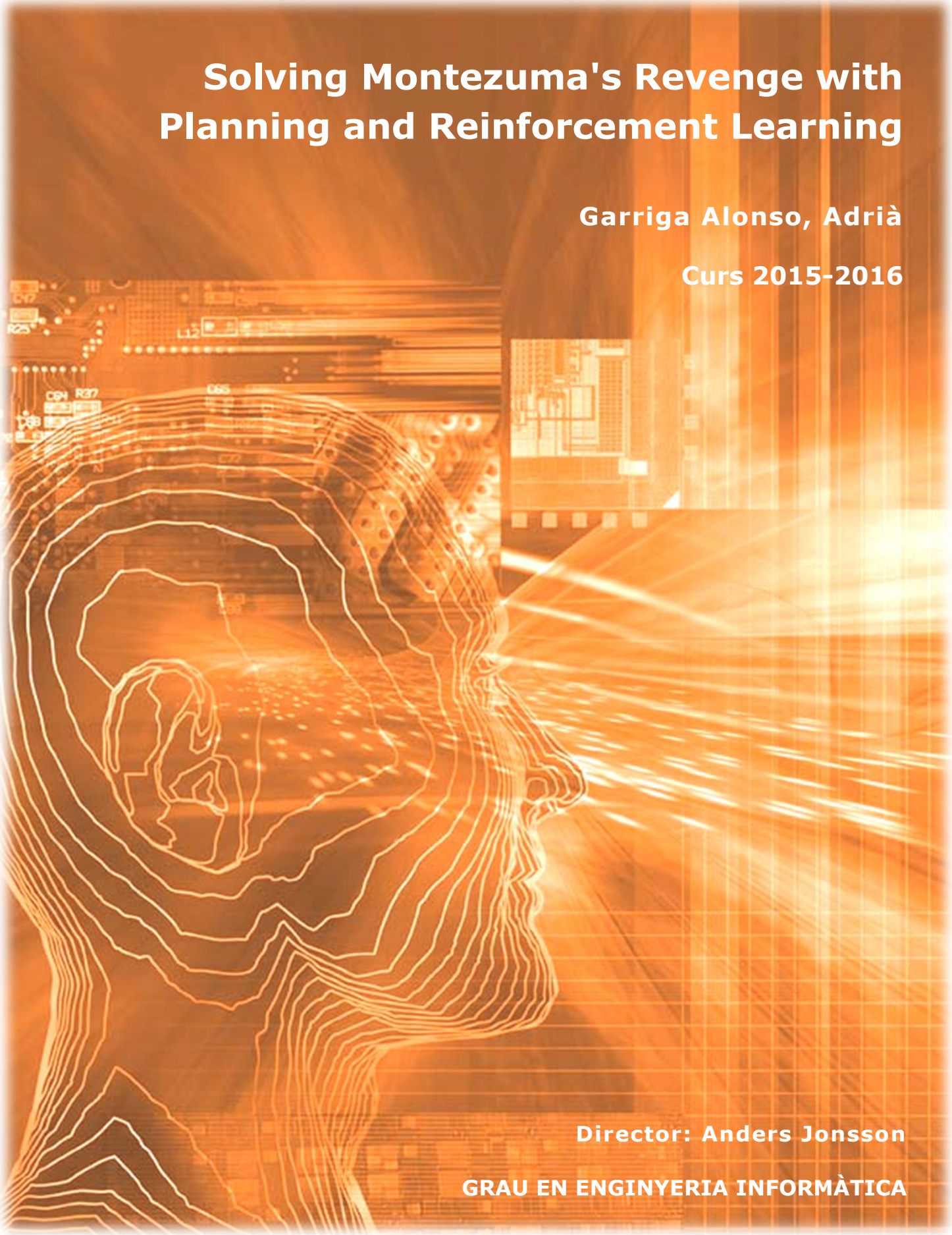
Solving Montezuma's Revenge with Planning and Reinforcement Learning

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

June 17, 2016

Bachelor in Computer Science

Department of Information and Communication Technologies

Acknowledgements

I wish to offer my thanks:

To my supervisor Anders Jonsson, for the guidance offered in navigating the literature, 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 to follow, 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. Both planning and reinforcement learning, methods for solving SDPs, have trouble with it.

With the rise to prominence of the Arcade Learning Environment (ALE) in the broader research community of sequential decision processes, one SDP featuring sparse feedback has become familiar: the Atari game Montezuma's Revenge. In this particular game, the great amount of knowledge the human player already possesses, and uses to find rewards, cannot be bridged by blindly exploring in a realistic time.

We apply planning and reinforcement learning approaches, combined with domain knowledge, to enable an agent to obtain better scores in this game.

We hope that these domain-specific algorithms can inspire better approaches to solve SDPs with sparse feedback in general.

Contents

Acknowledgements **iii**

Abstract **iv**

Contents **v**

Abbreviations **1**

1 Introduction **2**
1.1 The problem 2
1.2 Related Work 3

2 Background **5**
2.1 Sequential Decision Processes 5
2.1.1 Considerations and characteristics of SDPs 7
2.1.2 Returns 8
2.1.3 Markov Decision Processes 8
2.2 Optimal decision-making in SDPs 10
2.2.1 Policies and value functions 10
2.2.2 Optimal policy 11
2.2.3 Bellman equations 11
2.3 Reinforcement Learning 13
2.3.1 Value Iteration 13
2.3.2 Exploration-exploitation and ε-greedy policies 15
2.3.3 Sarsa 15
2.3.4 Shaping 17
2.4 Hierarchical Reinforcement Learning 17
2.4.1 Options 18
2.4.2 Semi-Markov options 18
2.4.3 Semi-Markov Decision Processes (SMDPs) 19

2.4.4 Action-value Bellman equation and Sarsa 20
2.5 Search in deterministic MDPs 20
2.5.1 Search problem formulation 21
2.5.2 Breadth First Search 23
2.5.3 Planning and Iterated Width 25

3 Methodology **28**
3.1 Montezuma's Revenge 28
3.1.1 Description 28
3.1.2 Memory layout of the Atari 2600 29
3.1.3 Reverse-engineering Montezuma's Revenge 29
3.2 Learning 34
3.2.1 State-action representation 34
3.2.2 Shaping function 34
3.2.3 Options 37
3.3 Planning 38
3.3.1 Width of Montezuma's Revenge 38
3.3.2 Improving score with domain knowledge 39
3.3.3 Implementation 41

4 Evaluation **45**
4.1 Planning 45
4.2 Learning 47
4.2.1 The pitfalls of shaping 49

5 Conclusions and Future Work **50**
5.1 Conclusions 50
5.2 Future work 50

Bibliography **53**

Abbreviations

AI Artificial Intelligence

ALE Arcade Learning Environment

BCD Binary Coded Decimal

BFS Breadth First Search

CPU Central Processing Unit

DQN Deep Q-Network

FIFO First In First Out

IW Iterated Width

MDP Markov Decision Process

MR Montezuma's Revenge

NN Neural Network

RAM Random Access Memory

RL Reinforcement Learning

ROM Read Only Memory

SDP Sequential Decision Process

SMDP Semi-Markov Decision Process

VI Value Iteration

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 articifial 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 subproblems: 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 deals with methods intrinsic to the agent of obtaining more frequent feedback

The first, by Kulkarni et al. (2016), proposes a hierarchical model (Section 2.4) 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 2.5.3) 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 2.5.3) 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

The immediate aim of this thesis is to produce a computer program that plays Montezuma's Revenge well. This problem statement suffices for most communication purposes, but does not give us enough understanding to reason about the problem and find ways to solve it. We first need to develop a formal definition of all the notions: "to play", "Montezuma's Revenge" and "well". We also need ways to know what to do to play well. Fortunately, most of the required work has already been done, by other authors.

In this chapter we will define mathematical models for the problem we are facing. We will also formally define the algorithms we will use to tackle it, without concerning ourselves with the details of their implementation on our computing environment.

2.1 Sequential Decision Processes

In this section, we describe the Sequential Decision Process (SDP) and related models. Most of the definitions are taken from the RL reference textbook by Sutton and Barto (1998). Some are from the AI reference textbook by Russell and Norvig (2009). Concrete citations will be given after some claims, but otherwise assume the concepts are taken from the first book.

Let us describe the SDP model from Sutton and Barto (1998, Section 3.1). There is an *agent* that, every *time step*, takes an *action* in the *environment*. The environment is a process that has a *state*. When the agent takes an action, the environment's state changes. The agent also receives a *reward* when it takes an action.

More formally. There is a series of discrete time steps, $t = 0, 1, 2, 3, \ldots$, in which the

agent and the environment interact. At time step t, environment is in state $s_t \in \mathcal{S}$. $\mathcal{S}$ is the finite, but usually very large, set of possible states. The agent takes an action a_t from the set of possible actions in the state, $a_t \in \mathcal{A}_{s_t}$. In the next time step, the agent receives a numerical reward $r_{t+1} \in \mathfrak{R}$, and the environment transitions to a new state $s_{t+1} \in \mathcal{S}$.

The environment defines the set of possible states $\mathcal{S}$, the possible actions in each state, which belong to the set of all possible actions $\mathcal{A}_s \subseteq \mathcal{A}$, the reward for each state $\mathcal{S}$ and the rules for transitioning to the new state in each time step. The agent simply chooses the action a_t in each time step. The interaction between agent and environment is illustrated in Figure 2.1.

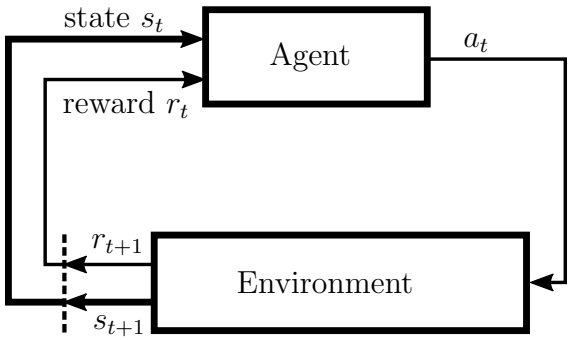


Figure 2.1: Diagram of interaction between agent and environment (Sutton and Barto, 1998, Section 3.1)

This model is really flexible. It does not constrain time steps to have the same length, so each can represent a decision branching point and not actual time. An example of this can be observed in go, chess, poker and others, where each move takes a different amount of wall clock time. It is also not constrained how a new state is chosen after an action: it may depend on anything, maybe even be stochastic and not deterministic.

The model also accepts different abstraction levels of states and actions: in a video game, they can be raw pixel data and controller input, or entity position representation and moving to a certain screen. This idea is the basis of hierarchical reinforcement learning, which is explained in Section 2.4.

It is important to understand that the agent is only the *process* that *decides* actions, not a physical object or entity. In the case of a robot, the agent is only the controlling program: the actuators, mechanisms and sensors are part of the environment. In the case of a video game, the code that emulates the world and accepts controls is the environment, and the code that trains a model or plans actions is the agent. This is

the case even if the actions to be taken are high-level, and not settings of force or torque on actuators or muscles.

Observe also that the reward is usually computed by the agent process itself, rather than given by the environment as the model description implies. However, in our formal model it is external to the agent, because the agent cannot change the reward function.

The model so far maps to two the notions we needed: "Montezuma's Revenge" is the environment, "to play" is to run the process so that the agent chooses actions.

2.1.1 Considerations and characteristics of SDPs

We may also call an SDP a *task*, when we are emphasising its nature as a problem that the agent has to solve.

In this whole work we assume all Sequential Decision Processes are *fully observable*. That is, the agent's sensations fully determine which state the environment is in. In general, that may not be the case. However, the formally defined notions cover only this case.

Finite and infinite SDPs

An SDP is *finite* if the set of states $\mathcal{S}$ and the set of possible actions, $\mathcal{A}$, are finite. Otherwise, it is *infinite*. We will treat only finite SDPs in this work.

Episodic and continuing SDPs

Sometimes it makes sense to divide a task in non-overlapping continued interactions between the agent and environment. Such tasks are called *episodic*, and they have one final time step. In contrast, *continuing* tasks never stop, in theory. (Sutton and Barto, 1998, Section 3.3)

Deterministic and stochastic SDPs

We have not yet mentioned how the next state of an SDP is determined. In general, the next state is drawn from a probability distribution over all possible states $\mathcal{S}$, that depends on the past history of actions, states and rewards.

Sometimes, the probability distribution has all its weight on a single state, that is, the next state is a function of the previous history of the process. Such SDPs are called *deterministic*. When an SDP is not deterministic, it is *stochastic*.

2.1.2 Returns

What is to play "well"? The agent's goal is, informally, to maximise the rewards it gets. In general, we maximise the expected future reward at any time step, that is, the expected *return* at time t, R_t. We could simply define R_t as the sum of all rewards until the last time step, T:

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T \tag{2.1}$$

However, we may be faced with a continuing task, and the final time step may be infinity. We could very well be faced with infinite return for each action. If we want to pick the action with maximal return, and all actions have an infinite return, we are forced to pick one at random.

Instead, we use a more general notion, that of *discounted return*:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

Where γ, the *discount rate*, is a parameter, $0 \leq \gamma \leq 1$. Observe that, if $\gamma = 1$, the return is simply the sum of all rewards, as in Equation (2.1). If $\gamma < 1$, however, we solve our infinite return problem: as the time step approaches infinity, the weight its reward is scaled by approaches zero, and R_t converges. (Russell and Norvig, 2009, Subsection 17.1.1)

2.1.3 Markov Decision Processes

The SDP formalism is not really used in practice. Markov Decision Processes (MDPs), which are a restricted case of SDPs, are used instead.

In general, the next state of an SDP may depend on all the past states, actions and rewards. A Markov Decision Process is a Sequential Decision Process that follows the Markov property (Sutton and Barto, 1998, Section 3.5; Russell and Norvig, 2009, Section 17.1), defined as follows.

$$P\{s_{t+1} = s, r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\} = P\{s_{t+1} = s, r_{t+1} = r | s_t, a_t\} \tag{2.3}$$

That is, for all $s \in \mathcal{S}$ and $r \in \mathfrak{R}$, the probability that in the next step the state is s and the reward is r is the same, whether conditioned on the whole history of past states, actions and rewards, or conditioned only on the current state and action. More concisely, the probability distribution over the next possible states and rewards depends only on the current state and action.

This property enables us to develop agents that choose an action based only on the current state: in any MDP, this decision is just as good as considering all past states, actions and rewards.

Additionally, algorithms and additional theory developed on top of MDPs can be easily adapted to any SDP. Turning an SDP into an MDP is trivial: let the state s'_t of the MDP be the sequence of current and previous states, actions and rewards of the SDP, $s_t, a_t, r_t, s_{t-1}, \ldots, s_n$. If the SDP depends on all of its history $n = 0$, otherwise we can take data until $n = t - m$. Indeed, the latter, excluding rewards, is the approach taken in Mnih et al. (2015), Kulkarni et al. (2016), and this thesis.

It is also possible for the state to encode an abstracted representation of the past actions and sensations. A repairer agent includes in its state the size of the screwdriver it grabbed a few seconds ago, not the sensations, actions and rewards it had while performing such task.

We usually specify an MDP task with the tuple $\langle \mathcal{S}, \mathcal{A}_s, \mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'}, \gamma \rangle$: (based on Sutton and Barto (1998, Section 3.6))

- The set of possible states, $\mathcal{S}$.
- The available actions for each state, as a function of the state, $\mathcal{A}_s$. Some formulations use the set of possible actions $\mathcal{A}$ instead.
- The matrix of transition probabilities from each state to another, given an action, $\mathcal{P}^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a)$.
- The expected reward given a state, an action and the next state, $\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$
- The discount factor for calculating returns, γ

Note that the model does not explicitly represent the probability distribution on

rewards, only the expectation.

The $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ matrices are usually intractably big, and indeed may be infinite if the MDP is infinite.

2.2 Optimal decision-making in SDPs

2.2.1 Policies and value functions

A *policy* π is a probability distribution for each state $s \in \mathcal{S}$, over the possible actions to take $a \in \mathcal{A}_s$. It is represented as a probability associated to each state-action pair: $P = \pi(s, a)$. We may also write $a = \pi(s)$, if the policy is deterministic: $\pi(s, a') = 1$ if $a' = a$ and $\pi(s, a') = 0 if a' \neq a$.

A *value* $V^\pi(s)$ is the expected return for an agent following the policy π, that is currently in state s. We define it as follows:

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

We can also define the *action-value function* $Q^\pi(s, a)$ of a policy π, which is the expected return from taking action a in state s and following π thereafter.

$$Q^\pi(s, a) = E_\pi\{R_t | s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \tag{2.5}$$

(Sutton and Barto, 1998, Section 3.7)

We can calculate the value of a state by doing the average of the expected values of all the actions, weighted by the probability of each action being taken. The probability of each action being taken is determined by the policy, so we get the following identity:

$$\begin{aligned} \sum_{a \in \mathcal{A}_s} \pi(s, a) Q^\pi(s, a) &= \sum_{a \in \mathcal{A}_s} \pi(s, a) E_\pi\{R_t | s_t = s, a_t = a\} \\ &= E_\pi\{R_t | s_t = s\} = V^\pi(s) \end{aligned} \tag{2.6}$$

2.2.2 Optimal policy

Some of the policies will have higher values than others. The policy with the maximum value for a state is called the *optimal policy* for that state, denoted by π_s^*. The optimal policy for a state is that which maximises its utility.

$$\pi_s^* = \arg\max_{\pi} V^{\pi}(s) \tag{2.7}$$

It is also important that the optimal policy is independent of the state it starts with, if we don't cut the return at a time-step before the final time-step (that is, the *horizon* is infinite) and we use discounted returns. So, we can just write π^* to refer to the optimal policy.

The value of a state when following the optimal policy, $V^{\pi^*}(s)$, is the *true value*, or optimal value, of a state. Thus, we will just write $V(s)$ to refer to it.

If we know the true value function of all the states and the transition model of the environment, we can calculate the optimal policy:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V(s) \tag{2.8}$$

Where ties are broken arbitrarily. Notice that we wrote it as a mapping from states to actions, and not as a mapping from states and actions to probability weights. This is because optimal policies take only one action.

(Russell and Norvig, 2009, Subsection 17.1.2)

2.2.3 Bellman equations

Both value and action-value functions satisfy a recursive relationship that is very widely used in reinforcement learning algorithms: the Bellman equations.

Starting with Equation (2.4), we separate the first reward from the sum of future

rewards to obtain the Bellman equation for values:

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t|s_t = s\} \\ &= E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle| s_t = s\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}\middle| s_t = s\right\} \\ &= \sum_{a\in\mathcal{A}_s}\left(\pi(s,a)\sum_{s'\in\mathcal{S}}\mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}\middle| s_{t+1} = s'\right\}\right]\right) \\ &= \sum_{a\in\mathcal{A}_s}\left(\pi(s,a)\sum_{s'\in\mathcal{S}}\mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right]\right) \end{aligned} \tag{2.9}$$

And let us do the same for action-values, starting with Equation (2.5):

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{R_t|s_t = s, a_t = a\} \\ &= E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}\middle| s_t = s, a_t = a\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}\middle| s_t = s, a_t = a\right\} \\ &= \sum_{s'\in\mathcal{S}}\mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}\middle| s_{t+1} = s'\right\}\right] \\ &= \sum_{s'\in\mathcal{S}}\mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right] \end{aligned} \tag{2.10}$$

(Sutton and Barto, 1998, Section 3.7)

And if we then substitute in Equation (2.6):

$$Q^\pi(s,a) = \sum_{s'\in\mathcal{S}}\mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma\sum_{a'\in\mathcal{A}_{s'}}\pi(s',a')Q^\pi(s',a')\right] \tag{2.11}$$

Bellman equations with the optimal policy

Recall from Equation (2.8) that the optimal policy takes the action that maximises the true value of the next state. So, let's put this notion into Equation (2.9):

$$\begin{aligned} V(s) &= \sum_{a \in \mathcal{A}_s} \left(\pi^*(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s') \right] \right) \\ &= \max_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s') \right] \end{aligned} \tag{2.12}$$

And Equation (2.11):

$$\begin{aligned} Q(s,a) &= \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \sum_{a' \in \mathcal{A}_{s'}} \pi^*(s',a') Q^\pi(s',a') \right] \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma \max_{a' \in \mathcal{A}_{s'}} Q^\pi(s',a') \right] \end{aligned} \tag{2.13}$$

(Russell and Norvig, 2009, Section 17.2.1, Sutton and Barto, 1998, Section 3.8)

These optimal Bellman equations are the basis of most modern reinforcement learning algorithms.

2.3 Reinforcement Learning

Unless stated otherwise, concepts in this section are taken from Sutton and Barto (1998). More concrete citations may also be given.

Reinforcement Learning (RL) is about an agent learning from experience how to behave to maximise rewards over time. This experience is usually gathered by interacting with the environment.

2.3.1 Value Iteration

Value Iteration (VI) is an algorithm, part of the Dynamic Programming collection of algorithms for RL. Those algorithms can compute optimal policies for an MDP, given a perfect model of the environment ($\mathcal{P}^a_{ss'}$, $\mathcal{R}^a_{ss'}$, and the parameter γ).

VI works by keeping a table with the values of all states, and turning the optimal value Bellman equation (Equation (2.12)) into an update rule for the table. VI is

described in Algorithm 1.

Algorithm 1 Value Iteration (Sutton and Barto, 1998, Section 4.4)

Initialize $V(s)$ arbitrarily, for example $V(s) = 0 \; \forall s \in \mathcal{S}$
repeat
  $\Delta \leftarrow 0$, is the maximum update magnitude this iteration
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right]$
    $\Delta \leftarrow \max(\Delta, |V(s) - v|)$
  end for
until $\Delta < \theta$ a small constant
Output optimal policy using Equation (2.8)

The value table kept in VI is guaranteed to converge the true value V^* under the same conditions that guarantee the existence of the latter

(Sutton and Barto, 1998, Section 4.4)

There is another variant of Value Iteration. In each outer iteration, we update only one of the values in the table. As long as none of the values stops being updated at a certain point in the computation, $V(s)$ will still converge to $V^*(s)$.

This does not decrease the amount of computation required to approximate the optimal value function. However, sweeping over all states is often infeasible, so this allows the algorithm to start making progress without having to do a single whole sweep.

We can take advantage of this, and update more often the more promising states, to be able to terminate Value Iteration earlier and still have a good enough policy. (Sutton and Barto, 1998, Section 4.5)

We could also just update the value of whatever state the agent ended up in from the previous iteration, provided that we make the agent eventually visit all states, and still converge to the optimal policy. This one of the basic ideas in the Sarsa algorithm in Subsection 2.3.3, Q-learning, Deep Q-Networks and many others similar in spirit.

2.3.2 Exploration-exploitation and ε-greedy policies

Agents that are interacting with an environment and learning while collecting rewards face the *exploration-exploitation tradeoff*. Should they take the current maximum return action, or take an action with less return, that may turn out to have a higher return when the internal value function is closer to the optimal function?

One way to deal with this tradeoff is to follow an *ε-greedy policy*. Recall from Subsection 2.2.2 that optimal policies, and "optimal" policies based on a sub-optimal value function, always take one action for any state. Instead of taking only that action, ε-greedy policies:

- Take $\pi^*(s)$ with probability $1 - \varepsilon$
- Take an uniformly randomly sampled $a \in \mathcal{A}_s$ with probability ε

Where ε is a parameter, $0 \leq \varepsilon \leq 1$. Often, $\varepsilon = 0.1$.

(Sutton and Barto, 1998, Section 2.2)

2.3.3 Sarsa

Suppose an agent knows the optimal value function, $V(s)$, and is in state s. How would it go about choosing its next action? Maybe it uses a ε-greedy optimal policy as seen in the previous section, but calculating the ε-greedy optimal policy requires calculating the optimal policy.

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} V(s) \qquad \text{(2.8 revisited)}$$

Note that we need a model of the environment, $\mathcal{P}^a_{ss'}$, as well as the value function $V(s)$. However, if we know the action-value function, we do not need a model of the environment.

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}_s} Q(s, a) \qquad (2.14)$$

For this reason, methods that learn action-value functions are called *model-free methods* (Russell and Norvig, 2009, Subsection 21.3.2).

π^ε_Q is an ε-greedy policy based on the optimal policy based on Q, as per Equation (2.14). It is possible to use other policies based on the optimal policy based on

Algorithm 2 Sarsa (Sutton and Barto, 1998, Section 6.4)

Initialize $Q(s, a)$ arbitrarily, for example $Q(s, a) = 0\ \forall s \in \mathcal{S}$
repeat for each episode
  Set s to the current, initial, state
  Choose action a for s, sampling from $a \sim \pi^{\varepsilon}_{Q}(s)$
  repeat for each step of episode
    Take action a, observe reward r, state s'
    Choose action a' for s', sampling from $a' \sim \pi^{\varepsilon}_{Q}(s')$
    $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha\left[r + \gamma Q(s', a')\right]$ (update step)
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal
end repeat

Q. However, the policy used must have a non-zero probability of choosing all the actions for convergence to the optimal policy to be guaranteed.

α is the *learning rate*, $0 \leq \alpha \leq 1$. Because we don't have the model or policy, only *samples* from them, we cannot completely update our Q following the Bellman equation. Thus, we instead *move our value towards* the Q-value based on the next one according to something analogous to the Bellman equation, but we keep $(1 - \alpha)$ of the old value and only account for the new value weighted by α.

Sarsa's name comes from the quintuple of values used in its update: $\langle s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}\rangle$.

(Sutton and Barto, 1998, Section 6.4)

Function approximation

For interesting problems, it is usually infeasible to store the Q function for all states and values. Instead, we use a learned function that approximates Q. Desirable approximate functions not only store values close to those of the states and actions the agent has seen, but also *generalise* to unseen states and actions. A very desirable method for learning such approximate functions is the use of Neural Networks (NNs), and Deep Q-Networks (DQNs) (Mnih et al., 2015) are a version of Sarsa that use NNs for approximating the action-value function.

When using function approximation in Sarsa, the only step changed is the Q update step. Instead of updating a table with the learning rate, it updates the function being learned, in a manner that depends on the function.

2.3.4 Shaping

Shaping is the practice of giving an agent intermediate rewards that are not present in the environment. They aim to make learning easier by giving the agent more frequent feedback. However, rewards added by shaping may change the behaviour of the agent from what would be the optimal behaviour with the original MDP. Indeed, this happened while conducting naive learning experiments with shaping for this work (Subsection 4.2.1).

The following definitions and observations are taken from, and proved in, Ng, Harada, and Russell (1999).

Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'}, \gamma \rangle$ be the MDP the agent interacts in. We change that MDP for another one, $\mathcal{M}' = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}^a_{ss'}, \mathcal{R}'^a_{ss'}, \gamma \rangle$, where $\mathcal{R}'^a_{ss'} = \mathcal{R}^a_{ss'} + F(s, a, s')$. $F : \mathcal{S} \times A \times \mathcal{S} \mapsto \mathbb{R}$ is a bounded real-valued function called the *shaping function*.

Let $\phi(s)$, $\phi : \mathcal{S} \mapsto \mathbb{R}$ be a potential function. A shaping function $F(s, a, s')$ does not alter the optimal policy if and only if it is a *potential-based function*. That is, there exists a potential function ϕ such that:

$$F(s, a, s') = \gamma\phi(s') - \phi(s) \tag{2.15}$$

Potential-based reward functions are robust: near-optimal policies in $\mathcal{M}$ remain near-optimal policies in $\mathcal{M}'$: if $|V^\pi_M - V^*_M| < \varepsilon$, then $|V^\pi_{M'} - V^*_{M'}| < \varepsilon$.

2.4 Hierarchical Reinforcement Learning

Suppose Alice wants to eat a salad. She needs the ingredients, so, she needs to go to the grocery store. To accomplish that, she needs to get out of the house, go out the door, ... To accomplish the first, she needs to get up from the chair, get out the room, and navigate to the front door. To get up from the chair, she needs to tense her leg muscles in this way, move her arms in that way, ...

Like most if not all humans (and animals), Alice accomplishes tasks by taking large abstract actions, that are divided into actions, that in turn are divided into actions, and so on until she reaches contractions of muscle fibres.

Each sub-task can be learned and perfected individually, in all instances it is performed. For example, learning to walk is useful for going to the grocery store or

going to school, and it gets perfected every time Alice (among other things) goes to either of the two places.

2.4.1 Options

How may we encode this helpful intuition into reinforcement learning agents? Sutton, Precup, and Singh (1999) have an answer. The agent may take *options* instead of actions at every state. Options are courses of action that last one or more time steps, and follow their own policy. Options take other options and have their own policy, which can be improved every time they are taken.

An option is a triple $\langle \mathcal{I}, \pi, \beta \rangle$:

- $\mathcal{I} \subseteq \mathcal{S}$, the set of states the action can be *initiated* the set of states the action can be *initiated* in.
- $\pi(s, a)$, $\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$, the option's policy.
- $\beta(s)$, $\beta : \mathcal{S} \mapsto [0, 1]$, the probability that the option is interrupted in state s.

The policy for an option only needs to be defined for a subset $S_o \subseteq \mathcal{S}$ of the states, as we can define $\beta(s) = 1$ for $s \in S_o$. Usually also, the action can be initiated wherever its policy is defined, that is, $\mathcal{I} = S_o$

Normal actions are a special case of options, of duration 1 time step. The option corresponding to an action a is defined as follows:

- $\mathcal{I} = \{s \in \mathcal{S} | a \in \mathcal{A}_s\}$ all the states the action can be taken in.
- $\pi(s, a') = 1$ if $a' = a$, otherwise $\pi(s, a') = 0$; for all $s \in \mathcal{I}$.
- $\beta(s) = 1$ for all $s \in \mathcal{S}$.

2.4.2 Semi-Markov options

Sometimes it is desirable that actions end after a certain "timeout", as well as in certain states, to avoid agents getting stuck. This case is accommodated with *semi-Markov options*, that depend on the whole history of states, actions and rewards since they start.

Let a semi-Markov option start in time t and ends in time τ. We call the sequence $s_t, a_t, r_t, s_{t+1}, a_{t+1}, \ldots, r_\tau, s_\tau$ the *history*, denoted by $h_{t\tau}$. The set of all possible $h_{t\tau}$ is Ω

An semi-Markov option is a triple $\langle \mathcal{I}, \pi, \beta \rangle$, with only π and β differing from the Markov option case:

- $\pi(h, a)$, $\pi : \Omega \times \mathcal{A} \mapsto \mathbb{R}$, the option's policy.
- $\beta(h)$, $\beta : \Omega \mapsto [0, 1]$, the probability that the option is interrupted in state s.

Options that take other options as actions are semi-Markov, even if all the underlying options are Markov options.

2.4.3 Semi-Markov Decision Processes (SMDPs)

A SMDP is defined by:

- A set of states $\mathcal{S}$.
- A set of actions. We call it $\mathcal{O}$, because this set will be the set of possible options in our case. Being a set of options, each has some possible initial states. We denote the options available in state s as $\mathcal{O}_s$.
- An expected cumulative discounted reward after taking action $o \in \mathcal{O}$ when in state $s \in \mathcal{S}$. We denote it by r_s^o. Let $t + k$ be the random time at which o terminates. Let $\varepsilon(o, s, t)$ be the event of option o being initiated in state s at time t. Then:

$$r_s^o = E\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{k-1} r_{t+k}\right\} \tag{2.16}$$

- A well defined joint distribution of next state and transit time, $p(s, o, s', k)$, $p : \mathcal{S} \times \mathcal{O} \times \mathcal{S} \times \mathbb{N} \mapsto [0, 1]$. For our purposes, we only need $p_{ss'}^o$:

$$p_{ss'}^o = \sum_{k=1}^{\infty} p(s, o, s', k)\gamma^k \tag{2.17}$$

  This describes the likelihood of reaching a state s' from state s when taking option o, discounted depending on the time taken to reach it. The usefulness of this term will be apparent in the SMDP's Bellman equation (Equation (2.18)).

- The discount factor γ.

So we treat it as the tuple $\langle \mathcal{S}, \mathcal{O}, r_s^o, p_{ss'}^o, \gamma \rangle$.

2.4.4 Action-value Bellman equation and Sarsa

The Bellman equation for action-values in SMDPs ends up looking very similar to the one for MDPs.

$$Q^\pi(s,o) = r_s^o + \sum_{s'} p_{ss'}^o \sum_{o' \in \mathcal{O}_{s'}} \pi(s',o') Q^\pi(s',o') \tag{2.18}$$

The factor $p_{ss'}^o$ is useful because it incorporates both the probability of reaching a state and the discount its action-value would incur. Thus, it is exactly what the Q-value of the state we arrive in should be weighted by.

And of course, we take the maximum action-value if we are defining the optimal policy:

$$Q^\pi(s,o) = r_s^o + \sum_{s'} p_{ss'}^o \max_{o' \in \mathcal{O}_{s'}} Q^*(s',o') \tag{2.19}$$

The Sarsa update looks like this (analogous to the Q-learning update from Sutton, Precup, and Singh (1999, Section 3.2)):

$$Q(s_t,o_t) \leftarrow (1-\alpha) Q(s_t,o_t) + \alpha \left[r_{t:t+k} + \gamma^k Q(s_{t+k}, o_{t+k}) \right] \tag{2.20}$$

Where s_t, o_t are the currently selected state and option, s_{t+k} and o_{t+k} are the next selected state and option, and k is the number of time steps between s_t and s_{t+k}. $r_{t:t+k}$ is the cumulative discounted reward over the indicated time range.

Note that all these expressions reduce to their ordinary MDP counterparts when the option corresponds to a primitive action.

2.5 Search in deterministic MDPs

It can be desirable for the agent to act "well" on the first try, without having to interact with the environment and learn by trial and error. If the agent has a model of the environment, this becomes possible.

This is effectively what Value Iteration does (Subsection 2.3.1). However, if the MDP has a deterministic transition model, a much more efficient class of solutions become possible: *search* algorithms.

2.5.1 Search problem formulation

A problem can formally defined by the tuple $\langle \mathcal{S}, s_0, \mathcal{A}_s, f, \mathcal{S}_G, c \rangle$:

- The set of states $\mathcal{S}$.
- The actions available in each state $\mathcal{A}_s$.
- An initial state $s_0 \in \mathcal{S}$.
- A deterministic transition function $f(s, a)$, $f : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$.
- A non-empty set of goal states $\mathcal{S}_G \subseteq \mathcal{S}$. Often defined with a function that tests if a state is in $\mathcal{S}_G$.
- A step cost function $c(s, a, s')$, $c : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$.

Starting from the initial state s_0, the agent must find a *solution*. A solution is a sequence of actions actions $a_1, a_2, \ldots, a_n$ that "leads to the goal". Since the transition function is deterministic, a sequence of actions always brings the agent to the same state. Thus, the sequence of actions generates the sequence of states $s_1, s_2, \ldots s_n$, where $s_n = f(s_{n-1}, a_n)$. That the sequence "leads to the goal" means that $s_n \in \mathcal{S}_G$.

The transition function f can be seen as defining a directed graph: the possible states are the nodes, and the actions are directed edges. Any possible sequence of states and actions is a path of this graph, so such sequences are also called *paths*. We can see the step cost function as a weight on each edge of the graph.

If possible, we want an agent to find an *optimal solution*, that is, one where the path has minimal cost. The cost of a path is the sum of the costs of all the state transitions taken, that is:

$$C(s_0, \ldots, s_n) = \sum_{i=0}^{n-1} c(s_i, a_{i+1}, s_{i+1}) \tag{2.21}$$

With this definition of path cost, we can view a search problem as finding a minimum weight path from s_0 to any state in $\mathcal{S}_G$ on the directed graph. Thus, graph minimum path search algorithms and algorithms for search problems are roughly the same. Indeed, we can use the well-known Dijkstra algorithm for finding optimal solutions to search problems.

(Russell and Norvig, 2009, Section 3.1)

Note that, since the transition is deterministic, we can write without loss of information $c(s, a) = c(s, a, f(s, a))$. Also, often the step cost is just $c(s, a) = 1 \forall s, a$, so the

path cost is the path length.

Analogy with MDPs

We can draw direct analogies between each element of a search problem and each element of an MDP. Search problems can be seen as a special case of MDPs. Reducing a search problem to an MDP means that we can create an MDP formulation such that an optimal policy for that MDP is also an optimal solution for the search problem. Reducing an MDP to a search problem is analogous.

For the following discussion, recall the formalisation of MDPs given in Subsection 2.1.3. Also remember the types of SDP from Subsection 2.1.1, which apply to MDP as well.

The set of states $\mathcal{S}$ and the actions available in each state $\mathcal{A}_s$ are directly analogous. The initial state s_0 is very clear in episodic MDPs, and even in continuing MDPs the agent has to start interacting in some state. For the transition model, we can define $\mathcal{P}^a_{ss'} = 1$ if $s' = f(s, a)$ and otherwise $\mathcal{P}^a_{ss'} = 0$.

In MDPs, we seek to *maximise* an expected reward function $\mathcal{R}^a_{ss'}$. In search problems, we seek to *minimise* a cost function $c(s, a, s')$. We can reduce a search problem to an MDP by defining $R^a_{ss'} = C - c(s, a, s')$, where C is a constant. C can be 0 if we allow negative rewards or costs, or it can be a number that bounds the cost function $C \geq c(s, a, s') \forall s, s' \in \mathcal{S}, a \in \mathcal{A}$. We can reduce a deterministic MDP to a search problem by doing the inverse procedure: $c(s, a, s') = C - R^a_{ss'}$ with C upper-bounding $R^a_{ss'}$.

We can account for the discount rate γ in the MDP we are reducing by slightly modifying the search problem. We can redefine the path cost to be:

$$C(s_0, \ldots, s_n) = \sum_{i=0}^{n-1} \gamma^i c(s_i, a_{i+1}, s_{i+1}) \tag{2.22}$$

We need only deal with the goal states now, $\mathcal{S}_G$. If we convert a search problem into an MDP, we can make it an episodic MDP and terminate it whenever a goal state would reached. But what if the MDP is continuing?

The on-line setting

A solution to a search problem is a path that starts in the initial state and ends in the goal. Therefore, algorithms that solve search problems *cannot terminate before*

reaching a goal state. Thus, we cannot somehow create a search problem without a goal and solve it.

We can instead use the *on-line setting* (used in Lipovetzky, Ramirez, and Geffner (2015), original source unknown). At each time step of the would-be continuing MDP, a new planning problem is created. Optimal paths to the goal are searched, but there is no goal. After a set amount of time, the search algorithm is terminated. The first action of the path with the least cost (so, the most reward) is taken.

This approach can also be used for episodic MDPs where the goal may be too far to be tractable with a certain search algorithm.

2.5.2 Breadth First Search

Breadth First Search (BFS) is one of the basic algorithms for solving search problems. The algorithm is breadth-first tree traversal, but adapted to graphs in general. We show it in Algorithm 3.

The algorithm uses the following strategy: first it expands the root node, then each of its successors, then each of the successors' successors, ... At each iteration, it expands the shallowest node that in the frontier, with ties broken by order of expanding them. To *expand* a node is to check if any of its children is a goal and add them to the frontier. The *frontier* is a data structure, in this case a First In First Out (FIFO) queue, that keeps the nodes we will expand in the future.

BFS is an instance of an *uninformed search* (or *blind search*) algorithm. It has no information about states beyond what is provided in the problem definition. This is in contrast to *informed* or *heuristic* search algorithms, that have some domain knowledge about which expanded nodes are more promising.

Note that BFS always finds a solution (it is *complete*), if there is one and it is not terminated prematurely. It is only an optimal solution if the path cost is a non-decreasing function of path length, which is true when all actions have the same cost. Thus, BFS is optimal in these conditions. The space and time complexity of BFS are $O(b^d)$, where b is the *branching factor*, or number of possible actions at each node, and d is the *depth* that is explored.

(Russell and Norvig, 2009, Sections 3.3, 3.4)

Algorithm 3 Breadth First Search (Russell and Norvig, 2009, Sections 3.3, 3.4)

```
function SOLUTION(node)
    if node.ACTION = ∅ then return an empty list
    end if
    s ← SOLUTION(node.PARENT)
    return LIST-CONCAT(s, node.ACTION)
end function
function CHILD-NODE(problem, parent, action)
    return a node with:
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
end function
function BREADTH-FIRST-SEARCH(problem)
    node ← a node with:
      STATE = problem.INITIAL-STATE, PATH-COST = 0,
      PARENT = ∅, ACTION = ∅
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    end if
    frontier ← an empty FIFO queue
    frontier ← QUEUE-INSERT(frontier, node)
    explored ← an empty set
    loop
        if EMPTY?(frontier) then return failure
        end if
        node ← POP(frontier)              ▷ Get the shallowest node in frontier.
        explored ← SET-INSERT(explored, node)
        for each action in problem.ACTIONS(node.STATE) do
                                          ▷ Expand each of the node's children.
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if GOAL-TEST(problem, child.STATE) then
                    return SOLUTION(child)
                end if
                frontier ← QUEUE-INSERT(frontier, child)
            end if
        end for
    end loop
end function
```

2.5.3 Planning and Iterated Width

The BFS search algorithm from the previous section did not use information about the structure of the states or the goal. BFS only checks if a state is equal to another and if it is a goal state. We say that the state representation is *atomic*.

Iterated Width (IW), in contrast, uses a *factored* state and goal representation. This means that the states are represented by vectors of values, and that goal checking checks conditions on those values. This may give us no more information than when checking whether an atomic state is a goal, but often goals in factored state representations check only one or two values. Search algorithms and problems with factored state representation are called *planning* algorithms and problems. (Russell and Norvig, 2009, Sections 2.4.7, 3.0)

The problem formulation

IW was introduced by Lipovetzky and Geffner (2012). For them, a planning problem is a tuple $\langle F, I, \mathcal{A}, G, f \rangle$:

- F is the set of boolean variables of the problem. Each element of F is either true or false in a given state, and a state is represented by the truth value of each of the variables. It is a representation of the finite set of states $\mathcal{S}$ of a search problem.
- I is the set $I \subseteq F$ of variables that are true in the initial state. It is thus a representation of the initial state s_0.
- $\mathcal{A}_f$ is the set of available actions in each state, that is, each possible combination of variables.
- G is the set $G \subseteq F$ of variables that are true in the goal states of the search problem, S_G.
- The state transition function f is not explicitly stated by them, but IW uses it.

Unstated is the cost step function, which is assumed to be $c(s, a, s') = 1$, so that the cost is always the path length.

The algorithm

We describe IW in Algorithm 4. IW is several successive runs of IW(i) for $i = 1, 2, \ldots$ until one of the runs returns a solution. IW(i) is a modified version of BFS, the difference is that it *prunes* some states, that is, it avoids putting them in the frontier after expanding them. IW(i) prunes a node n if its *novelty measure* is larger than i.

The novelty measure of a node is the size of the smallest *i-tuple* in it that has not been "seen" before in the search. An i-tuple is a tuple of i variables. That a tuple has been "seen" before means that, in a previously searched state (that is, one that is in the *explored* set), all of the boolean variables in that tuple have the same value as they do in the current state. See also the novelty measure example in Table 2.1.

Checking that the novelty measure is not greater than the current maximum width is carried out in the Check-Novelty function in Algorithm 4.

State	f_1	f_2	f_3	Novel tuples	Novelty
1	F	F	F	$\langle\rangle, \langle f_1\rangle, \langle f_2\rangle, \langle f_3\rangle, \langle f_1, f_2\rangle,$ $\langle f_2, f_3\rangle, \langle f_1, f_3\rangle, \langle f_1, f_2, f_3\rangle$	0
2	F	T	F	$\langle f_2\rangle, \langle f_1, f_2\rangle, \langle f_2, f_3\rangle, \langle f_1, f_2, f_3\rangle$	1
3	T	T	F	$\langle f_1\rangle, \langle f_1, f_2\rangle, \langle f_1, f_3\rangle, \langle f_1, f_2, f_3\rangle$	1
4	T	F	F	$\langle f_1, f_2\rangle, \langle f_1, f_2, f_3\rangle$	2
5	T	F	T	$\langle f_3\rangle, \langle f_1, f_3\rangle, \langle f_2, f_3\rangle, \langle f_1, f_2, f_3\rangle$	1
6	T	F	T	none	$4 = \lvert F\rvert + 1$
7	F	F	T	$\langle f_1, f_2, f_3\rangle$	3

Table 2.1: Example that shows the novelty measure for each new state, assuming they are expanded from top to bottom. f_i are the problem's variables, $f_i \in F$.

Lipovetzky and Geffner (2012) define the *width* w of a planning problem to be the minimum i such that IW(i) finds an optimal solution for that problem.[1] Then they find, experimentally, that w is small $w \leq 2$ for many planning problems in which the goal is restricted to a single variable, that is, $|G| = 1$. Therefore, IW is quite efficient for problems with low width, since IW(i) has complexity $O(|F|^i)$.

[1] The paper, as often papers do, is actually written in the reverse direction. First, the authors formally define width, and then using that notion they craft the IW algorithm.

Algorithm 4 Iterated Width (Lipovetzky and Geffner, 2012)

function UPDATE-NOVELTY($seen\_tuples, width, state$)
  for all tuples of variables, t, of size $width$ **do**
    $seen\_tuples[t, \text{VALUE-OF}(t)] \leftarrow true$
  end for
  return $seen\_tuples$
end function
function CHECK-NOVELTY($seen\_tuples, width, state$)
  for all tuples of variables, t, of size $width$ **do**
    if $seen\_tuples[t, \text{VALUE-OF}(t)] = false$ **then return** $true$
    end if
  end for
  return $false$
end function
function IW($problem = \langle F, I, \mathcal{A}, G, f\rangle, i$)
  $seen\_tuples \leftarrow$ map table from all the possible $width$-tuple–value pairs to a boolean. Takes up $\begin{pmatrix} |F| \\ width \end{pmatrix} \cdot 2^i$ bits. Initialize to all $false$.
  UPDATE-NOVELTY($seen\_tuples, i, I$)
  Perform BFS (Algorithm 3), with the following modification.
    When inserting a node to the frontier: only do so if CHECK-NOVELTY($seen\_tuples, i, node$.STATE) $= true$. Then, update $seen\_tuples \leftarrow$ UPDATE-NOVELTY($seen\_tuples, i, node$.STATE).
  return the return value of the performed BFS.
end function
function ITERATED-WIDTH($problem = \langle F, I, \mathcal{A}, G, f\rangle$)
  $i \leftarrow 1$
  repeat
    $r \leftarrow$ IW($problem, i$)
    $i \leftarrow i + 1$
  until r is not failure or $i > |F|$
  return r
end function

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 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 do almost anything. So the player can either open both doors in screen 1 and not see in the bottom floor, 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 Central Processing Unit (CPU), which can address 8 KB of memory. Addresses range from 0x0000 to 0x1FFF. Addresses larger than 0x1000, included, are mapped to the Read Only Memory (ROM) 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 Random Access Memory (RAM). The 2600 has only 128 bytes of RAM, 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 RAM 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.

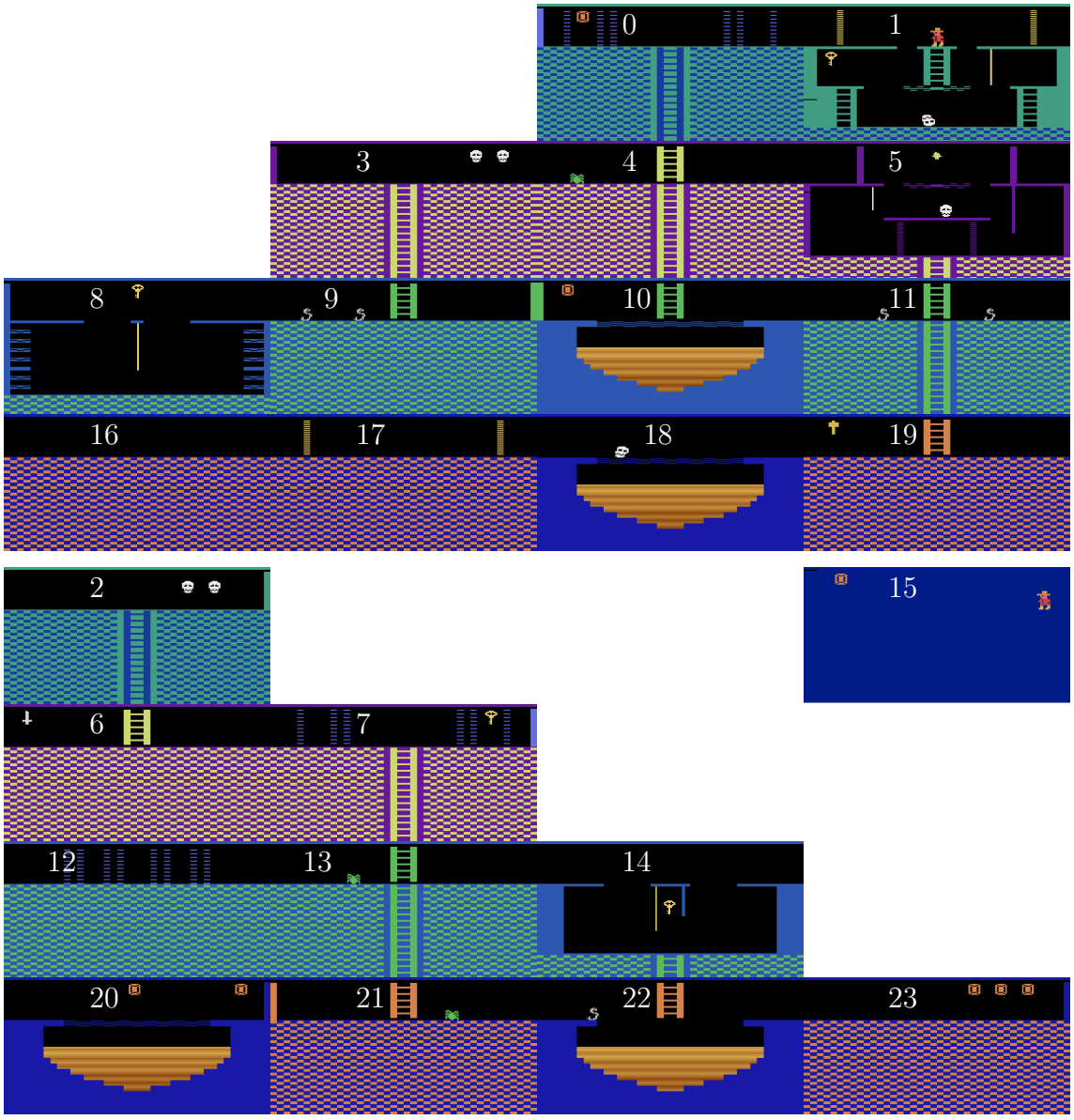


Figure 3.1: The complete map of Montezuma's Revenge. Rooms are numbered from left to right and from top to bottom. The pyramid they form has been cut to fit in the page. Room 15 is located to the left of room 16. The screens are not numbered in the game. The player starts in room 1 and finishes in room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

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.

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 RAM 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 RAM 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 Binary Coded Decimal. 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 2.1.3).

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

- Skull position: $v_1 = \text{RAM(0xAF)} - 0x16$, $v_1 \in [0, 51)$.
- 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: $v_3 = \text{RAM(0xAA)}$, $v_3 \in [0, 256)$.
- Joe's Y: $v_4 = \text{RAM(0xAB)}$, $v_4 \in [0, 256)$.
- Whether Joe has the key: $v_5 = \begin{cases} 0 & \text{if } \textsc{Bitwise-And}(\text{RAM(0xC1)}, 0x1E) = 0 \\ 1 & \text{otherwise} \end{cases}$
- Whether Joe will lose a life upon touching the ground, or has already done so: $v_6 = \begin{cases} 0 & \text{if RAM(0xD8)} \geq 8 \vee \text{RAM(0xBA)} < 5 \\ 1 & \text{otherwise} \end{cases}$

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 2.3.4.

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 5.

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 2.3.4):

$$F(s, a, s') = \gamma\phi(s') - \phi(s) \qquad \text{(2.15 revisited)}$$

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 5 The potential function for shaping.

function 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 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 PROGRESS($\langle x, y\rangle, [p_1 = \langle x_1, y_1\rangle, p_2 = \langle x_1, y_1\rangle, \ldots, p_n]$)
  $\forall i \in [1, n],\ t_i = 0$
  $\forall i \in [1, n),\ (p_{n+i}, t_{n+i}) = \text{PROJECT}(\langle x, y\rangle, p_i, p_{i+1})$
  $m = \arg\min_{i \in [1,2n) \text{ s.t. } t_i \leq 1} \text{DIST}(\langle x, y\rangle, p_i)$
  if DIST($\langle x, y\rangle, p_m$) > 10 **then**
    return 0
  end if
  $\forall j \in [1, n],\ \text{len}_j = \sum_{i=1}^{j-1} \text{DIST}(p_i, p_{i+1})$ ▷ 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 PHI(v_3, v_4, v_5, v_6)
  return $\begin{cases} 0 & \text{if } v_6 = 1 \\ \text{PROGRESS}(\langle v_3, v_4\rangle, p[v_5]) & \text{otherwise} \end{cases}$
end function
$p[0] = [\langle 100, 201\rangle, \langle 133, 201\rangle, \langle 133, 148\rangle, \langle 21, 148\rangle, \langle 21, 192\rangle, \langle 9, 207\rangle]$
$p[1] = [\langle 9, 207\rangle, \langle 21, 192\rangle, \langle 21, 148\rangle, \langle 133, 148\rangle, \langle 133, 201\rangle, \langle 72, 201\rangle, \langle 72, 251\rangle, \langle 153, 251\rangle]$

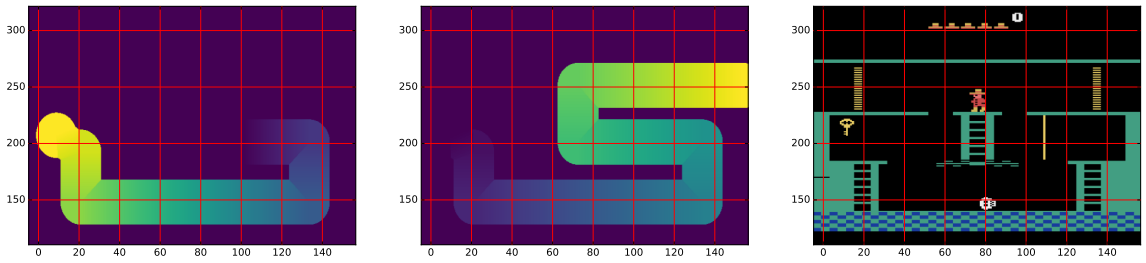


Figure 3.2: The potential function ϕ field used for shaping. From left to right: $v_5 = 0$, $v_5 = 1$, reference screenshot from the game. Vertical axis is v_4, horizontal axis is v_3. Yellow is $\phi(\cdot) = 2$, deep purple is $\phi(\cdot) = 1$.

3.2.3 Options

We created options to test with 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 7.
    - 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

Iterated Width was first applied to Atari games by Lipovetzky, Ramirez, and Geffner (2015). They used IW(1) only (Subsection 2.5.3) in an on-line setting (Subsection 2.5.1). Since IW operates only on boolean variables, they convert each of the RAM'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 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 $\langle 0\text{xAA}, 0\text{xAB}, 0\text{x83}\rangle$, 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 IW 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 "IW(3)" or "IW(3) on position", even the original IW(3) would prune far less often.

IW(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.

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

How is it possible that we can use IW(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).

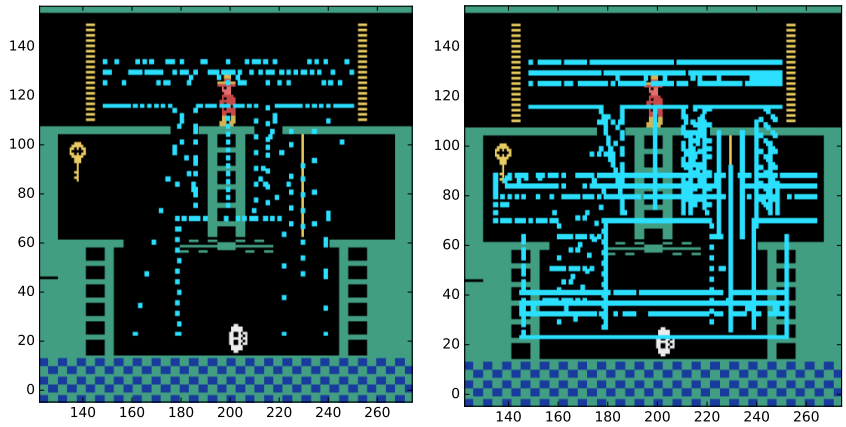


Figure 3.3: Comparison of exploration in the first screen by IW(1) and IW(3) on position. Observe that IW(1), to the left, prunes all paths that move to the right-middle platform, since their y coincides with the the central platform and their x coincides with the right-top platform. Our restricted IW(3) 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.

IW, like BFS (Algorithm 3), has a single FIFO 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 lucky frames, this makes the agent explore farther.

On unlucky frames, 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. At each frame, the newly generated branch is compared with the previous one, and the one with the most return is followed.

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 7, 8 and 6. Online-Setting-Episode in Algorithm 6 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 RAM 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 6 The agent in an on-line setting, using IW(3) for Montezuma's Revenge, along with some supporting functions for IW(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 $\leftarrow 0$
    $(a', R') \leftarrow \text{IW3}(\langle s, \mathcal{A}, f\rangle, max_{ef}, max\_wait, max\_backtrack, p_r)$
    if $R = [] \vee R'[1] > R[1]$ **then**
      $R \leftarrow R', a \leftarrow a'$
    end if
    for i from 1 to $\min(n_a, \text{LENGTH}(R))$ **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**
  $r_s = \text{BRANCH-RETURN}(c)\ \forall c \in s.\text{CHILDREN}$
  $r \leftarrow \max_{r \in r_s} r$, comparing by first element, ties broken by higher length.
  return APPEND($[s.\text{RETURN}], \gamma \cdot r$)
end function

Algorithm 7 Supporting functions for IW(3) (Algorithm 8)

function 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 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 ON-GROUND?(ram)
  return $ram[\text{0xD6}] = \text{0xFF} \wedge ram[\text{0xD8}] = 0$
end function
function FALLING?(ram)
  return $ram[\text{0xD8}] \neq 0$
end function
function LRUD?(a)
  return Whether the action $a \in \{\text{LEFT}, \text{RIGHT}, \text{UP}, \text{DOWN}\}$.
end function
function OBSTACLE-WAIT($f, obstacle\_child, q, max\_wait, max\_backtrack$)
  $p \leftarrow obstacle\_child.\text{PARENT}$, $p_p \leftarrow obstacle\_child$, $l_0 \leftarrow p[\text{0xBA}]$
  for $i \in [0, max\_backtrack)$ **do**
    $f^i(s, a) = f(f(\ldots f(s, a)\ldots, a), a)$, totalling $i + 1$ applications of f
    $n \leftarrow f^i(p, \text{NOOP})$
    if ON-GROUND?$(n) \wedge n[\text{0xBA}] = l_0$ **then**
      for $i \in [0, max\_wait)$ **do**
        $n_{\text{test}} \leftarrow f^2(n, p_p.\text{ACTION})$
        if ON-GROUND?$(n_{\text{test}}) \wedge n_{\text{test}}[\text{0xBA}] = l_0$ **then**
          return QUEUE-INSERT(q, n)
        end if
        $n \leftarrow f(n, \text{NOOP})$
      end for
    end if
    $p_p \leftarrow p$, $p \leftarrow p.\text{PARENT}$
  end for
  return q
end function

Algorithm 8 IW(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 $[0, 256)^2 \times [0, 24)$
  $visited\_screens[i] \leftarrow false$, $pruned\_screens[i] \leftarrow false$, $\forall i \in [0, 24)$
  $visited\_screens[s_0.ram[\text{0x83}]] \leftarrow true$
  $q \leftarrow [s_0]$, $q_l \leftarrow []$, FIFO 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[0xBA] $<$ s[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 8, 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 IW(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 FIFO queue, like BFS and IW.
  - q, q_l: two FIFO 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).

  FIFO queue, 2 FIFO 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 paremeters values are: $max\_wait = 20$, $max\_backtrack = 7$.

Name	max_{ef}	γ	n_a	p_r	FS	TR	Frontier	EB	OAV	EA	Score
IW(1)	150k	0.995	1		5	✓	q			✓	0
2BFS	150k	0.995	1		5	✓	2BFS			✓	540
IW(3)*	150k	0.995	1		5		q				4600
IW(3)*	1 500k	0.995	1		5		P. Dist.		1		2 500
IW(3)*	300k	0.995	1		5		q, q_l	✓	1		5 600
IW(3)*	300k	0.995	1		5		q, q_l	✓	1.1		8 000
IW(3)*	150k	0.99	1		10		q, q_l	✓	1.1		10 200
IW(3)*	75k	0.985	1		10		q, q_l	✓	1.1		0
IW(3)*	10k	0.98	1		20		q, q_l	✓	1.1		0
IW(3)	300k	0.995	1		6		q, q_l		1.2		100
IW(3)	300k	0.999	1		5		q, q_l		1.2		500
IW(3)	300k	0.995	1		5		q, q_l		1.2		6 700
IW(3)	300k	0.999	1		5		q, q_l	✓	1.2		7 100
IW(3)	300k	0.999	1	0.25	5		q, q_l	✓	1.2		4 700
IW(3)	300k	0.99	1	0.2	5		q, q_l	✓	1.2		11 000
IW(3)	300k	0.99	1	0.2	5		q, q_l	✓	2		13 600
IW(3)	150k	0.99	2	0.2	10	✓	q, q_l	✓	2		14 500
IW(3)	150k	0.995	2	0.2	5	✓	q, q_l		2		8 000
IW(3)	150k	0.995	2	0.2	5	✓	q, q_l	✓	2	✓	7 800
IW(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, 4.2, 4.3 and 4.4. 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.

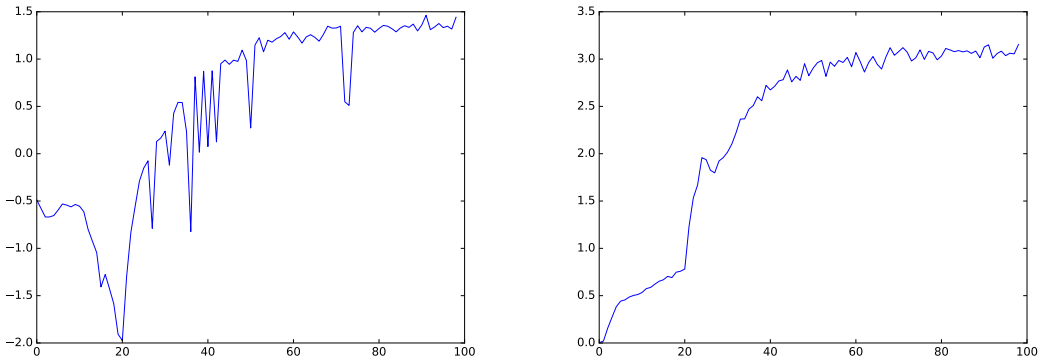


Figure 4.1: The reward over time, without options or annealing. y axis is reward, x axis is thousands of episodes.

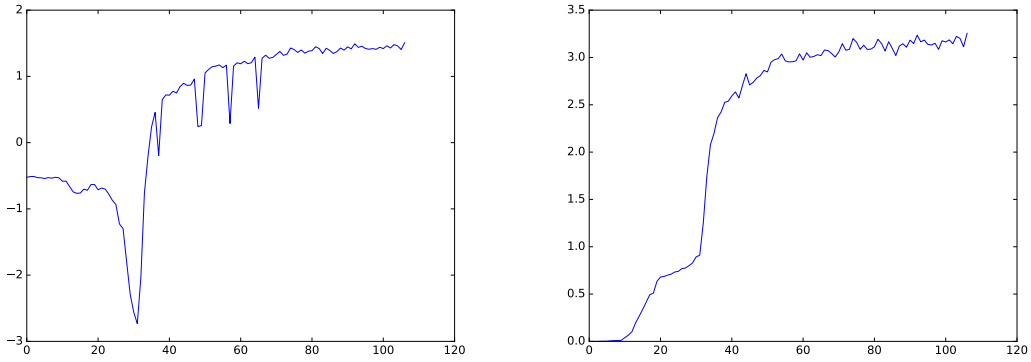


Figure 4.2: The reward over time, without options, with annealing. y axis is reward, x axis is thousands of episodes.

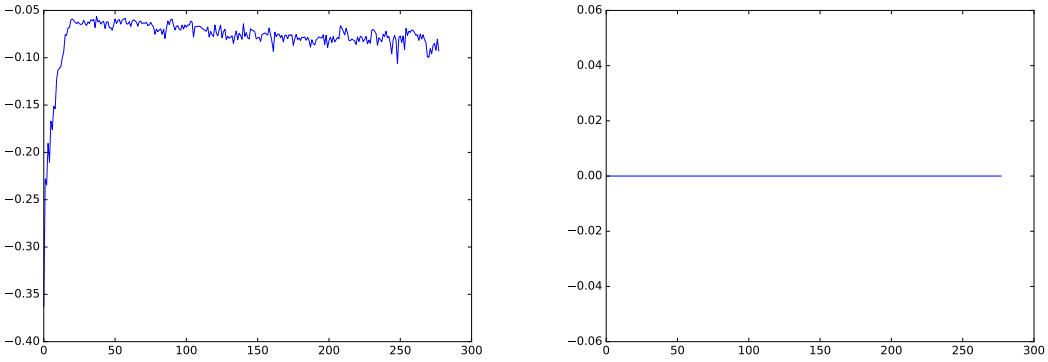


Figure 4.3: The reward over time, with options but without annealing. y axis is reward, x axis is thousands of episodes.

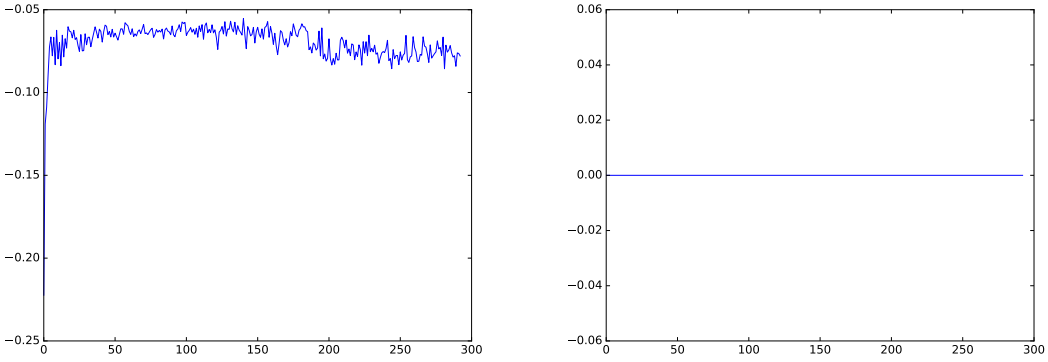


Figure 4.4: The reward over time, with options and with annealing. y axis is reward, x axis is thousands of episodes.

There is something odd about all the graphs. First, both graphs that do not use options (Figures 4.1 and 4.2), actually *decrease* 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 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 4.5).

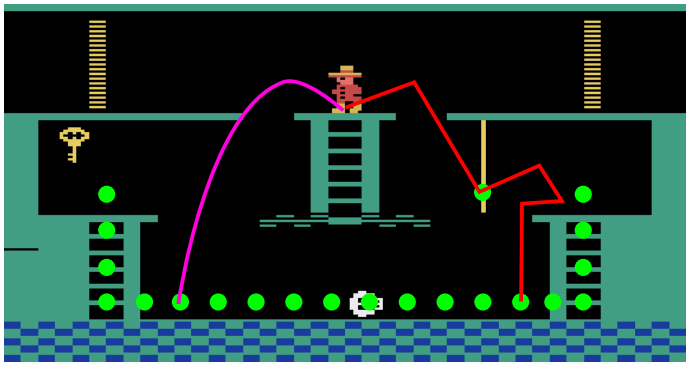


Figure 4.5: The original shaping rewards, and the two ways the agent found of defeating their purpose. First, it took the magenta path. After restricting the amount of rewards to be collected at the same time, it took the red path.

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, Iterated Width.

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 IW 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 using an *autoencoder* (such as Dumoulin et al., 2016) to learn an abstracted representation of the state, and then try to predict the abstracted representation of the next state. 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 targeted 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 Iterated Width, maybe with the novelty measure in Bellemare et al. (2016), could be ran on tuples in the autoencoder's latent variable 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: June 18, 2016. Link to Internet Archive.

Atari 2600 Specifications. http://problemkaputt.de/2k6specs.htm. Last retrieved: June 18, 2016. 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*.

Dumoulin, Vincent et al. (2016). "Adversarially Learned Inference". In: *arXiv preprint arXiv:1606.00704*.

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: June 18, 2016. 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.

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.