
**Lisbon University Institute**

Department of Science and Technologies of Information


# Hierarchical Reinforcement Learning: Learning Sub-goals and State-Abstraction


David Walter Figueira Jardim


A Dissertation presented in partial fulfillment of the Requirements
for the Degree of
**Master in Computer Science**

**Supervisor:**
Prof. Dr. Luís Miguel Martins Nunes, auxiliary professor,
ISCTE-IUL

September 2010

ISCTE IUL
Instituto Universitário de Lisboa

# Hierarchical Reinforcement Learning: Learning Sub-goals and State-Abstraction

David Walter Figueira Jardim

September

2010

*"As for me, all I know is that I know nothing."*

Socrates

# *Resumo*

Os seres humanos possuem a incrível capacidade de criar e utilizar abstracções. Com essas abstracções somos capazes de resolver tarefas extremamente complexas que requerem muita antevisão e planeamento. A pesquisa efectuada em *Hierarchical Reinforcement Learning* demonstrou a utilidade das abstracções, mas também introduziu um novo problema. Como encontrar uma maneira de descobrir de forma autónoma abstracções úteis e criá-las enquanto aprende? Neste trabalho, apresentamos um novo método que permite a um agente descobrir e criar abstracções temporais de forma autónoma. Essas abstracções são baseadas na *framework* das Options. O nosso método é baseado no conceito de que para alcançar o objectivo, o agente deve passar por determinados estados. Ao longo do tempo estes estados vão começar a diferenciar-se dos restantes, e serão identificados como sub-objectivos úteis. Poderão ser utilizados pelo agente para criar novas abstracções temporais, cujo objectivo é ajudar a atingir esses objectivos secundários. Para detectar sub-objectivos, o nosso método cria intersecções entre os vários caminhos que levam ao objectivo principal. Para que uma tarefa seja resolvida com sucesso, o agente deve passar por certas regiões do espaço de estados, estas regiões correspondem à nossa definição de sub-objectivos.

A nossa investigação focou-se no problema da navegação em salas, e também no problema do táxi. Concluímos que um agente pode aprender mais rapidamente em problemas mais complexos, ao automaticamente descobrir sub-objectivos e criar abstracções sem precisar de um programador para fornecer informações adicionais e de criar as abstracções manualmente.

**Palavras-chave:** Aprendizagem Automática, Aprendizagem Hierárquica por Reforço, Abstracções, Sub-objectivos.

# *Abstract*

Human beings have the incredible capability of creating and using abstractions. With these abstractions we are able to solve extremely complex tasks that require a lot of foresight and planning. Research in Hierarchical Reinforcement Learning has demonstrated the utility of abstractions, but, it also has introduced a new problem. How can we find a way to autonomously discover and create useful abstractions while learning? In this dissertation we present a new method that allows an agent to discover and create temporal abstractions autonomously based in the options framework. Our method is based on the concept that to reach the goal, the agent must pass through certain states. Throughout time these states will begin to differentiate from others, and will be detected as useful subgoals and be used by the agent to create new temporal abstractions, whose objective is to help achieve these subgoals. To detect useful subgoals, our method creates intersections between several paths leading to a goal. In order for a task to be solved successfully the agent must pass through certain regions of the state space, these regions will correspond to our definition of subgoals.

Our research focused on domains largely used in the study of the utility of temporal abstractions, which is the room-to-room navigation problem, and also the taxi problem. We determined that, in the problems tested, an agent can learn more rapidly in more complex problems by automatically discovering subgoals and creating abstractions without needing a programmer to provide additional information and handcraft the abstractions.

**Keywords:** Machine Learning, Reinforcement Learning, Abstractions, Subgoals.

# *Acknowledgements*

I would like to thank my advisor Prof. Dr. Luís Miguel Martins Nunes for his efforts, support and guidance all the way through this thesis. Passionately introducing me to this exciting area of Reinforcement Learning. His encouragement and advice have been extremely valuable to overcome all the obstacles throughout this endeavor.

I would like to thank Prof. Dr. Sancho Oliveira for his insightful contributions to this dissertation, where the ongoing discussions in meetings helped me to stimulate my research and development.

I gratefully acknowledge the love and support of all my family, specially my parents, Álvaro and Fátima Figueira, that with much sacrifice granted me the opportunity to have a better life through education. I would also like to thank my uncles, that in this last year treated me as their own son.

Last but not least, my wonderful girlfriend Joana, for her love, friendship, support, and encouragement throughout several phases of my life.

# Contents

# List of Figures

# List of Algorithms

# Abbreviations

| | |
|---|---|
| **AI** | **A**rtificial **I**ntelligence |
| **DD** | **D**iverse **D**ensity |
| **HAMs** | **H**ierarchical **A**bstract **M**achines |
| **HRL** | **H**ierarchical **R**einforcement **L**earning |
| **HSMQ** | **H**ierarchical **S**emi-**M**arkov **Q**-Learning |
| **MDP** | **M**arkov **D**ecision **P**rocess |
| **ML** | **M**achine **L**earning |
| **RL** | **R**einforcement **L**earning |
| **SG** | **S**ub **G**oal |
| **SMDP** | **S**emi **M**arkov **D**ecision **P**rocess |

*Dedicated to my family*

# Chapter 1

# Introduction

*"Only education is capable of saving our societies from possible collapse, whether violent, or gradual."*

Jean William Fritz Piaget

Imagine the everyday task of preparing your breakfast. Something as mundane as this reveals several degrees of complexity usually ignored by humans. If it was necessary to plan every single action of this process this would reveal to be very difficult or maybe even impossible. Eating cereals for example, is composed of several actions: walking to the closet, opening it, selecting the cereal box, grabbing it and retrieving it. Then it would be necessary to get a bowl, a spoon, the milk, etc. Each step is guided by goals, like pouring milk, and is in service of other goals, such as eating the cereals. Human beings have the incredible capability of creating and using abstractions. With these abstractions we are able to solve extremely intricate and complex tasks that require a lot of foresight and planning. Mcgovern (2002) refers that an abstraction can be a compact representation of the knowledge gained in one task that allows the knowledge to be re-used in related tasks. This representation allows the agent to plan and learn at multiple levels of abstraction.

Artificial intelligence (AI) researchers have tried to address the limitations of Reinforcement Learning in solving large and complex problems. One of these efforts resulted in the appearance of a new research area: Hierarchical Reinforcement Learning. HRL introduces various forms of abstraction that allow an agent to ignore irrelevant aspects of the current task, and plan on a more abstract level.

Two major forms of abstraction have been studied, namely:

- State abstraction
- Temporal abstraction

State abstraction is employed when some aspects of the state of the environment are ignored, creating a generalization or aggregation over state variables. For example, the movements required to rotate a door knob can also be applied to rotate a water tap, a generalization can be used to create an abstraction of *rotate-object*.

Temporal abstraction refers to an encapsulation of primitive actions into a single and more abstract action with temporally extended courses of action. Representing knowledge flexibly at multiple levels of temporal abstraction has the potential to greatly speed planning and learning on large problems. Using the previous example, *open-the-door* represents a temporal abstraction because of the planning involved, it does not require us to plan all the muscle movements consciously, we have learned these movements in our childhood and aggregated them in a sequence that usually leads to a good result (the door opens).

Research in HRL is currently focused on demonstrating how to use abstractions to improve learning, while doing that, it has introduced a new kind of problem:

How can we find a way of autonomously discover and create useful abstractions while learning?

Usually those abstractions needed to be handcrafted by the programmer, which turned out to be a daunting task. In this dissertation we present a method that allows an agent to discover and create temporal abstractions autonomously through learning based in the options framework from Sutton et al. (1999). Our method is aimed at problems where, the agent must pass through certain states to reach the goal.

Through time these states will begin to differentiate from other, and will be detected as useful subgoals , and will be used by the agent to create new temporal abstractions that achieve these subgoals. To detect useful subgoals, our method creates intersections between several paths leading to a goal. In order for a task to be solved successfully the agent must pass through certain regions of the state

space, these regions will correspond to our definition of subgoals. The core of our method is to search in the agent's experience (paths) for frequent states or regions. Our approach concurrently uses the experience obtained by a learning agent to identify useful subgoals and create options to reach them.

Our research focused on domains largely used in the study of the utility of temporal abstractions, which is the room-to-room navigation problem (Precup and Sutton, 2000), and also the taxi problem (Dietterich, 2000b). We present results where our methods are used to create temporal abstractions on the referred domain. By automatically discovering subgoals and creating abstractions, an agent can learn more rapidly in more complex problems without the need for the programmer to provide additional information and handcraft the abstractions.

## 1.1 Objectives

The purpose of the work developed in this dissertation, was, initially, to survey the most influential work done in the area of Hierarchical Reinforcement Learning, thus creating a solid basis of knowledge in the area of HRL. After gathering all the important information and defining the course of action, efforts were concentrated in overcoming some of the limitations encountered in Hierarchical Reinforcement Learning.

Artificial intelligence researchers have addressed the limitations of RL by creating the Hierarchical Reinforcement Learning, introducing various forms of abstraction intro problem solving and planning systems. Abstraction allows the agent to ignore details irrelevant to the task that it is trying to solve, reducing the complexity and the resources required. It also provides the idea of a subroutine that can call other subroutines which in turn can execute primitive actions providing a hierarchy dividing the problem in smaller parts, more easily learned and solved.

The hierarchization of the problem has brought several improvements in the performance of the agent while solving a specific task. But with those, genuine advances, other problems have risen. For example how to automatically discover the sub-goals of a problem? How to create an agent that, in a fully automated way, discovers hierarchies in the problem that it is trying to solve? These questions and others will be approached in more detail on the following sections of this dissertation in which we will describe the approaches of several authors that tried

to tackle these questions. We will also describe, the main drawbacks of these approaches, and our original contributions to this study.

Throughout this study, we hope to provide an insight of HRL and add a relevant contribution to the area.

## 1.2 Scientific Contribution

This dissertation presents the following contributions:

- Identifies and exposes the current limitations of Reinforcement Learning
- A review to several approaches to temporal abstraction and hierarchical organization developed recently by machine learning researchers
- Introduces a novel automated technique for discovering useful subgoals in a well-known problem
- Presents satisfactory results which improve the learning process of the agent

## 1.3 Structure of the Dissertation

In Chapter 2 we give an overview of reinforcement learning, hierarchical reinforcement learning and several existing approaches. We also describe the options framework created by Sutton et al. (1999), that we use to represent the temporal abstractions. In Chapter 3, we present our work in developing a software framework of HRL intended to support our investigation and experiments.

Chapter 4 describes some methods from several authors, for automatically discovering subgoals, with attempts of replicating their results. Then we present and describe our own algorithm of autonomous subgoal discovery using path intersections, with illustrated results and experiments in several grid-worlds tasks. In Chapter 5 we describe our method of dynamic option creation after discovering useful subgoals. After presenting the method, several experiments are discussed. Chapter 6 which is the final chapter, concludes and discusses our plans for future research in this area.

The next chapter discusses related research with several approaches in HRL and the following chapters present our methods and experimental results.

# Chapter 2

# Background Theory

*"Live as if you were to die tomorrow. Learn as if you were to live forever."*

M.K. Gandhi

## 2.1 Introduction

According to Sutton (1998) learning is the acquisition of knowledge, behaviors, and mental processes due to an interaction with the environment. By experimenting the world it is possible to derive meaning of the surrounding environment. One of the first contributors to the learning theory was Ivan Pavlov, which became widely known for first describing the phenomenon of classical conditioning. This phenomenon is a form of associative learning that combines a conditioned stimulus with an unconditioned stimulus, when these are repeatedly paired, eventually the two stimuli will become associated and the organism begins to produce a behavioral response to the conditioned stimulus.

The basis of reinforcement learning is the interaction with an environment. If we give it a thought, it will make perfect sense because throughout our lives, different kinds of interactions are the main source of knowledge about our environment and ourselves. In every single action that we take in order to achieve goals, we are aware that we will receive a response from our environment for the action taken, and afterwards we will frequently shape our behavior accordingly.

From an historical point of view we can consider that reinforcement learning which is an active area of machine learning, evolved mainly from two separated branches, one referring to trial and error originated in the psychology of animal learning. The other branch concerns the problem of optimal control using value function and dynamic programming. The joining of these two branches in the late 1980s brought us to Reinforcement Learning in the form that is known to the present days (Sutton, 1998).

## 2.2 Reinforcement Learning

Summarizing the Reinforcement Learning problem is easy: learning from interaction to achieve a goal or solve a problem. In RL exists an entity called the *agent* which is responsible for learning and make the decisions. While learning the agent interacts with the *environment* which comprises everything outside of it. Interaction occurs continually between these two entities over a sequence of discrete time steps, where the agent selects actions and the environment responds to those actions with a numerical reward and by giving new situations to the agent.

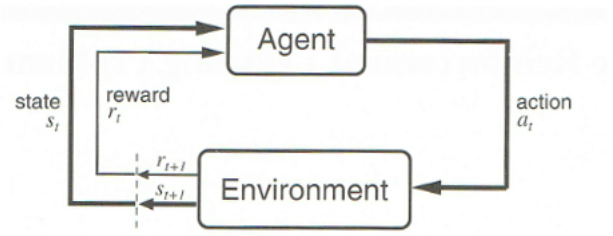



FIGURE 2.1: The Agent-Environment Interface (Sutton, 1998).

One way for the environment to give feedback to the agent is by giving rise to rewards, numerical values that the agent solely tries to maximize over time. The specification of the interface between these two entities is important because it allows us to formally define a task: comprised by the *actions* resulting from the choices made by the agent; the *states* represent all the possible situations in which the agent can be found.States are also the basis for making choices. Everything inside the agent is fully known by it and controllable, on the contrary, outside of it, the environment is usually not completely controllable and may even be only partially observable.

Other important concept is the *policy*, usually denoted by $\pi_t$. A policy is a mapping from each state, $s \in S$, and action, $a \in A(s)$, to the probability $\pi(s, a)$ of

executing action $a$ when in state $s$, mapping situations to actions. Reinforcement Learning methods specify how the agent will change its policy as a result of its experience over time.

### 2.2.1 Goals and Rewards

The purpose of the agent or goal is represented by a numerical value that acts like a reward passing from the environment to the agent. So the agent's goal is to maximize the total amount of reward it receives. Not the immediate reward when performing an action, but the cumulative reward on the long run. This can be seen in our example from the Rooms Problem, where the agent learns the way or the path that it must cover in order to reach a state previously defined as the goal. The reward is often zero until it reaches the goal, when it becomes +1 for example. When assigning rewards to certain states it is important to prioritize the main-goal, because if reward is given to certain sub-goals the agent might find a way of achieve them without achieving the main goal. We are assuming that the reward source is always placed outside of the agent, more specifically in the environment, assuring that the agent has no direct control over it.

### 2.2.2 Markov Decision Process

The Markov property refers to a property of a stochastic process, in which the conditional probability distribution of future states of the process, rely solely upon the present state. In reinforcement learning a tasks that satisfies the Markov property is called a *Markov Decision Process* (MDP). When the state and action space are finite it is called a *finite Markov Decision Process* and it is of extreme importance to the theory of reinforcement learning.

A finite MDP is a 4-tuple, defined by:

- A finite set of states **S**
- A finite set of actions **A**
- Equation 2.1 that represents the probability of each next possible state, $s'$ to any given state and action

- An expected value of the next reward, right after a transition occurs to the next state with the probability previously defined in 2.1, represented by the equation 2.2

$$P^{a}{}_{ss'} = Pr\{s_t + 1 = s' | s_t = s, a_t = a\} \tag{2.1}$$

$$R^{a}{}_{ss'} = E\{r_t + 1 | s_t = s, a_t = a, s_t + 1 = s'\} \tag{2.2}$$

These two last tuples are called *transition probabilities* and they specify the dynamics of a finite MDP.

### 2.2.3 Value Functions

Most part of the reinforcement learning algorithms are based on estimating *value functions* - functions of states (or state-action pairs) that estimate how good it is for an agent to perform a given action in a given state. We already know that the expected reward of the agent depends on what actions it will perform, accordingly value functions are defined by a particular policy.

The *value* of a state $s$ under a policy $\pi$, denoted by $V^{\pi}(s)$, is the expected return when starting in $s$ and following policy $\pi$ onwards. $V^{\pi}(s)$ can be defined by 2.3 and represents the $state - value\ function\ for\ policy\ \pi$.

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \tag{2.3}$$

In the same manner the value of taking action $a$ in state $s$ under a policy $\pi$, denoted as $Q^{s,a}$, as the expected return value starting from $s$, taking the action $a$, and thereafter the policy $\pi$ defined by 2.4 called the $action - value\ function\ for\ policy\ \pi$.

$$Q^{\pi}(s, a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\} \tag{2.4}$$

Both functions previously depicted can be estimated from the agent's experience satisfying particular recursive relationships. This means that for any policy $\pi$

and any state $s$ exists a relation between the value of $s$ and the value of its possible next state. The great advantage of this is that when the agent performs update operations, these operations transfer value information back to a state from its successor states.

### 2.2.4 Optimal Value Functions

As previously said, solving a reinforcement learning problem means, in short, finding a policy that achieves a great amount of reward in a long term. There are usually multiple policies to solve the same problem, some better than others, but there is always at least one policy better than all the others. This is called an *optimal policy* denoted by $\pi^*$, and it will have an $optimal\ state - value\ function$ $V*$ defined by

$$V^*(s) = \max_{\pi} V^{\pi}(s), \tag{2.5}$$

for all $s \in S$. An optimal policy also has an $optimal\ action - value\ function$, denoted $Q^*$ and defined as

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a), \tag{2.6}$$

for all $s \in S$ and for all $a \in A$. This function gives the expected return for taking action $a$ in state $s$ and thereafter following the optimal policy $\pi^*$.

## 2.3 RL Algorithms

There are many RL algorithms, but only two of them will be approached. More specifically Q-learning (Watkins and Dayan, 1992) due to the fact that was implemented in our project and also the well-known SARSA (Sutton, 1996) to get a different perspective.

### 2.3.1 Q-Learning

This algorithm remains as one of the most important discoveries in reinforcement learning and it is the one used in our implementation. With an $action - value\ function\ Q$ defined in 2.7, the agent observes a current state $s$, executes action $a$,

receives immediate reward $r$, and then observes a next state $s'$. Only one state-action pair is updated at a time, while all the others remain unchanged in this update.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[r_{t+1} + \gamma \max_a \; Q(s_{t+1}, a) - Q(s_t, a_t)\Big] \tag{2.7}$$

- $r_t - reward$
- $Q(s_t, a_t) - old\ value$
- $\gamma - discount\ factor$
- $\max_a\ Q(s_{t+1}, a) - max\ future\ value$

If eventually all the values from the actions available in the state-action pairs are updated, and $\alpha$ decaying with the increasing number of episodes, has been shown that $Q_t$ converge with probability 1 to $Q^*$. As long as these conditions are satisfied, the policy followed by the agent throughout the learning process is irrelevant, which is why this is called an off-policy temporal-difference control.The Q-learning algorithm from Sutton (1998) is shown in procedural form in alg. 1.

**Algorithm 1** Q-learning

```
Initialize Q(s, a)
for (each episode) do
  Initialize s
  for (each step of the episode) do
    Choose a from s using policy derived from Q
    Take action a, observe r, s'
    Q(s, a) ← Q(s, a) + α[r + γ max_a' Q(s', a') − Q(s, a)]
    s ← s'
  end for
  until s is terminal
end for
```

### 2.3.2 Sarsa

The Sarsa algorithm is very similar to Q-learning, except that the maximum action-value for the next state on the right side of function 2.7 is replaced by the action-value of the actual next state-action pair. In every transition from a

non-terminal state $s_t$ the value of the state-action pairs is updated. If $s_t + 1$ is the terminal state, then $Q(s_{t+1}, a_{t+1})$ is defined as zero. When a transition from one state-action pair occurs to the next, every element of the quintuple of events $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ is used. This quintuple is the reason why the algorithm was called as Sarsa defined as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[r_{t+1} + \gamma \;\; Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big] \tag{2.8}$$

Unlike Q-learning, Sarsa is an on-policy temporal-difference control algorithm, meaning that the policy followed during the learning is relevant. This affects the convergence properties of Sarsa, depending on the nature of the policy's ($\epsilon$ - greedy or $\epsilon$ - soft). Sarsa converges with probability 1 to an optimal policy and action-value function as long as each action is executed infinitely often in every state equally visited, and the policy converges in the limit to the greedy policy.

**Algorithm 2** Sarsa

```
Initialize Q(s, a)
for (each episode) do
  Initialize s
  Choose a from s using policy derived from Q
  for (each step of the episode) do
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q
    Q(s,a) ← Q(s,a) + α[r + γ Q(s',a') − Q(s,a)]
    s ← s'; a ← a';
  end for
  until s is terminal
end for
```

## 2.4 Hierarchical Reinforcement Learning

Until now applications of Reinforcement Learning were limited by a well-known problem, called the curse of dimensionality. This term was coined by Richard Bellman and represents the problem caused by the exponential growth of parameters to be learned, associated with adding extra variables to a representation of a state. Applying RL with a very large action and state space turned to be an impossible task.

To overcome this limitation and increase the performance some researchers started to develop different methods for Hierarchical Reinforcement Learning (Barto and Mahadevan, 2003). HRL introduces various forms of abstraction and problem hierarchization. With abstraction the agent only needs to know the minimum required to solve a problem, reducing unnecessary overhead, complexity and resources. Hierarchization will divide the main problem in sub-problems that can be solved using regular RL. Each sub-problem has its own sub-goal. The sequential resolution of several sub-goals takes us to the solution of the main problem.

Some of the most relevant approaches to HRL will be surveyed in section 2.5, all of which are based on the underlying framework of SMDPs addressed in the following sub-section.

### 2.4.1 Semi-Markov Decision Process

A semi-Markov decision process (SMDP) can be seen as a special kind of MDP, appropriate for modeling continuous-time discrete-event systems. Where the actions in SMDPs are allowed to take variable amounts of time to be performed with the intent of modeling temporally-extended courses of action. According to Barto and Mahadevan (2003) the system remains in each state for a random interval of time, at the end of the specified waiting time occurs an instantaneous transition to the next state.

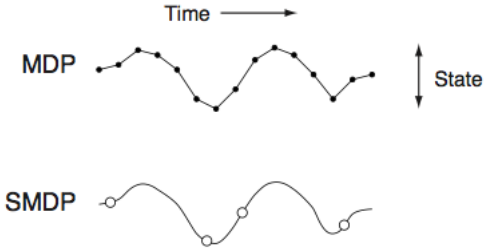



FIGURE 2.2: MDP VS SMDP (Sutton et al., 1999).

Let the random variable $\tau$ denote the positive waiting time for a state $s$ when action $a$ is executed. A transition from state $s$ to state $s'$ occurs after $\tau$ time steps when action $a$ is executed. This can be seen as a joint probability as $P(s', \tau|s, s)$. Then the expected immediate reward, $R(s, a)$ , give us the amount of discounted reward expected to accumulate over the waiting time in $s$ given action $a$. So the Bellman equations for $V^*$ and $Q*$ are represented in 2.9 and 2.10 respectively

$$V^*(s) = \max_{a \in A_s}[R(s,a) + \sum_{s',\tau} \gamma^\tau P(s', \tau|s, a)V^*(s')], \tag{2.9}$$

for all $s \in S$; and

$$Q^*(s,a) = R(s,a) + \sum_{s',\tau} \gamma^\tau P(s', \tau|s, a) \max_{a' \in A_{s'}} Q^*(s', a') \tag{2.10}$$

for all $s \in S$; and $a \in A_s$.

## 2.5 Approaches to HRL

### 2.5.1 Reinforcement Learning with Hierarchies of Machines

Optimal decision-making in human activity is unmanageable and difficult due to the complexity of the environment that we live on. The only way around is to provide a hierarchical organization for more complex activities.

In Parr and Russell (1998) section 3, the authors introduce the concept of Hierarchical Abstract Machines (HAMs).

HAMs consist of nondeterministic finite state machines, whose transitions may invoke lower-level machines. They can be viewed as a constraint on policies. For example a machine described as *"repeatedly choose right or down"* eliminates from consideration all policies that go up or left. This allows one to specify several types of machines, one that could allow only a single policy, or not constrain the policy at all. Each machine is defined by a set of states, a transition function, and a start function that determines the initial state of the machine.

There are 4 HAM state types:

- **Action** - states execute an action in the environment
- **Call** - states execute another HAM as a subroutine
- **Choice** - states non-deterministically select a next machine state

- **Stop** - states halt execution of the machine and return control to the previous call state

HAMs can be used in the context of reinforcement learning in order to reduce the time taken to learn a new environment, by using the constraints implicit on the machines to focus its exploration of the state space.

A variation of Q-Learning was created, called the HAMQ-Learning that learns directly in the reduced state space. A HAMQ-Learning agent is defined by **t**, the current environment state **n**, the current machine state $S_c$ and $\beta_c$, the environment state and machine state at the previous choice point **a**, the choice made at the previous choice point $r_c$ and $S_c$, the total accumulated reward and discount since the previous choice point. It also has an extended Q-table $Q([s, m], a)$.

From the experiments conducted by the authors Parr and Russell (1998) in section 5, they have measured the performance between Q-Learning and HAMQ-Learning, the last one appears to learn much faster, where Q-Learning required 9,000,000 iterations to reach the level achieved by HAMQ-Learning after 270,000 iterations.

With HAMs the constraining of the set of policies considered for a MDP is achieved efficiently, giving a boost in the speed of decision making and learning while providing a general method of transferring knowledge to an agent. But the application of HAMs to a more general partially observable domain remains as a more complicated and unsolved task.

### 2.5.2 Options

The options framework is used to define the temporally abstract actions that our agents automatically discover in this dissertation. The term *options* is used to define a generalization of primitive actions to include temporally extended courses of actions. Formalized by Sutton et al. (1999). An option consists of three components:

- a policy $\pi : S \times A \rightarrow [0, 1]$
- a termination condition $\beta : S^+ \rightarrow [0, 1]$

- and a initiation set $I \subseteq S$

The agent has the possibility of executing an option in state $s$ if and only if $s \in I$, when an option is taken, actions are selected according to the policy $\pi$ until the option terminates stochastically according to $\beta$. Assuming the agent current state is $s$, the next action is with a probability $\pi(s, a)$, the environment makes a transition to state $s'$, where the option could terminate with probability $\beta(s')$ or else continues , determining the next action $a'$ with probability $\pi(s', a')$ and so on. When the current option terminates, the agent can select another option according to it's inherent quality. The initiation set and termination condition of an option restrict its range of application in a useful way, limiting the range over which the option's policy need to be defined. An example of this from Sutton et al. (1999) where a handcrafted policy $\pi$ for a mobile robot to dock with its battery charger might be defined only for states $I$ in which the battery charger is in sight or available for use. It is also assumed that for Markov options all states where an option can continue are also state where the option might be started. With this the policy $\pi$ need only to be defined over $I$ rather than over all $S$.

An option is defined as a *Markov option* because its policy is Markov, meaning its sets action probabilities based only on the current state of the core MDP. In order to increase flexibility, much needed in hierarchical architectures, *semi - Markov options* must be included whose policies will set action probabilities based on the entire history of states, actions and rewards since the option was initiated. This is useful for options terminate after some period of time, and most importantly, when policies over options are considered.

After creating a set of options, their initiation sets implicitly define a set of available options $O_s$ for each state $s \in S$ (Sutton et al., 1999). These $O_s$ are similar to the set of available action $A_s$, with that similarity making it possible to unify these two sets by assuming that primitive actions can be considered as a special case of options. So each action $a$ will correspond to an option that always lasts exactly one time step. Making the agent's choice to be entirely among options, some of them persisting only for a single time step, others being more temporally extended.

Assuming that policies over options are considered, a policy $\mu$ over options selects option $o$ in state $s$ with probability $\mu(s, o)$, $o$'s policy in turn will select other options until $o$ terminates. In this way a policy over options, $\mu$, determines

a conventional policy over actions, or a *flat policy* $\pi = flat(\mu)$. Flat policies corresponding to policies over options are generally not Markov even if all the options are Markov. The probability of a primitive action at any time step depends on the current core state plus the policies of all the options currently involved in the hierarchical specification.

The authors have defined the value of a state $s \in S$ under a semi-Markov flat policy $\pi$ as the expected return if the policy started in s:

$$V^{\pi}(s) = \Big\{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... | \varepsilon(\pi, s, t) \Big\}, \tag{2.11}$$

where $\varepsilon(\pi, s, t)$ is the event of $\pi$ being initiated at time $t$ in $s$. The value of $s$ for a general policy $\mu$ over options can be defined as the value of the state under the corresponding flat policy: $Q^{\mu}(s, o) = V^{flat()\mu}(s)$ for all $s \in S$. Similarly the option-value function for $\mu$ can be defined as:

$$Q^{\mu}(s, o) = \Big\{ r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... | \varepsilon(o\mu, s, t) \Big\}, \tag{2.12}$$

where $o\mu$ is the semi-Markov policy that follows $o$ until it terminates at $t + 1$ with probability $\beta(ha_t r_{t+1} s_{t+1})$ and then continues according to $\mu$. In this dissertation we use only Markovian options. When updating primitive actions, the update rule is the same as that for Q-learning. For updating options the function 2.13 is used where $R_0 = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{rt+3} + ... + \gamma^{n-1} r_{t+n}$ is the number of time steps that option $o$ was executed, $s$ is the state where the option was initiated, and $s_{t+n}$ is the state where the option terminated.

$$Q(s, 0) = Q(s, o) + \alpha \left[ R_0 + \gamma^n \max_{a' \in A(s_{t+n})} Q(s_{t+n}, a') - Q(s, o) \right], \tag{2.13}$$

Usually the learned policies of the options, are policies for efficiently achieving *subgoals*, where a subgoal often is a state or a region of the state space, meaning that when the agent reaches that state or region is assumed to facilitate achieving of the main goal. In this dissertation the Options framework will be used to create policies to achieve the subgoals detected by our algorithm.

### 2.5.3 An Overview of MAXQ HRL

#### 2.5.3.1 Task Decomposition and Reinforcement Learning

Discovering and exploiting hierarchical structure within a Markov Decision Process (MDP) is the main goal of hierarchical reinforcement learning. Given an MDP, the programmer will be responsible for designing a task hierarchy (Figure 2.3) for a specific problem. This is achieved by decomposing the main task into several subtasks that are, in turn, also decomposed until a subtask is reached that is composed only by primitive actions as stated by Dietterich (2000a) in section 3. After obtaining the task hierarchy the goal of the reinforcement learning algorithm is defined as finding a recursively optimal policy.

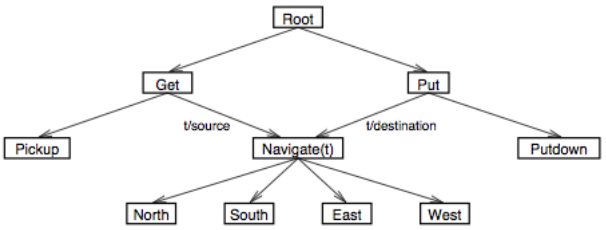



FIGURE 2.3: Representation of a task in the Taxi domain (Dietterich, 2000a).

To learn such policies the authors developed the Hierarchical Semi-Markov Q-Learning (HSMQ) algorithm to be applied simultaneously to each task contained in the task hierarchy. Each subtask **p** will learn its own Q function $Q(p, s, a)$, which represents the expected total reward of performing subtask **p** on an initial state **s**, executing action **a**.

#### 2.5.3.2 Value Function Decomposition and Reinforcement Learning

For the HSMQ algorithm a hierarchical reinforcement-learning problem is seen as a collection of simultaneous, independent Q-Learning problems. This algorithm does not provide a representational decomposition of the value function, here is where the MAXQ value function decomposition appears in Dietterich (2000a) in section 4, by exploiting the regularity of some identical value functions, the authors have managed to represent the value function only once.

MAXQ decomposes the $Q(p, s, a)$ value into the sum of two components:

- Expected total reward $V(a, s)$ received while executing **a**
- Expected total reward $C(p, s, a)$ of completing parent task **p** after **a** has returned

Representing $Q(p, s, a) = V(a, s) + C(p, s, a)$. Applied recursively it can decompose the Q function of the root task into a sum of all the Q values for its descendant tasks and represent the value function of any hierarchical policy as described by the equation 2.14.

$$V(p, s) = \max_a [V(a, s) + C(p, s, a)] \tag{2.14}$$

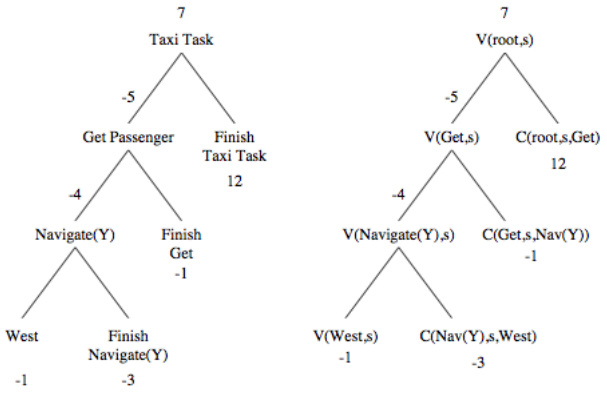



FIGURE 2.4: Example of MaxQ value function decomposition from Dietterich (2000a).

#### 2.5.3.3 State Abstraction and Hierarchical Reinforcement Learning

There are some subtasks where the value function does not depend on all of the state variables in the original MDP. These subtasks can be used to reduce the amount of memory required to store the value function as the amount of experience required to learn the value function by using state abstraction referred by Dietterich (2000a) in section 5.

State abstraction can be divided into three forms:

- **Irrelevant variables** - Variables that have no effect on the rewards received by the subtask and on the value function can be ignored

- **Funnel abstractions** - An action that causes a larger number of initial states to be mapped into a small number of states
- **Structural abstractions** - Results from constraints introduced by the structure of the hierarchy (Ex: There is no need to represent $C(root, s, put)$ in states where the passenger is not in the taxi)

With state abstraction the amount of memory required in the taxi problem is dramatically reduced, for example:

- Flat Q-Learning 3000 **Q** values
- HSMQ requires 14000 distinct **Q** values
- MAXQ Hierarchy requires 632 values for **C** and **V**

This also means that the learning process is faster, because learning experiences in distinct complete states become multiple learning experiences in the same abstracted state. One of the conclusions of Dietterich (2000a) is that MAXQ without state abstraction performs much worse than flat Q-Learning, but when state abstraction is implemented it is four times more efficient. Hierarchical reinforcement learning has proven to be much faster that flat reinforcement learning and the possibility to reuse subtask policies enhance all the learning process. Where the main disadvantage of the MaxQ approach is the need to hand code the hierarchical task structure.

### 2.5.4 Discovering Hierarchy in Reinforcement Learning with HEXQ

The biggest open problem in reinforcement learning is to discover in an automated way hierarchical structure. HEXQ is an algorithm, which automatically tries to decompose and solve a model-free factored MDP (Hengst, 2002). This is achieved by automatically discovering state and temporal abstraction, finding appropriate sub-goals in order to construct a hierarchical representation that will solve the overall MDP. Scalability is one of the biggest limitations in reinforcement learning, because sub-policies need to be relearnt in every new context. The perfect solution

would be to learn once each sub-task, and then reuse that whenever the skill was needed.

Decomposition of the state space into nested sub-MDP regions is attempted by HEXQ when the following conditions hold:

- Some of the variables contained in the state representation change less frequently than all others.
- Variables that change more frequently retain their transition properties in the context of the more persistent variables.
- The interface between regions can be controlled.

If these conditions are not true HEXQ will try to find abstractions where it can, in a worst-case scenario it has to solve the "flat" version of the problem.

#### 2.5.4.1 Automatic Hierarchical Decomposition

To construct the hierarchy HEXQ uses the state variables. The number of state variables dictates the maximum number of levels existing on the hierarchy. The hierarchy is constructed from a bottom up perspective, where the lower level, assigned as level 1, contains the variable that changes more frequently. The sub-tasks that are more often used will appear at the lower levels and need to be learnt first. In the first level of the hierarchy only primitive actions are stored. The top level will have one sub-MDP which is solved by executing several recursive calls to other sub-MDP's contained in other lower levels of the hierarchy.

#### 2.5.4.2 Variable Ordering Heuristic

The agent performs a random exploration throughout the environment for a given period of time in which statistics are gathered on the frequency that each state variable changes. Then all the variables are sorted based on their frequency of change.

#### 2.5.4.3 Discovering Repeatable Regions

Initially HEXQ tries to model the state transitions and rewards by randomly exploring the environment. The result obtained is a directed graph (DG) in which the vertices represent the state values and the edges are the transitions that occur when the agent executes each primitive action. After a period of time exploring the agent finds transitions that are unpredictable, these transitions are called ***exits***. According to the definition, an exit is a state-action pair $(s^e, a)$ where taking action a causes an *unpredictable transition.*

Transitions are ***unpredictable*** when:

- The state transition or reward function is not a stationary probability distribution
- Another variable may change value
- Or the task terminates

The result of the modeling is illustrated by the DG presented in figure 2.5, where connections represent predictable transitions between values of the first level state variable (primitive actions).

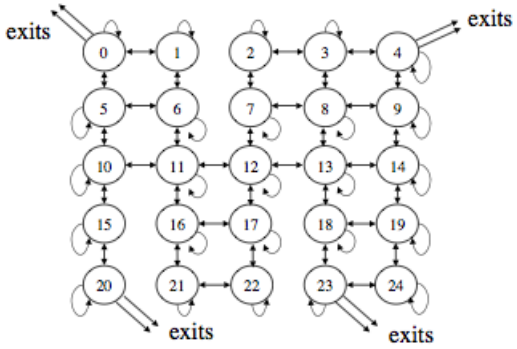



FIGURE 2.5: Directed Graph of state transitions for the taxi location (Hengst, 2002)

A procedure has to be made in order to decompose a DG into regions creating a valid hierarchical state abstraction. In first place the DG is decomposed into strongly connected components (SCCs). Combining several SCCs a region can be created where any exit state in a region can be reached from any entry with a

probability of 1. Then the total state space is partitioned by all the instances of the generic regions created. Each region has a set of states, actions, predictable transitions and reward functions. This allows the definition of a MDP over the region with a sub-goal (exit) as a transition to an absorbing state. A policy will be created as a solution to the sub-MDP that will take the agent to an exit state, starting from any entry. Multiple sub-MDPs will be constructed, more specifically, one for each unique hierarchical exit.

#### 2.5.4.4 State and Action Abstraction

To create the second level of the hierarchy a similar process from the first level has to be executed. A search is conducted in order to find repeatable regions, but now the states and actions are based on abstractions from the first level.

These abstract states are generated considering the following:

Each abstract state contains abstract actions that represent the policies to lead the agent to region exits contained in the lower level. When an abstract action is taken is state $s^e$ the corresponding sub-MDP from the level $e - 1$ is invoked and executed. The first level of the hierarchy was composed by a set of sub-MDPs, on the next level we have state and action abstraction, abstract actions take variable amounts of time to be executed, transforming the problem to a semi-Markov decision problem.

This process is repeated for each variable in the original MDP. When the last state variable is reached the top level sub-MDP is solved and represented by the final abstract states and actions, which solve the overall MDP.

#### 2.5.4.5 Hierarchical Value Function

HEXQ performs a hierarchical execution using a decomposed value function, this allows the algorithm to automatically solve the hierarchical credit assignment problem, and by relegating non-repeatable sub-tasks rewards to upper levels of the hierarchy were they become more meaningful and explanatory.

The name HEXQ derives from the equation 2.15.

$$Q^*_{em}(s^e, a) = \sum_{s'} T^a_{s^e s'}[R^a_{s^e} + V^*_{em}(s')] \tag{2.15}$$

Which translates into the following:

- Q-function at level e in sub-MDP m is the expected value of executing abstract action a in abstract state $s^e$.

To measure how HEXQ performs, several trials were performed and then compared against a "flat" learner using QLearning and also against the MAXQ algorithm. On the first trials HEXQ is the one that has a worst performance because it needs to order the variables and find exits at the first level of the hierarchy. After completing that task, the performance starts to improve and surpasses the flat learner after 41 trials, as it begins to transfer sub-task skills. Because MAXQ possesses additional background knowledge, it learns very rapidly in comparison, but in the end HEXQ converges faster.

In terms of storage requirements for the value function:

- "flat" learner requires 3000 values.
- HEXQ requires 776 values.
- MAXQ requires 632.

From these three approaches MAXQ is the one that requires less storage, but keep in mind that HEXQ, unlike MAXQ (Dietterich, 2000a), is not told which sub-tasks exist in the problem, it automatically discovers these sub-tasks. The main limitation of HEXQ is that in order to find decompositions, it depends on certain constraints in the problem, for example, the subset of the variables must form sub-MDPs and policies created can reach their exits with probability 1. If the problem to be solved is a deterministic shortest path problem, then HEXQ will find a globally optimal policy and with stochastic actions HEXQ is recursively optimal.

### 2.5.5 Automatic Discovery of Subgoals in RL using Diverse Density

Decomposing a learning problem into a set of simpler learning problems has several advantages, improving the possibilities of a learning system to learn and solve more complex problems. In RL one way of decomposing a problem is by introducing subgoals (McGovern and Barto, 2001, Mcgovern, 2002) with their own reward function, learn policies for achieving these subgoals, and then use these policies as temporally-extended actions, or options (Sutton et al., 1999) for solving the overall problem. This will accelerate the learning process and provide the ability of skill transfer to other tasks in which the same subgoals are still relevant.

The paper from McGovern and Barto (2001) presents a method for discovering useful subgoals automatically. In order to discover useful subgoals the agent searches for bottleneck regions in its observable state space. This idea arises from the study performed in the room-to-room navigation tasks where the agent should be able to recognize a useful subgoal representing doorway in the environment. If the agent recognizes that a doorway is a kind of a bottleneck by detecting that the sensation of being in the doorway always occurs in successful trajectories but not always on unsuccessful ones, then it can create an option to reach the doorway. Clearly this approach will not work for every possible problem since bottlenecks do not always make sense for particular environments.

It is very difficult to automatically find bottleneck regions in the observation space, particularly if is has to be done online. A first approach was simply look for the states that are most visited. Different state visitation histograms were created in order to observe the most visited states.

From figure 2.6 it is possible to infer that first-visit frequencies (Panel C) are better able to highlight the bottleneck states (states in the doorway) than are the every-visit frequencies (Panel B). But this alone has several limitations, one of them is that it is a noisy process, and from the results it is not obvious how to do this in problems with continuous or very large state spaces. With these limitations the authors opted in using the multiple-instance learning paradigm and the concept of diverse density to precisely define and detect bottlenecks.

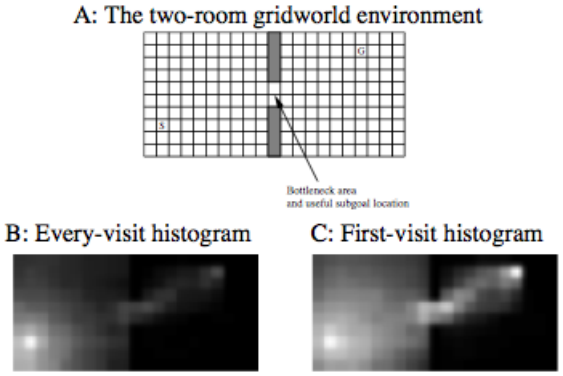



FIGURE 2.6: State visitation histograms taken from McGovern and Barto (2001)

#### 2.5.5.1 Multiple-Instance Learning

The concept of multiple-instance learning is described by Dietterich et al. (1997) as being a supervised learning problem in which each object to be classified is represented by a set of feature vectors, in which only one can be responsible for its observed classification. One specific example can be seen in Maron and Lozano-Pérez (1998), where are defined multiple positive and negative bags of instances. Each positive bag must contain at least one positive instance from the target concept, but may also contain also negative instances. In turn each negative bag must contain all negative instances. The goal is to learn the concept from the evidence presented by all the existing bags.

This can be applied for defining subgoals, where each trajectory can be seen as bag. Positive bags are the successful trajectories, negative bags are unsuccessful trajectories. The definition of a successful or unsuccessful trajectory in problem-dependent. For example in the room-to-room navigation task, a successful trajectory is any where the agent has reached a goal state. With this description a bottleneck region of observation space corresponds to a target concept in this multiple-instance learning problem: the agent experiences this region *somewhere* on every successful trajectory and not at all unsuccessful trajectories.

#### 2.5.5.2 Diverse Density (DD)

Also devised by Maron and Lozano-Pérez (1998) is the concept of diverse density to solve multiple-instance learning problems. This concept refers that the most diversely dense region in feature space, is the region with instances from the most

positive bags and the least negative bags. This region of maximum diverse density can be detected using exhaustive search or gradient descent. In McGovern and Barto (2001) exhaustive search is used since it proved feasible due to the nature of the tasks involved. So the region with maximum diverse density will be a bottleneck region which the agent passes through on several successful trajectories and not on unsuccessful ones.

Sometimes it is advantageous to exclude some states from a bag. For example, if the agent always starts or ends each trajectory in the same place, those states will have a diverse density since they occur in all the positive bags. So states surrounding the starting and ending of any bag are excluded.

#### 2.5.5.3 Forming New Options

The bottlenecks of most interest are those who appear in the initial stages of learning and persist throughout learning. A way to detect this is by using a running average of how often a state appears as a peak. if a concept is an early and persistent maximum, then its average will rise quickly and converge. If the average of a concept arises above a specified threshold then that concept is used to create new options.

As previously referred an option has several components which need to be initialized upon an option creation. The initiation set $I$ is the union of all such states over all of the existing trajectories. The termination condition $\beta$ is set to 1 when the subgoal is reached or when the agent is no longer in the initiation set, and is set to 0 otherwise. Finally the option's policy $\pi$, is initialized by creating a new value function that uses the same state space as the overall problem.

#### 2.5.5.4 Results

Trials were conducted on two simple grid-world problems. The agent has the usual four primitive actions of up, down, right and left. After identifying the subgoal an option was created, but the agent was limited to creating only one option per run.

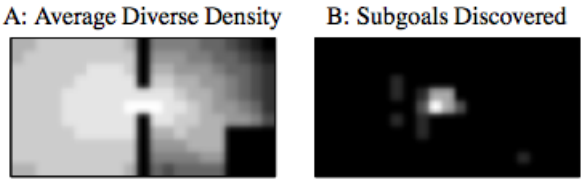



FIGURE 2.7: Average DD values and Subgoals locations (McGovern and Barto, 2001)

In figure 2.7a it is possible to observe the states with higher DD values are shaded more lightly. Also in figure 2.7b the states identified as subgoals by the agent are the ones near the door.

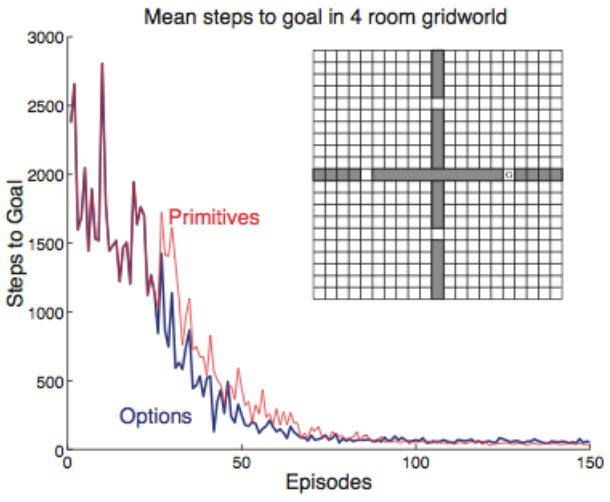



FIGURE 2.8: Average steps comparison with and without options (McGovern and Barto, 2001)

If the subgoals found by the agent are useful, the learning should be accelerated. From figure 2.8 it is clear that learning with automatic subgoal discovery has considerably accelerated learning compared with to learning with primitive actions only. The initial trials remain the same until the options are created, from that point the agent improves it's learning. In figure 2.9 the goal position has been changed and here it is also clear that the learned options continue to facilitate knowledge transfer.

In the environment presented the subgoal achievement has proven to be very useful, of course that this can not be said for every environment. Also the complexity of the environment may lead to a situation where is very difficult to define what constitutes the positive and the negative bags. Other limitation to this approach

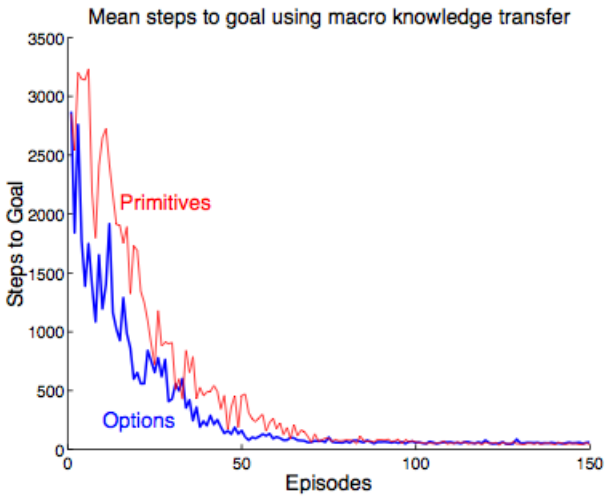



FIGURE 2.9: Performance of the learned options on task-transfer (McGovern and Barto, 2001)

is that the agent must first be able to reach the main goal using only primitive actions, limiting the problems to which it can be applied.

The next chapter gives a perspective on the software framework developed and on the different tasks approached.

# Chapter 3

# Initial Experiments

*"Throughout the centuries there were men who took first steps down new roads armed with nothing but their own vision ."*

Ayn Rand

## 3.1 Introduction

One of the main concerns upon developing the software that would be used in this dissertation, was to create a software framework to test several RL algorithms, which could supply an abstraction level in which abstract classes could provide generic functionality that can be selectively overridden or specialized by user code depending on the problem at study. In our case two different types of problems were approached using our framework, more specifically the Room-to-Room Navigation and the Taxi Problem. Since our framework has a default behavior, all we needed to do was to extend the abstract classes and override the behavior of the agent accordingly to the concerned problem.

In order to formally describe the structure of the system a simplified class diagram (Appendix A) was created with the most relevant entities in our framework.

## 3.2 Software Framework

When the application is executed, an initial frame appears, as in fig. 3.1, where the user has to define some initial parameters in order to setup the simulation environment. Such as the number of trials, episodes, the domain, which can be the Room Problem or the Taxi Problem.

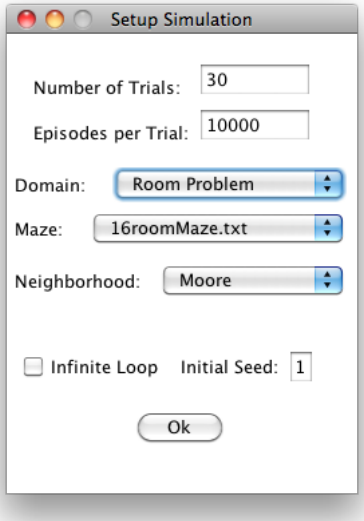
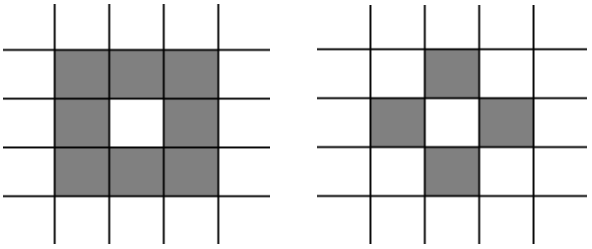



FIGURE 3.1: Initial Setup Frame

After choosing the domain, a maze can be selected from the existing ones, or a new one can be created. The set of primitive actions available to the agent are defined by the neighborhood option which can be Moore or Von Neumann neighborhood (fig. 3.2). In order to be able to recreate the experiments throughout development, the initial seed of our random number responsible for the random decisions taken by the agent, needed to be defined by the user. After all these parameters have been set, the simulation environment is ready to run.

Some events of the mouse were overridden allowing the user to manipulate several aspects of the simulation, specifically:

- **right-click** - By right-clicking the user can create the maze, by adding or removing grey rectangles.
- **left-click** - Changes the position of the main goal, and prompts its reward.
- **drag-n-drop** - Changes the initial position of the agent.



(A) Moore (B) Von Neumann

FIGURE 3.2: Surrounding cells comprising the neighborhood

- **scroll** - In the Taxi problem allows the user to create thin walls around the rectangle.

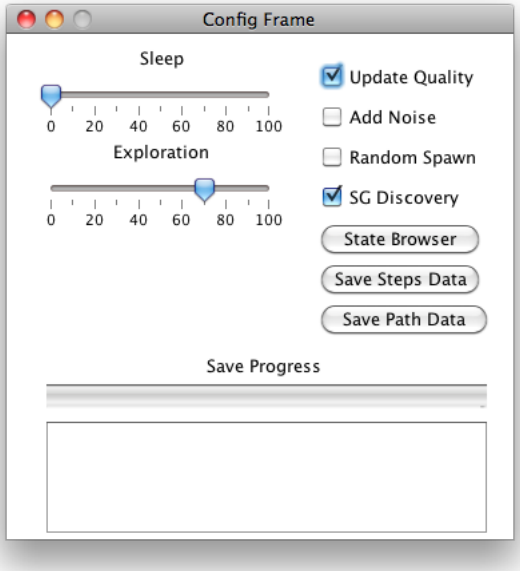



FIGURE 3.3: Configuration Frame

Another important frame is the Configuration Frame (fig. 3.3). Here the user can control the exploration rate of the agent, the running speed of the simulation, whether it should update the quality of the actions, add noise to the actions choosing, spawn the agent in a random location of the grid-world and enable the automated subgoal discovery.

When all the trials have terminated, the user has the possibility to save relevant data from the simulation. By pressing the `Save Steps Data` button, the user will create two excel files, one containing the average steps count, and other containing the average quality of the actions for all the trials with corresponding charts. The

`Save Path Data` button will save all the paths taken by the agent and sort them by classes in an excel file.

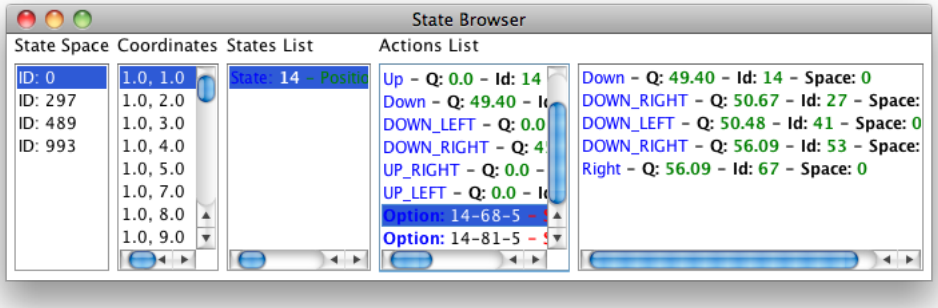



FIGURE 3.4: State Browser

For debugging purposes a state browser was implemented (fig. 3.4), which allow us to inspect all the states and the corresponding actions. From the two-dimensional coordinate of the grid-world, one or more states can exist, like in the Taxi problem. Each state has its own unique id and a set of actions which can be primitive or composite actions (Options). Finally a global representation of the menus of the application is shown in fig. 3.5 where the name of the menu are fairly self-explanatory.

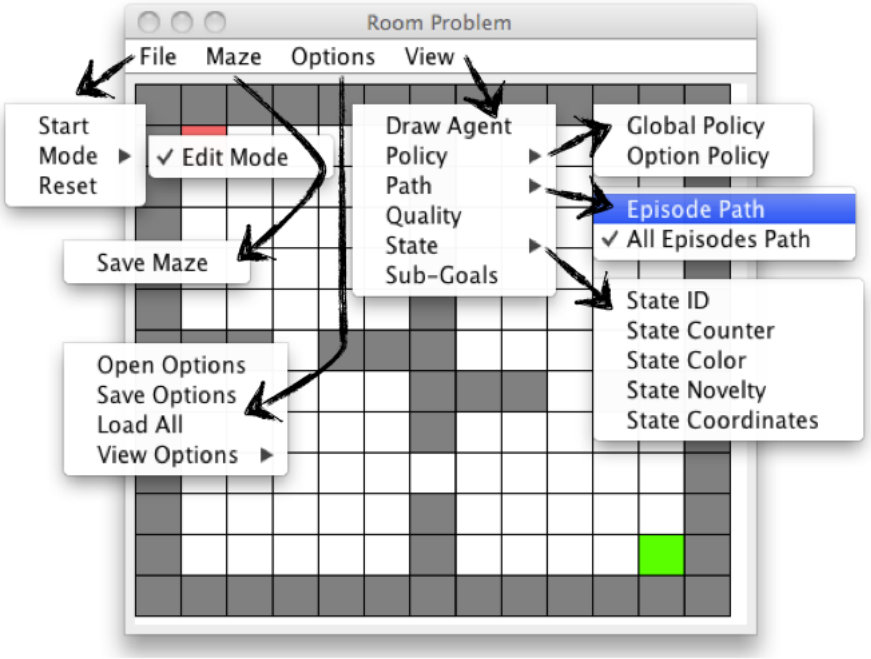
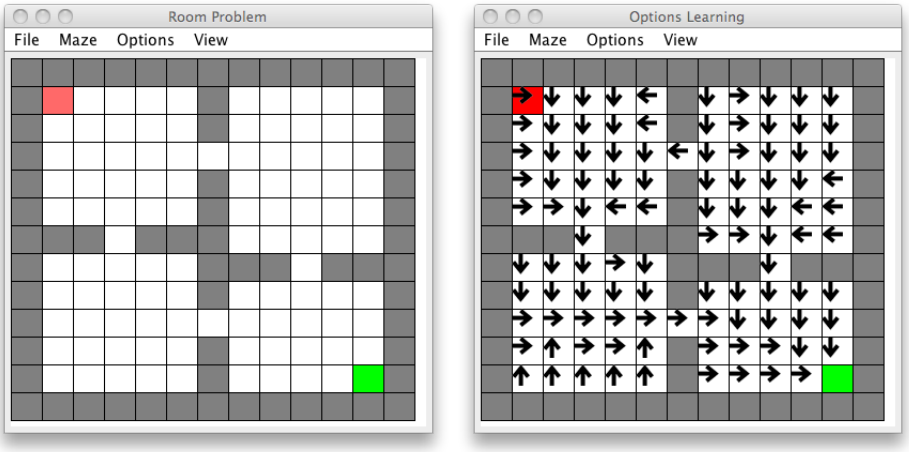



FIGURE 3.5: Application Menus

## 3.3 Room-to-Room Navigation

In order to create a solid starting point for our approach, we proposed on implementing the work done by Sutton et al. (1999), by using the same task and trying to replicate some of the results obtained.




(A) Agent and the main-goal (B) Flat Q-learning policy

FIGURE 3.6: The 4-room grid-world environment

Figure 3.6a is our representation of the four-room grid-world studied by Precup and Sutton (2000) where, the red rectangle represents the agent position, and the green rectangle represents the main goal of the agent. Our first task was to make the agent learn how to reach the main goal by using only Q-learning (Watkins and Dayan, 1992), then implement the Options framework from Sutton et al. (1999) and compare the results. When using only Q-learning a policy is obtained comprised by four primitive actions of up, down, right and left, like in fig. 3.6b.

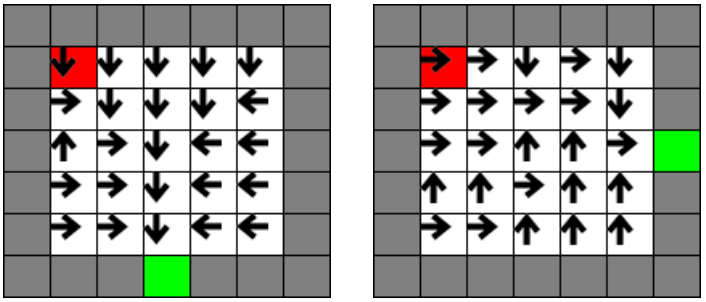


FIGURE 3.7: Two policies underlying two of the eight hallway options

In the Options framework (Sutton et al., 1999) the author refers that each of the four rooms has two built-in *hallway options* designed to take the agent from

anywhere within the room to one of the two existing doors in that room. The process of creating those options is not explained thoroughly, only that it can be handcrafted by the programmer, or in a more interesting way created by the agent. Our solution was to make the agent explore the environment each room at a time, creating a policy to reach the door of the respective room. Each time a policy is learned the programmer has the possibility of saving it to a file, so that it can be used later on different trials.

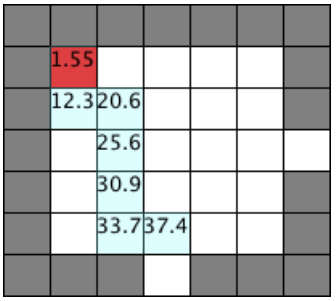



FIGURE 3.8: Visual representation of an Option

An option's policy $\pi$ will follow the shortest path within the room to its target door passage, while minimizing the chance of going into other doorway. A possible example of this policy is shown in fig. 3.7. On the other hand an example of a shortest path that will take the agent to a doorway is shown in fig. 3.8 represented by the blue rectangles. The termination condition $\beta(s)$ for each doorway option is zero for states $s$ within the room and 1 for states outside the room. The initiation set $I$ comprises the states within the room. The values inside of each rectangle represent the quality of the best action on that state. In our case an option represents an abstract action with a global quality, that global quality is obtained by calculating the average quality of all the actions contained in the option. After some exploration the agent will realize whether it is more advantageous to take options instead of primitive actions, depending on the quality values.

Since our options represent temporarily extended courses of action composed by primitive actions, each time the agent takes an option, it will recursively update all the primitive actions contained in the option, and when terminates it will update the quality of the option itself, as explained in algorithm 3.

**Algorithm 3** Q-learning with Options

Initialize $Q(s, a)$
**for** (each episode) **do**
  Initialize $s$
  **for** (each step of the episode) **do**
    Choose $a$ from $s$ using policy derived from $Q$
    **if** (a is Primitive) **then**
      Take action $a$, observe $r, s'$
      $Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma \max_{a'} \; Q(s',a') - Q(s,a)\Big]$
      $s \leftarrow s'$
    **else**
      Take option $o$
      **for** (all actions $a \in o$) **do**
        Take action $a$, observe $r, s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma \max_{a'} \; Q(s',a') - Q(s,a)\Big]$
        $s \leftarrow s'$
      **end for**
      $Q(s,o) \leftarrow \frac{1}{|o|} \sum_{i=1}^{|o|} Q(a_i, o)$
    **end if**
  **end for**
  until $s$ is terminal
**end for**

Remember that the options created are deterministic and Markov, and that an option's policy is never defined outside of its initiation set $I$. We denote the set of eight doorway options by $H$. After making the options available for the agent to take, the next step is to run a new trial with 10000 episodes with an exploration rate of 100% and with this trial a policy is obtained where the agent realizes that it is more advantageous to take options instead of primitive actions.

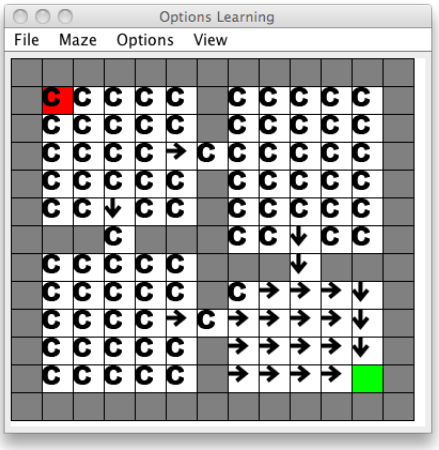



FIGURE 3.9: Policy with Options

From fig. 3.9 it is possible to observe the policy learned by the agent, the primitive action are represented with arrows and the option is represented by the *C* character, which is an abbreviation for *Composite Action*. Now it is time to compare results between the flat Q-learning and the Options approach. We expect that with options the learning process will be accelerated. For our experiments we had 30 trials, each with 10000 episodes, an exploration rate of 10% and without noise. In the chart in fig. 3.10 we can confirm that there is a substantial improvement in the learning process, especially in the early stages. This is due the fact, that our agent, rather than planning step-by-step, used the options to plan at a more abstract level (room-by-room instead of cell-by-cell) leading to a much faster planning. It is also interesting to see that it soon learns when not to use options (the last room).

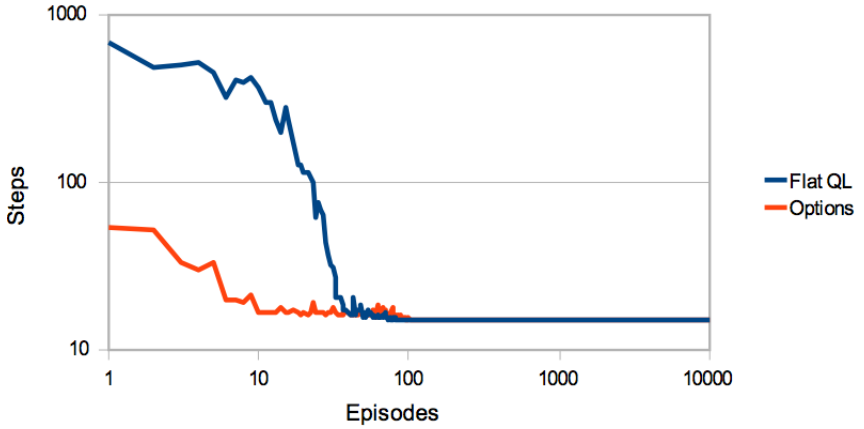



FIGURE 3.10: Comparison of the evolution of average number of steps per episode between Q-Learning and Options

It is also important to compare the average quality of the actions taken by the agent. So the same experiments of 30 trials, each with 10000 episodes were conducted using only flat Q-learning, and then repeated with options added. Results in fig. 3.11, show that using options translates in a clear overall gain of quality from the early stages up until the end of the trials.

Options can be used by the agent to facilitate learning on similar tasks. To illustrate how options can be useful for task transfer (Lin, 1993, Singh, 1991), the grid-world task was changed by moving the goal to the center of the upper right-hand room. We reused the options created for the agent to reach the doorways. Again, we had 30 separate trials, with the same conditions of the previous experiments. We compare the performance of the agent reusing the options to an agent learning with primitive actions only (fig. 3.12).

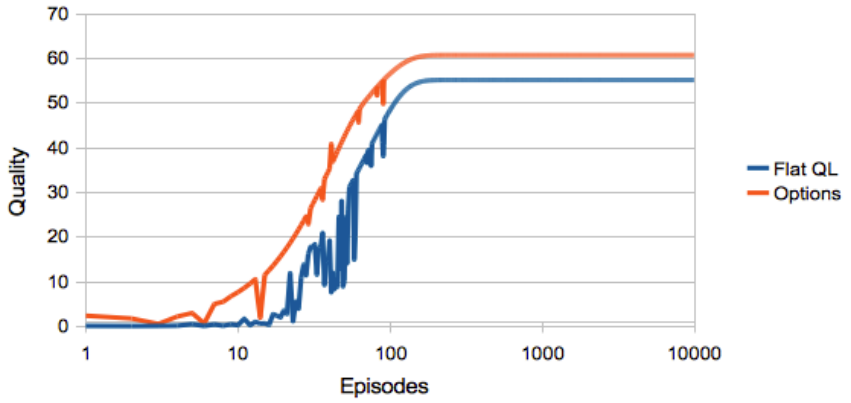



FIGURE 3.11: Comparison of the evolution of average quality of the value function per episode between Q-Learning and Options

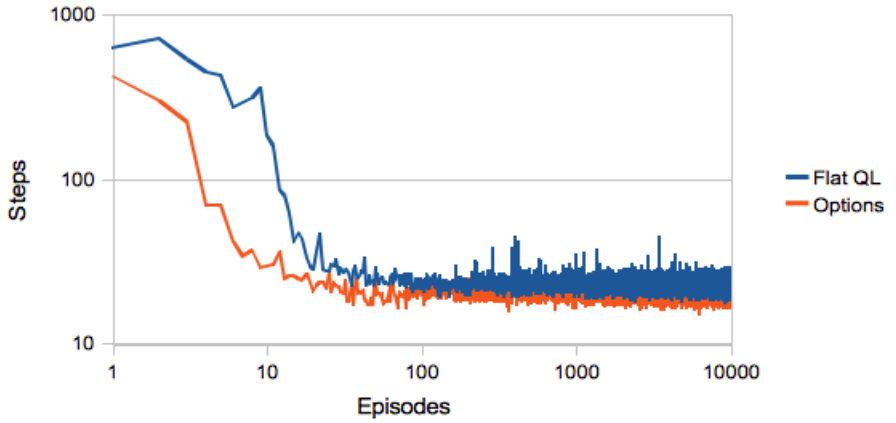
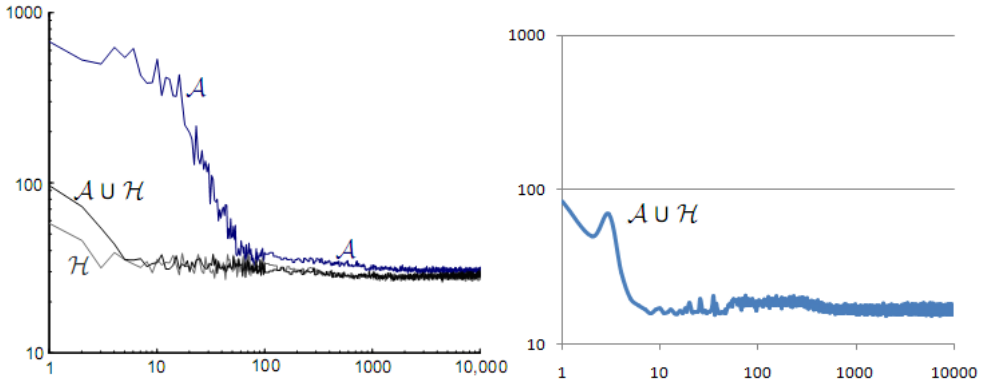



FIGURE 3.12: Comparison of the evolution of average number of steps per episode in knowledge transfer

By reusing the options, the agent was able to reach the goal more quickly even in the beginning of learning and it maintained this advantage throughout. With this experiment we demonstrated the usefulness of options in similar tasks, that considerably accelerated learning on the new task.

With our implementation of the Options we have already shown and proven the benefits of its use, but it is also important to compare our results with the results obtained by the original authors in Sutton et al. (1999). In order to replicate the results the conditions of the simulation had to be the same, so noise was added to our simulation. Noise is inserted as a probability of an action being switched by another, random, action. With probability 2/3, the actions may cause the agent to move one cell in the corresponding direction, and with probability 1/3, the agent moves in one of the other three directions. If the movement would take the agent into a wall then the agent remains in the same cell. We consider the case in which rewards are zero on all state transitions. The probability of occurring

random actions, $\epsilon$, is 0.1 and the step size parameter used was $\alpha = \frac{1}{8}$. These are the parameter reported to be used in Sutton et al. (1999).




(A) Simulation from Sutton et al. (1999)

(B) Our Simulation using primitive actions and Options

FIGURE 3.13: Comparison between the simulation done by Sutton et al. (1999) and our implemented simulation

Figure 3.13 shows two different charts, one represents the results obtained by Sutton (fig. 3.13a), and the chart from fig. 3.13b represents our simulation. The relevant distribution here is $A \cup H$, which stands for a policy that contains primitive actions and options. Apart from a small peak that occurs in our simulation, the results are very similar. This proves that our implementation of the Options Framework was successful and faithfully recreated the one presented in Sutton et al. (1999).

## 3.4 Taxi Problem

In order to gain a more general perspective in the use of Options, we decided to implement an also well known problem from Dietterich (2000b), which is the Taxi Problem. But instead of using MaxQ like the authors, we will be using the Options approach measuring it's performance in a more complex situation.

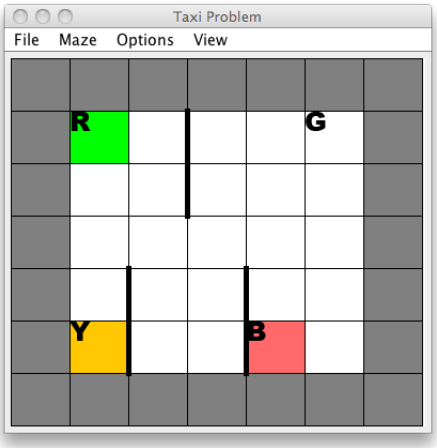



FIGURE 3.14: The taxi grid-world environment

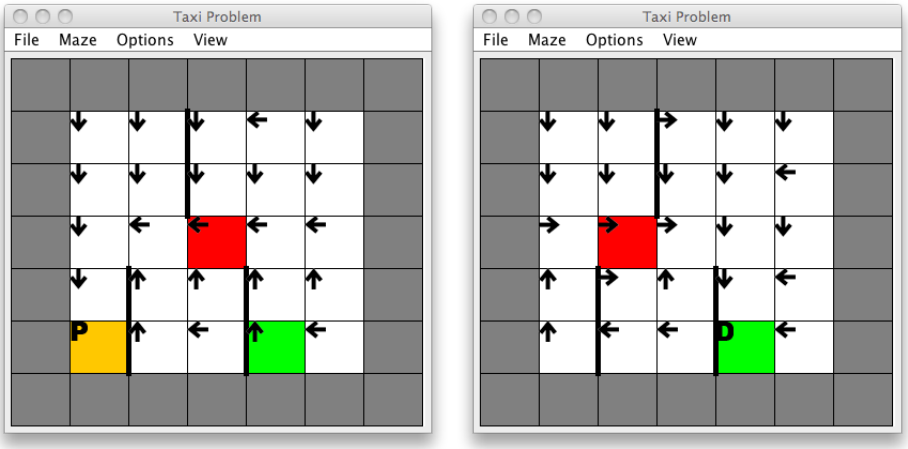
Figure 3.14 shows a 5-by-5 grid where the taxi agent is represented by the red rectangle. In this world exists four specially-designated locations, marked as **R**, **B**, **G**, and **Y**. In each episode, the taxi starts in a randomly-chosen square. There is a passenger at one of the four locations (chosen randomly), and that passenger wishes to be transported to one of the four locations (also chosen randomly). The taxi must go to the passenger's location represented by the yellow rectangle, pick up the passenger, go to the destination location represented by the green rectangle, and put down the passenger there. If the passenger is dropped at the destined location the episode ends.

In this domain the agent has six primitive actions:

- four navigation actions: Up, Down, Left, or Right
- a Pickup action
- a Putdown action

There is a reward of -1 for each action and an additional reward of +20 for successfully delivering the passenger. There is a reward of -10 if the taxi attempts

to execute the Putdown or Pickup actions illegally. If a navigation action would cause the agent to hit a wall, the agent remains in the same state, and there is only the usual reward of -1. In terms of state space there are 500 possible states: 25 squares, 5 locations for the passenger (counting the four starting locations and the taxi), and 4 destinations, being almost 5 times bigger that the state space of the room-navigation domain.




(A) Policy for Pick (B) Policy for Drop

FIGURE 3.15: Q-learning Policies

Figure 3.15a shows a policy composed only by primitive actions for a situation where the taxi has to pick up the passenger, represented by the yellow rectangle, the best action in that square is identified by a **P** which stands for Pick Up. On the other hand, after the agent has picked up the passenger, it has to navigate to the passenger's intended destination and drop him. That purpose is represented by the policy in fig. 3.15b, where the passenger destinations is at the green rectangle, and the best action to perform on that state is to drop the passenger (**D**).

The next step was to create the options for the taxi task. First we needed to identify which states could represent subgoals for the agent. In the room-navigation problem the subgoals were easily identified as the doorways that lead the agent into another room. In the Taxi problem there are no rooms, so subsequently there are no doorways, instead we have the four specially-designated locations ( **R**, **B**, **G**, and **Y**) that may represent a subgoal for the agent. With four separate runs we made the agent learn four policies on how to reach those four locations. An example for those policies is shown in fig.3.16 for location **B** and **G** respectively.

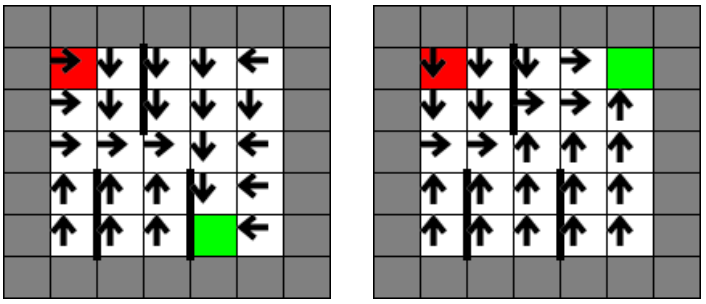


FIGURE 3.16: Two policies underlying two of the four source options

With the policies created the corresponding options were added to the share of actions of the agent. At this point in any state the taxi agent could choose between 10 different actions (6 primitive and 4 options). The algorithm used on this problem is the same that was used on the room-navigation problem(algorithm 3). Again when the agent has the possibility to choose actions more abstract than the primitive actions, it will make it plan at a higher level, where instead of moving one cell at a time, it can choose at any state to go directly to one of the four special locations that allow him to pick or drop the agent, this way reducing the amount of trial an error and increasing the chances of making a successful pick up or drop. This is clearly visible in fig. 3.17.

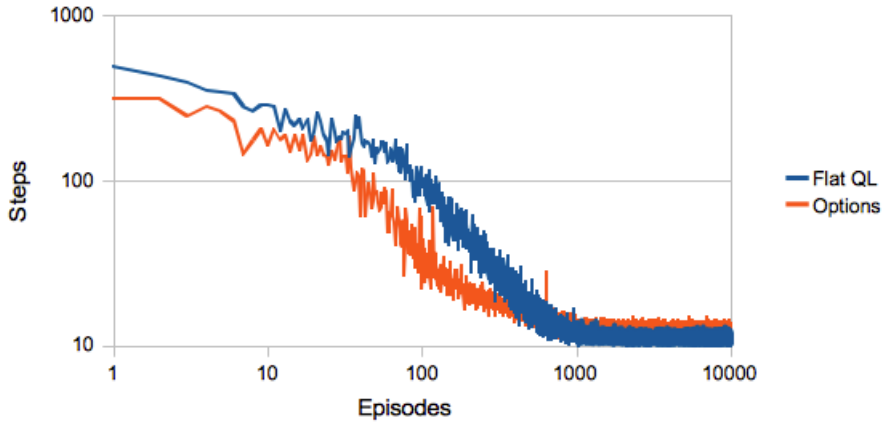



FIGURE 3.17: Comparison of the evolution of average number of steps per episode between Q-Learning and Options

The usefulness of Options has been proven, but in order to use them still exists a troublesome process that has to be made each time a new task is presented. Firstly the need to identify correctly the states that can be useful as subgoals to the agent, which sometimes could be difficult. Secondly this process needs to be done *offline*, when really, the most interesting would be for the agent to detect the subgoals while learning to solve the task at hand, and create corresponding options on the fly.

The next chapter introduces our method developed for identifying useful subgoals from the agent's initial interaction with its environment and the results obtained.

# Chapter 4

# Autonomous Subgoal Discovery

*"Imagination is more important than knowledge."*

Albert Einstein

## 4.1 Introduction

After implementing the Options framework (Sutton et al., 1999) in the rooms example and successfully replicating the results, the next step was to research methods to identify sub-goals. A sub-goal can be seen as a state that an agent must pass-through when making a transition to a novel or different region of the state-space. These states can be seen as sub-goals that lead the agent to the main goal.

Throughout this research, several articles were found, namely, Parr and Russell (1998), Dietterich (2000b), McGovern and Barto (2001), Hengst (2002) and Simsek and Barto (2004), Simsek et al. (2005). We have implemented a selection of the approaches found in these papers. The implemented approaches are discussed in the following sections.

## 4.2 State Counting

Initially we tried a very simple approach and started to count how many times the agent has been in a certain state, like we saw in Simsek and Barto (2004). Basically each time the agent reaches a state, we increment a state counter variable. After performing several trials of 1000 episodes, we obtained the following typical result as seen in (Figure 4.1).

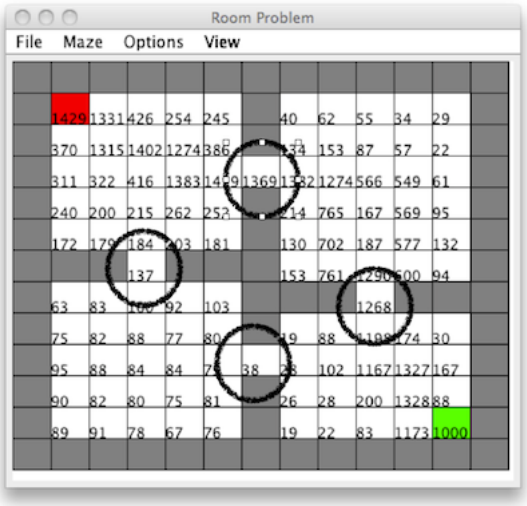



FIGURE 4.1: Number of times each state was visited

We expected that this experiment would show some differences in the state counts between the target and non-target states, but there were no visible differences. The states which we consider as sub-goals are designated with a black circumference around them, with the counting values of 137, 38, 1369 and 1268. There seems to be no relation between these states considering the values. We could not find any characteristic that distinguishes these states from all others. Sometimes there are also other states with the same counting value, that don't represent sub-goals.

So, a different view was implemented like the one depicted in (Figure 4.2), where it is possible to get a flood-like representation of the state counter, easier to read. The states where the blue color is darker represent the states in which the agent has been passing more frequently. In the end of the trial it is possible to distinguish the path chosen by the agent to reach the main goal.

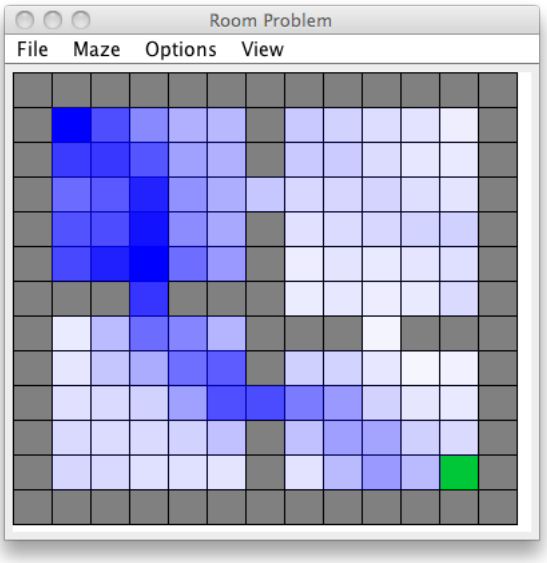



FIGURE 4.2: Colored states representing the number of times each state was visited

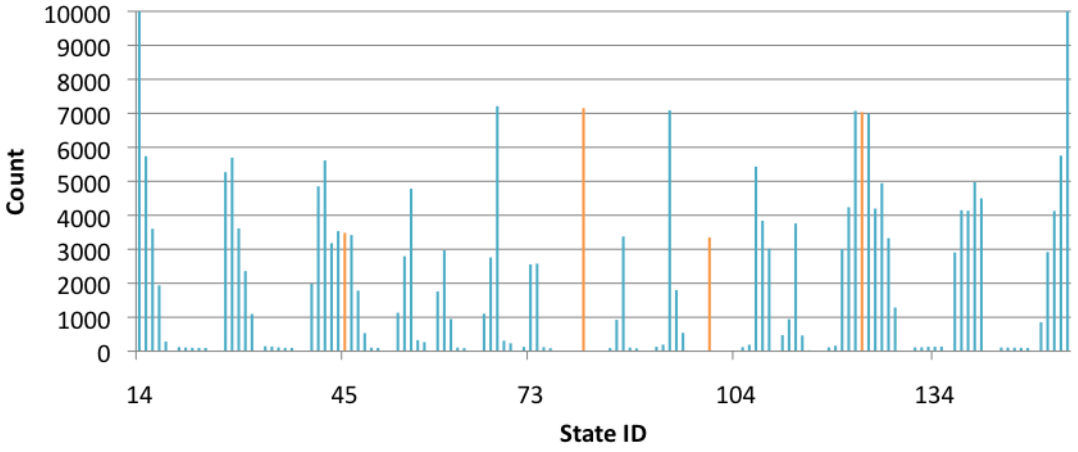



FIGURE 4.3: Average state counting in the four-room navigation task

Since in the isolated trials no differences were detected the next attempt was to consider statistics over multiple trials. We decided to make a 25 trials run, each with 10000 episodes. Results can be seen in the histogram (Figure 4.3) with the average counter values for each state.

The states considered as sub-goals were identified with the orange color. From this distribution we concluded that the state counters did not provide enough information to distinguish the target states and that another approach was required.

## 4.3 Using Relative Novelty to Identify Useful Temporal Abstractions in RL

After investigating the work done by Simsek and Barto (2004), the idea of Relative Novelty seemed promising and an implementation of their work was attempted, both to gain knowledge of the inner structure and to try to replicate some of the results obtained.

The main idea is to find differences between target, and non-target, states. To achieve that purpose the concept of relative novelty was introduced. Relative Novelty measures how much novelty a state introduces in a short-term perspective. To define novelty, we measured how frequently a state is visited by the agent since it has started to perform actions, and that part was already done as we described it previously. Then every time the agent visits a state, a counter is incremented and with a simple calculation of $\frac{1}{\sqrt{n_s}}$ the novelty is done.

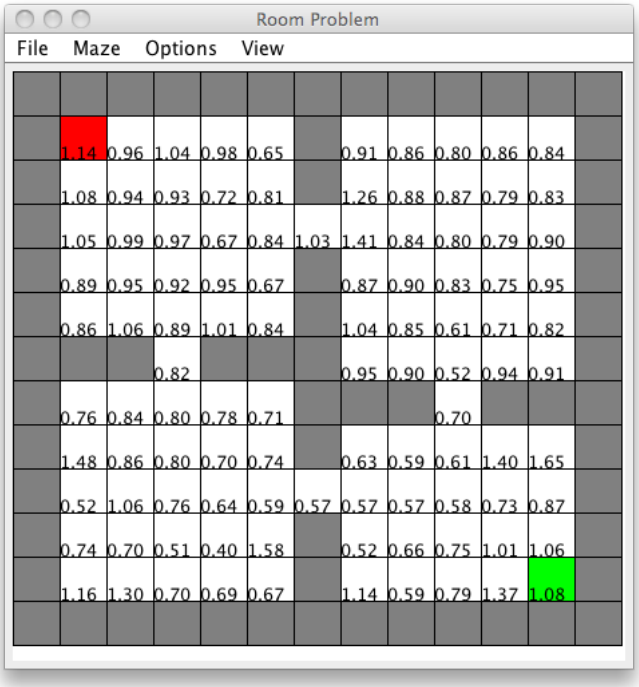



FIGURE 4.4: Relative Novelty value of each state represented at each cell

Novelty will allow us to calculate the relative novelty of a state (Figure 4.4). Relative novelty is the ratio of the novelty of states that followed the current state, to the states that preceded it. To calculate the value of the relative novelty we used a data structure to store the last 15 states the agent passed through.The relative

novelty of the state in the 7th position of the data structure can be calculated using the 7 preceding states and 7 states that followed it. This number of forward and backward transitions is a parameter of the algorithm described in Simsek and Barto (2004), section 2.2, called the **novelty lag**.

In this experiment the goal state is ignored and the agent is ordered to perform a 1000-step random walk, 1000 times, where at each time the agent is spawned at a random location. The objective is to identify differences between the distribution of the relative novelty in target and non-target states, expecting non-target states to have a higher score more frequently.

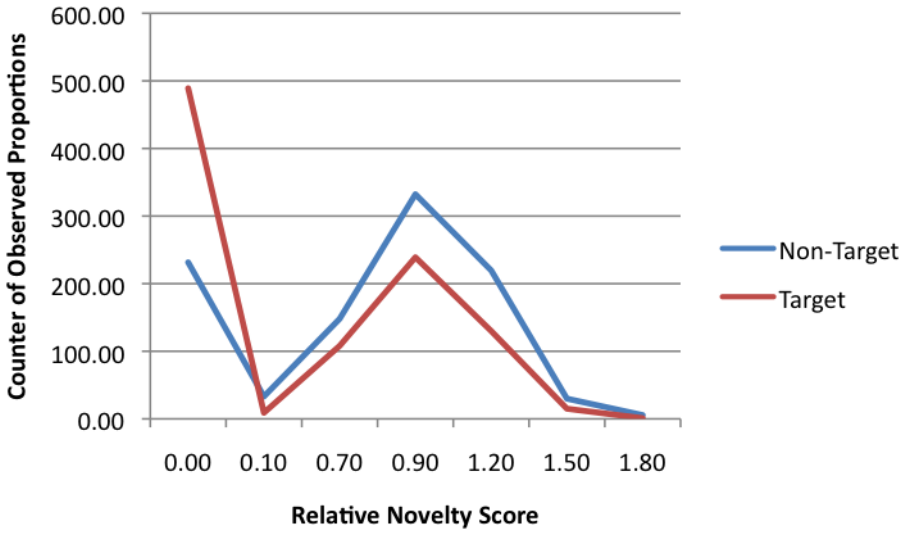



FIGURE 4.5: Relative Novelty Distribution between target and non-target states

Figure 4.5 explicitly depicts the difference of the relative novelty distribution between target and non-target states. We managed to obtain approximate distributions similar to those obtained by the authors, but this alone is only a part of the work done by them. Although this provides evidence that target and non-target states are distinguishable in that specific problem, the main question is whether that knowledge can be used to identify sub-goals in a different problem, because the parameters generated by the test are yet to prove applicable to other problems.That's why we chose to continue with another approach, one where the agent has no input from the programmer affecting the outcome of the algorithm.

## 4.4 Path Intersection

This was our final approach to the problem of automated sub-goal discovery. With the work done so far we gathered information that allowed us to define what is a sub-goal. We will look for a specific type of **sub-goal**, a state where the agent must pass in order to reach it's main goal.

In our approach we created the **path** definition, where a path $p$ is *the course taken by the agent to reach it's goal* and it is comprised by two components: a sequence of states $s_t, s_{t+1}, ..., s_{t+n}$ of length $n$, and by a class identification $id = 0, 1, 2, ...$ which defines to which class the path belongs. Somewhere in that path there is one or more sub-goals, our algorithm will be responsible for identifying them. A path is represented in Figure 4.6 by all the states with a cyan color, each path corresponds to a single episode. This problem has the particularity of being an episodic task, but for continuing tasks it would be necessary to segment the experience into finite-length trajectories in order to enable the creation of paths.

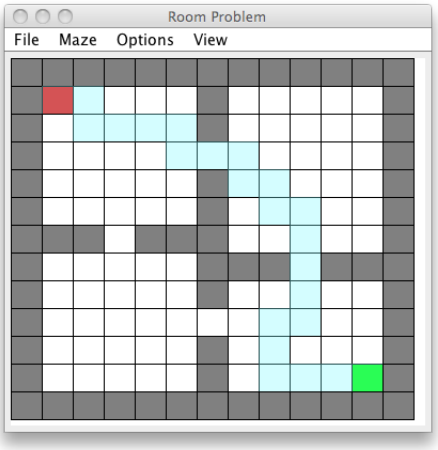



FIGURE 4.6: Example of a path

The agent has a set $P = \{p_e, p_{e+1}, ..., p_{e+n}\}$ containing all the different paths taken until the current episode. Each time an episode ends, a new path is created, a class identification (id) assigned to it, and then added to $P$. The class id attribute from a path is very important to our algorithm which relies on intersecting paths of the same class. In a perfect solution the number of existing classes on an environment would correspond to how many different ways the agent has to reach the main goal. In the four-room grid-world environment the agent has two obvious

ways of reaching the goal, meaning that at least, it will have two different classes (fig. 4.7).

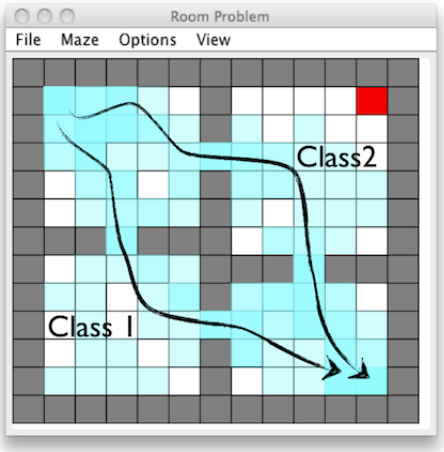



FIGURE 4.7: Existing path classes in the four-room navigation task

In algorithm 4 is described how the class id is assigned to a recently created path. Basically we define that a path belongs to a class if the number of states in common with an existing class is greater or equal than half the size of the path in question. It was important to create classes in order for us to intersect only paths that belong to the same class, if we intersected paths from different classes we might obtain an empty set and lose relevant states that could be potential sub-goals.

**Algorithm 4** Assign Class ID(currentPath)

```
classId ⇐ −1
commonStates ⇐ 0
for each path in Path-List do
  if currentPath ∩ path > commonStates then
    commonStates ⇐ currentPath ∩ path
  end if
  if commonStates > half the size of currentPath then
    classId ⇐ path class Id
  end if
end for
if classId = −1 then
  return new class Id
else
  return classId
end if
```

We are trying to find common regions amongst the paths, that contain states that could be identified as sub-goals. But not all of those regions have states that represent positive sub-goals (fig. 4.8). For example in our case the agent always starts or ends each path in the same set of states, because of that, those states will have a high probability of being identified as sub-goals reducing the relevance of the states representing actual sub-goals.

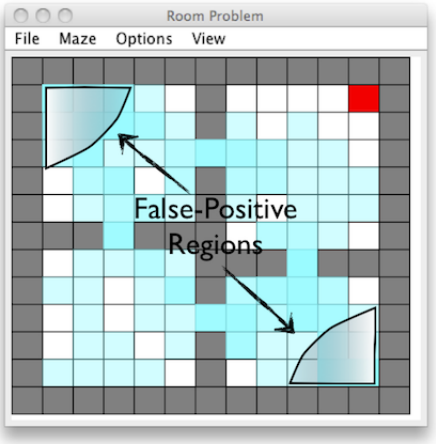



FIGURE 4.8: Excluded regions from the paths

We need to exclude those states surrounding the starting and ending of the paths. The procedure to exclude the initial states is described in algorithm 5. For removing the final states, the procedure is very similar to the one in alg. 5, the only difference is that the states are removed from the end of the path. After excluding those states everything is ready to start the intersections.

**Algorithm 5** Exclude Initial States(path1, path2)

```
state1 ⇐ path1 first Element
state2 ⇐ path2 first Element
while state1 equals state2 do
  path1 remove first Element
  state1 ⇐ path1 first Element
  path2 remove first Element
  state2 ⇐ path2 first Element
end while
```

The idea is to create intersections between paths of the same class. From the example in Figure 4.9 the concept becomes clearer, where a set of paths in blue color are covering different states leading to the same final state. The result of the intersection created is a set of states common to all the paths of the same class,

marked in an black circumference, meaning that independently of the path taken by the agent, it always has to go through some states that could potentially be used as sub-goals.

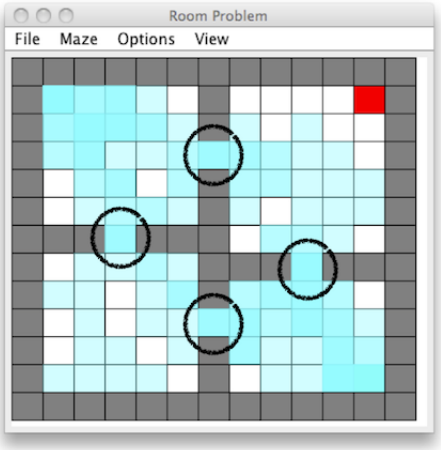



FIGURE 4.9: Expected resulting states from paths intersection

The summarized algorithm 6 describes the process for path intersection.

**Algorithm 6** Intersect Paths (classID)

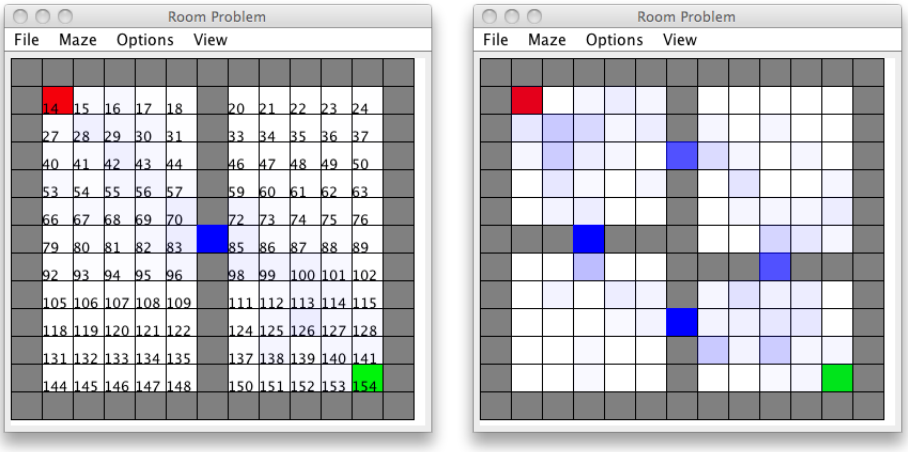
```
PathsById ⇐ get all the Paths with the same classID
while PathsById List Size > 1 do
    LastRemoved Path ⇐ PathsById remove last Element
    Last Path ⇐ PathsById get last Element
    Exclude initial/final states from Paths
    ResultingPath ⇐ LastRemoved Path ∩ Last Path
    Add ResultingPath to PathsById
end while
return PathsById
```

The sooner the agent discovers useful subgoals, the sooner it will create the corresponding options. We tried to achieve this behavior by trying to discover subgoals right after the second episode (it is necessary to have more than one path). Instead of trying to discover subgoals at each episode, every time the agent takes a shorter path than the previous shortest path to the goal, it will execute the process of discovering subgoals. This way, the agent reduces the amount of processing needed, which in long term could influence its performance. The process for subgoal discovery is described in the summarized algorithm 7.

**Algorithm 7** Sub-Goal Discovery(path)

```
Init Path-List to 0
pathSize ⇐ 1000
for each episode do
  Interact with environment (Learning)
  Set class ID to observed path
  Add observed path to Path-List
  if observed path length < pathSize then
    pathSize ⇐ observed path length
    Intersect Paths of the same class ID
    Identify resulting states as sub-goals
    Create Options
  end if
end for
```

We illustrate the ability of our intersection algorithm to highlight the doorway regions using the online experience of an RL agent learning in several grid-world environments (fig. 4.10). The doorways identified as subgoals appear represented in strong blue. The stronger the blue color, higher the number of times that the agent identified that state or region as a useful subgoal.




(A) Two-room navigation task (B) Four-room navigation task

FIGURE 4.10: Subgoal discovering histogram

The data used to discover the subgoals was collected from the online behavior of an agent using Q-learning in 30 trials of 10000 episodes, initially in a 11x11 grid-world with two, four and six rooms respectively. Then in a 20x20 grid-world with sixteen rooms. The goal state was always placed in the lower right-hand corner, and each episode started from a fixed state in the upper left-hand room. An episode ended when the agent reached the goal receiving a reward of 100 for

reaching the goal and 0 otherwise. The primitive actions available to the agent were up, down, right, left, up-left, up-right, down-right and down-left. The discount factor, $\gamma$, was 0.9. The learning rate was $\alpha = \frac{1}{8}$. The exploration used the $\epsilon$-greedy method where the greedy action was selected with probability $(1-\epsilon)$ and a random action was selected instead. We used 0.7 as the value for $\epsilon$.

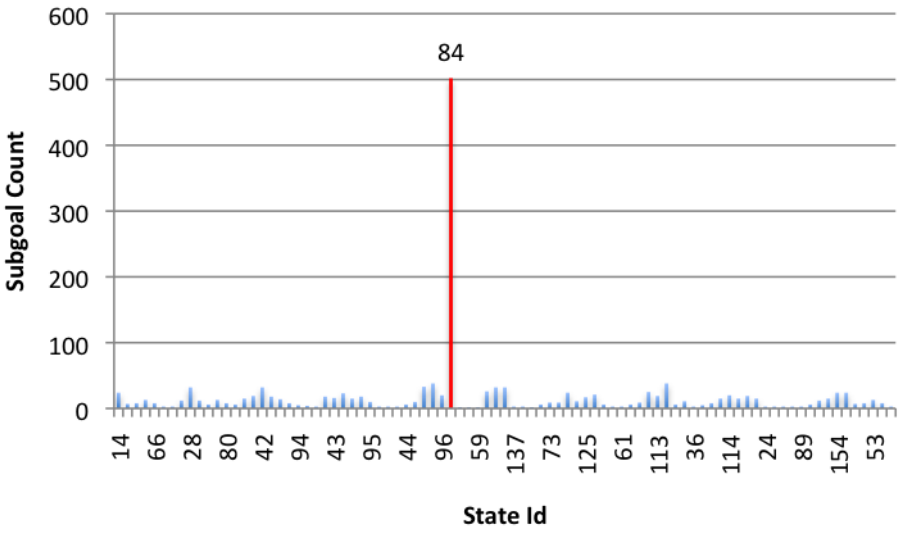



FIGURE 4.11: Target, and non-target, states (two-room)

In figure 4.11 it is possible to observe that during the trials, one state, namely state 84, representing the doorway between the two-rooms of the environment, really stands out from the rest of the states being considered a useful subgoal.

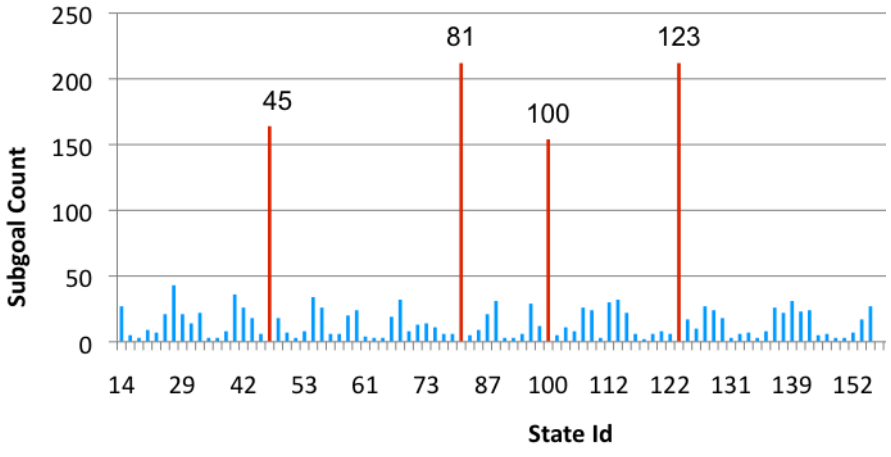
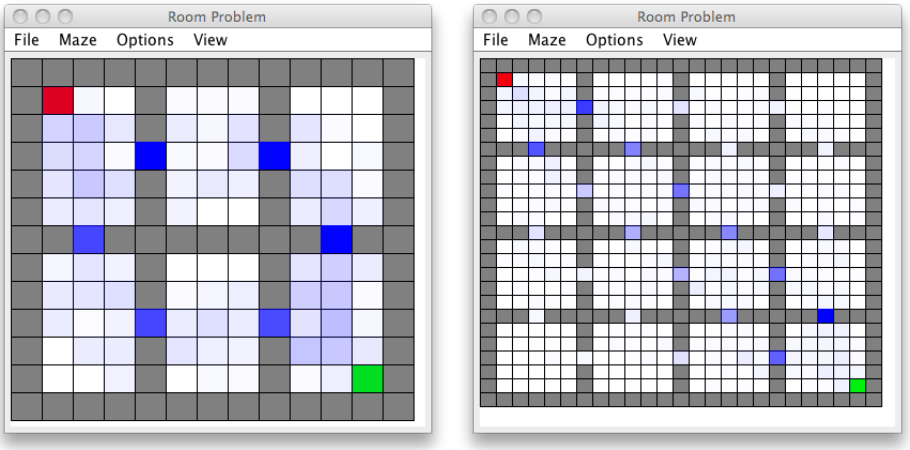



FIGURE 4.12: Target, and non-target, states (four-room)

In the four-room environment we expected the agent to discover 4 subgoals representing the doorways from one room to another, again, in fig. 4.10b, 4 different states are easily distinguishable from the other states. The counting difference

between the states identified as subgoals is related to the path class that they belong. For example state 81 and state 123 belong to the same class (similar counting values) and have higher counting values that state 45 and state 100, meaning that the agent took more times the path in which state 81 and 100 occurred.




(A) Six-room navigation task (B) Sixteen-room navigation task

FIGURE 4.13: Subgoal discovering histogram

From figure 4.13a we notice that for smaller environments all the subgoals are discovered successfully. Bear in mind that these histograms represent several runs for the same environment, with only one run the agent might only discover subgoals relevant to one path (or one class) leading to the main goal.

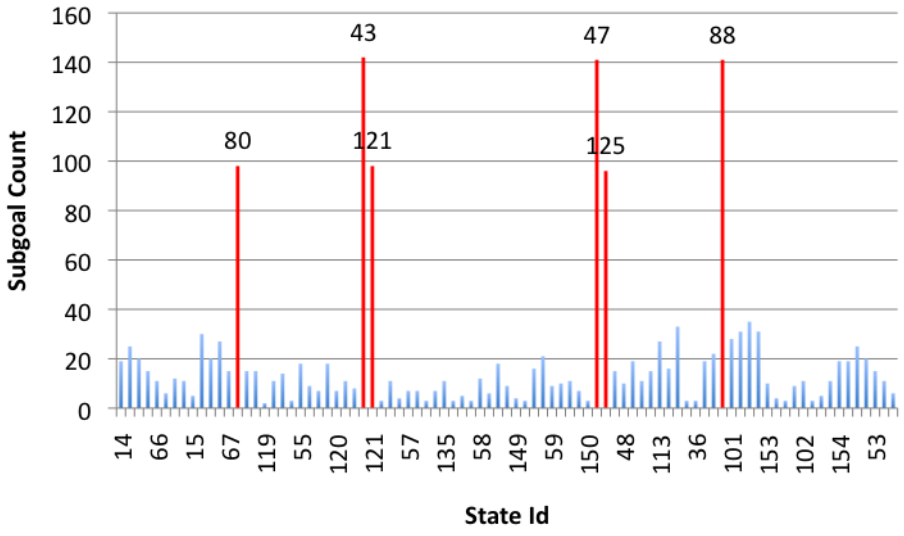



FIGURE 4.14: Target, and non-target, states (six-room)

But for a larger and more complex environment like the one in fig. 4.13b, if we want to discover all the subgoals the exploration rate needs to be increased. The exploration rate will affect the number of subgoals discovered by the agent. With a high exploration, a more diverse set of paths will be followed by the agent, resulting in more different path classes, subsequently more subgoals. But the agent may not be required to discover all the subgoals of the environment, in fact that might even be counterproductive by deviating the agent from the real goal.

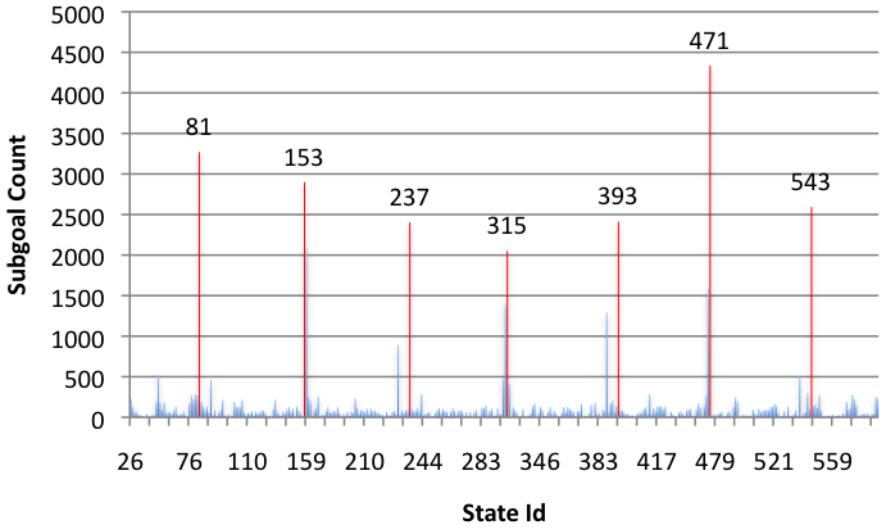



FIGURE 4.15: Target, and non-target, states (sixteen-room)

As shown in figure 4.15, even in a more complex environment the agent successfully discovers useful subgoals in the doorways, confirming the results that we anticipated. The sixteen-room grid-world has 24 doorways, our algorithm only considered 12 as relevant (fig. 4.13b). This is important because the agent only chose the doorways that will lead to the best way to reach the goal. For example, if the agent has to go through the upper right-hand rooms, that will represent a longer path for reaching the goal. The next chapter introduces our method for creating options online, corresponding to the useful subgoals discovered by the agent from it's interaction with its environment and presents results in the room-navigation domain.

# Chapter 5

# Option Creation

*"The main thing is to keep the main thing the main thing."*

Stephen Covey

## 5.1 Create Options Dynamically

Our algorithm of path intersection allows the agent to discover subgoals, but that alone will not improve the agent's performance. The usefulness of subgoals is that they allow the agent to create options to reach the subgoals, thus accelerating the learning process. Our algorithm for creating options dynamically is summarized in alg. 8 and it will be described in this section.

**Algorithm 8** Create Options( )

```
for each subgoal discovered do
    Set order of Appearence
    Create Option o = (I, π, β)
    Init Option's Initiation Set (Initial State, Final State)
    Init Option's value function
end for
for each Option created do
    for each State from I(o) do
        Create Composite Action
        State ⇐ Add Composite Action
    end for
end for
```

At the end of each episode, the agent saves the current path that leads to the goal. Then it will try to identify useful subgoals from the intersections performed with its saved paths. If new subgoals are discovered, it will dynamically create options that will lead the agent to the subgoals. If the agent has only a small number of paths due to the fact that it is still in the initial stages of learning, the subgoals discovered will be far from perfect. As the agent gains knowledge of the surrounding environment and additional paths are added to the path list, the states targeted as subgoals should stabilize. However, as the agent's policy continues to improve, the composite actions of the existing options that correspond to the optimal path, will begin to excel amongst the other composite actions. The states which truly represent subgoals are those that appear early with the path intersections and persist throughout learning, since the intersection process was designed to auto-exclude the false positive subgoals.

We don't limit the number of options created by the agent at each episode, but we are aware that too many actions can slow learning by creating a larger search space for the agent. To avoid this from happening, we only try to discover new subgoals and create new options, when the last path taken by the agent is shorter than the previous shortest. This will improve the agent's learning by making new composite actions available with a higher quality.

**Algorithm 9** Create Initiation Set( )

InitialState $\Leftarrow$ first state of the room
FinalState $\Leftarrow$ final state of the room
Sub-paths $\Leftarrow$ sub-paths starting with InitialState / ending with FinalState
Initiation Set $\Leftarrow$ union of all the different states from Sub-paths

Once a subgoal has been successfully discovered, it is used to create a new option. The option's initiation set, $I$, has to be initialized upon the option's creation. As we already have evidenced in chapter 3.3, the initiation set $I$ comprises the states within the room. At each room exists an initial and a final state. The initial state represents the state where the agent enters the room, and the final state represents the state where the agent exits the room. The method that we used to create the initiation set is described in the algorithm 9. The termination condition $\beta(s)$ for each doorway option is zero for states $s$ within the room and 1 for states outside the room. Meaning that the option executes either until the subgoal is achieved or until the agent exits the initiation set. The option's policy,

$\pi$, is initialized by creating a new value function that uses the same state representation as the overall problem. The option's value function is learned using Q-learning (Watkins and Dayan, 1992) as the overall problem value function.

In order for our intersection algorithm to work, we need to save all the paths taken by the agent throughout learning, this introduce questions about the efficiency of our method in terms of space. This has not proven to be a limitation over a number of different tasks that we had experimented. But we decided to observe the memory usage of our software by using a profiling tool.

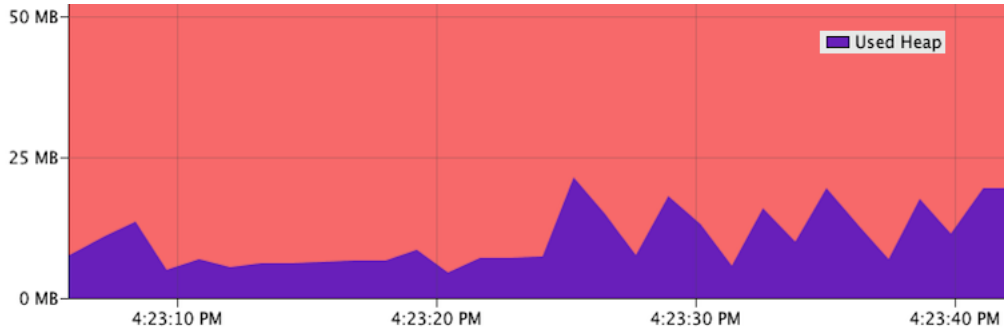



FIGURE 5.1: Memory Usage when using Flat Q-learning algorithm

The first experiment was comprised of 30 trials, each with 10000 episodes, only using flat Q-learning and observing the memory usage. From figure 5.1 we can tell that memory usage was always inferior to 25 megabytes (MB) with some value fluctuation.

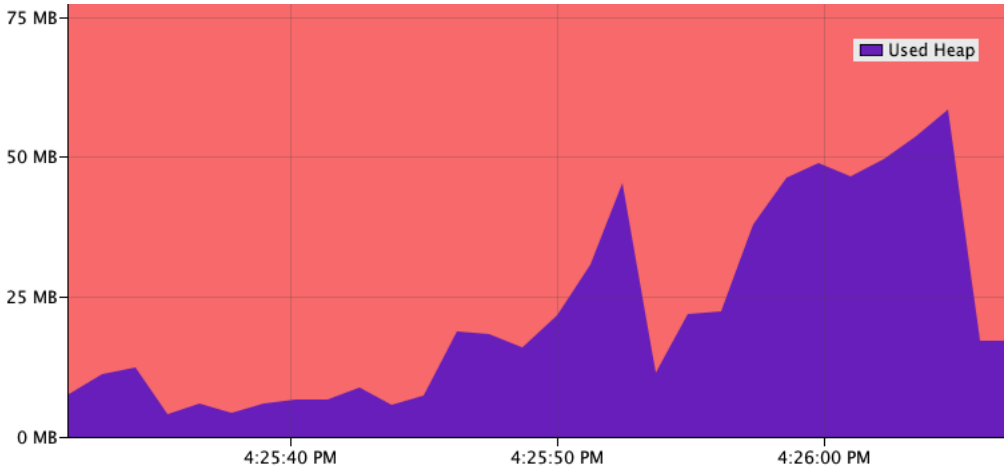



FIGURE 5.2: Memory Usage when using Subgoal Discovery with Options

In order to compare the memory usage of our algorithm, we made the same experiment, but this time, with subgoal discovery and option creation enabled. As

shown in figure 5.2 the implications of our algorithm in memory usage are quite relevant, since the amount of used heap towards the end of the experiment, now has exceeded 50 MB. With some optimization the amount of memory required could be lowered, by excluding the paths with the less occurring classes, or only saving the last $n$ paths.

## 5.2 Experimental Results

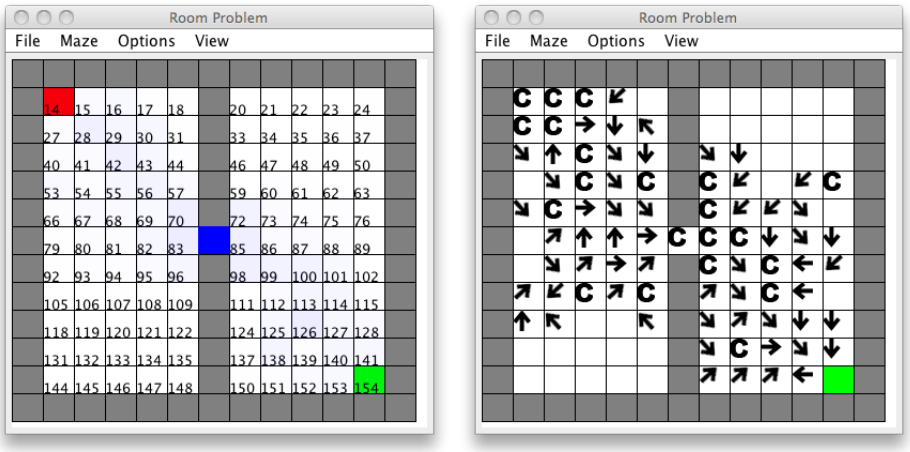
The following results will illustrate the performance gain obtained when using our subgoal discovery method and options creation. We will show why the newly created options prove to be important, and how the options facilitate the knowledge transfer between tasks. The learning parameters used for these experiments are the same as described in chapter 3.3, the results obtained will be described for several grid-world problems. The performance in each of the experiments is measured by the number of steps that the agent took to reach the goal on each episode averaged over 30 runs.

### 5.2.1 Two-Room Grid-world

The first experiment is the two-room environment shown in figure 5.3.The agent's purpose is to reach the main goal (represented in green), through the shortest route from a random state in the left-hand room. In this grid-world exists only one doorway, so only one sub-goal should be discovered, as shown in figure 5.3a.

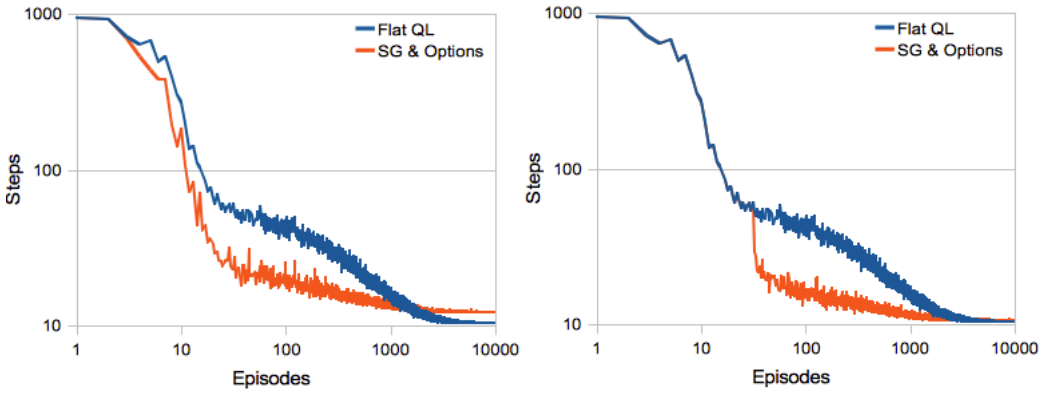
In figure 5.3b is possible to observe the policy learned by the agent after discovering the subgoal and creating the subgoals where in the initial states from the left-hand room the agent has chosen to take composite actions instead of primitive actions.

Figure 5.10 shows learning curves comparison between conventional Q-learning and Options. The initial episodes were the same until the agent, using subgoal discovery starts to use fewer steps to achieve the goal right after the second episode (fig. 5.4a). Learning with autonomous subgoal discovery and options has considerably accelerated learning compared to learning with primitive actions alone, but in the end, the agent using conventional Q-learning has obtained a better policy




(A) Subgoals discovered (B) Learned policy

FIGURE 5.3: Environment used for the two-room experiment



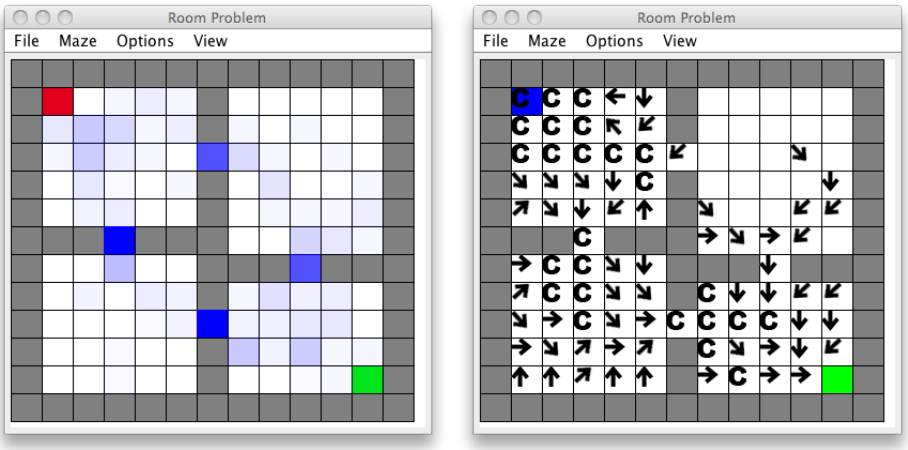

(A) Early SG discovery (B) Later SG discovery

FIGURE 5.4: Comparison of the evolution of average number of steps per episode between Q-Learning and Options in the two-room navigation task

than the one using options with a difference of more two steps. This unexpected behavior is due to the fact that exists a tradeoff between how quickly the agent starts to discover subgoals/create options and how approximate the options are in the overall task to an optimal policy. In figure 5.4a the agent tries to detect subgoals as soon as possible, but when converges fails to obtain an optimal policy. On the other hand, if we delay the subgoal discovery the agent manages to learn an optimal policy, like shown in figure 5.4b, where the agent only detects subgoals from episode 30. Again in this case, the advantage of using options is clear.

### 5.2.2 Four-Room Grid-world

As the next example, we used the four-room grid-world studied by Precup and Sutton (2000). This environment is shown in figure 5.5. The agent's task is to move from a randomly chosen start state in the left-hand room to the goal location in the lower right-hand room. The simulation conditions are the same as previously described. After 30 runs of 10000 episodes, the agent successfully identified four states located in the doorways as subgoals. An example of a policy containing primitive actions and options is shown in figure 5.5b.




(A) Subgoals discovered (B) Learned policy

FIGURE 5.5: Environment used for the four-room experiment

Again, we compare the results of learning with autonomous subgoal discovery to learning versus learning with primitive actions.

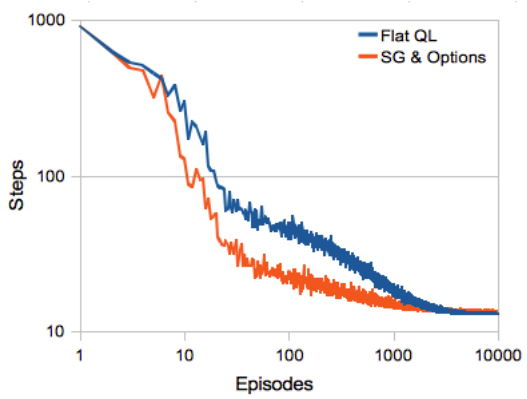
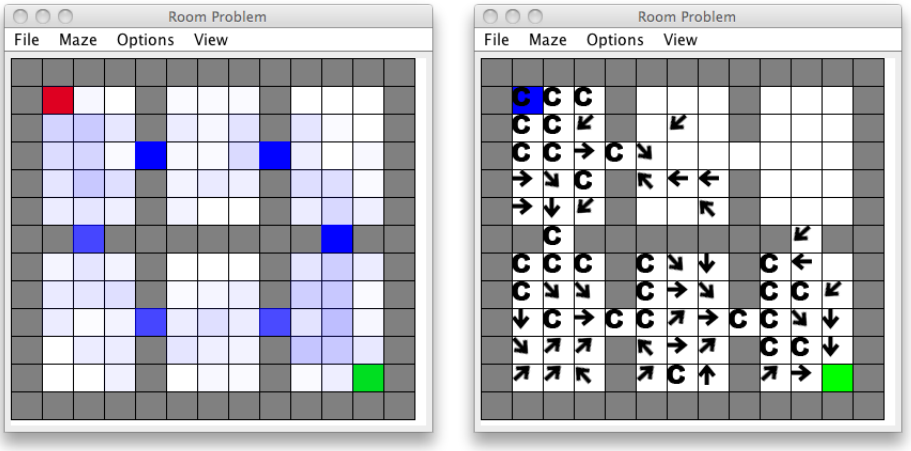



FIGURE 5.6: Comparison of the evolution of average number of steps per episode between Q-Learning and Options in the four-room navigation task

The comparison of the average number of steps that the agent needed to reach the goal state from the start state is illustrated in figure 5.6. Here the agent tried to detect subgoals as soon as possible, managing to detect some right after the second episode. From this point on the agent using options was able to learn an optimal policy faster than the agent using only primitive actions.

### 5.2.3 Six-Room Grid-world

To increase complexity we add two more rooms to the four-room grid-world, creating the six-room grid-world. Shown in figure 5.7. The agent has successfully identified the six existing subgoals in the environment. After 10000 episodes the agent learned a policy like the one shown in figure 5.7b.




(A) Subgoals discovered (B) Learned policy

FIGURE 5.7: Environment used for the six-room experiment

The results obtained from the average steps comparison between an agent with conventional Q-learning and an agent with options are illustrated in figure 5.8. And once more, our algorithm managed to obtain better results, taking less steps to reach the goal.

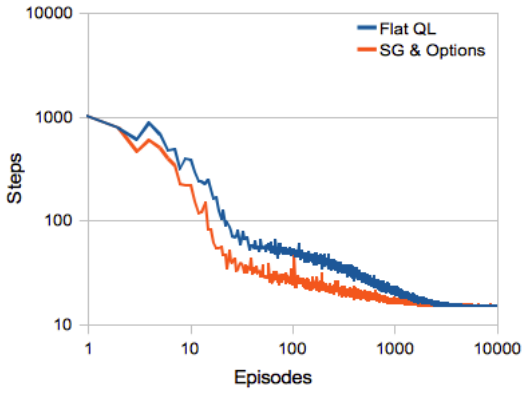
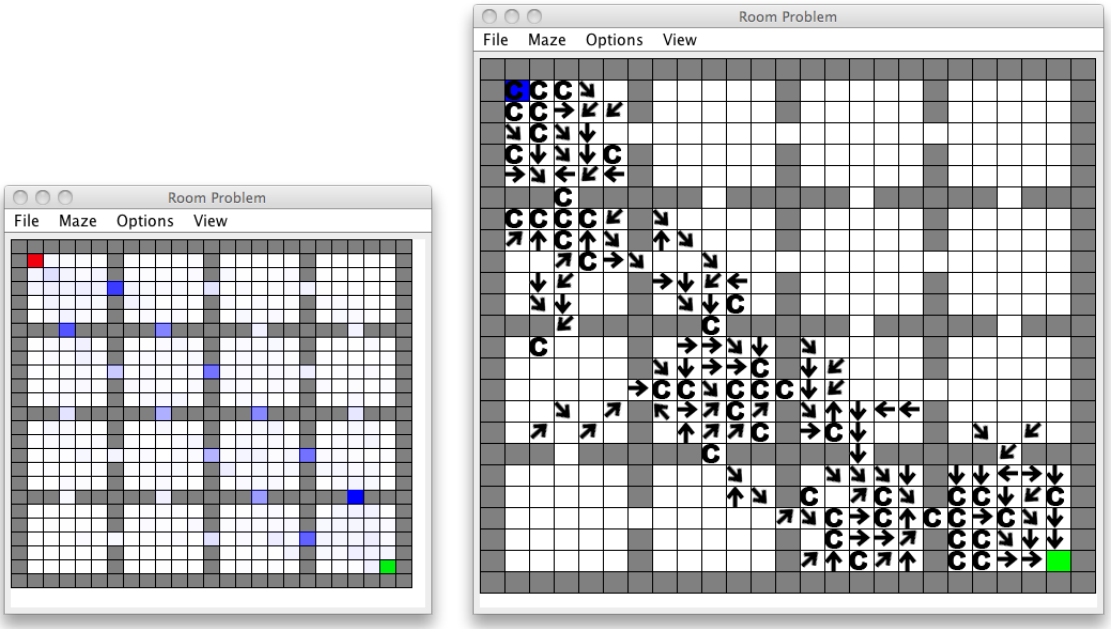



FIGURE 5.8: Comparison of the evolution of average number of steps per episode between Q-Learning and Options in the six-room navigation task

### 5.2.4 Sixteen-Room Grid-world

In the previous experiments, the number of rooms was increased, but the size of the environment remained the same. In this last experiment we intend to prove the robustness of our algorithm in autonomous subgoal discovery and dynamic option creation. Now the environment has 16 rooms, 24 doorways and a [20x20] state space. As we can observe in figure 5.9a, the agent has discovered the most relevant subgoals that can be found while taking the shortest path to reach the goal with a certain policy (fig. 5.9b).

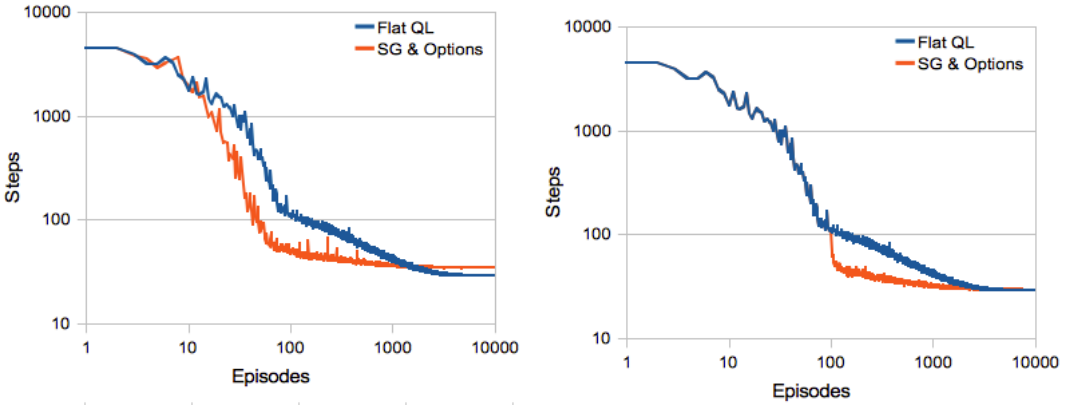



(A) Subgoals discovered (B) Learned policy

FIGURE 5.9: Environment used for the sixteen-room experiment

As we stated previously, there is a tradeoff between the quality of the options created and the time when the subgoals are discovered. In figure 5.10a we notice that the agent's performance initially, is worse when using autonomous subgoal discovery. Only on the tenth episode does it surpass the agent using conventional Q-learning which is also the one that learns a better policy for solving the task at hand. With little exploration and information of the environment, the process of discovering subgoals through intersection can be noisy. This leads to a creation of false-positive subgoals affecting negatively the agent's performance.

We tried to delay the subgoal discovery to a point where the final policy learned would be as good or even better that the one obtained through flat Q-learning. This was only possible by delaying the subgoal discovery 80 episodes (fig. 5.10b), which is a lot more than the previous 30 needed in the two-room grid-world. From this we assume that the more complex is the environment, more difficult is for the agent to learn an optimal policy while using options, even when the subgoals are correctly discovered.




(A) Early SG discovery (B) Later SG discovery

FIGURE 5.10: Comparison of the evolution of average number of steps per episode between Q-Learning and Options in the sixteen-room navigation task

Nevertheless, the results obtained corroborate the usefulness of the automated subgoal discovery, that associated with dynamic option creation, makes the agent a faster learner, reaching the goal in a more quickly way.

### 5.2.5 Task Transfer

We have already proven in chapter 3.3 that it is possible to transfer knowledge using options. In that case the options created were following an optimal policy

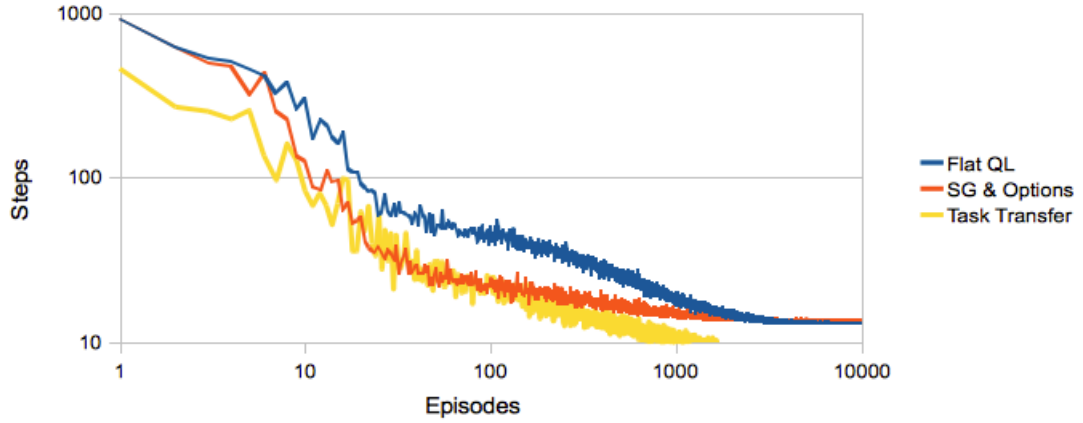



FIGURE 5.11: Comparison of the evolution of average number of steps per episode between flat Q-learning, subgoal detection with options and finally using knowledge transfer

and facilitated knowledge transfer in the referred tasks. The same experiment was made using autonomous subgoal discovery and dynamic option creation. The goal was moved to the middle of the upper-right room, and the options created previously to another task were made available to the agent. In figure 5.11 we compare the performance of the agent with and without the previously learned options and also with conventional Q-learning.

The agent reusing the options gained a performance boost right in the first episode, but after the tenth episode sometimes it performed worse. This could be related to the fact that the options sometimes will take the agent to the lower rooms, taking unnecessary steps. In the final episodes the agent obtains a better policy because the path to reach the goal now is shorter than the previous one. With this demonstration we have proven once again the capability of knowledge transfer between tasks using options.

The next chapter presents our conclusions and discusses possible future steps of this research.

# Chapter 6

# Conclusions and Future Work

*"What you get by achieving your goals is not as important as what you become by achieving your goals."*

Zig Ziglar

Our dissertation was focused on autonomously finding useful subgoals while the learning process was occurring. Several authors have approached the problem of discovering subgoals. We investigated the work done by Simsek and Barto (2004), Simsek et al. (2005) amongst others, which introduced the concept of Relative Novelty and Local Graph Partitioning. In our research we tried to develop a more simple approach that exploits the concept of discovering commonalities across multiple paths to a solution. This has already been tackled in the work of McGovern and Barto (2001) by using Diverse Density to identify useful subgoals. Our method is similar to the method created by McGovern and Barto (2001) in terms of concept, but different in terms of implementation and algorithm.

With the work developed in this dissertation we managed to introduce a novel method for automatically identifying useful subgoals called **Path Intersection**. From the experience obtained while interacting with an environment, a RL agent can identify useful subgoals and then create a particular type of temporal abstractions (Options). In order for our intersection algorithm to work, we need to save all the paths taken by the agent throughout learning. This allows us to make intersections between paths belonging to the same class. The resulting regions or states from those intersections will be identified as useful subgoals.

Several room-to-room navigation environments were used to test our subgoal detection algorithm. In every experiment useful subgoals were successfully detected. Then several simulated tasks where performed where options were dynamically created to achieve subgoals. Our method of subgoal discovery associated with options has proven that it can accelerate the agent's learning on the current task and facilitate transfer to related tasks.

The developed method could not be applied in every situation or task, still more researching and experimenting must be conducted to verify the scalability of our approach to different problems and environments. One of the main drawbacks of our method is that it requires several successful experiences from the agent to detect subgoals. Meaning that it first needs to reach the goal several times using only the given primitive actions and then try to detect useful subgoals. This in a very large and complex environment could reveal to be unfeasible or very time consuming. Future efforts should concentrate on optimizing this approach so that it applies to situations where the goal state cannot easily be reached using only primitive actions.

To increase the robustness of our method, the class path attribution should be optimized, making the process of subgoal detection less noisier. Other limitation refers to the option creation algorithm. The policy of our option is immutable, making the sequence of actions fixed once the option has been created. The option is created from the existing experience of the agent at that time contained in the previously saved paths. Adapting options to a changing environment, is only possible if the agent creates new better options and then forgets the old ones. An optimal solution would be for the agent to learn the policy option at the same time that it learns the global policy. This could be achieved with the use of pseudo-rewards and sub-state regions as the initial states of the options. Once an option is created, it remains accessible to the agent until the end of the task. There might be a point where the agent has more than one option referring a single subgoal. Currently, when that occurs, the agent simply chooses the one with a higher quality. In the long run one of these options could be negatively affecting the agent's performance by keeping the agent from discovering an even more efficient path to the goal

The concept of finding useful subgoals can be extended to the point that the agent could create abstractions that identify unwanted regions of the state space, like holes or cliffs. Focusing in the safest and rewarding areas of the environment.

Also some experiments were conducted with a real robot, namely a LEGO Mind-Storms NXT that learned to solve the taxi problem using conventional Q-learning. It would be interesting to see how our method could affect the performance of the robot, but due to time limitations that was not possible.

The results obtained confirms a different successful approach for automatic subgoal discovery. The next step is to extend our research to real life situations and see how it works.

# Appendices

# Appendix A

# Simplified Class Diagram

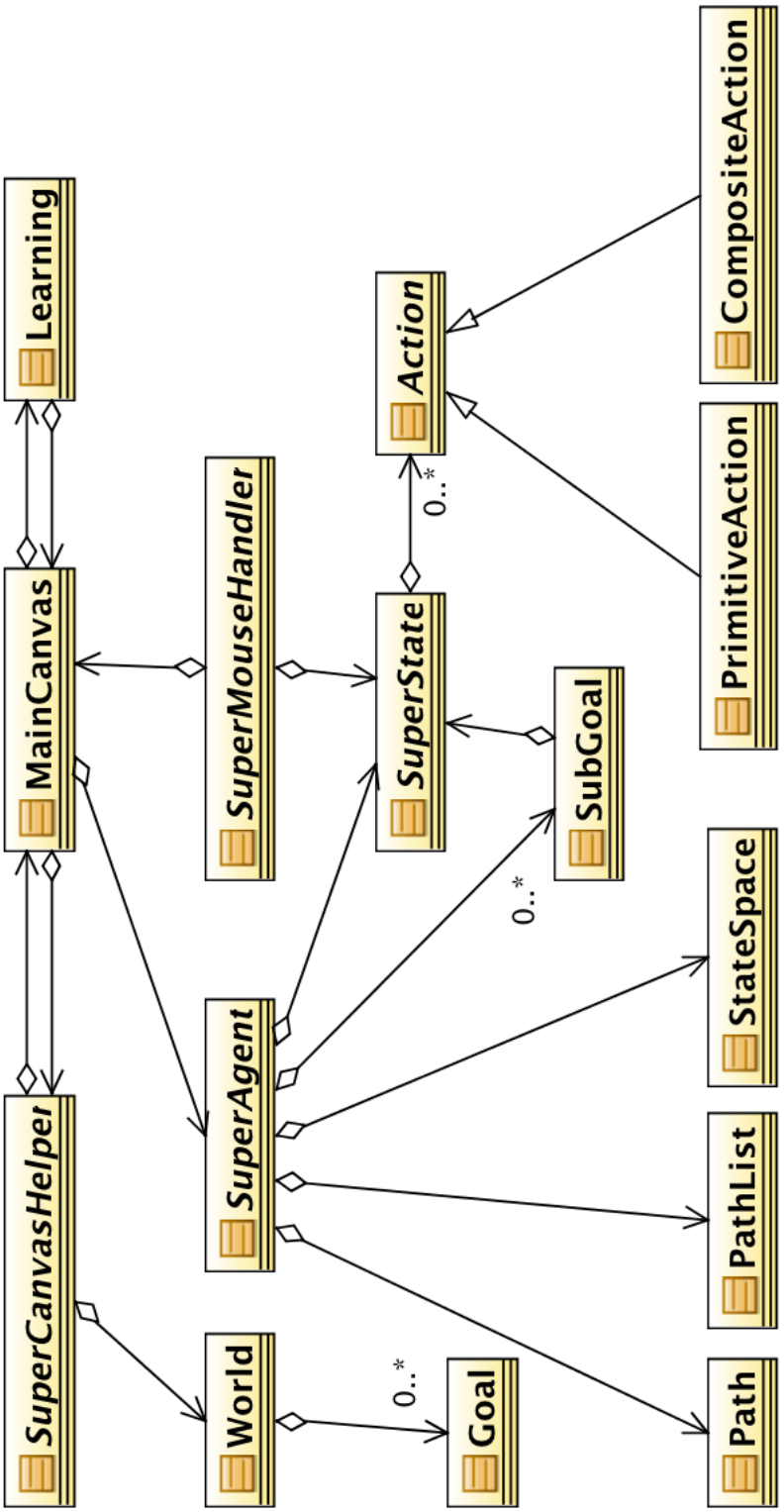



FIGURE A.1: Framework Simplified Class Diagram

# Bibliography

Andrew G. Barto and Sridhar Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(4):341–379, October 2003. doi: 10.1023/A:1025696116075. URL http://www.springerlink.com/content/tl1n705w7q452066.

TG Dietterich, RH Lathrop, and T Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 1997. URL http://linkinghub.elsevier.com/retrieve/pii/S0004370296000343.

Thomas Dietterich. *Abstraction, Reformulation, and Approximation*, volume 1864 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000a. ISBN 978-3-540-67839-7. doi: 10.1007/3-540-44914-0. URL http://www.springerlink.com/content/w11kby3dh4veehdn.

Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13 (1), 2000b. ISSN 1076-9757. URL http://portal.acm.org/citation.cfm?id=1622262.1622268.

Bernhard Hengst. Discovering Hierarchy in Reinforcement Learning with HEXQ. *Proceedings of the Nineteenth International Conference on Machine Learning*, 2002. URL http://portal.acm.org/citation.cfm?id=645531.656017.

LJ Lin. Reinforcement learning for robots using neural networks. 1993. URL http://en.scientificcommons.org/4909426.

O Maron and T Lozano-Pérez. A framework for multiple-instance learning. *Advances in neural information processing*, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.7638&rep=rep1&type=pdf.

A McGovern and AG Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. *MACHINE LEARNING-INTERNATIONAL*,

2001. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.2517&rep=rep1&type=pdf.

EA Mcgovern. Autonomous discovery of temporal abstractions from interaction with an environment. 2002. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.3079&rep=rep1&type=pdf.

Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Proceedings of the 1997 conference on Advances in neural information processing systems 10*, 1998. URL http://portal.acm.org/citation.cfm?id=302528.302894.

D. Precup and R.S. Sutton. Temporal abstraction in reinforcement learning. *University of Massachusetts Amherst*, 2000. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.849.

Özgür Simsek and Andrew G. Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. *ACM International Conference Proceeding Series; Vol. 69*, 2004. URL http://portal.acm.org/citation.cfm?id=1015353.

Özgür Simsek, Alicia P. Wolfe, and Andrew G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. *ACM International Conference Proceeding Series; Vol. 119*, 2005. URL http://portal.acm.org/citation.cfm?id=1102454.

SP Singh. Transfer of learning across compositions of sequential tasks. *, Eighth International Conference on Machine Learning*, 1991. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.32.6378&rep=rep1&type=pdf.

R. S. Sutton. *Reinforcement Learning: An Introduction*. MIT Press, May 1998. ISBN 0262193981. URL http://www.amazon.co.uk/dp/0262193981.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence*, 112:181–211, 1999. doi: 10.1.1.45.5531. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.5531.

RS Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*,

1996. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.51.4764&rep=rep1&type=pdf.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992698. URL http://www.springerlink.com/content/p120815501618373.