Table 6.5, STEPS is the number of steps required to converge, using the span semi-norm with $\epsilon = 0.1$ as stopping criterion (see [Boutilier et al., 2000]). TIME(SEC) is the time required to perform one sweep of value propagation using trees to represent impossible states (the time with the *ad hoc* rules described in Section 6.2 is indicated between parentheses). The number between parentheses in the VALUE SIZE lines gives the total number of states considered in the problem, the rate of impossible states can be derived from a comparison of that number for SVI and for IMPSVI.

Table 6.5 shows that, with the *Binary* representation, the number of steps varies little with SVI and IMPSVI, but the computation time increases dramatically with the size of the problem in SVI, not in IMPSVI. Thus IMPSVI can deal with much larger problems.

B-S-Y	BINARY		STACKS		BLOCKS	
	SVI	IMPSVI	SVI	IMPSVI	SVI	IMPSVI
3-3-3						
STEPS	71.6 ± 1.4	9.7 ± 1.75	15.4 ± 1.8	10.26 ± 1.5	10.1 ± 0.6	10.1 ± 1.5
VALUE SIZE	254(1024)	16(16)	67(128)	15(16)	64(64)	54(54)
POLICY SIZE	281	16	45	15	40	40
TIME (SEC)	0.4 ± 0.01	0.1 ± 0.02	0.05	0.055 (0.05)	0.08	0.11 ± 0.05 (0.08)
4-3-4						
STEPS	72.7 ± 1.3	14.2 ± 1.4	71.6 ± 1.8	12.9 ± 1.7	14.2 ± 1.6	13.7 ± 1.8
VALUE SIZE	1138(8192)	25(25)	101(250)	25(25)	255(256)	189(189)
POLICY SIZE	1541	25	141	25	177	120
TIME (SEC)	4.3 ± 0.3	0.17 ± 0.02	0.08	0.07 (0.07)	0.38 ± 0.06	0.5 ± 0.06 (0.27 ± 0.05)
4-4-3						
STEPS	74 ± 2	10.8 ± 1.4	71 ± 1.2	10.3 ± 1.5	51.4 ± 1.3	52.5 ± 2
VALUE SIZE	18427(2 ¹⁷)	55(55)	120(1250)	53(55)	619(625)	505(512)
POLICY SIZE	20004	55	416	53	475	383
TIME (SEC)	3030 ± 20	1.5 ± 0.2	0.2 ± 0.06	0.2 ± 0.03 (0.18 ± 0.03)	5.2 ± 0.2	10.6 ± 0.3 (4.4 ± 0.3)
5-4-4						
STEPS	-	13 ± 2.2	78.8 ± 1.8	13.5 ± 1.9	69.8 ± 2.2	73.3 ± 1.8
VALUE SIZE	-(2 ²¹)	90(90)	274(2592)	87(90)	3119(3125)	2255(2304)
POLICY SIZE	-	90	964	86	2438	1373
TIME (SEC)	-	5.4 ± 1	0.42 ± 0.02	0.4 ± 0.07 (0.3 ± 0.02)	1253 ± 98	254 ± 30 (91 ± 13)
6-5-5						
STEPS	-	21 ± 1.8	74.7 ± 3	17.9 ± 1.1	-	-
VALUE SIZE	-(2 ³¹)	334(334)	1175(33614)	331(334)	-(46656)	-(34375)
POLICY SIZE	-	334	12425	330	-	-
TIME (SEC)	-	367 ± 29	38.8 ± 2	7.06 ± 0.8 (0.8 ± 0.05)	-	-

Table 6.5: The BLOCKS WORLD problem: rate of impossible states and time to perform one step (in seconds). A “-” indicates that the value could not be obtained after three days of computation. IMPSVI uses trees to represent impossible states (the time with the *ad hoc* rules is indicated between parentheses).

With the *Stacks* representation, both the size of the problem and the rate of impossible states grow slower, thus the time difference between SVI and IMPSVI keeps small. Finally, the rate of impossible states is much smaller in the *Blocks* representation, making it possible to analyze the time overhead of our approach more precisely.

6.5.3 Performance and convergence speed of IMPSPITI

Second, we test the online performance in FRL context, when the transition model is learned simultaneously to the policy optimization.

Table 6.6 shows the performance of SPITI and IMPSPITI on deterministic and stochastic versions of the MAZE6 problem. In stochastic MAZE6, the agent chooses a random action in 10% of the decisions.

	Value size	Policy size	Time/step(sec)
SPITI det	214 ± 10	241 ± 4	1.7 ± 0.8
SPITI stoc	252 ± 14	240 ± 12	3.7 ± 1.2
IMPSPITI det	35 ± 2	35 ± 2	0.1 ± 0.08
IMPSPITI stoc	35 ± 2	35 ± 2	0.3 ± 0.1

Table 6.6: MAZE6 performance (cost in memory and time).

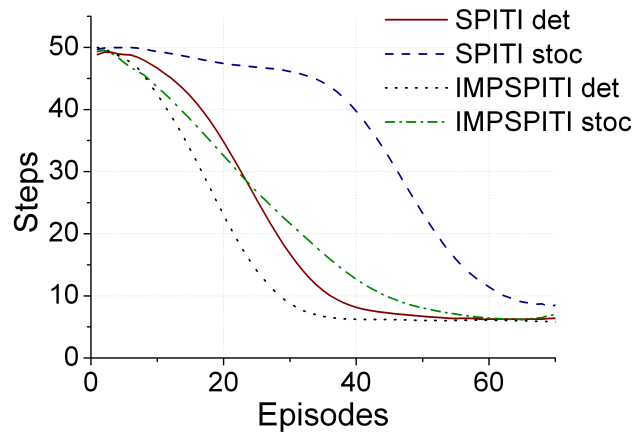


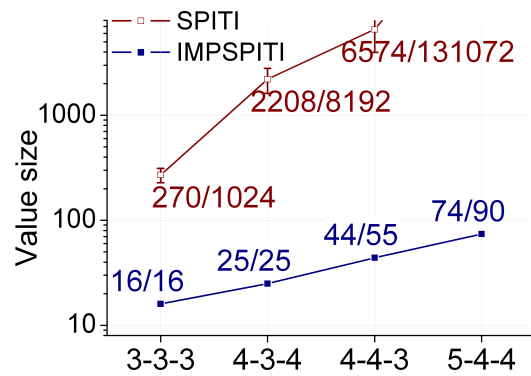
Figure 6.10: Convergence on the MAZE6 problem in number of steps needed to perform an episode as a function of the number of episodes.

As shown in Figure 6.10 and Table 6.6, IMPSPITI clearly outperforms SPITI both in the deterministic and in the stochastic case: it converges faster in time and in number of learning episodes, but it also requires less memory to represent the value and policy functions. More precisely, IMPSPITI only considers the 37 states that are actually possible (the variance in value and policy size comes from cases where the run ended before all states were explored).

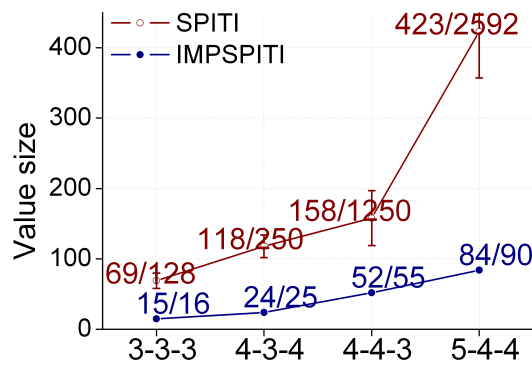
Figure 6.11 shows the value function size on different BLOCKS WORLD problems, computed as the number of leaves in $Tree[V]$ after 50 episodes.

Table 6.7 gives the rate of impossible states derived from the number of states shown in labels in Figure 6.11 and the time required to perform one learning step by SPITI and IMPSPITI respectively.

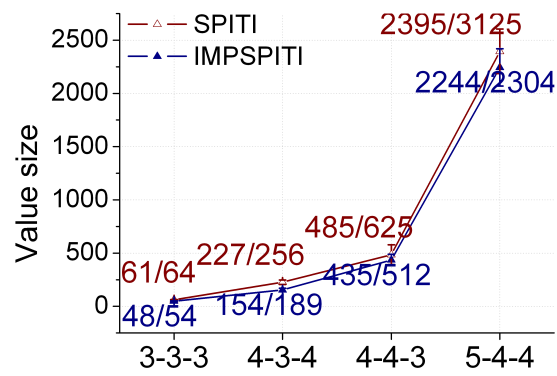
Figure 6.12 shows the convergence speed in number of episodes for the BLOCKS WORLD of size 4-3-4, that is a representative middle size problem. IMPSPITI takes less episodes than SPITI to reach the optimal policy with all representations. This result is explained by the fact that IMPSPITI uses smaller trees with a simpler structure, therefore



(A) BINARY

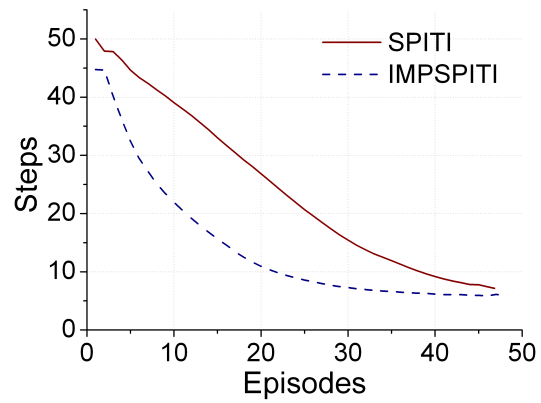


(B) STACKS

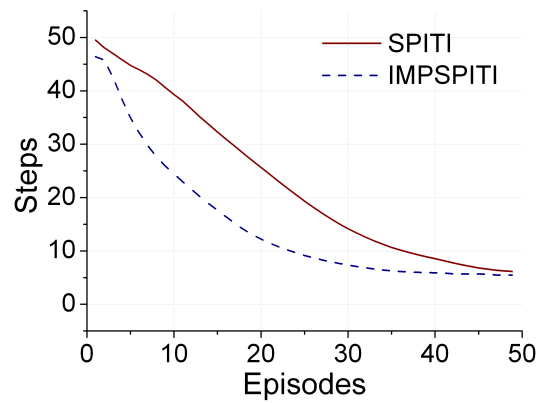


(C) BLOCKS

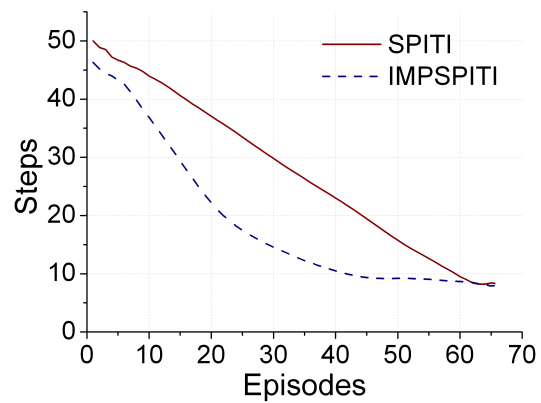
Figure 6.11: BLOCKS WORLD problem: value function size as a function of the size of the problem. Labels on points indicate this value / the total number of states considered. Note the log scale for *Binary*.



(A) BINARY



(B) STACKS



(C) BLOCKS

Figure 6.12: BLOCKS WORLD problem (size 4-3-4): performance along episodes. We run 70 episodes with *Blocks* to wait for convergence.

B-S-Y		BINARY		STACKS		BLOCKS	
		SPITI	IMPSPITI	SPITI	IMPSPITI	SPITI	IMPSPITI
3-3-3	% IMP.	98.5%		87.5%		15.6%	
	TIME	0.28 ± 0.03	0.04 ± 0.01	0.05 ± 0.02	0.01 (0.01)	0.04	0.03 (0.03)
4-3-4	% IMP.	99.7%		90%		26.1%	
	TIME	2.9 ± 0.4	0.08 ± 0.01	0.06 ± 0.02	0.01 (0.01)	0.17 ± 0.02	0.18 ± 0.01 (0.13 ± 0.01)
4-4-3	% IMP.	>99.99%		95.6%		18%	
	TIME	34 ± 7	1.2 ± 0.2	0.13 ± 0.06	0.02 (0.02)	0.34 ± 0.2	0.9 ± 0.07 (0.5 ± 0.03)
5-4-4	% IMP.	>99.99%		96.5%		26.3%	
	TIME	-	2.6 ± 0.3	0.2 ± 0.09	0.05 (0.04)	2.6 ± 0.3	6.5 ± 0.9 (3.1 ± 0.2)

Table 6.7: The BLOCKS WORLD problem: rate of impossible states and time to perform one step (in seconds). A “-” indicates that the value could not be obtained after three days of computation. IMPSPITI uses trees to represent impossible states (the time with the *ad hoc* rules is indicated between parentheses).

it takes less steps to propagate values over the trees. But, as expected, the difference is smaller when the rate of impossible states is smaller. Furthermore, even when there are very few impossible states as is the case with the *Blocks* representation, the policy improves faster with IMPSPITI.

Now, comparing the performance in time on a single step from Table 6.7, with the *Binary* representation, where the rate of impossible states grows very fast, IMPSPITI unquestionably outperforms SPITI in time and memory use. With the *Stacks* representation, both the size of the problem and the rate of impossible states grows slower, thus the time difference between SPITI and IMPSPITI keeps small. Finally, the rate of impossible states is much smaller in the *Blocks* representation, therefore both algorithms perform similarly for small BLOCKS WORLDS and SPITI takes less time per step than IMPSPITI as the problem size grows, due to the time overhead required to check for the existence of states. This is also true using *ad hoc* rules to detect impossible states, even if the overhead is smaller in that case.

6.5.4 Comparison with tabular algorithms

The maze environments are known to be one of the worst cases for the FMDP approach. Moreover, tabular algorithms perform well on such problems with a few possible states as they do not need to deal with impossible states. Thus, we want to compare the performance of both approaches.

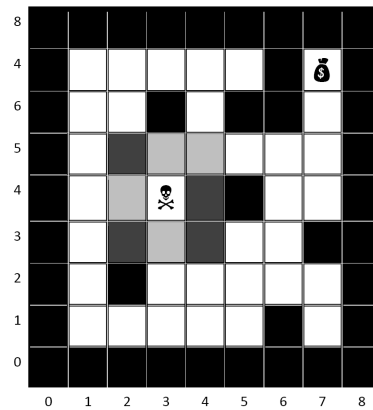



Figure 6.13: The D-MAZE problem.

With this aim in mind, we create a version of the MAZE6 problem that has a negative sub-goal (D-MAZE). If the agent moves to the specified “death” cell (represented by a ) , it receives a negative reward and has to restart from the random position. The zone surrounding the “death” cell is considered “dangerous” and also conveys a negative reward. The D-MAZE problem has the same number of states as MAZE6 (37 actual states, $3^8 = 6561$ states - with the combinatorial representation). In the “death” cell the agent receives a reward of -10 in the surrounding zone -5 and a reward of 10 in the goal cell. The experiments are performed on stochastic versions of MAZE6 and D-MAZE, that is agent chooses random action in 10% of the decisions.

We compare the performance of a tabular model based RL algorithm DYNA-Q (given in section 3.1.4) and IMPSPITI on MAZE6 and its modified version D-MAZE.

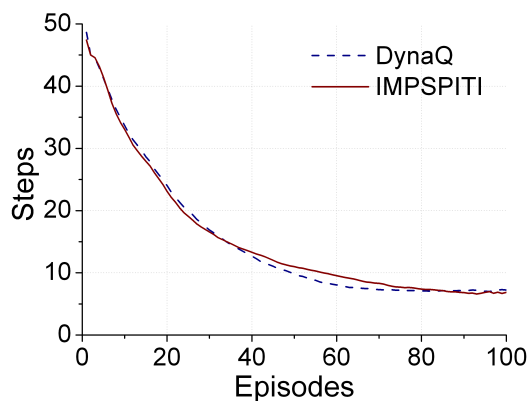


Figure 6.14: Convergence over episodes on MAZE6 with DYNA-Q and IMPSPITI.

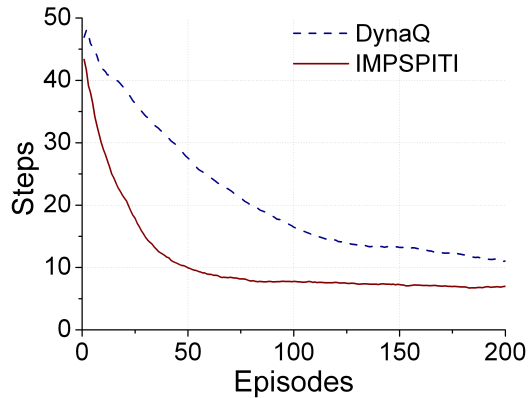


Figure 6.15: Convergence over episodes on D-MAZE with DYNA-Q and IMPSPITI.

		Policy size	Time/step(sec)
MAZE6	DYNA-Q	160 ± 20	0.02
	IMPSPITI	35 ± 1	0.3 ± 0.1
D-MAZE	DYNA-Q	160 ± 20	0.02
	IMPSPITI	35 ± 1	0.7 ± 0.2

Table 6.8: MAZE6 and D-MAZE performance (cost in memory and time) with DYNA-Q (in number of Q-functions) and IMPSPITI (in number of leaves in policy tree).

Figures 6.14 and 6.15 show the convergence in number of steps to complete each episode in MAZE6 and D-MAZE. Table 6.8 recaps the performance in policy size and execution time.

Unsurprisingly, on small problems like MAZE6 and D-MAZE, DYNA-Q outperforms IMPSPITI in time cost. As mentioned in Section 6.3, tree operations and functions discarding impossible states may result in a significant time overhead. Nevertheless, as the number of states in the problem grows, the algorithms based on tabular representations such as DYNA-Q cannot deal with explicit representation of all possible states, while the factored methods use compact representations, hence perform better in memory cost.

In order to compare the convergence speed in number of episodes as fairly as possible, DYNA-Q updates all state-action pairs. Doing so, the value update is propagated to all known states in both algorithms. IMPSPITI seems to perform better on the D-MAZE problem. This is because, in SPI and SVI, the value function is reinitialized every time the reward function changes. Thus, all the values are recomputed and reordered with the last reward function structure. This feature provides IMPSPITI algorithm with faster value propagation than DYNA-Q.

6.6 Discussion

The first message of this chapter is that, although using a factored representation in Reinforcement Learning results in the possibility to address larger problems, designing a factored representation so that it does not artificially increase the number of considered states may be very difficult. Consider the MAZE6 problem, where there are only 37 states, but a somewhat standard factored representation may result in 6561 potential states. This problem is an idealization of a standard robot navigation problem where, given usual robot sensors, one could not say in advance which sensory values cannot occur simultaneously. Similarly, if we take the BLOCKS WORLD problem, it proved difficult to design a representation that would fit the number of actual states. The *Blocks* representation results in fewer impossible states, but at the price of some redundancy that makes it very inefficient for larger problems (for instance, with a size 6-5-5 problem, we have 34375 possible states represented whereas there are only 334 actual states). We have shown that IMPSPITI is able to stick to the number of possible states of the problem given a representation, resulting in the possibility to address a much wider class of FMDPs than standard SDP methods.

Our second message is about the time overhead resulting from the necessity to check whether a state is possible. Considering the computation time per step and the number of steps required to converge, IMPSPITI always performs faster if there are enough impossible states. More precisely, as illustrated with the *Blocks* representation, the larger the problem, the larger the necessary rate of impossible states, since the tree of possible states will grow larger, resulting in a large overhead. However, using domain specific rules to detect impossible states generally results in a further gain in speed, but does so at the price of generality.

Finally, the fact that the policy improves faster with IMPSPITI than with SPITI even when there are very few impossible states, as is the case with the *Blocks* representation, tends to indicate that considering states as impossible until they are seen is an efficient heuristics even in the absence of actually impossible states. This heuristics itself deserves further analyses. In particular, it would be of much interest to study the potential interactions between this heuristics and using efficient optimistic exploration strategies such as “Optimism in the Face of Uncertainty” [Szita and Lőrincz, 2008] that drives the agent towards unseen states. At first glance, these heuristics are contradictory, since in the former we do not want to represent impossible states which we do not distinguish from unseen states, whereas in the latter we want to attribute a large value to unseen states, thus we need to represent them. One of the possibilities to combine the IMPSPITI approach with “optimistic” approaches can be achieved by giving positive values to non-explored state-actions pairs instead of states.

6.6.1 Limitations

As to the limitations, first the complexity of the algorithm depends on the representation of the impossible/possible states. In the best case, the problem relies on *ad hoc* rules to determine if one state is possible, the rate of impossible states is high as well as the factorization rate. In the worst case, the visited states must be registered, the rate of impossible states is low and the factorization rate is weak, that is each state is important and has a different value. Therefore, the impossible states representation has to be appropriate to the problem.

Second, considering unseen states as impossible results in a completely random exploration policy. That is the exploitation/exploration rate is fixed by the ϵ value of the ϵ -greedy exploration policy. In this way the unknown partitions of the state space are explored randomly, without taking into account the eventual importance of some directions. Since the policy is learned simultaneously, it would be interesting to adapt some active learning techniques like those proposed in [Strehl et al., 2007].

6.6.2 Extensions: Structured Prioritized Sweeping

Prioritized Sweeping (described in section 3.1.5) locally updates a subset of states previous to the state that causes the value function to change. Applied to SDP algorithms Prioritized Sweeping would be able to increase the speed of learning. [Dearden, 2000] proposed a structured version of Prioritized Sweeping (SPS), based on Generalized Prioritized Sweeping [Andre et al., 1997], that modifies the SPI algorithm to perform local decision-theoretic regression on a smaller part of the state space. In the new algorithm, the *PRegress* function takes an additional parameter ϕ that is a partial assignment of values to variables which define the set of previous states i.e. the parts of the tree that should be updated. For instance, a set ϕ_X is built adding the values that correspond to the possible ways of making $X = x$ by performing a_i . This is done by traversing the $Tree(P(x'|x)) \forall a_i \in A$ and for each leaf in which $X = x$ with non-zero probability, adding the corresponding values.

For example, in Figure 6.16 presented in [Dearden, 2001], if the value function has changed at the leaf $\bar{X}_3\bar{X}_2$, then $\phi_{X_2} = \{X_1\bar{X}_2, \bar{X}_1\bar{X}_2\}$ and $\phi_{X_3} = \{X_2\bar{X}_3, \bar{X}_2\bar{X}_3\}$, and the cross product is $\{X_1\bar{X}_2\bar{X}_3, \bar{X}_1\bar{X}_2\bar{X}_3\}$.

But if the generalized expression $\bar{X}_3\bar{X}_2$ “hides” an impossible combination $\bar{X}_1\bar{X}_2\bar{X}_3$ or if there is no transition between $X_1\bar{X}_2\bar{X}_3$ and $\bar{X}_1\bar{X}_2\bar{X}_3$, the method does not apply. Besides, the main point of prioritized sweeping is to update values of states that lead to the state bringing important change to the value function, but nothing guarantees that the sets of previous states, constructed by the method presented here above, actually corresponds exactly to all previous states. In fact, since the FMDP theory is built on the assumption of the independence of the variables and consequently uses compact

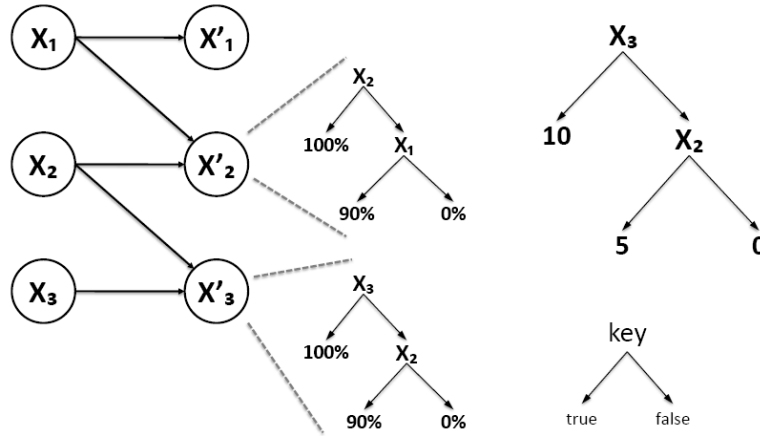


Figure 6.16: Action DBN, Conditional Probability Trees, Value and Reward function example.

representations of the set of states instead of individual states, it is impossible to reverse the representation to get individual previous states from the trees.

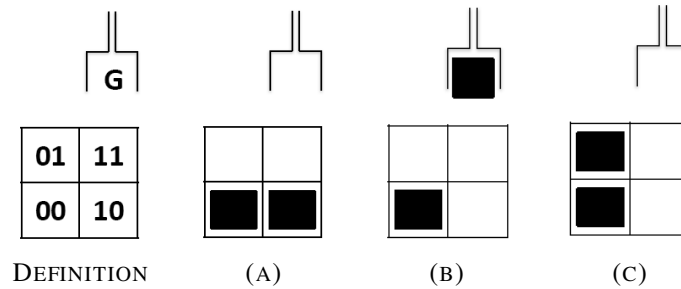


Figure 6.17: Binary 2-2-2 BLOCKS WORLD problem: example of possible states.

We modified SVI and IMPSVI algorithms as suggested in [Dearden, 2001] inside SPITI and IMPSPITI to perform tests on MAZE6 and BLOCKS WORLD problems and could not obtain any convergence. To illustrate this point, we take a small 2×2 BLOCKS WORLD example with 2 cubes. The variable definition and some states examples are given in Figure 6.17, where 00 means that there is a block in the 00 cell while $\bar{00}$ means the cell is empty.

Figure 6.18 shows the value tree and the transitions trees that are relevant for our example. If the value function has changed at the leaf $\{00, 10, \bar{11}\} = 309$ (that corresponds to the state (A) of Figure 6.17), then $\phi_{00} = \{00, \bar{11}\}$, $\phi_{10} = \{\bar{00}, 10\}$ and $\phi_{\bar{11}} = \{00, \bar{11}\}$, and the cross product is $\{00, \bar{11}\}$. The combination $\{00, \bar{11}\}$ corresponds to the states (B) and (C) of Figure 6.17, but the state (C) is not previous

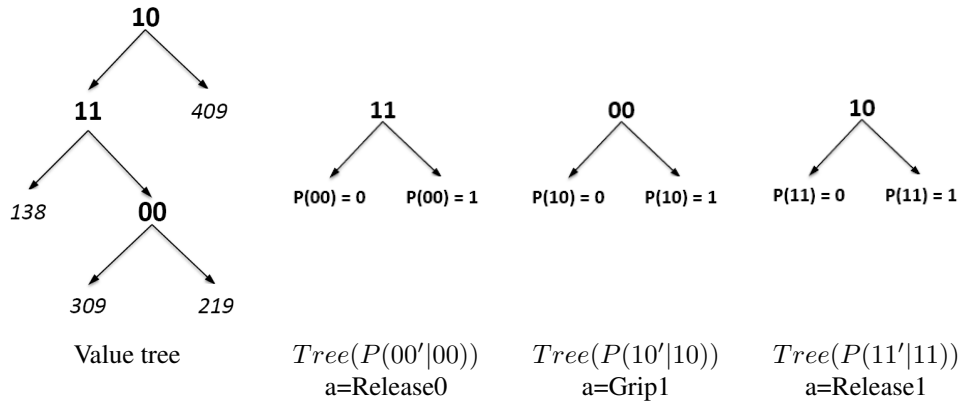


Figure 6.18: Binary 2-2-2 BLOCKS WORLD problem: value tree and transition trees.

to the state (A). There is no transition linking directly these two states. Thus, updating state (C) is not only irrelevant but also theoretically unsound, because this results in uncontrolled changes to the partition of the value function.

As shown in the example, in case of the presence of impossible states as well as in case of simultaneous structure and policy learning, the convergence of the algorithm is not guaranteed because of irrelevant updates in the value function that distort value propagation. Nevertheless, we believe that prioritized sweeping methods can improve the learning speed as soon as the correct way to update the relevant value function partitions is found. Thus we consider that as an open question for future work.

6.7 Conclusion

We have shown that there exists a practically relevant class of FMDPs between the “no synchronic arcs” and the “any synchronic arcs” classes that corresponds to problems where some combinations of state variable values do not occur. Though standard SDP algorithms such as SVI and their derivatives such as SPITI can be applied to this class of problems, we have shown that modifying the algorithm to take the presence of impossible states into account can result in significant performance improvements. Moreover, we have shown that, in the context where an agent has to explore its environment to learn its structure, considering as impossible the states that have not yet been encountered is also beneficial to the performance.

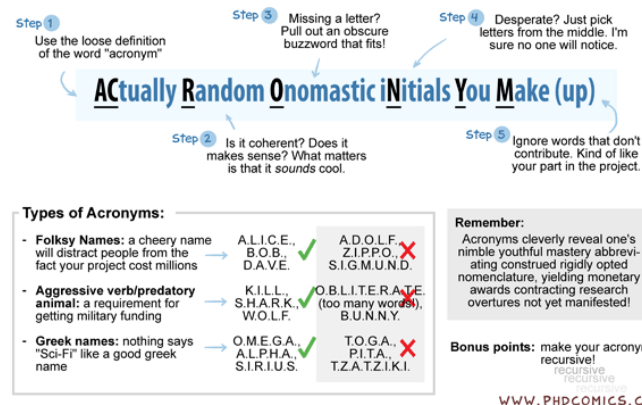
Note that the approach described in Section 6.3 can be applied to other standard SDP algorithms. Here, we focused on one particular FRL method, namely SPITI, but the impact on other instances of SDYNA using SPUDD or Guestrin’s linear programming approach [Guestrin et al., 2003] remains to be studied. Furthermore, it would be interesting to compare IMPSPITI with XACS [Butz et al., 2002], an Anticipatory

Learning Classifier System endowed with most FRL systems properties, but which uses genetic algorithm heuristics instead of SDP and incremental tree induction methods. A previous comparison between XACS and SPITI [Sigaud et al., 2009] has shown that XACS can deal efficiently with impossible states, but a closer comparison between the mechanisms of IMPSPITI presented here and those of XACS remains to be performed. Finally, the combination with “optimistic” approaches seems to be an interesting direction to explore in future work.

Chapter 7

TeXDYNA : hierarchical decomposition in FRL

Clever Acronyms: the Holy Grail of Academia



In previous chapters we presented the FRL, HMDP and SMDP formalisms and exemplified their utility to solve large stochastic and sequential decision making problems. First, we introduced the FMDP approach and showed how one can take advantage of the structure of the problem to solve large problems. Then, we introduced the HMDP approach that allows using hierarchical structure of the problem in order to address large and complex problems. Among various approaches to the hierarchical learning we singled out the options framework as one of the most efficient and elegant methods to introduce hierarchies into the learning and planning processes.

Several Planning methods use hierarchical representations, such as Hierarchical Task Networks or STRIPS. Besides, there are some hierarchical learning methods that identify

sub-goals or discover trajectory patterns. One of the challenging questions of the HRL domain consists in addressing both problems simultaneously. The goal is the automated discovery of the hierarchy that can be used directly in the planning phase. Such algorithms would be capable of online learning and consequently applicable to the problems with unknown structure. In order to address this question, we propose in this chapter to use existing FRL methods to solve large problems with unknown structure augmented with the hierarchical decomposition of the problem based on the options framework. The differences of this new approach with the existing ones will be highlighted in the Discussion section.

More precisely, we introduce *TeXDYNA*¹, a framework that combines the abstraction techniques of HMDPs to perform the automatic hierarchical decomposition of problems with an FRL method.

The combination of Hierarchical and Factored MDP formalisms corresponds to an HFMDP, that is hierarchically ordered set of imbricated sub-FMDPs.

TeXDYNA hierarchically decomposes an FMDP into an HFMDP by automatically splitting it into a set of options. Meanwhile, the local policy of each option is incrementally improved by a DYNA-like approach adapted to FMDPs [Degris, 2007] in the SDYNA architecture. The central contribution comes from the fact that the discovery of options and the construction of the model of the FMDP, as well as policy computation, are simultaneous. To achieve simultaneous SMDP structure learning, FMDP structure learning, local and global policy computation, *TeXDYNA* is built on top of SPITI [Degris et al., 2006b]. First, we make profit of the learning method used in SDYNA to learn only a local model for each option. As a result, the models are smaller and, therefore, easier to learn. Second, introducing options in the planning phase results in the possibility to plan over smaller partitions of the state-action space.

In the first place, the framework is designed to solve large problems using divide and conquer methods. Particularly, we take advantage of the HMDP techniques to decompose the overall task into smaller pieces easier to learn and plan individually using the internal structure of the problem. As presented in Chapter 5, hierarchical decomposition exploits the task structure by introducing stand-alone policies (activities, macro-actions, temporally-extended actions, options, or skills) that can take multiple time steps to execute. Particularly, the options framework, used here, introduces temporal abstraction methods into the problem representation that corresponds to “taking a short cut” in the path to goal. Furthermore, hierarchical decomposition methods are closely related to state abstraction provided by the FMDP framework, as it is, ignoring part of the available information to reduce the effective size of the state space.

From the other standpoint, while the FMDP representation technique reduces the size of the representation of state-action space of large problems by decomposing states into a

¹for *Temporally Extended SDYNA*

set of random variables, the HMDP representation decomposes the overall state-action space into a set of smaller state-action spaces each of which can be factored. While computing the policy with high level options, we obtain a factored structure of sub-spaces each of which is factored at its level. This “double factorization” with state-action space reduction techniques coming from FMDPs and HMDPs frameworks and represented by an HFMDP, provides a basis for solving more complex and larger problems.

First, in Section 7.2, we give a brief survey of the TeXDYNA algorithm. Then, in Section 7.3, we introduce the method for discovering options from the internal structure of the problem. Finally, in Section 7.4, we show how to take options into structured RL algorithms.

7.1 The LIGHT BOX problem

The examples to illustrate the algorithms presented in this chapter will be given on the LIGHT BOX problem.

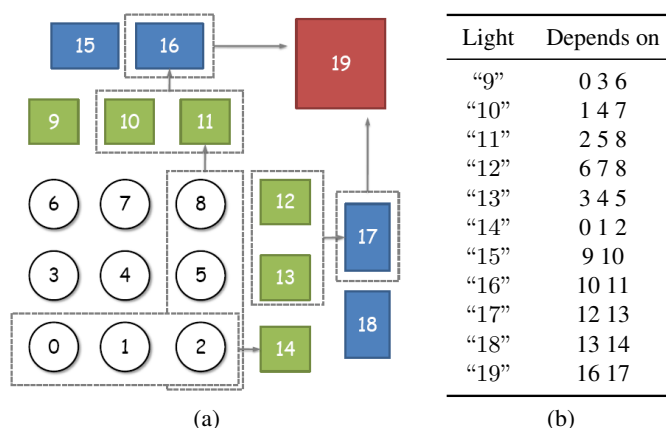


Figure 7.1: (a) The LIGHT BOX problem: number and color of “lights” with their dependencies. (b) Internal dependencies of the LIGHT BOX problem.

The LIGHT BOX domain proposed in [Vigorito and Barto, 2008b], presented in Figure 7.1 consists of a set of twenty “lights”, each of which is a binary variable corresponding to ON and OFF, named “0”, “1”, “2”, etc. Each light has a corresponding action that toggles the light ON or OFF. Thus there are twenty actions, $2^{20} \approx 1$ million states, thus approximately 20 million state-action pairs. The nine white lights are simple toggle lights that can be turned ON or OFF by executing their corresponding action. The green lights are toggled similarly, but only if certain configurations of white lights are ON, with each green light having a different set of dependencies. Similarly, the blue lights depend on certain configurations of green lights being ON, and the red light

depends on configurations of the blue lights. The goal is to turn the red light on, in which case the agent receives the reward of 20. The dependencies between lights, that we used in the experimentations, are given in Figure 7.1(b).

7.2 TEXDYNA: global view

Algorithm 7.1: TeXDYNA

input : FMDP \mathcal{F} , options hierarchy \mathcal{E}
output: option to execute o

- 1 **Learning:**
 - 1.a update global transition model
 $\mathcal{F} \leftarrow \text{UpdateFMDP}()$
 - 1.b discover options hierarchy
 $\mathcal{E} \leftarrow \text{UpdateOptions}(\mathcal{F})$
- 2 **Planning:**
 - 2.a update hierarchical policy
 - 2.b choose option to execute
 $o \leftarrow \text{SPITI WithOptions}(\mathcal{F}, \mathcal{E})$

In order to decompose hierarchically the overall FMDP into sub-FMDPs represented by options, TeXDYNA (Algorithm 7.1) builds a global transition function that represents the structure of the problem and uses this function to build an options hierarchy. For each option, TeXDYNA computes a local transition function and a local policy. This way, each option represents a part in the overall hierarchical decomposition, or else a sub-FMDP. Finally, the algorithm makes recursive calls like HSMQ down to the options hierarchy where each recursive loop follows 3 steps of SPITI (Figure 7.2).

Therefore, the TeXDYNA approach can be decomposed into two simultaneous processes:

1. Learning options: learning the transition function of the overall FMDP (updating the FMDP model with (s, a, s', r)) and updating options (Algorithms 7.2 and 7.3);
2. Planning with options: using a modified version of the SPITI algorithm for model learning, planning and acting with options, i.e. updating hierarchical policy $\pi = \langle \pi_{o_0}, \pi_{o_1}, \dots, \pi_{o_n}, \rangle$ (Algorithm 7.4).

7.3 Learning: Discovering options

During this phase of the TeXDYNA algorithm, the overall task transition function is learned in a decision tree form. The FMDP model provides the structure that will be used

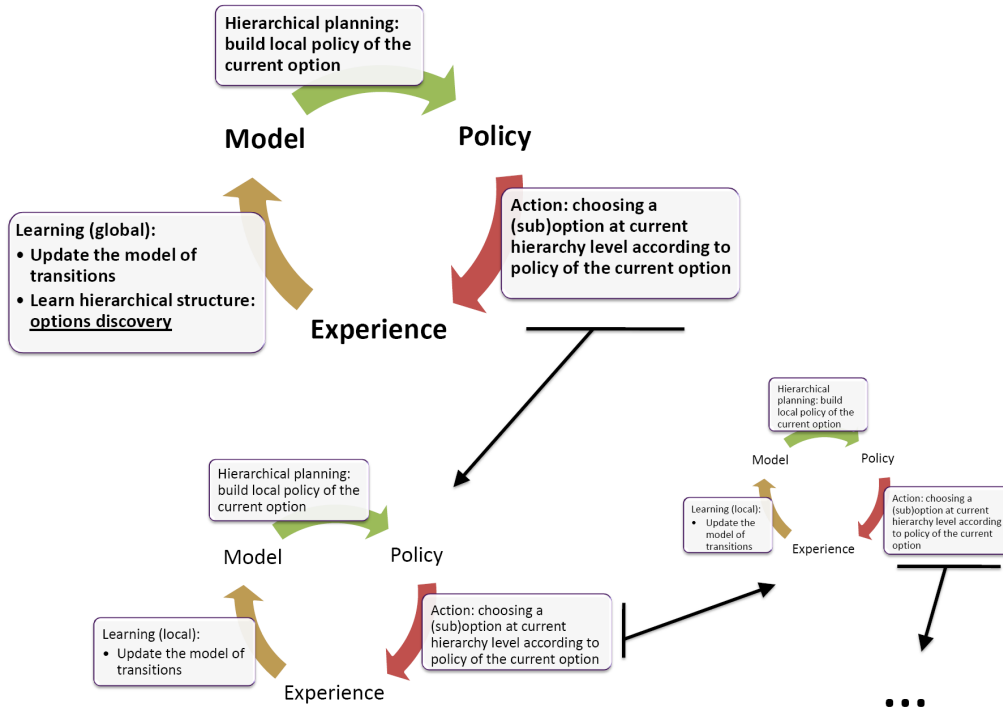


Figure 7.2: TeXDYNA global view.

for the options discovery process, as this structure represents the dependencies between variables and constraints under which those variables change their values.

Since the purpose of our approach is to decompose the overall FMDP into smaller subtasks or stand-alone policies that are mutually independent, we are interested in a representation that takes advantage of the FMDP structure introducing the temporal abstraction techniques. Among various SMDP formalisms, we use the options framework [Sutton et al., 1999]. As introduced in Section 5.1.1, options are a generalization of primitive actions including temporally extended courses of actions. Furthermore, we use specific options representation inspired from the goal-oriented exit options of VISA [Jonsson, 2006] and HEXQ [Hengst, 2002], where options are defined by their exit states that can be seen as sub-goals of the task.

Finally, each option represents a sub-MDP with its own structure and local policy. Particularly, a global model of transitions is used to discover options, then each option contains only a reduced model of transitions that is the local structure of the corresponding sub-MDP. In the next section, we explain how we introduce options into the learning and planning processes.

7.3.1 Introducing options

In order to highlight the exit oriented structure of options, we use a refined definition of an option o : $o = \langle \mathcal{I}, \pi, e \rangle$ where $\mathcal{I} \subseteq S$ is an initiation set, π is a policy executed in o and e is the related exit. Note that we do not use the termination function β (see Section 5.1.1) in the option definition since it is defined by the exit and computed at each time step during the option execution.

An exit corresponds to the changes of values of the variables linked to the reward function. We use an exit definition similar to the one used in HEXQ. However, unlike HEXQ, where exits are state-action pairs, we define exits as a tuple $\langle v, a, v_{ch}, c \rangle$, where:

- v is the variable whose value is changed by this exit,
- a is the exit action that makes the value of v change at the next state s' ,
- v_{ch} is a variable change, i.e. a pair of values $\langle x, x' \rangle$ where x is the value before a is executed and x' the value after a is executed. In the stochastic case, the variable change is a probability distribution over v_{ch} and the highest probability is kept in the exit definition.
- $c = \{x_1, \dots, x_n\}$ is the context, that is the set of constraints (i.e. assignment of values to a subset of state variables) that makes this exit available.

In this representation, the primitive actions have an empty context. Thus, to each action corresponds at least one option. Consequently there is at most one exit per action and per variable change and, consequently, each option corresponds to one unique exit.

The exit discovery procedure is slightly modified depending on the chosen transition trees representation ($Tree[P(x'|x)]$) - one tree per action per variable or $Tree[P_a(x'|x)]$ - one tree per variable, the action being an attribute). Below we give an example of exit discovery process for each representation.

Figure 7.3 shows the transition trees (one tree per variable) for the LIGHT BOX problem for variables representing the lights “16” and “11”. Note that according to the standard FMDP notation of the CPDs as $P[X'_i | \text{Parents}(X'_i), a]$, the transitions trees of Figure 7.3 should be noted as $Tree[P(\text{“16”}' | \text{“16”}, \text{“10”}, \text{“11”})]$ and $Tree[P(\text{“11”}' | \text{“11”}, \text{“2”}, \text{“8”}, \text{“15”})]$ but we keep the reduced notation in order to simplify the writing. For each transition tree, the algorithm iterates through its leaves and looks for the action attribute in the branch that changes the variable value. In this example the action *toggle16* is changing the value of the variable “16” from “OFF” (0) to “ON” (1) under the conditions represented by the nodes of the branch. Therefore, we introduce the exit

$e = \langle \text{“16”}, \text{toggle16}, 0 \rightarrow 1, \{ \text{“10”} = 1; \text{“11”} = 1 \} \rangle$ where

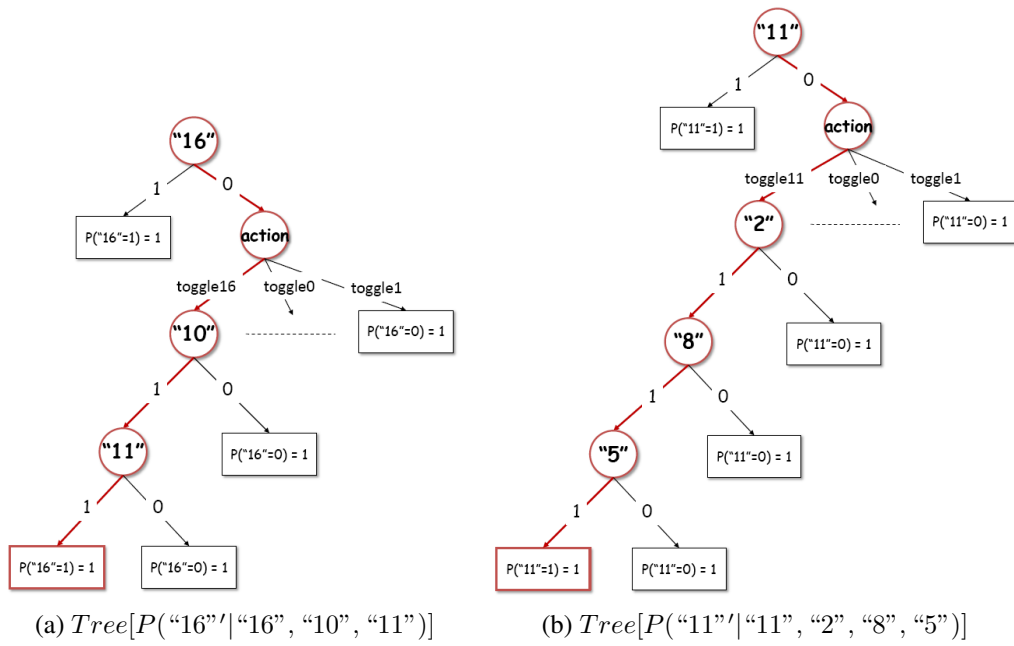


Figure 7.3: Transition trees for the variables “16” and “11” in the LIGHT BOX problem.

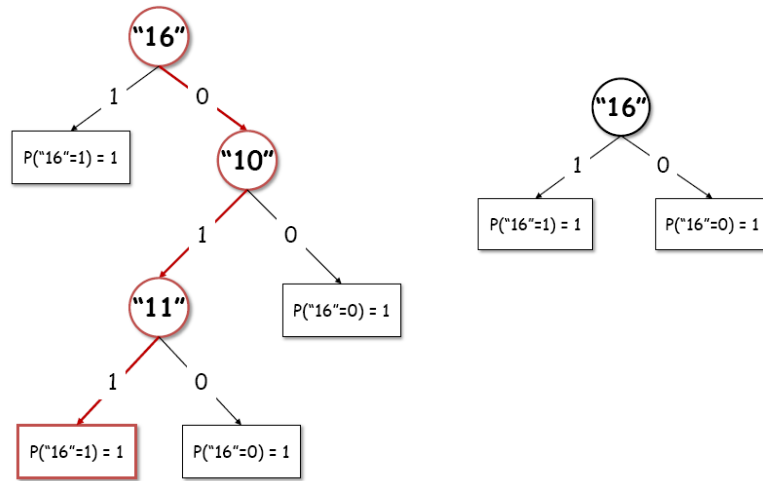
- $v = \text{"16"};$
- $a = \text{toggle16};$
- $v_{ch} = \langle 0, 1 \rangle;$
- $c = \{ \text{"10"} = 1; \text{"11"} = 1 \}.$

Similarly we define $e = \langle \text{"11"}, \text{toggle11}, 0 \rightarrow 1, \{ \text{"2"} = 1; \text{"8"} = 1; \text{"5"} = 1 \} \rangle;$

Figure 7.4 shows the transition trees (one tree per variable and per action) for a variable representing the light “16” and actions *toggle16* and *toggle2*. In this representation, the change in the variable values appears only in the $Tree[P(\text{"16"}' | \text{"16"}, \text{toggle16})]$. Thus we introduce the same exit $e = \langle \text{"16"}, \text{toggle16}, 0 \rightarrow 1, \{ \text{"10"} = 1; \text{"11"} = 1 \} \rangle.$

Finally, we introduce an option for each exit. An option is introduced using the following procedure:

1. Create (or update) an option o for each exit;
2. Initialize the local policy π_o ;
3. Add sub-options;
4. Compute the option rank, that is the place of the option in the hierarchy;



(a) $Tree[P_{toggle16}("16"|"16", "10", "11")]$ (b) $Tree[P_{toggle2}("16"|"16")]$

Figure 7.4: Transition trees for the variable “16” and actions *toggle16* and *toggle2* in the LIGHT BOX problem.

Algorithm 7.2: Update options (model of transitions: one tree per variable)

```

init : options set  $\mathcal{E} = \emptyset$ 
input: FMDP  $\mathcal{F} = \{\forall x_i \in X : Tree[P(x'|x)]\}$ 
1 forall the transition tree  $Tree[P(x'|s)] \in \mathcal{F}$  do
2   forall the leaf  $l$  of the  $Tree[P(x'|s)]$  do
3     if branch contains an action  $a$  &  $a$  modifies the value of the variable  $x$  in leaf  $l$ 
4       then
5         if  $\mathcal{E}$  does not contain a definition of option  $o$  with the exit corresponding to
6         variable  $x$  and action  $a$  then
7           introduce new option  $o$  defined by exit  $e : \langle v, a, v_{ch}, c \rangle$  with :
8           • variable  $v \leftarrow x$ 
9           • action  $a \leftarrow$  current action  $a$ 
10          • variable change  $v_{ch} \leftarrow \langle$  value in the branch, value in the leaf  $\rangle$ 
11          • context  $c \leftarrow$  variables of the branch that leads to the leaf  $l$ 
10        else if  $\mathcal{E}$  contains a partial definition of  $o$  then
11          update  $o$  with new information

```

5. Compute the Initialization set I_o .

Algorithms 7.2 and 7.3 describe the procedures for discovering options within the two representations of the model of transitions. Note the difference in line 3 where the exit action a is either defined by the transition tree or taken from the branch nodes. To ensure the relevance of discovered options, they are updated every time the model of transitions changes. Despite the fact that most of the options would be discovered earlier than the

Algorithm 7.3: Update options (model of transitions: one tree per variable per action)

```

init : options set  $\mathcal{E} = \emptyset$ 
input: FMDP  $\mathcal{F} = \{\forall x_i \in X, \forall a_j \in A : Tree[P(x'|x, a)]\}$ 
1 forall the transition tree  $Tree[P_a(x'|s)] \in \mathcal{F}$  do
2   forall the leaf  $l$  of the  $Tree[P_a(x'|s)]$  do
3     if action  $a$  modifies the value of the variable  $x$  in leaf  $l$  then
4       if  $\mathcal{E}$  does not contain a definition of option  $o$  with the exit corresponding to
       variable  $x$  and action  $a$  then
5         introduce new option  $o$  defined by exit  $e : \langle v, a, v_{ch}, c \rangle$  with :
6         • variable  $v \leftarrow x$ 
7         • action  $a \leftarrow$  current action  $a$ 
8         • variable change  $v_{ch} \leftarrow$   $\langle$ value in the branch, value in the leaf $\rangle$ 
9         • context  $c \leftarrow$  variables of the branch that leads to the leaf  $l$ 
10        else if  $\mathcal{E}$  contains a partial definition of  $o$  then
11          update  $o$  with new information

```

complete structure of the problem, some options might be incomplete or incorrect. The update procedure that handles this issue is discussed in Section 7.3.3.

7.3.2 Building the hierarchy of options

The set of discovered options \mathcal{E} is organized into a hierarchy according to the internal structure of the options that defines the dependencies between options. This internal structure is represented by the set of sub-options and the initiation set. Following the dependencies between options and sub-options, it is possible to represent the overall hierarchy of the task. Finally, the set of sub-options and the initiation set for each option define its corresponding sub-FMDP and consequently the structure of the global HFMDP. In this section we give the definition of sub-options and initiation set. Then we explain how these notions are used to build the hierarchy of options and how they impact the internal structure of options.

Sub-options

The sub-options are the options available in the sub-FMDP represented by the corresponding parent option. The sub-options are added in the following way: for each variable in the context of the option, if the set of options \mathcal{E} contains an option that modifies the value of this variable, this option is added to the set of sub-options. Note that, when computing the context of an exit, the exit variables are excluded from the context to avoid cross-dependencies between options. For example, the option defined by the exit $e = \langle \text{"16"}, toggle16, 0 \rightarrow 1, \{\text{"10"} = 1; \text{"11"} = 1\} \rangle$ will have two

sub-options *toggle10* and *toggle11*. In turn, the option *toggle11* defined by the exit $e = \langle \text{"11"}, toggle11, 0 \rightarrow 1, \{ \text{"2"} = 1; \text{"5"} = 1; \text{"8"} = 1 \} \rangle$ will have the following sub-options: *toggle2*, *toggle5* and *toggle8* as the actions that change the values of the variables "2", "5" and "8" respectively.

Since each primitive action is defined as an option (with an empty context), therefore there are only options in the problem definition. Then, when the definitions of these options are modified with new information we keep the same names in order to keep it simple.

This procedure supposes that there is one option per variable change and that each option changes at most one variable value. For instance, if some options change more than one variable, then one variable can be changed by more than one option and consequently these options would be sub-options of one another creating cycles in the hierarchy. Nevertheless this constrain can be relaxed by reorganizing the hierarchy once the options have been added. The procedure is the following: if a cross-dependency is detected between two options, the hierarchical link between them is removed and both options are attributed the lowest rank of the two. This way, it is possible to have more than one option that have the same exit variable.

Initiation set

The initiation set I_o of option o defines the state space where this option can be executed. On the one hand, it determines if the resources necessary for the successful execution of the option are available, and, on the other hand, it specifies the state space from where the exit of the option is reachable.

In practice, it is the union of its own exit context, the variable changes and exit contexts of its sub-options. If a sub-option is a primitive action, its exit variable is added to I_o with all possible values. Otherwise, the exit variable of the sub-option is added with its value change. By convention, a sub-option with an empty initiation set is admissible everywhere. This is particularly true for primitive actions. Therefore, all the values of the corresponding exit variables of these options are accepted. Thus, I_o contains all the states from which the exit of the option is reachable. This property of direct reachability is ensured by the fact that the exit context copies the constraints of the corresponding branch in the transition tree. In other words, the nodes of the branch represent the variables of the context. When those variables can be changed by a sub-option, the exit of the parent option becomes reachable from all the states where its sub-options are accessible. For instance, the Initiation set for the option defined by the exit $e = \langle \text{"16"}, toggle16, 0 \rightarrow 1, \{ \text{"10"} = 1; \text{"11"} = 1 \} \rangle$ is $I = \langle \text{"16"} = \{0\}; \text{"10"} = \{0, 1\}; \text{"11"} = \{0, 1\} \rangle$.

Hierarchical structure

The hierarchical structure of the set of options is determined by the variables interdependencies expressed in the structure of the transition function. However, for the purpose of soundness of planning algorithms, a rank is assigned to each option. In this way, the planning algorithm chooses an option to execute at each abstraction level in the hierarchy by going down from the most abstract options to primitive actions. The rank of the option is computed as the incremented highest rank of its sub-options.

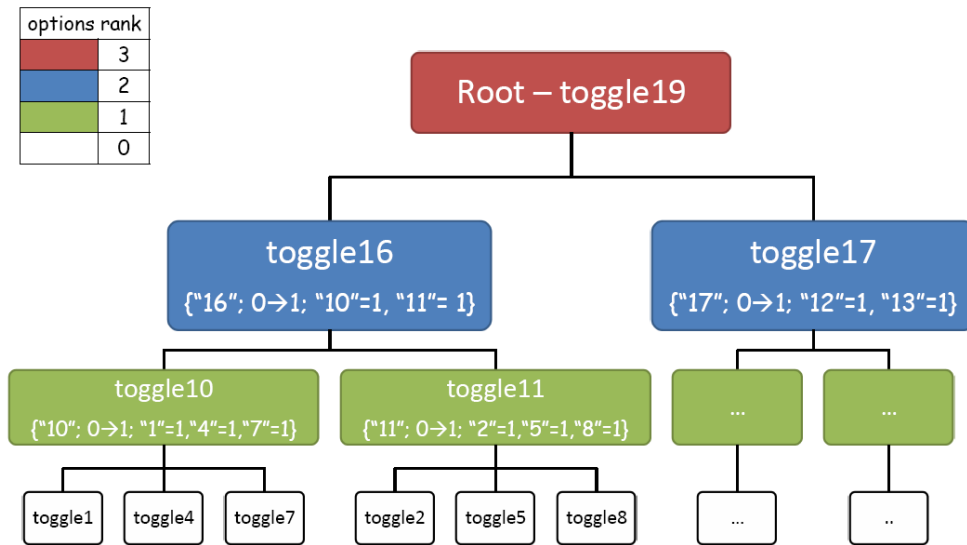


Figure 7.5: Example of the options discovered in the LIGHT BOX problem.

Since the learning process is online, both transition model learning and options discovery are simultaneous. Thus the options discovered first represent the transitions learned first and consequently have less constraints as they represent the most accessible sub-goals. If the problem has an hierarchical structure, this gives a bottom-up direction to the options discovery. First the options with the lowest abstraction level are discovered, then their execution gives access to more complex (i.e. having more constraints) options.

An example of options hierarchy obtained on the LIGHT BOX problem is given in Figure 7.5. The dependency of the variable “16” over the variables “10” and “11” is directly transposed from the transition trees shown in Figure 7.3.

The HFMDP structure of the options

Defined in this manner, each option represents a sub-FMDP $\mathcal{M}_o = \langle S_o, O_o, \Psi, T_o, R_o \rangle$ containing a reduced partition of the initial state-action space, where S_o is a set of context variables, O_o is a set of sub-options, Ψ is a set of admissible state-option pairs defined by

the initiation set, T_o is the local transition function and R_o is the local reward function. In this respect, when an option is created, we initialize its local FMDP tree structure composed of the transition trees for its context variables, as well as a local instance of planning algorithm. Thus, when learning the internal FMDP structure of the option, the states injected in the learning algorithm are reduced to contain only variables x_i such that $\forall i, x_i \in X_o$. For instance, the complete state representation in the LIGHT BOX problem is given by 20 variables corresponding to 20 lights, but the states used to update the internal structure of the FMDP corresponding to the option *toggle16* contains only 2 variables “10” and “11”. As a consequence, the sub-FMDP has 2 variables \times 2 sub-options instead of 20 variables \times 20 actions. Finally, the set of all the sub-FMDPs corresponding to all available options represents the global HFMDP. If we consider the state-action space of the global HFMDP as a sum of the sizes of its sub-spaces (that is total memory occupation), it would be smaller than the non-hierarchical one, as illustrated by the results presented in Section 7.5.2.

7.3.3 Incremental update

The options are defined over the model of the problem while this model is learned. Thus some are incomplete or erroneous especially in the first stages of the learning process. To handle this issue, the algorithm checks if the set of the options \mathcal{E} already contains an option defined by the same action and variable, but with a different context definition. If so, it updates it (line 10 in Algorithms 7.2 and 7.3).

For instance, the tree shown in Figure 7.6 gives rise to the option defined by the exit $e = \langle \text{“11”}, \text{toggle11}, 0 \rightarrow 1, \{ \text{“2”} = 1; \text{“7”} = 1 \} \rangle$, while the correct combination of lights to turn on the light “11” is “2”, “5” and “8”. That means that this option contains an irrelevant sub-option *toggle7*, its internal transition function contains a tree for variable “7”, its initiation set contains the irrelevant variable “7” and it lacks the information about variables “8” and “5”.

With respect to this observation, the update of the option is performed using the following procedure:

1. Update the exit context c of the option;
2. Update the local model of transitions \mathcal{F}_o by adding missing transition trees and discarding irrelevant ones;
3. Update the sub-options list by adding missing sub-options and discarding irrelevant ones;
4. Re-compute the rank k of the option;
5. Re-compute the Initialization set I_o .

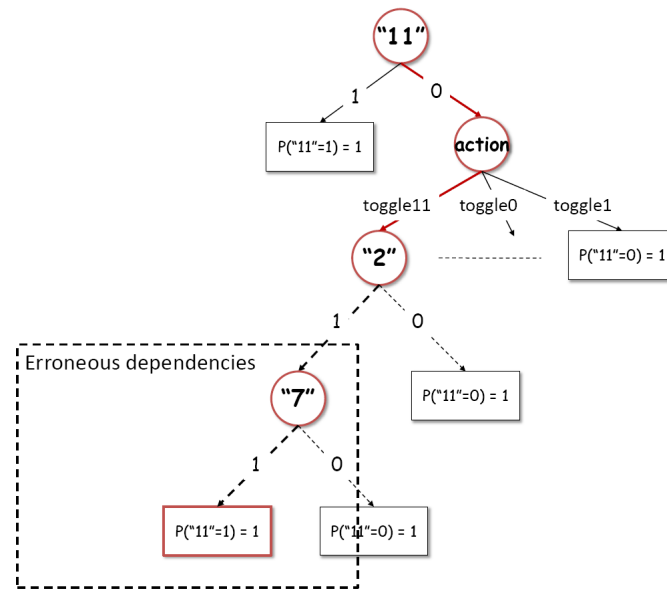


Figure 7.6: Example of incomplete transition tree with erroneous dependencies (irrelevant variable “7”, missing variables “8” and “5”).

This procedure introduces an option each time there is an action that can change one variable value. In fact, there are exactly as many options as there are possible variable value changes. Thus there is no need to remove incorrect options given that as soon as their context is correct, they become accurate.

In other respects, inaccurate options can influence planning and exploration by building an erroneous policy. We will discuss these questions in the next section.

7.4 Planning: FRL over options

The planning phase builds a hierarchical policy over options by incrementally improving and modifying it during the simultaneously with the learning process. The planning algorithm, built upon the ideas coming from HSMQ-like algorithms (Section 5.1.2) and FRL methods based on DYNA-like architecture, is given in Algorithm 7.4. To operate with options and take into account the hierarchical structure of the policy, we use a modified instance of SPITI of the SDYNA framework (Section 4.3.1) for model learning, planning and acting with options. The algorithm executes options recursively by going down the options hierarchy up to primitive actions that can be executed by the agent in its environment, then performs the updates with the returning information on the environment by going up in the hierarchy. This upward update guarantees that the model of transitions includes the changes that occur during the incremental learning process. Thus, local planning is performed for each option with respect to the current structure of

its local model of transitions. To sum up, the planning algorithm implements hierarchical FRL (HFRL) to efficiently learn in hierarchical and factored environment.

7.4.1 SPITI with options

The inner loop of SPITI is decomposed into three phases (Figure 7.2):

- *Acting*: choosing an action according to the current policy using ϵ -greedy as exploration method;
- *Learning*: updating the model of the transition and reward functions of the local FMDP from $\langle X, a, X', R \rangle$ observations using the ITI algorithm;
- *Planning*: updating the value function $Tree[V]$ and policy $Tree[\pi]$ using one sweep of the SVI algorithm.

Each phase of the algorithm is adapted to use the options representation instead of primitive actions. Action phase with options is described in the next section.

Then, in the learning phase, the ITI algorithm is modified to work with options by replacing primitive actions definitions by options in trees and allowing online modifications in the set of transition trees, that is removing or adding new trees while learning. Moreover, ITI uses reduced states representations containing only variables present in the context of the option, so that the local model is updated with the observations of the form $\langle S_o, a, S_o', R \rangle$ where S_o is a set of context variables.

Finally, in the planning phase, the adaptation of SDP algorithms like SVI and IncSVI used to the options framework is straightforward. First of all, the primitive actions available in the environment are all initialized as options with empty exit context and initiation set. As a result, the algorithm deals only with options and does not have to make distinction between options and primitive actions. Since the action space is restricted by the number of sub-options available on a given hierarchical level, the algorithm has an additional argument that specifies the list of options to iterate through. As to incrementality, [Degris, 2007] proposes to re-initialize the value function each time the reward function changes, mainly because the structure of the value function results from the reward function. In the incremental options learning case, the value function tree is reset each time the reward function changes in order to take into account every modification that changes the structure of the policy. The SPITI with options algorithm is given in Algorithm 7.4.

7.4.2 Acting: choosing options

As mentioned in Section 7.3.3 on the incremental update of the option, in the first iterations of the learning process, the options hierarchy and often options themselves

Algorithm 7.4: SPITI with options

input : FMDP \mathcal{F} , options hierarchy \mathcal{E}

for each time step t

1 **if** no option is running **then**

 | option $o \leftarrow \text{ChooseOption}(s, \text{Tree}[\pi], \mathcal{E})$

2 **if** terminal condition of o is satisfied **then**

2.1 | execute exit action a ; observe next state s' and immediate reward r

2.2.a | update local FMDP \mathcal{F}_o with (s, a, s', r)

2.2.b | update parent FMDP $\mathcal{F}_{\text{parent}(o)}$ with (s, o, s', r)

2.3.a | update local policy π_o with \mathcal{F}_o

2.3.b | update parent policy $\pi_{\text{parent}(o)}$ with $\mathcal{F}_{\text{parent}(o)}$

3 **else**

 | sub-option $i \leftarrow \text{ChooseOption}(s, \text{Tree}[\pi_o], \mathcal{E})$

 | **if** sub-option i is primitive action **then**

3.1 | | execute i ; observe s' and r

3.2 | | update local FMDP \mathcal{F}_o with (s, i, s', r)

3.3 | | update local policy π_o with \mathcal{F}_o

3.4 | | **return**: i

 | **else**

4 | | call SPITI over sub-option i : $i \rightarrow \text{SPITI}(\mathcal{F}_i, \mathcal{E})$

are inaccurate or irrelevant. This is why the procedure to choose options must take inaccuracies into account and use a strategy that:

- favors exploration at the beginning,
- prevents options from being stuck and going round in circles within erroneous policies,
- chooses the options at the right level of the hierarchy.

Algorithm 7.5 shows the procedure to choose options.

Algorithm 7.5: Choose Option

input: current state s , policy $\text{Tree}[\pi]$, options hierarchy \mathcal{E} , level k

1 Get option o from the current policy, $o = \max_o[\text{leaf}(\text{Tree}[\pi]|s)]$

2 **if** o is null **then**

 | choose option o from \mathcal{E} accessible in the current state s with current rank k

3 **while** o is null **do**

 | choose option o from \mathcal{E} accessible in the current state s with rank $k - 1$

 | (if $k = 0$ choose random primitive action)

return o

The root node of the option hierarchy represents the overall FMDP. The first option o is chosen according to the root policy while its initiation set contains the current state s . The sub-options are selected according to the internal policy π_o of the option o augmented with an ϵ -greedy exploration policy. If the policy is incomplete and does not contain any information about options available in a current state, then an option is chosen randomly from the options set \mathcal{E} on the current rank level with respect to its initialization set. If no option is available with a given rank, the algorithm chooses from options with a lower rank and so forth. Thus, in the first stages of learning, only the options with rank 0 are available, that is primitive actions chosen randomly and in the end the preference always goes to the options with the higher level of abstraction. This provides a high level of exploration at the beginning and then for unexplored parts of the state space, therefore this speeds up the overall learning process (see Section 7.5.5 for empirical evaluation).

In order to avoid an option from being stuck within an erroneous policy that fails to achieve its exit, we introduce the notion of *selection penalty* that forbids one option for a given number of time steps. In practical terms, when an option reaches its exit state but fails to change the corresponding variable value, that means that its structure is inaccurate. In this case we forbid this option selection for $\frac{(\text{max time steps per episode})}{10}$ time steps in order to let the model of transitions of its parent option be improved.

Furthermore, in practice, high level options often form a partition of the state space, thus the option selection procedure generally returns the only admissible option.

7.4.3 Hierarchical policy

The methods described in this section build the hierarchical policy $\pi = \langle \pi_{o_0}, \pi_{o_1}, \dots, \pi_{o_n} \rangle$ where a global policy is a set of lower level policies. Each option follows the policy of its FMDP, its sub-options follow their respective policies and so on. Figure 7.7 gives an example of a subset of the hierarchical policy built in the LIGHT BOX problem. Since there is no possibility to toggle the light “16” OFF, the algorithm choses a sub-option randomly.

Moreover, to propagate the external rewards to the local policies of options, when a high level option is discovered, an additional reward, named “internal reward” r_o (by contrast with the external reward received from the environment) is assigned to its exit action. We set $r_o = \frac{R_o}{2}$, where R_o is the maximal internal reward of the parent option. For the options on the higher level of the hierarchy, R_o is the maximal immediate external reward that the agent can get. This heuristics is inspired by the “salient event” heuristics introduced in [Singh et al., 2005]. With respect to this heuristics, the internal reward function of the options in the LIGHT BOX problem is the following: successfully toggling light “19” results in receiving 20; lights “15, 16, 17, 18” results in receiving 10; lights “9, 10, 11, 12, 13, 14” results in receiving 5.

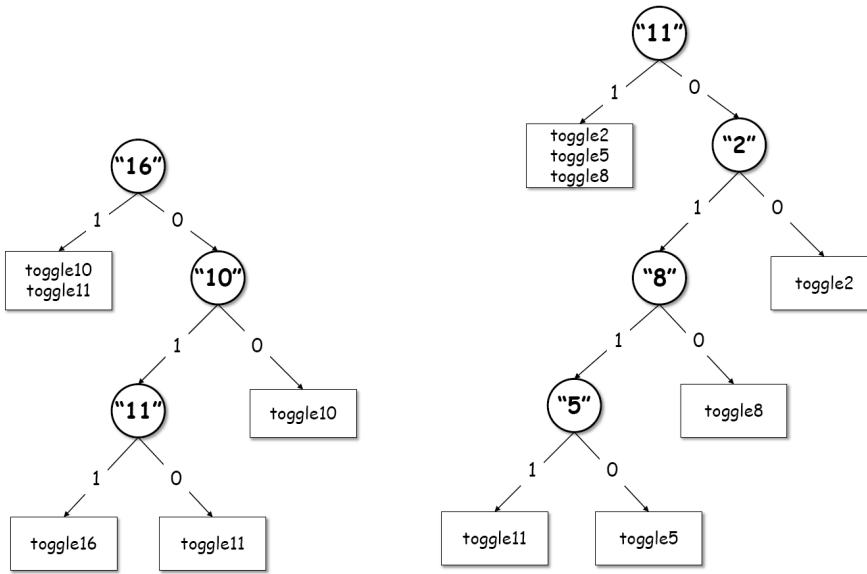


Figure 7.7: Local policy of the options *toggle16* and *toggle11*

The last point concerns the local structure used in planning on each hierarchical level. As mentioned in Section 7.3.2, each option builds its own representation of the model of transitions, then uses it to compute the policy function. This is why, at each level, the model of the underlying FMDP is smaller hence easier to learn and consequently the policy computation is simpler and faster.

7.5 Experimental study

We have chosen to use the following benchmarks when performing the experimentations on the Hierarchical Learning approaches: the TAXI and LIGHT BOX problems. The LIGHT BOX problem [Vigorito and Barto, 2008b], presented in the beginning of this chapter, is a good benchmark for testing the hierarchical decomposition since it has a strict hierarchical structure.

First, we present the TAXI problem, then the results on the overall performance of our algorithms while solving the TAXI and LIGHT BOX problems. Then we take a closer look at the impact of the localization of the transition function on policy learning as well as the transition function representation and its impact on the options learning and planning.

The results presented here below are averaged over 20 runs of 150 episodes where each episode is limited to 300 steps. The curves are smoothed by computing the moving average weighted over ten neighboring values. The algorithms use the following

parameters: $N = 300$ in DYNA-Q and $\epsilon = 0.1$ in ϵ -greedy. The ITI algorithm uses $\chi^2 = 30$ in stochastic problems and $\chi^2 = 0$ in deterministic ones. The algorithms are coded in C# and run on Intel Core2Duo 1.80GHz processor with 2Go RAM.

7.5.1 The TAXI problem

The TAXI problem, first proposed in [Dietterich, 1998] has since become a classical benchmark to test Hierarchical Learning algorithms.

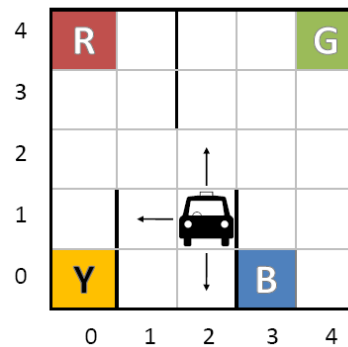


Figure 7.8: The TAXI problem.

The taxi problem is presented in Figure 7.8. A taxi is in a 5-by-5 grid world. There are four special locations, named R , G , Y and B . The taxi problem is episodic, there are 500 possible states. In each episode, the taxi starts in a randomly-chosen state. There is a passenger at one of the four special locations (chosen randomly), and that passenger wishes to be transported to one of the three other locations (also chosen randomly). The taxi must go to the passenger's location, pick her up, go to the destination location, and drop her off there. The episode ends when the passenger is at her destination location or when a predefined number of steps has been reached. At each time step, the taxi can execute one possible action out of six: *Move* the taxi one square *North*, *South*, *East*, or *West*, *PickUp* or *PutDown* the passenger. In a stochastic version, instead of moving in the selected direction, a *Move* action moves in a random direction with probability 0.2. There is a penalty of -1 for each action and an additional reward of 20 for successfully dropping off the passenger. There is a penalty of -10 if the taxi attempts to execute the *PutDown* or *PickUp* actions illegally. The state variables domains are given in Table 7.1 and action constraints in Table 7.2.

Variable	Domain
Taxi location (X,Y)	[0, 1, ..., 5][0, 1, ..., 5]
Passenger location	[R, G, Y, B]
Passenger destination	[R, G, Y, B]
Passenger in the taxi	[Yes, No]

Table 7.1: State variables of the TAXI problem.

Action	Constraints
MoveNorth	-
MoveSouth	-
MoveEast	-
MoveWest	-
PickUp	Taxi location = Passenger location
PutDown	Taxi location = Passenger destination.

Table 7.2: Actions of the TAXI problem.

7.5.2 Performance and convergence speed

The TAXI problem

The results, presented in this section, are published in [Kozlova et al., 2008] and [Kozlova et al., 2009a]. In this work, in order to compare our results to those found in the literature, we use a version of the TAXI problem that contains 800 states. In this version, the options are differentiated by their contexts rather than by variable changes. Thus, the algorithm for discovering options starts by discovering the tree of exits and then builds the tree of the corresponding options.

Figure 7.9 shows an example of options discovered on the TAXI problem after 10 episodes.

Figure 7.10 shows the performance in number of time steps required to complete one episode of the stochastic version of the TAXI problem. The results below are given for SPITI, for a simplified version of TeXDYNA where options are given and for TeXDYNA where options are learned. Table 7.3 recaps the average time in seconds per step within three experimental contexts.

TeXDYNA clearly outperforms SPITI in computation time and in number of learning episodes needed to converge but also in memory requirements. It needs about 40 episodes to converge when options are given, almost 100 episodes with primitives

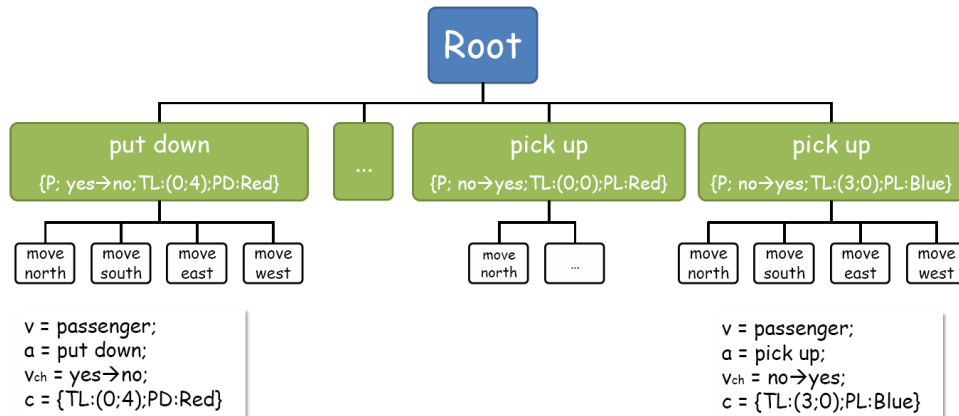


Figure 7.9: Example of options discovered in the TAXI problem. The option *PickUp* changes the value of variable *Passenger* from *No* (not in the taxi) to *Yes* (in the taxi). Its exit context contains 2 variables: *Taxi Location* and *Passenger Location*. It has 4 sub-options.

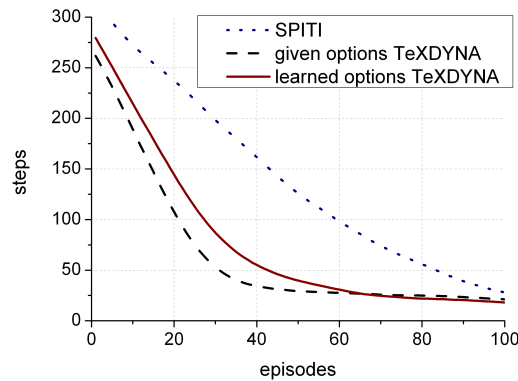


Figure 7.10: Convergence over episodes on the stochastic TAXI problem.

actions only and 60 episodes when options are learned. As expected, this result is intermediate between the one with given options and the one without options.

As to the size on the value function, SPITI builds the complete tree representing the 800 states of the problem, whereas TeXDYNA operates with 8 options (see Table 7.4) each of which only considers 25 states. Thus TeXDYNA requires less time to perform one step since it works on a smaller representation. Even considering the options representing the primitives actions, the hierarchy of options provides a simplification of the global structure. Moreover, simultaneously discovering and refining options while learning the

	Time/step(sec)
SPITI	1.1 ± 0.4
given options TeXDYNA	0.09 ± 0.03
learned options TeXDYNA	0.24 ± 0.08

Table 7.3: Performance on the stochastic TAXI problem.

FMDP structure speeds up the global process since it builds most of the partial policies before the model of transitions is completely learned.

Option	Exit (variable, action, change, context)
PickUpRed	$\langle P, PickUp, No \rightarrow Yes, \{TL = (0; 4), PL = Red\} \rangle$
PickUpYellow	$\langle P, PickUp, No \rightarrow Yes, \{TL = (0; 0), PL = Yellow\} \rangle$
PickUpBlue	$\langle P, PickUp, No \rightarrow Yes, \{TL = (3; 0), PL = Blue\} \rangle$
PickUpGreen	$\langle P, PickUp, No \rightarrow Yes, \{TL = (4; 4), PL = Green\} \rangle$
PutDownRed	$\langle P, PutDown, Yes \rightarrow No, \{TL = (0; 4), PD = Red\} \rangle$
PutDownYellow	$\langle P, PutDown, Yes \rightarrow No, \{TL = (0; 0), PD = Yellow\} \rangle$
PutDownBlue	$\langle P, PutDown, Yes \rightarrow No, \{TL = (3; 0), PD = Blue\} \rangle$
PutDownGreen	$\langle P, PutDown, Yes \rightarrow No, \{TL = (4; 4), PD = Green\} \rangle$

Table 7.4: Options of the TAXI problem (P - Passenger, TL - Taxi Location, PL - Passenger Location, PD - Passenger Destination).

With respect to results found in the literature, our model performs better than HEXQ, which needs about 160 episodes to converge and than MAXQ-Q [Dietterich, 1998], which needs about 115 episodes. Furthermore, both MAXQ-Q and HEXQ converge slower than SPITI. That can be explained by the fact that the former does not explicitly use the factored structure of the problem and the latter spends a significant number of episodes evaluating the order of the variables before building the hierarchy.

In the TAXI problem, the option hierarchy is limited to two levels. In order to experiment on multilevel hierarchies discovery, we apply TeXDYNA to the LIGHT BOX problem that has a four levels hierarchy.

The LIGHT BOX problem

The results, presented in this section, are published in [Kozlova et al., 2010]. Figure 7.11 shows the performance in number of time steps required to complete one episode of the LIGHT BOX problem within three experimental contexts: random policy, TeXDYNA and DYNA-Q applied to the stochastic and deterministic versions of the LIGHT BOX problem.

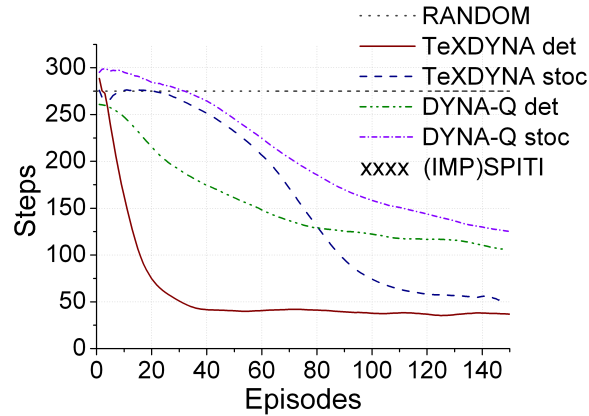


Figure 7.11: Convergence over episodes on the LIGHT BOX problem.

We could not obtain the curves for SPITI and IMPSPITI because SPITI attempts to build the complete value tree representing all possible combinations that is 1 million leaves. IMPSPITI (see Chapter 6) performs better but is still a magnitude of 4 behind TeXDYNA in terms of performance insofar as we could not perform enough runs to compute an average convergence curve in number of steps. This assertion is explained by the results presented in Table 7.5 that recaps the average time in seconds per step and the size of the functions within the stochastic experimental context. In the same way as in the TAXI problem, TeXDYNA requires less time to perform one step since it works on a smaller representation. It starts by discovering the options of the second hierarchical level that allows to switch the green lights on. At this level, each option has 2^3 states and 3 sub-options, the solution being trivial. Learning those policies provides a quick access to the higher level policies of blue and red lights. While SPITI and IMPSPITI are struggling to check if the red light is dependent on the white light, the policy built by TeXDYNA goes straight to the goal state by achieving sub-goals one by one at each level.

	TeXDYNA	SPITI	IMPSPITI	DYNA-Q
Transition function size	780 ± 14	790 ± 25	790 ± 25	–
Value function size	240 ± 20	> 15000	754 ± 54	> 10000
Policy function size	180 ± 8	> 15000	920 ± 62	> 10000
Time/step(sec)	0.04	> 100	> 100	> 2

Table 7.5: Performance on the LIGHT BOX problem (Policy and value function size in total number of nodes in decision trees).

Finally we perform the experimentation on a stochastic version of the LIGHT BOX problem where the agent performs random actions with a 10% probability. As shown in Table 7.5, the algorithm finds the same structure in the stochastic case as in the

deterministic one. One can notice that it takes more time to the algorithm to learn a stochastic transition function, but in the end it discovers the same options hierarchy and the same policy and value functions. Similarly, stochastic DYNQ needs more episodes than its deterministic counterpart DYNQ, to converge.

In order to explain the lesser performance of SPITI and IMPSPITI compared to TeXDYNA, we perform the following test. We record the policy function size for SPITI and TeXDYNA algorithms, visited states tree size for IMPSPITI and number of state-action pairs in DYNQ, at the end of each episode. Figure 7.12 shows this evolution of the corresponding functions size over episodes. TeXDYNA quickly reaches a plateau on its optimal policy size, while the policy discovered by DYNQ continues to grow up until representing all possible states. Therefore, the DYNQ results given in Figure 7.11 are biased by the fact that even if it discovered a kind of sub-optimal policy in a reasonable time, the algorithm is unable to perform the complete policy computation. As to SPITI, the system fails to achieve convergence because of a too strong memory requirement. Indeed, as soon as one tree size exceeds 15000 nodes, the system runs out of memory. In the case of IMPSPITI, the same problem concerns the tree that represents visited states. Even if the policy and value functions represent only relevant variables combinations, the maintenance of the visited states record becomes impossible. In fact, if we take the results presented in [Degris, 2007] as basis, SPITI performs well on trees that do not exceed 6000 nodes, while SPUDD can deal with ADDS containing over 20000 nodes mostly because of good engineering optimization solutions.

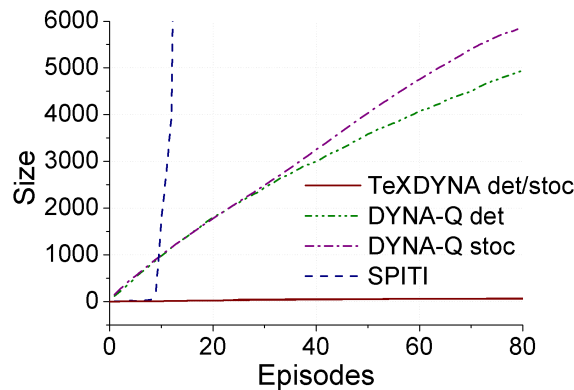


Figure 7.12: Policy size on the LIGHT BOX problem (IMPSPITI refers to the record of visited states).

Further code optimization can improve those performances, but the main point is that TeXDYNA scales much better, thanks to exponential task structure simplification.

7.5.3 Transition function representation

In this section, we study the impact of the use of the two possible transition function structures on the process of learning options and the overall performance throughout the task. The experiments are performed on the deterministic 500 states TAXI problem and deterministic LIGHT BOX problem.

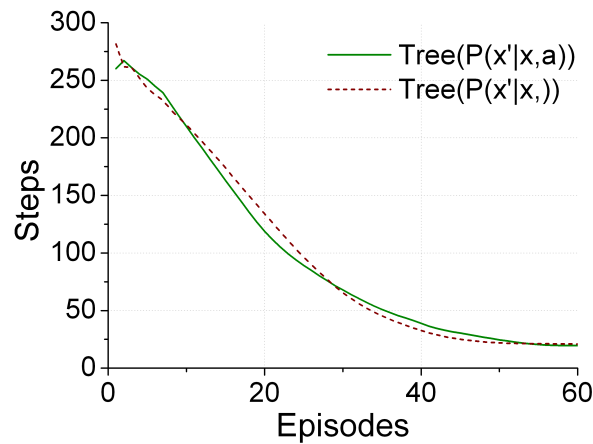


Figure 7.13: Convergence over episodes on the TAXI problem using a transition function representation with one tree per variable per action - $Tree[P_a(x'|s)]$ or one tree per variable where the actions are attributes - $Tree[P(x'|s)]$.

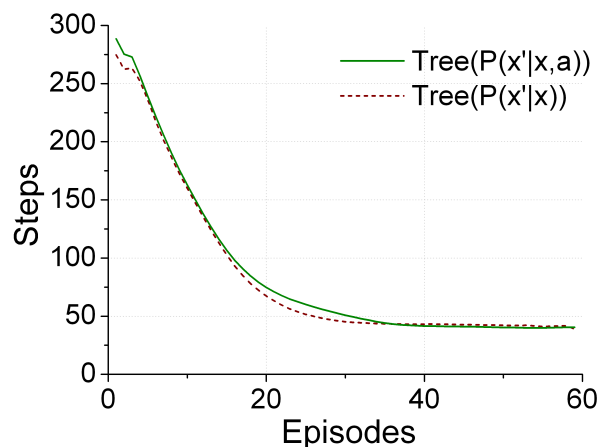


Figure 7.14: Convergence over episodes on the LIGHT BOX problem using transition function representation with one tree per variable per action - $Tree[P_a(x'|s)]$ or one tree per variable where the actions are attributes - $Tree[P(x'|s)]$.

	$Tree[P_a(x' s)]$	$Tree[P(x' s)]$
Transition function size	333 ± 2	320 ± 5
Value function size	196 ± 3	196 ± 4
Policy function size	195 ± 3	196 ± 3
Time/step(sec)	0.047 ± 0.009	0.046 ± 0.007

Table 7.6: Performance on the TAXI problem with 2 transition function representations.

As shown in Figures 7.13 and 7.14, there is no impact of the speed of convergence in number of steps required to achieve one episode. The results on both problem are quasi identical. As to the transition function size, the $Tree[P_a(x'|s)]$ representation is a little bigger but this is not significant. Considering the policy and value function size, as well as the execution time, there is no significant difference.

	$Tree[P_a(x' s)]$	$Tree[P(x' s)]$
Transition function size	780 ± 14	574 ± 7
Value function size	62 ± 8	78 ± 8
Policy function size	58 ± 7	78 ± 9
Time/step(sec)	0.03 ± 0.001	0.026 ± 0.003

Table 7.7: Performance on the LIGHT BOX problem with 2 transition function representations.

These results confirm the robustness of the algorithm showing that it is independent of the transition function representation. Therefore it leads the way to using the TeXDYNA framework with other factored representations, such as rules, ADDS, DBNS or linear functions. Indeed, any representation compatible with the explicit notation of the variable values as consequence of actions can be adapted to the present framework. We come back to this assertion in the final discussion.

7.5.4 Localization of the transition function

We compare the impact of the localization of the transition function on the deterministic TAXI problem with 500 states. In the first place, only the global transition function is learned and each option uses the trees of this function during SVI sweeps. In the second trial, each option has its own local transition function that it learns while executing this option.

Figure 7.15 shows the convergence within both experimental contexts, and Table 7.8 shows the average time per step in seconds and the size of the value and policy function as a sum of all local functions.

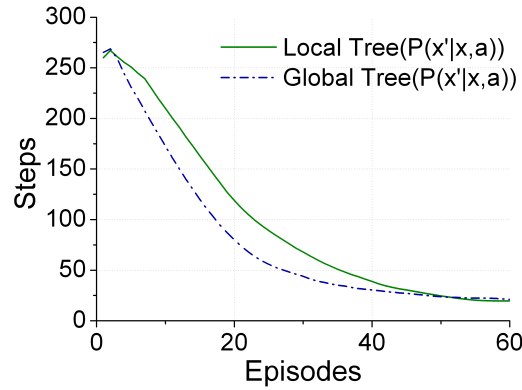


Figure 7.15: Convergence over episodes on the TAXI problem using local or global transition functions.

	Local	Global
Transition function size	333 ± 2	334 ± 2
Value function size	196 ± 3	2195 ± 408
Policy function size	195 ± 3	2076 ± 325
Time/step(sec)	0.047	0.43 ± 0.2

Table 7.8: Performance on the TAXI problem using local or global transition function.

Although the difference between the both curves is not significant in convergence speed in number of episodes (Mann-Whitney test with $p = 6.2E^{-4}$) since both representations lead to a convergence in approximately 50 episodes, the use of the global transition function representation makes the convergence in number of steps a bit faster (Figure 7.15), that is ignoring the fact that it is an order of magnitude slower considering execution time per step (Table 7.8). That is explained by the fact that with a local representation, each option, when first initiated, must “re-learn” its internal transition function, while with a global representation, it uses a shared representation completed before a given option has been discovered. In this case, each option has access to all the decision trees representing the transition function of the overall problem. But, as exemplified in Table 7.8, this global representation results in bigger value and policy functions over options, and thus takes more time for computation. In fact, the value functions of options using the global transition function representation contains additional dependencies irrelevant to these options but introduced in their structure

within SVI. On the other hand, local models combined with state reduction to the set of context variables ignoring anything else provides a significant state-action space reduction within the structure of options.

7.5.5 State space exploration

As mentioned in Section 7.4.2, the option selection mechanism provides a higher level of exploration in the beginning, when the options hierarchy is not accurate. We performed state space exploration test on the TAXI problem with TeXDYNA, SPITI and DYNA-Q algorithms.

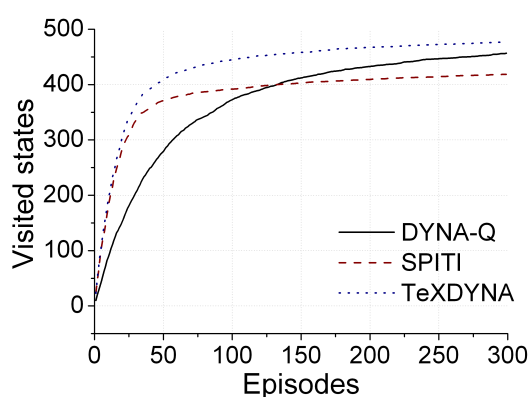


Figure 7.16: Number of visited states over episodes in the TAXI problem.

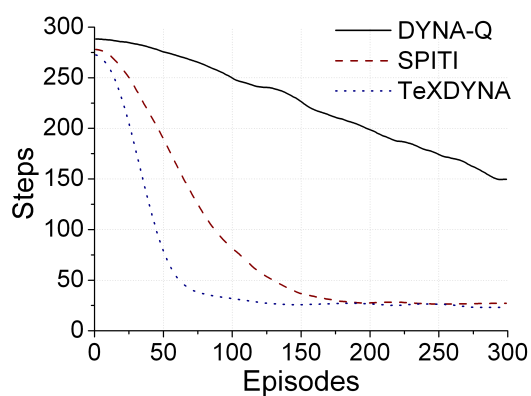


Figure 7.17: Convergence over episodes on the TAXI problem.

Figure 7.16 shows the number of states visited by each algorithm: TeXDYNA, SPITI and DYNA-Q over episodes, compared to the convergence speed in number of steps to

complete each episode shown in Figure 7.17. One can see that TeXDYNA visits more states than SPITI and DYNA-Q and converges faster. In fact, whereas all three algorithms use random exploration, SPITI and DYNA-Q choose between 6 actions available within 500 states while in TeXDYNA each option chooses between 4 sub-options (primitive “Move” actions) within a space of 25 states. Consequently, in TeXDYNA, each action has a bigger probability to be chosen. Furthermore, each time an inaccurate option is detected (when an option fails to achieve its exit), it is forbidden for 30 steps, giving rise to other available options exploration.

7.6 Discussion

We have presented a framework for Reinforcement Learning in hierarchically structured worlds. TeXDYNA ideas are first inspired by Sutton’s Dyna architecture [Sutton, 1991], enriched and adapted to FMDPs by [Degris, 2007]. Second, as to the exit oriented options representation, some ideas come from the HEXQ [Hengst, 2002] and VISA [Jonsson, 2006] frameworks, where the exit definition proposed in HEXQ is extended to include variable change and context in order to address more complex structures. Finally, this work is parallel to the Incremental-VISA approach [Vigorito and Barto, 2008b, Vigorito and Barto, 2008a] (see Section 5.2), in a sense of discovering options in FMDPs with unknown structure.

We showed on the TAXI and LIGHT BOX examples that the method leads to efficient and fast options discovery simultaneously with building the hierarchical policy. Although we have shown that our approach performs better than reference algorithms, there are still many opportunities for improvement. Indeed, through the whole chapter, we used a basic ϵ -greedy exploration/exploitation balance. Introducing optimistic exploration strategies such as the one proposed in [Szita and Lőrincz, 2009] would bring an important advantage.

We mentioned briefly in Section 7.5.3 the possibility of instantiate TeXDYNA with other model representations such as rules, ADDS or linear functions. We show that TeXDYNA performs similarly well with two variants of transition function representations in a decision tree form and extrapolate the possibility to extend its use to other models. Those developments are to be done but it is an advantage to have this possibility that leaves freedom to the designer to choose the adapted transition function representation dependent of the task and its utility. Finally, we used the SPITI instance of SDYNA. The application of these concepts to SPI rather than SVI is straightforward. But, it is also possible to adapt other planning techniques like linear programming as proposed by [Guestrin et al., 2003] or SPUDD [Hoey et al., 1999].

Yet another improvement possibility lays in the transition function localization. As exemplified in Section 7.5.4, the model with local function has to learn from scratch its

transition function. To overcome this flaw, one can transpose global function knowledge already acquired into the newly initiated local functions. The application is quite straightforward and would be interesting for problems with options that have large similar structures.

7.6.1 Related work: Incremental-VISA

The global hierarchy built by TeXDYNA appears similar to a Task option of the VISA framework [Jonsson and Barto, 2006]. However TeXDYNA builds an options hierarchy online and directly from the transition trees taking advantage of their structure, whereas VISA builds a variable influence graph from the given DBNs and then builds transition graphs and reachability trees to determine the initiation sets of the options. Furthermore, it only discovers options linked to the variables directly connected to reward, while TeXDYNA performs a backpropagation of the reward among sub-goals.

In this respect, it is interesting to compare the TeXDYNA approach to the work that extends VISA to the incremental learning case proposed by [Vigorito and Barto, 2008a, Vigorito and Barto, 2008b] (see Section 5.2). Like TeXDYNA, this work attempts to simultaneously learn the hierarchical (options discovery) and factored (DBN learning) structure of the MDP. The major differences between TeXDYNA and the above mentioned approach is, for the moment, that the latter is limited to the deterministic case, whereas TeXDYNA is adapted to stochastic problems. The second difference lies in the way of introducing options into planning: the incremental-VISA approach waits the option to be “mature enough” before introducing it in the hierarchy by using a measure of entropy on the transition functions whereas TeXDYNA inserts options directly in the hierarchy in order to accelerate its completion. In the same way as VISA, Incremental-VISA uses the DBNs to discover dependencies between state variables. Therefore, it needs to build intermediate graphs and trees to catch the internal hierarchical structure, while TeXDYNA operates on decision trees and discovers the structural link directly. Unfortunately, we were not able to compare the computation performance mainly because of the absence on common metrics. Indeed, the experimental results published in [Vigorito and Barto, 2008a, Vigorito and Barto, 2008b] are based on the number of value changes and the time to compute the policy without giving the corresponding metrics criteria.

7.6.2 Limitations

At this point in time, the algorithm presents some limitations. Firstly, as to the problem representation, our option-specific state abstraction is strongly goal-oriented, that is reaching a unique exit context. That can result in the creation of excessive number of options in problems where an action can change more than one variable at the same time.

This problem is also related to the choice of structure learning algorithm. Through this work we use ITI, which is not able to take into account the variables that independent of the agent's actions. In fact, these variables can appear in transition trees with some arbitrary dependencies, and consequently giving rise to options with irrelevant variables in their context. That is why, this version of *TeXDYNA* cannot be directly applied in industrial simulation problems where some events are independent of the agent's behavior. Further development is needed to replace ITI with the algorithms adapted to such environments.

Furthermore, instead of using "internal reward" to propagate the external reward to local policies so that all options have fixed interest, the options discovery algorithm could be combined with task-specific knowledge to identify useful, salient or challenging subroutines.

Secondly, we consider that there is at most one option per variable value change. This assumption simplifies computations within the algorithms, but can be relaxed.

Thirdly, the options hierarchy is strictly ordered, that means that we cannot address problems where the *FMDP* structure includes synchronic arcs or post-action variable dependencies, because it would introduce cross-dependencies between options and cycles in the options hierarchy.

7.7 Conclusion

We have presented the *TeXDYNA* approach that implements the *HFRL* methods combining incremental hierarchical decomposition with the *FRL* framework. On the one hand, the algorithm performs state/goal abstraction by decomposing the overall task into sub-tasks. On the other hand, it introduces temporal abstraction by using options, that are discovered automatically.

Our framework is built on three main ideas:

- the use of the transition function structure represented as decision trees to discover options,
- the localization of the model of transitions hence the reduction of the state-action space,
- the use of the just discovered options immediately in the planning process.

The method proposed in this chapter is simple and its application to *FMDP* planning is straightforward. In the case where the model of transitions is given, the options hierarchy emerges directly from the structure of the decision trees and the options discovery algorithm can be applied without modification. Two sweeps are necessary to build a

sound hierarchy: first - to discover all the options, second - to reorder the hierarchical structure by computing the rank of the options with the complete knowledge of the dependencies underlying the sub-options.

Furthermore, the performance is improved by the so called “localization of transition function”. In fact, instead of taking transition trees from the global structure where the trees may contain the dependencies irrelevant on the given level, the model of transitions is learned locally for each options represented as an FMDP and containing a restricted state-action space. This results in an important acceleration of the solution process.

However, in our presentation, we payed a particular attention to the incremental case where the learning and planning phases are intermixed. An option is inserted in the decision cycle as soon as it is discovered. Even if an option is not perfect, its use in the planning phase speeds up the learning of its internal structure and of its parent option structure.

Chapter 8

Applications

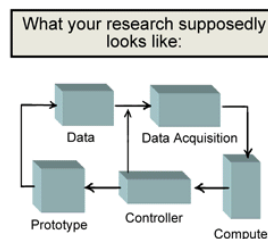


Figure 1. Experimental Diagram



Figure 2. Experimental Mess

WWW.PHDCOMICS.COM JORGE CHAN © 2008

In this thesis we proposed a solution for discrete stochastic sequential decision problems. We showed that factored and hierarchical approaches can be used efficiently to solve large and complex problems. As mentioned in Introduction, we are interested in testing RL methods, studied in this thesis, in human behavior simulation. Two constraints are to be taken into account. From the one hand, to look realistic to the user, this simulated behavior must follow some requirements such as adaptability to changes in the environment, compromise between actions and persistence of the behavior just to mention the most important ones. From the other hand, the algorithm must be able to use the correspondent problem representation.

As to this second constraint, TeXDYNA takes as input discrete states and atomic actions, and during the learning procedure determines the effect of the actions on the environment. When used in a real application such as industrial simulation of crowd behavior in the crisis situation, TeXDYNA meets some difficulties related to the points mentioned here above. In fact, many of the variables used to define states of such simulations have continuous values, actions are not atomic and their effect is neither precise nor well defined in time, without mentioning the fact that the variables are not

always independent. An engineering solution must be undertaken in order to filter the problem representation into its algorithmic form. In the meantime, we propose to test the first constraint concerning the realistic appearance of the behavior on an *ad hoc* simulation problem.

In this Chapter, we present a toy simulation problem that illustrates the potential of TeXDYNA to solve this type of problems. First, we define a simulation scenario, then we explain how we implement TeXDYNA to fit the problem niceties. Then, we perform the experiments and discuss the results as to the transition function and options hierarchy discovery, overall performance, adaptation properties and problem scaling followed by the conclusion remarks.

8.1 Application problem definition

Considering the requirements listed here above, we choose an application problem that exhibits some of these questions in order to show how TeXDYNA can be used to solve them.

In order to facilitate the development and further analyses, we do not implement real world physical constraints. We define a simulation problem that illustrates the capacities of scaling the problem size in number of states, task reuse and generalization of TeXDYNA within a civil security environment. It is possible to define various scenarios. Here we present the “Terrorist vs. Guard” scenario (Figure 8.1).



Figure 8.1: Terrorist vs. Guard scenario.

The scenario is the following. Three human agents are placed in the environment composed of several connected halls (e.g. the metro station). In each hall, there is a fire extinguisher and a gun placed in a predefined locations. There is a passenger who has no influence on the environment, the terrorist who sets fire in the hall where he is and the guard who has to extinguish the fire and kill the terrorist. If the guard extinguishes the fire but the terrorist is still alive, the terrorist will set fire again. Therefore, the guard has to figure out that he has to get the extinguisher, get the gun and kill the terrorist before extinguishing the fire. The guard receives reward of 20 if he extinguishes the fire successfully. There is a negative reward of -1 for each action. Importantly, there is no direct reward for killing the terrorist.

Variable	Domain
Hall 1	[0,1]
Hall 2	[0,1]
Hall 3	[0,1]
Hall 4	[0,1]
Agent location (X,Y)	[0, 1, ..., 5][0, 1, ..., 5]
Gun	[true, false]
Extinguisher (Ext.)	[true, false]
Terrorist	[hall nb, dead]
Passenger	[hall nb]
Fire	[hall nb, no]

Table 8.1: State variables of Terrorist vs. Guard scenario with 4 halls.

Action	Constraints
Move North	-
Move South	-
Move East	-
Move West	-
Take Gun	Agent loc. = Gun loc.
Take Ext.	Agent loc. = Extinguisher loc.
Shoot	Agent loc. hall = Terrorist loc. hall
Extinguish	Extinguisher = true, Agent loc. hall = Fire loc. hall

Table 8.2: Actions of Terrorist vs. Guard scenario.

The state variables and the actions of the problem are given in the Tables 8.1 and 8.2. The problem with 2 halls has 14 400 states, with 4 halls – 640 000 states, 6 halls – 11 289 600 states. This problem can be extended to any number of halls, since the options (policies) learned for one hall can be reused in others.

This representation gives rise to impossible states because only one variable among variables representing the halls can be equal to 1. Additionally the terrorist can set fire only in the Hall where he is, so all the states where Terrorist and Fire locations are different are impossible. Therefore, all the combinations like $\{Hall1 = 1, Hall2 = 1, Hall3 = 0, Hall4 = 0, Ext = true, etc.\}$, $\{Hall1 = 0, Hall2 = 0, Hall3 = 0, Hall4 = 0, Ext = true, etc.\}$ or $\{Hall1 = 1, \dots, Hall4 = 0, Terrorist = Hall2, Fire = Hall3, etc.\}$ are impossible. Finally, there are 120 000 impossible states in 4 halls problem (that is 18%) and 940 800 in the 6 halls problem (8%).

8.2 TeXDYNA and IMPTeXDYNA implementation

In order to use TeXDYNA to solve the “Terrorist vs. Guard” problem, we represent the problem in a schematic way as shown in the Figure 8.2. There is a terrorist in one hall chosen randomly. Each hall contains a gun and an extinguisher positioned in a specified place and all the halls are identical, e.g. the extinguisher and the gun have the same location in each hall, so the agent can reuse a skill learned in one hall in another. The position of the agent is defined by the set of halls values and a position inside the hall.

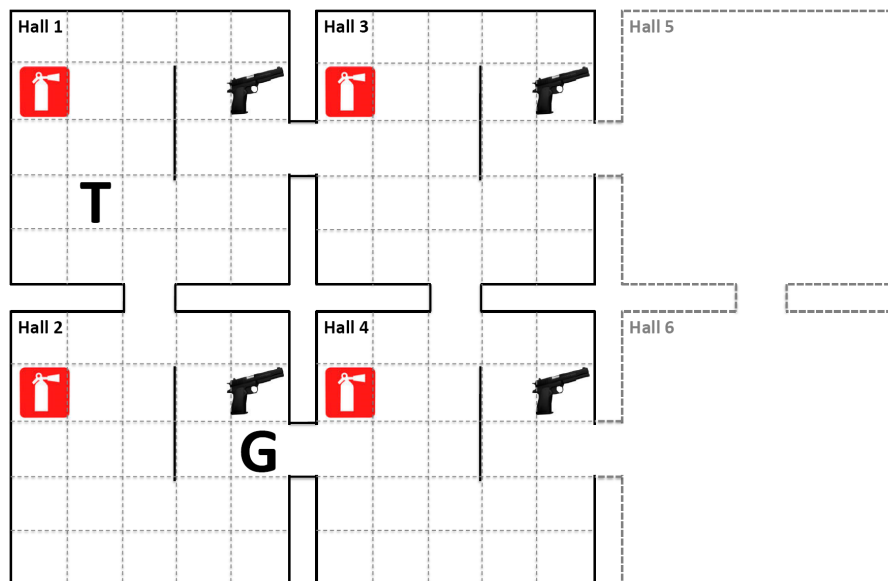


Figure 8.2: Terrorist(T) vs. Guard(G) scenario. Schematic representation.

Since the problem contains impossible variable combinations (or impossible states) we also can use the version of TeXDYNA (IMPTeXDYNA) with IMPSVI instead of SVI in its planning phase. In this case, we use *ad hoc* rules to determine if one state is possible or not by computing the number of Hall variables set to 1. If the result is different from 1,

the corresponding variable combination is impossible. Similarly, if the Terrorist and the Fire locations are different, the state is considered impossible.

The basic procedure of sub-options discovery presented in Section 7.3.2 operates on case where each option changes at most one variable value, but this constraint does not always hold. For instance, in the ‘‘Terrorist vs. Guard’’ problem, the position of the agent is defined by the set of binary variables representing the halls and the location inside the hall. Thus, in some locations, the *Move* actions displace the agent to move from one hall to another and consequently change the value of three variables at the same time: agent location, hall justed leaved and hall just entered. An example of the variables changes if the agent is in hall 2 and moves to hall 4 taking action *Move West* is given in Table 8.3. In a case, if no precaution is taken, cycles may appear in the options hierarchy because options of the same hierarchical level became sub-options of each other changing the same variables. For instance option ‘‘Move to Hall 1 from Hall 2’’ have sub-option ‘‘Move to Hall 2 from Hall 1’’ and vice versa. In this case we use modified options discovery procedure presented in Section 7.3.2 where the cross-dependencies are removed and the corresponding options are given the lowest rank of the two.

Variables	State s	State s'
Hall 1	0	→ 0
Hall 2	1	→ 0
Hall 3	0	→ 0
Hall 4	0	→ 1
Agent location (X,Y)	[4,2]	→ [0,2]
Gun	true	→ true
Extinguisher	true	→ true
Terrorist	Hall 4	→ Hall 4
Passenger	Hall 3	→ Hall 3
Fire	Hall 4	→ Hall 4

Table 8.3: State variable value changes for the action *Move West* of Terrorist vs. Guard scenario with 4 halls.

8.3 Experimental results

We performed the experiment on two sizes of the Terrorist vs. Guard scenario: with 4 halls and with 6 halls. The results presented here below are averaged over 20 runs of 400 episodes where each episode is limited to 300 steps. The curves are smoothed by computing the moving average weighted over ten neighboring values. The algorithms use the following parameters: $N = 500$ in DYNA-Q and $\epsilon = 0.1$ in ϵ -greedy. The

algorithms are coded in C# and run on Intel Core2Duo 1.80GHz processor with 2Go RAM.

8.3.1 Transition function

First, we examine the transition structure built by our algorithms. As mentioned earlier, we are interested in the transition and policy functions representation that can be read by an uninformed user.

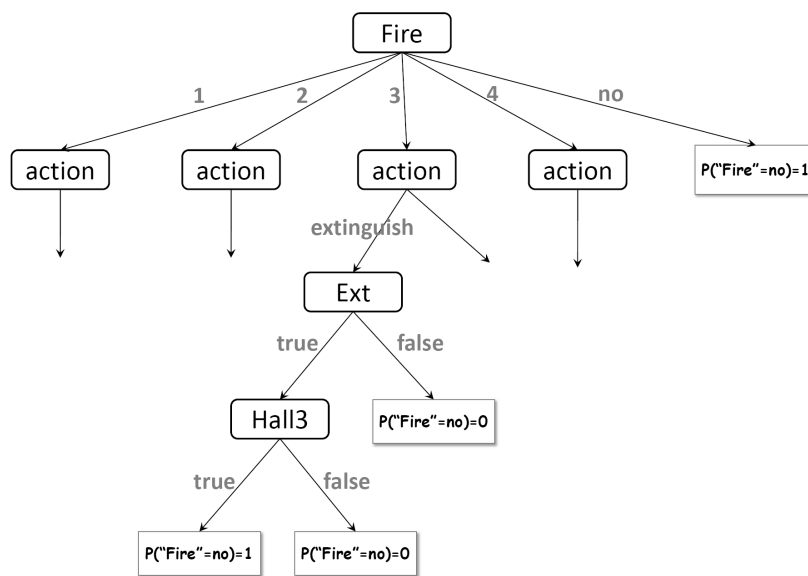


Figure 8.3: Terrorist vs. Guard scenario. Transition function for variable “FIRE”.

With this purpose in mind, we choose to use transition trees in a form of one tree per variable (with the actions as attributes). Figure 8.3 shows an example of the transition tree built by TeXDYNA algorithm for the variable “FIRE”. One can see that if there is a fire in Hall 3, and the Guard is in Hall 3, and he has the extinguisher and he performs the action “Extinguish”, the value of the variable will switch to *NO*. This simple example shows how TeXDYNA learns the internal dependencies between variables building it in the same way as a human operator would do.

As to the reward function, the trees in Figures 8.4 and 8.5 are following the same logic, showing the inter variable dependencies and corresponding expected reward for related subspaces. For example, the system captures the dependency of the “FIRE” variable on the state of Terrorist, because if he is not dead, the fire will be started again.

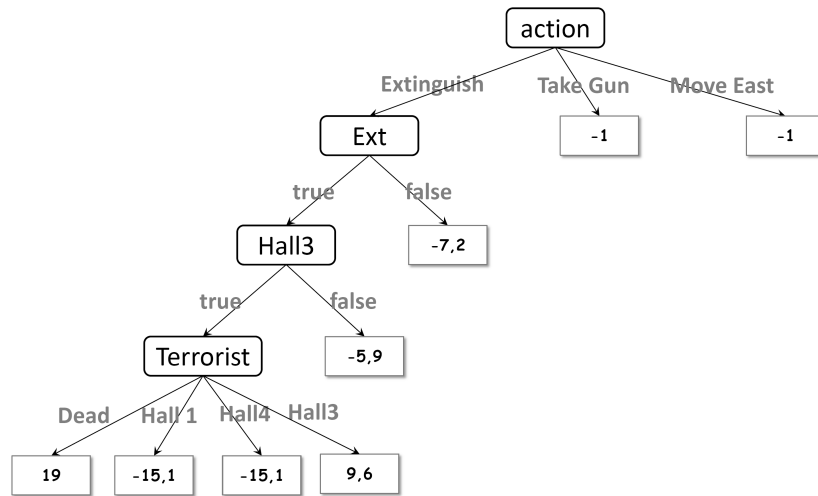


Figure 8.4: Terrorist vs. Guard scenario. Reward function example fore extinguishing fire related subspace.

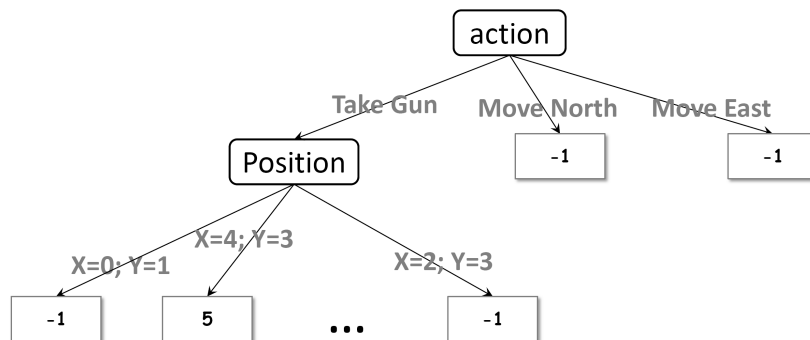


Figure 8.5: Terrorist vs. Guard scenario. Reward function example fore taking a gun related subspace.

8.3.2 Hierarchical policy

Figure 8.6 gives the hierarchy of options discovered in the problem. Basically, the algorithm discovers that the main purpose is to extinguish the fire in a given room and it builds the corresponding options. At the same time, the actions that change the values of the variables Terrorist, Gun, Extinguisher and Hall give rise the sub-options, introduced in the options hierarchy according to their internal dependencies.

Figure 8.7 shows the local policy of the options “Extinguish Hall 4” and one of its sub-options “Shoot Terrorist in Hall 4”. In the policy trees, the primitive actions are given by their names and the options are given with their corresponding exit. Practically, that

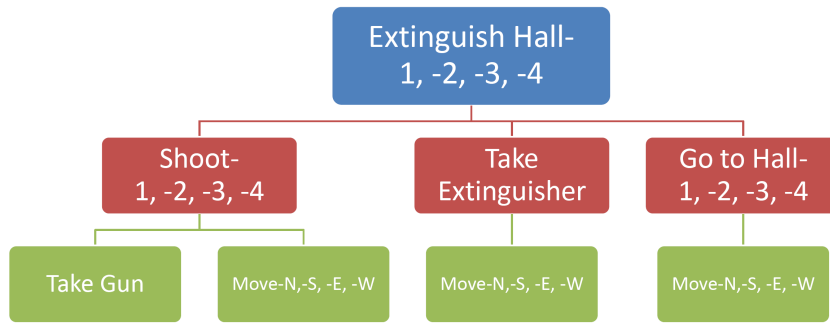
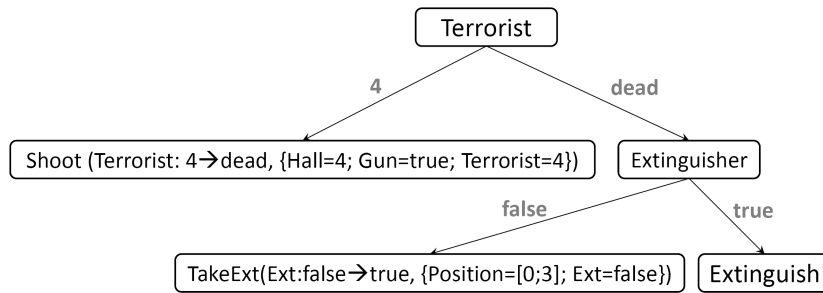
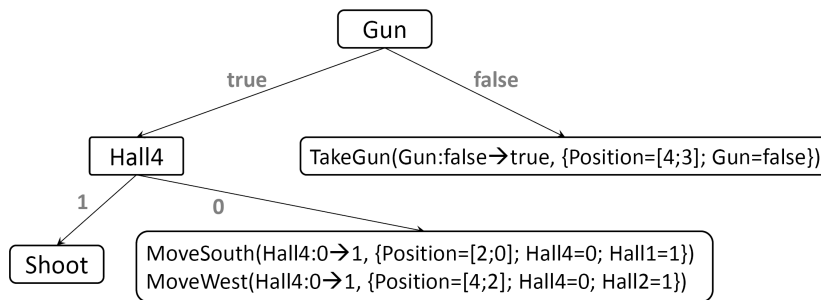


Figure 8.6: Terrorist vs. Guard scenario. Hierarchy of options.



(A) "EXTINGUISH FIRE IN HALL 4"



(B) "SHOOT TERRORIST IN HALL 4"

Figure 8.7: Examples of the policies of the options.

means that the agent starts by choosing the appropriate “Extinguish” options and then executes the subroutines, i.e. sub-options of the local policy. The sub-options then call upon their local policies to choose sub-options, their (sub-)sub-options call upon their own local policies and so on. For instance, as shown in Figure 8.7, the option “Extinguish fire in Hall 4”, if the Terrorist is not dead, chooses sub-option “Shoot Terrorist in Hall 4”. In turn, the sub-option “Shoot Terrorist in Hall 4” uses its local policy to choose a sub-options or a primitive action to execute. This example describes the case when all the options are correctly discovered. On the other hand, during the learning process some options may not be discovered yet or have an incomplete or incorrect definition. The example of the evolution of the option “Shoot Terrorist in Hall 4” is given in Table 8.4 and of the option “Extinguish fire in Hall 4” in Table 8.5. When an option is incomplete or incorrect, its exit condition does non result in the expected variable value change. In this case, the option is forbidden for some number of time steps to let the model of transitions of its parent option be improved (see Section 7.4.2).

Episode	Option definition
20	–
50	Shoot ⟨ Terrorist change: 4→dead { Room4 1, Gun True } ⟩
80	Shoot ⟨ Terrorist change: 4→dead { Room4 1, Gun True, Terrorist 4 } ⟩
110	Shoot ⟨ Terrorist change: 4→dead { Position X=4,Y=1, Room4 1, Terrorist 4 } ⟩
130	Shoot ⟨ Terrorist change: 4→dead { Position X=2,Y=3, Room4 1, Terrorist 4 } ⟩
160	Shoot ⟨ Terrorist change: 4→dead { Room4 1, Gun True, Terrorist 4 } ⟩
190	Shoot ⟨ Terrorist change: 4→dead { Room4 1, Gun True, Terrorist 4 } ⟩
220	Shoot ⟨ Terrorist change: 4→dead { Room4 1, Gun True, Terrorist 4 } ⟩
250	Shoot ⟨ Terrorist change: 4→dead { Room4 1, Gun True, Terrorist 4 } ⟩

Table 8.4: Evolution of the option “Shoot Terrorist in Hall 4” during the discovery process.

Episode	Option definition
20	–
50	Extinguish ⟨ Fire change: 4→no { Terrorist dead, Fire 4 } ⟩
80	Extinguish ⟨ Fire change: 4→no { Position X=1,Y=2, Room4 1, Fire 4 } ⟩ ⟩
110	Extinguish ⟨ Fire change: 4→no { Position X=4,Y=1, Room4 1, Fire 4 } ⟩
130	Extinguish ⟨ Fire change: 4→no { Extinguisher True, Room4 1, Terrorist dead, Fire 4 } ⟩
160	Extinguish ⟨ Fire change: 4→no { Extinguisher True, Room4 1, Terrorist dead, Fire 4 } ⟩
190	Extinguish ⟨ Fire change: 4→no { Extinguisher True, Room4 1, Terrorist dead, Fire 4 } ⟩
220	Extinguish ⟨ Fire change: 4→no { Extinguisher True, Room4 1, Terrorist dead, Fire 4 } ⟩
250	Extinguish ⟨ Fire change: 4→no { Extinguisher True, Room4 1, Terrorist dead, Fire 4 } ⟩

Table 8.5: Evolution of the option “Extinguish fire in Hall 4” during the discovery process.

Note that those policies are built with IMPTeXDYNA in order to avoid showing impossible (irrelevant) variable combinations in the final policy trees.

8.3.3 Performance

We evaluate the performance of the TeXDYNA and IMPTeXDYNA algorithms in comparison with DYNA-Q (that uses tabular representations).

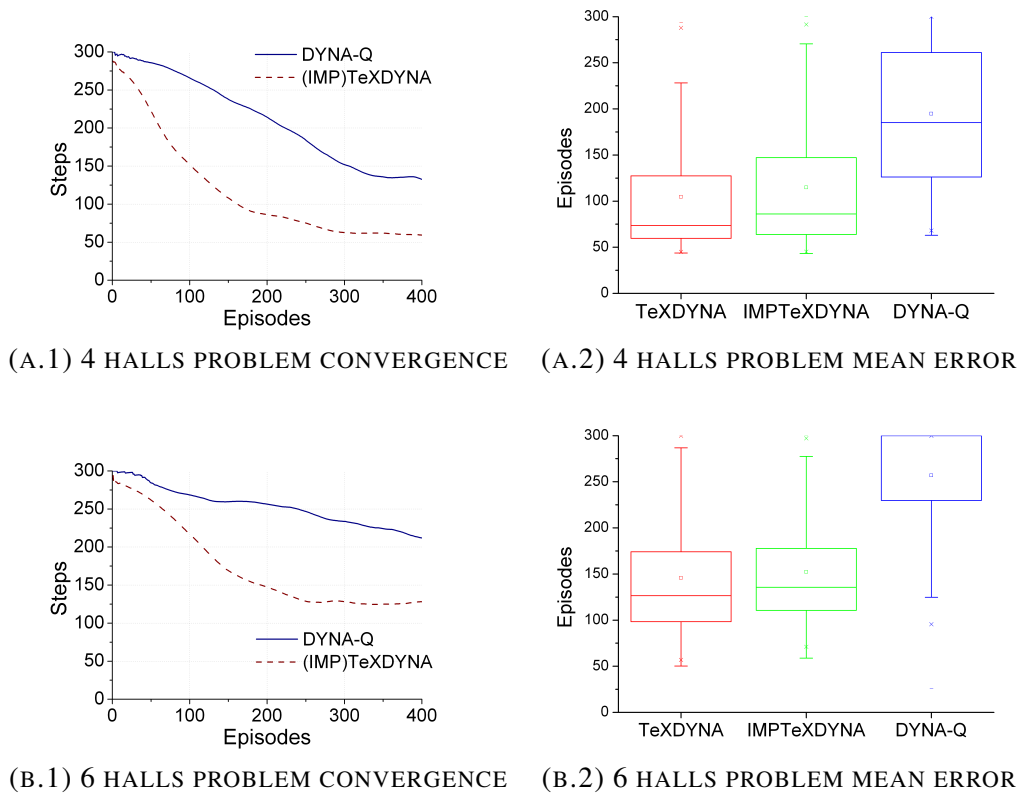


Figure 8.8: Performance in 4 and 6 halls problems.

Figure 8.8 shows the convergence over episodes in number of steps to perform each episode as well as the comparative error to the mean. TeXDYNA and IMPTeXDYNA (curve (IMP)TeXDYNA) perform similarly (Mann-Whitney test with $p = 8.3E^{-4}$ for 4 hall problem and $p = 6.8E^{-4}$ for 6 halls version) in both problem sizes while clearly outperforming DYNA-Q. This performance is explained by the fact that (IMP)TeXDYNA uses structured representations and hierarchical decomposition simultaneously. This way it can take advantage of the compact representation and decomposition into sub-problems. Besides, similar performance of TeXDYNA and IMPTeXDYNA in convergence speed in time and in number of episodes is due to the fact that, on the one hand, we use

ad hoc rules in IMPTeXDYNA and, on the other hand, the rate of impossible states in the problem is not very high (about 18% in the 4 halls problem, 8% in the 6 halls and diminishing so on). But using IMPTeXDYNA can be interesting to avoid the occurrence of impossible combinations in the resulting trees.

8.3.4 Adaptation to changing environment

As mentioned in Section 1.1, the property of adaptability, when the environment or the goals of the agent changes, is one of the most important requirements for the industrial simulation domain. In order to illustrate the possibilities of adaptation of the learned policy to the changing conditions, we redefine our “Terrorist vs. Guard scenario” in the following way:

- For the first 250 episodes, the goal of the agent is to kill the terrorist and to extinguish the fire. There is a reward of +20 for extinguishing fire and then the internal reward of the options representing the sub-goals is computed as $max_k/2$ where max_k is the maximum reward that can be received by the options of level k .
- Then, the reward definition is changed. The agent has to kill the terrorist and does not try to deal with the fire. Therefore, we define the reward distribution as -10 for trying to extinguish the fire and +20 for killing the terrorist.
- Finally, after the episode 600, the goal is inverted. This time the goal of the agent is just to extinguish the fire and avoid killing people, terrorist or not. In this case, there is a reward of +20 for extinguishing the fire and -10 for killing the terrorist.

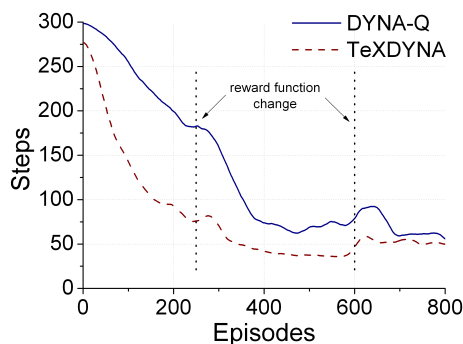


Figure 8.9: Terrorist vs. Guard scenario. Convergence in number of steps when the reward function is modified at episodes 250 and 600.

The results, averaged over 800 episodes, are shown in Figure 8.9. First, it shows that the policy takes into account the changes in the reward definitions and adapts its structure consequently. In fact, one can see a slight augmentation of the number of steps needed to accomplish one episode just after the changes in the reward structure. Second, besides structured representations and scaling properties described in previous sections, TeXDYNA takes advantage of the options structure and reuses the sub-options already discovered while the options that must be changed are relearned. While, unsurprisingly TeXDYNA performs better than DYNA-Q, some points must be clarified. For instance, the curve representing the performance of DYNA-Q in Figure 8.9 drops faster than the one that represents TeXDYNA. First, one can see that when the problem has 2 goals: kill the terrorist and extinguish the fire (until the episode 250), TeXDYNA converges faster. In fact, because of the model-based structure of TeXDYNA, it deals more efficiently with multiple goals problems than DYNA-Q. Then, when the reward function changes, it has only one goal and, after a short period of relearning, the performance of DYNA-Q improves very fast, while those of TeXDYNA have the same speed of adaptation each time the reward function changes. Finally, this adaptability is an inherent property of the RL algorithms. That is why both TeXDYNA, and DYNA-Q deal relatively easily with this kind of difficulty.

8.3.5 Problem size scaling

Figure 8.10 shows the policy size in number of nodes in trees for (IMP)TeXDYNA and (IMP)SPITI, and in number of state-action pairs for DYNA-Q. In the case of the 4 halls problem, TeXDYNA and IMPTeXDYNA reach the optimal number of states in the representation to find a policy rather quickly, while the number of state-action pairs in DYNA-Q continues to grow up to the total number of state-action pairs. As to SPITI and IMPSPITI, we are witnessing the combinatorial explosion from the very beginning because the number of possible states, even factored, is still too big to be represented entirely in one overall policy tree.

We could not obtain the results for SPITI and IMPSPITI for the 6 halls problem because of the combinatorial explosion. At the same time, TeXDYNA and IMPTeXDYNA exploit the hierarchical structure giving access to the options that can be reused in various situations.

	4 Halls	6 Halls
DYNA	0.42 (170 ± 24)	0.836 (500 ± 56)
TeXDYNA	0.08 (20 ± 3)	0.105 (31 ± 6)
IMPTeXDYNA	0.076 (20 ± 3)	0.101 (31 ± 5)

Table 8.6: Terrorist vs. Guard scenario. Performance in time (sec/step). The global time is given in parenthesis.

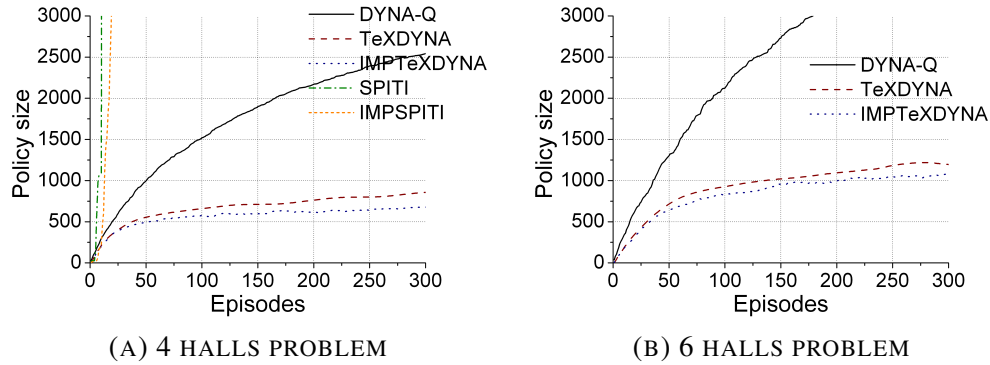


Figure 8.10: Policy size on 4 and 6 halls problem.

Table 8.6 shows the performance in computation time in seconds per step. Once again these results illustrate the advantage of the hierarchical decomposition into options. While the execution time of DYN-Q doubles from one problem size to another, the (IMP)TeXDYNA time raises much slower.

8.4 Discussion

In this chapter, we have presented the training simulation domain and given an example of scenario that can be learned with the TeXDYNA approach. As mentioned in Section 1.1, the main challenges for such applications concern the legibility of the results and commutation performance on the one hand, and the adaptability, compromise and persistence of the behavior of the artificial agents on the other hand.

8.4.1 Contributions

First, we show that the transition function structure represented in a tree form gives the kind of legibility to make it accessible to the uninformed user. This way it is easy to check the correctness of the model and, in future applications, to include external knowledge to improve or correct more complex models.

Second, we are building options directly linked to one of the variable values. The use of options results in building hierarchical policy representing the behavioral strategy of the agents.

Third, as to the adaptability of the behavior to changing situations, we show on the example of changing reward function that model-based RL algorithms exhibit the property of adaptation that allows the agent to adjust its behavior when the goal changes.

Moreover, we show that TeXDYNA can adapt the options that are needed to, while reusing the others as they stand.

8.4.2 Limitations

With the application presented in this chapter, TeXDYNA suffers from the same limitations as revealed in the previous chapter (see Section 7.6.2).

First, the system is using ad hoc “internal rewards”, but in more realistic simulations it would be interesting to use task-specific knowledge to identify useful subroutines. Second, the hierarchy of options is strictly ordered, forbidding the construction of subroutines sharing the same resources.

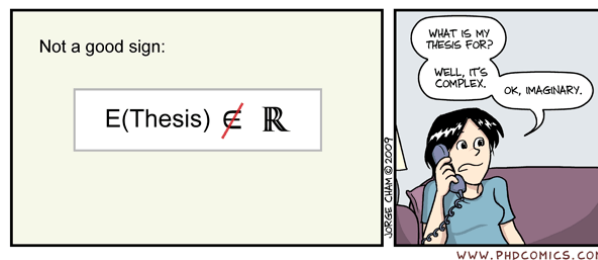
Finally, the policy learned for each separate task exhibits a sort of compromise between options by choosing the one with the bigger outcome. Moreover, it would be interesting to build options of the type “Shoot terrorist in Hall x” where x=Terrorist location. But such constraints type requires first order logic representations. This challenging issue is subject of the ongoing research in the Relational Reinforcement Learning community [Croonenborghs et al., 2008].

8.5 Conclusion

In this chapter we have shown on an example the kind of application where HRL and FRL techniques like TeXDYNA are useful. We have shown that TeXDYNA can be used as an action selector for simulation problems that require such properties as adaptability, compromise and persistence of the behavior. In other respects, TeXDYNA translates the internal structured representation of states transitions into the hierarchy of subgoals. This feature is interesting for discovering internal dependencies between state-action sub-spaces in the environment but also provides some kind of legibility of the resulting policies. In brief, we think that the problems that require near-optimal solutions, combining repeatable skills learning and hierarchical scaling are a good “playing field” for TeXDYNA. On the other hand, we point out that TeXDYNA computes near optimal solutions that maximize one or a set of goals, that may seem simplistic as to the spontaneous human reactions. That is why we insist that human behavior simulation needs further research to be fully exploitable in the industrial training simulation domain.

Chapter 9

Discussion & Conclusion



This thesis explored Reinforcement Learning in factored and hierarchical domains. We have proposed a new approach to stochastic sequential decision problems that combines HRL and FRL techniques. Throughout this thesis, we demonstrated how HRL and FRL methods can be put together in order to address complex large-scale problems in online learning systems. This chapter discusses and sums up the contributions of this work followed by a look at perspectives.

From the very beginning, this work is meant to tackle some well known problems in the stochastic sequential decision making domain. The main ones that stand out above all the rest are large problems, complex structure and real world applications. The contributions of this thesis bring a piece of response to some of these questions. At this point, a popular wisdom comes to mind : *In theory, there is no difference between theory and practice. But, in practice, there is.*

In fact, as to the first point, we can efficiently solve large problems like LIGHT BOX that has 1 million states. At the same time, if we imagine a very large maze with a single goal, the combination of HRL and FRL would not bring any improvement. Further, as to the structural complexity, the problems that have multiple, sequential or contradictory goals can be effectively addressed by such kind of approaches, as shown on

the TAXI problem, the LIGHT BOX problem and Terrorist vs. Guard application scenario examples. On the other hand, as soon as state variables values are continuous, actions are continuous, the actions influence many variables at the same time or else the environment is not markovian, these solutions would not be adequate. Finally, when thinking about applicative validation of the theoretical approaches, such common observations come to mind like very smart and very complicated algorithms applied to toy problems or big complex problems solved only by *ad hoc* algorithms and heuristics that are impossible to generalize. This “generic vs. specialized algorithms dilemma” is one of the most challenging dilemmas in Computer Science and its solution is not straightforward.

To materialize these ideas, recall that all the methods studied in this thesis require some strong assumptions as to the characteristics of the problem, for instance, discrete time, discrete variable values, atomic primitive actions, state or action dependent reward function, clear goal definition, episodic runs, etc. Once these restrictions are taken into account, we propose the solutions that are, on the one hand, based on FRL methods, and on the other hand, the HMDPs, where the problem is decomposed in different levels of abstraction: state factorization, action factorization and in a certain sense MDP factorization by building the hierarchy of sub-tasks. In what follows, we give some comments and future directions on the main contributions of this work starting by factored structure learning, then the work on impossible states, TeXDYNA and its application to simulation domain.

9.1 Learning the problem structure

Throughout this work, we used the incremental tree induction algorithm proposed in [Degris et al., 2006b] along with χ^2 information measure criteria. This algorithm is based on ITI algorithms from [Utgoff et al., 1997] that uses Supervised Learning methods to build decision trees directly from the stream of examples extracted from the observations, without searching to construct the corresponding DBNs. The algorithm is straightforward and provides the kind of legibility and simplicity of the results that are required for building user friendly systems in an industrial context.

On the other hand, these algorithms are subject to the *Post hoc ergo propter hoc* bias, that is taking one event following the other as its consequence. In other words, the variables that change their value independently of the agent’s actions can appear in transition trees with some arbitrary dependencies since the goal of these algorithms is to discover dependencies, randomness not being considered. For instance, in the example of application presented in Chapter 8, if the passenger moves randomly across the halls, the transition tree of the variable “Passenger” would have decision nodes for other changing variables like the position of the agent. Such issues are solved using methods of post and pre-pruning, as well as statistical significance criteria. Only the right choice of the appropriate method requires some prior knowledge about the environment

dynamics and structure, that enters in contradiction with the primary idea of learning from scratch.

Another calling into question is induced by the factorization property of the problem, that is if a given task can be factorized efficiently or not. This property depends on the choice of relevant features to represent states, as well as on the internal structure of the problem. Whereas factored representations are used as a solution to the “curse of dimensionality” because they allow compact representation by grouping similar states, some feature combinations can lead to the introduction of impossible states or in additional dependency constraints.

9.2 IMPSVI, IMPSPITI & Impossible states

In Chapter 6, we studied the case where SDP algorithms create the combinations of state variables that never occur in the problem to be solved. In such circumstances, resulting trees include these impossible combinations, and consequently, the computations are more complicated in time and memory space and the final solutions is less legible. We proposed a new algorithm based on SPITI [Degris et al., 2006b] that modifies the standard SDP algorithms like SVI in order to get rid of the impossible states in a theoretically sound way.

We have shown in Section 6.5.4 and the following Discussion that even if the questioning on the relevance of such representations is legitimate for standard benchmark problems, it is still one of the most efficient solutions to the “curse of dimensionality” problem. For instance, in some cases, non-factored approaches, like tabular representations, can be more efficient but as soon as the problem has more than one final goal, SDP algorithms perform faster. Therefore, if the problem can generate a lot of impossible states, it can be judicious to use the versions of the SDP algorithms that take such combinations into account. Another question concerns the generalization property offered by the standard SDP approach, that is not respected by IMPSVI, since it excludes all non-visited states. Therefore, the question of relevance of these approaches is still open since other learning algorithms, that do not use factored representations, and therefore do not introduce impossible combinations, can be used to solve such problems.

Finally, as exemplified in Chapter 8, when FRL algorithms are used in the domains where the resulting policy trees have to be accessible to an uninformed user, the use of IMPSVI and IMPSPITI is justified by the concern for having only relevant (visited by the agent) situations represented in the final solution.

9.3 TeXDYNA

In Chapter 7, we proposed the TeXDYNA approach that combines HRL and FRL methods (HFRL). TeXDYNA performs incremental hierarchical decomposition of the FMDP, that is identifying subgoal states and generate temporally extended actions (options) that take the agent to these states. The method is based on the automatic discovery of options directly from transition function structure represented as decision trees. Then each option represents a sub-FMDP and computes its own local policy from its own local transition function. Furthermore, these algorithms are online, that is options are learned during the exploration and inserted in the decision cycle as soon as they are discovered. We have shown on benchmark problems the advantages of these methods in memory occupation, computation burden and convergence speed.

At the same time, we point out the directions to be explored in order to improve these properties. First, it would be interesting to generalize the transfer of the relevant parts of the accumulated knowledge to the options, taking inspiration from Transfer Learning techniques [Taylor and Stone, 2009]. Second, reorganizing options hierarchy along with the policy computation and taking into account this hierarchy when building a policy is an interesting point to explore.

In other respects, further comparisons to other automated options discovery methods, and particularly Incremental-VISA, would bring more clarity as to the place of these algorithms in the overall pool of hierarchical approaches.

9.4 Application to the industrial simulation domain

Finally, we also performed experiments on a problem that exemplifies the contribution that such methods can make in a human behavior simulation domain (Chapter 8). Obviously, it would be more interesting to test TeXDYNA “in the field” directly in the simulators, but we did not have the possibility to apply TeXDYNA in the software developed by THALES. The application example we proposed has demonstrated the properties of adaptability, persistence and compromise between contradictory actions achieved by using hierarchical and factored approaches to RL. As mentioned in the beginning of the Discussion, the algorithms we study in this thesis assume some characteristics of the environment, like atomic primitive actions and discrete state definitions that are not always respected in real life simulations. Further research is necessary to integrate these learning methods to field simulations.

At last, as to the demonstration of the learning techniques in the simulation domain, we have presented an example of problem where learning is useful. The main challenge comes from the fact that such techniques are meant to search for the optimal policy, but the main concern in human behavior simulation is not about optimality, but about realism

and autonomy of the artificial agents. From this point of view, optimization techniques such as the ones investigated in this thesis do not provide exactly the expected result. Nevertheless, using motivational exploration techniques, introducing emotions in the variable set and using external knowledge seems to be the right directions to explore in the future to improve the realism of the behavior of the agents.

Finally, we have to mention that industrial applications are often based on *ah hoc* empirical solutions, to the detriment of genericness. The problems like the one used in this chapter can be efficiently solved by handcrafted navigation rules without implication of learning algorithms. Nevertheless, we argue that these learning techniques can bring new competences to the simulation software such as adaptability, new engineering solutions, faster developments, etc.

9.5 Perspectives

Apart from the perspectives already mentioned here above, we consider the following open problems as the most valuable directions for future research.

First, emotional and motivational learning approaches seem relevant especially in the human behavior simulation problems. In the same context, more inspiration can be taken from the psychological study on human decision making (biases and heuristics). In this thesis we focused on the case where the solution must be learned entirely from scratch, but humans and animals use previous knowledge such as “rules of thumb”, educated guesses and intuitive judgments to find the solution [Gigerenzer, 2007]. Therefore, introducing such techniques would be a valuable direction for future research.

Second, when considering real life simulations and robotic applications, it is important to be able to learn in continuous spaces with non-atomic actions. In this perspective, further investigation of the SMDP techniques is necessary.

Then, more detailed study of the exploration/exploitation dilemma should be foreseen. In fact, we used basic ϵ -greedy exploration strategy, but we expect that more sophisticated techniques like active learning would give better results.

And finally, the experiments should be undertaken on more complex real life applications problems. Once the limitations such as learning decision trees with independent exogenous events and dealing with continuous spaces would be successfully addressed, it would become realistic to foresee the application of TeXDYNA in industrial real life simulators.

Appendices

Résumé en français

Résumé du Résumé

Cette thèse a été réalisée dans un contexte de simulation industrielle qui s'intéresse aux problèmes de la modélisation du comportement humain dans les simulateurs d'entraînement militaire ou de sécurité civile.

Nous avons abordé cette problématique sous l'angle de l'apprentissage et de la planification dans l'incertain, en modélisant les problèmes que nous traitons comme des problèmes stochastiques de grande taille dans le cadre des Processus de Décision Markoviens (MDP).

Les MDP factorisés (FMDP) sont un cadre standard de représentation des problèmes séquentiels dans l'incertain, où l'état du système est décomposé en un ensemble de variables aléatoires. L'apprentissage par renforcement factorisé (FRL) est une approche d'apprentissage indirecte dans les FMDP où les fonctions de transition et de récompense sont inconnues *a priori* et doivent être apprises sous une forme factorisée. Par ailleurs, dans les problèmes où certaines combinaisons de variables n'existent pas, la représentation factorisée n'empêche pas la représentation de ces états que nous appelons *impossibles*.

Dans la première contribution de cette thèse, nous montrons comment modéliser ce type de problèmes de manière théoriquement bien fondée. De plus, nous proposons une heuristique qui considère chaque état comme impossible tant qu'il n'a pas été visité. Nous en dérivons un algorithme dont les performances sont démontrées sur des problèmes jouet classiques dans la littérature, MAZE6 et BLOCKS WORLD, en comparaison avec l'approche standard.

Pour traiter les MDP de grande taille, les MDP hiérarchiques (HMDP) sont aussi basés sur l'idée de la factorisation mais portent cette idée à un niveau supérieur. D'une *factorisation d'état* des FMDP, les HMDP passent à une *factorisation de tâche*, où un ensemble de situations similaires (définies par leurs buts) est représenté par un ensemble

de sous-tâches partiellement définies. Autrement dit, il est possible de simplifier le problème en le décomposant en sous-problèmes plus petits et donc plus faciles à résoudre individuellement, mais aussi de réutiliser les sous-tâches afin d'accélérer la recherche de la solution globale. Le formalisme des options qui inclut des actions abstraites à durée étendue, permet de modéliser efficacement ce type d'architecture.

La deuxième contribution de cette thèse est la proposition de *TeXDYNA*, un algorithme pour la résolution de MDP de grande taille dont la structure est inconnue. *TeXDYNA* combine les techniques d'abstraction hiérarchique de l'apprentissage par renforcement hiérarchique (HRL) et les techniques de factorisation de FRL pour décomposer hiérarchiquement le FMDP sur la base de la découverte automatique des sous-tâches directement à partir de la structure du problème qui est elle même apprise en interaction avec l'environnement.

Nous évaluons *TeXDYNA* sur deux benchmarks, à savoir les problèmes TAXI et LIGHT BOX, et nous montrons que combiner l'abstraction d'information contextuelle dans le cadre des FMDP et la construction d'une hiérarchie dans le cadre des HMDP permet une compression très efficace des structures à apprendre, des calculs plus rapides et une meilleure vitesse de convergence. Finalement, nous estimons le potentiel et les limitations de *TeXDYNA* sur un problème jouet plus représentatif du domaine de la simulation industrielle.

Mots-clés : processus de décision markovien factorisé, apprentissage par renforcement, options, décomposition hiérarchique, programmation dynamique structurée, états impossibles

Ce résumé de la thèse est organisé de la façon suivante. Nous commençons l'introduction par la présentation du contexte industriel de ce travail. Puis, nous posons brièvement, les bases théoriques de l'apprentissage par renforcement. Dans la Section 9.5, nous présentons l'essentiel des contributions de cette thèse, dont nous discutons les implications dans la Section 2.b.

Introduction

Cette thèse a été financée par la convention CIFRE en partenariat avec l'entreprise THALES et réalisée au sein du département Simulation conjointement avec l'équipe Synthetic Environments & Simulation à ThereSIS (Thales European Research center for E-Gov & Secured Information Systems). Ces départements proposent les solutions de test, d'entraînement ou de contrôle pour la simulation du comportement humain dans des simulateurs d'entraînement militaire ou de sécurité civile.

Le principal défi qui se pose devant les applications de ce type est de modéliser le comportement des entités artificielles autonomes de façon réaliste, tout en gardant les structures de la solution accessible à un opérateur humain.

Par conséquent, le but applicatif de ce travail de thèse est de tester les méthodes d'apprentissage à des fins de modélisation du comportement humain dans ce type de simulation.

Simulation industrielle

Au sein du domaine de la simulation industrielle, les Environnements Synthétiques sont utilisés pour simuler les environnements de test ou d'entraînement, plus faciles à réaliser sur support artificiel que dans des conditions réelles. Les techniques de simulation permettent de traiter une variété de problèmes de planification et d'apprentissage dans les environnements stochastiques, tels que : la simulation du comportement des agents adaptatifs dans des environnements dynamiques, la résolution de problèmes d'optimisation ou encore la découverte de la structure interne d'un environnement inconnu.

La simulation du comportement humain présente quelques contraintes particulières. Tout d'abord, le comportement doit apparaître réaliste à l'utilisateur. Par la suite, le comportement obtenu doit être compréhensible pour un utilisateur non-averti. Par exemple, lors du traitement des tâches critiques où l'intervention humaine s'avère nécessaire, il est primordial de fournir une solution claire et rapidement identifiable. C'est pour cette raison que nous avons choisi les représentations structurées et, notamment, les arbres de décision. De plus, les simulateurs sont souvent des systèmes en

temps réel, donc requièrent des performances de calcul élevées pour pouvoir modéliser plusieurs entités simultanément.

Dans cette optique, le comportement des agents est modélisé sous forme d'un mécanisme de sélection d'action (ASM). Un ASM implémente le processus de choix d'action la plus appropriée par rapport à la situation dans l'environnement de l'agent à chaque pas de temps. Ces systèmes doivent répondre à un certain nombre d'exigences, notamment :

- adaptabilité à des environnements dynamiques par apprentissage de réponses nouvelles,
- compromis entre les actions en choisissant l'action la plus adaptée à long terme,
- persistance des actions qui permettent d'atteindre le but choisi par opposition à l'hésitation entre plusieurs activités,

pour ne citer que ceux que nous traitons dans cette thèse. Notre but ici est de proposer un ASM qui implémente ces exigences dans le contexte des problèmes complexes de simulation.

Apprentissage par Renforcement

L'apprentissage par renforcement (RL) [Sutton and Barto, 1998] est un mécanisme d'apprentissage par essai-et-erreur en interaction avec l'environnement. L'agent ne possède pas de connaissance *a priori* sur son environnement et apprend à partir des conséquences de ses actions en percevant un signal de récompense ou de punition. Le but de ces algorithmes est de choisir des actions qui maximisent la récompense accumulée par l'agent. Ce type de problèmes est habituellement modélisé sous forme de Processus de Décision Markovien (MDP) [Puterman, 1994], où le problème est représenté par un ensemble d'états et d'actions. Chaque état représente la situation de l'agent dans son environnement à un instant donné. L'agent passe d'un état à un autre en entreprenant des actions pour lesquelles il reçoit des récompenses ou des punitions. Les transitions entre les états sont formalisées par la fonction de transition et les valeurs accordées aux états par la fonction de récompense.

Malgré l'efficacité démontrée des algorithmes RL standards, les difficultés apparaissent dès que la taille du problème augmente et/ou la structure interne du problème se complexifie.

Afin de surmonter ces difficultés, les MDP factorisés (FMDP) [Boutilier et al., 1995] exploitent la structure interne du problème. Les transitions entre les états sont représentées par un réseau bayésien dynamique (DBN) [Dean and Kanazawa, 1989] qui modélise les dépendances entre des variables. Ainsi les états sont factorisés par

rapport aux variables, et plus précisément les valeurs des variables qui permettent de prédire les transitions entre ces états. Les états similaires sont regroupés pour réduire la taille de l'espace d'état global. Par contre, en pratique, la connaissance exacte des transitions entre les états n'est que rarement accessible. Ce cas est traité par apprentissage par renforcement factorisé (FRL), une approche des FMDP où les fonctions de transition et de récompense sont appris en utilisant les méthodes d'apprentissage supervisé [Degris et al., 2006b]. Une des représentations factorisées est basée sur la structuration sous forme d'arbres de décision qui permettent de représenter la fonction de transition de façon plus compacte que les représentations tabulaires classiques. C'est cette représentation que nous utilisons au cours de ce travail, principalement parce qu'elle permet l'apprentissage aisé des structures et parce que nos algorithmes de référence sont basés sur cette même représentation.

Une autre approche pour résoudre les problèmes de grande taille, mais aussi des problèmes dont la structure est complexe, provient des méthodes d'apprentissage par renforcement hiérarchique (HRL). HRL est basé sur les MDP hiérarchiques (HMDP) d'un côté et le cadre des options [Sutton et al., 1999], [Precup, 2000] de l'autre. Les HMDP permettent de simplifier le problème par la décomposition en un ensemble de sous-problèmes plus faciles à résoudre individuellement, tandis que le cadre des options permet de représenter ces sous-problèmes par une généralisation des actions primitives pour inclure des actions abstraites à durée étendue. Souvent, cette décomposition hiérarchique est faite par un opérateur humain mais, dans beaucoup de problèmes complexes ou de grande taille, cette hiérarchie est difficile, voir impossible, à fournir. Par conséquent, la construction automatique des hiérarchies est un des problèmes importants dans le domaine de HRL. Des algorithmes comme HEXQ [Hengst, 2002], VISA [Jonsson and Barto, 2006] ou Incremental-VISA [Vigorito and Barto, 2008b] visent à résoudre ce problème. HEXQ et VISA construisent une hiérarchie des options à partir d'une structure du problème donnée sous forme de DBN. Incremental-VISA, parallèlement au travail présenté ici, augmente l'algorithme VISA d'un apprentissage incrémental de la structure du problème.

Dans la suite, nous présentons les contributions de ce travail sur la base des formalismes qui viennent d'être cités.

Contributions

Les contributions de cette thèse peuvent être appréhendées sur deux plans : théorique et applicatif. D'un point de vue théorique, nous proposons un nouvel algorithme pour la résolution des problèmes d'apprentissage par renforcement factorisés et hiérarchiques quand leur structure est inconnue. Les contributions principales dans ce domaine concernent la gestion des états impossibles et la décomposition hiérarchique dans le cadre FRL. Par ailleurs, en ce qui concerne l'application pratique, le but de ce

travail est de vérifier si les méthodes de HRL et FMDP peuvent être combinées pour traiter les problèmes de simulation du comportement humain dans des environnements synthétiques d'une manière efficace.

IMPSVI, IMPSPITI & Gestion des états impossibles

Nous appelons *états impossibles* des combinaisons des variables qui n'existent pas dans le problème mais qui peuvent apparaître dans sa représentation factorisée. Nous montrons sur l'exemple du problème de BLOCKS WORLD comment ces états apparaissent et comment les modéliser en vue d'optimiser la recherche de solution. Le problème BLOCKS WORLD est défini par b blocs distribués sur s piles. Le but étant de poser y blocs sur la première pile. La représentation factorisée de ce problème peut se baser soit sur la notation binaire de chaque cellule de chaque pile, soit sur le nombre de blocs par pile ou encore la place de chaque bloc. Dans tous les cas, certaines combinaisons de variables n'apparaissent pas dans le problème, ce qui est illustré sur la Figure 6.3.

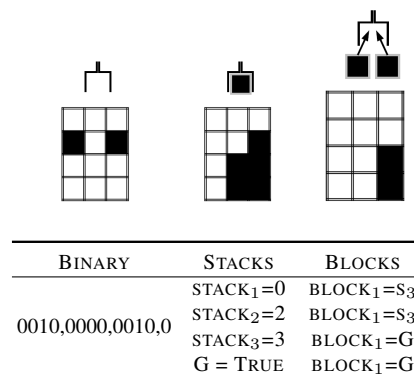


FIGURE 1: Quelques combinaisons de variables qui représentent les états impossibles dans le problème BLOCKS WORLD.

Nous proposons donc une nouvelle classe d'algorithmes (IMPSVI et sa version FRL-IMPSPITI) qui adaptent les algorithmes FRL existants pour traiter les problèmes en prenant en compte ce type de contraintes. Basés sur les algorithmes de Programmation Dynamique Structurée (SDP) et plus précisément sur SVI (Itération de Valeur Structurée) [Boutilier et al., 2000], ces algorithmes permettent d'intégrer l'information sur l'existence des états pour ensuite exclure les états impossibles lors des premières phases de calcul afin de simplifier et d'accélérer la recherche de la solution.

Par ailleurs, nous montrons que considérer comme impossibles les états que l'on a pas encore visités est une heuristique efficace. Nous argumentons la pertinence de notre approche par le fait que cette situation arrive souvent en pratique et que modifier ainsi les algorithmes standards permet une résolution plus efficace que d'ignorer le phénomène.

Nous avons conduit une série d’expériences portant sur la performance générale et la vitesse de convergence de nos algorithmes (IMPSVI et IMPSPITI), pour ensuite les comparer avec les algorithmes tabulaires standards. Les expériences ont été menées sur les problèmes MAZE6 et BLOCKS WORLD. Ici nous présentons rapidement les résultats principaux sur le problème BLOCKS WORLD.

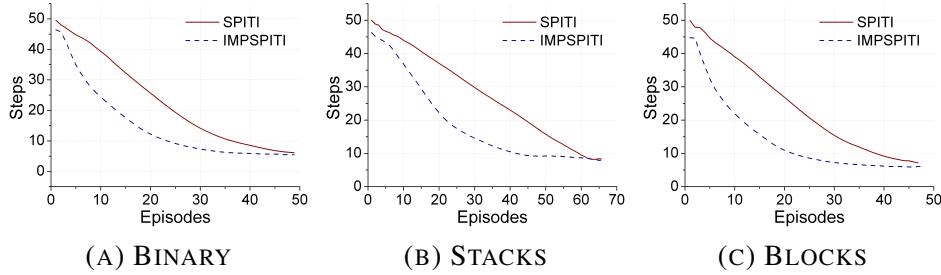


FIGURE 2: Problème BLOCKS WORLD (taille 4-3-4) : performance sur épisodes.

		BINARY		STACKS		BLOCKS	
B-S-Y		SPITI	IMPSPITI	SPITI	IMPSPITI	SPITI	IMPSPITI
3-3-3	% IMP.	98.5%		87.5%		15.6%	
	TIME	0.28 ± 0.03	0.04 ± 0.01	0.05 ± 0.02	0.01 (0.01)	0.04	0.03 (0.03)
4-3-4	% IMP.	99.7%		90%		26.1%	
	TIME	2.9 ± 0.4	0.08 ± 0.01	0.06 ± 0.02	0.01 (0.01)	0.17 ± 0.02	0.18 ± 0.01 (0.13 ± 0.01)
4-4-3	% IMP.	>99.99%		95.6%		18%	
	TIME	34 ± 7	1.2 ± 0.2	0.13 ± 0.06	0.02 (0.02)	0.34 ± 0.2	0.9 ± 0.07 (0.5 ± 0.03)
5-4-4	% IMP.	>99.99%		96.5%		26.3%	
	TIME	-	2.6 ± 0.3	0.2 ± 0.09	0.05 (0.04)	2.6 ± 0.3	6.5 ± 0.9 (3.1 ± 0.2)

TABLE 1: Le problème de BLOCKS WORLD : le taux d’états impossibles et le temps nécessaire pour accomplir chaque pas (en secondes). “-” signifie que nous n’étions pas en mesure d’obtenir les résultats après trois jours de calcul. IMPSPITI utilise les arbres pour représenter les états impossibles. Les résultats sont donnés pour le cas où les états visités sont enregistrés dans un arbre de décision (le temps entre parenthèses représente les résultats avec les règles *ad hoc* pour définir les états impossibles).

La Figure 2 représente la vitesse de convergence de BLOCKS WORLD de taille 4-3-4 (i.e. 4 blocs, 3 piles avec le but de poser 4 blocs sur la première pile), représentative d’un problème de taille moyenne. Il y a 2^{17} états avec la représentation binaire (dont 55 possibles), 1250 avec la représentation par pile (dont 55 possibles) et 625 avec la représentation par bloc (dont 512 possibles). IMPSPITI nécessite moins d’épisodes que SPITI pour converger vers une politique optimale. Ce résultat s’explique par le fait que IMPSPITI utilise des arbres de transitions plus petits avec une structure plus simple ce qui permet une propagation de valeurs plus rapide à travers les arbres. Mais, comme prévu, la différence est d’autant moins importante que le taux d’états impossibles est plus petit. Néanmoins, même dans le cas de la représentation par bloc où il n’y a pas beaucoup

d'états impossibles (Figure 2 (c) *Blocks*), la politique s'améliore plus rapidement avec IMPSPITI. La correspondance entre le temps en seconde mis pour exécuter chaque pas en corrélation avec le taux d'états impossibles est représentée dans le Tableau 1.

La limitation principale de cette approche réside dans la représentation des états impossibles, le meilleur cas étant évidemment la possibilité d'utiliser les règles pour définir si un état existe ou pas. Par contre, si les états doivent être enregistrés, dans les problèmes avec un nombre important des deux types d'états, cela peut ralentir les calculs.

Pour finir, nous rappelons que, bien que les représentations factorisées permettent de traiter des problèmes plus grands, la présence d'états impossibles est difficile à éviter et dans ce cas les algorithmes qui permettent de gérer ce problème apportent une amélioration significative en ce qui concerne la vitesse de convergence, l'occupation d'espace mémoire et/ou le temps de calcul.

Ces travaux ont fait l'objet des publications suivantes : [Sigaud et al., 2009], [Kozlova et al., 2009b].

TEXDYNA : Décomposition hiérarchique dans FRL.

TeXDYNA est un algorithme qui découvre automatiquement la décomposition hiérarchique des problèmes de décision séquentielle en combinant les techniques d'abstraction de HMDP avec les méthodes FRL. TeXDYNA utilise le cadre des options [Sutton et al., 1999], [Precup, 2000] pour représenter les sous-tâches organisées hiérarchiquement. La notion d'option désigne une généralisation des actions primitives pour inclure des actions dont la durée dans le temps est variable. Ainsi, les options introduisent dans la représentation du problème l'abstraction temporelle qui correspond à des "raccourcis" à travers l'espace des états menant directement vers un but.

Algorithm .1: TeXDYNA

entrée: FMDP \mathcal{F} , hiérarchie des options \mathcal{E}

sortie : option à exécuter o

1 Apprentissage :

1.a mettre à jour le modèle de transitions global

$\mathcal{F} \leftarrow \text{UpdateFMDP}()$

1.b découvrir la hiérarchie des options

$\mathcal{E} \leftarrow \text{UpdateOptions}(\mathcal{F})$

2 Planification :

2.a mettre à jour la politique hiérarchique et

2.b choisir une option à exécuter

$o \leftarrow \text{SPITI AvecOptions}(\mathcal{F}, \mathcal{E})$

TeXDYNA décompose hiérarchiquement un FMDP en un ensemble d'options, tandis que la politique locale de chaque option est améliorée d'une façon incrémentale par SPITI-une version d'un algorithme FRL, SDYNA proposé dans [Degris, 2007]. La hiérarchie est

construite à partir de la structure interne du problème représentée sous forme d'arbres de décision. Ainsi, TeXDYNA peut être décomposé en deux processus simultanés : apprentissage et planification (Algorithme .1).

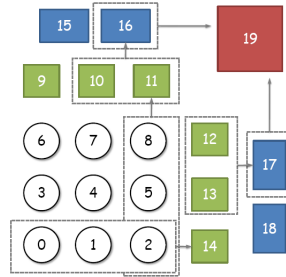


FIGURE 3: Le problème de LIGHT BOX : numéros et couleurs des “lumières” avec leurs dépendances.

Nous avons étudié les performances de TeXDYNA par rapport à sa vitesse de convergence, la forme des arbres de décision, la représentation de la fonction de transition (locale à chaque option ou globale pour toutes) et l’exploration d’espace d’états sur les problèmes de TAXI et de LIGHT BOX. Dans ce résumé nous ne mentionnons que les résultats sur la performance globale et la vitesse de convergence.

Dans le problème LIGHT BOX (Figure 3), le but de l’agent est d’allumer la lumière rouge numéro 19, sachant qu’elle ne peut être allumée que si une combinaison des lumières bleues est allumée, qui à leur tour ne peuvent être allumées une par une que si une combinaison des lumières jaunes propre à chacune est allumée et ainsi de suite. Avec 20 variables et 20 actions, il y a $2^{20} \approx 1$ million d’états, et donc 20 millions de paires état-action. TeXDYNA découvre une hiérarchie des options représentée sur la Figure 4.

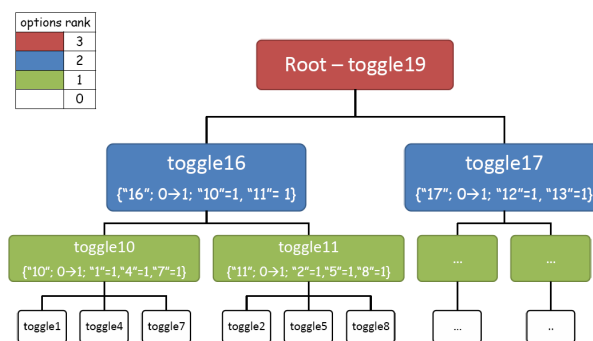


FIGURE 4: Exemple des options découverts dans le problème de LIGHT BOX.

La Figure 5 présente la performance en nombre de pas nécessaires pour terminer un épisode dans trois contextes expérimentaux : une politique aléatoire, TeXDYNA et DYNA-Q (un algorithme d’apprentissage par renforcement indirect basé sur des représentations

tabulaires [Sutton, 1991]) appliqué dans les versions stochastiques et déterministes du problème LIGHT BOX.

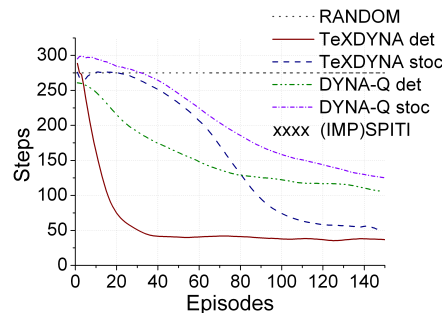


FIGURE 5: Convergence en nombre d'épisodes dans le problème LIGHT BOX.

Effectivement, TeXDYNA converge plus vite. Ce résultat est expliqué par les informations données dans le Tableau 2 qui représentent le temps moyen par pas de temps et les tailles des fonctions de politique, valeur et transition dans le cas stochastique. En effet, TeXDYNA opère sur des représentations plus petites et donc plus rapides à construire. L'algorithme commence par la découverte des options du deuxième niveau de la hiérarchie qui permettent d'allumer les lumières vertes. Dans ce niveau il y a 2^3 états et 3 sous-options, la solution étant triviale. L'apprentissage de ces politiques fournit un accès rapide à des politiques du niveau plus abstrait, celles qui permettent d'allumer les lumières bleues et la lumière rouge. Ainsi TeXDYNA trouve le but final plus rapidement en complétant les sous-buts. Ces résultats illustrent le cas où TeXDYNA permet d'obtenir une compression efficace des structures à apprendre, d'accélérer les calculs et d'augmenter la vitesse de convergence.

	TeXDYNA	SPITI	IMPSPITI	DYNA-Q
fonction de transition	780 ± 14	790 ± 25	790 ± 25	–
fonction de valeur	240 ± 20	> 15000	754 ± 54	> 10000
fonction de la politique	180 ± 8	> 15000	920 ± 62	> 10000
Temps/pas(sec)	0.04	> 100	> 100	> 2

TABLE 2: Performance sur le problème LIGHT BOX (la taille des fonctions de politique et de valeur en nombre total des nœuds dans les arbres de transition).

Pour résumer, notre approche est basée sur trois idées principales :

- l'utilisation de la fonction de transition sous forme d'arbres de décision pour découvrir les options,
- la localisation du modèle des transitions par rapport à chaque option d'où la réduction de l'espace d'états-actions,

- l'utilisation des options dans le processus de planification immédiatement après leur découverte.

Ces trois principes appliqués à la décomposition hiérarchique permettent d'associer un sous-FMDP à chaque option et ainsi d'attribuer à cette dernière une politique locale et la fonction de transition locale. En bref, TeXDYNA effectue une abstraction d'information contextuelle par la décomposition du problème en sous-problèmes d'un côté, et, de l'autre côté, introduit l'abstraction temporelle par l'utilisation des options découvertes automatiquement.

Ces travaux ont fait l'objet des publications suivantes : [Kozlova et al., 2008], [Kozlova et al., 2009a], [Kozlova et al., 2010].

Application au domaine de simulation

Comme cela a été mentionné dans l'introduction, pour répondre aux attentes industrielles de ce travail, toutes les solutions proposées sont basées sur des représentations structurées où les fonctions de transition, de valeur et de politique sont représentées sous forme d'arbres de décision pour être accessible à un utilisateur non-averti. Pour tester les techniques de HRL et FRL dans le domaine de simulation, nous proposons un problème jouet qui représente d'une façon simplifiée les contraintes de ce type d'application. Nous proposons un scénario qui symbolise un agent qui doit trouver un terroriste et éteindre le feu dans une station de métro. L'environnement est déterminé par la position de l'agent, la présence d'un terroriste, d'un passager et de feu dans un hall, une arme et un extincteur. L'agent peut effectuer les actions suivantes : tirer un coup de feu, se déplacer, aller chercher le pistolet ou l'extincteur et éteindre le feu. L'agent est récompensé s'il parvient à éteindre le feu.

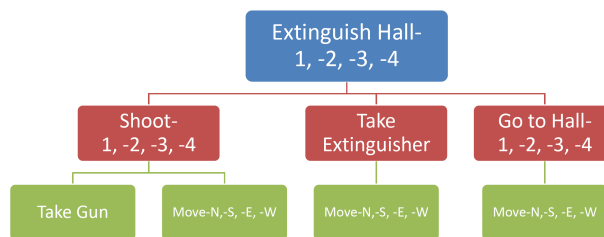


FIGURE 6: Scénario Terrorist vs. Guard. Hiérarchie des options.

Nous avons conduit les expériences sur la performance générale, l'adaptation à un environnement changeant et la sensibilité à la taille du problème. La hiérarchie apprise par TeXDYNA est présentée dans la Figure 6.

La Figure 7 montre les performances de TeXDYNA et DYNA-Q dans le cas où la fonction de récompense change. A partir de l'épisode 250, l'agent est récompensé pour tuer

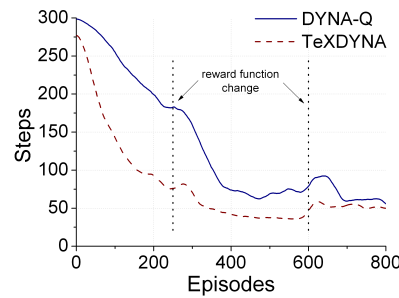


FIGURE 7: Scénario Terrorist vs. Guard. Convergence en nombre de pas de temps quand la fonction de récompense est modifiée à l'épisode 250 et 600.

le terroriste (et puni pour éteindre le feu) et à partir de l'épisode 600, la situation est inversée. En effet, l'utilisation des représentations structurées et la réutilisation des options déjà apprises permet à TeXDYNA d'adapter très vite la politique.

Ainsi, sur un scénario-type, nous montrons les capacités de TeXDYNA à construire une hiérarchie des tâches et à adapter la solution à un environnement dynamique. Nous évoquons aussi les limitations de notre approche quant aux simulations plus réalistes et indiquons les futures directions possibles pour pouvoir traiter des problèmes de simulation dans le cadre d'une application industrielle.

Discussion & Conclusion

Cette thèse explore le domaine de l'Apprentissage par Renforcement Factorisé et Hiérarchique. Nous avons proposé une nouvelle approche pour résoudre les problèmes de prise de décision dans les environnements stochastiques qui combine les techniques de HRL et de FRL. A travers cette thèse, nous avons démontré comment les méthodes HRL et FRL peuvent être utilisées conjointement pour résoudre les problèmes de grande taille dont la structure est complexe. Néanmoins, ces méthodes se basent sur les hypothèses telles que le temps discret, les valeurs de variables discrètes, les actions primitives atomiques, la définition claire des buts, l'exécution épisodique, etc., qui restreignent fortement la représentation du problème. Dans ce qui suit, nous indiquons les directions de recherche futures et commentons les contributions principales de ce travail.

IMPSVI, IMPSPITI & Etats impossibles

Dans cette partie, nous avons étudié le cas d'application des algorithmes de SDP dans des problèmes où certains états, pourtant représentés, n'apparaissent jamais. Dans ces circonstances, les arbres résultants incluent les combinaisons impossibles et, par conséquent, les calculs sont plus compliqués, demandent plus d'espace mémoire et la

solution finale est moins lisible. Les nouveaux algorithmes proposés dans cette thèse permettent une gestion efficace de ces états impossibles ce qui accélère les calculs tout en optimisant l'utilisation de la mémoire. Par ailleurs, la solution rendue par IMPSVI et IMPSPITI est plus lisible pour l'utilisateur puisqu'elle ne comporte pas d'état qui n'ont pas été rencontré dans le problème.

D'un autre point de vue, la question de pertinence de la représentation explicite des états impossibles reste ouverte. En effet, les algorithmes non-factorisés permettent de résoudre efficacement ce type de problèmes quand le nombre d'états global n'est pas important. Il est clair que le choix de la représentation du problème est primordial et le choix d'algorithme approprié en dépend. Une autre question vient des propriétés de généralisation des algorithmes SDP, où une branche de l'arbre de la fonction de valeur représente un ensemble d'états y compris les états non encore visités auxquels les valeurs d'états sont attribuées. Cette propriété n'est pas respectée par IMPSVI puisque tous les états non-visités sont exclus. Enfin, il reste des recherches à mener concernant l'exploration de l'espace d'états. En effet, tout au long de ce travail, nous avons utilisé une exploration ϵ -greedy qui est une approche aléatoire. Il serait intéressant d'y intégrer les techniques d'exploration "optimistes".

TeXDYNA

L'approche TeXDYNA combine les méthodes HRL et FRL pour accomplir une décomposition hiérarchique du FMDP simultanément à l'apprentissage de sa structure. La décomposition hiérarchique équivaut à l'identification des états sous-butts et la génération des options correspondantes. L'algorithme est basé sur la découverte automatique des options directement à partir de la structure de la fonction de transition représentée sous forme d'arbres de décision. Ainsi chaque option représente un sous-FMDP et calcule sa propre politique locale et sa propre fonction de transition locale. Nous avons démontré sur les problèmes benchmarks les avantages de ce type d'approche en matière d'occupation mémoire, de charge de calculs et de vitesse de convergence.

Toutefois, l'algorithme est dépendant du choix de la méthode d'apprentissage de la structure puisque c'est cette structure qui détermine la forme et la hiérarchie des options et leur nombre. De plus, comme la hiérarchie des options est strictement ordonnée, TeXDYNA ne peut pas être utilisé dans des problèmes où existent les dépendances entre les variables qui ne sont pas liées à des actions. Dans ce cas, les cycles peuvent apparaître dans la hiérarchie, ce qui empêchera la construction d'une politique valide. Par ailleurs, comme dans la contribution précédente, adapter des méthodes d'exploration "optimiste" pourrait apporter un avantage considérable dans les performances de TeXDYNA. De plus, ces propriétés peuvent être améliorées par une réorganisation de la hiérarchie lors de la construction de la politique, ainsi que la réutilisation par le transfert entre les options des parties pertinentes des connaissances accumulées.

Application dans le domaine de simulation

En dernier lieu, nous n'avons pas pu tester TeXDYNA sur un simulateur industriel, mais nous avons appliqué TeXDYNA à un problème jouet qui représente certaines contraintes des simulations du comportement humain. Cette application a exhibé les propriétés d'adaptation, de compromis et de pertinence demandées, mais les applications actuelles de TeXDYNA sont limitées à des environnements discrets et leur adaptation dans le cas continu s'avère nécessaire pour une utilisation dans les simulateurs réels.

En outre, l'intégration de nos techniques au sein d'une architecture motivationnelle et émotionnelle semble nécessaire pour simuler de façon convaincante le comportement humain, qui ne se contente pas d'optimiser une mesure de performance à tout instant.

Perspectives

En dehors des perspectives déjà évoquées, nous pensons que les directions suivantes présentent un intérêt pour des travaux futurs. Tout d'abord, en vue d'une application à des problèmes de robotique ou simulation de la vie réelle, il est impératif de pouvoir apprendre dans des environnement continus. De plus, une étude plus approfondie du dilemme exploration/exploitation doit être menée.

Bibliography

- [Andre et al., 1997] Andre, D., Friedman, N., and Parr, R. (1997). Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems*. 88
- [Andre and Russell, 2002] Andre, D. and Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *National Conference in Artificial Intelligence (AAAI)*, pages 119–125. MIT Press. 59
- [Barto et al., 1983] Barto, A., Sutton, R., and Anderson, C. (1983). Neuron-like adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13:834–846. 39
- [Barto et al., 1993] Barto, A. G., Bradtke, S. J., and Singh, S. P. (1993). Learning to act using real-time dynamic programming. Technical Report UM-CS-1993-002, University of Massachusetts Amherst. 38
- [Bellman, 1957] Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ. 31, 35
- [Bellman, 1961] Bellman, R. (1961). *Adaptive Control Processes: A Guided Tour*. Princeton University Press. 36
- [Bertsekas, 1995] Bertsekas, D. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific. 31
- [Boutillier, 1997] Boutillier, C. (1997). Correlated action effects in decision theoretic regression. In *Proceedings of the 13th Uncertainty in Artificial Intelligence*, pages 30–37. AUAI Press. 68
- [Boutillier et al., 1995] Boutillier, C., Dearden, R., and Goldszmidt, M. (1995). Exploiting structure in policy construction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1104–1111. 25, 43, 45, 150

- [Boutilier et al., 2000] Boutilier, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–10. 13, 45, 47, 48, 50, 68, 73, 74, 77, 79, 152
- [Bradtke and Duff, 1995] Bradtke, S. J. and Duff, M. O. (1995). Reinforcement learning methods for continuous-time markov decision problems. In *Advances in Neural Information Processing Systems 7*, pages 393–400. MIT Press. 60
- [Breiman et al., 1984] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth. 54
- [Butz et al., 2002] Butz, M. V., Goldberg, D. E., and Stolzmann, W. (2002). The Anticipatory Classifier System and Genetic Generalization. *Natural Computing*, 1(4):427–467. 69, 90
- [Croonenborghs et al., 2008] Croonenborghs, T., Driessens, K., and Bruynooghe, M. (2008). Learning relational options for inductive transfer in relational reinforcement learning. In *Proceedings of the 17th Annual International Conference on Inductive Logic Programming, ILP 2007, Corvallis, Oregon, USA, (Blockeel, et al. Eds.) Lecture Notes in Computer Science*, volume 4894, pages 88–97. 138
- [Dayan and Hinton, 1993] Dayan, P. and Hinton, G. E. (1993). Feudal reinforcement learning. In *Advances in Neural Information Processing Systems 5*, pages 271–278. Morgan Kaufmann. 60
- [Dean and Kanazawa, 1989] Dean, T. and Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, 5:142–150. 44, 150
- [Dearden, 2000] Dearden, R. (2000). *Learning and Planning in Structured Worlds*. PhD thesis, University of British Columbia. 88
- [Dearden, 2001] Dearden, R. (2001). Structured prioritized sweeping. In *Proceedings of the 18th International Conference on Machine Learning*. 88, 89
- [Degris, 2007] Degris, T. (2007). *Reinforcement Learning in Factored Markov Decision Processes (in french)*. PhD thesis, University Paris VI. 94, 106, 115, 120, 154
- [Degris et al., 2006a] Degris, T., Sigaud, O., and Wuillemin, P.-H. (2006a). Chi-square tests driven method for learning the structure of factored MDPs. In *Proceedings of the 22nd Conference Uncertainty in Artificial Intelligence*, pages 122–129, Massachusetts Institute of Technology, Cambridge. AUAI Press. 52
- [Degris et al., 2006b] Degris, T., Sigaud, O., and Wuillemin, P.-H. (2006b). Learning the structure of factored markov decision processes in reinforcement learning problems. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 257–264, Pittsburgh, Pennsylvania. ACM. 52, 53, 55, 94, 140, 141, 151

- [Dietterich, 1998] Dietterich, T. G. (1998). The MAXQ method for hierarchical reinforcement learning. In Kaufmann, M., editor, *Proceedings of the 15th International Conference on Machine Learning*. 59, 110, 113
- [Dietterich, 2000] Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303. 59
- [Ghavamzadeh, 2005] Ghavamzadeh, M. (2005). *Hierarchical Reinforcement Learning in Continuous State and Multi-Agent Environments*. PhD thesis, University of Massachusetts Amherst. 61
- [Gigerenzer, 2007] Gigerenzer, G. (2007). *Gut feelings: The intelligence of the unconscious*. New York: Viking Press. 143
- [Gratch and Marsella, 2001] Gratch, J. and Marsella, S. (2001). Tears and fears: modeling emotions and emotional behaviors in synthetic agents. In *Agents*, pages 278–285. 24
- [Guestrin et al., 2003] Guestrin, C., Koller, D., Parr, R., and Venkataraman, S. (2003). Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468. 45, 90, 120
- [Guestrin et al., 2002] Guestrin, C., Patrascu, R., and Schuurmans, D. (2002). Algorithm-directed exploration for model-based reinforcement learning in factored MDPs. In *Proceedings of the 19th International Conference on Machine Learning*, pages 235–242, Sydney, Australia. 45
- [Hengst, 2002] Hengst, B. (2002). Discovering hierarchy in reinforcement learning with HEXQ. In *Proceedings of the 19th International Conference on Machine Learning*, pages 243–250. 62, 97, 120, 151
- [Hoey et al., 1999] Hoey, J., St-Aubin, R., Hu, A., and Boutilier, C. (1999). SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*, pages 279–288, Stockholm. 52, 120
- [Howard, 1971] Howard, R. (1971). *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes*. 31, 34
- [Jonsson, 2006] Jonsson, A. (2006). *A causal approach to hierarchical decomposition in reinforcement learning*. PhD thesis, University of Massachusetts Amherst. 97, 120
- [Jonsson and Barto, 2006] Jonsson, A. and Barto, A. (2006). Causal graph based decomposition of factored MDPs. *Journal of Machine Learning Research*, 7:2259–2301. 62, 121, 151

- [Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285. 37
- [Konidaris and Barto, 2009] Konidaris, G. and Barto, A. (2009). Efficient skill learning using abstraction selection. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 1107–1112. 61
- [Korf, 1985a] Korf, R. E. (1985a). *Learning to Solve Problems by Searching for Macro-Operators*. Pitman, Boston, MA. 59
- [Korf, 1985b] Korf, R. E. (1985b). Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77. 59
- [Kozlova, 2006] Kozlova, O. (2006). Emotional action selection model based on Soar cognitive architecture (in french). Master’s thesis, University Paris V, Rene Descartes. 24
- [Kozlova et al., 2008] Kozlova, O., Sigaud, O., and Meyer, C. (2008). Apprentissage par renforcement hiérarchique dans les MDP factorisés (in french). In *Actes JFPDA*, pages 93–102, Metz, France. 111, 157
- [Kozlova et al., 2009a] Kozlova, O., Sigaud, O., and Meyer, C. (2009a). Automated discovery of options in factored reinforcement learning. In *Proceedings of the ICML/UAI/COLT Workshop on Abstraction in Reinforcement Learning*, pages 24–29, Montreal, Canada. 111, 157
- [Kozlova et al., 2010] Kozlova, O., Sigaud, O., and Meyer, C. (2010). TeX-DYNA:hierarchical reinforcement learning in factored MDPs. In *11th International Conference on the Simulation of Adaptive Behavior, From Animals to Animats (SAB 2010)*, page to appear. 113, 157
- [Kozlova et al., 2009b] Kozlova, O., Sigaud, O., Willemin, P.-H., and Meyer, C. (2009b). Considering unseen states as impossible in factored reinforcement learning. In *ECML PKDD 2009, Part I, LNAI 5781-0721*, pages 721–735. Springer. 68, 154
- [Laird et al., 1987] Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64. 24
- [Laird et al., 1986] Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46. 59
- [Mahadevan et al., 1997] Mahadevan, S., Marchalleck, N., Das, T., and Gosavi, A. (1997). Self-improving factory simulation using continuous-time average reward reinforcement learning. In *Proceedings of the 14th International Conference on Machine Learning*, pages 182–190. 60

- [McGovern and Barto, 2001] McGovern, A. and Barto, A. G. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the 18th International Conference on Machine Learning*, pages 361–368. 61
- [Moore and Atkeson, 1993] Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130. 38, 41
- [Ortony et al., 1988] Ortony, A., Clore, G. L., and Collins, A. (1988). *The cognitive structure of emotions*. Cambridge, UK, Cambridge University Press. 24
- [Parr and Russell, 1998] Parr, R. and Russell, S. (1998). Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10*, pages 1043–1049. MIT Press. 59
- [Peng and Williams, 1993] Peng, J. and Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 1(4):437–454. 41
- [Precup, 2000] Precup, D. (2000). *Temporal abstraction in reinforcement learning*. PhD thesis, University of Massachusetts, Amherst. 59, 151, 154
- [Puterman, 1994] Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons Inc. 25, 31, 59, 150
- [Quinlan, 1986] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106. 54
- [Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann. 54
- [Ravindran, 2004] Ravindran, B. (2004). *An Algebraic Approach to Abstraction in Reinforcement Learning*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, USA. 60, 61
- [Russell and Norvig, 2003] Russell, S. J. and Norvig (2003). *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall. 24
- [Sigaud, 2004] Sigaud, O. (2004). *Comportements adaptatifs pour des agents dans des environnements informatiques complexes (in french)*. Habilitation a diriger des recherches, Universite Pierre et Marie Curie, Paris 6. 25
- [Sigaud et al., 2009] Sigaud, O., Butz, M. V., Kozlova, O., and Meyer, C. (2009). *Anticipatory Learning Classifier Systems and Factored Reinforcement Learning*, pages 321–333. Springer. 68, 91, 154
- [Sigaud and Wilson, 2007] Sigaud, O. and Wilson, S. W. (2007). Learning Classifier Systems: a survey. *Journal of Soft Computing*, 11(11):1065–1078. 78

- [Simsek et al., 2005] Simsek, Ö., Wolfe, A. P., and Barto, A. G. (2005). Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 816–823. 61
- [Singh et al., 2005] Singh, S., Barto, A., and Chentanez, N. (2005). Intrinsically motivated reinforcement learning. *Advances in Neural Information Processing Systems*, 18:1281–1288. 108
- [Singh, 1993] Singh, S. P. (1993). Learning to solve markovian decision processes. Technical Report UM-CS-1993-077, University of Massachusetts Amherst. 61
- [Sloman, 2001] Sloman, A. (2001). Beyond shallow models of emotion. *Cognitive Processing*, pages 177–198. 24
- [St-Aubin et al., 2000] St-Aubin, R., Hoey, J., and Boutilier, C. (2000). Apricodd: Approximate policy construction using decision diagrams. In *Advances in Neural Information Processing Systems*, pages 1089–1095. 52
- [Strehl et al., 2007] Strehl, A. L., Diuk, C., and Littman, M. L. (2007). Efficient structure learning in factored-state MDPs. In *AAAI*, pages 645–650. 88
- [Sutton et al., 1999] Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211. 59, 97, 151, 154
- [Sutton, 1991] Sutton, R. S. (1991). DYNA, an integrated architecture for learning, planning and reacting. In *Working Notes of the AAAI Spring Symposium on Integrated Intelligent Architectures*. 38, 39, 52, 120, 156
- [Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA. 25, 38, 150
- [Szepesvári, 2009] Szepesvári, C. (2009). Reinforcement learning algorithms for MDPs. Technical Report TR09-13, Department of Computing Science, University of Alberta. 32
- [Szita and Lörincz, 2008] Szita, I. and Lörincz, A. (2008). The many faces of optimism: a unifying approach. In *Proceedings of the 25th International Conference on Machine Learning*, pages 1048–1055, New York, NY, USA. ACM. 87
- [Szita and Lörincz, 2009] Szita, I. and Lörincz, A. (2009). Optimistic initialization and greediness lead to polynomial time learning in factored MDPs. In Bottou, L. and Littman, M., editors, *Proceedings of the 26th International Conference on Machine Learning*, pages 1001–1008, Montreal. Omnipress. 120
- [Tate, 1977] Tate, A. (1977). Generating project networks. In *International Joint Conference on Artificial Intelligence*, pages 888–893. 58

- [Taylor and Stone, 2009] Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(1):1633–1685. 142
- [Thorndike, 1911] Thorndike, E. L. (1911). *Animal Intelligence*. Macmillan. 25, 37
- [Traum et al., 2004] Traum, D., Marsella, S., and Gratch, J. (2004). Emotion and dialogue in the MRE virtual humans. In *Proceedings of the Tutorial and Research Workshop on Affective Dialogue Systems*. Kloster, IRsee. 24
- [Tyrrell, 1993] Tyrrell, T. (1993). *Computational Mechanisms for Action Selection*. PhD thesis, Centre for Cognitive Science, University of Edinburgh. 23
- [Utgoff et al., 1997] Utgoff, P. E., Berkman, N. C., and Clouse, J. A. (1997). Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44. 52, 140
- [Velásquez, 1998] Velásquez, J. D. (1998). When robots weep: Emotional memories and decision-making. In *AAAI/IAAI*, pages 70–75. 24
- [Vigorito and Barto, 2008a] Vigorito, C. and Barto, A. (2008a). Hierarchical Representations of Behavior for Efficient Creative Search. In *AAAI Spring Symposium on Creative Intelligent Systems, Palo Alto, CA*. 63, 120, 121
- [Vigorito and Barto, 2008b] Vigorito, C. M. and Barto, A. G. (2008b). Autonomous Hierarchical Skill Acquisition in Factored MDPs. In *Yale Workshop on Adaptive and Learning Systems*, New Haven, Connecticut. 63, 95, 109, 120, 121, 151
- [Watkins and Dayan, 1992] Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292. 38
- [Wolfe and Barto, 2006] Wolfe, A. P. and Barto, A. G. (2006). Decision tree methods for finding reusable MDP homomorphisms. In *AAAI*. 61
- [Zang et al., 2009] Zang, P., Zhou, P., Minnen, D., and Isbell, C. (2009). Discovering options from example trajectories. In Bottou, L. and Littman, M., editors, *Proceedings of the 26th International Conference on Machine Learning*, pages 1217–1224, Montreal. Omnipress. 61

