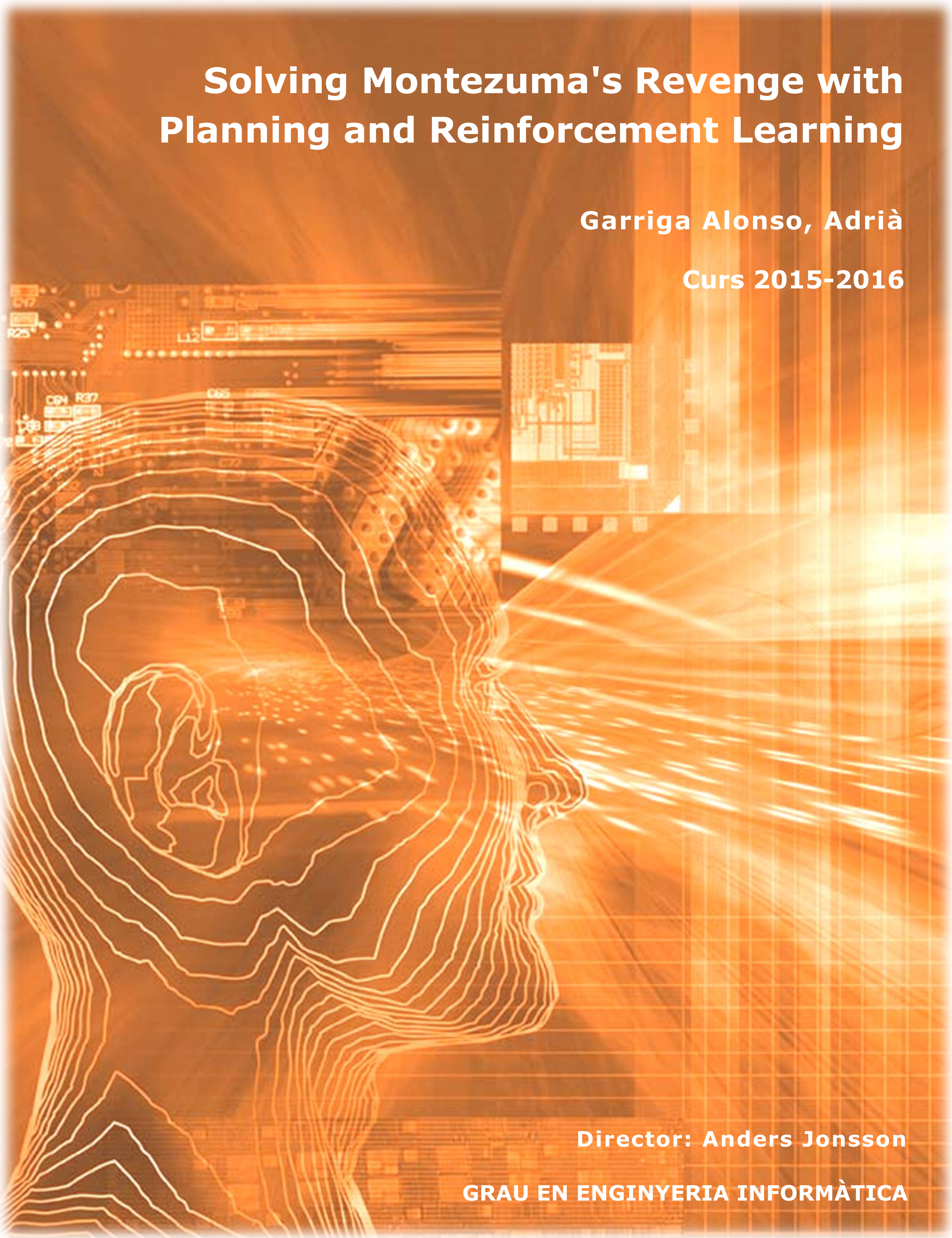

Solving Montezuma's Revenge with Planning and Reinforcement Learning

Garriga Alonso, Adrià

Curs 2015-2016

Director: Anders Jonsson

GRAU EN ENGINYERIA INFORMÀTICA

upf. Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Treball de Fi de Grau

Solving Montezuma's Revenge with planning and reinforcement learning

Adrià Garriga Alonso

Supervised by Anders Jonsson

May 3, 2018

Bachelor in Computer Science

Department of Information and Communication Technologies

Universitat Pompeu Fabra Barcelona

Acknowledgements

I wish to offer my thanks:

To my supervisor Anders Jonsson, for the guidance offered in navigating the literature and in carrying out this project. I also want to thank him for getting me interested into the fascinating field of RL.

To my good friend and upperclassman Daniel Furelos, for being an academic role model, and for the offered advice.

To Miquel Ramírez, for sharing the source code from his paper on Iterated Width, along with Geffner and Lipovetzky.

To my parents and sister for the moral support, and the support of a noisy, electricity-hungry computer that is continuously learning and planning.

Abstract

Traditionally, methods for solving Sequential Decision Processes (SDPs) have not worked well with those that feature sparse feedback. With the resounding success of Deep Q-Networks (Mnih et al., 2015) in Atari games, one of them which features sparse feedback has become infamous for its difficulty: Montezuma's Revenge.

Using the Iterated Width (Lipovetzky, Ramirez, and Geffner, 2015) planning algorithm and domain knowledge we nearly solve Montezuma's Revenge, obtaining 14900 points, up from the previous best score of 3439 (on average, Bellemare et al., 2016). We also use reward shaping to learn a part of the game using Sarsa.

Our approaches do not automatically generalise to other games. Regardless, we identify directions for future research, and hope that these domain-specific algorithms can inspire better solutions for SDPs with sparse feedback in general.

Code used in this thesis, data generated by it, and documentation can be found online, at https://github.com/rhaps0dy/solving-mr-planning-rl.

Contents

Abbreviations

AI	Artificial Intelligence
ALE	Arcade Learning Environment
DQN	Deep Q-Network
MDP	Markov Decision Process
ML	Machine Learning
MR	Montezuma's Revenge
RL	Reinforcement Learning
SDP	Sequential Decision Process
SMDP	Semi-Markov Decision Process

Chapter 1

Introduction

1.1 The problem

Us *homo sapiens* are notoriously proud of our intelligence. Intelligence is what allows us to handle the world we live in: understand our surroundings, predict the future, and manipulate it according to our will. What will happen if I move my hand to a pen and put my fingers around it? I will grasp it, and then using my muscles I will be able to use it.

It is not at all obvious how we perform this process. Indeed, this question has been philosophised on for thousands of years. The field of Artificial Intelligence (AI) tries to go even further: researchers try understand how we think, in order to build machines that exhibit those same properties.

Work on AI famously started on the summer of 1956 at Dartmouth College. John McCarthy and others proposed that a "2 month, 10 man study of artificial intelligence" would make "significant advance in one or more of [how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves] if a carefully selected group of scientists work on it together for a summer". (Russell and Norvig, 2009, Section 1.3)

60 years later, we are still working on all of these problems. But this spark ignited the tinder, and people started working on all kinds of sub-problems: computer vision, robotics, machine learning, automatic reasoning, natural language processing...

The one we are concerned about in this document is sequential decision-making. How might an agent take decisions, that have consequences, in a changing world? Much research in this topic has been done on classical games, such as checkers, chess and go, and on video games. These problems provide domains where actions have to be taken sequentially and have consequences on the future.

One, important and recent, of such advances appeared in 2015 in Nature. The paper "Human-level control through deep reinforcement learning" (Mnih et al., 2015) proposed a neural algorithm that played many of the video-games in the Atari console, knowing only the buttons it has, the score and the image, just like a human player. Their key contribution is the Deep Q-Network (DQN) algorithm, which is the successful application of deep convolutional neural networks (used in computer vision) as a function approximator for RL.

Of the Atari games, Montezuma's Revenge is one that their agent has trouble playing. The problem with this game is the *sparsity* of the rewards: it is almost impossible to get any positive feedback just by randomly hitting buttons on the console. To successfully get feedback, an agent has to understand the objects on the screen, understand what is their character and how does it move, and then purposefully plan a path to the rewards. Thus, the game has become infamous as difficult, and many RL researchers are interested in it now.

In this thesis we get around the problem of understanding the world by encoding our own, human, understanding in the machine. It is an exercise to find out how much must the machine know about the world, how few *assumptions* must it make, in order to be successful in it.

1.2 Related Work

Two very relevant papers have been recently published. They both deal with methods intrinsic to the agent of obtaining more frequent feedback.

The first, by Kulkarni et al., (2016), proposes a hierarchical model (Section **??**) with two levels. The higher level, the *meta-controller*, learns and decides towards which object of the screen the character should move, and the lower level, the *controller* learns and decides how to get there. They encode the knowledge of which are plausible objects to move towards and where is their controllable character to the computational agent.

Some of the objects are closer to the initial position than the objects that increase score in the game, so the controller can get some feedback and learn how to move. Once the controller can move between objects, the objects which produce reward are only a few abstract time steps away for the meta-controller, and it can successfully learn too. Work on replicating this paper is in progress.

The second, by Bellemare et al., (2016), deals with estimating how *novel* (not to be confused with the novelty measure in Subsection **??**) a state is, even if the agent has never seen it. This is done by examining the components of the new state (like in Subsection **??**) and the number of occurrences of each previous component, and computing a single number synthesising that. Additional reward is then given to visited states, proportional to the square root of this measure. Thus, the learner is incentivised to visit new state areas, and eventually find the environment reward in them.

Chapter 2

Background

2.1 Reinforcement Learnings

Reinforcement learning is an area of Machine Learning (ML) that models a problem focusing on maximizing a numerical reward. To achieve this, the agent obtains a representation of the situation of the problem, named *state*; then selects an *action*; interacts with the environment, by applying that action; and finally it may obtain some *reward*. After all this process the situation of the problem changes and the agent finds itself a new state. Repeating this steps several times, the agent should explore the different states of the problem and learn how to map situations to actions with the purpose of maximizing the reward.

The learning process is usually organized in *episodes*. Each episode is composed by an initial state s_0 , the first one that the environment generates; intermediate states s_t ; and may have several final states, which establish the end of an episode. The agent's goal is to maximize the total amount of reward it receives in a complete episode. When an episode ends the the environment resets and starts again from s_0 or some other initial state that belongs to the set of initial states. In some cases the agent interacts infinitely without reaching any terminal state, we call this *continual tasks*.

One of the challenging aspects of reinforcement learning is the tradeoff between *exploration* and *exploitation*. In the kind of environments this thesis is focused on, the actions applied affect not only immediate rewards but also future ones. This implies that taking actions with high immediate reward is not always the best choice, there might be some series of actions with which it obtains higher rewards in the future. In reinforcement learning, the agent should exploit immediate rewards in order to obtain feedback of how is it performing. Nevertheless, it may also explore different actions that do not have as high immediate reward in order to find higher future rewards.

2.1.1 The environment

We will describe this as in the textbook *Reinforcement Learning: An Introduction* from Sutton and Barto, (1998, Section 3.1).

The *environment* is the main source of information about the problem that we want to model with which the *agent* is able to interact. This interactions and information about the problem must be stated in a specific way.

The state is the peace of information, from the environment, that our agent uses to make decisions. Interactions are transferred in the form of actions that the environment makes available for the agent. The information given about the problem must change based on the actions applied.

The agent and environment interact with each other in a sequence of discrete time steps, $t = 0, 1, 2, \ldots$ At each time step, t, the agent receives some state $s_t \in S$ and on that basis selects an action $a_t \in A(s_t)$. One time step later, the agent receives a numerical reward, $r_{t+1} \in \Re$, and finds itself a new state, s_{t+1}. The state transition and reward must depend on the sequence of past actions and states, if not the agent will not be able to figure out what to do for obtaining the highest rewards. The methodology in which the agent interacts step by step with the environment is named Sequential Decision Process (SDP).

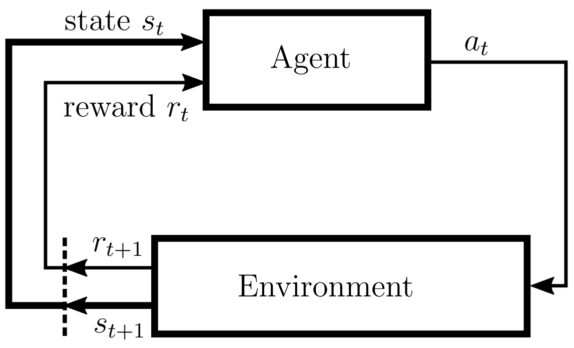

Figure 2.1: Diagram of interaction between agent and environment (Sutton and Barto, 1998, Section 3.1)

This is an abstract and very flexible framework that fits in a large variety of problems.

The different components of this interface can be arbitrarily defined to model different kinds of actions, time steps, states and environments. But it restricts rewards to the real domain. For example, the state can be any kind of information about the environment, from just an image of some maze to complex market statistics. This fact is important to us because the algorithm proposed in this thesis uses domain knowledge to define a state representation of the environment composed by images and additional information.

2.1.2 Markov Decision Processes

A Markov Decision Process (MDP) is a restricted case of SDPs (Sutton and Barto, 1998, Section 3.5). In MDPs, the state s_{t+1} generated by applying action a_t in state s_t only depends on this two factors, unlike in SDPs that may depend on any previous states, rewards or actions. This fact is named the *Markov property* and is formally described as:

$$P\{s_{t+1} = s, r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\} = P\{s_{t+1} = s, r_{t+1} = r | s_t, a_t\} \tag{2.1}$$

As stated, the probability distribution over states and rewards only depends on the immediate previous state and action, not the full history. Bear in mind that the state s_t may contain some representation of previous states, actions and rewards but not the full history. This abstract could be as simple as the number of actions taken. All concepts presented in this thesis assumes that the environment can be defined as an MDP. The problem can be expressed as a tuple $\langle \mathcal{S}, \mathcal{A}_s, \mathcal{P}_a(s, s'), \mathcal{R}_a(s, s'), \gamma \rangle$ where:

- $\mathcal{S}$ is the state space.
- $\mathcal{A}_s$ is the set of actions available in state s.
- $\mathcal{P}_a$ is the transition probability function $/mathcalP_a : \mathcal{S} \times \mathcal{S} \rightarrow [0, 1]$ for action $a \in \mathcal{A}_s$. To clarify, $\mathcal{P}_a(s, s')$ is the probability that taking action a in state $s \in S$ will lead to state $s'S$.
- $\mathcal{R}_a$ is the reward function $\mathcal{R}_a : \mathcal{S} \times \mathcal{S} \rightarrow \Re$ for action $a \in \mathcal{A}_s$. To clarify, $\mathcal{R}_a(s, s')$ is the expected reward for transiting from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ by taking action a.

- $\gamma \in [0,1]$ is the discount factor of future rewards used to calculate returns (Subsection 2.1.3)

$\mathcal{S}$ and $\mathcal{A}_s$ may be infinite sets which implies that $\mathcal{P}_a(s,s')$ and $\mathcal{R}_a(s,s')$ may also be infinite. If this happens the MDP is named infinite MDP.

2.1.3 Returns

We have defined the goal of the agent, that is to maximize the total reward obtained in an episode. We also said that the agent must learn to reach that goal, but we have not defined yet how the agent should select the actions to achieve it. The expected return R must be defined in a way that encourages the agent to learn, so R is some specific function of the reward sequence. The simplest way of computing it is just adding all the future rewards.

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \tag{2.2}$$

Where r_T is the reward obtained in some terminal state. In continual tasks the time step $T = \inf$ so the return, which is what we are trying to maximize, will also be infinite. To avoid this problem we must introduce the *discounted return*, which adds in a discount factor γ. This factor modifies the previous formulation in the following way:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.3}$$

where γ is a parameter, $0 \leq \gamma \leq 1$. Observe that as k approaches infinity the weight applied to r decreases, if $\gamma < 1$. If the final time step T tends to infinity then this new formulation converges, not like the previous one. The discount factor is a hyperparameter related to how farther rewards should be taken into account or not. If we use a value near 0 then the algorithm is capable to rewards that are few time steps away.

2.1.4 Reward shaping

Sometimes an agent isn't "smart" enough to reach the environment rewards. In this scenario some new rewards can be added to guide the agent towards the original rewards.

Specifically, applying reward shaping to a MDP is to modify its reward function

$\mathcal{R}_a(s, s')$ getting a new one $\mathcal{R}'_a(s, s')$, the difference between them is defined by the *shaping reward function.* More formally $\mathcal{R}'_a(s, s') = \mathcal{R}_a(s, s') + \mathcal{F}(s, a, s')$ where $\mathcal{F} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow$ is a bounded real-valued function.

The function $\mathcal{F}$ should be chosen using expert knowledge about the domain or in a general manner that works for any MDP. This choice should be made carefully, because modification in the rewards of the MDP should guide the agent to reach the same optimal policy π^* and not another policy which is suboptimal for the original problem. To guarantee consistency with the optimal policy, $\mathcal{F}$ must be *potential-based.* We say $\mathcal{F}$ is potential-based if there exists a real-valued function $\phi : \mathcal{S} \rightarrow$ such that for all $s \in S \setminus \{s0\}, a\mathcal{A}, s' \in \mathcal{S}$

$$\mathcal{F}(s, a, s') = \gamma\phi(s') - \phi(s) \tag{2.4}$$

By defining $\mathcal{F}$ in this way we can ensure that the reward shaping is robust, near-optimal policies in the original MDP remain near-optimal in the new MDP.

2.1.5 Policy

As described by Sutton and Barto, Section 1.3, a policy defines the learning agent's way of behaving at a given time. It selects the action, given the environment state. During the learning process of the agent may vary multiple times its policy, looking for the best mapping of state-action pairs that fits the problem. This is call the optimal policy * and the objective of any reinforcement learning algorithm is to approximate its policy to the optimal policy.

In most cases the policy is a probability distribution over the actions that can be choosed in some state s. Is denoted as (a,s) where s belongs to the state space and a to the action space, fulfilling the following condition:

Chapter 3

Methodology

3.1 Montezuma's Revenge

3.1.1 Description

You, the player, are Panama Joe, an intrepid explorer-archaeologist. Your latest trip has brought you to discover the entrance to an Aztec pyramid. Filled with excitement, you rush to search the treasures that surely await inside. But the pyramid is full of traps and monsters. You will need all of your wits and agility to get out alive![1]

In the game, Panama Joe can run, jump and climb ladders, ropes and poles. The pyramid he can explore is divided in 24 screens, numbered 0–23, depicted in Figure 3.1. The player starts the game in screen 1 and ends when collecting the gems in screen 15. When the player completes the game, it simply resets with a different colour scheme.

Joe has a number of lives, initially 5. When they are over and Joe loses a life again, the game is over. Lives can be lost by touching monsters, touching blue wall traps (such as those in screen 12), falling into quicksand pits or falling from too high.

The player can gain score for a number of things: collecting gems (+1000), keys (+100), the sword (+100), the torch (+3000) or the mallet (+200); opening a door (+300); or killing a monster with the sword (+2000). A life is gained for every 10 000 points gained, with a maximum of 6 lives. The torch allows you to see in the lowest floor of the pyramid (which is otherwise black). The sword allows you to kill one monster. The mallet allows you to be immune to monsters for a period of time.

[1]Game background from the web review by Adair, (2007).

The game can be rendered impossible to complete, since there are 6 doors but only 4 keys. The two doors in screen 17 need to be opened to finish, and either one of the doors in screen 1 needs to be opened to be able to advance. So the player can either open both doors in screen 1 and have the bottom floor remain black, or leave one door in each of the screens 1 and 4 unopened.

At each time step, the player can take 8 different actions: Noop (stay still), Fire (jump straight up), Up, Right, Left, Down, LeftFire (jump to the left), RightFire. The rest of the 18 actions permitted by the Atari overload to one of these.

3.1.2 Memory layout of the Atari 2600

The Atari 2600 uses the 6507 (index˙cmd), which can address 8 KB of memory. Addresses range from 0x0000 to 0x1FFF. Addresses larger than 0x1000, included, are mapped to the (index˙cmd) cartridge that contains the code of the game. Addresses lower than that are used for drawing to screen, looking at the controller input, . . . and, most importantly, the (index˙cmd). The 2600 has only 128 bytes of index˙cmd, which are addressed in the range 0x0080–0x00FF, both included. (*Atari 2600 Specifications*)

Throughout this document we will prefix a number with “0x” if it is expressed in base 16. If a described memory position has four hexadecimal digits, it is an absolute processor address. Otherwise, it is assumed to be in the range 0x0000–0x00FF.

3.1.3 Reverse-engineering Montezuma’s Revenge

We used the Arcade Learning Environment (Bellemare et al., 2013) to take several simultaneous screen and index˙cmd snapshots. We usually took 5 to 10 snapshots in the space of 1 or 2 seconds, while performing certain action in the game. Then, we looked at the bytes that changed value from snapshot to snapshot.

We also used the debugger built-in to the Atari 2600 emulator, Stella (Mott et al., 1995). Using the command `breakif <condition>`, that pauses the game and shows the debugger if a condition is met, enabled us to play and check whether a memory position behaved as we suspected. In some cases we used the disassembled code in that debugger too.

In Table 3.1 and in the list below we reproduce the layout of the Atari 2600’s main memory and what each position affects in Montezuma’s Revenge.

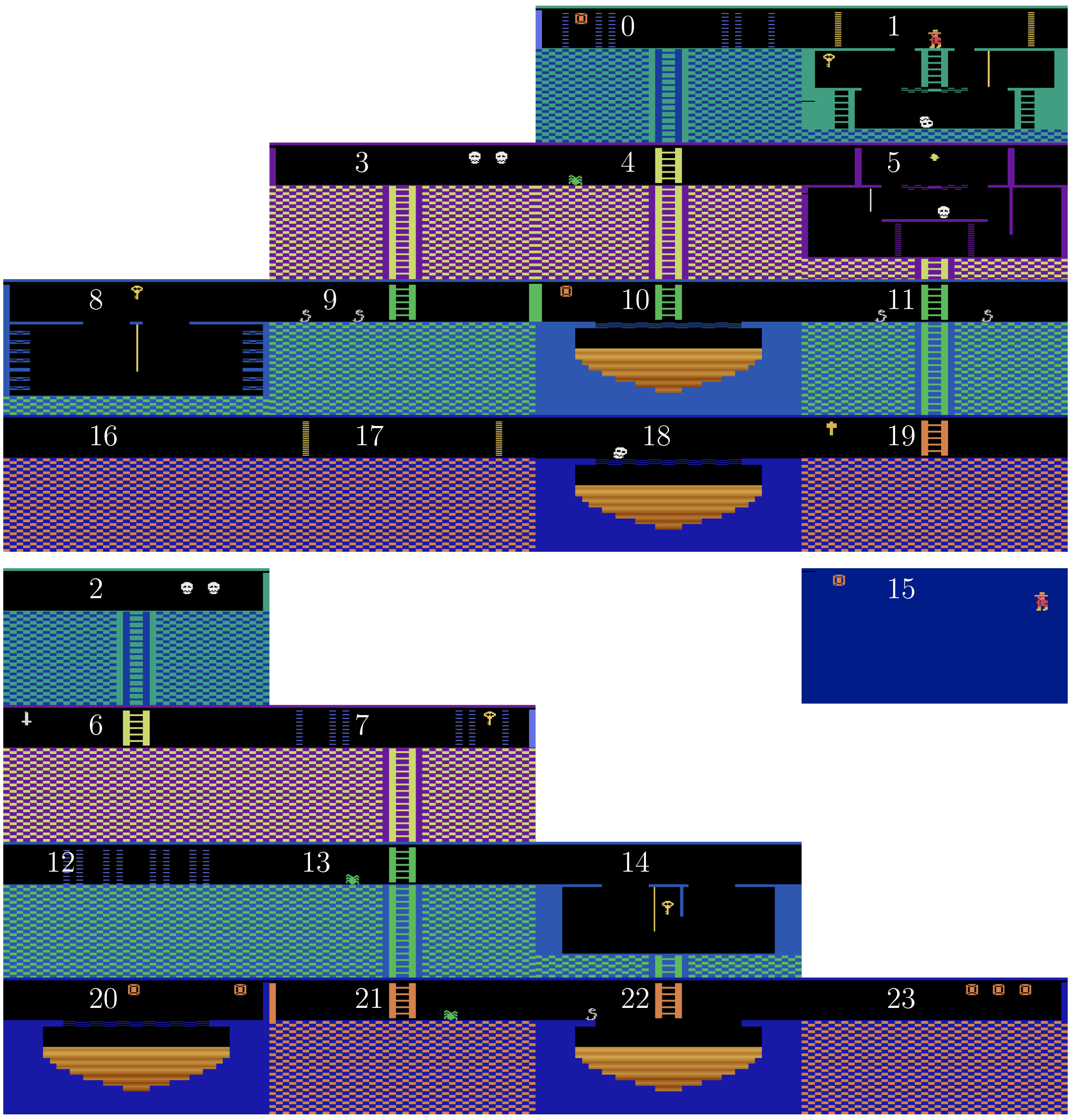

Figure 3.1: The complete map of Montezuma's Revenge. Rooms are numbered from left to right and from top to bottom. The pyramid they form has been cut to fit in the page. Room 15 is located to the left of room 16. The screens are not numbered in the game. The player starts in room 1 and finishes in room 15.

Some entries can be modified in the Stella debugger when the game is running, and affect the game. If an entry is not editable, it will be marked with an asterisk (*). The values that are not marked editable may be editable in other circumstances, and are probably editable in the middle of the computations within a frame. However, their value has been observed only going back to what it was if modified between frames.

We have a very strong suspicion that nowhere in the index˙cmd is stored the layout of the screen, or where collisions can occur. This is coded into the programming path (with branches depending on the character's position), or stored somewhere in the ROM. A learner that hopes to generalise between screens needs to have access to that information.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	80			83												
9				93	94	95									9E*	
A											AA	AB				
B		B1	B2		B4						BA				BE	BF
C		C1	C2*	C3											AE	AF
D					D4		D6		D8							
E											EA*					
F																

Table 3.1: The known index˙cmd layout for Montezuma's Revenge.

1. **0xBA**: Editable. The number of lives the player has left, that is, the number of times the player can die and continue the game afterwards. Controls the number of hats displayed at the top. Panama Joe starts with 5 lives. The counter can go up to 6 without graphical problems.

2. **0x83**: Editable. The current screen. If edited, the new screen will only be partially drawn. Sometimes, one can exit the screen and reenter it by playing and the issue will go away.

3. **0xAA**: Editable. The X position of the character. If set to the middle of the air, Panama Joe will fall.

4. **0xAB**: Editable. The Y position of the character. If set to the middle of the air, and there is a platform below, the character will not fall! Instead, it will behave as if it was on a ladder. The Y values of the three floors that every level has are 0x94 or 0x9C, 0xC0, and 0xEB.

5. **0xD6**: Editable. The current frame of the jump. Set to 0xFF when in the ground. Set to 0x13 when the jump starts. When jumping, the game adds to

the Y of Panama Joe the values from the array starting at memory position 0x1E47. Thus, if set to higher than 0x13, the game behaves oddly. It also can be reset to whatever value at any time, causing Panama Joe to start a jump, even in mid-air.

6. **0xD8**: Editable. The current frame of the fall. Normally set to 0x00. When falling off an elevated ground, or off a jump, this value will begin to count up. If it is 0x08 or higher when Panama Joe touches the ground, he will die.
7. **0x93**, **0x94**, **0x95**: Editable. The score, represented in . This is, every nibble represents a decimal digit.
8. **0xC1**: Editable. The contents of the player's inventory. Each possible object in it is associated to a bit, that is set if the object is in the inventory. At most 6 objects can be carried without causing graphical corruptions. The objects and their associations are:

0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01
torch	sword	sword	key	key	key	key	mallet

   If the inventory's value is changed, collecting items by touching them stops working.
9. **0xC2 (bits 3, 2)**: Not editable. Whether the doors in the screen are closed or open. Only means this in screens 1, 5 and 17. When the bit is set, the door is closed. Bit 3 controls the door in the left, bit 2 the one in the right.
10. **0x80**: Editable. The current frame. This memory position starts the game at 0x00 and increments by one every frame.
11. **0xBE**: Editable. The frame of the rotating skull's animation, in screens where there is one.
12. **0xAF**: Editable. X of the rotating skull, when there is one (screens 1 and 18). It is not in the same scale as the player's X. Its values range from 0x16 to 0x48, inclusive.
13. **0xAE**: Editable. Y of the rotating skull. Also in its own scale, and cannot take it away from its floor.
14. **0xEA**: Not editable. The number of times the rotating skull in the first screen has changed direction. Remains even after changing screen. If untouched, the lowest byte indicates the direction the skull is moving in. Can be changed, but it does not change the direction of the skull.

15. **0xBF**: Editable. relative Y position of the jumping skulls, in screens where they are present. It oscillates between 0x00 and 0x0F, where 0 is the topmost position. The game makes relative changes to this value, so if set to F while the skulls on mid-air, they will not go below that point afterwards.

16. **0xC3 (bit 1)**: Editable. Whether the rotating skull is moving (set) or not (unset). The function of the rest is unknown.

17. **0x9E**: Not editable. The current sprite drawn for Panama Joe. This is what changes every few frames to show the character moving. Possible values: (0x00) standing still, (0x2A) walking frame, (0x3E) still, on a ladder, (0x52) ladder climbing frame (0x7B) still, on a rope, (0x90) climbing a rope, (0xA5) mid-air, (0xBA) upside down, left foot up, (0xC9) upside down, right foot up, (0xDD, 0xC8) alternate flashing frames when dead by a monster.

18. **0xB4 (bit 3)**: Editable. Whether Joe is looking to the left (set) or the right (unset). The function of the rest is unknown.

19. **0xB1**: Editable. The collectable sprite that is drawn. Each screen has an associated position where a sprite that may be collected, or a monster, is drawn. The things that are drawn, associated with the value of the byte that draws them, are: (0) no sprite, (1) jewel, (2) sword, (3) mallet, (3) key, (5) jumping skeleton, (6) torch, (7) blinking snake-torch, (8) snake, (9) blinking snake-spider, (A) walking spider. The rest of the values cause corruption. The colour of this sprite is controlled by memory position 0xB2.

20. **0xB2**: Editable. The colour of the collectable sprite. All values of the byte seem produce a valid colour and no corruption.

21. **0xD4**: Editable. Modifies collectables (from 0xB1), monsters and ropes. The values and their effects are: (0) one sprite, (1) two sprites, (2) two sprites, separated with enough space for another sprite, (3) three sprites, filling the space in value 2, (4) two sprites, very separated, (5) the sprites become wide, (6) three very separated sprites, (7) a very wide sprite. Only the three least significant bits seem to affect anything.

3.2 Learning

3.2.1 State-action representation

Using the reverse-engineered memory layout (Subsection 3.1.3), we crafted a state representation of screen 1, without allowing for life loss, in Montezuma's Revenge. This representation is aliased several times in the actual game's state, but it contains enough information to satisfy the Markov property (Subsection 2.1.2).

The state representation is a vector $v_1, \dots, v_6$ of 6 values, calculated based on the index˙cmd of the game state in the emulator and on the previous state. We will use index˙cmd(0xx) to denote the current value of the memory position x. The intervals of possible values are over the natural numbers.

- Skull position: $v_1 = \text{index˙cmd(0xAF)} - 0x16$, $v_1 \in [0, 51)$.
- Skull direction: $v_2 = 1$ initially, set to 0 if $v_1 = 0$, set to 1 if $v_1 = 50$, otherwise keep the same value as the last state.
- Joe's X: $v_3 = \text{index˙cmd(0xAA)}$, $v_3 \in [0, 256)$.
- Joe's Y: $v_4 = \text{index˙cmd(0xAB)}$, $v_4 \in [0, 256)$.
- Whether Joe has the key: $v_5 = \begin{cases} 0 & \text{if } \textsc{Bitwise-And}(\text{index˙cmd(0xC1)}, 0x1E) = 0 \\ 1 & \text{otherwise} \end{cases}$
- Whether Joe will lose a life upon touching the ground, or has already done so: $v_6 = \begin{cases} 0 & \text{if index˙cmd(0xD8)} \geq 8 \vee \text{index˙cmd(0xBA)} < 5 \\ 1 & \text{otherwise} \end{cases}$

We will use the restricted set of 8 actions of Montezuma's Revenge.

In total, we have $\prod_i |\text{set}(v_i)| = 51 \cdot 2 \cdot 256 \cdot 256 \cdot 2 \cdot 2 = 26\,738\,688$ possible states, which multiplied by 8 restricted actions in Montezuma's Revenge gives us 213 909 504 possible state-action pairs. If we store the value of $Q(s, a)$ for each of them as a 32-bit floating point number, we use $(213\,909\,504 \cdot 4\,\text{bytes})/10^6\,\text{bytes} \approx 855\,\text{MB}$. This is a large, but not outlandish, amount of memory.

3.2.2 Shaping function

One of the main problems with Montezuma's Revenge is that the rewards are very far apart. To eliminate this hurdle, we used shaping as explained in Subsection **??**.

We define our potential function $\phi : \mathcal{S} \mapsto [1,2]_{\mathcal{R}}$, $\phi(\langle v_1, \ldots, v_6 \rangle) = 1 + \textsc{Phi}(v_3, v_4, v_5, v_6)$. [2] Phi is described in Algorithm 1.

The agent will receive positive rewards for climbing potential. But why is our function function ϕ always positive? In the shaped MDP, the additional reward is given by (Subsection **??**):

$$F(s, a, s') = \gamma\phi(s') - \phi(s) \tag{?? revisited}$$

Consider the case where $\phi(s) = \phi(s')$. Then, if $\phi(s) < 0$, $F(s, a, s') > 0$! Our agent is rewarded for doing nothing and remaining in a "bad" position, and will prolong the episode as much as possible without moving towards where we are interested. In contrast, if $\phi(s) > 0$, the agent will be incentivised to remain in that potential as little as possible.

The function Phi takes a few seconds to compute for all values of v_3, v_4, v_5, as we do and cache before running Sarsa. A depiction of ϕ and Phi for all values of v_3, v_4, v_5 is shown in Figure 3.2.

Explanation of ϕ

Dist is Euclidean distance with the y scaled, analogous to the equation of an ellipse.

Project projects the point $\langle x_1, y_1 \rangle$ to the line defined by $\langle x_2, y_2 \rangle$ and $\langle x_2', y_2' \rangle$, perpendicularly. The value it returns comes from the system of equations: $x_1 + v_{x_1} t_1 = x_2 + v_{x_2} t_2$, $y_1 + v_{y_1} t_1 = y_2 + v_{y_2} t_2$.

Progress : $[0, 256)^2_{\mathbb{N}} \times ([0, 256)^2_{\mathbb{N}})^+ \mapsto [0, 1]_{\mathbb{R}}$ measures the amount of progress of a point p along a polygonal line. Let p' be the point in the line closest to p. The progress is the length of the line from the first point of the line to p' divided by the total length of the line. However, if p is too far from the line ($\textsc{Dist}(p, p') > 10$), the progress is 0.

Finally, $p[0]$ and $p[1]$ are sequences of points determining a polygonal line in the direction we want Joe to move in, before and after getting the key, respectively. Note that the end of $p[0]$ coincides with the start of $p[1]$, reversed.

[2] The subscript $\mathbb{R}$ means the interval is defined over the real numbers. If no subscript is in the interval, assume it is over $\mathbb{N}$.

Algorithm 1 The potential function for shaping.

```
function $\textsc{Dist}(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle)$
    return $\sqrt{(x_2 - x_1)^2 + \left(\frac{(y_2-y_1)}{2}\right)^2}$
end function
function $\textsc{Project}(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \langle x'_2, y'_2\rangle)$
    $v_{x_2} = x'_2 - x_2, \quad v_{y_2} = y'_2 - y_2$
    $v_{x_1} = -v_{y_2}, \quad v_{y_1} = v_{x_2}$
    return $(v_{x_1}(y_2 - y_1) - v_{y_1}(x_2 - x_1)) \,/\, (v_{y_1}v_{x_2} - v_{x_1}v_{y_2})$
end function
function $\textsc{Progress}(\langle x, y\rangle, [p_1 = \langle x_1, y_1\rangle, p_2 = \langle x_1, y_1\rangle, \ldots, p_n])$
    $\forall i \in [1, n],\ t_i = 0$
    $\forall i \in [1, n),\ (p_{n+i}, t_{n+i}) = \textsc{Project}(\langle x, y\rangle, p_i, p_{i+1})$
    $m = \arg\min_{i\in[1,2n) \text{ s.t. } t_i \leq 1} \textsc{Dist}(\langle x, y\rangle, p_i)$
    if $\textsc{Dist}(\langle x, y\rangle, p_m) > 10$ then
        return $0$
    end if
    $\forall j \in [1, n],\ \text{len}_j = \sum_{i=1}^{j-1} \textsc{Dist}(p_i, p_{i+1})$        ▷ Note that $\text{len}_1 = 0$
    if $m \leq n$ then
        return $\text{len}_m \,/\, \text{len}_n$
    else
        return $(\text{len}_{m-n} + t_m \cdot (l_{m+1} - l_m)) \,/\, \text{len}_n$
    end if
end function
function $\textsc{Phi}(v_3, v_4, v_5, v_6)$
    return $\begin{cases} 0 & \text{if } v_6 = 1 \\ \textsc{Progress}(\langle v_3, v_4\rangle, p[v_5]) & \text{otherwise} \end{cases}$
end function
$p[0] = [\langle 100, 201\rangle, \langle 133, 201\rangle, \langle 133, 148\rangle, \langle 21, 148\rangle, \langle 21, 192\rangle, \langle 9, 207\rangle]$
$p[1] = [\langle 9, 207\rangle, \langle 21, 192\rangle, \langle 21, 148\rangle, \langle 133, 148\rangle, \langle 133, 201\rangle, \langle 72, 201\rangle, \langle 72, 251\rangle, \langle 153, 251\rangle]$
```

`img/shaping.pdf`

Figure 3.2: The potential function ϕ field used for shaping. From left to right: $v_5 = 0$, $v_5 = 1$, reference screenshot from the game. Vertical axis is v_4, horizontal axis is v_3. Yellow is $\phi(\cdot) = 2$, deep purple is $\phi(\cdot) = 1$.

3.2.3 Options

We created options to test with Semi-Markov Decision Process (SMDP) Sarsa. We have 8 options, that correspond to the 8 possible minimal actions.

- NOOP: Take action NOOP during *frame_skip* frames. Primitive actions in Atari games treated as SMDP also last *frame_skip* frames, so this is just the primitive NOOP action.
- UP, DOWN, LEFT, RIGHT: the normal directions are followed until:
  - Their coordinate (y for UP and DOWN, x for the rest) stops changing for a long enough time. For example, when Joe bumps into a wall.
  - It wasn't possible to move in the other axis when the action started and it is possible now, or vice versa. This is implemented by generating tentative moves in the other axis every *frame_skip* frames, and checking if their x, y coordinates are different.
  - Joe will lose a life or start falling in $n_{\text{backtrack}} \cdot frame\_skip$ frames. This is implemented by backtracking generated states when these things happen, in a similar manner to function OBSTACLE-WAIT, from Algorithm 3.
    * If the option starts when Joe is already in mid-air, it behaves like the NOOP option.
- FIRE, LEFTFIRE and RIGHTFIRE: Take the corresponding action once, then take action NOOP until the character lands again.

It is possible to take these actions at all times, their initiation set, $\mathcal{I}$, is the set of all states, $\mathcal{S}$.

3.3 Planning

was first applied to Atari games by Lipovetzky, Ramirez, and Geffner, (2015). They used IW(1) only (Subsection ??) in an on-line setting (Subsection ??). Since index˙cmd operates only on Boolean variables, they convert each of the index˙cmd's memory positions to 256 variables, one for every possible value of the byte.

We downloaded, read and run their kindly provided implementation[3]. Their agent behaved erratically until it got to the bottom floor, past the skull or without the

[3]https://github.com/miquelramirez/ALE-Atari-Width

skull. Then, it made a beeline for the key.

3.3.1 Width of Montezuma's Revenge

A successful player of MR must visit the same location several times. She needs to take certain paths and backtrack them, to wait for obstacles to cycle between passable and impassable, to take into account doors that may or may not be opened and the contents of her inventory.

At least, the path to the solution will involve being at a certain location, for more than one step of time. Thus, the search algorithm must not prune a state when the time (as given by, for example, memory position 0x80) changes and the position does not.

Location of Joe can be represented by the contents of the memory positions $\langle \text{0xAA}, \text{0xAB}, \text{0x83} \rangle$, that is, x, y, and screen location. This, combined with position 0x80, intuitively suggests that MR has a *width* of at least 4.

The authors of index˙cmd discarded applying a higher width than 1 to Atari games because the number of tuples to record is too large. We get around this limitation using domain knowledge: we prune only on the 3-tuple representing location. All the other memory positions are considered to not change in value. Henceforth, we will refer to this algorithm as "index˙cmd(3)" or "index˙cmd(3) on position", even the original index˙cmd(3) would prune far less often.

index˙cmd(3) on position prunes a movement when Joe does not move to a different place. Thus, it allows for exploring the whole screen, while pruning several redundant moves such as applying different actions while in the middle of a jump. As shown in Figure 3.3, this makes for much better exploration of the environment.

How is it possible that we can use index˙cmd(3) on a problem with width ≥ 4? The key lies in the on-line setting. Rather than looking for all the paths until the end, the algorithm only explores to a certain point and then picks an action. Thus, focusing on spatial exploration works relatively well (Section 4.1).

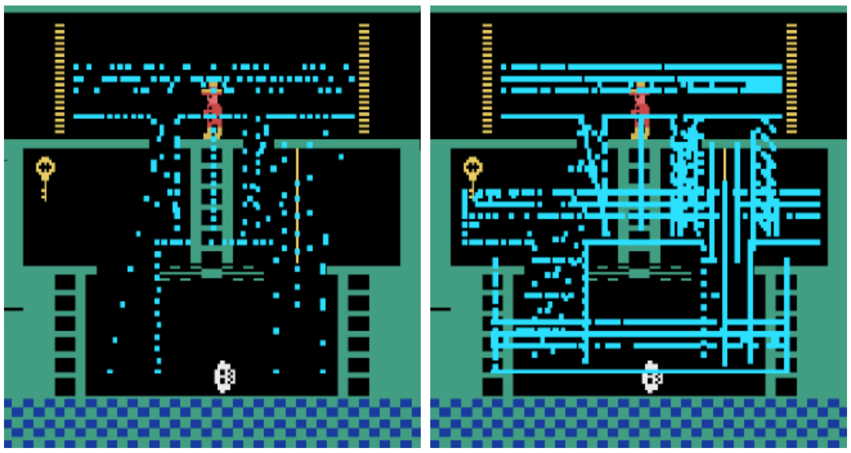
Figure 3.3: Comparison of exploration in the first screen by IW(1) and IW(3) on position. Observe that IW(1), to the left, prunes all paths that move to the right-middle platform, since their y coincides with the the central platform and their x coincides with the right-top platform. Our restricted IW(3) only prunes for repeated positions, so it has no problems finding the key.

3.3.2 Improving score with domain knowledge

Caring about life

This one is noted and suggested by Lipovetzky, Ramirez, and Geffner, (2015). Since the death of Panama Joe does not reduce the score, the algorithm dies often just to instantly move to a desired location. This would be fine if the agent never lost a life unintentionally. However, by the nature of its over-pruned planning, this is not the case.

index˙cmd, like (index˙cmd) (Algorithm **??**), has a single (index˙cmd) queue as frontier. We add another, low-priority, queue, that is only used when the first queue is empty. In the low priority queue, we put the nodes where Panama Joe has lost a life.

The agent still dies unnecessarily sometimes, when the agent has explored first sequences of actions that lead to reward and death. In some of those cases, this causes the nonlethal ways to get to the score to be pruned too early.

Incentivising room exploration

More often than not, our agent would follow a path of rewards to the bottom floor of the pyramid (room 20 or 23, Figure 3.1), and then be stuck there, not finding any positive rewards. Thus, we added a small reward (+1) to exploring new screens.

This created perverse incentives. The agent often enters room 5 from room 6, or room 17 without having any keys, and then immediately leaves, unable to obtain more score in there. However, overall, it helps performance.

Randomly pruning rooms

In each tree expansion, upon each first visit to a room (except the first one), that room is pruned with a certain probability. When a room is pruned, we prune all the nodes that end in that room. On some expansions this makes the agent explore farther.

On unlucky tree generations, the agent may find the return of all the actions being zero. We mitigate this by, after expanding the tree, not just taking the first action of the branch with highest return, but storing the whole branch. After each generation, the new branch is compared with the previous one, and the one with the most return is followed and stored for the next expansion.

Prioritising long paths

Ties in branch return are broken by length of the branch. Except for the first action, where ties are broken uniformly randomly.

Overriding pruning near timed obstacles

The basic intuition is: when losing a life because of running into an obstacle that will disappear after some time, wait.

In practice, this means:

- Lose a life after walking into the obstacle, not jumping.
- Backtrack to a position “outside” the obstacle, that allows you to survive until the same time instant.
- Wait until the obstacle goes away, by testing moves into the direction of the direction you backtracked from, or a maximum time to wait.
- Add the resulting node to the frontier.

Preventing short-sighted door opening

A fundamental shortcoming of our agent is that it opens doors not to explore what is behind them, but because doing so increments the score. As a consequence of this (and of shortsightedness), it opens both the doors in each screen 1 and 5, rendering the game impossible to complete (as explained in Subsection 3.1.1). We penalised this behaviour by giving a penalty to opening the wrong doors ($-10\,000$).

3.3.3 Implementation

Our agent is described in Algorithms 3, 4 and 2. Online-Setting-Episode in Algorithm 2 is the entry point.

Each time step, we reuse the search tree created in the last time step. Emulating frames is the most computationally expensive step in the search, so we restrict each time step to emulating max_{ef} frames (Lipovetzky, Ramirez, and Geffner, 2015).

As an efficiency improvement, we can take several actions of the planned sequence before re-calculating the search tree. Since our algorithm is neither optimal nor complete, this may degrade or enhance performance.

$s[i]$ is used to denote the index˙cmd position i in node s. When a node s is created from the transition function f, s.Return contains the reward accumulated while emulating the action in s.

The transition function $s' = f(s, a)$ is implemented in the ALE by first loading the state s to the emulator, then applying action a for $frame\_skip$ frames. In the end we observe the resulting state s', with its reward. The $frame\_skip$ constant is very commonly used for playing Atari games, since the state changes little every frame (Bellemare et al., 2013, Lipovetzky, Ramirez, and Geffner, 2015, Mnih et al., 2015, Kulkarni et al., 2016, . . .).

Noop, Left, Right, Up and Down are actions the character can take, corresponding to standing still and moving in a certain direction without jumping, respectively.

Algorithm 2 The agent in an on-line setting, using index˙cmd(3) for Montezuma's Revenge, along with some supporting functions for index˙cmd(3).

```
procedure ONLINE-SETTING-EPISODE
    Initialise action and return 1-based-index sequences, $a \leftarrow []$, $R \leftarrow []$
    $n_a$: number of actions to take without re-planning
    $p_r$: probability that a room is pruned
    $f$ is the emulation function
    $max_{ef}$ maximum number of frames to emulate per search
    $max\_wait$: max. n. of actions to wait for an obstacle to become passable
    $max\_backtrack$: max. n. of nodes to backtrack when running into an obstacle
    repeat
        Observe state $s$.
        $s$.RETURN $\leftarrow 0$
        $(a', R') \leftarrow$ IW3$(\langle s, \mathcal{A}, f\rangle, max_{ef}, max\_wait, max\_backtrack, p_r)$
        if $R = [] \lor R'[1] > R[1]$ then
            $R \leftarrow R', a \leftarrow a'$
        end if
        for $i$ from 1 to $\min(n_a$, LENGTH$(R))$ do
            Take action $a[i]$, observe and tally reward
        end for
        $R \leftarrow R'[n_a + 1, n_a + 2, \dots]$, $a \leftarrow a'[n_a + 1, \dots]$
    until the game is over
end procedure
function BRANCH-RETURN($s$)
    if $s$ is a leaf node and has opened a wrong door, $s$.RETURN $\leftarrow -10\,000$ end if
    if $s$ is a leaf node return [$s$.RETURN] end if
    $r_s$ = BRANCH-RETURN$(c)\,\forall c \in s$.CHILDREN
    $r \leftarrow \max_{r \in r_s} r$, comparing by first element, ties broken by higher length.
    return APPEND([$s$.RETURN], $\gamma \cdot r$)
end function
```

Algorithm 3 Supporting functions for index˙cmd(3) (Algorithm 4)

```
function $\textsc{Update-Novelty}(seen\_tuples, visited\_screens, pruned\_screens, ram, p_r)$
    if $\neg visited\_screens[ram[\text{0x83}]]$ then
        $visited\_screens[ram[\text{0x83}]] \leftarrow true$
        With probability $p_r$: $pruned\_screens[ram[\text{0x83}]] \leftarrow true$
    end if
    $seen\_tuples[\langle ram[\text{0xAA}], ram[\text{0xAB}], ram[\text{0x83}]\rangle] \leftarrow true$
end function
function $\textsc{Check-Novelty}(seen\_tuples, pruned\_screens, ram)$
    return $\neg(pruned\_screens[ram[\text{0x83}]] \vee$
                    $seen\_tuples[\langle ram[\text{0xAA}], ram[\text{0xAB}], ram[\text{0x83}]\rangle])$
end function
function $\textsc{On-Ground?}(ram)$
    return $ram[\text{0xD6}] = \text{0xFF} \wedge ram[\text{0xD8}] = 0$
end function
function $\textsc{Falling?}(ram)$
    return $ram[\text{0xD8}] \neq 0$
end function
function $\text{LRUD?}(a)$
    return Whether the action $a \in \{\textsc{Left}, \textsc{Right}, \textsc{Up}, \textsc{Down}\}$.
end function
function $\textsc{Obstacle-Wait}(f, obstacle\_child, q, max\_wait, max\_backtrack)$
    $p \leftarrow obstacle\_child.\textsc{Parent}$, $p_p \leftarrow obstacle\_child$, $l_0 \leftarrow p[\text{0xBA}]$
    for $i \in [0, max\_backtrack)$ do
        $f^i(s,a) = f(f(\dots f(s,a)\dots,a),a)$, totalling $i+1$ applications of $f$
        $n \leftarrow f^i(p, \textsc{Noop})$
        if $\textsc{On-Ground?}(n) \wedge n[\text{0xBA}] = l_0$ then
            for $i \in [0, max\_wait)$ do
                $n_{\text{test}} \leftarrow f^2(n, p_p.\textsc{Action})$
                if $\textsc{On-Ground?}(n_{\text{test}}) \wedge n_{\text{test}}[\text{0xBA}] = l_0$ then
                    return $\textsc{Queue-Insert}(q, n)$
                end if
                $n \leftarrow f(n, \textsc{Noop})$
            end for
        end if
        $p_p \leftarrow p$, $p \leftarrow p.\textsc{Parent}$
    end for
    return $q$
end function
```

Algorithm 4 index˙cmd(3) for Montezuma's Revenge, optimised with domain knowledge

```
function IW3($problem = \langle s_0, \mathcal{A}, f\rangle, max_{ef}, max\_wait, max\_backtrack, p_r$)
    $seen\_tuples \leftarrow false$ for all tuples $[0,256)^2 \times [0,24)$
    $visited\_screens[i] \leftarrow false$, $pruned\_screens[i] \leftarrow false$, $\forall i \in [0,24)$
    $visited\_screens[s_0.ram[\text{0x83}]] \leftarrow true$
    $q \leftarrow [s_0]$, $q_l \leftarrow []$, index˙cmd queues representing the frontier
    while $\neg$EMPTY?($q$) $\wedge$ $\neg$EMPTY?($q_l$) $\wedge$ num. emulated frames $< max_{ef}$ do
        Get $s \leftarrow$ POP($q$), or POP($q_l$) if $q$ is empty.
        $obstacle\_child \leftarrow \varnothing$
        for each child $c = f(s,a)\, \forall a \in \mathcal{A}$ do
            if CHECK-NOVELTY($seen\_tuples, pruned\_screens, c$) then
                UPDATE-NOVELTY($seen\_tuples, visited\_screens, pruned\_screens, c, p_r$)
                if $c[\text{0xBA}] < s[\text{0xBA}]$, this node loses a life then
                    if ON-GROUND?($c$) $\wedge$ ON-GROUND?($s$) $\wedge$ LRUD?($a$) then
                        $obstacle\_child \leftarrow c$
                    end if
                    $q_l \leftarrow$ QUEUE-INSERT($q_l, c$)
                else
                    if FALLING?($c$) $\wedge$ ON-GROUND?($s$) $\wedge$ LRUD?($a$) then
                        $obstacle\_child \leftarrow c$
                        if $a =$ DOWN then $q \leftarrow$ QUEUE-INSERT($q, c$) end if
                    else
                        $q \leftarrow$ QUEUE-INSERT($q, c$)
                    end if
                end if
            end if
        end for
        if $obstacle\_child \neq \varnothing$ then
            $q, \leftarrow$ OBSTACLE-WAIT($f, obstacle\_child, q, max\_wait, max\_backtrack$)
        end if
    end while
    return BRANCH-RETURN($s_0$)
end function
```

Chapter 4

Evaluation

4.1 Planning

We evaluated the domain-specific planning algorithm described in Algorithm 4, with different subsets of enhancements and different parameters. We used the ALE, with deterministic games (`repeat_action_probability=0`). Our code is based in the one available from Lipovetzky, Ramirez, and Geffner, (2015).

Recall that our index˙cmd(3) only evaluates position for pruning, not on any other tuple.

- max_{ef}: maximum nodes emulated per frame.
- γ: The discount factor.
- n_a: number of actions to take without re-planning.
- p_r: probability that a room is pruned. Blank means zero.
- *FS*: frame skip, the amount of frames each action is taken for.
- *TR*: Whether the search tree is reused, or the nodes are re-emulated in every frame.
- Frontier: the data structure/s used to store the frontier:
  - q: a single index˙cmd queue, like index˙cmd and index˙cmd.
  - q, q_l: two index˙cmd queues, one with a lower priority.
  - P. Dist.: Priority queue that prioritises nodes more distant (in game coordinates, Euclidean distance) to the root.

  – 2BFS Two priority queues: one prioritising low novelty, breaking turns by largest accumulated return, and one prioritising large accumulated return, breaking ties by lowest novelty (Lipovetzky, Ramirez, and Geffner, 2015).

  index˙cmd queue, 2 index˙cmd queues, or a priority queue.

- *EB*: Exploration Bonus. Whether the agent gains 1 reward on exploring a new screen.
- *OAV*: Obstacle Algorithm Version. The algorithm that waits for obstacles to disappear has some variants. Blank means the absence of such thing. Version 1 is the nodes that lead into an obstacle are re-enqueued in the frontier. Versions between 1 and 2 are other semi-successful modifications of it. Version 2 is the one explained in Subsections 3.3.2 and 3.3.3.
- *EA*: Extended Action set. Whether the algorithm uses the 18 actions possible with the Atari or the 8 distinct actions in MR.

Additionally, algorithms with an asterisk (*) receive negative rewards (-5000) on death. The remaining parameters' values are: $max\_wait = 20$, $max\_backtrack = 7$.

Name	max_{ef}	γ	n_a	p_r	FS	TR	Frontier	EB	OAV	EA	Score
index˙cmd(1)	150k	0.995	1		5	✓	q			✓	0
2BFS	150k	0.995	1		5	✓	2BFS			✓	540
index˙cmd(3)*	150k	0.995	1		5		q				4600
index˙cmd(3)*	1 500k	0.995	1		5		P. Dist.		1		2 500
index˙cmd(3)*	300k	0.995	1		5		q, q_l	✓	1		5 600
index˙cmd(3)*	300k	0.995	1		5		q, q_l	✓	1.1		8 000
index˙cmd(3)*	150k	0.99	1		10		q, q_l	✓	1.1		10 200
index˙cmd(3)*	75k	0.985	1		10		q, q_l	✓	1.1		0
index˙cmd(3)*	10k	0.98	1		20		q, q_l	✓	1.1		0
index˙cmd(3)	300k	0.995	1		6		q, q_l		1.2		100
index˙cmd(3)	300k	0.999	1		5		q, q_l		1.2		500
index˙cmd(3)	300k	0.995	1		5		q, q_l		1.2		6 700
index˙cmd(3)	300k	0.999	1		5		q, q_l	✓	1.2		7 100
index˙cmd(3)	300k	0.999	1	0.25	5		q, q_l	✓	1.2		4 700
index˙cmd(3)	300k	0.99	1	0.2	5		q, q_l	✓	1.2		11 000
index˙cmd(3)	300k	0.99	1	0.2	5		q, q_l	✓	2		13 600
index˙cmd(3)	150k	0.99	2	0.2	10	✓	q, q_l	✓	2		14 500
index˙cmd(3)	150k	0.995	2	0.2	5	✓	q, q_l		2		8 000
index˙cmd(3)	150k	0.995	2	0.2	5	✓	q, q_l	✓	2	✓	7 800
index˙cmd(3)	150k	0.999	3	0.2	5	✓	q, q_l	✓	2		14 900

Table 4.1: The results of different planning algorithm variations

The algorithm described in Subsection 3.3.2 obtains the same score as the latest one, but it avoids opening the two doors. Instead, it finds a glitch in the game that allows it to spend the two keys without a door. A video of it is available online. The glitch happens around 2:52.

To obtain this massive increase in score, we have heavily tweaked the algorithm to this game. The strategies employed will likely not generalise to all classes of problems. Some of them might be useful for games that happen in a 2D spatial environment, such as pruning on position, waiting for obstacles. The random room pruning heuristic may also be useful in other problems in the on-line setting that demand exploring multitudes of similar paths.

4.2 Learning

We trained agents on the first screen of Montezuma's Revenge using the Sarsa algorithm, with and without options, and using our shaping function (Subsection 3.2.2).

We used a learning rate $\alpha = 0.01$ and a discount rate $\gamma = 0.995$. We also used the *annealing* training technique, which consists in reducing the ε for the ε-greedy strategy every episode. When training without annealing we used $\varepsilon = 0.1$, and when training with annealing we used $\varepsilon = \textsc{Max}(0.7 - 3 \cdot 10^{-5} \cdot n_e, 0.1)$, where n_e is the episode number. This encourages extra exploration at the beginning.

The average reward over time can be seen in Figures 4.1 and 4.2. In each of those figures, at the left the reward including the shaping reward is shown, and at the right the environment reward alone is shown.

There is something odd about all the graphs. First, the graph that does not use options (Figure 4.1), actually *decreases* in accumulated reward during the first $\sim 25\,000$ episodes. However, the accumulated reward without accounting for the shaping function is almost monotonically increasing. We suspect that the return in the starting state is also monotonically increasing, and it is the unique form of the shaping function $F(s, a, s') = \gamma\phi(s') - \phi(s)$ that permits this to happen.

It is also somewhat worthy of note that the annealed version takes longer start increasing in reward, but once it does it converges earlier. We attribute this to an early exploration of "accidental" states that makes the agent learn them thoroughly at first, and then make it have no problems when spuriously encountering them afterwards.

As for the versions with options, they do not work at all. Videos of the agent acting img/sarsa/without_options.pdf

Figure 4.1: Reward per episode, averaged every 1000 episodes, as the training progresses. No options are used. Blue uses annealing, red does not. Darker lines, with higher value, do not include the shaping reward $F(\cdot)$; lighter lines show the reward as the agent experiences it. y axis is reward, x axis is thousands of episodes.

img/sarsa/with_options.pdf

Figure 4.2: Reward per episode, averaged every 1000 episodes, as the training progresses. Options are used. Meaning of each element is the same as in Figure 4.1. Note that the two reward lines without shaping are overlapping at zero reward.

show that it learns to jump to the left, dying as fast as possible. The reward it gains in doing so is negative, and less than it gains if it jumps to the right and follows with the plan as the agents without options, but the options do not seem to fit.

4.2.1 The pitfalls of shaping

Prior to reading about shaping functions that do not vary the optimal policy, we tried to lay down a path of "pellets" that the agent would get reward for collecting, from the start to the key. We arranged them in a line, and made it so that collecting one pellet also collected all the previous ones. We tried to mitigate this by reducing the number of pellets the agent could get at once, but then it found another unexpected way to quickly grab them (Figure 4.3).

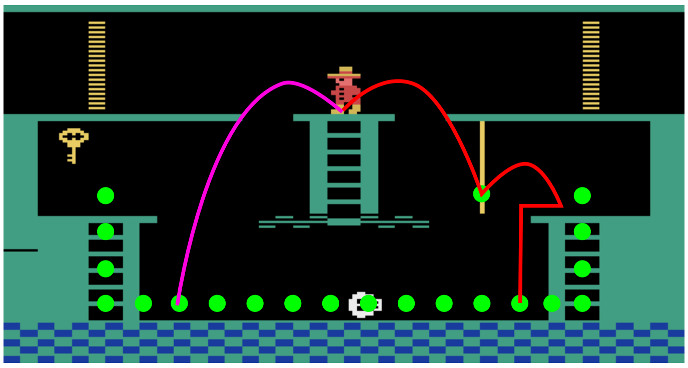

Figure 4.3: The original shaping rewards, and the two ways the agent found of defeating their purpose. First, it took the magenta path. After restricting the amount of rewards to be collected at the same time, it took the red path.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

First, we looked at the basics of Sequential Decision Processes. We learned about basic Reinforcement Learning algorithms, and ways to make them learn better: shaping and options. We then looked at search methods, especially a promising planning algorithm, .

To apply this into practice, we reverse engineered some features of Montezuma's Revenge. Using those, we crafted several methods to increase exploration, and modified index˙cmd with them to perform very well, in this problem.

We also found that reward shaping makes for fast and effective learning, and that options that do not fit well are worse than nothing.

5.2 Future work

Using planning, it was possible to find the sparse reward in the first screen from the beginning. An attractive research avenue would be to use experience acquired during planning to train a learning algorithm. Dyna-Q (Sutton and Barto, 1998, Section 9.2) is similar to this, but it would be interesting to use a better planning algorithm, and a learner with function approximation.

To do planning, the agent needs a model of the world. Often that model, unlike in this case, is not readily available. One possible avenue of research would be to try and predict the next world state from the current state and a given action.

One way to do that would be to use something similar to an *autoencoder* to learn an abstracted representation of the state, and then try to predict the abstracted representation of the next state, as did Oh et al., (2015). Ideally, that would be generalised over several platforming games, or a synthetic procedural environment that follows the laws of 2D physics.

The learner could also be supervised to learn a high-level graph-like representation of the screen, showing ladders and platforms as edges and the places where they join as nodes, for example.

Additional information for some of the above things could be had by adding features such as the current moving entities obtained, for example, with Gaussian mixture models of background (Stauffer and Grimson, 1999).

Planning algorithms that prune on novelty of the state, based on , maybe with the novelty measure in Bellemare et al., (2016), could be ran on approximate tuples in the autoencoder's state representation space, which has lower dimensionality than the input.

Bibliography

Adair, Rob (2007). *Montezuma's Revenge.* http://www.ataritimes.com/index.php?ArticleIDX=592. Last retrieved: May 3, 2018. Link to Internet Archive.

Atari 2600 Specifications. http://problemkaputt.de/2k6specs.htm. Last retrieved: May 3, 2018. Link to Internet Archive. Author pseudonym: "Nocash".

Bellemare, M. G. et al. (2013). "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47, pp. 253–279.

Bellemare, Marc G et al. (2016). "Unifying Count-Based Exploration and Intrinsic Motivation". In: *arXiv preprint arXiv:1606.01868*.

Kulkarni, Tejas D et al. (2016). "Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation". In: *arXiv preprint arXiv:1604.06057*.

Lipovetzky, Nir and Héctor Geffner (2012). "Width and Serialization of Classical Planning Problems." In: *ECAI*, pp. 540–545.

Lipovetzky, Nir, Miquel Ramirez, and Hector Geffner (2015). "Classical planning with simulators: results on the Atari video games". In: *Proc. International Joint Conference on Artificial Intelligence (IJCAI-15)*.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533.

Mott, B. et al. (1995). *Stella: a multi-platform Atari 2600 VCS emulator.* http://stella.sourceforge.net/. Last retrieved: May 3, 2018. Link to Internet Archive.

Ng, Andrew Y, Daishi Harada, and Stuart Russell (1999). "Policy invariance under reward transformations: Theory and application to reward shaping". In: *ICML*. Vol. 99, pp. 278–287.

Oh, Junhyuk et al. (2015). "Action-conditional video prediction using deep networks in atari games". In: *Advances in Neural Information Processing Systems*, pp. 2863–2871.

Russell, S. and P. Norvig (2009). *Artificial Intelligence: A Modern Approach.* 3rd. Prentice Hall Press, Upper Saddle River, NJ, USA. ISBN: 0136042597, 9780136042594.

Stauffer, Chris and W Eric L Grimson (1999). "Adaptive background mixture models for real-time tracking". In: *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.* Vol. 2. IEEE.

Sutton, Richard S and Andrew G Barto (1998). *Reinforcement Learning: An Introduction.* MIT Press.

Sutton, Richard S, Doina Precup, and Satinder Singh (1999). "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial intelligence* 112.1, pp. 181–211.

Appendix A

Figures enhanced. Learning revisited.

Since the presentation of this work on 25 July, 2016, it has been edited: most figures have been enhanced readability-wise, but its content is otherwise unchanged. In addition, the learning results have been improved by using a somewhat different methodology.

The Sarsa agent that employs the options described in Section 3.2.3, the results of which are shown in Figure 4.2, has been improved. The cause of its lack of success was indeed having too negative a reward in the direction we intended to nudge it, but that was not caused by inadequate options. Instead, the reason was that the discount rate was too low.

The options skipped no frames ($frame\_skip = 1$), as opposed to $frame\_skip = 4$ for the non-option version. Thus, the frequency of negative reinforcement by the shaping function increased, and caused the observed effect. To correct for that, we set the discount rate $\gamma = 0.9995$, which is larger than $\sqrt[4]{0.995}$ but close.

The results can be seen in Figure A.1. Observe that the rewards towards the end are much lower than in the option-less agent in Figure 4.1. Also, the non-annealed agent has trouble finding the reward consistently in the beginning, but does find it within the first few thousand episodes.

Counter-intuitively, the agent was still learning more slowly with options than without. This seemed be in great part because of the relatively infrequent updates: the option-based agent updated the Q table once per option, which usually last more than 10 frames, while the agent without options updated once every 4 frames.

`img/sarsa/with_options_9995.pdf`

Figure A.1: Same as Figure 4.2, but with $\gamma = 0.9995$. Annealed, blue uses $\varepsilon = \textsc{Max}(0.1, 0.7 - 10^{-4} \cdot n_e)$. Red uses $\varepsilon = 0.1$.

Accordingly, we changed the algorithm to update the action-value of an option every frame. Let the agent initiate an option o_t at time t, which finishes at time $t+k$. Because of how we defined our options (Section 3.2.3), we know that for all t' such that $t \leq t' < t+k$, $s_{t'} \in \mathcal{I}_{o_t}$. That is, an option can be started in any state that is reached when executing it. Thus, we can use the update in Equation (**??**) for all intermediate states t':

$$\begin{aligned} Q(s_t, o_t) &\leftarrow (1-\alpha)Q(s_t, o_t) + \alpha \left[r_{t:t+k} + \gamma^k Q(s_{t+k}, o_{t+k})\right] \\ Q(s_{t+1}, o_t) &\leftarrow (1-\alpha)Q(s_{t+1}, o_t) + \alpha \left[r_{t+1:t+k} + \gamma^{k-1} Q(s_{t+k}, o_{t+k})\right] \\ &\dots \\ Q(s_{t+k-1}, o_t) &\leftarrow (1-\alpha)Q(s_{t+k-1}, o_t) + \alpha \left[r_{t+k-1} + \gamma Q(s_{t+k}, o_{t+k})\right] \end{aligned} \tag{A.1}$$

This algorithm is very similar to the forward view k-step return Sarsa(λ) (Sutton and Barto, 1998, Sections 7.1, 7.2, 7.5). In this case, k is not necessarily equal every time an option is executed.

The results of using this improvement can be seen in Figure A.2. The blue agent, the best performing, achieves 0.5 more average reward per episode than the previously best performing option-less agent, in 1/25 the number of training episodes. It is worth noting that the option-less agent uses $\varepsilon = 0.1$ in the end, compared to $\varepsilon = 0.01$ from this agent, and that performing the wrong option has more far-reaching consequences than choosing the wrong action.

Seeing the success of this modification, we applied it back to the option-less agent. We framed each action with $frame\_skip = 4$ as an option that takes the 1-frame action for 4 frames, and applied the same algorithm. The results can be seen in figure Figure A.3. In general we observe that how fast the agent learns depends on the annealing rate, but the final performance depends only on the final ε.

`img/sarsa/with_options_each_frame.pdf`

Figure A.2: Reward per episode, averaged every 1000 episodes, as the training progresses. Options are used. Red uses $\varepsilon = \textsc{Max}(0.01, 0.2 - 10^{-3} \cdot n_e)$, blue uses $\varepsilon = \textsc{Max}(0.01, 0.2 - 10^{-4} \cdot n_e)$, amber uses $\varepsilon = 0.1$ and green uses $\varepsilon = 0.05$. All use $\gamma = 0.9995$. Darker lines, with higher value, do not include the shaping reward $F(\cdot)$; lighter lines show the reward as the agent experiences it. y axis is reward, x axis is thousands of episodes.

img/sarsa/without_options_each_frame.pdf

Figure A.3: Reward per episode, averaged every 1000 episodes, as the training progresses. No options are used. Red uses $\varepsilon = \textsc{Max}(0.01, 0.7 - 10^{-4} \cdot n_e)$, blue uses $\varepsilon = \textsc{Max}(0.01, 0.7 - 5 \cdot 10^{-5} \cdot n_e)$, amber uses $\varepsilon = \textsc{Max}(0.1, 0.7 - 10^{-4} \cdot n_e)$ and green uses $\varepsilon = 0.1$. All use $\gamma = 0.9995$. Darker lines, with higher value, do not include the shaping reward $F(\cdot)$; lighter lines show the reward as the agent experiences it. y axis is reward, x axis is thousands of episodes.