Learning Tasks Faster using Spatio-Temporal Abstractions

Miguel Fidalgo Martins

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Professor Manuel Fernando Cabido Peres Lopes

Examination Committee

Chairperson: Professor Miguel Nuno Dias Alves Pupo Correia
Supervisor: Professor Manuel Fernando Cabido Peres Lopes
Member of the Committee: Professor Francisco António Chaves Saraiva de Melo

November 2019

To my family and friends

Resumo

Algoritmos simples de aprendizagem por reforço, como Q-learning, são ótimos para aprender a resolver problemas com informações e definições mínimas do domínio do problema, o que é poderoso porque permite o uso de um único algoritmo para resolver problemas diferentes. No entanto, abordagens simples como o Q-learning não tentam entender o problema nem o mundo em que estão, elas resolvem o problema ao tentarem de tudo e aprendendo/memorizando o que fazer em cada passo. Esta abordagem irá falhar rapidamente ao tentar aprender tarefas mais complexas, as quais poderá não conseguir resolver em tempo útil.

Neste trabalho, apresento uma abordagem que integra aprendizagem por reforço com Abstração Espaço-Temporal, com o objetivo de permitir uma aprendizagem mais rápida de tarefas, especialmente à medida que a sua complexidade aumenta. O foco desta abordagem está sempre na compreensão do mundo, criando e atualizando uma base de conhecimentos que é então tida em consideração ao tomar uma decisão. Esta abordagem cria, representa e aprende autonomamente conhecimento de maior complexidade e generalização que métodos comuns de aprendizagem por reforço, como o Q-learning. Apresento também uma implementação desta abordagem, que chamo de ”Abstraction Agent” ou Agente de Abstração, sendo este agente o resultado da integração de um agente Q-learning com a minha abordagem.

Resultados experimentais mostram que esta abordagem aumenta a velocidade de aprendizagem do agente, reduzindo a quantidade total de passos necessários para que o agente alcance consistentemente o objectivo e resolva a tarefa. É também mostrado que este aumento no desempenho permanece até em mundos sem nenhuma abstração intuitiva.

Palavras-chave: Abstração Espaço-Temporal, Aprendizagem por Reforço, Generalização, Motivação Intrínseca, Deteção de Subobjetivos

Abstract

Simple reinforcement learning algorithms, such as Q-learning, are great in learning how to solve problems while being provided minimal information and definition of the problem domain, which is powerful because it allows the use of a single algorithm to solve different problems. However simple approaches like Q-learning don't try to understand the problem nor the world they are in, they solve it by trying everything and learning/memorizing what to do in each step. This approach will rapidly show its flaw when trying to learn more complex tasks, where it might not be able to solve in useful time.

In this work I present an approach that integrates reinforcement learning with Spatio-Temporal Abstraction, with the objective of allowing a faster learning of tasks specially as their complexity increases. This approach's focus is always in understanding the world by creating and updating a knowledge base that is then exploited and considered when making a decision. This approach autonomously creates, represents and learns knowledge of higher complexity and generalization than common reinforcement learning methods like Q-learning. I also present an implementation of this approach which I call Abstraction Agent, this agent being the result of the integration of a Q-learning agent with my approach.

Experimental results show that this approach increases the learning speed of the agent, reducing the total amount of steps necessary for the agent to consistently reach the goal and solve the task. This performance increase is also shown to remain even in worlds with no intuitive abstraction.

Keywords: Spatio-Temporal Abstraction, Reinforcement Learning, Generalization, Intrinsic Motivation, Subgoal Discovery

Contents

Resumo v
Abstract vii
List of Algorithms xi
List of Figures xv

1 Introduction **1**
1.1 Background 1
1.2 Contributions 4
1.2.1 World abstraction 4
1.2.2 Knowledge abstraction 4
1.3 Document Organisation 4

2 Related Work **6**
2.1 Integration 6
2.2 Temporal Abstraction 7
2.3 State Abstraction 7
2.4 Spatio-Temporal Abstraction 8
2.5 Intrinsic Motivation 9

3 Learning Tasks Faster using Spatio-Temporal Abstractions **11**
3.1 Abstraction Agent 11
3.1.1 World Abstraction 12
3.1.2 Knowledge Abstraction 15
3.1.3 Integration 17
3.2 Implementation 18
3.2.1 World Abstraction 18
3.2.2 Knowledge Abstraction 22
3.2.3 Integration 24
3.2.4 Real time savings 29

4 Experimental Results **32**
4.1 Systems and parameters settings 32

4.1.1 Domain 32
4.1.2 Reward System 32
4.1.3 Running System 33
4.1.4 Termination System 33
4.1.5 Termination Conditions 33
4.1.6 Performance evaluation graphs and parameters 35
4.1.7 Worlds and maps 36
4.2 Toy world evaluation 36
4.2.1 World abstraction 36
4.2.2 Knowledge Abstraction 43
4.2.3 World and Knowledge Abstraction 43
4.2.4 World Abstraction flaw 48
4.2.5 Experience Replay 51
4.3 Playroom 53
4.4 Minecraft 54

5 Conclusions **58**

6 Future Work **60**

Bibliography **62**

List of Algorithms

1 Segmented S-cut algorithm 20
2 Action step 28
3 Learning step 30
4 Running system 34

List of Figures

3.1 Example abstraction of a world. The bottom layer is an MDP or the model of the world and the top layer is an abstraction of that world, the top layer is the representation that the agent tries to create in the world abstraction part of the agent. 16

4.1 4-rooms map, the simplest map of the toy world. The black squares are empty space and the grey squares are wall or obstacles. The blue square is the source and the green square is the goal. The image also shows an agent's learned trajectory from source to goal. 37

4.2 Graph comparing performance evolution between the Q-learning agent and the Abstraction Agent in the 4-rooms map. Graph is the cumulative rewards the agent receives in a episode over said episodes. As we can observe the cumulative rewards received in a episode increase as the agent learns and both agents learning evolution is similar. It can be seen that they stabilize for around 200 episodes before stopping because the $nrEpisodesForError = 200$. 37

4.3 Comparison between Q-learning agent and the Abstraction Agent with the world abstraction active. As expected the world abstraction provides faster learning. 38

4.4 Comparison between the Q-learning agent and the Abstraction Agent with world abstraction active in the 24-rooms map. 39

4.5 Visual representation of the abstraction done over the world. Minimum cut quality = 1000. 40

4.6 Comparison between abstraction agent's with different minimum cut quality. Different quality thresholds above 1000 have a similar performance because they all create meaningful abstractions even if in different quantities. In contrast, the abstraction made with a quality threshold of 200 accepts such low quality cuts that it over-abstracts the state-space, increasing the quantity of abstract states while decreasing the average quality of them, thus creating difficulties in learning. 41

4.7 Abstraction comparison with minimum cut quality = 200, 5000 and 50000, respectively . . 41

4.8 Comparison of the state frequency (Q-learning is left and Abstraction Agent is right). The brighter a position in the map the more it was visited. It's possible to observe how the Q-learning agent over-explores the states closer to the start while the Abstraction Agent has a more uniform state frequency. This graph is of only 1 run in of each agent but the average results are similar. Q-learning: highest state frequency was around 70000 with lowest being around 700 and the average around 27000; Abstraction Agent: highest state frequency was around 14000 with lowest being around 400 and the average around 3700. 42

4.9 Comparison between the Q-learning agent and the Abstraction Agent in the open room map. Abstraction made is also shown. 44

4.10 Comparison between the Q-learning agent and the Abstraction Agent in the maze map. Abstraction made in the shown. 45

4.11 Comparison between Abstraction Agent's with different minimum cut quality in the open-room map. Variance not shown so it's easier to compare the final stages. 46

4.12 Abstraction comparison with minimum cut quality = 200, 5000 and 50000, respectively . . 46

4.13 Comparison between the Q-learning agent and the Abstraction Agent with only the knowledge abstraction active, in the Lava-24-rooms. In the first picture the lava kills and in the second it doesn't. 47

4.14 Comparison between the Q-learning agent and the Abstraction Agent with both components active, in the Lava-24-rooms. In the first picture the lava kills and in the second it doesn't. In the 1st image, the Abstraction Agent with only the world abstraction active isn't present because the agent wasn't able to solve the map due to the flaw explained bellow. 49

4.15 Comparison between the Q-learning agent and the Abstraction Agent in the Lava-96-rooms with lava killing, not showing variance. The Abstraction Agent isn't able to solve the Lava-96-rooms with lava that doesn't kill because of the flaw explained bellow. 50

4.16 Map to demonstrate flaw. A minimum cut quality of 25000 is used in order to generate the necessary abstraction to exemplify the flaw. It's possible to see in the 2nd image how the agent goes back and forth between those 2 map positions. This happens because as we can see in the 3rd image, the 1st abstract state is separated at the "door" from the 2nd one, so right after the option leads the agent from the 1st to the 2nd abstract state the agent is confronted with a "dead end" of lava pits, having no choice but to retreat. However after it retreats, that option is still the "best" choice and it still leads it there, because the option's objective is to lead the agent from the 1st to the 2nd abstract state while maximizing reward and that "door" is closer, so that is the policy that maximizes the option's intrinsic reward. With this abstraction the agent is almost never successful in learning the best global policy. 50

4.17 Map to demonstrate flaw. The default minimum cut quality of 1000 is used. It's still possible to see in the 2nd image how the agent goes back and forth between those 2 map positions. However, in this case the Abstraction Agent is able to learn a good policy because the 1st abstract state (bottom left) no longer has a single option with 2 separate "doors" as goals but has actually 2 options, one to go to the upper left abstract state and another to go to the bottom right abstract state, and as the agent continuously goes back and forth between those 2 abstract state (the 2 map positions from the 2nd image are in different abstract states), the value of the option to go to the bottom right abstract state eventually decreases enough for the agent to consider and choose the option to go to the upper left abstract state. 51
4.18 Comparison between the agent's using replay or not in the 24-rooms map. The second image is a zoom of the first one. 52
4.19 Comparison between the agent's using replay or not in the Lava-24-rooms map with lava that "kills". Not showing variance so it's easier to observe. 53
4.20 Comparison between the Q-learning agent and the Abstraction Agent. Learning rate = 0.1. The 2nd image is a zoom of the first one but not showing variance. 55
4.21 Comparison between the Q-learning agent and the Abstraction Agent. Learning rate = 0.01. The 2nd image is a zoom of the first one. 56
4.22 The level has 2 floors. The agent starts in middle room of the bottom floor which is composed of 9 rooms. In order to reach the top floor the agent needs to break the blocks that block the exits of the middle room, it has to go to one of the 2 corner rooms that have a stairs and then it has to climb the stairs to the top floor. This first floor is surrounded by lava that will kill the agent, immediately ending the episode and giving a negative reward. In the top floor the agent just has to reach the emerald green block in the center. 56

Chapter 1

Introduction

Classic logic based A.I. methods, e.g. expert systems, are based on creating knowledge representations of domains/worlds in order to solve problems or perform tasks in them, but creating these representations of the knowledge domain is a very complex and time consuming task.

Modern methods like reinforcement learning take a dynamic and behavioral approach, trying to learn how to behave instead of basing it on a knowledge domain. In reinforcement learning instead of using a definition of the knowledge domain to know how to behave and decide, it tries to learn how to behave and make decisions by trial-and-error. However, by avoiding the definition of the knowledge domain these methods are more limited in terms of generalization and representation of complex behavior, hurting their effectiveness when learning more complex tasks.

This work's motivation is to try to integrate of the qualities of logical based systems into reinforcement learning and testing to discover if it allows for a faster learning of complex tasks, without the manual, complex and time consuming definition and representation of the knowledge domain that is necessary in logical based systems.

1.1 Background

Logical based systems have higher generalization capabilities than machine learning methods because information about the world and a model or representation of it are painstakingly manually coded into those systems. Some logical based systems have tools which use concepts such as inference to derive new information from the information given but, although this helps alleviate the need to explicitly explain the world, when the world or domain changes all that information is of no use.

Machine learning methods, such as reinforcement learning which is the focus of this work, have a higher adaptability than logical based systems because they rely on none to little previous knowledge of the domain/world to be able to learn what to do and find a solution, allowing the use of a single algorithm to solve different problems. A simple approach that is able to learn solutions from 0, is to simply learn what action to take in each state by randomly exploring the world, one of the topics of discussion being when to stop trying to find and improve solutions and start exploiting what we know

(exploration vs exploitation). However this approach doesn't understand the world and as such isn't able to generalize what it learns, it has a very "shallow learning" similar to memorization. This approach is the best approach when the world is not very big or complex because exploring the world without stopping is faster than stopping to understand it, however as the world gets bigger and more complex it will gradually stop being efficient and eventually stop being effective.

An excellent example is given in "The gardens of learning: a vision for AI" [13] :

"(...) a bright 3 year old (...) can easily be taught to say, "29 x 29 = 841." The child then knows that 29 x 29 = 841. You can prove it by asking the child. (...) However, if you then ask, What is 17 x 17? he will either reply, "841" or say "Er." A child of 9, however, not only knows what 29 x 29 is but also can work out what 17 x 17 is.".

Imagine a goal which is to find out the result of a multiplication between 2 numbers. If we consider a world where it exists only the numbers 1 to 10, then the best approach is to use "shallow learning", i.e. memorize the answers through trial and error. But as the world gets bigger and more complex, from numbers 1 to 100, then from 1 to 1000, 1 to 10000, ..., this approach rapidly stops being viable.

So a different approach is required, an approach where the agent autonomously tries to actually understand the world, an approach where the agent instead of trying to learn directly what action to do in a state, tries to understand the state, reasoning the implications of the values of a state and generalizing the state, performing a **state abstraction**. There are many ways to perform a state abstraction, some works create abstract states by ignoring specific state features and grouping the different states that match after the change, others create abstract states by grouping states "closer" to each other relying on state transition data. In summary, an abstract state is a generalization of states, i.e. it is a group of states that have something in common, e.g. the same reaction/state change to an action or a similar location for behaviour/action decision purposes. It provides additional meaning to a state, the state stops being just itself, but itself and part of a group of states.

A famous state generalization method are Neural Networks, being quite effective in combination with reinforcement learning as we can see in the paper "Human-level control through deep reinforcement learning" [10], and it basically relies on using multi-layered neural networks to abstract the state-space multiple times over, where each layer is a level of abstraction. For example, if you train a deep neural network to classify images, the hope is that the first layer will train itself to recognize very basic things like edges, the next layer will train itself to recognize collections of edges such as shapes, the next will train itself to recognize collections of shapes like eyes or noses, and a last one could learn even higher-order features like faces. A layer in neural networks is like abstract states from states, the next layer then being "meta" abstract states from these previous abstract states, creating a hierarchy of abstraction like $pixels- > edges- > shapes- > eyes- > faces$, quite similar to how we abstract some of our knowledge of the world, like $person- > mammal- > animal$.

State abstraction is an essential step towards providing generalization capabilities to behavioral approaches.

In "The gardens of learning: a vision for AI" [13] the authors discuss not only what learning is and "shallow learning" or memorization, as can be seen in the earlier quote, but they also discuss how

learning should build upon itself in order to keep expanding the range of doable tasks, and one of the proposals to achieve this is the concept of adding macro-actions to the action-space to be able to learn and represent more complex behaviour. These macro-actions are simply a list of actions to be performed in order, they are the simplest forms of a temporally abstract action which are sometimes referred as skills. One work deeply related to this concept is the options framework [15], where an option is added to the action-space like in the gardens of AI proposal, but instead of being a simple macro-action it is a closed-loop control rule, i.e. instead of being a specific sequence of actions to perform, it is a new independent policy that at each time-step chooses an action to perform based on the state, which means it is responsive to state changes and has no specific number of actions and as such doesn't abstract a specific time interval.

Temporal abstraction, such as options, allow the representation of behaviour of higher complexity, which when learned accelerates the overall learning speed of complex tasks.

There are many factors that can affect the complexity of a task and how difficult it is to learn it, some of them are:

1. the dimension of the world or state-space;
2. the specificity of the behaviour required to achieve a goal or simply how difficult it is to reach a goal;
3. the scarcity of rewards or feedback given.

The impact of factor 1 and 2 can be mitigated by state abstraction and temporal abstraction respectively and the impact of factor 3 can be mitigated by using intrinsic motivation.

Intrinsic motivation provides an agent the capability of learning and acting independently of external feedback. Intrinsic motivation and it's integration with temporal abstraction has been discussed in works like [1] and it is even discussed with reinforcement learning in mind in "Intrinsically Motivated Reinforcement Learning"[2].

The approach we are looking for, an approach that provides a higher understanding of the world and generalization capabilities to reinforcement learning that could help it solve tasks faster and solve more complex tasks, should be a combination of **state abstraction**, **temporal abstraction** and **intrinsic motivation**.

In reinforcement learning methods the ultimate goal is for the agent to learn how to get as much reward as possible. Simple methods like Q-learning go for the straightforward approach of learning/memorizing what to do in each step. More complex methods, that use a combination of the concepts of state and temporal abstraction with intrinsic motivation in order to learn tasks faster (specially as the task complexity increases), already exist and some of them are:

- Integration of temporal abstraction, which they call options, and intrinsic motivation with reinforcement learning - [15, 2];
- Online discovery of subgoals to be used as the option's goal - Q-cut[9] and L-cut[14];
- Integration of state abstraction with temporal abstraction - [7] and HEXQ[5];

- Integration of state abstraction using neural networks with reinforcement learning - [10].

In my approach the ultimate goal is the same but the focus is always in understanding the world and later exploit this knowledge. My approach merges some qualities of logical based systems with behavioral approaches by in fact a combination of state abstraction, temporal abstraction and intrinsic motivation, and it has components that are based or inspired on components from all of the works above.

1.2 Contributions

In this document I present my approach, implement a reinforcement learning agent that integrates this approach with a Q-learning agent, which I call ”Abstraction Agent”, and also evaluate this new agent as being able to learn complex tasks faster than the original Q-learning agent it was built upon.

The Abstraction Agent autonomously creates and updates a knowledge base constituted by both space and temporal abstractions. This knowledge base will represent the understanding the agent has of the world. This knowledge base will be composed of 2 separate abstraction systems: World Abstraction and Knowledge Abstraction.

1.2.1 World abstraction

This abstraction system will be the main focus of this work. It will exploit the locality aspect of states to create and continuously update an abstract model of the world as the agent explores and learns the existing model of the world. This abstract model of the world is similar to the actual model of the world, but it is an abstraction of it, where instead of states it has abstract states and instead of actions it has abstract actions. An example of this abstraction can be seen in Fig.3.1. This abstract model is created independently of rewards, allowing it to abstract the world even when there is only a single reward signal, the reward for the goal.

1.2.2 Knowledge abstraction

This abstraction system will complement the approach. It consists of the use of a neural network to abstract a state using its features and then try to predict the immediate reward of each action. It does this by learning correlations between the feature values of a state and the reward received when performing an action. This allows the agent to generalize this knowledge between multiple states that share feature values. It is constructed based on the work in [10].

1.3 Document Organisation

In chapter 2 I will discuss various concepts and components used in my work, and other works that served as a base or inspiration for them.

In chapter 3 I will provide a description of both my approach and the implementation of the Abstraction Agent.

In chapter 4 I will evaluate the Abstraction Agent against the Q-learning agent it was build upon in order to evaluate the impact of my approach on the performance of the agent.

In chapter 5 I will provide a quick summary of the work and in chapter 6 I will discuss possible extensions and future work.

Chapter 2

Related Work

In this chapter it's reviewed various concepts used in my work and various works related to them.

2.1 Integration

The motivation of this work is integrating the qualities of logical based systems into reinforcement learning. The qualities from logical based systems come from the provided knowledge base of the world which is manually created. One of the main inspirations in achieving this integration came from the work "The gardens of learning: a vision for AI"[13] which discusses learning as an whole and from which I took the **approach** I use in this work: don't learn what to do directly from an observation, but learn what we observe or what the observation means, and then use that knowledge to improve the decision making. Basically make a reinforcement learning agent that is able to autonomously create its own knowledge base of the world from its observations. So the problem to solve or the objective is in creating an agent that autonomously creates its own knowledge base of the world.

A world is constituted by states and actions, and as such the knowledge base of the world contains, hopefully generalized, information of the states and actions in the world. This additional information of the world, the one that constitutes the knowledge base, is obtained or represented by performing what is usually called an abstraction of the world and as such, because the world is constituted by states and actions, there are 2 types of abstraction: State Abstraction and action abstraction which is most commonly called Temporal Abstraction. Both these abstractions are essential to the solution I propose in this work.

Another essential concept in this work is intrinsic motivation. Many complex tasks are complex because they are scarce on reward signals. This is to be expected because as the task complexity rises, the job of creating a meaningful and helpful reward function or system (that alleviates the complexity burden of the agent) becomes harder and more similar to the job of manually creating the knowledge base. As such the agent is expected to create the knowledge base using intrinsic motivation, and then exploit it for extrinsic motivations.

There are many works about both State and Temporal abstraction and also intrinsic motivation. Some

of them have approaches similar to this one, combining these concepts and even creating their own ”knowledge base”. I will now discuss these concepts in more detail, noting work related to them and their impact in my work.

2.2 Temporal Abstraction

Temporal Abstraction consists in the creation of abstract actions (groups of actions) which represent behaviour that takes undetermined steps to end (which is why its called ”Temporal” abstraction) and is composed by the original or ”primitive” actions which always take a single step to end.

The simplest form of temporal abstraction is a macro-action, which is simply a list of actions to be performed in order. One work deeply related to temporal abstraction is the options framework [15], where an option is an abstract action and is added to the action-space like in the gardens of learning [13] proposal, but instead of being a simple macro-action, it is a closed-loop control rule, i.e. instead of being a specific sequence of actions to perform, it is an independent policy that at each time-step chooses an action to perform based on the state, which means it is responsive to state changes and has no specific number of actions, i.e. it doesn’t abstract a static time interval.

”Intrinsically Motivated Reinforcement Learning”[2] is a work that integrates and discusses the use of options with reinforcement learning in more detail and also integrates the use of intrinsic motivation. In [2] the agent finds ”salient events” and creates options whose goal is to reach or create these ”salient events”. The definition of what constitutes a ”salient event” is manual and dependent on the world, and as such not a good fit for the approach we are looking for, for the Abstraction Agent. However, this work has provided great insight on the usefulness of options and their integration with Q-learning, providing ideas such as Intra-option learning (improving an option’s policy even when not following the option) which were essential when creating the Abstraction Agent.

The Abstraction Agent also uses options, but uses them in a way different from the options framework, some examples are:

- it integrates them with the concept of abstract state;
- it doesn’t actually add them to the action-space, and as such;
- the option’s policy only contains primitive actions.

2.3 State Abstraction

State Abstraction consists in the creation of abstract states (group of states) which contain information relevant to the original or ”primitive” states it’s constituted by.

A temporal abstraction, or option or sometimes called skill, exists for a reason, it has a goal, its policy representing the behaviour to achieve it. However finding those goals autonomously has been the discussion of many works.

Q-cut[9] and L-cut[14] use graph theory to analyse the agent's state transitions and find bottleneck states, these bottleneck states being the "border states" of strongly connected areas that would disconnect the graph if removed. Creating, learning and using options whose objective is to reach the bottleneck states allows an agent to search and navigate the state space more efficiently. Q-cut analyses the global state-space transitions known while L-cut only stores and analysis a local subset of the global state-space transitions known.

"Learning Purposeful Behaviour in the Absence of Rewards"[8] also analyses the agent's state transitions to find correlated features (features whose values change together), it then clusters these feature changes into a "purpose" or goal and creates an option whose goal is to perform the purpose, i.e. to create that feature's change. An example which is based on one that exists in the paper[8]: imagine the agent leaves a building; several of its features might change because of a single action (temperature, lighting, soil), and a new abstract feature or purpose (is-inside-building) that contains those features might be created along with an option with the goal of changing that feature. In sum, the algorithm creates an abstract state from correlated feature changes and creates an option to enter or leave that abstract state.

When these works divide a graph or state-space into 2 they are performing a state abstraction which creates 2 abstract states, however these works perform state abstractions with the sole purpose of finding good goals for the options, so some of them don't even store the abstract states found. The Abstraction Agent uses spectral clustering to find bottleneck states, however it stores and integrates the abstract states with the notion of options to perform spatio-temporal abstractions of the world, the abstractions that compose the abstract model of the world that is the core of the "world abstraction" component.

"Human-level control through deep reinforcement learning"[10] is a work which consists on the creation and usage of a neural network to learn and predict the Q-values of a state, a network which they call Q-network. By separating a state into its features and then providing them independently to the neural network as input, the neural network will predict the Q-value based on the individual features and the relation between them, instead of based on the state as a whole. This allows the neural network to learn about relations between individual feature values and the Q-value, knowledge that can be generalized to multiple states with those individual feature values present. In the work[10] a convolutional neural network is used because image's pixel information are used as input. The Abstraction Agent doesn't use images, however this work and its integration of neural networks with reinforcement learning were the base for the creation of the "Knowledge Abstraction" component of the Abstraction Agent. The "knowledge abstraction" component relies solely on state abstraction and not on temporal abstraction.

2.4 Spatio-Temporal Abstraction

Spatio-temporal Abstractions are an integration between space and temporal abstractions.

In HEXQ[5] the underlying model of the world is abstracted into separate layers or abstract states and the agent learns policies or temporal abstractions to navigate through these layers. HEXQ abstracts the

state-space of a single feature into an ”abstract state-space layer”, from now on just ”layer”, abstracting each single feature one at the time. The maximum number of layers is thus equal to the number of features. It starts with the feature that changes more frequently, following the idea that those features should be in a lower level of abstraction. From the agent’s trajectory it creates a graph where nodes are the feature’s different values and edges are transitions/actions. When creating the graph all other features are ignored, but ”exits” are added to the graph, exits being a state-action pair (s,a) where performing action ”a” in state ”s” will change a feature other than the one the graph and layer are about. Policies (or temporal abstractions) to lead the agent to an exit are created and learned. This will create the first layer; the next layer will cluster the abstract states (of that previous layer) into ”meta-abstract states” in which these ”next layer”’s feature changes and the previously learned policies are the abstract actions that change this ”next layer”’s feature. This creates hierarchical layers of abstraction. HEXQ seems to be a good approach only in specific worlds that don’t have many ”exits”, where the states are already constructed in a format where it’s features already have a hierarchy (like s = (room position, room number, house number)). However the idea of abstracting the state-space into abstract states and then creating abstract actions to navigate from one abstract space to another was a clear idea on how to integrate state and temporal abstractions and thus create spatio-temporal abstractions and their representation. The Abstraction Agent also creates abstract states and then abstract actions to navigate from one abstract state to a neighboring one.

”Hierarchical Reinforcement Learning using Spatio-Temporal Abstractions and Deep Neural Networks”[7] is a work that creates an abstraction structure and spatio-temporal abstractions similar to the Abstraction Agent. In this work[7] the agent stores it’s state transitions into a transition matrix (a graph), uses spectral clustering to find suitable state abstractions of the graph and then creates temporal abstractions or policies to go from one abstract state to another one. The resulted abstraction should be somewhat similar to HEXQ[5] without the need for a built in hierarchy in the state’s features. As said above, the Abstraction Agent also creates abstract states and abstract actions to navigate from one abstract state to another one, and this work provided a better and even clearer idea on how to integrate state and temporal abstractions. The Abstraction Agent also uses spectral clustering, however it performs state abstractions similar to the Segmented Q-cut[9] work and then creates abstract actions to go from one abstract state to only the neighboring ones, instead of all existing abstract states like in ”Hierarchical Reinforcement Learning using Spatio-Temporal Abstractions and Deep Neural Networks”[7].

2.5 Intrinsic Motivation

Intrinsic motivations are motivations that lead the agent into performing actions that might not be aligned with their extrinsic value, i.e. it leads the agent into performing actions that the agent doesn’t consider the best action to reach the goal. This can be helpful because performing a not ”optimal action” now, and thus reaching the goal later, might provide information and experience that allows the agent to reach the goal even faster in future.

In ”Intrinsically motivated learning of hierarchical collections of skills”[1] it’s said that it’s not concrete

in the psychology literature what drives intrinsic motivation but it should probably involve novelty, surprise, incongruity and complexity. In the playroom example, present in [1, 2], new options or temporal abstractions are generated based on the surprise factor, based on how salient an event is, and the intrinsic reward is based on a combination of novelty and incongruity. As is said in this paper, ”It (surprise) cannot account for what makes many forms of exploration and manipulation ’interesting’, thus other factors like novelty, incongruity and complexity might be necessary to be taken into account when learning on bigger and more complex worlds”.

This was taken into account when creating the Abstraction Agent and their impact is:

- when considering an action, the agent overvalues the option’s value by pretending that the agent can always reach the option’s goal in 1 step - this represents ”complexity”, it allows the agent to start choosing complex actions sooner
- when starting an option, the agent doesn’t stop following the option’s policy even when it’s bad and contradicts the global extrinsic policy - this represents novelty, it allows the agent to follow a new option’s policy and improve it faster even if it delays reaching the goal
- when discovering a bridge (transition between 2 abstract states) it has never seen before, the agent will sometimes recheck if the abstraction is ”good” - this represents incongruity, it allows the agent to reunite abstract states that are later found to be connected or that should have never been separated.

Chapter 3

Learning Tasks Faster using Spatio-Temporal Abstractions

In this chapter I present my approach which I called the ”Abstraction Agent”, an agent that is able to learn complex tasks faster (in fewer steps) by integrating state abstraction, temporal abstraction and intrinsic motivation with Q-learning. Then the Abstraction Agent implementation is discussed and explained in further detail, such as presenting pseudo-code/algorithms of important parts of the agent.

3.1 Abstraction Agent

In reinforcement learning methods the ultimate goal is for the agent to learn how to get as much reward as possible, with simple methods like Q-learning going for the straightforward approach of learning/memorizing what to do in each step. In my approach the ultimate goal is the same but the focus is always in understanding the world, creating a knowledge base that is then exploited and considered when making a decision.

The most important concepts are:

- **State Abstraction** - Not every state is completely different from one another, being possible to generalize knowledge or a behaviour to multiple states.
- **Temporal Abstraction** - It doesn't need to exist only an action that takes 1 step and a policy to reach the goal, there can be actions that take multiple steps, these being policies to intrinsic goals or sub-policies (in this work these are called ”options”, but other names include temporal abstraction and abstract action).

The abstraction agent will be composed of mainly 3 parts:

- **World Abstraction** - Abstracts the state space into multiple abstract states. These abstract states are separated by bottleneck states. Options are created with these bottleneck states as intrinsic goals. By separating the state space into smaller abstract states and creating policies/options to

go from one abstract state to another, the agent learns to navigate the state space independently of extrinsic rewards and explore the state-space more efficiently.

- **Knowledge Abstraction** - Uses a neural network to abstract the state, separating a state into its features and then providing them independently to the neural network as input. The neural network will predict the reward of each action based of the individual features and the relation between them instead of basing it on the state as a whole. This allows the neural network to learn about relations between individual feature values and the action's reward, knowledge that can be generalized to multiple states that share those individual feature values, allowing it to estimate the reward of each action in states it has never seen before.

- **Integration** - the core of the agent where it decides which action to take and learns. This part consists of the usual Q-learning implementation with additional considerations in order to integrate it with the world and knowledge abstraction.

3.1.1 World Abstraction

This component integrates both state and temporal abstraction, creating an abstract representation of the state space that allows it to learn how to better navigate the state space independently of extrinsic rewards and explore the state-space more efficiently, thus learning tasks faster.

In this component's spatio-temporal abstraction representation, abstract states are divisions of space (or subsets of S) separated by bottleneck states, and options always belong to an abstract state and are policies on how to go from the abstract state they belong to into a neighboring abstract state, through the bottleneck states that separate them. Abstract states are basically containers of options that are useful for traversing the state-space through its bottleneck states. A visual representation of this can be seen in Fig.3.1. There are also options whose goal is an actual extrinsic goal and not a neighboring abstract state, although not necessary, these options also increase the learning speed.

Options - temporal abstraction

In normal reinforcement learning approaches, such as Q-learning, the agent is trying to learn a policy to maximize the rewards it receives. If we give a task for the agent to learn in the way of a reward, it will learn a policy that says what to do to in each state to get that reward. However, it's hard to learn complex behaviour directly because the agent needs to perform the complex behaviour at least once to receive the reward, and even when it does, it takes time to converge into a desirable policy.

A better method to learn how to achieve a complex goal (or solve a complex problem) is to split it into smaller pieces. The abstraction agent learns multiple policies to achieve sub-goals. These policies are **temporal abstractions or options** which could be sub-policies of the desired global policy and in turn accelerate learning of complex behaviour. These options also allow the agent to instead of just considering actions that take him 1 step away and whose value might be fairly similar, consider following a policy that will take him multiple steps away to someplace in the state space that can be more valuable,

which allows the agent to explore the state space more efficiently by being able to navigate to more valuable places quicker.

The temporal abstraction or options used in the agent are based on the options or option framework presented on these works [15, 2]. Formally an option on the options framework is a triple o = (I, p, T) :

- I : initiation set : the set of states in which the option can be chosen or started
- p : is the policy followed while the option is being followed
- T : termination conditions: declares when the the option ends

In my approach some changes are made to the options that simplify and differentiate them and their use from the traditional options from the works[15, 2]. Some differences are:

- My options always belong to an abstract state and can be started anywhere on that abstract state but only on that abstract state. Therefore my options are only a pair o = (p,g), p being the policy and g the goal states of the option. This is because the I: initiation set is not only already implicit but somewhat static, the initiation set doesn't grow like the usual options, it is always equal to the set of states that constitute the abstract state and only grows as a side effect of the abstract state growing;
- My options don't have an option model, they don't have a transition probability model nor a reward model. The value of an option is updated similarly to a normal action when the option ends;
- My options learn and store the extrinsic value of each bridge transition so the agent can make a choice between multiple bridges if necessary;

The problem is finding useful intrinsic goals or subgoals to perform the desired task. We humans usually know the goal we want to achieve when we create subgoals, but the agent doesn't. As such, the agent will need to autonomously discover useful subgoals based on its observations and with no knowledge of the goal.

Segmented S-cut - state-space abstraction and subgoal discovery

The state space abstraction and sub-goal discovery, or Segmented S-cut, is based on the Segmented Q-cut [9] with inspiration on other works [14, 7].

The agent autonomously discovers useful subgoals based on its observations and with no knowledge of the goal. It does this by saving the state transitions of the agent in the form of a graph and then using spectral clustering to find the 2 clusters that best divide the graph, i.e. to find the best cut of the graph. Each cluster will be an abstract state. The state transitions from one cluster/abstract state to another, or bridges of the graph, will be an approximate equivalent to the bridges that would be found in a Min-Cut/Max-Flow of the graph, and if a cut is "good" they will provide good bottleneck states to be used as subgoals.

The algorithm initializes with 1 abstract state that contains only the first state. When the agent finds a state that has never seen before (that doesn't belong to any abstract state) he attributes that state to

the abstract state that contains the previous state. After performing a spectral clustering on the graph corresponding to that abstract state, if a "good" cut is found, the abstract state and it's state transition graph will be divided into 2 abstract states, each with its own graph, and 2 options will be created using the bottleneck states as intrinsic goals (subgoal of the global policy), these options being policies to go from one abstract state to the other and vice-versa.

The calculation of the quality of a cut or how "good" it is, is similar to the one in Q-cut[9], i.e. $\frac{(Nr_of_states_in_1st_cluster)*(Nr_of_states_in_2nd_cluster)}{Nr_of_bridges}$

The only difference is that in the Abstraction Agent, the $Nr_of_Bridges$ is equal to the number of bridges in the cut **plus** the number of bridges (to other abstract states) that the abstract state being cut already had. This will allow the agent to consider the connectivity of the abstract state being cut to other abstract states in the calculation of the quality of the cut, not cutting abstract states whose cut quality might only be significant in the local spectrum.

I will call S-cut the process of dividing an abstract state into 2 using spectral clustering and creating the corresponding options.

This process will repeat itself recursively, i.e. if S-cut finds a good cut and splits an abstract state into 2, an S-cut will also be performed in each of the new 2 abstract states, and in being able to find good cuts those 2 new abstract states will again be each split into 2 and an S-cut will be performed on them. This recursion ends when an S-cut was performed on all abstract states and no more good cuts were found. This recursion is equivalent to the one in Segmented Q-cut in [9], where each segment in Segmented Q-cut will correspond to an abstract state in Segmented S-cut.

The differences between Segmented S-cut and Segmented Q-cut are:

- Segmented S-cut uses spectral clustering to find a good cut, which doesn't need a source state nor a target state like in Segmented Q-cut;
- For the spectral clustering algorithm, the arcs in the graph have a fixed capacity of 1 or no capacity. This is because it's beneficial to consider arcs equal to one another instead of rating them on some measure (like frequency) that is normally exploration based and doesn't necessarily hold any information about the underlying state transition structure that can be helpful in abstracting it;
- The state transition's frequency are saved and used when calculating if a cut is good. Regardless of the cut's quality, there are 2 conditions that need to be met in order for the S-cut to proceed (in the following examples the S-cut is trying to split an abstract state into abstract state A and abstract state B):
  1. There needs to be at least 1 bridge from abstract state A to B and another from abstract state B to A. This exists so a cut can't happen without the knowledge that both abstract states can reach the other.
  2. All bridges from A to B **or** from B to A need to have a certain minimum frequency value. This exists to make sure that the cut is real and the abstraction good, and not just an illusion created by a lack of exploration. A minimum frequency value of 3 is used because the lower

the value the sooner the abstractions are made, which is beneficial, and values above 3 show the same results.

- With each cut only 2 options are created, one to go from one abstract state to the other and vice-versa. The way the goals are attributed to options is explained in further detail in the implementation.

Intrinsic Motivation - incongruity

The work "Intrinsically motivated learning of hierarchical collections of skills"[1] refers 4 major sources for intrinsic motivation: novelty, surprise, incongruity and complexity. Surprise isn't considered because it can be related to both novelty and incongruity. Novelty and complexity are used when choosing an action. Incongruity however will be helpful here in the world abstraction.

As said in the beginning of the chapter, the goal of the world abstraction is to create and update a spatio-temporal abstraction representation of the state space, and the sooner the representation evolves, this is, the sooner the world is partitioned into abstract states and options, the faster its benefits will be of use to the Abstraction Agent, and as such the faster the Abstraction Agent will learn the task.

It's then important that the agent abstracts the world as soon as possible, however the faster it tries to abstract the world the worst the abstractions will be. It's a trade off between abstraction speed and the abstraction's quality.

This is where incongruity comes in, when the agent finds a new bridge between 2 abstract states, whether its between abstract states that already had bridges between them or discovering that 2 abstract states are actually connected, it will reassess the quality of the cut between those 2 abstract states, reuniting them if the quality is no longer sufficient. This allows the agent to abstract the world as fast as possible without lowering the abstractions quality, because when and if bad abstractions happen they can always be fixed later.

Also in the latter case, where the agent discovers that 2 abstract states are actually connected, because the abstract states were not separated by the result of a S-cut, there are no options to go from one abstract state to the other and vice-versa, even though they are neighboring abstract states. In this case the necessary options are created.

3.1.2 Knowledge Abstraction

As said above, not every state is completely different from one another, being possible to generalize a behaviour to multiple states. In the world abstraction we give extra meaning to states by grouping them by their proximity to one another, this proximity information having to be learnt from observations. In the knowledge abstraction we give extra meaning to states by finding extra meaning in its features, this extra feature meaning also having to be learnt from observations.

To do this the agent constantly updates a neural network with observations, the neural network trying to estimate the immediate reward of performing an action on a state. More specifically, the user manually classifies the features that it wants the neural network to pay attention to and their feature size

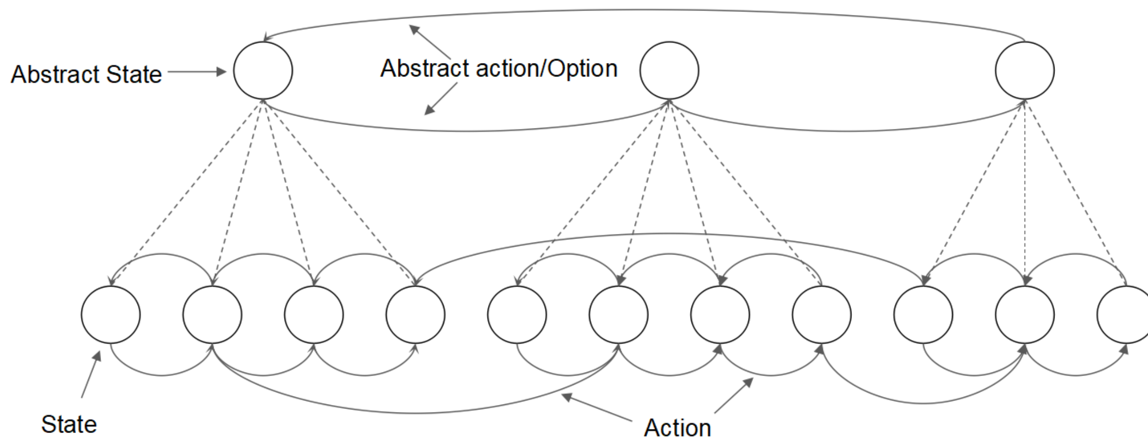


Figure 3.1: Example abstraction of a world. The bottom layer is an MDP or the model of the world and the top layer is an abstraction of that world, the top layer is the representation that the agent tries to create in the world abstraction part of the agent.

(number of different values the feature can have). Generally the user will tell the neural network to pay attention to all features that are not continuous and each of those feature's size. This is a sort of manual pre-abstraction that is common to all states.

After this pre-abstraction the state is fed as input to the neural network, which will then try to find suitable abstractions and correlations from the feature value's input in order to justify the immediate reward it observed. This allows the agent to be able to generalize behaviour to different states that have similar individual feature value's, including states it has never seen before. Example: the state is made of 4 features (x,y,z, HasLavaInFront). The objective is for the knowledge abstraction to learn that moving forward is bad when HasLavaInFront is true, regardless of x,y,z.

This neural network integration with reinforcement learning is similar to the one on the work[10], with some differences being:

- The use of a normal neural network is used, in comparison with the convolutional neural network used in [10], because there is no "locality" present in the features, i.e. the states are not image pixel information whose feature distance in the state is relevant

- The agent creates 2 databases of observations and updates them every step. Every X steps the agent will update the neural network using the timesteps saved in these databases. One database stores timesteps whose reward is (approximately) 0 and the other one stores all other timesteps. This separation exists so the timesteps with reward don't become diluted in the much larger number of timesteps that have no reward signal, allowing the agent to keep updating the neural network with useful timesteps that have a reward signal without overfitting. Other details are explained in the implementation part of this chapter.

- This neural network will predict only the immediate reward of an action, not the Q-values which take future action's Q-value into consideration.

3.1.3 Integration

This integration component is the core of the agent, where the base Q-learning will integrate the options from the world abstraction and the immediate reward estimates from the knowledge abstraction, where the inner mechanics of how the agent chooses an action and learns are. The integration of the options with Q-learning is based on the work "Intrinsically motivated reinforcement learning"[2] with some differences, a major one being:

- Options aren't added to the action space, and as such the action space is fixed and all policies only consider the actions in the action space. Options aren't considered in any policy. Learning an option's value and considerations for choosing an option (when an action needs to be chosen) are done separately. This is explained in further detail in implementation.

A simplified overview of how the agent chooses an action:

- With epsilon chance it chooses a random action - end, otherwise
- If an option is active
  - Check if the option is still available in the current state
    - If not available, deactivate the option
    - If available, choose the option's highest value action - end
- If no option is active
  - Find the action with the best value
  - Check the options available, if the highest valued option has higher value than the action found
    - That option becomes active and the option's highest valued action is chosen (an option has it's own policy) - end
  - Else
    - The previously found action is chosen. - end

("- end" signifies the action step ends, returning the action chosen).

A pseudo-code of this action step is showed in Algorithm2. The value of an action, regardless of the policy it's present on, is an addition of the learned action-value or Q-value and an estimation of the immediate action reward given by the knowledge abstraction.

The way the agent learns is:

- Timestep update is received (timestep update is (state, action, next-state, reward, episode-ended), explained in further detail later);
- Timestep update is sent to both world and knowledge abstraction components;
- All options available in that state use the timestep update to update their policy;

- If an option has reached it's goal, and therefore ended, its option value (in the global spectrum) is updated.

- The Q-learning global policy uses the timestep to update its policy.

3.2 Implementation

I will now explain in detail the implementation of the agent and its components. The agent's interaction with the environment relies on the basic reinforcement learning loop.

- Repeat
  - Environment has a state
  - Agent receives the state and decides what action to take - **Action step**
  - Action is performed in the environment, which returns the next-state, the reward, and if the episode ended with this action
  - Agent receives the timestep update and learns - **Learning step**
  - state $\leftarrow$ next-state

A timestep update or from now on simply timestep is the quintuple t = (state, action, next-state, reward, episode-ended) that contains all the information necessary to learn from the agent's interaction with the environment in a single step. The quintuple t = (state, action, next-state, reward, episode-ended) is:

- State - the current state of the world (for the agent)
- Action - the action the agent decided to take
- Next-State - the state of the world (for the agent) after if performs the Action
- Reward - the scalar reward signal correspondent to the interaction with the environment
- Episode-ended - if the episode ended because of the interaction with the environment (values are True or False)

3.2.1 World Abstraction

The world abstraction is constituted by the Segmented S-cut algorithm which creates the abstract states and the options.

Abstract State

An abstract state is composed of:

1. A graph: it has all the states that belong to the abstract state and the transitions between them (nodes are states and edges are transitions);
2. An option mapping: it lists all options belonging to that abstract state and their value.

Option

Temporal abstractions or abstract actions or options are based on the options framework. Formally an option on the options framework[15] are a triple o = (I, p, T), however our options are different and integrated with abstract states, as such our options are a double o = (p,g):

- p : is the policy followed while the option is active;
- g : a goal bridge mapping : it lists all goal bridges of the option and their value.

Bridges are state transitions composed by a source state and a destination state that belong to different abstract states. An option is available to be chosen and activated when the agent is in the abstract state that the option belongs to. An option ends and is deactivated when the agent leaves the abstract state that the option belongs to, hopefully by reaching one of the goal states of the option (the goal states of the option are the set of destination states from the goal bridges of the option). The objective of an option's policy is to lead the agent to that option's goal while maximizing reward.

Bridge values The "goal bridge mapping" of the option saves the goal bridges and maps them to their **bridge values**, which is necessary because the option's policy is only influenced by the option's intrinsic reward (and a possible immediate extrinsic reward) and as such all option's goals are equally important in the option's policy although being unequally important in the global picture, in the global policy. This option's reward system is necessary because the option's policy needs to be independent of the global policy in order to converge faster and therefore be useful faster. As such there can be actions with the same "option" value but with different "real" values. This presents a problem when the agent has to consider between multiple actions that take him outside of the option, i.e when the agent needs to choose between multiple bridges that all have the same "option" value but different "real" values. This is the purpose of the **bridge values**, it's a mapping where a separate value is given to each bridge of the option, this value being the value of performing that action in the global policy, their extrinsic value; then when there are multiple actions with the same value, instead of choosing one at random, the agent will use the bridge values to choose the action with the highest value in the global policy. This means that although multiple actions have equal value in the option's perspective, the agent will still choose the best action in the global perspective.

A summarized overview of the world abstraction can be found in the pseudo-code 1.

The $timestepUpdateWa$ procedure is called every timestep at the **Learning step** phase, the $doCuts$ procedure is called at the end of every episode.

S-cut

S-cut uses spectral clustering to find a good cut of the required state transition graph. The state transition graph is constructed using the python library "networkx"[4] and converted into a "scipy-sparse-matrix" that is then provided as input to the spectral clustering algorithm. The specific spectral clustering algorithm used is from the python library "scikit-learn"[12] and can be seen in `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.spectral_clustering.html` where it notes "This algo-

Algorithm 1 Segmented S-cut algorithm

Initialization:
-Create list of abstract states L
-Add an initial empty abstract state AS_0 to L
procedure TIMESTEPUPDATEWA(t)
    if $state = next_state$ **then**
        return
    end if
    $curStateAS, nextStateAS \leftarrow getAbstractStates(state, next_state)$
    if $curStateAS = nextStateAS$ **OR** $nextStateAS = None$ **then**
        Add state transition from the timestep to $curStateAS$
    else
        $incongruenceSystem(state, next_state, curStateAS, nextStateAS)$
    end if
end procedure
procedure DOCUTS
    for each AS_n in L **do**
        $cut \leftarrow doSpectralCut(AS_n)$
        $cutQuality \leftarrow calculateCutQuality(cut)$
        if $cutQuality > threshold$ **then**
            - Remove AS_n from L
            - Split AS_n into 2 new AS's (abstract states)
            - Change ownership of options in AS_n to the corresponding new AS's
            - Create options of cut and add them to the new AS's
            - Add the new AS's to L
        end if
    end for
    if any AS was split **then**
        call $doCuts()$
    end if
end procedure
procedure INCONGRUENCESYSTEM($state, next_state, curStateAS, nextStateAS$)
    if $(state, next_state)$ is a bridge from any option belonging to the $curStateAS$ (state transition is known) **then**
        return
    end if
    (get all the options that go from $curStateAS$ to $nextStateAS$)
    $options \leftarrow getOptionsFromAbstStates(curStateAS, nextStateAS)$
    if $options$ is empty (there are no options) **then**
        create an option that goes from $curStateAS$ to $nextStateAS$ using $next_state$ as it's goal
    else if nr of $options$ = 1 **then**
        $option \leftarrow options[0]$
        $addBridge(option, (state, next_state))$
        $cutQuality \leftarrow calculateCutQuality(curStateAS, nextStateAS)$
        if $cutQuality < threshold$ **then**
            - Remove $curStateAS$ and $nextStateAS$ from L
            - Unite $curStateAS$ and $nextStateAS$ into 1 AS (abstract state)
            - Delete options from $curStateAS$ to $nextStateAS$ and from $nextStateAS$ to $curStateAS$
            - Change ownership of options in $curStateAS$ and $nextStateAS$ to the corresponding new AS
            - Add the new AS to L
        end if

    else
        (there should never be 2 options to the same abstract state, but if there are, then unite the abstract states, fixing the situation)
        - Remove $curStateAS$ and $nextStateAS$ from L
        - Unite $curStateAS$ and $nextStateAS$ into 1 AS (abstract state)
        - Delete options from $curStateAS$ to $nextStateAS$ and from $nextStateAS$ to $curStateAS$
        - Change ownership of options in $curStateAS$ and $nextStateAS$ to the corresponding new AS
        - Add the new AS to L
    end if
end procedure

rithm solves the normalized cut for k=2: it is a normalized spectral clustering." k=2 meaning 2 clusters, which is the numbers of clusters S-cut is looking for. When the number of nodes/states in the graph is higher than 1000 it also uses the special eigen-solver "amg"[11] because "It can be faster on very large, sparse problems (...)".

This spectral clustering algorithm returns the 2 clusters of nodes it found and then S-cut iterates through all the edges/state transitions of the graph to get the edges that connect nodes in different clusters, these edges being the bridges. From the 2 clusters of nodes/states and the bridges, all the information necessary is present: the bridge's frequency, information to calculate the cut quality, where to separate the graph/abstract state into 2 and the goal states of the options to be created.

Abstract State division

Abstract states contain options and upon abstract state division those options also have to be distributed and maybe even divided. After an abstract state division the options contained in it can be of 2 types:

1. an option whose goal bridges are all contained into one of the new abstract states;
2. an option whose goal bridges are contained in both the new abstract states.

In the first case the situation is simple, the option will be delivered and contained by the new abstract state that contains all of its goal bridges and the option's policy is updated by removing all state-action pairs whose state does not belong in this new abstract state.

However in the second case the option can't simply be delivered to one of the new abstract states because it has goal bridges from both. In this case, the option will be divided into 2 new options, each abstract state will contain one of those options, and each option will have the part of the original goal bridges that belong to its new abstract state and will also update its policy by removing all state-action pairs whose state does not belong in their new abstract state.

After this, the options to go from one abstract state to the other are created. Their creation is explained bellow.

Option creation and goal attribution

The way goals are attributed to an option is the following:

- A good cut in found in abstract state h, which results into 2 abstract states k and l and the bridges between them
- Options are created for each abstract state, the option y being created for k
- The goal bridges of option y will be, from all the bridges, the ones that are state transitions from k to l

With this goal attribution, the option y will be a policy on how to go from anywhere in k into l, and a similar option is also created in l but instead to go from anywhere in l into k. Additionally, when the agent

finds an extrinsic goal, it will also create an option in which its goal bridges will be the state transitions to that extrinsic goal found, and then it will add that option to the current abstract state.

Incongruence System

In this system the agent will reunite abstract states that should be together and create and update options with new information. This system is used when the world abstraction receives a state transition from one abstract state to the another and it can be seen in greater detail in the procedure $incongruenceSystem()$ of algorithm 1 " Segmented S-cut ".

3.2.2 Knowledge Abstraction

In the knowledge abstraction a neural network is used to learn about relations between individual feature values and the action's reward, knowledge that can be generalized to multiple states that share those individual feature values, allowing it to estimate the reward of each action in states it has never seen before. This neural network is implemented using the "Keras" python API[3]. To do this I update a neural network similar to the one in "Human level control" paper [10].

The neural network:

- Has no convolutional layers
- Receives a processed state as input
- Has 4 layers: input layer with size equal to the input, if necessary this needs to be manually changed and it is static
- Outputs a value estimate for each action
- Is trained every X steps from a dataset built from observations
- States are processed, using an explicitly human defined manual abstraction and encoded using One Hot Encoding, before being used has input

The entire process is:

- Knowledge abstraction receives observation
- Observation is stored in dataset and if X steps have passed the neural network training commences
- If the training commences, for each observation in the dataset:
  - The state is processed using the explicitly human defined manual abstraction
  - Processed state is encoded using One Hot Encoding
  - Encoded state along with the action and reward from the observation are used to update the neural network

By doing this, with enough training, if there is extra meaning in the feature values the neural network will learn it and will be able to estimate the immediate value of performing an action in states it has never seen or performed that action on.

State processing

The states are processed by using a manual abstraction list, that is manually input by a person before the agent starts, and then by applying one-shot encoding on the state. It is this new processed state that is given has input to the neural network.

This manual abstraction list is necessary because the knowledge abstraction needs to know the variable size of the features so it can know the size of the neural network architecture to create. Also knowing which features are categorical is important because the knowledge abstraction doesn't consider continuous features (continuous features are ignored, although the manual abstraction list is a choice made be a person).

The knowledge abstraction focuses on the categorical features because they are the ones that will most likely "explain" why a reward is received and the ones were the potential for an increase in performance is higher. It allows the agent to focus on the features that are important and be able to predict faster and more precisely.

So, the agent needs the manual abstraction list in order to perform the one-shot encoding of the states and create the neural network whose size is not dynamic.

An example manual abstraction list for states s = (position x, position y, orientation, lavaInFront) is $L = [0, 0, 4, 5]$*. The first 2 features are ignored because they are ordinal and considered continuous. The last 2 features are then used, but although they are discrete they are still considered ordinal for the neural network when learning. This means the neural network would think North is equal to 2*South if North had a value of 4 and South a value of 2. To avoid this the manual abstraction list is used to one-shot encode the state, creating a new state that would have 9 binary features in total, 4 for the "orientation" feature and 5 for the "blockInFront" feature.

Example: applying one-shot encoding to the state $s = (34, 74, 3, 0)$ using the manual abstraction list $L = [0, 0, 4, 5]$ gives the new state ns = ((),(),(0,0,1,0),(0,0,0,0,1)), which summarized is $ns = (0, 0, 1, 0, 0, 0, 0, 0, 1)$. It's this new state that is given as input to the neural network.

*(the first 2 features are ignored so a 0 is given; 4 because the "orientation" feature as 4 possible values: North, South, West, East; and 5 because the "blockInFront" feature has 5 possible values: " " - empty space, "X" - wall, "L" - Lava, "S" - source or spawn, "G" - goal)

Dataset creation

As said previously, there are actually 2 datasets, one for observations with no reward and another for observations where reward is observed. This is also necessary because most observations will likely carry no reward and with only 1 (finite) dataset the observations with no reward would take the space for the ones that have, creating an illusion that no observation with reward was observed, causing the

neural network to learn nothing.

This was even more accentuated because of the knowledge abstraction itself. As the agent would learn, after some time, it's behaviour would change and the observations would change too, e.g. new observations where a negative reward was seen could disappear because the agent learned to avoid them, clearing the dataset of the few useful observations it had encountered as it would substitute them with more recent useless observations, which would in turn lead the agent to forget what it had learned.

Both datasets are built identically:

- A dataset is a list D of size Y, where Y is the number of features after processing a state or the sum of each feature size
- Each element in D is a queue, a fixed sized list that holds the 5 most recent observations where it's feature is present (where it's feature value is a 1)

So basically D contains 5 observations for each feature value.

Example: $L = [0, 0, 4, 5]$ means a processed state would have 9 binary features, so $Y = 9$ and D has 9 queues. Processing $s = (34, 74, 3, 0)$ results in $ns = (0, 0, 1, 0, 0, 0, 0, 0, 1)$, and as such the observation related to this state would be added to the 3rd and the last queues of D (where the 1's in ns are present) The dataset is built this way so the agent has always a diverse dataset to learn for, with observations from different feature values always stored, which is necessary in order for the agent to not overfit to only the most recent observations and to not forget what it has learned.

Confidence system

When asking for a prediction from the knowledge abstraction, there is a system that counts the number of times the neural network has predicted correctly for each specific feature value. If the neural network has predicted correctly enough times in the past then the prediction is sent, if not then the prediction sent is an blank estimate, neutral reward for all actions. This is a confidence system that allows the knowledge abstraction to only send predictions it has a certain amount of confidence on.

Neural network architecture

The specific neural network architecture used in this implementation consists of:

- An input layer the size of the processed state;
- 2 hidden layers, the first with a size of 1000 nodes and the second with 50 nodes;
- An output layer with the same size has the action space,
- All the layers are fully connected.

3.2.3 Integration

The core of the agent where Q-learning unites with world and knowledge abstraction. This integration can be separated into mainly 2 parts: **Action step** and **Learning step**.

Option choosing and learning

The way the abstraction agent chooses how to act and learns using options is different from the option framework[15] and this work[2], from where the options are based upon, because the abstraction agent maintains a higher degree of separation between actions and options.

In works that involve options, such as [15, 2], options are considered as actions that simply take multiple steps to end, and the original actions that take one step are called **primitive actions**. As such actions usually refer to both options and primitive actions.

In this work, because there is a higher degree of separation between actions and options, options are referred as options and primitive actions as just actions, there being no name for their combination. However the name "primitive actions" will sometimes be used for clarification and explanation purposes in order to create a clear distinction about what is being referred.

In the work "Intrinsically motivated reinforcement learning"[2] each option is added to the action set A and a value is learned and stored for each pair $(s, a)_{s \in S, a \in A}$, in both the global policy and each of the option's policy. This means that adding an option y not only increases the number of policies to be learnt by 1, y's policy, but every other policy would have to also learn and store the value of each pair $(s, y)_{s \in S}$. The number of values to learn and store is thus (Nr of states * Nr of actions * Nr of policies) or more concretely (A is number of actions, O is number of options and S is number of states):

$S * (A + O) * (O + 1)$

or

$S * (O^2 + (A + 1) * O + A)$,

which grows polynomially with the number of options. This is not scalable to a large amount of options, specially as the state-space size increases. Some methods could be implemented to reduce this problem:

1. reduce the learned state-space by using the option's initiation set I to reduce the number of states the option's policy has to learn to only pairs $(s, a)_{s \in I, a \in A}$;

2. reduce the learned action-space by considering other option's initiation set I when learning another option's policy, for example, when learning option y's policy learn only pairs (s, a) where a's initiation set contains s, being a an option.

This methods and others are all implicitly or explicitly implemented in the **Abstraction Agent** by a combination of the following:

1. The agent doesn't have a list of options, the agent has a list of abstract states and the abstract states contain options.

2. The action set is fixed, it only contains actions and is separated from the options.

3. Options will have a value attributed to them, which will represent how valuable it is to perform the option and will thus be used when choosing an action.

The implications that result from this are:

- the global policy learns pairs $(s,a)_{s\in S,a\in A}$ as in usual Q-learning, so the number of values is $S*A$.
- an option's policy only learns in states belonging to its abstract state, e.g. considering an abstract state l which contains the option y, y's policy only learns pairs $(s,a)_{s\in l,a\in A}$. The number of states in an abstract state varies but because the entire world is partitioned into abstract states, the states of every abstract state combined is equal to S. Considering AO as the average number of options in each abstract state, this means the number of values learned in all option's policies combined is equal to $S*AO$.
- the agent will only learn and store 1 value for each option, representing the value of performing that option, so the number of values is O.

As such the number of values to learn and store is thus (Nr of values in the global policy + Nr of values in the option's policies + Nr of option values) or more concretely:

$S*A+S*AO+O$

or

$S*(A+AO)+O$.

Both S and A are fixed so the number of values grows linearly with both O and AO. This is much more scalable to a large amount of options, even when state-space size increases.

(Note: AO can but doesn't always increase when O increases. This is because, with AS being the number of abstract states, $AO=\frac{O}{AS}$ and O can increase as a consequence of AS increasing.

Example: we have 1 abstract state with 2 options, so $AO=2$, and the abstract state is separated into 2, so 2 new options will be created, 1 for each abstract state, and still $AO=\frac{4}{2}=2$. This is because the abstract state's initiation set was split into 2, so although there are more options, the options became more focused, learning about a smaller subset of S.)

However, the value of performing an option will change depending on the state the agent is in, and the abstraction agent only stores a single value per option, not learning the value of pairs (state, option). This is because the abstraction agent uses a combination of the option's policy and the option's value to calculate the value of performing/starting the option in a specific state.

1. The value of an option is updated when that option reaches it's goal. The new value used for the update is the value of the highest valued action or option at the goal state. The value of an option will thus be an approximation of the average highest valued action in the goal states, it represents how good it is for the agent to be at the goal of the option.
2. The objective of the option's policy is to reach its goal, and the value of the highest valued action at a state, in the option's policy, represents how close that state is to the goal of the option.

Combining both "how good the goal of the option is" and "how close a state is to the goal of the option" we can get "how good the goal of the option is in a state".

Being y_v the value of option y or "how good the goal of the option is", $y_s max$ the $max_{a\in A}Q_y(s,a)$ or "how close the state is to the goal of the option", and IR the intrinsic reward for reaching an option's goal:

- IR is the highest possible value in the policy of any option;
- $y_s max/IR$ is a number between [0,1] that represents the same thing as $y_s max$ just on a different scale;
- $y_{v,s} = (y_s max/IR) * y_v$ is a number between [0,y_v] that represents how good the option y is in the state s.

As such, $y_{v,s}$ will be similar to the $Q(s, y)$ used in the work [2] but using the option's policy and the option's value, which were both already necessary.

Because of the separation between the actions and the options, when choosing an action the agent will compare the highest valued action from the global policy with the highest valued option from the options available in the current abstract state, i.e. it will compare $max_{a \in A} Q(s, a)$ with $max_{y \in O(s)} y_{v,s}$ and choose from the 2 the one with the highest value. ($O(s)$ is the set of options contained by the abstract state where s belongs)

Action step

Q-learning with an ϵ-greedy policy would just return the highest valued action for that state but the abstraction agent also has options and the knowledge abstraction to consider. An overview explanation of the algorithm was showed in the previous chapter. A simplification of the action selection algorithm is available in algorithm 2.

The procedure $getAction()$ in algorithm 2 isn't called directly, this is because the agent runs with an ϵ-greedy policy and sometimes the agent returns a random action instead of calling that procedure. However, this agent actually runs a smarter ϵ-greedy policy. This is necessary because in large worlds (where the optimal solution to the goal sometimes takes around 1000 steps) that also contain "kill "states (states that give negative reward and end/reset the episode) a simple ϵ-greedy policy can make it almost impossible for the agent to reach the goal without a randomly chosen action landing the agent on a "kill" state and thus ending the episode prematurely.

The way this smart ϵ-greedy works is:

- The agent saves the reward received when the agent arrives at a kill state* (kill reward - kr)
- When the agent needs to choose an action randomly:
  - The agent uses the neural network from Knowledge Abstraction to predict the value of each action (predicted action value - pav)
  - If the $pav < kr + tolerance$, the agent assumes the action will lead the agent to a kill state and that action will be removed from the action list where a random action will be chosen

*(it actually saves the lowest reward ever received when arriving at a kill state, although generally a reward system would have all kill states with the same negative reward since the end result would be the same)

Smart ϵ-greedy thus works like regular ϵ-greedy but allows the agent to use it's learned knowledge to avoid choosing a random action that leads to a kill state.

The $tolerance$ value should take the reward system into consideration. In this work, for the reward system used in the evaluation a $tolerance = 5$ is used.

Notes for the Action step algorithm 2:

1. In $getBestPrimitiveAction()$ when no option is provided the "default option" is used, the "default option" being the default global Q-learning policy and not really an option.

2. $getOptions(state)$ returns the set of options contained by the abstract state where $state$ belongs.

Algorithm 2 Action step

---

procedure GETACTION($state$)
    $availableOptions \leftarrow getOptions(state)$
    if $currentOption \neq None$ **AND** $currentOption$ **in** $availableOptions$ **then**
        return $getBestPrimitiveAction(state, currentOption)$
    end if
    $currentOption \leftarrow None$
    $action \leftarrow getBestAction(state)$
    if $action$ **is a primitive action then**
        return $action$
    end if
    $currentOption \leftarrow action$
    return $getBestPrimitiveAction(state, currentOption)$
end procedure
procedure GETBESTACTION($state$)
    $action, value \leftarrow getBestPrimitiveAction(state)$
    for $option, optionValue$ **in** $getOptions(state)$ **do**
        if $optionValue \geq value$ **then**
            $action, value \leftarrow option, optionValue$
        end if
    end for
    return $action, value$
end procedure
procedure GETBESTPRIMITIVEACTION($state, option$)
    if $option = None$ **then**
        $option \leftarrow "default"$
    end if
    Both lists of action values will be added element wise, into a single list
    $actionValues \leftarrow$
    $getActionValues(state, option) + getActionValuesPrediction(state)$
    $action, value \leftarrow getMostValuedAction(actionValues)$
    return $action, value$
end procedure

Learning step

The learning step is where the agent uses the timestep to perform various value updates, it's where the learning happens.

With simple Q-learning there would be only a simple update but with the world abstraction there are option policies which need to be updated and the timestep also needs to be sent to the knowledge abstraction.

The agent also uses intra-option learning similar to the paper [2], which means the agent will update the global policy and the policy of all necessary options with the same timestep. Example: in a timestep where the agent reaches an option's goal the agent will:

- update the value of performing the action in the global policy;
- update the value of performing the option (option-value) in the global policy;
- update the value of performing the action in the policy of each option available in the current abstract state (also updates the "bridge values" previously discussed).

Notes for the Learning step algorithm 3:

- $getBestActionRealValue()$ is similar to the $getBestAction()$ used in algorithm 2, i.e. the agent chooses the action with the highest value, the set of actions from where to choose being the primitive actions plus the available options. The difference between one another is how the value of the options is calculated. In $getBestAction()$ the values of the options used are directly the ones stored, it is as if the agent could teleport and be at the option's goal in one step regardless of the distance to get there, i.e $y_{v,s} = y_v$ being y_v the value of option y. This is done because it incentivises the choice of options which have a good value regardless of the quality of it's policy, which then results in them improving their quality faster and be of greater use faster. This as been previously discussed as a form of intrinsic motivation related to complexity, which increases the overall learning speed. In $getBestActionRealValue()$ the value used is $y_{v,s} = (y_s max/IR) * y_v$, shown previously in "Option choosing and learning".

3.2.4 Real time savings

For the purpose of this work and the creation of the Abstraction Agent we are only focused on decreasing the amount of steps necessary for the agent to learn how to achieve a goal and use this amount as the metric for the agent's performance. However it is undeniable that updating a simple Q-table is much faster than updating a neural network, in terms of computational time, and the agent performs a number of extra calculations such as spectral clustering and option creation and posterior consideration. As such, there are some methods that were implemented that diminish the overall computational time contrast between a Q-learning agent and the Abstraction Agent.

Those methods are:

- Instead of performing the cutting process* at the end of every episode, use a real time timer that will only start the process if X seconds have passed since the last time the process was performed, X increasing with time.
- Instead of performing a spectral clustering in all abstract states in the cutting process, only perform it on the abstract states that changed enough to consider the possibility of a different outcome to happen. To do this: save the reason why no cuts were performed after a spectral clustering and

Algorithm 3 Learning step

```
intrinsic_reward = 100
procedure LEARN(timestep)
    state, action, next_state, reward, episode_end ← timestep

    if state = next_state then
        reward ← −defaultQValue ∗ 2
    end if
    timestepUpdateWa(timestep)
    sendToKnowledgeAbstraction(timestep)

    Learn the lowest negative reward that ends an episode
    if end = True AND end_neg_reward < reward < 0 then
        end_neg_reward ← reward
    end if

    Option update (option policy update and option value in global policy update):
    abstState ← getAbstState(state)
    for option in getOptions(abstState) do
        if (state, next_state) is a bridge of option then
            Option value in global policy update:
            setOptionPrimitiveActionValue(intrinsic_reward, state, action, option)
            futureBestActionValue ← getBestActionRealValue(next_state)
            setOptionBridgeValue(state, action, bestActionValue, option)

            Option policy update:
            optionValue ← getOptionValue(option)
            newOptionValue ← getUpdatedValue(optionValue,
                futureBestActionValue, lr = 1/getBridgeCount(option))
            setOptionValue(option, newOptionValue)

        else if abstState = getAbstState(next_state) then
            optionPActionValue ← getOptionPrimitiveActionValue(state, action, option)
            futureOptionPActionValue ← getOptionPrimitiveActionValue(next_state, action, option)
            newOptionPActionValue ← getUpdatedValue(optionPActionValue,
                futureOptionPActionValue, reward)
            setOptionPrimitiveActionValue(newOptionPActionValue, state, action, option)

        else
            setOptionPrimitiveActionValue(0, state, action, option)
        end if
    end for

    Global policy (primitive action) update:
    primitiveActionValue ← getQValue(state, action)
    futurePrimitiveActionValue ← getQValue(next_state, action)
    newPrimitiveActionValue ← getUpdatedValue(primitiveActionValue,
        futurePrimitiveActionValue, reward)
    setQValue(newPrimitiveActionValue, state, action)
end procedure
procedure GETUPDATEDVALUE(oldValue, future, discount_factor, learning_rate, reward)
    return oldValue + learning_rate ∗ (reward + discount_factor ∗ future − oldValue)
end procedure
```

the amount of states the abstract state had at that time; when performing the cutting process, check the saved information, if the abstract state was not cut because spectral clustering didn't found any good cuts (because the quality of the cut was insufficient) and the amount of states remains similar, then a good assessment can be made that the spectral clustering will not find a good cut again, and as such the spectral clustering is not performed on the abstract state until it has changed enough. The value used in the implementation is that the abstract state needs to have 1.2 times the number of states it had in the previous spectral clustering in order to test for good cuts again.

- The knowledge abstraction has a buffer that stores its last 10 state value predictions. When asked for a prediction the knowledge abstraction will first try to find the prediction in the buffer before actually getting a prediction from the neural network, and return that prediction if found. This allows the agent to avoid constantly asking a prediction for the same states from the neural network.
- Only updating the neural network from the knowledge abstraction every X steps, instead of after every step, because updating the neural network from the database is very computationally expensive. X increases in proportion with the size of the database, and the database maximum dimension in turn increases in proportion with the state space size (more specifically with the manual abstraction list size Y, see the already discussed "Dataset Creation" part for more information).

*(the cutting process is the $doCuts()$ procedure of Algorithm 1)

Chapter 4

Experimental Results

I now present the parameters and systems used to evaluate the agent and then demonstrate the Abstraction Agent's improvement in relation to a Q-learning agent.

4.1 Systems and parameters settings

4.1.1 Domain

This agent is implemented for performing a task in a world with discrete actions, discrete states and an episodic setting. A task is characterized by a number of specific states, the goals, which the agent needs to achieve in order to perform the task. An episode ends after a task is completed (the goal was reached) or after a certain timeout has been reached. I use a timeout that is much larger than the necessary for the agent to optimally reach the goal because certain maps are large and it's beneficial to let the agent explore, and if possible achieve the goal as fast as possible instead of exploring the same parts of the state space because of too frequent episode resets. As such the timeout generally used is 1.000.000 timesteps or 20 seconds.

The world doesn't need to be deterministic, for optimal performance a learning rate of 1 is used for deterministic worlds and 0.01 or 0.1 for stochastic worlds.

The discount factor can be between]0,1[but it needs to be high enough that the reward for achieving the goal can propagate to the entire state space. The abstraction agent is meant for complex tasks that sometimes require more than 1000 steps to reach the goal, so a discount factor of 0.999 is used, although any higher discount factors like 0.9999 or 0.99999 could also be used, as long as it is lower than 1.

4.1.2 Reward System

The worlds change but the underlying reward system remains the same. 100 reward for reaching a goal, a reward between [-90,-10] for a negative action (generally -50) and -0.01 for any neutral action.

4.1.3 Running System

The running system is the system that mediates the interaction between the agent and the environment. The running system consists of 3 loops inside each other: the run loop, the episode loop and the timestep loop. An approximated/summarized view of the running system can be seen in algorithm 4. Additionally to running the agent, the running system also collects information of the agent's performance, for later analysis and to decide when the agent has stopped learning and the current run should stop.

4.1.4 Termination System

The run should stop when the agent has stopped learning, and the agent has stopped learning when its performance stabilizes. There are primarily 2 values that represent the performance of the agent: timesteps to goal and cumulative episode reward. When learning the agent minimizes timesteps to goal and maximizes cumulative episode reward. For this evaluation all the worlds share the same reward system (shown above) where neutral actions have -0.01 reward, and as such there are no actions with 0 reward which makes it so that maximizing the cumulative episode reward implicitly minimizes the timesteps to goal. With this in consideration, the termination system only uses the cumulative episode reward as a metric for the performance of the agent.

All agents run with an epsilon-greedy policy, which means the metrics for the agent's performance would vary. In order to solve this the agent does an exploitation episode (episode with epsilon = 0) every X episodes. The different stats or metrics used to measure the progress of the agent's performance, for both the termination system and the results/graphs, are obtained from this exploitation episodes which present lower variance.

The termination system has a variable called $nrEpisodesForError$ which represents the interval of episodes considered for judging if an agent has stopped learning. There are always 20 exploitation episodes in this interval. The value generally used is $nrEpisodesForError = 200$. This means the termination system will judge if the agent has stopped learning with basis on the variation of the agent's performance over the last 200 episodes. Since there are always 20 exploitation episodes, this means that every 10 episodes there is an exploitation episode (10=200/20). The termination system checks if the agent has stopped learning at the end of every exploitation episode.

4.1.5 Termination Conditions

The termination system collects the information of the agent's performance over $nrEpisodesForError$ episodes and then has to judge if the agent has stopped learning. In order to perform that judgement the system has **termination conditions** and performs calculations about the information to see if the conditions are reached. The termination conditions are:

- The agent reached the goal in all the considered episodes - this is important because the agent could temporarily stabilize without having reached the goal

Algorithm 4 Running system

Create world
$world \leftarrow getWorld()$

Create agent
$agent \leftarrow getAgent()$

Create database
$database \leftarrow getDatabase()$

$number_of_runs \leftarrow X$
Start run's loop
for $number_of_runs$ **do**
    $resetAgent(agent)$

    Start episode loop
    while agent is still learning and improving **do**
        Reset world
        $episodeReset(world)$
        $end_episode \leftarrow False$

        Start timestep loop
        while !$end_episode$ **do**
            Get the world state
            $state \leftarrow getState(world)$

            Ask agent for an action
            $action \leftarrow getAction(agent, state)$

            Perform action in the world
            $next_state, reward, end_episode \leftarrow doAction(world, action)$

            Give feedback for the agent to learn
            $timestep \leftarrow (state, action, next_state, reward, end_episode)$
            $learn(agent, timestep)$

            Reset the timestep ($state \leftarrow next_state$)
            $timestepReset(world)$

            $end_episode \leftarrow checkTimeout()$

            Save relevant information
            (such as $timestep$ or cumulative rewards)
            $saveStats(database, information)$
        end while

        Let the agent perform end of episode procedures
        (such as the $doCuts()$ in the world abstraction)
        $episodeReset(agent)$

        Check if the agent has stopped learning and improving
        agent has stopped learning $\leftarrow terminationSystem(database)$
        if agent has stopped learning **then** $break$
    end while

    Save information about the run
end for

All runs completed!
Compile and show run's information for evaluation

- The agent's performance has stabilized

In order to judge if the agent's performance has stabilized, i.e. if the "cumulative episode reward" results have stabilized, the system:

- Saves the agent's best (cumulative episode reward) result in its entire history
- Calculates the average result (of the exploitation episode) in the last $nrEpisodesForError$ episodes
- Calculates the maximum error allowed, which increases with the number of episodes and reaches its maximum value of 1 when the agent has performed $10 * nrEpisodesForError$

The system considers the agent's performance has stabilized when $BestResult - Avg \leq MaxError$, i.e. when the average result is close enough to the best result for the system to consider that the agent consistently achieves a result close to the best result and has thus stabilized. If the agent still hadn't finished learning and hadn't stabilized, the best result would rise and be distant from the average, and the system wouldn't consider the agent as having stopped learning.

However the system only considers the last $nrEpisodesForError$ episodes, so it's possible for the agent to converge fast enough for the system to consider the agent has stopped learning although the agent's best result is a sub-optimal result and could still be improved. This problem can be mitigated by increasing the value of $nrEpisodesForError$ as necessary.

4.1.6 Performance evaluation graphs and parameters

Most graphs to demonstrate and compare the performance of different agents are in the format:

- reward - cumulative rewards the agent receives in a episode (y-axis);
- timesteps - cumulative timesteps of the agent over the entire run, or, episodes - the episode count (x-axis).

Parameters used in the worlds unless otherwise specified:

- epsilon = 0.1
- discount factor = 0.999
- learning rate = 1
- $nrEpisodesForError$ = 200
- Graphs are an average of 10 runs
- Timeout = 1.000.000 timesteps or 20 seconds.

4.1.7 Worlds and maps

3 worlds are used for evaluation: a toy world with different maps mostly composed of rooms, a playroom world and the minecraft world.

The toy world is where most of the evaluation is performed. This toy world has the agent travel through a 2 dimensional world and try to reach a goal position in the world. The states are s = (position x, position y, orientation, spaceInFront), e.g. s = (6, 56, up, "Lava") means the agent is in the coordinates (6,56) in the world, is facing up (or north) and has a lava pit in front of him. The action space is composed of only 3 actions: move forward, turn right and turn left. A position (x,y) in the map can be either: an empty space where the agent can be, a wall or obstacle, or a lava pit. The agent's spawn/source position and goal position are special empty places with those properties. Each position in the map has a correspondent "spaceInFront" feature value of: empty space - " ", wall - "X", lava - "L", source - "S" and goal - "G". There exists many maps with different layouts to test different principles but most of the maps are composed of rooms.

The playroom world is like the one in the work [2].

The minecraft world is similar to the toy world, except it is in 3 dimensions and it has 26 more features in the state because the "spaceInFront" feature is replaced with the information of the type of block that exists in the 3x3x3 cube space that surrounds the agent.

4.2 Toy world evaluation

Lets now explore and see how the world abstraction and the knowledge abstraction affect the agent's performance. As a baseline a simple Q-learning agent is used because the abstraction agent's core is also Q-learning. The abstract agent's world and knowledge abstraction can be individually deactivated, and if both are deactivated then only the Q-learning core remains, thus making the abstraction agent equal to a simple q-learning agent. This is demonstrated in Fig.4.2 by comparing the learning evolution of the Q-learning agent versus the Abstraction Agent with both components deactivated. The simplest map, 4-rooms, is used. This map can be seen in Fig. 4.1 below.

In order to properly evaluate both components, world and knowledge abstraction, they will be separately evaluated and then a joint evaluation with both components active is made.

4.2.1 World abstraction

As explained previously, the world abstraction abstracts the world into abstract states separated by bottleneck states, and options are policies on how to go from the abstract state they belong into a neighboring abstract state through the bottleneck states that separate them. A visual representation of this can be seen in Fig.3.1.

As such, optimal maps to show the world abstraction's strength and usefulness are maps like the 4 rooms map, where the map is constituted of rooms and the agent needs to go through several bottlenecks to achieve the goal. So it is expected that in such maps the abstraction agent is able to learn

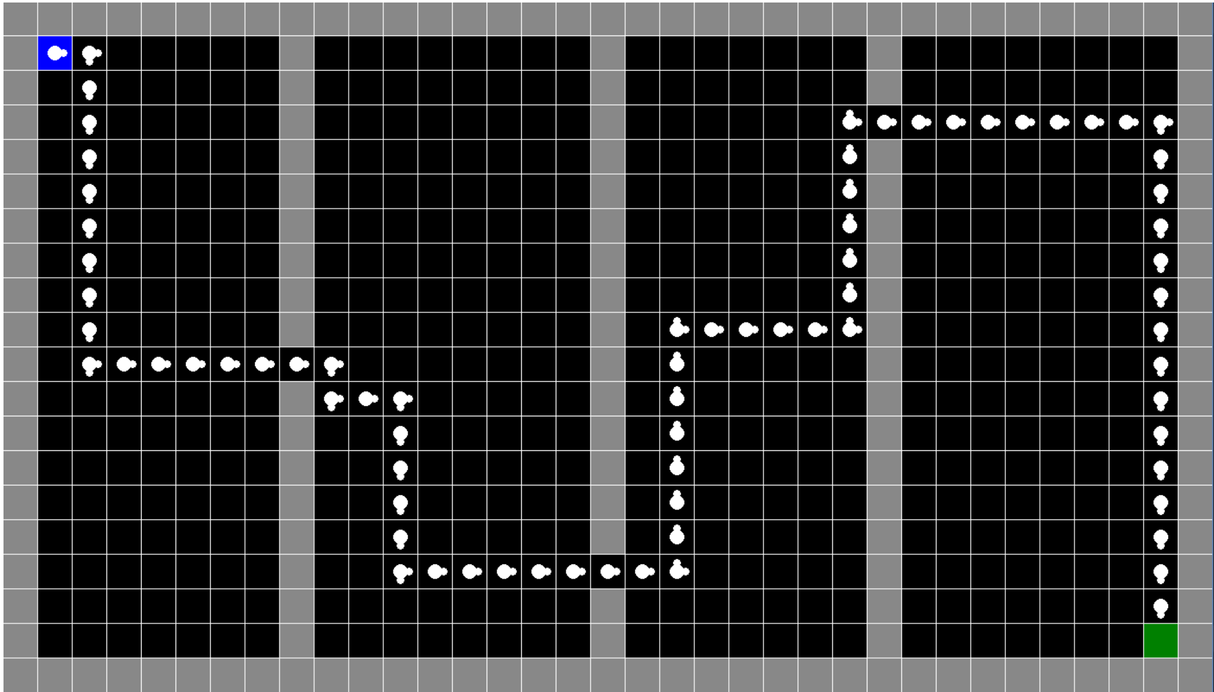


Figure 4.1: 4-rooms map, the simplest map of the toy world. The black squares are empty space and the grey squares are wall or obstacles. The blue square is the source and the green square is the goal. The image also shows an agent's learned trajectory from source to goal.

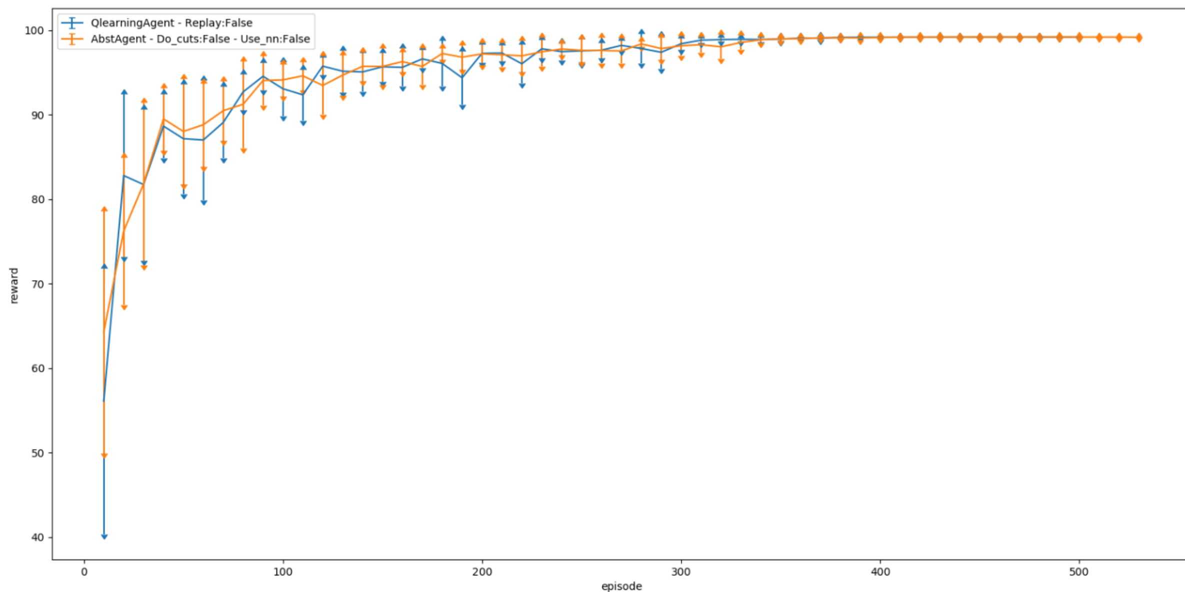


Figure 4.2: Graph comparing performance evolution between the Q-learning agent and the Abstraction Agent in the 4-rooms map. Graph is the cumulative rewards the agent receives in a episode over said episodes. As we can observe the cumulative rewards received in a episode increase as the agent learns and both agents learning evolution is similar. It can be seen that they stabilize for around 200 episodes before stopping because the $nrEpisodesForError = 200$.

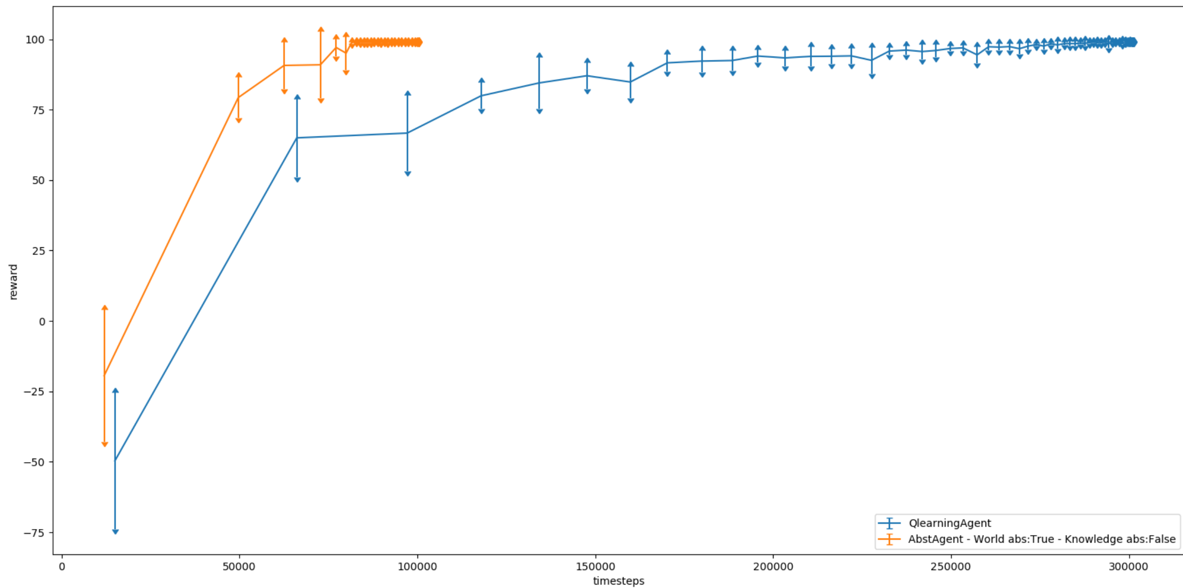


Figure 4.3: Comparison between Q-learning agent and the Abstraction Agent with the world abstraction active. As expected the world abstraction provides faster learning.

faster how to reach the goal than the Q-learning agent, and this is demonstrated in Fig.4.3.

In a map similar to the 4-rooms map but bigger we can observe the improvement even better. The same comparison in the bigger 24-rooms map can be seen in the Fig.4.4. A visual representation of the abstraction done in the world can be seen in Fig.4.5.

As it can be seen in the Fig.4.5 the abstraction didn't actually create the intuitive abstraction of 1 abstract state for every room but actually further divided each room into 2 abstract states. This happens because although the cut that separates the room into 2 isn't nearly as good as the cut that separates 2 rooms, it's still sufficiently good. To be more precise, the cut that separates 2 rooms has a cut quality of around $125000(= 500 * 500/2)$ while the cut that separates the room into 2 has a cut quality of around $4000(\simeq 250 * 250/14)$ but because the quality threshold is 1000 the cut is still good enough. The quality threshold's effect on the abstraction and the performance of the agent can be seen in Figs. 4.6, 4.7.

The world abstraction and its options allow the agent to learn faster by improving its exploration and allowing a faster "policy to goal" convergence. In a map like the 4-rooms, the abstraction agent will eventually have the ability to go from start to goal in fewer actions/options, e.g. 4 actions/options, when in fact the start and goal can be 60+ primitive actions away from one another. By having an option that allows the agent to go to a neighboring abstract state the agent has always available an action which can take it multiple steps away from where it is, rapidly allowing it to explore higher valuable localizations (which also contain other options), and because an option value is also affected by the value of other available options, then the value of an option represents the value of its abstract state and the discounted value of its neighboring abstract states, similar to how a state is valued in Q-learning. This relation between options makes them similar to actions but that just take more than 1 step to end. These improvements can be seen in the Fig.4.8 where its possible to visually see the impact of the options on the exploration of the agent.

In fact, the abstraction agent begins to learn how to reach the goal before it has ever reached the

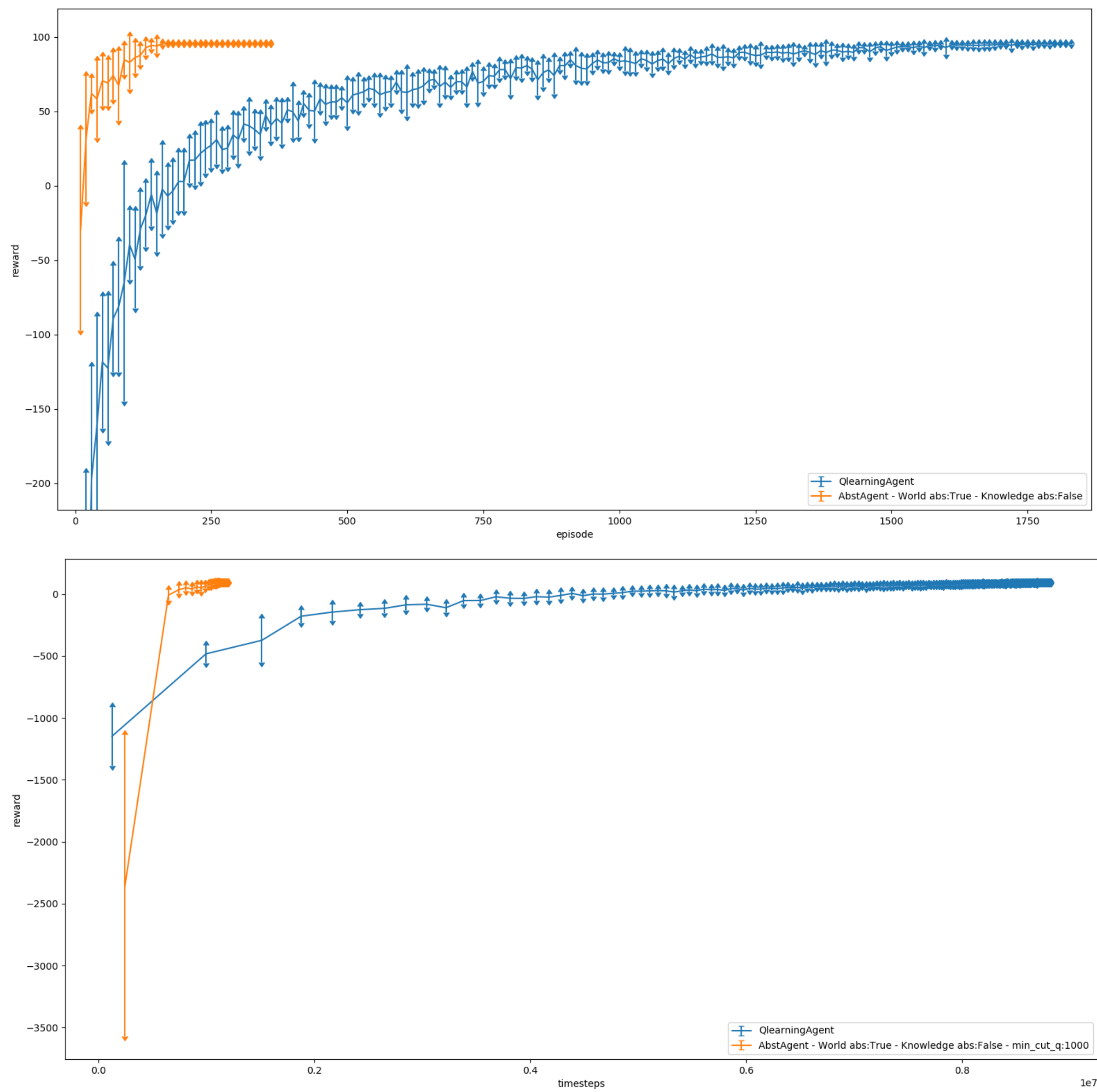


Figure 4.4: Comparison between the Q-learning agent and the Abstraction Agent with world abstraction active in the 24-rooms map.

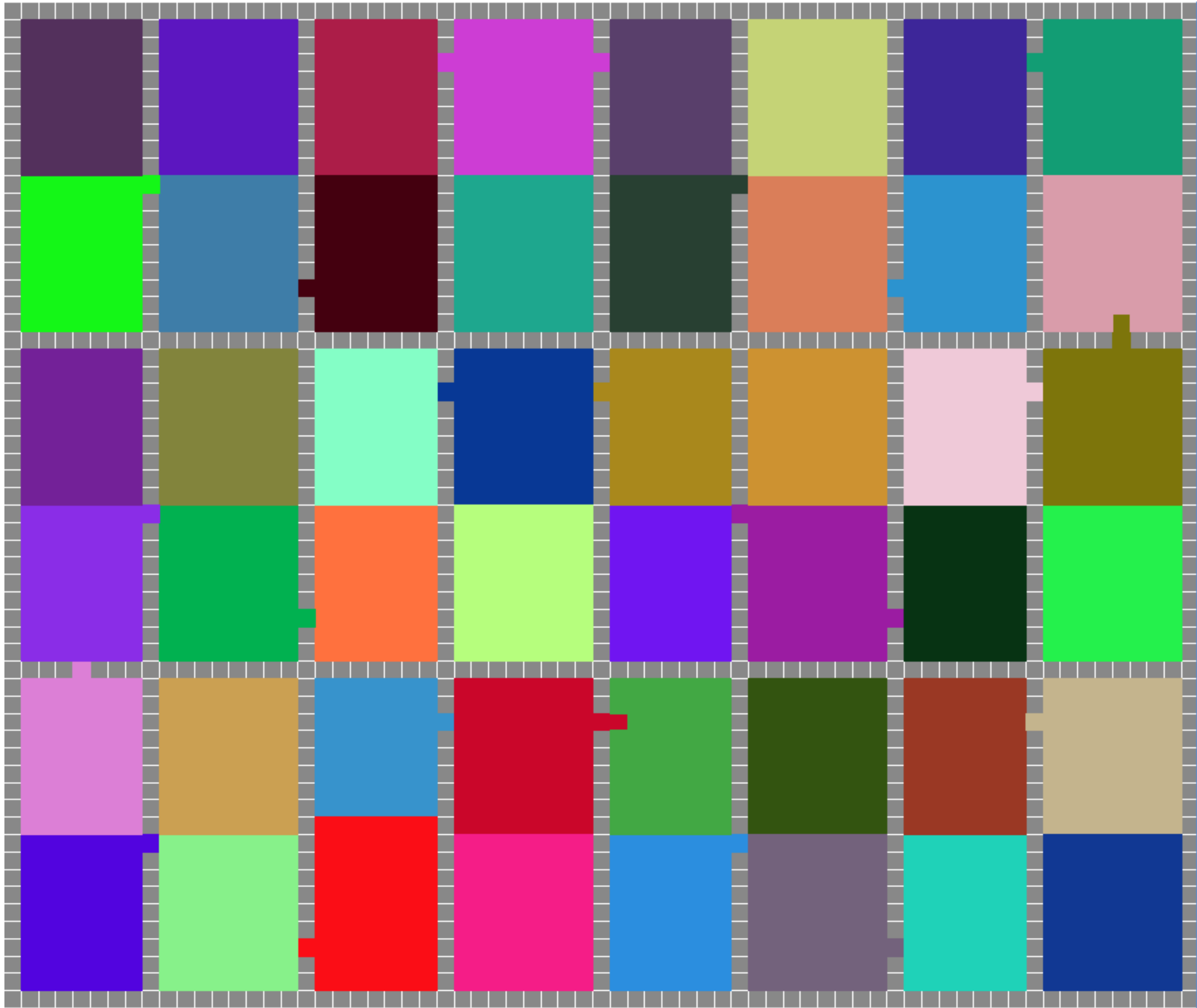


Figure 4.5: Visual representation of the abstraction done over the world. Minimum cut quality = 1000.

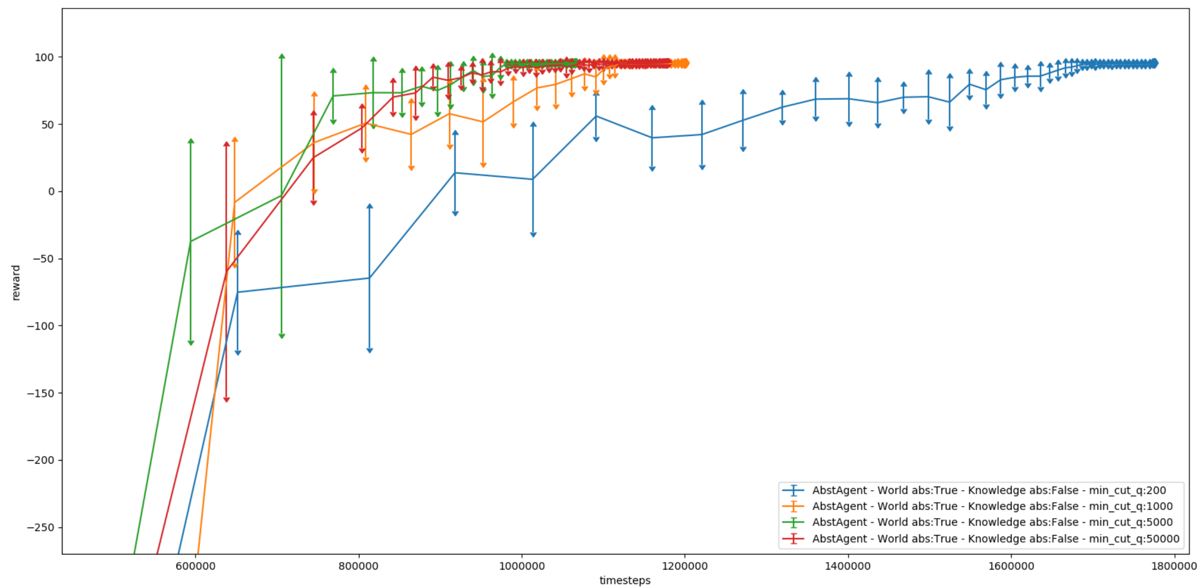


Figure 4.6: Comparison between abstraction agent's with different minimum cut quality. Different quality thresholds above 1000 have a similar performance because they all create meaningful abstractions even if in different quantities. In contrast, the abstraction made with a quality threshold of 200 accepts such low quality cuts that it over-abstracts the state-space, increasing the quantity of abstract states while decreasing the average quality of them, thus creating difficulties in learning.

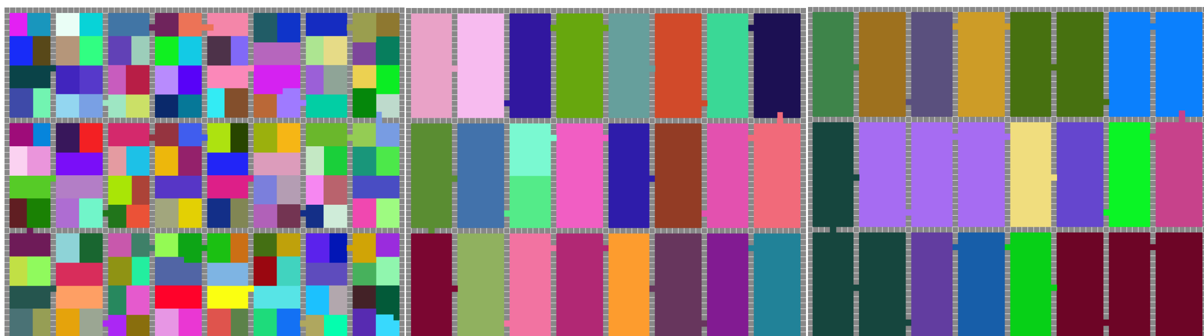


Figure 4.7: Abstraction comparison with minimum cut quality = 200, 5000 and 50000, respectively

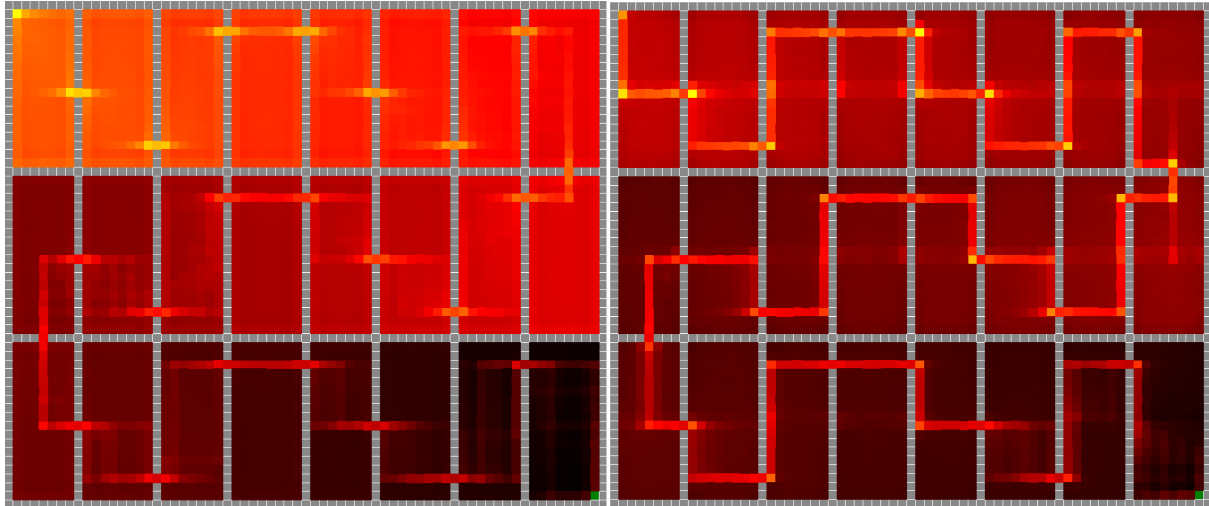


Figure 4.8: Comparison of the state frequency (Q-learning is left and Abstraction Agent is right). The brighter a position in the map the more it was visited. It's possible to observe how the Q-learning agent over-explores the states closer to the start while the Abstraction Agent has a more uniform state frequency. This graph is of only 1 run in of each agent but the average results are similar. Q-learning: highest state frequency was around 70000 with lowest being around 700 and the average around 27000; Abstraction Agent: highest state frequency was around 14000 with lowest being around 400 and the average around 3700.

goal, because it learns these option's policies independently of extrinsic rewards and some of these policies are sub-policies of the global "policy to goal".

Another benefit of the world abstraction is that the agent is learning options/sub-policies independently of extrinsic rewards and it's updating and improving them all at the same time (in the same episode). In a way, the abstraction agent instead of trying to learn the whole global policy is learning multiple sub-policies at the same time and then joining them.

Example: there is a map where the source and goal are 100 steps apart. An optimal policy would thus be able to lead the agent to the goal in 100 steps. In the best case scenario the Q-learning agent needs at least 100 episodes to learn this optimal policy, one episode for each step value it needs to propagate backwards (from the goal to the source), and each episode needs at least 100 steps to reach the goal and restart the episode. This means that Q-learning needs at least 10000 steps (100*100) to learn the optimal policy. The world abstraction component allows the Abstraction Agent to learn the same optimal policy in less steps.

Lets imagine the world abstraction abstracts the world into 10 different abstract states. Then there are 10 options whose combined optimal sub-policies are similar to the global optimal policy. Because the agent learns these options independently of extrinsic rewards, for each episode it is able to propagate backwards 1 step value for each sub-policy/option, which means in each episode the agent is now propagating backwards 10 step values. The optimal policy which needed at least 100 episodes to be learnt, now has been divided into 10 sub-policies, each with 10 (=100/10) steps to its intrinsic goal, and as such each only needing at least 10 episodes to be learn. With this abstraction, the abstraction agent learns to combine the options into the optimal global policy in 10 episodes (1 episode for each option value to propagate backwards) with each still needing at least 100 steps to end, which means

that the abstraction agent needs at least 2000 steps (20*100) to learn the optimal policy (1000 to learn the sub-policies in ”parallel” and another 1000 to combine them).

It’s as if a road which needed to be traveled from start to finish was divided into 10 parts or sub-roads which are then traveled at the same time and later joined to form the whole road. Experience replay increases the learning speed because it increases the amount of values that are propagated backwards in a episode. The impact of experience replay in both agents is similar because the increase in the amount of values that are propagated backwards is equal.

The improvement in the exploration of the agent and the ”simultaneous solving of sub-problems” benefit of the world abstraction allows it to actually improve the learning speed of the abstraction agent even when the map has no rooms or no apparent good abstractions. What this means is that the improvement provided by the world abstraction is affected by the map, but even in maps with no apparent good abstractions the state-space is still abstracted and the improvement in the learning speed still exists. This can be seen in Fig.4.9 where the map is a completely open room and the Fig.4.10 where the map is a maze. In the open room there is no bottleneck states and in the maze one can look as if all states are bottleneck states or no state is a bottleneck state. In both cases the world abstraction still abstracts the state-space and still improves the learning speed of the abstraction agent.

Using the open room map it’s even easier to observe the impact of the minimum cut quality in the abstraction and learning, Figs.4.11,4.12.

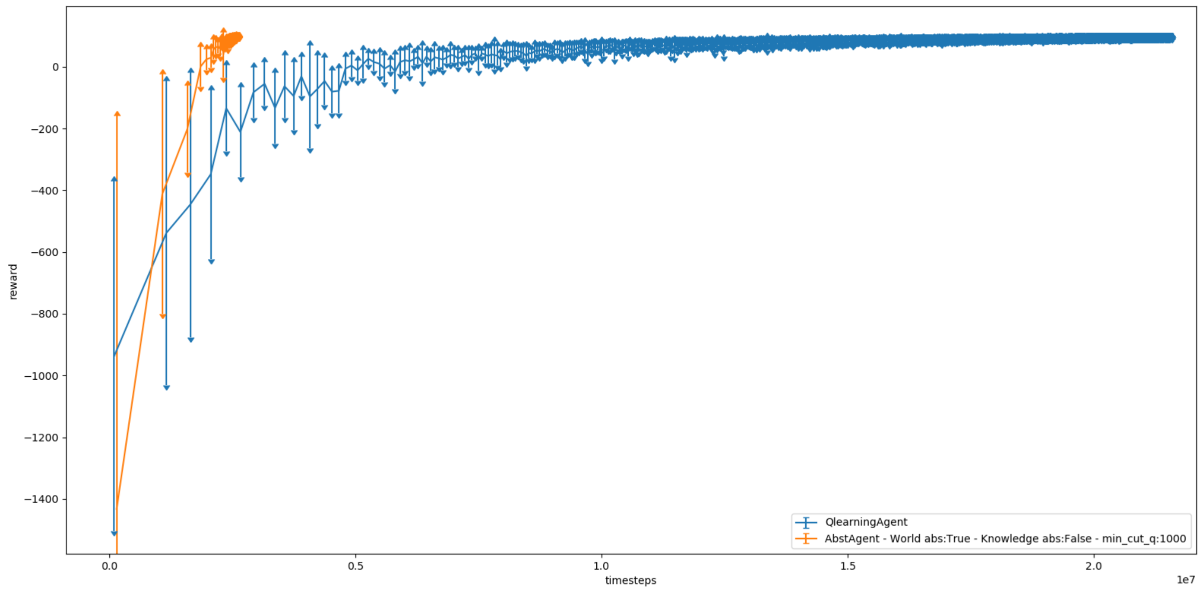
4.2.2 Knowledge Abstraction

The knowledge abstraction tries to find extra meaning in the features of a state from the agent’s observations. More concretely the knowledge abstraction (or its neural network) tries to predict the reward the agent will receive for each action using the state as input. In order to evaluate the knowledge abstraction lava pits were added to the 24-rooms map. The map has 2 versions, one where the lava kills (a negative reward is given and the episode immediately ends) and one where it doesn’t kill (just a negative reward is given). The agent is evaluated in both and the results can be seen in Fig.4.13. The input states are s = (position x, position y, orientation, spaceInFront).

The knowledge abstraction is able to abstract the states, learn and then predict that when in a state with the feature spaceInFront = ”L” the action ”MoveForward” gives a negative reward. By doing this it is able to generalize this knowledge to all states that have the feature spaceInFront = ”L”, even in states it has never seen or visited. The Q-learning agent has to first fall into a lava pit to learn that in that specific state the action ”MoveForward” is bad. As such the Q-learning agent will fall into every lava pit it has never seen, while the abstraction agent after learning that generalized knowledge will not fall into lava pits anymore. This allows the abstraction agent to learn faster as can be seen in Fig.4.13.

4.2.3 World and Knowledge Abstraction

A joint evaluation of both components working together will now be presented. The evaluation is made not only in the lava-24-rooms but also in the lava-96-rooms, which is the biggest map with lava created



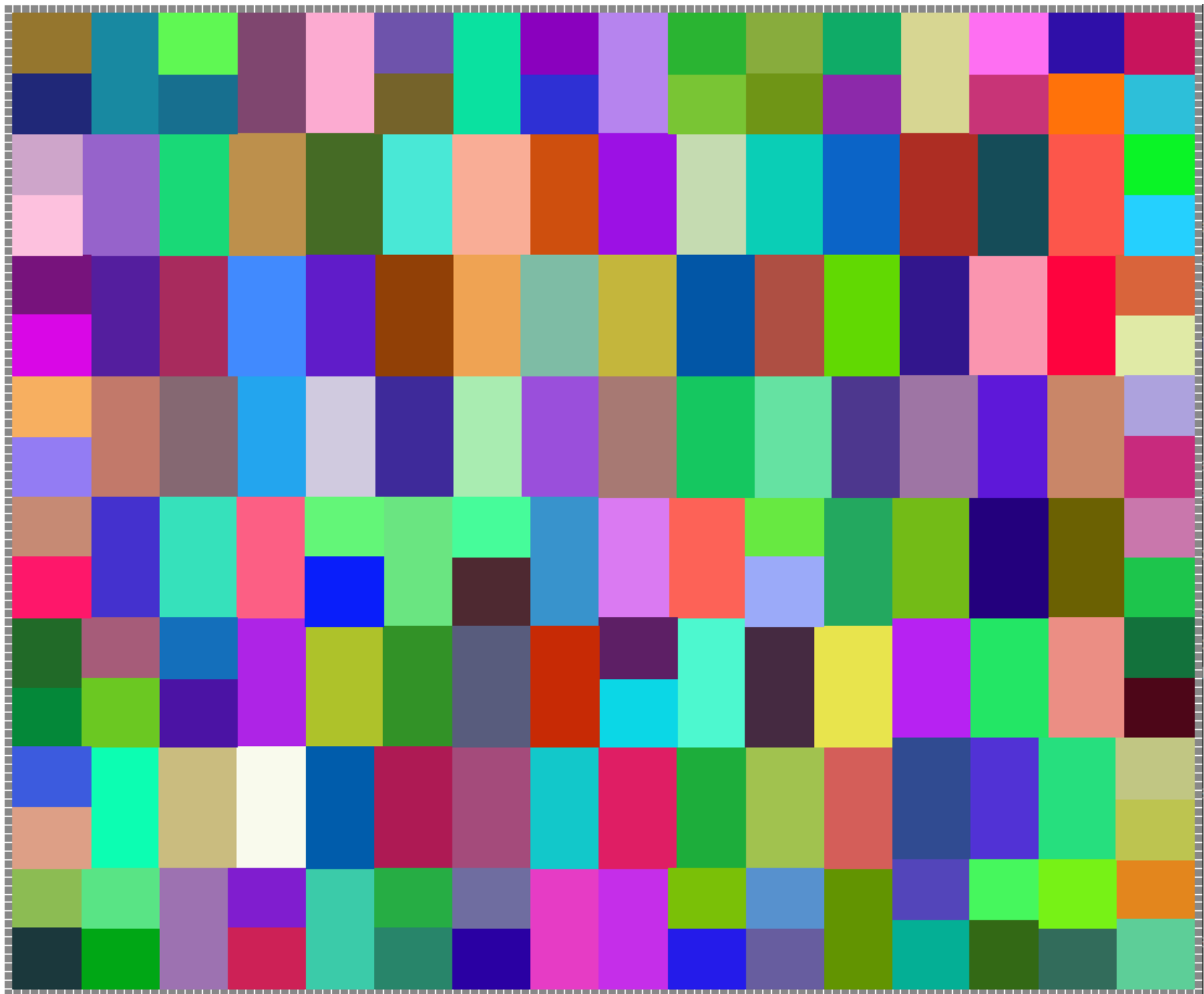
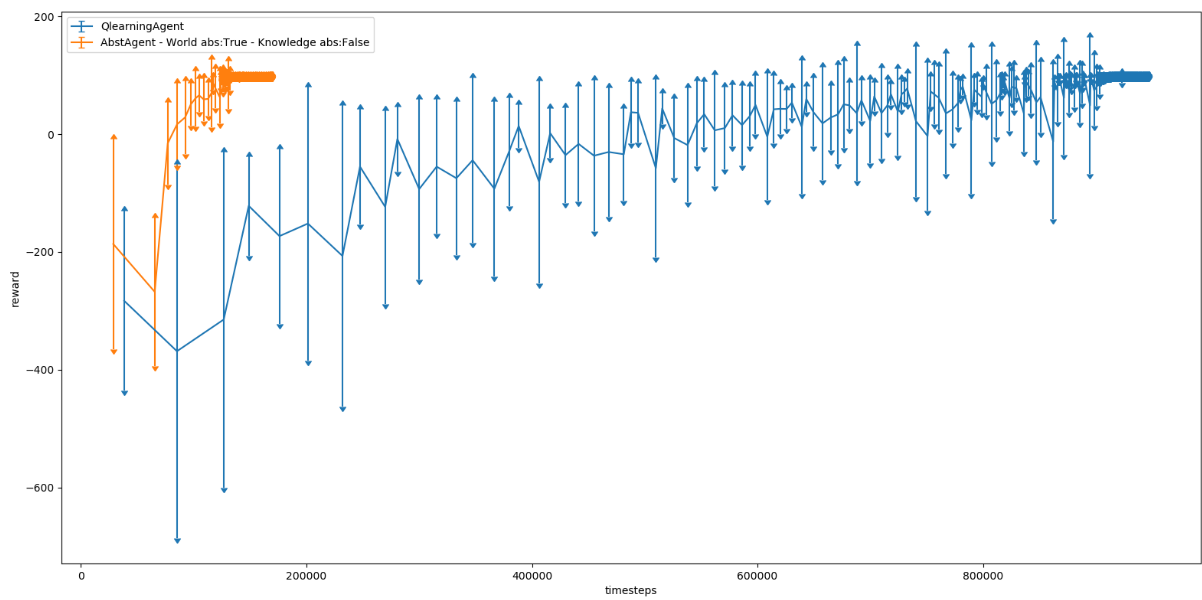


Figure 4.9: Comparison between the Q-learning agent and the Abstraction Agent in the open room map. Abstraction made is also shown.



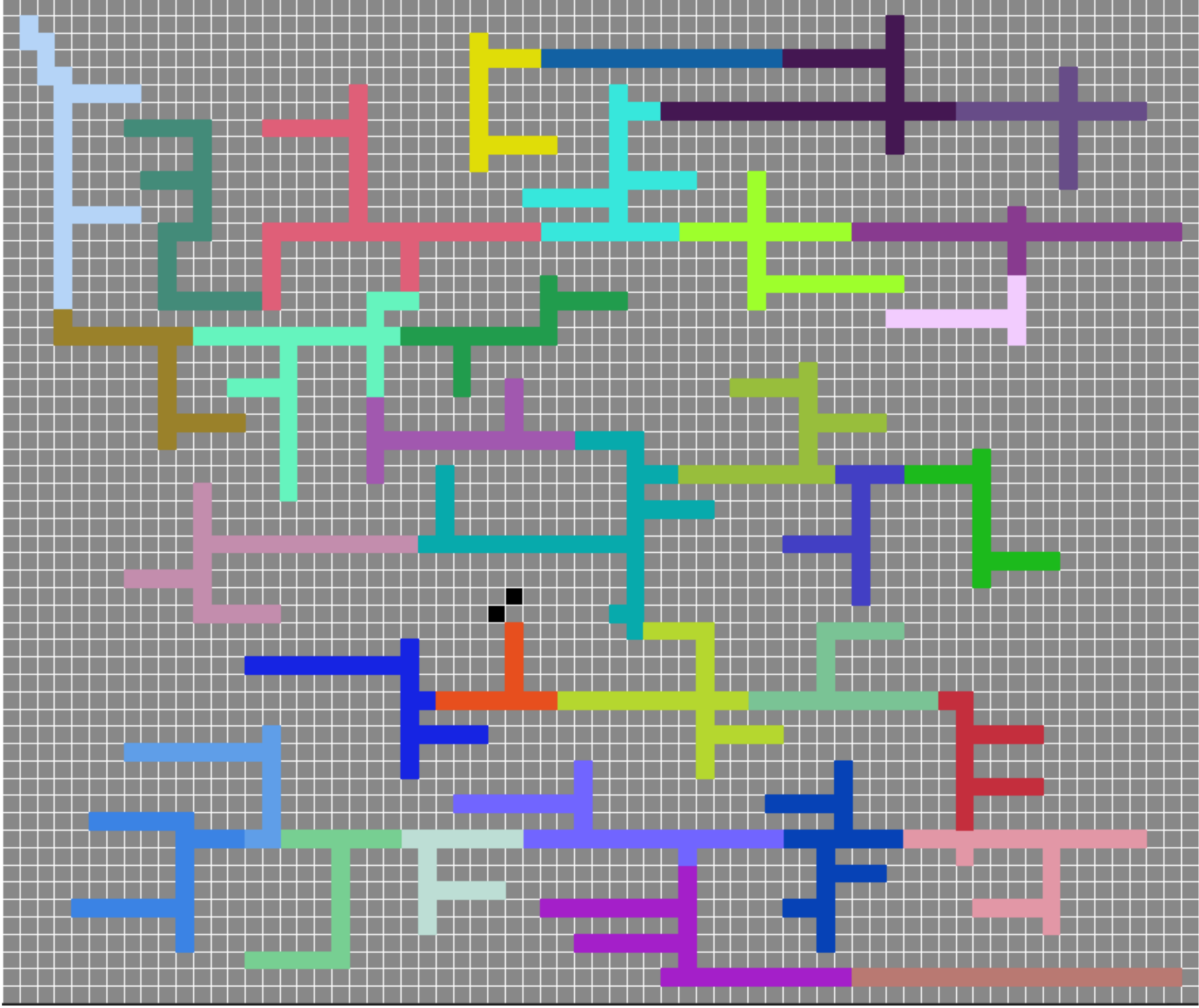


Figure 4.10: Comparison between the Q-learning agent and the Abstraction Agent in the maze map. Abstraction made in the shown.

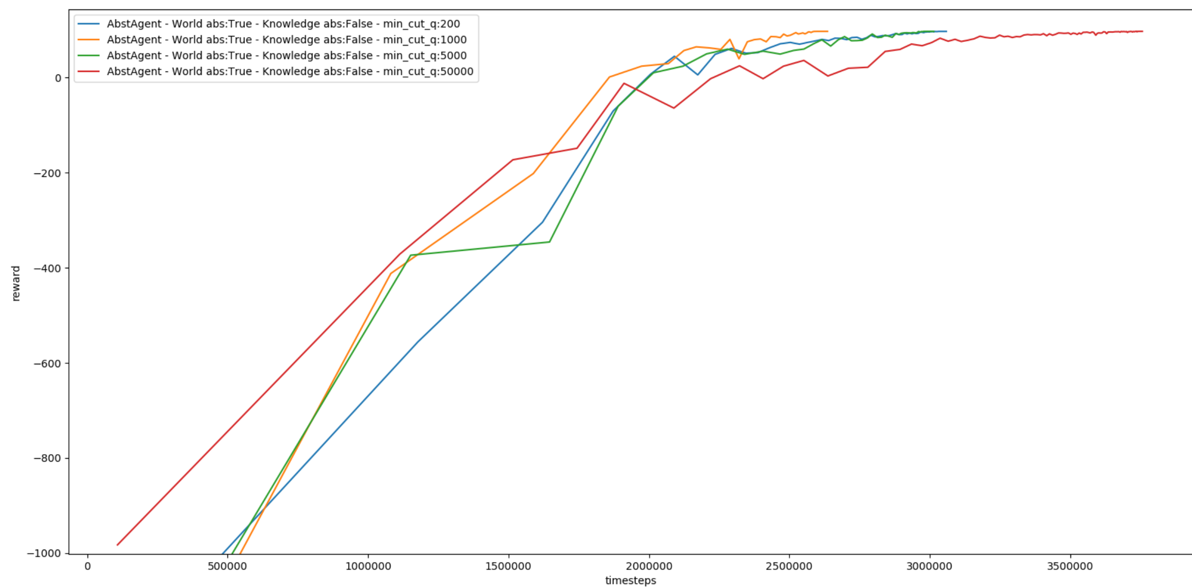


Figure 4.11: Comparison between Abstraction Agent's with different minimum cut quality in the open-room map. Variance not shown so it's easier to compare the final stages.

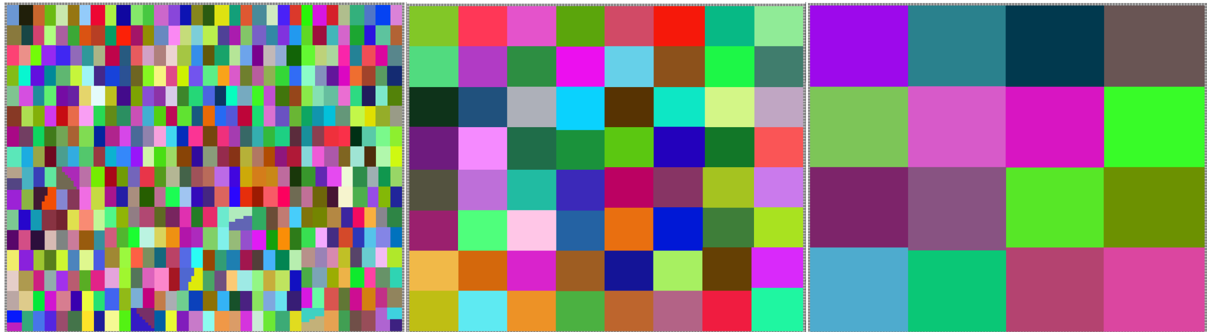


Figure 4.12: Abstraction comparison with minimum cut quality = 200, 5000 and 50000, respectively

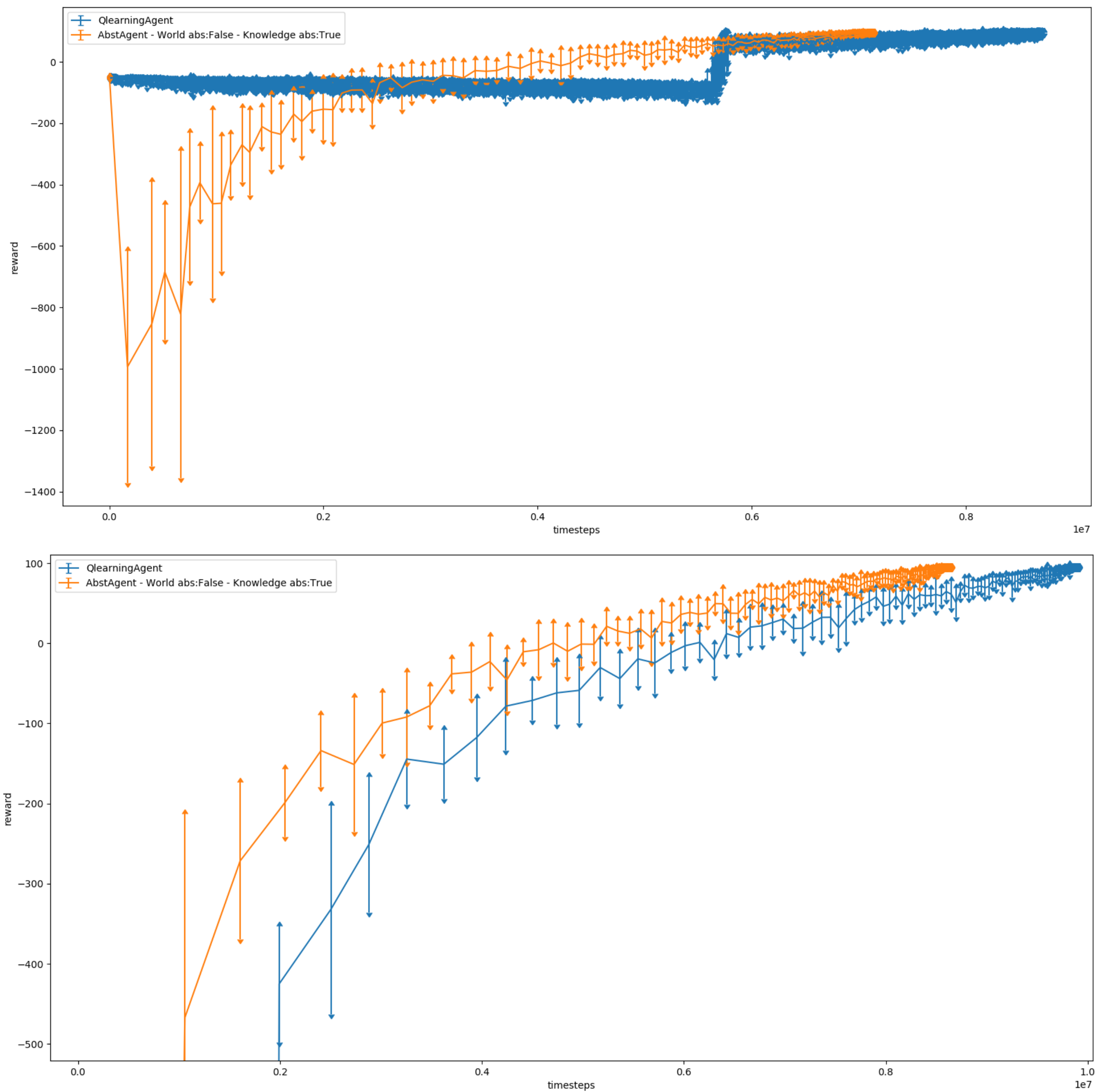


Figure 4.13: Comparison between the Q-learning agent and the Abstraction Agent with only the knowledge abstraction active, in the Lava-24-rooms. In the first picture the lava kills and in the second it doesn't.

for this evaluation with 4 times the size of the lava-24-rooms. As with the previous map with lava there is also are 2 versions of this map: one where the lava kills and one where the lava doesn't kill. For this joint evaluation the timeout used will still be 1.000.000 timesteps but the seconds are increased to 60 seconds (from 20 seconds). The evaluation can be seen in Figs.4.14,4.15.

4.2.4 World Abstraction flaw

There can be seen some instabilities at the end (cumulative reward) result of the policy from the Abstraction Agent in the maps where the lava doesn't "kill". These instabilities were not expected when performing this evaluation and they were a warning that allowed for a flaw to be discovered upon further analysis. These instabilities are created by the option's policy grading.

Option's policy grading: as said previously, "the objective of an option's policy is to lead the agent to that option's goal while maximizing reward", in this case maximizing reward means in the fewer steps possible and avoiding lava pits. The option's policy has in consideration or is dependent on the immediate extrinsic rewards (like the negative reward given by a lava pit) and the goal's intrinsic reward. More concretely, in an option's policy, all state-action pairs that lead to a goal (of that option) have a value of "intrinsic reward" + "possible extrinsic reward", which means that realistically almost all option's goals (or state-action pairs that lead to an option's goal) have the same value in the option's policy although having different extrinsic values or Q-values (different "values" in relation to completing the task or reaching the goal). This was also discussed when explaining the need for the option's "bridge values", which served for the agent to choose between different bridges (or state-action pairs that lead to an option's goal). However, this "bridge values" solution is insufficient because the Abstraction Agent is only able to perform this consideration when he is already at a border state choosing the best bridge (or state-action pair), and all the considered bridges could be worse than another distant bridge.

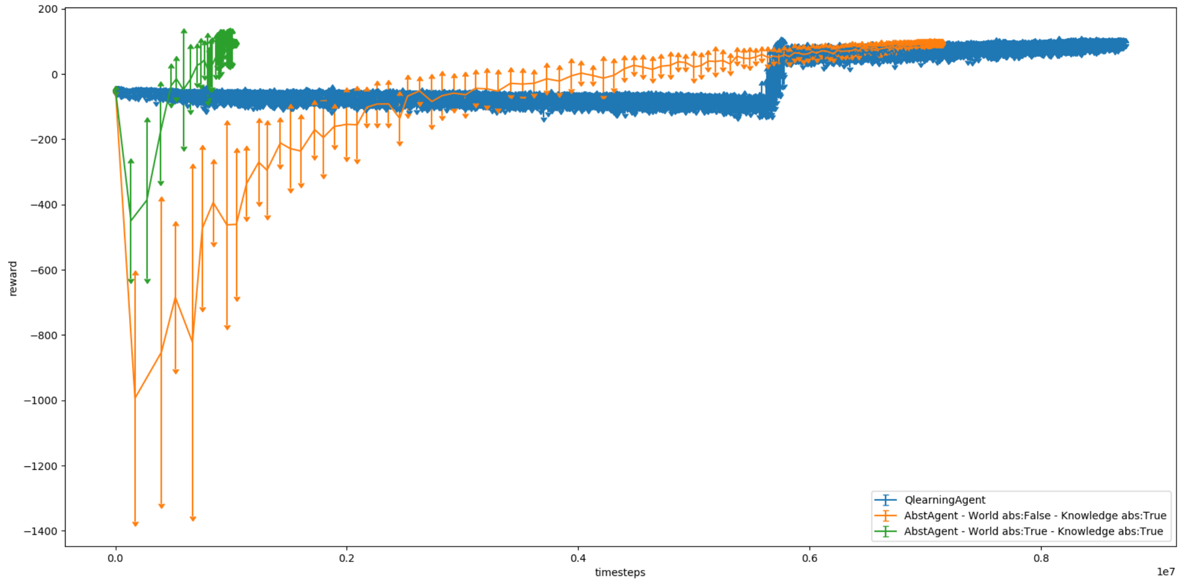
Basically, the objective of an option's policy will be to lead the agent to **any goal** of that option while maximizing intrinsic reward **regardless** of the extrinsic value of that goal. This usually results in leading the agent to the closest goal of that option even if that goal is worse (in the global spectrum) in relation to others.

This "unprecision" in choosing where the option should lead the agent to, wasn't a problem when there was no lava pits, because the agent couldn't receive a significant negative reward as a result of this "unprecision". However when there are lava pits, then this "unprecision" causes the agent to go to "dead ends" where it needs to go through a lava pit to reach the goal. In Figs.4.164.17 is presented a simple map where the flaw in the option's policy grading can be more easily seen and understood.

Possible solutions:

A simple solution would be learning an option for each goal but this would greatly increase the number of options being created and updated which would greatly increase the time and memory required.

Another solution would be to allow a state-action pair that leads to a goal to have a value of "intrinsic reward" + "possible extrinsic reward" + "Q-value of that state-action pair", this Q-value already existing in the global Q-learning policy. This would allow the goals to have different values that depend on the



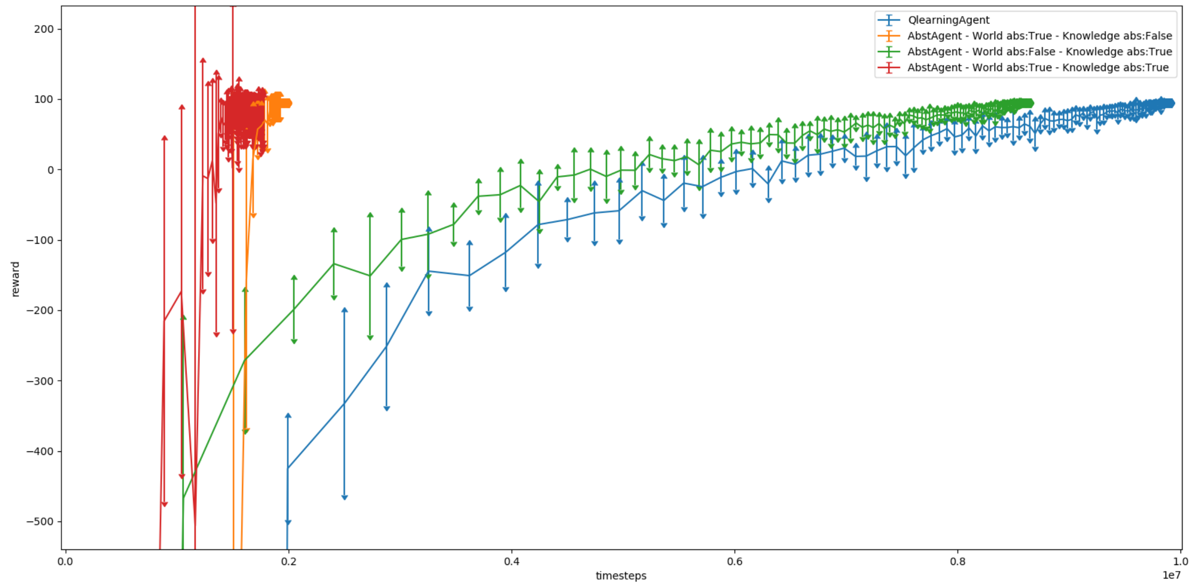


Figure 4.14: Comparison between the Q-learning agent and the Abstraction Agent with both components active, in the Lava-24-rooms. In the first picture the lava kills and in the second it doesn't. In the 1st image, the Abstraction Agent with only the world abstraction active isn't present because the agent wasn't able to solve the map due to the flaw explained bellow.

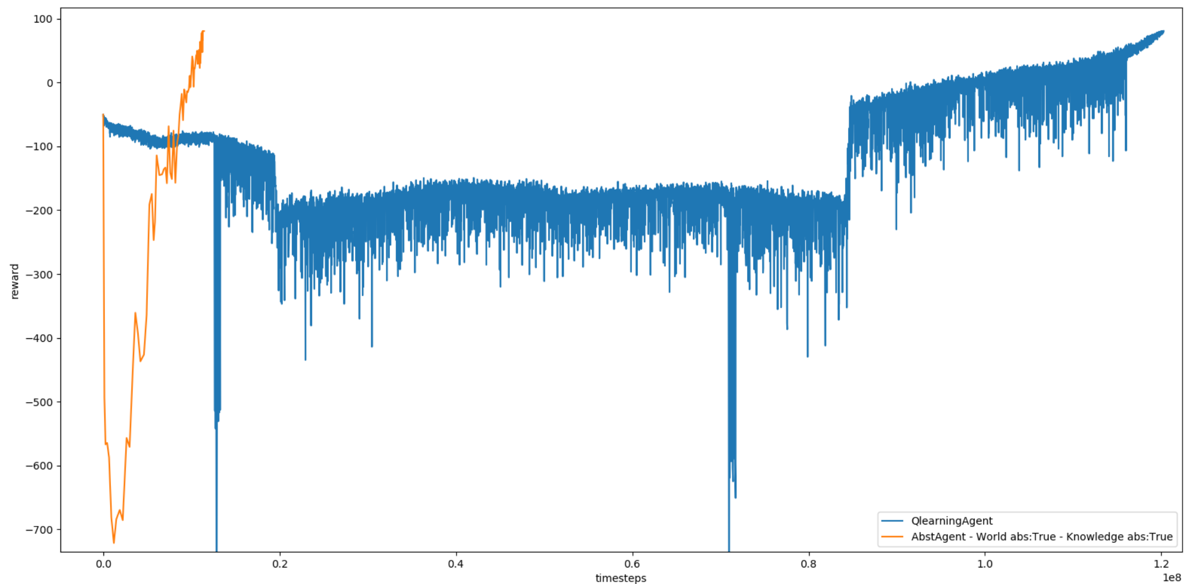


Figure 4.15: Comparison between the Q-learning agent and the Abstraction Agent in the Lava-96-rooms with lava killing, not showing variance. The Abstraction Agent isn't able to solve the Lava-96-rooms with lava that doesn't kill because of the flaw explained bellow.

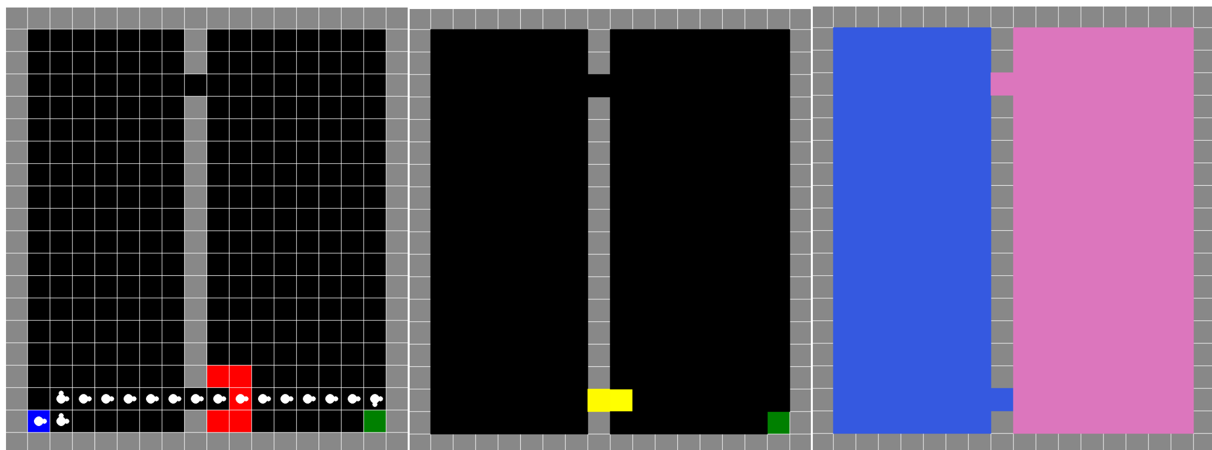


Figure 4.16: Map to demonstrate flaw. A minimum cut quality of 25000 is used in order to generate the necessary abstraction to exemplify the flaw. It's possible to see in the 2nd image how the agent goes back and forth between those 2 map positions. This happens because as we can see in the 3rd image, the 1st abstract state is separated at the "door" from the 2nd one, so right after the option leads the agent from the 1st to the 2nd abstract state the agent is confronted with a "dead end" of lava pits, having no choice but to retreat. However after it retreats, that option is still the "best" choice and it still leads it there, because the option's objective is to lead the agent from the 1st to the 2nd abstract state while maximizing reward and that "door" is closer, so that is the policy that maximizes the option's intrinsic reward. With this abstraction the agent is almost never successful in learning the best global policy.

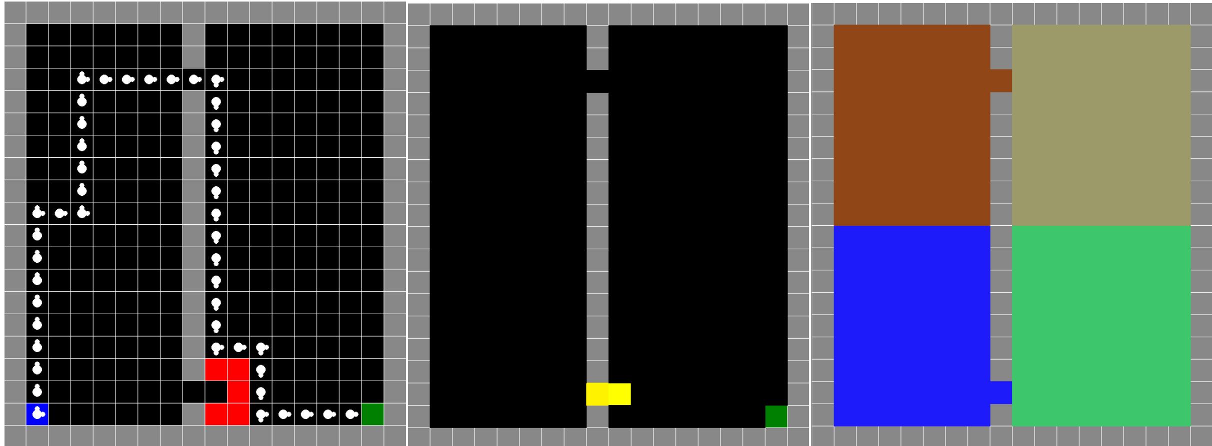


Figure 4.17: Map to demonstrate flaw. The default minimum cut quality of 1000 is used. It's still possible to see in the 2nd image how the agent goes back and forth between those 2 map positions. However, in this case the Abstraction Agent is able to learn a good policy because the 1st abstract state (bottom left) no longer has a single option with 2 separate "doors" as goals but has actually 2 options, one to go to the upper left abstract state and another to go to the bottom right abstract state, and as the agent continuously goes back and forth between those 2 abstract state (the 2 map positions from the 2nd image are in different abstract states), the value of the option to go to the bottom right abstract state eventually decreases enough for the agent to consider and choose the option to go to the upper left abstract state.

extrinsic value or Q-value of that state. However this has already been tested before the "bridge values" solution and because the Q-value is not fixed, but a learned value that is constantly updated, the option's policy stops being independent of extrinsic rewards and also starts being constantly updated, the end result being that the Abstraction Agent's performance becomes equal to that of the Q-learning agent.

A good solution would require an intrinsic value that is somewhat fixed but still dependent on the extrinsic value of the goals.

This will left for future work, however an approximate solution might be to separate the goals into groups, grouping goals that have similar extrinsic value. Goals in the same group would have the same fixed intrinsic value. Groups whose goals have a higher extrinsic value would wield a higher fixed intrinsic value. Using the example map and abstraction in Fig.4.16: upon seeing goals with high variance of extrinsic reward, the goals of the option would be separated into 2 groups, goals in the best group would give a fixed intrinsic reward of 100 and the goals in the other group would give a fixed intrinsic reward of 90 (100-10). The option's policy would still be somewhat independent of rewards and fixed but it would have a higher consideration of the extrinsic reward of the goals, and allow the agent to not simply go to a neighboring abstract state but go to a neighboring abstract state through the best border. This solution would replace the "bridge values" flawed solution.

4.2.5 Experience Replay

Experience replay is a common method used in reinforcement learning that can allow faster backwards value propagation and therefore learning. It usually consists on the agent having a buffer memory, a list of observations or timesteps, which upon reaching certain conditions the agent will use that memory

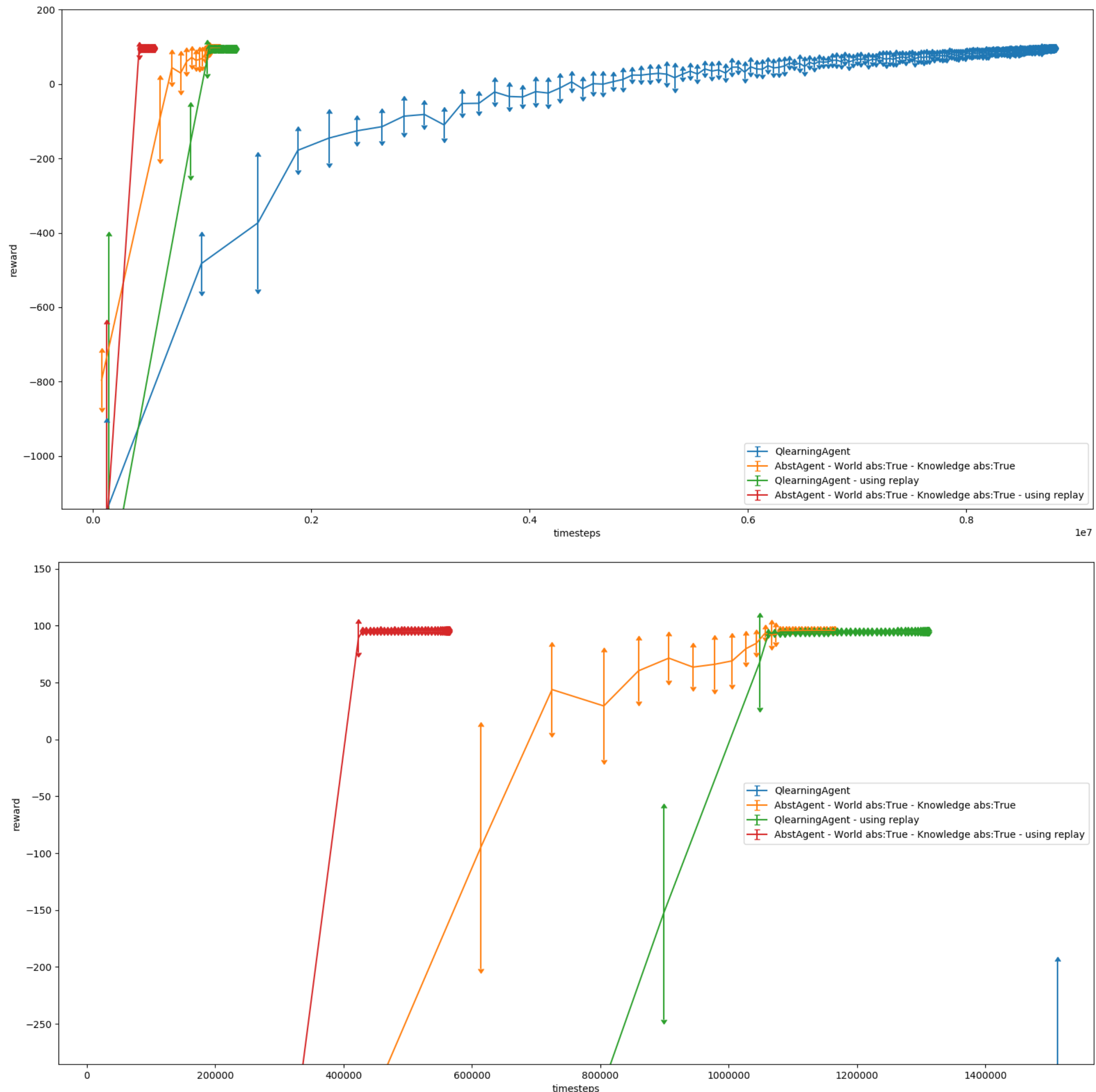


Figure 4.18: Comparison between the agent's using replay or not in the 24-rooms map. The second image is a zoom of the first one.

to virtually learn using those observations stored. Experience replay was also implemented in both Q-learning and Abstraction agents in order to test its impact and viability with the Abstraction Agent. The details of the implemented experience replay are: the agent stores its observations in the buffer memory, and when the buffer memory has 1000 observations or the episode ends, the agent will go through the observations in the reverse order they were stored/observed and perform a learning step (algorithm 3) using that observation. It will then empty the buffer memory. This allows the agent to backwards propagate the values faster, which allows for faster learning that can be seen in Figs.4.18,4.19.

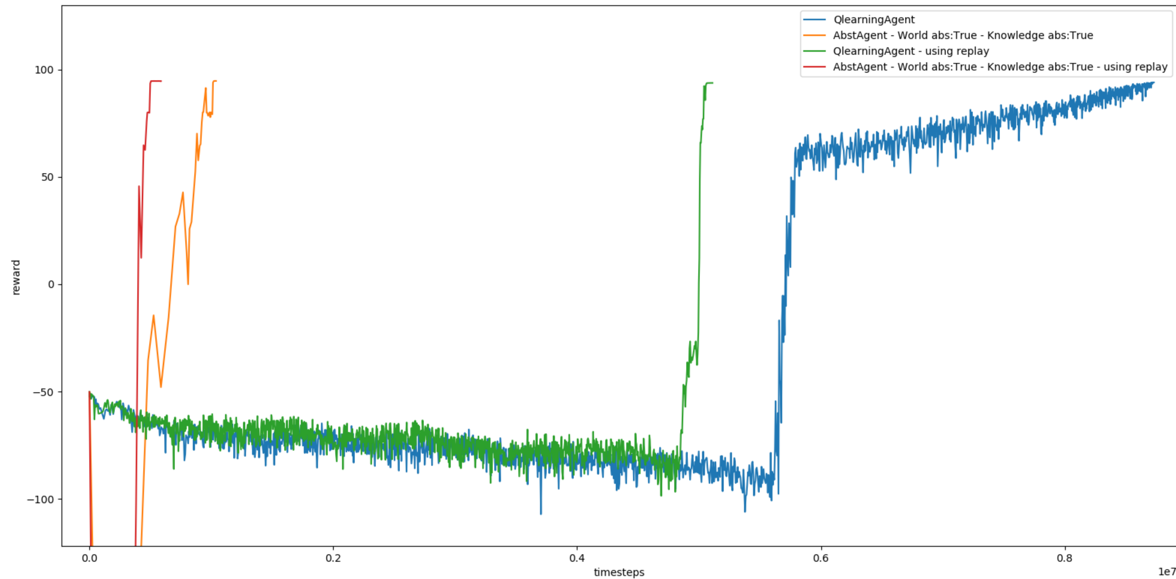


Figure 4.19: Comparison between the agent's using replay or not in the Lava-24-rooms map with lava that ”kills”. Not showing variance so it's easier to observe.

4.3 Playroom

This playroom world is, if not identical, at least similar to the playroom in ”Intrinsically Motivated Reinforcement Learning”[2]. A summary of the playroom domain is:

- The playroom has objects: a light switch, a ball, a bell, two movable blocks that are also buttons for turning music on and off, as well as a toy monkey that can make sounds.
- The agent has an eye, a hand, and a visual marker which interact with the objects.
- The agent's actions are: 1) move eye to hand, 2) move eye to marker, 3) move eye one step north, south, east or west, 4) move eye to random object, 5) move hand to eye, and 6) move marker to eye.
- The goal is for the toy monkey to make frightened sounds, which happens when simultaneously the room is dark, the music is on and the bell is rung.
- To reach the goal the agent has to 1) get its eye to the light switch, 2) move hand to eye, 3) push the light switch to turn the light on, 4) find the blue block with its eye, 5) move the hand to the eye, 6) press the blue block to turn music on, 7) find the light switch with its eye, 8) move hand to eye, 9) press light switch to turn light off, 10) find the bell with its eye, 11) move the marker to the eye, 12) find the ball with its eye, 13) move its hand to the ball, and 14) kick the ball to make the bell ring.
- The agent's sensors tell it what objects (if any) are under the eye, hand and marker. The states are [object beneath eye, object beneath hand, object beneath marker, music state(on or off), light state(on or off)].

Parameters used in the playroom world are equal to the ones used in the toy world expect the learning rate is equal to 0.1 in the Fig.4.20 and 0.01 in the Fig.4.21. This lower learning rate values are used because this world is not deterministic, i.e. a state-action pair doesn't always leads the agent to the same "next-state".

The world abstraction performs a total of 3 S-cuts in this world, partitioning it into 4 abstract states: (music off, light off), (music on, light off), (music off, light on), (music on, light on).

The S-cut can be seen bellow in the following format (cut quality - example bridge - amount of states in 1st abstract state, amount of states in 2nd abstract state):

- (10077 - (?, ?, ?, ?, False), (?, ?, ?, ?, True) - 686, 426) - this cut divides the world into 2 abstract states, creating an abstract state composed by states with "lights off" and another composed by states with "lights on"
- (4201 - (?, ?, ?, False, True), (?, ?, ?, True, True) - 343, 343) - this cut divides the abstract state where the "lights are on" into 2, one with states where "music is off" and another with states where "music is on"
- (1917 -(?, ?, ?, False, False), (?, ?, ?, True, False) - 213, 216) - this cut divides the abstract state where the "lights are off" into 2, one with states where "music is off" and another with states where "music is on"

4.4 Minecraft

Minecraft is a popular sandbox game that features a 3D world constituted by blocks, which means is that at every (x,y,z) coordinate there is a type of block (air, stone, lava) that interact differently with the player or the agent. In order to test the agent a custom level was created, this level can be seen in Figs.4.22.

The interaction between the agents and the minecraft world is possible thanks to the malmo plataform[6].

There are mainly 4 types of block: Air, Stone, Lava and Emerald. The states are composed by the types of block in the 3x3x3 space around the agent, a total of 27 blocks or features in the state, plus the (x,y,z) coordinate values, which are 3 individual features, and the orientation of the agent (north, south, east, west). The state has then a total of 31 features.

There are 5 actions available: "Move forward", "Turn right", "Turn left", "Break block" and "Jump forward". The "Break block" action breaks a block in front of the agent and the "Jump forward" action is used to climb the stairs.

Parameters used in the minecraft world:

- epsilon = 0.1
- discount factor = 0.999
- learning rate = 0.1
- The agents use experience replay.

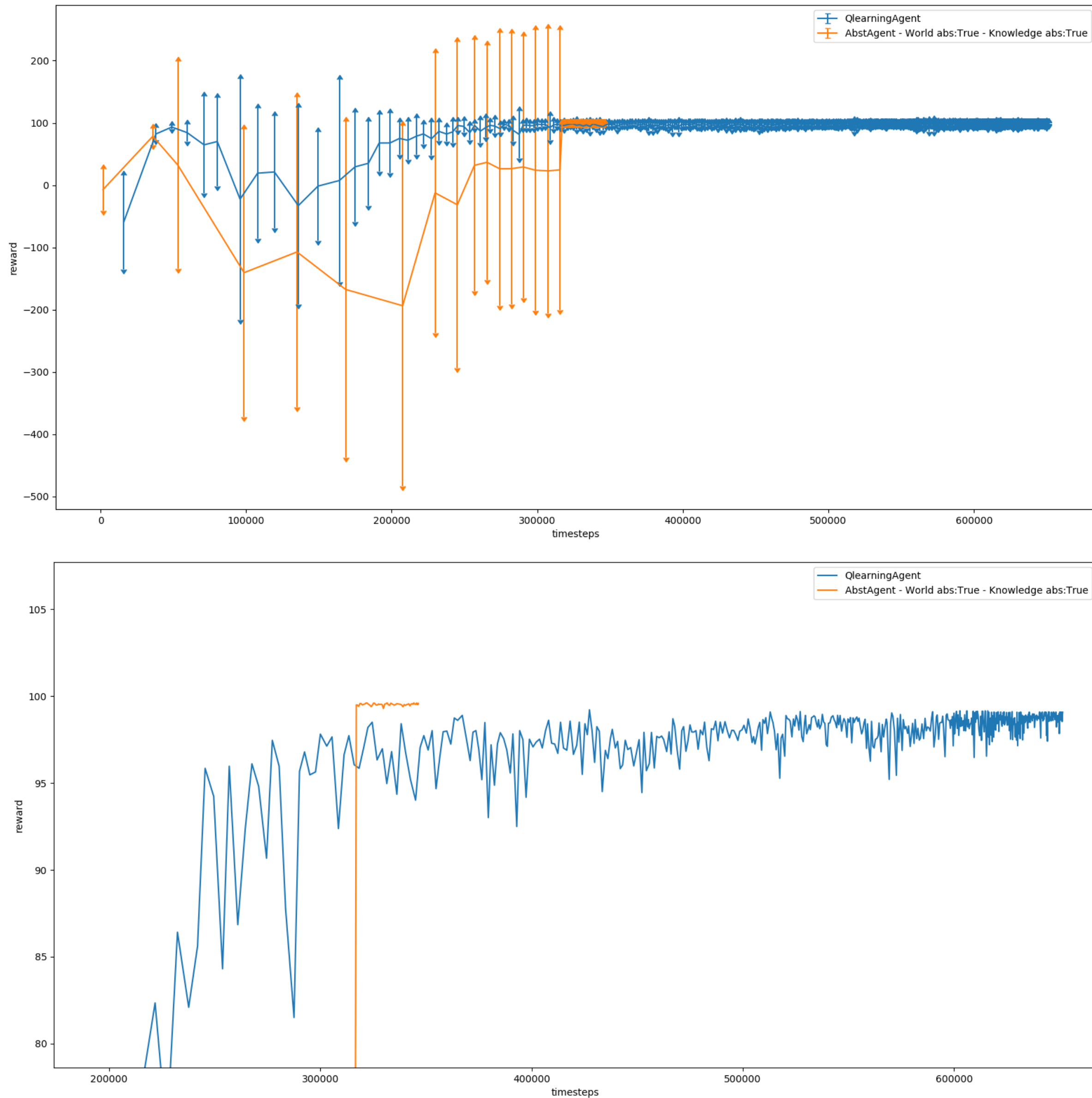


Figure 4.20: Comparison between the Q-learning agent and the Abstraction Agent. Learning rate = 0.1. The 2nd image is a zoom of the first one but not showing variance.

Figure 4.21: Comparison between the Q-learning agent and the Abstraction Agent. Learning rate = 0.01. The 2nd image is a zoom of the first one.

Figure 4.22: The level has 2 floors. The agent starts in middle room of the bottom floor which is composed of 9 rooms. In order to reach the top floor the agent needs to break the blocks that block the exits of the middle room, it has to go to one of the 2 corner rooms that have a stairs and then it has to climb the stairs to the top floor. This first floor is surrounded by lava that will kill the agent, immediately ending the episode and giving a negative reward. In the top floor the agent just has to reach the emerald green block in the center.

A learning rate of 0.1 is used because the blocks can be broken and as such the world is not deterministic, i.e. a state-action pair doesn't always leads the agent to the same ”next-state”.

A different and not optimal way to evaluate the agents was used because performing a step in the minecraft world requires (a lot) more real time than the other worlds, with 200.000 steps taking about 12h to run.

To evaluate the agents 200.000 timesteps of Q-learning exploration were saved and then experience replay was turned on and 300.000 timesteps of Q-learning exploration with replay were saved. A dataset with a total of 500.000 timesteps of exploration was created, where the agent reached the goal 422 times.

The agents simulated learning in the minecraft world by using this dataset. To evaluate when the agents have stopped learning, the agents were submitted to runs, a run being the agent using the entire dataset of observations to learn. After each run the value of the best action (primitive action or option) in the source state (the state the agent begins) is saved. When the (best value of the last 3 runs) - (average value of the last 3 runs) ¡ 1, i.e. when the value improvement stabilizes, the agent is considered to has finished learning.

This might be a good evaluation of the Q-learning agent's performance, however one of the highest benefits of the Abstraction agent, provided by the world abstraction, is its improved exploration capabilities, which cannot be evaluated by using a dataset created from the exploration of a Q-learning agent.

Nonetheless, the Q-learning agent stabilized after 17 runs while the Abstraction Agent only required 14 runs.

Another interesting information can be seen, specially well in the 2nd run of the Abstraction Agent, by looking at the Abstraction Agent's action decision possibilities in the source state.

Source state (x,y,z,(...),orientation) = (50, 227, 50, (...), 0)

Global policy - [9.844024229555746, 9.839254626147357, 9.856200440190825, -10.175017028161168, 9.84457855974118]

Option - bridge: ((51, 227, 50,(...), 0), (51, 227, 51,(...), 0)) value: 70

Option - bridge: ((50, 227, 50(...), 0), (50, 227, 51,(...), 0)) value: 458

Option - bridge: ((50, 227, 46, (...), 2), (50, 227, 45,(...), 2)) value: 516

As it can be seen, just after 2 runs the Abstraction agent already has options of high value which could be used to lead the agent to more valuable places of the state space, while the Q-values of the actions are all still bellow their initial value of 10.

Chapter 5

Conclusions

The objective of this work is to integrate the qualities of logical based systems into reinforcement learning so that it's able to learn more complex tasks and faster without a manual, complex and time consuming definition and representation of the knowledge domain that is necessary in logical based systems.

In the introduction 1 it's discussed that the approach we are looking for, to perform the objective mentioned above, should be a combination of **state abstraction**, **temporal abstraction** and **intrinsic motivation**.

In the related work 2, multiple works are mentioned that already use these concepts and integrate them with reinforcement learning. Many parts of my approach are based on these works.

It is then, in chapter 3, that I present my approach or ideas for an integration of reinforcement learning with the autonomous creation, representation and learning of knowledge of higher complexity and generalization then common reinforcement learning methods like Q-learning, by using the 3 concepts above. My approach is composed of 3 components: the world abstraction, the knowledge abstraction and its integration with reinforcement learning.

The world abstraction autonomously discovers bottleneck states by analysing state transition graphs. These bottlenecks are seen as potential subgoals. It abstracts the state space into multiple abstract states, which are separated by these bottleneck states, and creates temporal abstractions or options using these bottleneck states as intrinsic goals. The world abstraction thus integrates both state and temporal abstraction and intrinsic motivation to create an abstract representation of the underlying model of the world, an abstract model that allows it to learn and represent more complex behaviour on how to navigate the state space independently of extrinsic rewards. This abstract model allows it to explore the state-space more efficiently and thus learn tasks faster.

The knowledge abstraction uses a neural network to abstract the state, separating a state into its features and then providing them independently to the neural network as input. This neural network is able to learn knowledge that can be generalized to multiple states that share individual feature values, allowing it to estimate the reward of actions in states it has never seen before.

The integration component is where the world and knowledge abstraction are integrated with the learning process and decision making of reinforcement learning, specifically, where the base Q-learning

will be integrated with the options from the world abstraction and the immediate reward estimates from the knowledge abstraction.

I implement this approach and thus create an agent which I call Abstraction Agent, this agent being the integration of a Q-learning agent with my approach.

The Abstraction Agent is compared with a Q-learning agent in order to evaluate the impact of my approach in the agent's performance.

Experimental results show that:

The world abstraction is able to autonomously create a useful abstract model of the world which contains learned representations of more complex behaviour on how to navigate in it, increasing the learning speed of the agent mainly by increasing the efficiency of its exploration. This performance increase is shown to remain in different worlds, even in worlds with no ”rooms”, i.e. with no intuitive abstraction.

The world abstraction is also shown to have a flaw, which can be seen in 4.2.4. In summary, because an option's policy is independent of the extrinsic goal, the objective of an option's policy will be to lead the agent to **any goal** of that option while maximizing intrinsic reward, **regardless** of the extrinsic value of that goal. This usually results in leading the agent to the closest goal of that option even if that goal is worse (in the global spectrum) in relation to others. This affects the agent's performance in worlds with negative rewards (like the maps with lava that were used), where the choice of intrinsic goal of an option might lead the agent to a part of the state-space where it has to receive a negative reward in order to reach the goal.

The knowledge abstraction is shown to able to learn generalized knowledge that allows the agent to predict the value of actions in states it has never been in, thus increasing its performance.

In summary, the experimental results conclude that my approach, that integrates spatial-temporal abstraction and intrinsic motivation with reinforcement learning, allowing an agent to autonomously create, represent and learn knowledge of higher complexity and generalization then common reinforcement learning methods like Q-learning, is able to increase the overall learning speed, reducing the total amount of steps necessary for the agent to consistently reach the goal and solve the task.

Chapter 6

Future Work

The most important improvement would be substituting the ”bridge values” solution with a better solution that resolves the flaw mentioned in the chapter Experimental results in 4.2.4. A possible solution is also mentioned in that chapter.

Other improvements that could be considered for future work are:

1. Create hierarchies or layers of abstraction by trying to abstract the current abstraction done of the world. The current abstraction of the world can be considered as an abstraction layer of the world, as it can be seen in 3.1, and subsequent layers of abstraction that abstract the layer bellow could be created by considering the abstract states and options of an abstraction layer as a new world to be abstracted, where the abstract states would be states and the options would be actions.

2. Currently the knowledge abstraction needs that ”manual abstraction list” in order to create the neural network, because the network structure is fixed upon creation. A useful improvement would be to remove the need for a ”manual abstraction list”.

3. Implement a forgetting mechanism that allows the world abstraction to forget the state transition graphs of it’s abstract states that have ”stabilized” and probably won’t be divided anymore.

4. The option’s policies are independent from any extrinsic goal, which means that upon a goal change the option’s policies don’t change (although they might need to in order to solve the mentioned flaw4.2.4). However the option’s value depends on the extrinsic goal and most importantly on the global Q-policy, which takes a long time to update it’s values upon a change of goal. A future work might be able to use the intrinsic nature of the option’s policies and grade the value of an option differently, in a way that gives the agent a faster task switching ability or the ability to relearn faster upon a goal change.

Bibliography

[1] A. G. Barto, S. Singh, and N. Chentanez. Intrinsically motivated learning of hierarchical collections of skills. In *Proceedings of the 3rd International Conference on Developmental Learning*, 2004.

[2] N. Chentanez, A. G. Barto, and S. P. Singh. Intrinsically motivated reinforcement learning. In *Advances in Neural Information Processing Systems 17: Proceedings of the 2004 Conference*, 2005.

[3] F. Chollet et al. Keras. `https://keras.io`, 2015.

[4] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[5] B. Hengst. Discovering hierarchy in reinforcement learning with hexq. In *ICML*, volume 2, pages 243–250, 2002.

[6] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell. The malmo platform for artificial intelligence experimentation. In *IJCAI*, pages 4246–4247, 2016.

[7] R. Krishnamurthy, A. S. Lakshminarayanan, P. Kumar, and B. Ravindran. Hierarchical reinforcement learning using spatio-temporal abstractions and deep neural networks. 05 2016.

[8] M. C. Machado and M. Bowling. Learning purposeful behaviour in the absence of rewards. *arXiv preprint arXiv:1605.07700*, 2016.

[9] I. Menache, S. Mannor, and N. Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *European Conference on Machine Learning*, pages 295–306. Springer, 2002.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[11] L. N. Olson and J. B. Schroder. PyAMG: Algebraic multigrid solvers in Python v4.0, 2018. URL `https://github.com/pyamg/pyamg`. Release 4.0.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and

E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[13] O. G. Selfridge. The gardens of learning: a vision for ai. *AI Magazine*, 14(2):36, 1993.

[14] Ö. Şimşek, A. P. Wolfe, and A. G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning*, pages 816–823. ACM, 2005.

[15] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.