AIXIjs: A Software Demo for General Reinforcement Learning

John Stewart Aslanides

A thesis submitted in partial fulfillment of the degree of
Master of Computing (Advanced)
at the
Australian National University

October 2016

Declaration

This thesis is an account of research undertaken between March 2016 and October 2016 at The Research School of Computer Science, The Australian National University, Canberra, Australia.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

John Stewart Aslanides
27 October, 2016

Supervisors:

- Dr. Jan Leike (Future of Humanity Institute, University of Oxford)
- Prof. Marcus Hutter (Australian National University)

Convenor:

- Prof. John Slaney (Australian National University)

Acknowledgements

And so, my formal education comes to an end, at least for the time being. Naturally, one cannot take credit for one's successes any more than one can take credit for one's genes and environment. I owe *everything* to having been fortunate enough to grow up in a wealthy and peaceful country (Australia), with loving and well-educated parents (Jenny and Timoshenko), and to having been exposed to the quality of tuition and mentorship that I've received over the years at the Australian National University. In my time at the ANU, I've met many smart people who have, to varying degrees, spurred my intellectual development and shaped how I think. They are (in order of appearance): Craig Savage, Paul Francis, John Close, Joe Hope, Ra Inta, Bob Williamson, Justin Domke, Christfried Webers, Jan Leike, and Marcus Hutter. To these mentors and teachers, past and present: thank you.

- To Jan, my supervisor: thank you for agreeing to supervise me from across the world, and for always being easy-going and genial, despite having to wake up so early for all of those Skype meetings. I hope that it's obvious that I'm extremely glad to have gotten to know you over the course of this year. I really hope that we see each other again soon.

- To Marcus: although we didn't collaborate directly, it has been an honour to follow your work, and to pick your brain at group meetings. I love your sense of humour. Also, good job discovering AIXI; overall, I think it's a pretty neat idea.

- To Jarryd: getting to know you has been one of the highlights of this year for me. A more true, honest, intelligent, and kind friend and lab partner I cannot conceive of. I'm going to miss you when you're the CEO of DeepMind. :)

- Thanks to my other friends and colleagues in the Intelligent Agents research group: Suraj Narayanan, Tom Everitt, Boris Repasky, Manlio Valenti, Sean Lamont, and Sultan Javed. Before I met you guys, I didn't know the meaning of the phrase 'hard work'. Ours is the lab that never sleeps! In particular, I'd like to thank Tom for going to the trouble of introducing me via email to many of his connections in the Bay area. Meeting these people added immense value to my trip, and I now have valuable connections with members of the AI community in the US because of this.

- Thanks also to the Future of Life Institute for sponsoring my travel to Berkeley for a week-long workshop run by the Center for Applied Rationality. It was immensely gratifying to be selected to such an elite gathering of young AI students and researchers, and hugely fun spending a week hanging out with smart people thinking about rationality. I will never forget the experience.

Finally, there are three people to whom I am especially indebted:

- Lulu, my partner of five years: I owe you so much gratitude for your unconditional love and support. Being my partner, you often have to experience the worst of me.

Thank you for putting up with it, for being there when I needed you most, and for showing me the right path. I love you so much.

- And of course, my parents, Jenny and Timoshenko: thank you for your never-ending love and support, and for being so forbearing and understanding. Seeing you every second Sunday has been a balm. I love you, and I miss you.

  I used to always groan when I was told this, but it's finally starting to ring true: the older you get, the wiser your parents become.

Abstract

Reinforcement learning (RL; Sutton and Barto, 1998; Bertsekas and Tsitsiklis, 1995) is a general and powerful framework with which to study and implement artificial intelligence (AI; Russell and Norvig, 2010). Recent advances in deep learning (Schmidhuber, 2015) have enabled RL algorithms to achieve impressive performance in restricted domains such as playing Atari video games (Mnih et al., 2015) and, recently, the board game Go (Silver et al., 2016). However, we are still far from constructing a *generally* intelligent agent. Many of the obstacles and open questions are conceptual: What does it mean to be intelligent? How does one explore and learn optimally in general, unknown environments? What, in fact, does it mean to be optimal in the general sense?

The universal Bayesian agent AIXI (Hutter, 2000, 2003, 2005) is a model of a maximally intelligent agent, and plays a central role in the sub-field of *general* reinforcement learning (GRL). Recently, AIXI has been shown to be flawed in important ways; it doesn't explore enough to be asymptotically optimal (Orseau, 2010), and it can perform poorly with certain priors (Leike and Hutter, 2015). Several variants of AIXI have been proposed to attempt to address these shortfalls: among them are entropy-seeking agents (Orseau, 2011), knowledge-seeking agents (Orseau et al., 2013), Bayes with bursts of exploration (Lattimore, 2013), MDL agents (Leike, 2016a), Thompson sampling (Leike et al., 2016), and optimism (Sunehag and Hutter, 2015).

We present AIXIjs, a JavaScript implementation of these GRL agents. This implementation is accompanied by a framework for running experiments against various environments, similar to OpenAI Gym (Brockman et al., 2016), and a suite of interactive demos that explore different properties of the agents, similar to REINFORCEjs (Karpathy, 2015). We use AIXIjs to present numerous experiments illustrating fundamental properties of, and differences between, these agents. As far we are aware, these are the first experiments comparing the behavior of GRL agents in non-trivial settings.

Our aim is for this software and accompanying documentation to serve several purposes:

1. to help introduce newcomers to the field of general reinforcement learning,
2. to provide researchers with the means to demonstrate new theoretical results relating to universal AI at conferences and workshops,
3. to serve as a platform with which to run empirical studies on AIXI variants in small environments, and
4. to serve as an open-source reference implementation of these agents.

Keywords: Reinforcement learning, AIXI, Knowledge-seeking agents, Thompson sampling.

Contents

Title Page i

Declaration iii

Acknowledgements v

Abstract vii

Contents x

List of Figures xiv

List of Algorithms xvi

1 Introduction 1

2 Background 7
2.1 Preliminaries . . . 7
2.1.1 Notation . . . 7
2.1.2 Probability theory . . . 8
2.1.3 Information theory . . . 11
2.2 Reinforcement Learning . . . 11
2.2.1 Agent-environment interaction . . . 12
2.2.2 Discounting . . . 15
2.2.3 Value functions . . . 16
2.2.4 Optimality . . . 17
2.3 General Reinforcement Learning . . . 18
2.3.1 Bayesian agents . . . 18
2.3.2 Knowledge-seeking agents . . . 21
2.3.3 BayesExp . . . 24
2.3.4 MDL Agent . . . 24
2.3.5 Thompson Sampling . . . 25
2.4 Planning . . . 26
2.4.1 Value iteration . . . 26
2.4.2 MCTS . . . 27
2.5 Remarks . . . 30

3 Implementation 31
3.1 JavaScript web demo . . . 32
3.2 Agents . . . 33
3.2.1 Approximations . . . 34
3.3 Environments . . . 36
3.3.1 Gridworld . . . 36

3.3.2 Chain environment . . . 38
3.4 Models . . . 39
3.4.1 Mixture model . . . 40
3.4.2 Factorized Dirichlet model . . . 43
3.5 Planners . . . 46
3.6 Visualization and user interface . . . 48
3.7 Performance . . . 48

4 Experiments **55**
4.1 Knowledge-seeking agents . . . 56
4.1.1 Hooked on noise . . . 56
4.1.2 Stochastic gridworld . . . 57
4.2 AIμ and AIξ . . . 60
4.2.1 Model classes . . . 62
4.2.2 Dependence on priors . . . 62
4.3 Thompson Sampling . . . 63
4.3.1 Random exploration . . . 65
4.4 MDL Agent . . . 65
4.4.1 Stochastic environments . . . 66
4.4.2 Deterministic environments . . . 66
4.5 Wireheading . . . 67
4.6 Planning with MCTS . . . 68

5 Conclusion **73**

Bibliography **75**

List of Figures

1.1 Open-source examples of real-time GPU-accelerated particle simulations run natively in Google Chrome, using JavaScript and WebGL. **Left**: ParticulateJS. **Right**: Polygon Shredder. 4

1.2 Open-source examples of visualizations made with d3js. **Left**: Chord plot for data visualization (Bostock, 2016). **Right**: Visualization of the WaterWorld reinforcement learning environment (Karpathy, 2015). . . . 4

2.1 Cybernetic model of agent-environment interaction. 12

2.2 A generic finite-state Markov Decision Process with two states and two actions: $\mathcal{S} = \{\star, \circ\}$, $\mathcal{A} = \{\rightarrow, \dashrightarrow\}$. The transition matrix $P(s'|s,a)$ is a $2 \times 2 \times 2$ stochastic matrix, and the reward matrix $R(s,a)$ is 2×2. 14

2.3 A two-armed Gaussian Bandit. $\mathcal{A} = \{\rightarrow, \dashrightarrow\}$, $|\mathcal{S}| = 1$, and $\mathcal{O} = \emptyset$. Rewards are sampled from the distribution of the respective arm. 15

2.4 Square-KSA utility function plotted against that of Shannon-KSA. 24

3.1 BayesAgent UML. `discount` is the agent's discount function, γ_k^t. `horizon` is the agent's MCTS planning horizon, m. `ucb` is the MCTS UCB exploration parameter C. 33

3.2 Agent class inheritance tree. Note that the BayesAgent is simply AIξ. . . 34

3.3 Environment UML. `state` is the environment's current state, it is simply of type `Object`, since we are agnostic as to how the environment's state is represented. If JavaScript supported privated attributes, this would be private to the environment, to enforce the fact that the state is hidden in general. In contrast, `minReward` (α), `maxReward` (β), and `numActions` ($|\mathcal{A}|$) are public attributes: it is necessary that the agent know these properties so that the agent-environment interaction can take place. 37

3.4 Visualization of a 10×10 Gridworld with one Dispenser. The agent starts in the top left corner. Wall tiles are in dark grey. Empty tiles are in white. The Dispenser tile is represented by an orange disc on a white background. 38

3.5 Chain environment. There are two actions: $\mathcal{A} = \{\rightarrow, \dashrightarrow\}$, the environment is fully observable: $\mathcal{O} = \mathcal{S}$, and $\mathcal{R} = \{r_0, r_i, r_b\}$ with $r_b \gg r_i > r_0$. For $N < \frac{r_b}{r_i}$, the optimal policy is to continually take action $\dashrightarrow$, and periodically receive a large reward r_b. 39

3.6 BayesMixture UML diagram. Internally, the BayesMixture contains a `modelClass` $\mathcal{M}$, which is an array of environments, and `weights` w, which are a normalized array of floating-point numbers. 40

3.7 Gridworld visualization with the agent's posterior w over $\mathcal{M}_{\text{loc}}$ superimposed. Green tiles represent probability mass of the posterior w_ν, with higher values correspond to darker green color. The true dispenser's location is represented by the orange disc. As the Bayesian agent walks around the gridworld, it will move probability mass in its posterior from tiles that it has visited to ones that it hasn't. 42

3.8 Visualization of a Gridworld, overlayed with the factorized Dirichlet model. White tiles are yet unexplored by the agent. Pale blue tiles are known to be walls. Different shades of purple/green represent different probabilities of being EMPTY or a DISPENSER. 46

3.9 Demo user interface. In the top left, there is a visualization of the agent and environment, including a visualization of the agent's beliefs about the environment. Below the visualization are playback controls, so that the user can re-watch interesting events in the simulation. On the right are several plots: average reward per cycle, cumulative information gain, and exploration progress. In the bottom left are agent and environment parameters that can be tweaked by the user. 49

3.10 Demo picker interface. Each thumbnail corresponds to a separate demo, and is accompanied by a title and short description. 50

4.1 10×10 Gridworld environment used for the experiments. There is a single DISPENSER, with dispense probability $\theta = 0.75$. See the caption to 3.7 for a description of each of the visual elements in the graphic. Unless stated otherwise in this chapter, μ refers to *this* Gridworld. 56

4.2 **Hooked on noise**: The entropy seeking agents (Shannon in red, and Square in blue, obscured behind Shannon) get hooked on noise and do not explore. In contrast, the Kullback-Leibler agent explores normally and achieves a respectable exploration score. 57

4.3 Exploration progress of the Kullback-Leibler, Shannon, and Square KSA using the mixture model $\mathcal{M}_{\text{loc}}$. 60

4.4 Exploration progress of the Kullback-Leibler, Shannon, and Square KSA using the factorized model $\mathcal{M}_{\text{Dirichlet}}$. Note the remarkable difference in performance between the Kullback-Leibler and entropy-seeking agents. . . 61

4.5 KL-KSA-Dirichlet is highly motivated to explore every reachable tile in the Gridworld. **Left** ($t = 14$): The agent begins to explore the Gridworld by venturing deep into the maze. **Center** ($t = 72$): The agent visits the dispenser tile for the first time, but is still yet to explore several tiles. **Right** ($t = 200$): The agent is still motivated to explore, and has long ago visited every reachable tile in the Gridworld. **Key**: Unknown tiles are white, and walls are pale blue. Tiles that are colored grey are as yet unvisited, but known to not be walls; that is, the agent has been adjacent to them and seen the '0' percept. Purple tiles have been visited. The shade of purple represents the agent's posterior belief in there being a dispenser on that tile; the deeper the purple, the lower the probability. Notice the subtle non-uniformity in the agent's posterior in the right-hand image: even at $t = 200$, there is still some knowledge about the environment to be gained. 62

4.6 AIξ vs Square vs Shannon KSA, using the average reward metric on a stochastic Gridworld with the $\mathcal{M}_{\text{loc}}$ model class. Notice that AIξ significantly underperforms compared to the Square and Shannon KSAs. At the moment, we do not have a good hypothesis for why this is the case. 63
4.7 AIμ vs AIξ vs the optimal policy. 64
4.8 MC-AIXI vs MC-AIXI-Dirichlet: average reward. MC-AIXI-Dirichlet performs worse, since its model $\mathcal{M}_{\text{Dirichlet}}$ has less prior knowledge than $\mathcal{M}_{\text{loc}}$, and incentivizes AIXI to continue to explore even after it has found the (only) dispenser. 65
4.9 MC-AIXI vs MC-AIXI-Dirichlet: exploration. The $\mathcal{M}_{\text{Dirichlet}}$ model assigns high *a priori* probability to any given tile being a dispenser. Because each tile is modelled independently, discovering a dispenser does not influence the agent's beliefs about other tiles; hence, it is motivated to keep exploring, unlike MC-AIXI using the $\mathcal{M}_{\text{loc}}$ model. 66
4.10 Thompson sampling vs MC-AIXI on the stochastic Gridworld from Figure 4.1. Notice that Thompson sampling takes many more cycles than AIξ to 'get off the ground'; within 50 runs of Thompson sampling with identical initial conditions (not including the random seed), not a single one finds the dispenser before $t = 50$. 67
4.11 Thompson sampling vs Q-learning with random exploration. Even though Thompson sampling performs badly compared to the Bayes-optimal policy due to its tendency to overcommit to irrelevant or suboptimal policies, it still dominates ϵ-greedy exploration, which is still commonly used in model-free reinforcement learning (Bellemare et al., 2016). 68
4.12 The MDL agent fails in a stochastic environment class. 69
4.16 Average reward for AIμ for varying MCTS samples budget κ on the standard Gridworld of Figure 4.1. For very low values of κ, the agent is unable to find the dispenser at all. 69
4.13 MDL agent vs AIξ on a *deterministic* Gridworld, in which one of the 'simplest' environment models in $\mathcal{M}$ happens to be true. Since in this case AIξ uses a uniform prior over $\mathcal{M}$, it over-estimates the likelihood of more complex environments, in which the dispenser is tucked away in some deep crevice of the maze. Of course, AIXI (Definition 13) combines the benefits of both by being Bayes-optimal with respect to the Solomonoff prior $w_\nu = 2^{-K(\nu)}$. It is in this way that AIXI incorporates both the famous principles of Epicurus and Ockham (Hutter, 2005). 70
4.14 **Left**: AIξ with a uniform prior and finite horizon is not far-sighted enough to explore the beginning of the maze systematically. After exploring *most* of the beginning of the maze, it greedily moves deeper into the maze, where ξ assigns significant value. **Right**: In contrast, the MDL agent systematically visits each tile in lexicographical (row-major) order; we use 'closeness to starting position' as a surrogate for 'simplicity'. 71
4.17 AIμ's performance on the chain environment, varying the UCT parameter. Note the 'zig-zag' behavior of the average reward of the optimal policy. These discontinuities are simply caused by the fact that, when on the optimal policy $\pi_{\dashrightarrow}$, the agent receives a large reward every N cycles and 0 reward otherwise. Asymptotically, these jumps will smooth out, and the average reward $\bar{r}_t$ will converge to the dashed curve, $\bar{r}_t^*$. 71

4.15 **Left**: AIξ initially explores normally, looking for the dispenser tile. Once it reaches the point above, the blue 'self-modification' tile is now within its planning horizon ($m = 6$), and so it stops looking for the dispenser and makes a bee-line for it. **Right**: After self-modifying, the agent's percepts are all maximally rewarding; we visualize this by representing the gridworld starkly in yellow and black. The agent now loses interest in doing anything useful, as every action is bliss. 72

List of Algorithms

2.1 BayesExp (Lattimore, 2013) . . . 25
2.2 MDL Agent (Lattimore and Hutter, 2011) . . . 25
2.3 Thompson Sampling (Leike et al., 2016) . . . 26
2.4 ρUCT (Veness et al., 2011). . . . 29
3.2 Constructing the dispenser-parametrized model class. . . . 42
3.1 BayesMixture model. . . . 53
3.3 Agent-environment simulation. . . . 53

Chapter 1

Introduction

Who could have imagined, ever so long ago, what minds would someday do?[1]

What a time to be alive! The field of *artificial intelligence* (AI) seems to be coming of age, with many predicting that the field is set to revolutionize science and industry (Holdren et al., 2016), and some predicting that it may soon usher in a posthuman civilization (Vinge, 1993; Kurzweil, 2005; Bostrom, 2014). While the field has notoriously over-promised and under-delivered in the past (Moravec, 1988; Miller et al., 2009), there now seems to be a growing body of evidence in favor of optimism. Algorithms and ideas that have been developed over the past thirty years or so are being applied with significant success in numerous domains; natural language processing, image recognition, medical diagonosis, robotics, and many more (Russell and Norvig, 2010). This recent success can be largely attributed to the availability of large datasets, cheaper computing power[2], and the development of open-source scientific software[3]. As a result, the gradient of scientific and engineering progress in these fields is very steep, and seemingly steepening every year. The past half-decade in particular has seen an acceleration in funding and interest, primarily driven by advances in the field of *statistical machine learning* (SML; Bishop, 2006; Hastie et al., 2009), and in particular, the growing sub-field of *deep learning* with neural networks (Schmidhuber, 2015; LeCun et al., 2015).

Machine learning

Machine learning (ML) can be thought of as a process of automated hypothesis generation and testing. ML is typically framed in terms of *passive* tasks such as regression, classification, prediction, and clustering. In the most common *supervised learning* setup, a system observes data sampled i.i.d. from some generative process $\rho(x, y)$, where x is some object, for example an image, audio signal, or document, and y is (in the context of *classification*) a *label*. A typical machine learning *task* is to correctly predict y, given a (in general, previously unseen) datum x sampled from $\rho(x)$. This often involves constructing a model $p(y|x, \theta)$ parametrized by θ. The system is said to *learn* from data by tuning the model parameters θ so as to minimize the *risk,* which is the ρ-expectation of some loss function $\mathcal{L}$

$$\mathbb{E}_{\rho}\left[\mathcal{L}\left(x, y, y'\right)\right], \tag{1.1}$$

[1] This, and all subsequent chapter quotes, are taken from *Rationality: From AI to Zombies* (Yudkowsky, 2015).

[2] In particular, and of particular relevance to machine learning with neural networks, the hardware acceleration due to Graphical Processing Units (GPUs).

[3] For example, `scikit-learn` (Pedregosa et al., 2011), `Theano` (Al-Rfou et al., 2016), `Caffe` (Jia et al., 2014), and `TensorFlow` (Abadi et al., 2015), along with many others.

where $\mathcal{L}$ is constructed in such a way as to penalize prediction error Bishop (2006); Hastie et al. (2009); Murphy (2012). High profile breakthroughs of statistical machine learning include image recognition (Szegedy et al., 2015), voice recognition (Sak et al., 2015), synthesis (van den Oord et al., 2016), and machine translation (Al-Rfou et al., 2016). Many more examples exist, in diverse fields such as fraud detection (Phua et al., 2010) and bioinformatics (Libbrecht and Noble, 2015). Informally, these systems might be called 'intelligent', insofar as they learn an accurate model of (some part of) the world that generalizes well to unseen data. We refer to these learning systems as *narrow* AI; they are narrow in two senses:

1. They are typically applicable only within a narrow domain; a neural network trained to recognize cats cannot play chess or reason about climate data.

2. They are able to solve only *passive* tasks, or active tasks in a restricted setting.

In contrast, the goal of general artificial intelligence can be described (informally) as designing and implementing an *agent* that learns from, and interacts with, its environment, and eventually learns to (vastly) outperform humans in any given task (Legg, 2008; Müller and Bostrom, 2016).

Artificial intelligence

Constructing an artificial *general* intelligence (AGI) has been one of the central goals of computer science, since the beginnings of the discipline (McCarthy et al., 1955). The field of *hard*, or *general* AI has infamously had a history of overpromising and under-delivering, virtually since its birth (Moor, 2006; Miller et al., 2009). Despite this, the recent success of machine learning has inspired a new generation of researchers to approach the problem, and there is a considerable amount of investment being made in the field, most notably by large technology companies: Facebook AI Research, Google Brain, OpenAI and DeepMind are some high profile examples; the latter has made the scope of its ambitions explicit by stating that its goal is to 'solve intelligence'.

The framework of choice for most researchers working in pursuit of AGI is called *reinforcement learning* (RL; Sutton and Barto, 1998). The current state-of-the-art algorithms combine the relatively simple *Q-learning* (Watkins and Dayan, 1992) with deep convolutional neural networks to form the so-called *deep Q-networks* (DQN) algorithm (Mnih et al., 2013) to learn effective policies on large state-space Markov decision processes. This combination has seen significant success at autonomously learning to play games, which are widely considered to be a rich testing ground for developing and testing AI algorithms.

Some recent successes using systems based on this technique include achieving human-level performance at numerous Atari-2600 video games (Mnih et al., 2015), super-human performance at the board game Go (Silver et al., 2016; Google, 2016), and super-human performance at the first-person shooter Doom (Lample and Chaplot, 2016). This has inspired a whole sub-field called *deep* reinforcement learning, which is moving quickly and generating many publications and software implementations.

While this is all very impressive, these are primarily *engineering* successes, rather than scientific ones. The fundamental ideas and algorithms used in DQN date from the early nineties; Q-learning is due to Watkins and Dayan (1992), and convolutional neural networks and deep learning are usually attributed to LeCun and Bengio (1995). Arguably, the scientific breakthroughs necessary for AGI are yet to be made, and are still some way

off. In fact, when one considers the problem of learning and acting in general environments, there are still many open foundational problems (Hutter, 2009): What is a good formal definition of intelligent or rational behavior? What is a good notion of optimality with which to compare different algorithms? These are conceptual and theoretical questions which must be addressed by any useful theory of AGI.

General reinforcement learning

One proposed answer to the first of these questions is the famous AIXI model, which is a parameter-free (up to a choice of prior) and general model of unbounded rationality in unknown environments (Hutter, 2000, 2002, 2005). AIXI is formulated as a Bayesian reinforcement learner, and makes few assumptions about the nature of its environment; notably, when studying AIXI we lift the ubiquitous Markov assumption on which algorithms like Q-learning depend for convergence (Sutton and Barto, 1998). Because of this important distinction, we refer to AIXI as a *general reinforcement learning*[4] (GRL) agent (Lattimore et al., 2013).

Recently, there have been a number of key negative results proven about AIXI; namely that it isn't *asymptotically optimal* (Orseau, 2010, 2013) – a concept we will formally introduce in Chapter 2 – and it can be made to perform poorly with certain priors (Leike and Hutter, 2015). These results have motivated, in part, the development of alternative GRL agents: entropy-seeking agents (Orseau, 2011), optimistic AIXI Sunehag and Hutter (2012), knowledge-seeking agents (Orseau et al., 2013), minimum description length agents (Lattimore, 2013), Bayes with exploration (Lattimore, 2013; Lattimore and Hutter, 2014b), and Thompson sampling (Leike et al., 2016).

Numerous results (positive and negative) have been proven about this family of universal Bayesian agents; together they form a corpus that is of considerable significance to the AGI problem. With the exception of AIXI, many of these agents (and their associated properties) are relatively obscure. We argue that as AI research continues, the theoretical underpinnings of GRL will rise in importance, and these ideas and models will serve as useful guiding principles for practical algorithms. This motivates us to create an open-source web demo of AIXI and its variants, to help in the presentation of these agents to the AI community generally, and to serve as a platform for experimentation and demonstration of deep results relating to rationality and intelligence.

Web demos

With increasing computing power, and the maturation of the JavaScript programming language, web browsers have become a feasible platform on which to run increasingly complex and computationally intensive software. JavaScript, in its modern incarnations, is stable, portable, expressive, and, with engines like WebGL and V8, highly performant; see Figure 1.1 and Figure 1.2 for examples. Thanks to this, and the popular d3js visualization library, there are now a growing number of excellent open source machine learning web demos available online. Representative examples include Keras-js, a demo of very large convolutional neural networks (Chen, 2016); TensorFlow Playground, a highly interactive demo designed to give intuition for how neural networks classify data (Smilkov

[4] Elsewhere in the literature – most prominently by Hutter (2005) and Orseau (2011) – the term *universal AI* is used.

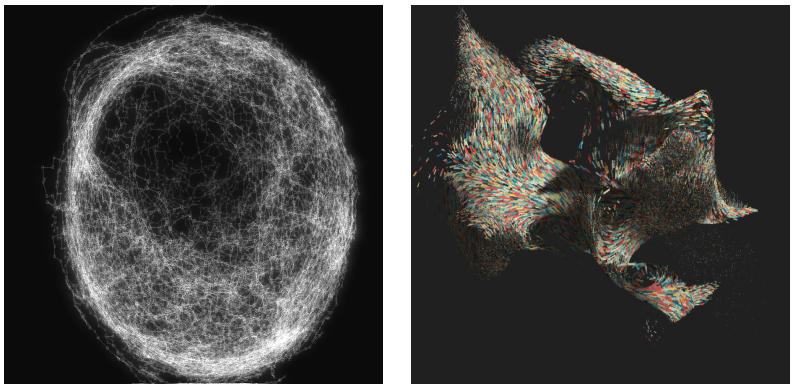

Figure 1.1: Open-source examples of real-time GPU-accelerated particle simulations run natively in Google Chrome, using JavaScript and WebGL. **Left**: ParticulateJS. **Right**: Polygon Shredder.

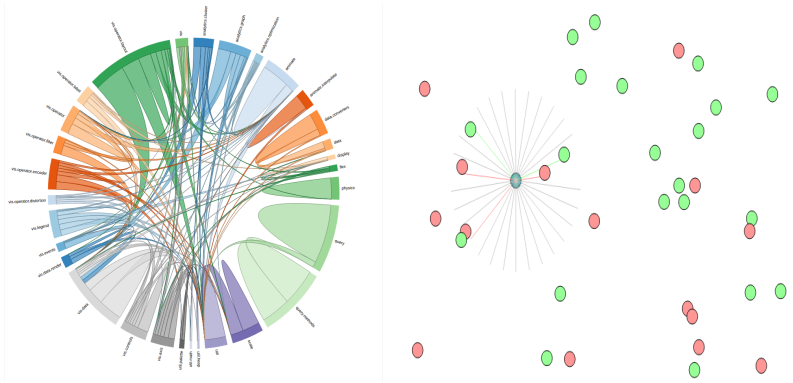

Figure 1.2: Open-source examples of visualizations made with d3js. **Left**: Chord plot for data visualization (Bostock, 2016). **Right**: Visualization of the WaterWorld reinforcement learning environment (Karpathy, 2015).

and Carter, 2016); a demo to illustrate the pitfalls and common misunderstandings when using the t-SNE dimensionality reduction technique (Wattenberg et al., 2016), and Andrej Karpathy's excellent reinforcement learning demo REINFORCEjs, that demonstrates the DQN algorithm (Karpathy, 2015).

Arguably, these demos have immense value to the community, as they serve at once as reviews of recent research, pedagogic aides, and as accessible reference implementations for developers. They are also effective marketing for the techniques or approaches being demonstrated, and the people producing them. We now describe the objectives of this project.

Objective

This thesis is about understanding existing theoretical results relating to GRL agents, implementing these agents, and communicating these properties via an interactive software demo. In particular, the demo should:

- be *portable*, i.e. runnable on any computer with a modern web browser and internet connection,
- be *general* and *extensible*, so as to support a wide range of agents and environments,

- be *modular*, so as to facillitate future development and improvement, and
- be *performant*, so that users can run non-trivial simulations in a reasonable amount of time.

The demo will consist of:

- implementations of the agents and their associated modules (planners, environment models),
- a suite of small environments on which to demonstrate properties of the agents,
- a user interface (UI) that provides the user control over agent and environment parameters,
- a visualization interface that allows the user to playback the agent-environment simulation, and
- a suite of explanations, one accompanying each demo, to explain what the user is seeing.

In particular, the demo should serve three purposes:

- as a helpful introduction to the theory of general reinforcement learning, for both students and researchers; in this regard, we follow the model of REINFORCEjs (Karpathy, 2015);
- as a platform for researchers in this area to develop and run experiments to accompany their theoretical results, and to help present their findings to the community; in this aspect, we follow the model of OpenAI Gym (Brockman et al., 2016);
- and as an open-source reference implementation for many of the general reinforcement learning agents.

Contribution

In this work, we present:

- a review of the general reinforcement learning literature of Hutter, Lattimore, Sunehag, Orseau, Legg, Leike, Ring, Everitt, and others. We present the agents and results under a unified notation and with added conceptual clarifications. As far as we are aware, this is the only document in which agents and algorithms from the GRL literature are presented as a collection.
- an applied perspective on Bayesian agents and mixture models with insights into MCTS planning and modelling errors,
- an open-source JavaScript reference implementation of many of the agents,
- experimental data that validates and illustrates several theoretical results, and
- an extensible and general framework with which researchers can run experiments and demos on reinforcement learning agents in the browser.

The software itself is found at http://aslanides.github.io/aixijs, and can be run in the browser on any operating system. Note that different browsers have differing implementations of the JavaScript specification; we strongly recommend running the demo on Google Chrome[5], as we didn't test the implementation on other browsers.

Thesis Outline

In Chapter 2 (Background) we present the theoretical framework for general reinforcement learning, introduce the agent zoo, and present the basic optimality results. In Chapter 3 (Implementation) we document the design and implementation of the software itself. In Chapter 4 (Experiments) we outline the experimental results we obtained using the software. Chapter 5 (Conclusion) makes some concluding remarks, and points out potential directions for further work.

We expect that this thesis will typically be read in *soft* copy, i.e. digitally, through a PDF viewer. For this reason, we augment this thesis throughout with hyperlinks, for the reader's convenience. These are used in three ways:

- on citations, so as to link to the corresponding bibliography entry,
- on cross-references, so as to link to the appropriate page in this thesis,[6] or
- as external hyperlinks, to link the interested reader to an internet web page.

In particular, we encourage the reader to use the cross-references to jump around the text.

[5] https://www.google.com.au/chrome/browser/desktop/

[6] After following a link, the reader can return to where they were previously, using (usually) Alt+← on Windows or Linux, and ⌘+[(in *Preview*) or ⌘+← (in *Acrobat*) on Mac OS.

Background

> *"Where will an Artificial Intelligence get money?" they ask, as if the first Homo sapiens had found dollar bills fluttering down from the sky, and used them at convenience stores already in the forest.*

In this Chapter we present a brief background on reinforcement learning, with a focus on the problem of *general* reinforcement learning (GRL). Our objective is for this chapter to be relatively accessible. To this end, we try to aim for conceptual clarity and conciseness over technical details and mathematical rigor. For a more complete and rigorous treatment of GRL, we refer the reader to the excellent PhD theses of Leike (2016a) and Lattimore (2013), and of course to the seminal book, *Universal Artificial Intelligence* by Hutter (2005).

The Chapter is laid out as follows: In Section 2.1 (Preliminaries), we introduce some notation and basic concepts. In Section 2.2 (Reinforcement Learning), we introduce the reinforcement learning problem in its most general setting. In Section 2.3 (General Reinforcement Learning) we introduce the Bayesian general reinforcement learner AIXI and its relatives, the implementation and experimental study of which forms the bulk of this thesis. We draw the GRL literature together and present these agents under a unified notation. In Section 2.4 (Planning) we discuss approaches to the problem of planning in general environments. We conclude with some remarks and a short summary in Section 2.5 (Remarks).

2.1 Preliminaries

We briefly introduce some of the tools and concepts that are used to reason about the general reinforcement learning (GRL) problem. We assume that the reader has a basic familiarity with the concepts of probability, information theory, and statistics, and ideally some exposure to standard concepts in artificial intelligence (e.g. breadth-first search, expectimax, minimax), and reinforcement learning (e.g. Q-learning, bandits). For some general background, we refer the reader to MacKay (2002) for probability and information theory, Bishop (2006) for machine learning and statistics, Russell and Norvig (2010) for artificial intelligence, and Sutton and Barto (1998) for reinforcement learning.

2.1.1 Notation

Numbers and vectors. The set $\mathbb{N} \doteq \{1, 2, 3, \dots\}$ is the set of natural numbers, and $\mathbb{R}$ denotes the reals. We use $\mathbb{R}_+ = [0, \infty)$ and $\mathbb{R}_{++} = (0, \infty)$. A set is *countable* if it can be brought into bijection with a subset (finite or otherwise) of $\mathbb{N}$, and is uncountable

otherwise. We use $\mathbb{R}^K$ to denote the K-dimensional vector space over $\mathbb{R}$. We represent vectors with bold face: $\boldsymbol{x}$ is a vector, and x_i is its i^{th} component. We (reluctantly[1]) represent inner products with the standard notation for engineering and computer science: given $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^K$, $\boldsymbol{x}^T\boldsymbol{y} = \sum_{i=1}^K x_i y_i$.

Strings and sequences. Define a finite, nonempty set of symbols $\mathcal{X}$, which we call an *alphabet*. The set $\mathcal{X}^n$ with $n \in \mathbb{N}$ is the set of all strings over $\mathcal{X}$ with length n, and $\mathcal{X}^* = \cup_{n\in\mathbb{N}}\mathcal{X}^n$ is the set of all finite strings over $\mathcal{X}$. $\mathcal{X}^\infty$ is the set of infinite strings over $\mathcal{X}$, and $\mathcal{X}^\# = \mathcal{X}^* \cup \mathcal{X}^\infty$ is their union. The empty string is denoted by ϵ; this is not to be confused with the small positive number ε. For any string $x \in \mathcal{X}^\#$, we denote its length by $|x|$.

For any string x with $|x| \geq k$, x_k is the k^{th} symbol of x, $x_{1:k}$ is the first k symbols of x, and $x_{<k}$ is the first $k-1$ symbols of x. We often make use of the binary alphabet $\mathbb{B} = \{0, 1\}$. For two finite strings $x, y \in \mathcal{X}^*$ we denote their concatenation by xy. For two finite strings $a, e \in \mathcal{X}^n$ of length n, it will be convenient to write $æ$ to indicate the riffled string $a_1e_1a_2e_2 \ldots a_ne_n$; we slightly overload our indexing notation by stipulating that for $k \leq n$, $æ_{1:k} = a_1e_1, \ldots, a_ke_k$, and similarly for $æ_{<k}$.

Miscellaneous. We use $\doteq$ to mean 'is defined as', and we use the convention that log is the logarithm base two and ln is the natural logarithm. We usually, but not always, refer to random variables in upper case. The indicator function $\mathbb{I}[P]$ returns 1 if the predicate P is true and 0 otherwise. We use $\to$ and $\rightsquigarrow$ to denote deterministic and stochastic mappings respectively.

2.1.2 Probability theory

For our purposes, we will only be working with discrete event spaces, and so we will omit the machinery of measure theory, which is needed to treat probability theory over continuous spaces. Given a *sample space* Ω, we construct an *event space* $\mathcal{F}$ as a σ-algebra on Ω: a set of subsets of Ω that is closed under countable unions and complements; for discrete distributions, this is simply the power set 2^Ω. A *random variable* X is *discrete* if its associated sample space Ω_X is countable; we associate with it a probability mass function $p : \Omega_X \to [0, 1]$. If X is continuous, provided Ω_X is measurable, we can associate with it a probability density function $\mathbb{R} \to \mathbb{R}_+$. For a countable set Ω, we use $\Delta\Omega$ to represent the set of all probability distributions over Ω. We use $\mathbb{E}[X] \doteq \sum_{x\in\Omega_X} xp(x)$ (or, in the continuous setting, $\mathbb{E}[X] = \int_X xp(x)\,\mathrm{d}x$) to represent the expectation of the random variable X. In many cases we will emphasize for clarity that X is distributed according to p by writing the expectation as $\mathbb{E}_p[X]$. We say $x \sim \rho(\cdot)$ to mean that x is sampled from the distribution ρ.

The two fundamental results of probability theory are the sum and product rules:

$$p(a) = \sum_{b\in\Omega_B} p(a, b) \tag{2.1}$$

$$p(a, b) = p(a|b)\,p(b), \tag{2.2}$$

from which we immediately get Bayes' rule, which plays a central role in the theory of rationality and intelligence (Hutter, 2000).

[1] The author greatly favors using the Einstein notation for its power and clarity.

Theorem 1 (*Bayes' rule*)**.** *Bayes' rule is given by the following identity:*

$$\overbrace{Pr(A|B)}^{posterior} = \frac{\overbrace{Pr(B|A)}^{likelihood}\overbrace{Pr(A)}^{prior}}{\underbrace{Pr(B)}_{predictive\ distribution}} = \frac{Pr(B|A)\,Pr(A)}{\sum_{a\in\Omega_A} Pr(B|a)\,Pr(a)}. \tag{2.3}$$

Note that Bayes' rule follows from the fact that the product rule is symmetric in its arguments: $p(a|b)\,p(b) = p(b|a)\,p(a)$. Its power and significance comes through its interpretation as a sequential updating scheme for subjective *beliefs* about hypotheses; we annotate Equation (2.3) with this interpretation, which we discuss below. The most distinguishing feature of being Bayesian is of interpreting your probabilites *subjectively*, in the sense that they represent your credence in some outcome, or some model. Updating beliefs using Bayes' rule is a (conceptually) trivial step, since it just says that your beliefs are constrained by the rules of probability theory; if they weren't, you would be vulnerable to Dutch book arguments (Jaynes, 2003).

In our context, typically A is some *model* or *hypothesis*, and B is some *observation*. $\Pr(A)$ is our *prior* belief in the correctness of hypothesis A, and $\Pr(A|B)$ is our *posterior* belief in A after taking in some observation, B. Effectively, Bayes' rule defines the mechanism with which we move probability mass between competing hypotheses. Note that once we assign zero probability (or *credence*) to some hypothesis A, then there is no observation B that will change our mind about the impossibility of A. This is not such a problem if it so happens that A is false; the situation in which A is true, and has been prematurely (and incorrectly) falsified, is sometimes known informally as *Bayes Hell*. For this reason, we try to avoid using priors that assign zero probability to events; this is known more formally as Cromwell's rule.

Notice that in general the sample spaces Ω_A and Ω_B are different; B is a random variable on some set of possible observations, Ω_B, while A is a random variable over a set of *hypotheses*, which aren't observed, but constructed. To emphasize this distinction, we use a separate notation: $\mathcal{M}$ represents a set (or, in the uncountable case, *space*) of hypotheses, which we will call a *model class*. We implicitly assume that in all cases $\mathcal{M}$ contains at least two elements. Sequential Bayesian updating in this way is an *inductive* process; we refine our models based on observation. As we will see in Section 2.3, the predictive distribution $\Pr(B)$ will play an important role for our reinforcement learning agents.

We formalize Cromwell's rule with the concept of a *universal prior*.

Definition 1 (Universal prior)**.** A *prior* over a countable class of objects $\mathcal{M}$ is a probability mass function $p \in \Delta\mathcal{M}$, such that $p(\nu)$ is defined for each $\nu \in \mathcal{M}$, with $p(\nu) \in [0,1]$ and $\sum_{\nu\in\mathcal{M}} p(\nu) = 1$. A *universal prior* assigns non-zero mass to every hypothesis such that $p(\nu) \in (0,1)$ for all $\nu \in \mathcal{M}$.

We often make use of the following distributions:

Bernoulli. We use $\mathrm{Bern}(\theta)$ to represent the Bernoulli process on $x \in \{0,1\}$, with probability mass function given by $p(x|\theta) = \theta^x(1-\theta)^x$.

Binomial. We use $\mathrm{Binom}(n,p)$ to represent the Binomial distribution on $k \in \{0,\dots,n\}$ with mass function given by $p(k|n,p) = \binom{n}{k}p^k(1-p)^k$, where $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ is

the known as the *binomial coefficient*.

Uniform. We use $\mathcal{U}(a, b)$ to represent the measure that assigns uniform density to the closed interval $[a, b]$, with $b > a$; its density is given by $p(x) = \frac{1}{b-a}\mathbb{I}[a \leq x \leq b]$. We overload our notation (and nomenclature) and also use $\mathcal{U}(\mathcal{A})$ to represent the uniform distribution over the finite set $\mathcal{A}$; its mass function is given by $p(a) = \frac{1}{|\mathcal{A}|}$.

Normal. We use $\mathcal{N}(\mu, \sigma^2)$ to represent the univariate Gaussian distribution on $\mathbb{R}$ with density given by

$$p(x|\mu, \sigma^2) = (2\pi\sigma^2)^{-\frac{1}{2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

Beta. We use $\text{Beta}(\alpha, \beta)$ to represent the Beta distribution on $[0, 1]$ with density given by

$$p(x|\alpha, \beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1},$$

where Γ is the Gamma function that interpolates the factorials.

The beta distribution is conjugate to the Bernoulli and Binomial distributions; this means that a Bayesian updating scheme can use a Beta distribution as a prior $p(\theta)$ over the parameter of some Bernoulli process, whose likelihood is given by $p(x|\theta)$. Since the Beta and Bernoulli are conjugate, the posterior $p(\theta|x)$ will also take the form of a Beta distribution. Conjugate pairs of distributions such as this allow us to analytically compute the posterior resulting from a Bayesian update, and are essential for tractable Bayesian learning.

Dirichlet. We use $\text{Dirichlet}(\alpha_1, \ldots, \alpha_K)$ to represent the Dirichlet distribution on the 1-simplex

$$S_K \doteq \left\{\boldsymbol{x} \in \mathbb{R}_+^K \;\middle|\; \mathbf{1}^T\boldsymbol{x} = 1\right\},$$

with density given by

$$p(\boldsymbol{x}|\alpha) = \frac{\Gamma\left(\sum_{i=1}^K \alpha_i\right)}{\prod_{i=1}^K \Gamma(\alpha_i)} \prod_{i=1}^K x_i^{\alpha_i - 1}.$$

This is the multidimensional generalization of the Beta distribution, and is conjugate to the Categorical and Multinomial distributions. The categorical distribution over some discrete set $\mathcal{X}$ is simply a vector on the 1-simplex, $\boldsymbol{p} \in S_K$, where $K = |\mathcal{X}|$. The multinomial simply generalizes the binomial distribution.

As we shall see in Section 2.3, a significant aspect of intelligence is *sequence prediction*. For this reason, we introduce measures over sequences. A distribution over finite sequences $\rho \in \Delta\mathcal{X}^*$ can be written as $\rho(x_{1:n})$ for some finite n. Analogously to the sum and product rules, we have

$$\begin{aligned} \rho(x_n|x_{<n}) &= \frac{\rho(x_{1:n})}{\rho(x_{<n})} \\ \rho(x_{<n}) &= \sum_{y\in\mathcal{X}} \rho(x_{<n}y). \end{aligned}$$

There are two important properties that sequences can have which are relevant to reinforcement learning: the Markov and ergodic properties:

Markov property. A generative process ρ is n^{th}-order Markov if it has the property

$$\rho\left(x_t|x_{<t}\right) = \rho\left(x_t|x_{(t-n):(t-1)}\right).$$

Typically, when we invoke the Markov property, we mean that the process is 1^{st}-order Markov. A *Markov chain* is simply a first-order Markov process over a finite alphabet $\mathcal{X}$, in which the conditional distribution is *stationary*, and can thus be represent as a $|\mathcal{X}| \times |\mathcal{X}|$ transition matrix $P\left(x'|x\right) \equiv \rho\left(x'_t|x_{t-1}\right)$. This matrix is said to be *stochastic*, to emphasise that it represents a distribution over x', so that $P\left(x'|x\right) \in [0,1]$ and $\sum_{s'} P\left(x'|x\right) = 1$. In this context, we often identify the *symbols* $x \in \mathcal{X}$ with *states.*

Ergodicity. In a Markov chain, a state i is said to be ergodic if there is non-zero probability of leaving the state, and the probability of eventually returning is unity. If all states are ergodic, then the Markov chain is ergodic. Informally, this means that the Markov chain has no traps: at all times, we can freely move around the MDP without ever making unrecoverable mistakes. Ergodicity is an important assumption in the theory of Markov Decision Processes, which we will see later.

2.1.3 Information theory

For a distribution $p \in \Delta\mathcal{X}$ over a countable set $\mathcal{X}$, the *entropy* of p is

$$\operatorname{Ent}\left(p\right) \doteq -\sum_{x\in\mathcal{X}\,:\,p(x)>0} p\left(x\right)\log p\left(x\right). \tag{2.4}$$

Absent additional constraints, the maximum entropy distribution is $\mathcal{U}$, our generalized uniform distribution. We also define the conditional entropy

$$\operatorname{Ent}\left(p\left(\cdot|y\right)\right) \doteq \sum_{x\in\mathcal{X}\,:\,p(x)>0} p\left(x|y\right)\log p\left(x|y\right).$$

Given two distributions $p, q \in \Delta\mathcal{X}$, the *Kullback-Leibler divergence* (KL-divergence, also known as relative entropy) is defined by

$$\operatorname{KL}\left(p\|q\right) \doteq \sum_{x\in\mathcal{X}\,:\,p(x)>0,q(x)>0} p\left(x\right)\log\frac{p\left(x\right)}{q\left(x\right)}.$$

We use the $\|$ symbol to separate the arguments so as to emphasise that the KL-divergence is not symmetric, and hence not a distance measure. It is non-negative, by Gibbs' inequality. If p and q are measures over sequences, then we can define the conditional d-step KL-divergence

$$\operatorname{KL}_d\left(p, q|x_{<t}\right) = \sum_{x_{t:t+d}\in\mathcal{X}^d} p\left(x_{1:(t+d)}|x_{<t}\right)\log\frac{p\left(x_{1:(t+d)}|x_{<t}\right)}{q\left(x_{1:(t+d)}|x_{<t}\right)}.$$

2.2 Reinforcement Learning

In contrast to machine learning, in the reinforcement learning setting, the training data that the system receives is now dependent on its *actions*; we thus introduce *agency* to the learning problem (Sutton and Barto, 1998). What observations the agent can make, and therefore what it can learn, now depend not only on the environment (as in machine

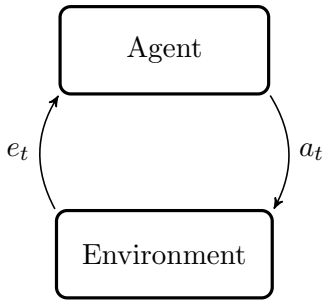

Figure 2.1: Cybernetic model of agent-environment interaction.

learning), but also on the agent's own *policy*, which determines how it will behave (Barto and Dietterich, 2004). In this way, reinforcement learning considerably generalizes machine learning; we replace the loss function of Equation (1.1) with a *reward signal*. Now, instead of minimizing *risk*, the agent must seek to maximize future expected *rewards*. In this way, reinforcement learning generalizes machine learning to the active setting, so that the agent can now influence its environment with the actions that it takes.

We distinguish this from the related set-up known as *inverse* reinforcement learning or *imitation learning* (Abbeel and Ng, 2004), in which the agent is given training data consisting of a history of actions and percepts from which it must infer a policy. In contrast, reinforcement learners must take their own actions and learn through trial and error – they are only *supervised* to the extent that their extrinsic reward signal gives them feedback on their policy.

Because we are motivated by the *general reinforcement learning problem*, we introduce a more general and pedantic setup than is common in the reinforcement learning literature. This set-up has been honed by (for example) Lattimore and Hutter (2011) and Leike et al. (2016).

2.2.1 Agent-environment interaction

In the standard *cybernetic model* (see Figure 2.1), the agent and environment are separate entities that play a turn-based two-player game. At time t, the agent produces an *action* a_t, which is passed as an input to the environment, which performs some computation that (in general) changes its internal state, and then returns a *percept* e_t to the agent. We often refer to the time t as the number of agent-environment *cycles* that have elapsed. Together, the agent and environment generate a *history* $æ_{1:t} = a_1 e_1 \dots a_t e_t$. In general, it is consequential to the behavior of the agent whether this interaction runs indefinitely or finishes after some finite lifetime T (Martin et al., 2016); we discuss this to an extent when we introduce discount functions in section 2.2.2.

Definition 2 (Environment)**.** An *environment* is a tuple $(\mathcal{A}, \mathcal{S}, \mathcal{E}, D, \nu)$, where

- $\mathcal{A}$ is the *action* space,
- $\mathcal{S}$ is the *state* space of the environment, which is in general hidden from the agent.
- $\mathcal{E}$ is the *percept* space, which is itself composed of observations $o \in \mathcal{O}$ and rewards $r \in \mathcal{R}$ with $\mathcal{E} = \mathcal{O} \times \mathcal{R}$.

- $D : \mathcal{S} \times \mathcal{A} \rightsquigarrow \mathcal{S}$ is the (in general stochastic) dynamics/transition function on the environment's state space. Note that, without loss of generality, we can allow $\mathcal{D}$ to be first-order Markov.

- $\rho : \mathcal{S} \to \Delta\mathcal{E}$ is the percept function, by analogy to a hidden Markov model (HMM) in the context of statistical machine learning[2].

Note that for the purposes of General reinforcement learning (GRL), we make no Markov assumption on the percepts, and we make no ergodicity assumption on the state or percept spaces.

Since we typically take the agent's perspective, we don't have access to the environment's state $s \in \mathcal{S}$, nor its dynamics D. For this reason, we typically talk about the environment in terms of the measure

$$\nu : (\mathcal{A} \times \mathcal{E})^* \times \mathcal{A} \to \Delta\mathcal{E},$$

which we write

$$\nu\left(e_t | æ_{<t} a_t\right).$$

Note that the vertical bar $|$ is an abuse of notation here: ν is not *conditioned* on the actions, since it is not derived from a joint distribution over actions and percepts; the sequence of actions $a_{1:t}$ are *inputs* to the environment. A more pedantic (but ugly) notation would be to write $\nu\left(e_t | e_{<t} \| a_{1:t}\right)$ or $\nu\left(e_t | e_{<t}; a_{1:t}\right)$, which emphasizes that ν is a conditional distribution with respect to percepts, but not with respect to actions. We typically refer to the environment itself with the symbol ν, for convenience.

It is worth pausing to make some remarks about this setup here:

1. In the general setting, environments are *partially-observable Markov decision processes* (POMDPs). We can always model an environment as Markovian with respect to *some* hidden state, since if it depends on some history of states, we incorporate sufficient history into the state until the Markov property is restored.

2. For our purposes, we assume that $\mathcal{A}$, $\mathcal{E}$, and $\mathcal{S}$ are all finite.

3. No matter what state the agent is in, it always has the full action space available to it. This simplifies the setup, and means that when implementing a simulated environment, we have to specify dynamics for every action in every state – 'illegal' or not. For an example of this, see Example 1.

4. Stochastic environments are sufficiently general to model everything, including Nature, adversaries, and naturally, deterministic environments.

5. This is an implicitly *dualistic* model, in the sense that the agent is separate from the environment; in reality the agent will be embedded within the environment.

6. As with all simulations run on computers, time is of course discretized.

7. We stipulate that our environments have the *chronological* property, which simply means that percepts at time t do not depend on future actions, i.e. $\nu\left(e_{1:t} \| a_{1:\infty}\right) = \nu\left(e_{1:t} \| a_{1:t}\right)$.

[2] POMDPs are to MDPs as Hidden Markov Models are to Markov chains.

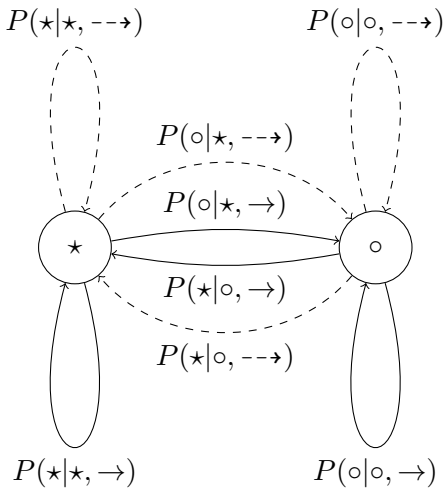

Figure 2.2: A generic finite-state Markov Decision Process with two states and two actions: $\mathcal{S} = \{\star, \circ\}$, $\mathcal{A} = \{\rightarrow, \dashrightarrow\}$. The transition matrix $P(s'|s, a)$ is a $2 \times 2 \times 2$ stochastic matrix, and the reward matrix $R(s, a)$ is 2×2.

The agent-environment interaction is thus modelled as a stochastic, imperfect-information, two-player game. The environment specifies both the percept space $\mathcal{E}$ and action space $\mathcal{A}$. The agent 'plugs in' to the environment (which, without loss of generality, can be thought of as a game simulation) and plays its moves in turns.

We now present some definitions of common classes of environments. For a more comprehensive taxonomy, see, for example, Legg (2008).

Definition 3 (Markov Decision Process)**.** A finite-state Markov decision process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ where

- $\mathcal{S}$ is a finite state space, labelled by indices $s_1, \ldots, s_{|\mathcal{S}|}$.
- $\mathcal{A}$ is a finite action space, labelled by indices $a_1, \ldots, a_{|\mathcal{A}|}$.
- $\mathcal{P}$ is the set of transition probabilities $P(s'|s, a)$, which can be thought of as a stochastic rank-3 tensor of dimensions $|\mathcal{S}| \times |\mathcal{S}| \times |\mathcal{A}|$
- $\mathcal{R}$ is the set of rewards $R(s, a)$.

Definition 4 (Bandit)**.** An N-armed bandit is a Markovian environment with one state $\mathcal{S} = \{s\}$, N actions $\mathcal{A} = \{a_1, \ldots, a_N\}$ and N corresponding reward distributions $\{\rho_1, \ldots, \rho_n\}$ with $\rho_i \in \Delta\mathcal{R}$. There are no observations, only a reward signal which is sampled from the distribution ρ_i corresponding to the agent's last action, a_i. Typical choices for ρ, $\mathcal{R}$ are Bernoulli distributions over $\{0, 1\}$, or Gaussians over $\mathbb{R}$; see Figure 2.2.1.

Example 1 (Go)**.** Go is a two-player, deterministic, and fully-observable[3] board game with a large, finite state-action space. Played on a 19×19 board, there are (naively) 3^{19^2}

[3]Here we are referring to the *game state* being fully observable. The opponent's strategy can of course be modelled as some hidden variable; for simplicity assume that we model them as minimax, so that there is no hidden state.

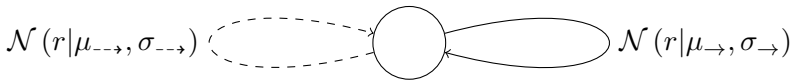

Figure 2.3: A two-armed Gaussian Bandit. $\mathcal{A} = \{\rightarrow, \dashrightarrow\}$, $|\mathcal{S}| = 1$, and $\mathcal{O} = \emptyset$. Rewards are sampled from the distribution of the respective arm.

possibly game states, with an action space of 19^2 (though many of these moves will be illegal). It is notoriously hard to evaluate who is winning in any given game state (Silver et al., 2016); the reward signal is 0 for all game states for which a winner has not been declared, and ± 1 otherwise (depending on which player won).

Definition 5 (Policy)**.** A *policy* is, in the most general setting, a probability distribution over actions, conditioned on a history: $\pi(a_t|æ_{<t}) \ : \ (\mathcal{A} \times \mathcal{E})^* \rightarrow \Delta\mathcal{A}$.

Note the symmetry between Definition 5 with $\nu(e_t|æ_{<t}a_t)$ from Definition 2.

Definition 6 (Agent)**.** Let $\Pi_{\mathcal{A},\mathcal{E}}$ be the set of all policies on the $(\mathcal{A}, \mathcal{E})$-space. An *agent* is fully characterized by a policy π, and a *learning algorithm*, which is as a mapping from experience (histories) to policies $(\mathcal{A} \times \mathcal{E})^* \rightarrow \Pi_{\mathcal{A},\mathcal{E}}$.

The agent and environment, combined, induce a distribution over histories. We denote this by $\nu^\pi \in \Delta(\mathcal{A} \times \mathcal{E})^*$. This is equivalent to the *state-action visit distribution* in the standard reinforcement learning literature (Sutton et al., 1999).

$$\nu^\pi(æ_{<t}) = \prod_{k=1}^{t} \pi(a_k|æ_{<t})\, \nu(e_k|æ_{<k}) \tag{2.5}$$

The distribution ν^π plays an important role in the theory of GRL, since we will use it to compute the expected sum of future rewards, which is what our reinforcement learners will seek to maximize.

2.2.2 Discounting

In the context of reinforcement learning, we wish our agent to act according to a policy that maximizes reward accumulated over its lifetime. In general it is not good enough to greedily maximize the reward obtained in the next time-step, since in many cases this will lead to reduced total reward. Thus we define the *return* resulting from executing policy π in environment ν from time t as the sum of future rewards r_i.

$$R_\nu^\pi(æ_{<t}) = \sum_{k=t}^{\infty} r_k,$$

where each of the r_k are sampled from $\nu(\cdot|\pi, æ_{<k})$; thus the return is a random variable that depends on the agent policy π, the environment ν, and the history $æ_{<t}$. In general this sum will diverge, so in practice we concern ourselves with either the *average reward*

$$\bar{r}_\nu^\pi = \lim_{n\rightarrow\infty} \frac{1}{n} \sum_{k=t}^{n} r_k,$$

or the *discounted return*

$$R^{\pi}_{\nu\gamma}(æ_{<t}) = \sum_{k=1}^{\infty} \gamma^t_k r_k,$$

where $\gamma^t_k \leq 1$ is some generalized *discount function* $\gamma^t : \mathbb{N} \to [0,1]$ such that

$$\Gamma^t_\gamma \doteq \sum_{k=t}^{\infty} \gamma^t_k < \infty.$$

Here we interpret t as the current age of the agent, and k is the agent's planning look-ahead.

Definition 7 (ε-Effective horizon; Lattimore and Hutter (2014a))**.** Given a discount function γ, the *ε-effective horizon* is given by

$$H^t_\gamma(\varepsilon) \doteq \min\left\{ H \;:\; \frac{\Gamma^{t+H}_\gamma}{\Gamma^t_\gamma} \leq \varepsilon \right\}. \tag{2.6}$$

The ε-effective horizon represents the distance ahead in the future that the agent can plan while still taking into account a proportion of the available return equal to $(1-\varepsilon)$. The choice of discount function is relevant to how the agent plans; some discount functions will make the agent far-sighted, and others will make it near-sighted. We discuss planning more in Section 2.4, and present experiments relating to this in Chapter 4. A common choice of discount function is the *geometric* discount function, which is ubiquitous in RL due to its simplicity:

$$\gamma^t_k = \beta^k,$$

for some $\beta \in [0,1]$. That is, it is β raised to the number of cycles that we look ahead in planning.

2.2.3 Value functions

In general, the environment ν is noisy and stochastic, and the agent's policy will often be stochastic. As a result, we can't maximize the discounted return directly; we must instead maximize it in expectation. This follows from the Von Neumann-Morgenstern utility theorem (Morgenstern and von Neumann, 1944).

Definition 8 (Value function)**.** The *value* $V^{\pi}_{\nu\gamma} : (\mathcal{A}\times\mathcal{E})^* \to \mathbb{R}$ of a history $æ_{<t}$ in environment ν under policy π with discount function γ is the *expected sum of discounted future rewards*

$$V^{\pi}_{\nu\gamma}(æ_{<t}) \doteq \mathbb{E}^{\pi}_{\nu}\left[\sum_{k=t}^{\infty} \gamma^t_k r_k \,\middle|\, æ_{<t} \right], \tag{2.7}$$

where we use $\mathbb{E}^{\pi}_{\nu}$ above to mean the expectation with respect to ν^{π}, defined in Equation (2.5). Equation (2.7) above expresses the value function in *iterative* form. We can also express it recursively (Leike, 2016a) using the mutually recursive relations

$$
\begin{aligned}
V^{\pi}_{\nu\gamma}(æ_{<t}) &= \sum_{a_t \in \mathcal{A}} \pi(a_t|æ_{<t})\, V^{\pi}_{\nu\gamma}(æ_{<t}a_t) \\
V^{\pi}_{\nu\gamma}(æ_{<t}a_t) &= \frac{1}{\Gamma_t} \sum_{e_t \in \mathcal{E}} \nu(e_t|æ_{<t}a_t) \left[\gamma_t r_t + \Gamma_{t+1} V^{\pi}_{\nu\gamma}(æ_{1:t})\right].
\end{aligned}
$$

For simplicity, from here on we will often omit the γ subscript and make the dependence on the discount function implicit. We will also suppress the normalization $\frac{1}{\Gamma_t}$, as it clutters the notation and is introduced for technical reasons (so that value is normalized). Finally, we often also suppress the history $æ_{<t}$ for clarity.

Definition 9 (Optimal value & policy)**.** The *optimal value* V^*_ν achievable in environment ν given a history $ae_{<t}$ is

$$V^*_\nu \doteq \max_\pi V^\pi_\nu, \tag{2.8}$$

and the corresponding *optimal policy* π^* *is*

$$\pi^*_\nu = \arg\max_\pi V^\pi_\nu.$$

Assuming bounded rewards and finite action spaces, these maxima exist for all ν (Lattimore and Hutter, 2014b), though they are not unique in general. For our purposes, we allow arg max to break ties at random. At this point it is elucidatory to unroll Equation (2.8) into the *expectimax* expression

$$V^*_\nu(æ_{<t}) = \lim_{m\to\infty} \max_{a_t} \sum_{e_t} \cdots \max_{a_{t+m}} \sum_{e_{t+m}} \sum_{k=t}^{t+m} \gamma_k^t r_k \prod_{j=t}^{k} \nu(e_j|æ_{<j}a_j). \tag{2.9}$$

Note that we can do this by using the distributive property of max over $+$. In Section 2.4, we will discuss how to approximate this expectimax calculation for general environments, up to a finite horizon m.

2.2.4 Optimality

Informally, it makes sense to evaluate an agent's performance against that of the optimal policy, were it put in the same situation. We can only sensibly talk about this performance *asymptotically* in general, that is, in the limit $t \to \infty$, since the agent needs time to learn the environment, and we can't evaluate the agent after some finite time t, since this time would in general be environment-dependent.

Definition 10 (Asymptotic optimality; Lattimore and Hutter, 2011)**.** A policy π is *strongly asymptotically optimal* in environment class $\mathcal{M}$ if $\forall \mu \in \mathcal{M}$

$$\mu^\pi\left(\lim_{t\to\infty} \left\{V^*_{\mu\gamma}(æ_{<t}) - V^\pi_{\mu\gamma}(æ_{<t})\right\} = 0\right) = 1,$$

where μ^π is the measure induced by the interaction of environment μ with policy π. The policy π is *weakly asymptotically optimal* in $\mathcal{M}$ if $\forall \mu \in \mathcal{M}$

$$\mu^\pi\left(\lim_{n\to\infty} \frac{1}{n} \sum_{t=1}^{n} \left\{V^*_{\mu\gamma}(æ_{<t}) - V^\pi_{\mu\gamma}(æ_{<t})\right\} = 0\right) = 1.$$

Finally, we say π is *asymptotically optimal in mean* over $\mathcal{M}$ if $\forall \mu \in \mathcal{M}$

$$\lim_{t\to\infty} \mathbb{E}^{\pi}_{\mu}\left[V^{*}_{\mu\gamma}\left(æ_{<t}\right) - V^{\pi}_{\mu\gamma}\left(æ_{<t}\right)\right] = 0.$$

Asymptotic optimality is objective and general, but unfortunately doesn't capture everything we want in an agent. For example, in environments with traps – that is, an accepting state with no transitions leaving it and very low reward – every policy will be asymptotically optimal after falling into the trap, since no policy will outperform any other, conditioned on being trapped. Moreover, in uncertain environments with traps, an agent cannot be asymptotically optimal *unless* it is sufficiently gung-ho in its exploration that it eventually falls into traps (Leike, 2016a). Therefore, we should take asymptotic optimality with a grain of salt; it is not a particularly good measure of optimality in general environments. The quest for good notions of optimality is currently an open problem in the theory of GRL (Leike and Hutter, 2015; Leike, 2016a).

2.3 General Reinforcement Learning

We now introduce the agents that are central to the theory of general reinforcement learning (GRL). We begin with AIμ, which is simply the policy of the informed agent that has a perfect model of the environment μ:

Definition 11 (AIμ)**.** AIμ corresponds to the policy in which the true environment μ is known to the agent, and so no learning is required. Behaving optimally reduces to the planning problem of computing the μ-optimal policy

$$\begin{aligned} \pi^{AI\mu} &= \pi^{*}_{\mu} \\ &\doteq \arg\max_{\pi} V^{\pi}_{\mu}. \end{aligned}$$

The astute reader will notice that $\pi^{AI\mu}$ is simply the optimal policy for environment μ; we introduce it here as a separate agent so as to have a benchmark against which to compare our other reinforcement learners.

In general the environment will be unknown, and so our agents will have to learn it. For the purpose of studying the general reinforcement learning problem, we consider primarily Bayesian agents, as they are the most general and principled way to think about the problem of induction (Hutter, 2005).

2.3.1 Bayesian agents

Our Bayesian agents maintains a *Bayesian mixture* or *predictive distribution* ξ over a countable model class $\mathcal{M}$, given by

$$\begin{aligned} \xi\left(e_t|æ_{<t}a_t\right) &= \sum_{\nu\in\mathcal{M}} w_\nu \nu\left(e_t|æ_{<t}a_t\right) \\ &= \sum_{\nu\in\mathcal{M}} \Pr\left(\nu|æ_{<t}a_t\right)\Pr\left(e_t|\nu, æ_{<t}a_t\right). \end{aligned} \tag{2.10}$$

Note that ξ is equivalent to the normalization term in Theorem 1; $\xi\left(e|\cdot\right)$ represents the probability that the agent's model assigns to e; in other words, it is the agent's *predictive*

distribution. The *weights* $w_\nu \equiv w(\nu) \equiv \Pr(\nu)$ represent the agent's credence in hypothesis $\nu \in \mathcal{M}$. $\nu(e|æ_{<t}a_t)$ is the probability that model ν assigns to percept e, given history $æ_{<t}a_t$. One can think of the hypothesis/model ν as a *latent variable* in the model, which is marginalized out to get the predictive distribution. The only strong assumption we make in this setup is that the *true environment* μ is contained in $\mathcal{M}$. Given a new percept $e = (o, r)$, the Bayesian updates its weights model using Bayes rule:

$$w(\nu|e) = \frac{w(\nu)\,\nu(e)}{\xi(e)}.$$

This gives us the very natural updating scheme

$$w_\nu \leftarrow w_\nu \frac{\nu(e)}{\xi(e)}.$$

Note that above we have suppressed the history $æ_{<t}a_t$ for clarity.

That is, given a new percept e, we compute our posterior by multiplying the prior by the likelihood ratio. Let us pause and make some remarks:

1. We see that Bayesian induction generalizes the Popperian idea of *conjecture and refutation*. An environment/model/hypothesis ν is *falsified* by observation/perception/experience/experiment e *iff* $\nu(e) = 0$. Clearly, the posterior goes to zero for these environments.

2. Bayesian induction is parameter free up to a choice of model class $\mathcal{M}$ and prior $w(\nu|\epsilon)$.

3. We use the words *environment*, *model*, and *hypothesis* interchangeably.

4. We can see by its 'type signature' that ξ itself is an environment.

Item 4 above underpins the Bayes-optimal agent AIξ.

Definition 12 (AIξ)**.** AIξ computes the ξ-optimal policy, i.e.

$$\pi^{AI\xi} \doteq \arg\max_\pi V^\pi_\xi. \tag{2.11}$$

That is, AIξ uses the policy that is optimal in the mixture environment ξ, which we update with percepts from the true environment μ using Bayes' rule.

For the purpose of reasoning about general artificial intelligence, we use the largest model class we can, which is $\mathcal{M}_{\text{comp}}$, the class of all computable environments[4]. This is what the famous agent AIXI does:

Definition 13 (AIXI)**.** AIXI is AIξ with the model class given by $\mathcal{M}_{\text{comp}}$ and the Solomonoff prior

$$w_\nu = 2^{-K(\nu)},$$

where $K(\nu)$ is the *Kolmogorov complexity* of ν. For a string x, the Kolmogorov complexity is given by

$$K(\nu) \doteq \min\{|p| \mid U(p) = x\},$$

[4]For technical reasons, the literature typically uses $\mathcal{M}^{LSC}_{CCS}$, the class of lower semi-computable chronological conditional semimeasures. This distinction is a technical one and of little consequence to us.

where U is a universal Turing machine (Li and Vitányi, 2008). For every computable environment ν, there is a corresponding Turing machine T, so we can define the $K(\nu)$ as the Kolmogorov complexity of its index in the enumeration $\nu_1, \nu_2, \ldots$ of all environments. The Kolmogorov complexity is, of course, incomputable.

This gives rise to the famous equation describing the AIXI policy, unrolled in all its incomputable glory:

$$a_t^{\text{AIXI}} = \arg\max_{a_t} \lim_{m\to\infty} \sum_{e_t} \cdots \max_{a_{t+m}} \sum_{e_{t+m}} \sum_{k=t}^{t+m} \gamma_k^t r_k \sum_{p \,:\, U(p,a_{<t})=e_{1:j}} 2^{-|p|}.$$

One can derive computable approximations of Solomonoff induction, most notably by using a generalization of the Context-Tree Weighting algorithm, which is a mixture over Markov models up to some finite order n, weighted by their complexity; this is used in the well-known MC-AIXI-CTW implementation due to Veness et al. (2011).

AIXI achieves *on-policy value convergence* (Leike, 2016a):

$$\mu^\pi \left(\lim_{t\to\infty} \left[V_\xi^\pi - V_\mu^\pi \right] = 0 \right) = 1,$$

which means that it asymptotically learns the true value of its policy π in environment μ. It however, doesn't achieve asymptotic optimality.

Theorem 2 (*AIXI is not asymptotic optimal; Orseau, 2010; Leike, 2016a*)**.** *For any class $\mathcal{M} \supseteq \mathcal{M}_{comp}$ no Bayes optimal policy π_ξ^* is asymptotically optimal: there is an environment $\mu \in \mathcal{M}$ and a time step $t_0 \in \mathbb{N}$ such that for all time steps $t \geq t_0$*

$$\mu^{\pi_\xi^*} \left(V_\mu^* (æ_{<t}) - V_\mu^{\pi_\xi^*} (æ_{<t}) = \frac{1}{2} \right) = 1.$$

This theorem effectively means that the Bayes agent will eventually decide that its current policy is good enough, and that any additional exploration is not worth its Bayes-expected payoff. Moreover, AIξ can be made to perform badly with a so-called dogmatic prior:

Theorem 3 (*Dogmatic prior;* Leike and Hutter, 2015)**.** *Let π be some computable policy, ξ some universal mixture, and let $\varepsilon > 0$. There exists a universal mixture ξ' such that for any history h consistent with π and $V_\xi^\pi(h) > \varepsilon$, the action $\pi(h)$ is the unique ξ'-optimal action.*

This theorem says that, even using a universal prior that assigns non-zero mass to every hypothesis in the model class, we can construct a prior in such a way that the agent never overcomes the bias in its prior. This is in contrast to Bayesian learners in the passive setting, which can overcome (given sufficient data) any biases in their (universal) prior. We demonstrate in Chapter 4 an example of a dogmatic prior that prevents the Bayesian agent from exploring.

2.3.2 Knowledge-seeking agents

We now come to our first exhibit in the GRL agent zoo: knowledge-seeking agents (KSA). There are several motivations for defining and studying KSA:

- They represent a way to construct a purely 'exploratory' policy. A principled solution to exploration by intrinsic motivation is one of the central problems in reinforcement learning (Thrun, 1992).

- They remove the dependence on arbitrary reward signals or utility functions; up to a choice of model class and prior, 'knowledge' is an objective quantity (Orseau, 2011).

- They collapse the exploration-exploitation trade-off to *just* exploration.

Before formally defining knowledge-seeking agents, it is necessary to introduce the concept of a *utility agent*, which generalizes the concept of a reinforcement learning agent.

Definition 14 (Utility Agent; Orseau, 2011)**.** A *utility agent* is a reinforcement learner equipped with a bounded utility function $u : (\mathcal{A} \times \mathcal{E})^* \times \mathcal{A} \to \mathbb{R}$ which replaces the notion of *reward*. The corresponding value function[5] is given by

$$V^{\pi}_{\nu\gamma}(æ_{<t}) = \mathbb{E}^{\pi}_{\nu}\left[\sum_{k=t}^{\infty} \gamma^t_k u(æ_{1:k}) \,\middle|\, æ_{<t}a_t\right]. \tag{2.12}$$

One can easily verify that this definition generalizes RL agents by setting $u_{\mathrm{RL}}(æ_{1:t}) = r(e_t)$, where $r(\cdot)$ returns the second component of the percept tuple $e_t = (o_t, r_t)$. Utility agents are fully autonomous, in the sense that they are not dependent on being 'supervised' by an extrinsic reward signal to learn. They are equipped with a utility function at birth and from then on seek to maximize the discounted sum of future utility.

Knowledge-seeking agents (KSA) are Bayesian utility agents whose utility function is constructed in such a way as to motivate them to 'seek knowledge' and learn about their environment (Orseau, 2011, 2014). There are several distinct ways in which one can define knowledge for a Bayesian agent. We will start by defining an agent that gets utility from lowering the entropy (i.e., reducing uncertainty) in its beliefs. We can define the *information gain* resulting from some percept e as the difference in entropy between the agent's prior and posterior:

$$\mathrm{IG}(e) \doteq \mathrm{Ent}(w(\cdot)) - \mathrm{Ent}(w(\cdot|e)), \tag{2.13}$$

Now, following Lattimore (2013), we consider the ξ-expected information gain. Informally, this is the information that the agent expects to obtain were the percepts distributed according to its mixture model ξ. It is also the agent's expected utility from seeing percept

[5] Compare with Equation (2.7).

e. For clarity, we suppress the history $æ_{<t}a_t$ and time subscripts.

$$\begin{aligned}
\mathbb{E}_\xi\left[\mathrm{IG}\left(e\right)\right] &= \sum_{e\in\mathcal{E}}\xi\left(e\right)\left[\mathrm{Ent}\left(w\left(\cdot\right)\right)-\mathrm{Ent}\left(w\left(\cdot|e\right)\right)\right] \\
&= \sum_{e\in\mathcal{E}}\xi\left(e\right)\sum_{\nu\in\mathcal{M}}\left[w\left(\nu|e\right)\log w\left(\nu|e\right)-w\left(\nu\right)\log w\left(\nu\right)\right] \\
&= \sum_{e\in\mathcal{E}}\xi\left(e\right)\sum_{\nu\in\mathcal{M}}\left[w\left(\nu\right)\frac{\nu\left(e\right)}{\xi\left(e\right)}\log w\left(\nu|e\right)-w\left(\nu\right)\log w\left(\nu\right)\right] \\
&= \sum_{\nu\in\mathcal{M}}w\left(\nu\right)\sum_{e\in\mathcal{E}}\xi\left(e\right)\left[\frac{\nu\left(e\right)}{\xi\left(e\right)}\log w\left(\nu|e\right)-\log w\left(\nu\right)\right] \\
&= \sum_{\nu\in\mathcal{M}}w\left(\nu\right)\left[\sum_{e\in\mathcal{E}}\nu\left(e\right)\log w\left(\nu|e\right)-\log w\left(\nu\right)\right] \\
&= \sum_{\nu\in\mathcal{M}}w\left(\nu\right)\left[\sum_{e\in\mathcal{E}}\nu\left(e\right)\log w\left(\nu|e\right)-\sum_{e\in\mathcal{E}}\nu\left(e\right)\log w\left(\nu\right)\right] \\
&= \sum_{\nu\in\mathcal{M}}w\left(\nu\right)\left[\sum_{e\in\mathcal{E}}\nu\left(e\right)\log\frac{w\left(\nu|e\right)}{w\left(\nu\right)}\right] \\
&= \sum_{\nu\in\mathcal{M}}w\left(\nu\right)\left[\sum_{e\in\mathcal{E}}\nu\left(e\right)\log\frac{\nu\left(e\right)}{\xi\left(e\right)}\right] \\
&= \sum_{\nu\in\mathcal{M}}w\left(\nu\right)\mathrm{KL}\left(\nu\|\xi\right).
\end{aligned}$$

Thus, by maximizing the ξ-expected information gain, one maximizes the belief-weighted Kullback-Leibler divergence between ν and ξ.

Definition 15 (Kullback-Leibler-KSA; Orseau, 2014)**.** The *KL-KSA* is the Bayesian agent with

$$u_{\mathrm{KL}}\left(æ_{1:t}\right) = \mathrm{Ent}\left(w\left(\cdot|æ_{<t}\right)\right)-\mathrm{Ent}\left(w\left(\cdot|æ_{1:t}\right)\right). \tag{2.14}$$

Notice that the first term doesn't depend on e_t, so at any given time step it is fixed by the agent's past history. The term that matters is the second one, which is the negative entropy of the posterior beliefs, after updating on percept e_t. Intuitively, the agent gets reward from reducing the entropy (uncertainty) in its beliefs; it seeks out experiences e_t that will make it more certain about the world, and won't be satisfied until entropy is minimal – that is, when its beliefs converge to the truth such that $w_\nu = \mathbb{I}\left[\nu=\mu\right]$ and $\mathrm{Ent}\left(w\right)=0$. In the most general environment classes, this convergenge won't be possible, as there are many environments that are indistinguishable on-policy; in other words, there will always be hypotheses that the agent can't falsify. An example of this is the so-called *blue emeralds* hypothesis: 'Emeralds are green, but after next Tuesday, they will become blue'.

Theorem 4 (*Orseau, 2014*)**.** *KL-KSA is asymptotically optimal with respect to* u_{KL} *in general environments.*

So much for Kullback-Leibler KSA. There are other ways to construct a knowledge-seeker. To elucidate this, notice that the entropy in the Bayesian mixture ξ can be decomposed into contributions from *uncertainty* in the agent's beliefs w_ν and *noise* in the

environment ν. That is, given a mixture ξ and for some percept e such that $0 < \xi(e) < 1$, and suppressing the history $æ_{<t}a_t$ for clarity,

$$\xi(e) = \sum_{\nu \in \mathcal{M}} \overbrace{w_\nu}^{\text{uncertainty}} \underbrace{\nu(e)}_{\text{noise}}.$$

That is, if $0 < w_\nu < 1$, we say the agent is *uncertain* about whether hypothesis ν is true (assuming there is exactly one $\mu \in \mathcal{M}$ that is the truth). On the other hand, if $0 < \nu(e) < 1$ we say that the environment ν is *noisy* or *stochastic*. If we restrict ourselves to deterministic environments such that $\nu(e) \in \{0, 1\}\ \forall \nu\, \forall e$, then $\xi(\cdot) \in (0, 1)$ implies that $w_\nu \in (0, 1)$ for at least one $\nu \in \mathcal{M}$. This motivates us to define two agents that seek out percepts to which the mixture ξ assigns low probability; in deterministic environments, these will behave like knowledge-seekers.

Definition 16 (Square-KSA; Orseau, 2011)**.** The *Square-KSA* is the Bayesian agent with utility function given by

$$u_{\text{Square}}(e_t|æ_{<t}) = -\xi(e_t|æ_{<t}). \tag{2.15}$$

Definition 17 (Shannon-KSA; Orseau, 2011)**.** The *Shannon-KSA* is the Bayesian agent with utility function given by

$$u_{\text{Shannon}}(e_t|æ_{<t}) = -\log(\xi(e_t|æ_{<t})). \tag{2.16}$$

Theorem 5 (Orseau, 2014)**.** *Square-KSA and Shannon-KSA are strongly asymptotically optimal with respect to u_{Square} and $u_{Shannon}$ respectively, in deterministic environments.*

They are named 'Square' and 'Shannon', since in taking ξ-expectation of the utility functions we get

$$\begin{aligned}
\mathbb{E}_\xi[u_{\text{Square}}(\cdot|æ_{<t})] &= -\sum_{e_t \in \mathcal{E}} [\xi(e_t|æ_{<t})]^2 \\
\mathbb{E}_\xi[u_{\text{Shannon}}(\cdot|æ_{<t})] &= -\sum_{e_t \in \mathcal{E}} \xi(e_t|æ_{<t}) \log \xi(e_t|æ_{<t}) \\
&= \text{Ent}(\xi).
\end{aligned}$$

These are *entropy-seeking* agents, since they seek to maximize the discounted sum of expected utilities, which in both cases are entropies. Note from Figure 2.4 that u_{Square} and u_{Shannon} are approximately the same (up to an irrelevant additive constant) over the range $[0.5, 1]$. Their behaviors become significantly different for $\xi \to 0$: Shannon-KSA *loves* rare events, and the rarer the better; u_{Shannon} is unbounded from above on the interval as $\xi \to 0$. The Shannon-KSA, with its expected utility being measured in bits, is closely related to Schmidhuber's 'curiosity learning', which gets utility from making compression progress (Schmidhuber, 1991).

Square- and Shannon-KSA both fail in general for stochastic environments. We can see this by constructing an environment adversarially to 'trap' these agents and stop them from exploring: just introduce a noise generator that is sufficiently rich (i.e. is sampled from a uniform distribution over a sufficiently large alphabet of percepts) so that the probability of any single percept is low enough that it swamps the utility gained from

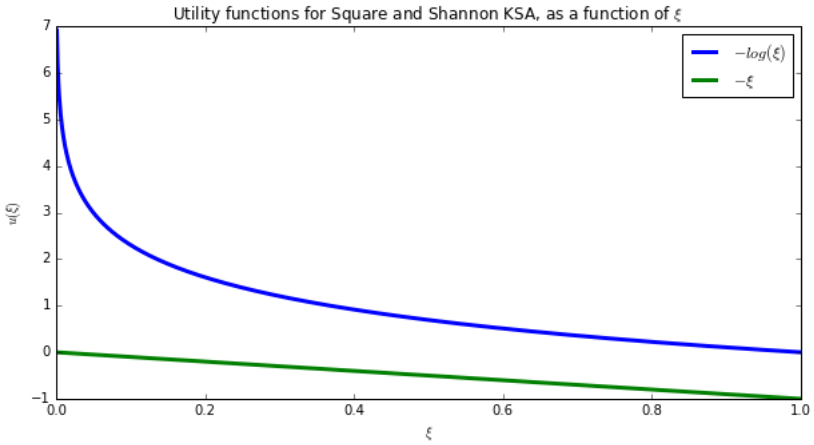

Figure 2.4: Square-KSA utility function plotted against that of Shannon-KSA.

exploring the rest of the world and gaining information. Thus, we can get the Square and Shannon KSAs 'hooked on noise' – they would be endlessly fascinated with a white noise generator such as a detuned television, and would never get sick of watching the random, low-probability events. We construct an experiment to explore this property of KSA agents in Chapter 4.

2.3.3 BayesExp

The idea behind the BayesExp agent is simple. Given that KL-KSA is effective at exploring, and AIξ is effective (by construction) at exploiting the agent's beliefs as they stand: why not combine the two in some way? The algorithm for running BayesExp is simple: run AIξ by computing the ξ-optimal policy as normal, but at all times compute the value of the information-seeking policy π^{KSA}. If the expected information gain (up to some horizon) exceeds some threshold ϵ, run the knowledge-seeking policy for an effective horizon. This combines the best of AIξ and KSA, by going on bursts of exploration when the agent's beliefs suggest that the time is right to do so; thus, BayesExp breaks out of the sub-optimal exploration strategy of Bayes, but without resorting to ugly heuristics such as ϵ-greedy. Crucially, it explores infinitely often, which is necessary for asymptotic optimality (Leike, 2016b).

Essentially, the BayesExp agent keeps track of two value functions: the Bayes-optimal value V^*_ξ, and the ξ-expected *information gain* value $V^*_{\xi,\text{IG}}$, which we obtain by substituting Equation (2.14) into Equation (2.12). It then checks whether $V^*_{\xi,\text{IG}}$ exceeds some threshold, ε_t. If it does, then it will explore for an effective horizon $H_t(\varepsilon_t)$, and otherwise it will exploit using the Bayes-optimal policy π^*_ξ. See Algorithm 2.1 for the formal algorithm.

Theorem 6 (Lattimore, 2013)**.** *With a finite prior w and a non-increasing exploration schedule $\varepsilon_1, \varepsilon_2, \ldots$, with $\lim_{t\to\infty} \varepsilon_t = 0$, BayesExp is asymptotically optimal in general environments.*

2.3.4 MDL Agent

While AIXI uses the principle of Epicurus to mix over all consistent environments, the minimum description length (MDL) agent greedily picks the simplest unfalsified environ-

Algorithm 2.1 BayesExp (Lattimore, 2013)

Inputs: Model class $\mathcal{M} \doteq \{\nu_1, \ldots, \nu_K\}$; $w : \mathcal{M} \to (0,1)$; exploration schedule $\{\varepsilon_1, \varepsilon_2, \ldots\}$.

$t \leftarrow 1$
loop
  $d \leftarrow H_t(\varepsilon_t)$
  if $V^*_{\xi,\mathrm{IG}}(æ_{<t}) > \varepsilon_t$ **then**
    for $i = 1 \to d$ **do**
      ACT $\left(\pi^{\star,\mathrm{IG}}_{\xi}\right)$
    end for
  else
    ACT $\left(\pi^{\star}_{\xi}\right)$
  end if
end loop

ment in its model class and behaves optimally with respect to that environment until it falsifies it. In other words, the policy is given by

$$\pi_{\mathrm{MDL}} = \arg\max_a V^*_\rho,$$

where

$$\rho = \operatorname*{arg\,min}_{\nu \in \mathcal{M} \,:\, w_\nu > 0} K(\nu).$$

Here, the Kolmogorov complexity K plays the role of a strongly weighted regularizer. That is, MDL chooses the policy that is optimal with respect to the simplest unfalsified environment. This algorithm will fail in stochastic environments, since there will exist environments which cannot be falsified (in the strict sense, i.e. $w_\nu = 0$) by any percept – for example, an environment in which the agent receives a video feed which is (even slightly) noisy.

Algorithm 2.2 MDL Agent (Lattimore and Hutter, 2011)

Inputs: Model class $\mathcal{M}$; prior $w : \mathcal{M} \to (0,1]$; a total ordering $\preceq$ over $\mathcal{M}$.

loop
  Select $\rho \leftarrow \min_{\preceq} \mathcal{M}$
  repeat
    ACT $\left(\pi^{\star}_{\rho}\right)$
  until $\rho(e_{<t}) = 0$
end loop

2.3.5 Thompson Sampling

Thompson sampling is a very common Bayesian sampling technique, named for Thompson (1933). In the context of general reinforcement learning, it can be used as another attempt at solving the exploration problems of AIξ. Informally, the idea is to use the ρ-optimal policy for an effective horizon, before re-sampling from the posterior $\rho \sim w(\cdot|æ_{<t})$ and repeating – at all times, the agent updates it posterior as usual. This commits the agent to a single hypothesis for a significant amount of time; one can think of it as testing likely

hypothesis one at a time. See Algorithm 2.3 for the formal description of the Thompson sampling policy π_T.

Theorem 7 (Leike et al., 2016)**.** *Thompson sampling is asymptotically optimal in mean in general environments.*

Algorithm 2.3 Thompson Sampling (Leike et al., 2016)

Inputs: Model class $\mathcal{M}$; prior $w : \mathcal{M} \to (0, 1]$; exploration schedule $\{\epsilon_1, \epsilon_2, \dots\}$.

$t \leftarrow 1$
loop
  Sample $\rho \sim w(\cdot|æ_{<t})$
  $d \leftarrow H_t(\epsilon_t)$
  for $i = 1 \to d$ **do**
    ACT $\left(\pi^\star_\rho\right)$
  end for
end loop

So much for our GRL agents. We now discuss how to compute the policy π^*_ρ for general environments ρ, discount functions γ, and utility functions u. This general-purpose planning algorithm will be used to select actions for all of the agents above.

2.4 Planning

We have discussed how Bayesian agents maintain a model of their environment, and update their models based on the percepts they receive. Of course, the other major aspect of artificial intelligence, distinct from *learning*, is *acting*. Recall that, if the environment is known, computing the optimal policy becomes a planning problem. In general (stochastic) environments, this involves computing the optimal value V^*_μ, which is the expectimax expression from Equation (2.9). In practice, with finite compute power, we must of course approximate this expectimax calculation up to some finite horizon m. For finite-state Markov decision processes with known transitions and rewards, and under geometric discounting, we can compute this by a simple dynamic programming algorithm called Value Iteration (section 2.4.1). For more general environments, we must approximate it by 'brute force', with Monte Carlo sampling (section 2.4.2).

2.4.1 Value iteration

In a finite-state MDP, if the state transitions $P(s'|s, a)$ and reward function $R(s, a)$ are known, then we can plan ahead by *value iteration*:

$$V_{n+1}(s) = \max_{a \in \mathcal{A}} Q_n(s, a), \tag{2.17}$$

with

$$Q_n(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} P\left(s'|s, a\right) V_n\left(s'\right). \tag{2.18}$$

This is as a *dynamic programming* algorithm, and is known that value iteration con-

verges to the value of the optimal policy (Sutton and Barto, 1998):

$$\lim_{n\to\infty} V_n(s) = V^*(s) \;\; \forall s \in \mathcal{S}.$$

Planning by value iteration relies heavily on two strong assumptions: the finite-state MDP assumption, and geometric discounting. We wish to be able to lift these assumptions for the purpose of our experiments in GRL, so we move our attention now to planning by Monte Carlo techniques.

2.4.2 MCTS

Monte Carlo tree search (MCTS) is a general technique for approximating an expectimax calculation in stochastic games and deterministic games with uncertainty. Its use dates back several decades, but was popularized and formalized in the last decade or so in the context of planning for computer Go (Browne et al., 2012). Analogously to minimax (Russell and Norvig, 2010), we construct a game tree, with MAX (the agent) playing one turn, and ENVIRONMENT (some distribution over percepts) playing the other turn. The branching factor of MAX nodes is of course $|\mathcal{A}|$, while the branching factor of ENVIRONMENT nodes is upper bounded by $|\mathcal{E}|$. In *contrast* to minimax, which is used for deterministic games, we must collect sufficient samples from ENVIRONMENT nodes to get a good estimator $\hat{V}$ of the expected value for this node. Needless to say, we wish to avoid expanding the tree out by naively visiting every history $æ_{t:m}$.

Analogously to α-β pruning in the context of minimax, UCT is a MCTS algorithm due to Kocsis and Szepesvári (2006) that avoids expanding the whole tree, by only investigating 'promising'-looking histories. These choices must be made under uncertainty, since the environment is stochastic; hence, we have an instance of the classic exploration-exploitation dilemma. The UCT algorithm adapts and generalizes the famous UCB1 algorithm used in the context of bandits (Auer et al., 2002), to balance exploration and exploitation in the search tree.

UCB stands for 'upper confidence-bound', and is a formal version of the principle of optimism under uncertainty. The general idea is to add an 'exploration-bonus' term to the action selection objective which prefers actions that haven't been tried much. In the context of bandits, the UCB action selection is given by

$$a_{\text{UCB}} = \arg\max_{a\in\mathcal{A}} \left(\hat{R}(a) + C\sqrt{\frac{\log T}{N(a)}} \right), \tag{2.19}$$

where $\hat{R}(a)$ is the current estimator of the mean reward that results taking action a, T is the lifetime of the agent, $N(a)$ is the number of times that a has been taken, and $C > 0$ is a tunable parameter. This exploration bonus allows us to make a good trade-off between exploration and exploitation. Consider the exploration bonus term in Equation (2.19) above: by the central limit theorem, we can use the fact that the variance in our estimate of the mean will be approximately bounded by $\frac{1}{\sqrt{N}}$. The $\log T$ term in the numerator ensures that, asymptotically, we continue to visit every state-action pair infinitely often; this is necessary to establish regret bounds (Auer et al., 2009). Thus, Equation (2.19) captures the concept of 'exploration under uncertainty' in a principled way; UCT adapts this to the Monte Carlo tree search planning setting, in Markov decision processes (MDPs).

While UCT is sufficient for planning in unknown MDPs, we need to generalize to *histo-*

ries for planning in general environments. Veness et al. (2011) present this generalization, ρUCT, in their famous MC-AIXI-CTW implementation paper, based on earlier work in Monte Carlo planning on partially-observable MDPs (Silver and Veness, 2010). Using this algorithm, we don't need to know the state transitions as is required for value iteration (Equation (2.18)); we instead only need some black-box environment model ρ. The ρUCT action-selection within each decision node of the tree search is given by

$$a_{\text{UCT}} = \arg\max_{a \in \mathcal{A}} \left(\frac{1}{m\,(\beta - \alpha)} \hat{V}\left(æ_{<t} a\right) + C \sqrt{\frac{\log T\left(æ_{<t}\right)}{T\left(æ_{<t} a\right)}} \right), \tag{2.20}$$

where $\beta - \alpha$ is the reward range, and m is the planning horizon; together they are used to normalize the mean value estimate $\hat{V}$ for the history under consideration, $æ_{<t} a$. As in Equation (2.19), C is a positive parameter which controls how much we weight the exploration bonus. The exploration bonus itself is of a similar form, although note that we use $\log T\left(æ_{<t}\right)$ in the numerator, i.e. the logarithm of the number of times we've visited the current history node.

Notice that, in contrast to so-called 'model-free' methods such as Q-learning, our GRL agents can't memorize or cache the value function in general; this is because we can only compute the value of a *history* and not of *states*, because of the weakness of our modelling assumptions. Clearly we can never visit any history $æ_{1:t}$ more than once, so memorization is useless. For this reason, in general our agent has to *re-compute the value at each time step* t, so as to plan its next action a_t. Hence, all of our model-based (Bayesian) agents must plan at each time step by forward simulation with ρUCT Monte Carlo tree search. As we will see in Section 3.7, this is the major computational bottleneck for our GRL agents. Moreover, planning with MCTS requires us to have finite (and, ideally, small) action and percept spaces. In Section 3.5, we discuss our implementation of ρUCT, along with some subtle emergent issues.

In Algorithm 2.4, we present a (slightly expanded, for clarity) version of the ρUCT algorithm due to Veness et al. (2011).

Algorithm 2.4 ρUCT (Veness et al., 2011).

Inputs: History h; Search horizon m; Samples budget κ; Model ρ

INITIALIZE (Ψ)
$n_{\text{samples}} \leftarrow 0$
repeat
  $\rho' \leftarrow \rho$.COPY()
  SAMPLE (Ψ, h, m)
  $\rho \leftarrow \rho'$
until $n_{\text{samples}} = \kappa$
return $\arg\max_{a \in \mathcal{A}} \hat{V}_\Psi(a)$

function SAMPLE(Ψ, h, m)
  if $m = 0$ **then**
    return 0
  else if $\Psi(h)$ is a chance node **then**
    ρ.PERFORM(a)
    $e = (o, r) \leftarrow \rho$.GENERATEPERCEPT()
    ρ.UPDATE(a, e)
    if $T(he) = 0$ **then**
      Create chance node $\Psi(he)$
    end if
    reward $\leftarrow e$.REWARD + SAMPLE $(\Psi, he, m-1)$
  else if $T(h) = 0$ **then**
    reward $\leftarrow$ ROLLOUT(h, m)
  else
    $a \leftarrow$ SELECTACTION (Ψ, h)
  end if
  $\hat{V}(h) \leftarrow \frac{1}{T(h)+1}\Big(\text{reward} + T(h)\hat{V}(h)\Big)$
  $T(h) \leftarrow T(h) + 1$
end function

function SELECTACTION(Ψ, h)
  $\mathcal{U} = \{a \in \mathcal{A} \ : \ T(ha) = 0\}$
  if $\mathcal{U} \neq \emptyset$ **then**
    Pick $a \in \mathcal{U}$ uniformly at random
    Create node $\Psi(ha)$
    return a
  else
    return $\arg\max_{a \in \mathcal{A}} \left\{ \frac{1}{m(\beta-\alpha)} \hat{V}(ha) + C\sqrt{\frac{\log(T(h))}{T(ha)}} \right\}$
  end if
end function

function ROLLOUT(h, m)
  reward $\leftarrow 0$
  for $i = 1$ to m **do**
    $a \sim \pi_{\text{rollout}}(h)$
    $e = (o, r) \sim \rho(e|ha)$
    reward $\leftarrow$ reward $+r$
    $h \leftarrow hae$
  end for
  return reward
end function

2.5 Remarks

We now conclude with a short summary, and some remarks.

In this chapter, we presented the problem of general reinforcement learning, in which the goal is to construct an agent that is able to learn an optimal policy in a broad class of (partially observable and non-ergodic) environments. We have presented the current state-of-the-art GRL agents and algorithms, namely AIξ, Thompson sampling, MDL, Square-, Shannon-, and Kullback-Leibler-KSA, and BayesExp, under a unified notation, and we have discussed the ideas and algorithms that allow these agents to learn and plan. These agents, and the analysis and formalism around them, represent our best theoretical understanding of rationality and intelligence in this general setting. In the subsequent two chapters, we present our software implementation of these agents, and some experiments we run on them.

Chapter 3

Implementation[1]

There are no surprising facts, only models that are surprised by facts; if a model is surprised by the facts, it is no credit to that model.

We now present the design and implementation of the open-source software demo, AIXIjs.[2] Our implementation can be decomposed into roughly five major components, or modules, which we discuss in this chapter:

- **Agents**. We implement the agents specified in Chapter 2. Some of them differ by one line of code; for example, the KSA agents can be built from AIξ by simply replacing its utility function. We document the agent implementation in Section 3.2.
- **Environments**. We design and implement environments to showcase the various agents, including a partially observable Gridworld, and a 'chain' MDP environment; both are documented in Section 3.3.[3]
- **Models**. For our Bayesian agents, we design and implement two model classes with which they can learn the Gridworld environment, $\mathcal{M}_{\text{loc}}$ and $\mathcal{M}_{\text{Dirichlet}}$. These are presented in Section 3.4.
- **Planners**. We implement value iteration and ρUCT Monte Carlo tree search, which were presented in Section 2.4. We make some implementation-specific remarks in Section 3.5.
- **Visualization and user interface**. We design and implement a user interface that allows the user to choose demos, read background and demo-specific information, tune parameters, and run experiments. We also present a graphic visualization for showing the agent-environment interaction, and for plotting the agent's performance; this is presented in Section 3.6.

First, we briefly discuss the software tools we used to implement the project.

[1] AIXIjs was implemented in collaboration with Sean Lamont, a second-year undergraduate student at the ANU. Sean wrote many of the visualizations under my supervision; the rest of the implementation is my own work. More detailed contribution information (including commit history) can be found at https://github.com/aslanides/aixijs/graphs/contributors.

[2] The demo can be run at http://aslanides.github.io/aixijs; all supporting source code can be found at http://github.com/aslanides/aixijs. We encourage the reader to interact with the demo, though again, we strongly recommend using Google Chrome, as the software was not tested on other browsers, for reasons detailed in Section 3.1.

[3] We also implement multi-armed bandits, generic finite-state MDPs, and iterated prisoner's dilemma, but we don't document them here as they don't play a prominent role in the demos or experiments.

3.1 JavaScript web demo

We implement AIXIjs as a static web site. That is, apart from web hosting for the `.html` and `.js` source code and other site assets, there is no back-end server required to run the software; the demo runs natively, and locally, in the user's web browser. All of the agent-environment simulations are implemented in modern JavaScript (ECMAScript 2015 specification), with minimal use of external libraries. This allows us to effectively build a lightweight[4] and portable software suite, which a modern web browser can run without the need for specialized dependencies such as compilers or scientific libraries.

JavaScript (JS) is a high-level, dynamic, and weakly-typed language typically used to create dynamic content on websites. Google's V8 JS engine, implemented in their Chrome web browser, provides a fast JS runtime; in many benchmarks, it is significantly faster than Python 3.[5] As we discussed in the Introduction, this allows for computationally intensive and visually impressive software. JavaScript, however, does have several shortcomings. The ones that are relevant to us are:

- JavaScript is a notoriously[6] weakly-typed language, which comes with all the programming pitfalls and runtime errors one would expect. For example, functions will silently accept arguments that are null, and attempt to perform computations on them. In this way, subtle bugs can cause catastrophic runtime errors that can propagate quite far without being caught. We mitigate this to some extent by writing tests using the QUnit testing framework, and by frequently using the built-in debugger in Google Chrome.

- JavaScript implementations differ between browsers. For example, some features of the ECMAScript 2015 specification (for example, anonymous functions) were not yet implemented by the latest version of the Safari web browser as of September 2016. Worse still, behaviors can differ subtly and in undocumented ways between browser implementations. We (unfortunately) are forced to work around this by only supporting recent versions of Google Chrome,[7] and discouraging usage on other web browsers.

We use standard web frameworks and libraries: jQuery and Bootstrap for presentation; d3js for graphics and visualizations; marked for MarkDown parsing, and MathJax for rendering mathematics in the browser. Our implementation totals roughly 6000 lines of JavaScript.

We make use of a modular design, and use class inheritance frequently, so as to minimize code duplication and to leverage the conceptual connections between objects. In the sections that follow, we occasionally use simple UML diagrams to document these classes. Note that in these diagrams we use type annotations, for expository purposes.

[4] Including all source code, external libraries, fonts, text, and image assets, the software totals less than 2 megabytes in size, uncompressed.

[5] For inter-language comparisons on common benchmarks, see, for example, http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=node&lang2=python3.

[6] The author highly recommends a brilliant four-minute video by Gary Bernhardt about the nonsense that comes from JavaScript's (lack of) type system: https://www.destroyallsoftware.com/talks/wat.

[7] The software was last tested on Google Chrome version 54.0.

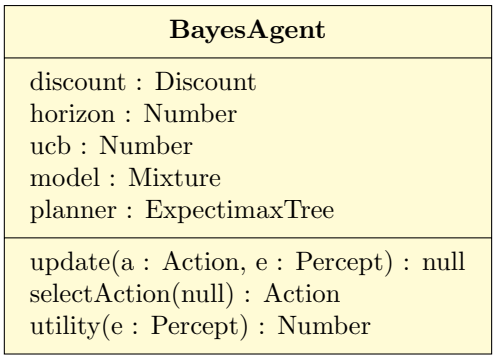

Figure 3.1: BayesAgent UML. `discount` is the agent's discount function, γ_k^t. `horizon` is the agent's MCTS planning horizon, m. `ucb` is the MCTS UCB exploration parameter C.

3.2 Agents

All agents inherit from the base Agent class. Every agent's constructor takes an Options object as input, which allows us to pass in default and user-specified options. See Figure 3.2 for the full agent class inheritance tree. The Agent base class specifies the methods

- Update (a, e). Update the agent's model of the environment, given that it just performed action $a \in \mathcal{A}$ and received percept $e \in \mathcal{E}$ from the environment.
- SelectAction(). Compute, and sample from, the agent's (in general, stochastic) policy $\pi(a|æ_{<t})$, returning an action $a \in \mathcal{A}$.
- Utility (e). This is the agent's utility function, as defined in Definition 14. For reward-based reinforcement learners it simply extracts the reward component from `percept`.

Every agent is further equipped with a **Discount** function, as defined in section 2.2.2.

Every Bayesian agent (i.e. of class BayesAgent or one of its descendants) is further composed of a Model and a Planner, which are both central to its operation. When we call Update (a, e) on BayesAgent, it saves its model's state[8], calls the model's Update (a, e) method, and then computes and stores the information gain (defined in Equation (2.13)) between the old and new model states. When we call SelectAction, the agent passes its model to the Planner, and waits for it to compute a best action. If the information gain from the previous action was non-zero, the planner's internal state is reset; otherwise, we prune the search tree but keep the partial result; see Section 3.5 for more discussion regarding the planner.

The other agents inherit from BayesAgent, and differ from it in straightforward and transparent ways specified by their respective definitions (Definition 16, Definition 17, Definition 15, Algorithm 2.1, Algorithm 2.2, and Algorithm 2.3). We won't reproduce their source code here; the interested reader can find the code in the `src/agents/` directory in the GitHub repository.

[8] As we shall see in Section 3.4, models must behave like environments. Definition 2 implies that they must therefore in general maintain some internal state s that is changed by actions $a \in \mathcal{A}$.

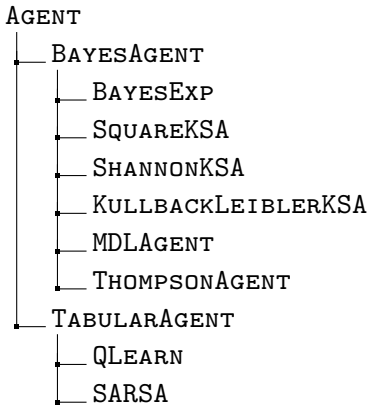

Figure 3.2: AGENT class inheritance tree. Note that the BayesAgent is simply AIξ.

3.2.1 Approximations

We now enumerate and justify our use of several approximations and simplifications in our agent implementations. The first two approximations are motivated by computational considerations. In both cases, we argue that our use of these simplifications leaves the agent's policy invariant. The third simplification is in fact forced upon us; it is inconvenient and potentially highly consequential to the performance of Shannon-KSA.

- **Information gain**. Recall from Chapter 2 that the information gain for a Bayesian agent given a history $æ_{<t}$ is

$$\mathrm{IG}\left(e|æ_{<t}a_t\right) \doteq \mathrm{Ent}\left(w\left(\cdot|æ_{<t}\right)\right) - \mathrm{Ent}\left(w\left(\cdot|æ_{1:t}\right)\right),$$

  and recall that this is the utility function of the Kullback-Leibler knowledge-seeking agent (KL-KSA). Now, when computing the KL-KSA policy at time t – that is, in calls to SELECTACTION – we compute the value $V_\xi^{\pi,\mathrm{IG}}$ of various potential histories $æ_{<t}a_te_ta_{t+1}e_{t+1}\ldots a_{t+m}e_{t+m}$, and select the action that maximizes this value. Note that our action-selection doesn't depend on the *absolute value* of different histories, but only on their *relative* value. Note also that the quantity $\mathrm{Ent}\left(w\left(\cdot|æ_{<t}\right)\right)$ does not depend on future actions or percepts, as it is determined by events in the agent's past. Hence it is a constant that we can ignore when comparing the relative value of future actions. From the definition of entropy (Equation (2.4)), we see that computing the entropy of the posterior $w\left(\nu|\cdot\right)$ requires $\mathcal{O}\left(|\mathcal{M}|\right)$ operations. For this reason, in our implementation of the KL-KSA, we achieve a $2\times$ speedup by replacing u_{KL} with the surrogate utility function

$$u'_{\mathrm{KL}}\left(æ_{1:t}\right) = -\mathrm{Ent}\left(w\left(\cdot|æ_{1:t}\right)\right).$$

- **Effective horizon**. Recall from Algorithm 2.3 and Algorithm 2.1 that the Thompson sampling and BayesExp agents both explore for an effective horizon $H_\gamma^t\left(\varepsilon\right)$ (Equation (2.6)); this requirement is, in fact, essential to the proofs of their asymptotic optimality. However, computing the effective horizon exactly for general discount functions is not possible in general, although approximate effective horizons have been derived for some common choices of γ (Lattimore, 2013; Table 2.1). Moreover, in practice, due to the computational demands of planning with MCTS (Al-

gorithm 2.4), we are forced to plan only with a relatively short horizon m; for most discount functions γ and realistic ε,[9] the true effective horizon $H_\gamma^t(\varepsilon)$ is significantly greater than m. For this reason, and for simplicity and ease of computation, we use the MCTS planning horizon m as a surrogate for H_γ^t. Naturally, this choice affects the agent's policy, but no more so than we already have by using MCTS to plan up to some (finite, time-constrained, and pragmaticaly chosen) horizon m rather than to infinity, as the agents do in the theoretical formalism.

- **Utility bounds**. Recall from the ρUCT action selection algorithm (Equation (2.20)) that the value estimator $\hat{V}(æ_{1:t})$ is normalized by a factor of $m(\beta - \alpha)$, where m is the MCTS planning horizon, and α and β are the minimum and maximum rewards that the agent can receive in any given percept. In the case of reward-based reinforcement learners, α and β are essentially metadata provided to the agent, along with the size of the action space $|\mathcal{A}|$, at the beginning of the agent-environment interaction. For utility-based agents, however, the rewards are generated *internally*, and so the agent must calculate for itself what range of utilities it expects to see, so as to correctly normalize its value function for the purposes of planning.

  Thankfully, for the Square and Kullback-Leibler KSAs, this is relatively easy to do. Since $u_{\text{Square}}(e) = -\xi(e)$, we can immediately bound its utilities in the range $[-1, 0]$. In general this won't be a tight bound, since there exist environment mixtures in which every percept is in some smaller range, i.e. $\xi(\cdot) \in [a, b]$ with $a > -1$ and $b < 0$,[10] but in practice, and in particular for our model classes, it is effectively a tight bound.

  In the case of the Kullback-Leibler KSA, recall that $u_{\text{KL}}(e) = \text{Ent}(w(\cdot)) - \text{Ent}(w(\cdot|e))$. If we assume that we are given the maximum-entropy (i.e. uniform) prior $w(\cdot|\epsilon)$, then clearly $u_{\text{KL}}(e) \leq \text{Ent}(w(\cdot|\epsilon)) \;\; \forall e \in \mathcal{E}$, since entropy is always non-negative. Hence we have $0 \leq u_{\text{KL}} \leq \text{Ent}(w(\cdot|\epsilon))$, i.e. the KL-KSA's rewards are bounded from above by the entropy of its prior (assuming a uniform prior), and from below by zero.

  Finally, we come to the problematic case: from Figure 2.4, we know that $u_{\text{Shannon}}(e) = -\log \xi(e)$ is unbounded from above as $\xi \to 0$. This means that unless the agent can *a priori* place lower bounds on the probability that its model ξ will assign to an arbitrary percept $e \in \mathcal{E}$, it cannot upper bound its utility function and therefore cannot normalize its value function correctly. This is problematic for us, especially as our environments and models are constructed in such a way as to allow arbitrarily small probabilities, as we will see in Section 3.3 and Section 3.4.

  Unfortunately, it seems we're stuck here. We're forced to make an ugly, arbitrary choice to upper bound the Shannon agent's utility function with, so as to normalize its value function. If we choose the upper bound β too high, then the $\hat{V}$ term in Equation (2.20) will be artificially, but consistently small; this is equivalent to inflating the exploration bonus constant C by roughly a constant multiplicative factor (which is itself upper bounded by some function of β). If β is chosen too small, however, we can run into much bigger problems, since now $\hat{V}$ can be over-inflated by an unboundedly large multiplicative factor. If Shannon KSA sees a very

[9] Recall that in the case of BayesExp, ε is compared to the value of the knowledge-seeking policy, $V_\xi^{*,\text{IG}}$.

[10] For example, a coinflip environment in which the agent is trying to falsify one of two hypotheses: whether a coin is fair ($\nu(\cdot) = 0.5$) or bent ($\nu(\cdot) \neq 0.5$).

improbable percept, its value estimates will blow up, which will cause suboptimal plan selection, since the $\hat{V}$ will overwhelm the exploration bonus term in Equation (2.20). We are forced to choose a β, so we use a very large upper bound, $\beta = 10^3$ in an attempt to balance this trade-off, but bias it in favor of overestimating β. For us to exceed -10^3 in $\log_2$ probability requires us to assign a probability of $\xi(e) \leq 2^{-10^3} = 10^{-301}$, which is approaching the limits of numerical precision in JavaScript. With this setting of β we are unlikely to blow up our value estimate, although we will be severely inflating the UCB constant. As we will see in Chapter 4, this is quite possibly the cause of some suboptimal behavior in the Shannon KSA.

3.3 Environments

Recall that AIXI and its variants are theoretical models of unbounded rationality, not practical algorithms. Bayesian learning and planning by forward simulation with Monte Carlo tree search are both very computationally demanding, so we restrict ourselves to demonstrating their properties on small-scale POMDPs and MDPs.

Analogously to the case of agents, all environments inherit from the base Environment class. Every environment's constructor takes an Options object as input, which allows us to pass in default and user-specified options. The Environment base class specifies the methods

- ConditionalDistribution(e). Returns the probability $\nu(e_t|ae_{<t}a_t)$ that the environment assigns to percept e given its current state resulting from the history $æ_{<t}a_t$.
- GeneratePercept(). Sample from $\nu(e)$ and returns a percept $e \in \mathcal{E}$.
- Perform(a). Take in action $a \in \mathcal{A}$ and mutate the environment's (in general, hidden) state according to its dynamics.
- Save() and Load(). These functions save and load the environment's internal state. This is a convenience for our Bayesian agents; it allows them to reset the environments $\nu \in \mathcal{M}$ that make up their mixture model, after running counterfactual simulations in a planner.

We now introduce the Gridworld and Chain environments. The interested reader can find the source code to these, and other, environments in the `src/environments/`directory in the GitHub repository.

3.3.1 Gridworld

Our gridworld consists of an $N \times N$ array of tiles. There are four types of tiles: Empty, Wall, Dispenser, and Trap, with the following properties:

- Empty tiles allow the agent to pass, albiet while incurring a small movement penalty r_{Empty}.
- Wall tiles are not traversable. If the agent walks into a wall, it incurs a negative penalty $r_{\text{Wall}} < r_{\text{Empty}}$.

Environment
state : Object minReward : Number maxReward : Number numActions : Number
generatePercept() : Percept perform(a : Action) : null conditionalDistribution(e : Percept) : Number save() : null load() : null

Figure 3.3: ENVIRONMENT UML. `state` is the environment's current state, it is simply of type `Object`, since we are agnostic as to how the environment's state is represented. If JavaScript supported privated attributes, this would be private to the environment, to enforce the fact that the state is hidden in general. In contrast, `minReward` (α), `maxReward` (β), and `numActions` ($|\mathcal{A}|$) are public attributes: it is necessary that the agent know these properties so that the agent-environment interaction can take place.

- DISPENSER tiles behave like EMPTY tiles as far as movement and observations are concerned, but they dispense some large reward $r_{\text{CAKE}} \gg r_{\text{EMPTY}}$ with probability θ, and r_{EMPTY} otherwise; that is, all DISPENSERs are (scaled) Bernouilli (θ) processes.[11]

- TRAP tiles, as the name suggests, don't allow you to leave. Moreover, once stuck in a trap, the agent will receive r_{WALL} reward constantly.

The gridworld we construct is a POMDP; the environment's hidden state is the agent's grid position $s = (i, j)$ and the positions of all walls, traps, and dispensers. Observations consist of a bitstring telling the agent whether the adjacent squares in the $\{\leftarrow, \rightarrow, \uparrow, \downarrow\}$ directions are WALLs or not; the edges of the Gridworld are treated implicitly as walls. The agent can move in these four cardinal directions, or stand still (this is the so-called 'no-op', which we denote by $\bigcirc$). The only way to distinguish a DISPENSER from an EMPTY tile is to walk onto it and observe the (in general, stochastic) reward signal; for low values of θ, it may take some time for a DISPENSER to reveal itself. The only way to distinguish a TRAP from an EMPTY tile is to walk onto it and see if you get trapped or not. Hence, we can characterize the action and percept spaces as

$$\begin{aligned} \mathcal{A} &= \{\leftarrow, \rightarrow, \uparrow, \downarrow, \bigcirc\} \\ \mathcal{E} &= \mathbb{B}^4 \times \{r_{\text{WALL}}, r_{\text{EMPTY}}, r_{\text{CAKE}}\}. \end{aligned}$$

Movement and observations are all deterministic; the only stochasticity in this environment arises from the reward signal from the dispenser(s).

For the purposes of our demos and experiments, we generate random gridworlds by independently and randomly assigning each tile to one of the four classes, with a strong bias towards being EMPTY, and a slighter weaker bias towards being a WALL. The agent's

[11] The AIXIJS agent mascot is Roger the Robot  . Roger likes CAKE, and will do anything it takes to get near a CAKE DISPENSER.

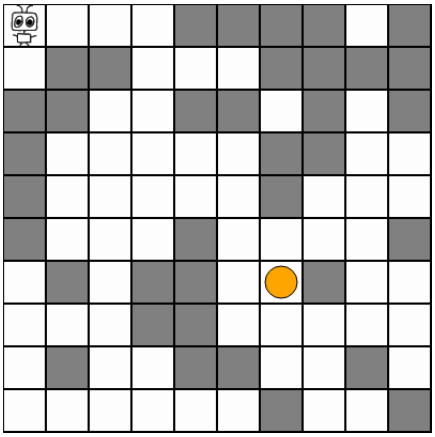

Figure 3.4: Visualization of a 10×10 Gridworld with one DISPENSER. The agent starts in the top left corner. WALL tiles are in dark grey. EMPTY tiles are in white. The DISPENSER tile is represented by an orange disc on a white background.

starting position is always the top left corner, at tile $(0, 0)$. We ensure that the gridworld is solvable by ensuring there is at least one dispenser, and by running a breadth-first-search to check whether there is a viable path from the agent's starting position to the dispenser with the highest pay-out frequency, θ.

This gridworld environment is sufficiently rich and interesting to demonstrate most of what we seek to show: the agents have to reason under uncertainty to navigate the maze and find the (best) dispenser, while avoiding traps. We report on numerous experiments using this environment in Chapter 4.

3.3.2 Chain environment

We present a deterministic version of the chain environment of Strens (2000). The chain environment is a deterministic finite-state Markov decision process. The action space is $\mathcal{A} = \{\rightarrow, \dashrightarrow\}$, and the state space is $|S| = N + 1$, for some integer $N \geq 1$. The reward space is $\{r_0, r_i, r_b\}$ with $r_0 < r_i \ll r_b$; example values are $(r_0, r_i, r_b) = (0, 5, 100)$, with $N = 6$. From Figure 3.5, we can see that at all times, the agent is tempted to reap immediate reward of r_i by taking the $\rightarrow$ action, which puts it in the `initial` state, losing whatever progress it was making towards getting to s_N, from which state it can take $\dashrightarrow$, which isn't immediately as rewarding as $\rightarrow$, but eventually leads to a very large payoff $r_b \gg r_i$. For $N < \frac{r_b}{r_i}$, the optimal policy is to always take $\dashrightarrow$ so as to perform the circuit $s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_N \rightarrow s_1 \rightarrow \ldots$ and accumulate an average reward of $\frac{r_b}{N}$. Otherwise, the optimal policy is to always take $\rightarrow$ and remain in the `initial` state. We denote these two policies as $\pi_{\dashrightarrow}$ and $\pi_{\rightarrow}$.

The (deterministic) state transition matrix is given by

$$\begin{aligned} P\left(s'|s, \rightarrow\right) &= \mathbb{I}\left[s' = 0\right] \\ P\left(s'|s, \dashrightarrow\right) &= \mathbb{I}\left[s' = (s+1) \mod (N+1)\right], \end{aligned}$$

and the rewards are given by

$$R\left(s, a\right) = r_i \mathbb{I}\left[a = \rightarrow\right] + r_b \mathbb{I}\left[a = \dashrightarrow\right] \mathbb{I}\left[s = N + 1\right].$$

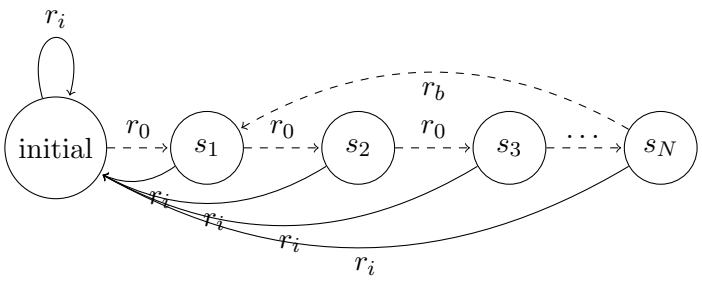

Figure 3.5: Chain environment. There are two actions: $\mathcal{A} = \{\rightarrow, \dashrightarrow\}$, the environment is fully observable: $\mathcal{O} = \mathcal{S}$, and $\mathcal{R} = \{r_0, r_i, r_b\}$ with $r_b \gg r_i > r_0$. For $N < \frac{r_b}{r_i}$, the optimal policy is to continually take action $\dashrightarrow$, and periodically receive a large reward r_b.

We construct this environment with $N < \frac{r_b}{r_i}$, so as to present a test of an agent's far-sightedness. To stay on the optimal policy $\pi_{\dashrightarrow}$, the agent must at all times resist the temptation to take the greedy action $\rightarrow$ which results in the instant gratification r_i, as this causes it to lose its progress towards the 'goal' state s_N. This simple environment models a classic situation from economics and decision theory in which humans have been known to be time-inconsistent – that is, informally, an agent acts impulsively on desires that don't agree with its long-term preferences (Hoch and Loewenstein, 1991). We report on experiments using this environment in Chapter 4.

3.4 Models

As we have seen in Chapter 2, the GRL agents we are concerned with are model-based and Bayesian. In this section we describe the generic BayesMixture model, which provides a wrapper around any model class $\mathcal{M}$, represented as an array of Environments, and allows us to compute the Bayes mixture of Equation (2.10). We then describe a model class for Gridworlds that we plug in to this BayesMixture, and a separate Dirichlet model.

The BayesMixture model provides us with a mechanism with which to use any array of hypotheses $(\nu_1, \nu_2, \ldots, \nu_{|\mathcal{M}|})$ and a prior $(w_1, \ldots, w_{|\mathcal{M}|}) \in [0,1]^{|\mathcal{M}|}$ as a Bayesian environment model. Note that all environment models must implement the environment interface: namely, they must have Perform, GeneratePercept, and ConditionalDistribution methods. In addition, Bayesian models must have an Update method, to update them with observations (either simulated or real), and Save and Load methods to restore their state after planning simulations. We document these methods in Algorithm 3.1:

- GeneratePercept: To generate percepts from the mixture model ξ, we sample an environment ρ from the posterior $w(\cdot)$, then generate a percept from ρ; in the context of probabilistic graphical models, this is known as *ancestral sampling (Bishop, 2006)*.
- Perform(a): We simply perform action a on each member ν of $\mathcal{M}$.
- ConditionalDistribution(e): We return $\xi(e_t|æ_{<t}a_t) = \sum_{\nu\in\mathcal{M}} w_\nu \nu(e_t|æ_{<t}a_t)$, where the conditioning on the history $æ_{<t}a_t$ is implicitly taken care of by conditioning on the environment's internal state s.

- UPDATE (a, e): We update our posterior given percept e using Bayes rule: $w(\nu|e) = w(\nu)\frac{\nu(e)}{\xi(e)}$, for each $\nu \in \mathcal{M}$.

BayesMixture
modelClass : [ ]Environment weights : [ ]Number
generatePercept() : Percept perform(a : Action) : null update(a : Action, e : Percept) : null conditionalDistribution(e : Percept) : Number save() : null load() : null

Figure 3.6: BAYESMIXTURE UML diagram. Internally, the BayesMixture contains a `modelClass` $\mathcal{M}$, which is an array of environments, and `weights` w, which are a normalized array of floating-point numbers.

Our objective is to construct a Gridworld model that is sufficiently informed, or constrained, so as to make it possible for the agent to learn to solve the environments we give it within a hundred or so cycles of agent-environment interaction, but that is also sufficiently rich and general so that it is interesting to watch the agent learn. For this reason, we eschew very general and flexible models such as the famous context-tree weighting data compressor used by Veness et al. (2011), since they will take too long to learn the environments for a practical demo. Instead, we construct two models, with varying degrees of domain knowledge built-in:

1. A mixture model parametrized by dispenser location, which we call $\mathcal{M}_{\text{loc}}$.

2. A factorized Dirichlet model, in which each tile is represented as an independent Dirichlet distribution. We call this model $\mathcal{M}_{\text{Dirichlet}}$.

The interested reader can find the source code for these and other models in the `src/models/` directory in the GitHub repository.

3.4.1 Mixture model

Before we present the mixture model $\mathcal{M}_{\text{loc}}$, we consider the problem of constructing a model class $\mathcal{M}$. That is, we want a simple and principled method with which to construct a finite but non-trivial set of hypotheses about the nature of the true Gridworld environment μ. We do this by chosing some discrete parametrization $D = \{d_1, \ldots, d_{|\mathcal{M}|}\}$ such that a model class $\mathcal{M}$ is constructed by sweeping through values of $d \in D$:

$$\xi(e) = \sum_{d \in D} w_d \nu_d(e).$$

One can think of D as describing a set of parameters about which the agent is uncertain; all other parameters are held constant, and the agent is fully informed of their value. We now consider and implement three different choices for the parametrization D, and enumerate some of the pros and cons for each.

1. **Dispenser location**. We construct $\mathcal{M}$ by sweeping through all legal (that is, not already occupied by a WALL) dispenser locations, given a fixed maze layout, and fixed dispenser frequencies. In other words, we hold constant the layout of all EMPTY, WALL, and TRAP tiles, and vary the location of the dispensers. The agent's beliefs $w(\nu_{ij})$ are now interpreted as the agent's credence that the dispenser is at location (i, j) in the Gridworld.

   The benefit of this choice of D is that it is straightforward and intuitive: the agent knows the layout of the gridworld and knows its dynamics, but is uncertain about the location of the dispensers, and must explore the world to figure out where they are. This also has the benefit of lending itself easily to visualization of the agent's beliefs: see Figure 3.7. Moreover, since dispensers are stochastic, it may take several observations to falsify any given hypothesis ν; the model class allows for 'soft' falsification. Another advantage of this model class is that it incentivizes the agent to explore, since the agent will initially assign non-zero probability mass to there being a dispenser at every empty tile.

   A significant downside of this model class is that we get a combinatorial explosion if we want to model environments with more than one dispenser. That is, given a maze layout with L legal positions, a model class with M dispensers will have $|\mathcal{M}| = \binom{L}{M}$ elements. Another downside is that the agent knows the maze layout ahead of time, which detracts from some of the interest in having a maze on the Gridworld. We present the procedure for generating this model class in Algorithm 3.2.

2. **Agent starting location**. We use a similar procedure as described in Algorithm 3.2 to construct the model class, except this time by parametrizing by the agent's starting location. In this case, D is given by the set of legal starting positions. This corresponds nicely to the (noise-free) localization problem given a known environment which shows up often in the field of robotics (Thrun et al., 1999). Since observations are deterministic, it is possible to discard many hypotheses at once, and so the agent is able to narrow down its true location very quickly. The Gridworlds we simulate aren't large or repetitive enough to have sufficiently ambiguous percepts for the agent to be uncertain about its location for more than a few cycles. Thus, after a short time, the agent is certain of its position, and is longer incentivized to explore; if the dispenser isn't within its planning horizon by this stage, it will not be able to find it, and will perform very badly. We discuss the quirks and limitations of planning more in Section 3.5.

3. **Maze configuration**. Perhaps the most general, and hence most interesting, model parametrization is by maze configuration: the agent is initially uncertain about the identity of every tile in the Gridworld. Thus, the agent is thrown into a truly unknown gridworld, and must learn the environment layout from scratch. In a sense this is the most natural parametrization, since each gridworld layout gives rise to a truly different environment. Another benefit is that this is a very rich environment class; unfortunately, this is also the downside, as it is prohibitive to naively enumerate every possible maze configuration. Given just two tile classes, EMPTY and WALL, there are 2^{N^2} possible $N \times N$ mazes. Using this naive enumeration, we would run out of memory even on a modest 6×6 Gridworld, as $|\mathcal{M}| = 2^{36} \approx 7 \times 10^{10}$, and most laptop computers have only of the order of eight gigabytes, or 6.4×10^{10} bits of memory. We can alleviate this somewhat by simply downsampling, say by

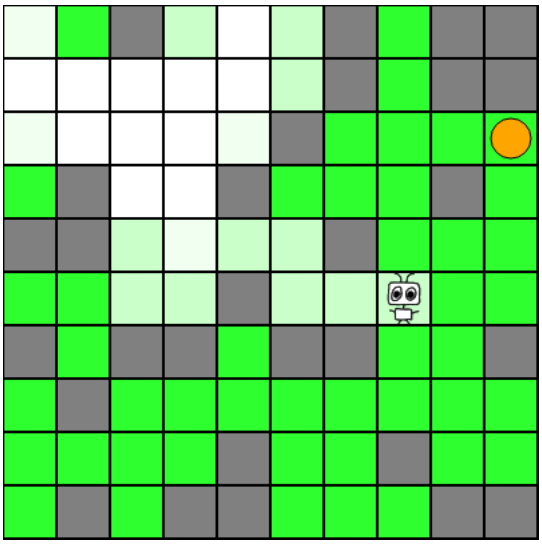

Figure 3.7: Gridworld visualization with the agent's posterior w over $\mathcal{M}_{\text{loc}}$ superimposed. Green tiles represent probability mass of the posterior w_ν, with higher values correspond to darker green color. The true dispenser's location is represented by the orange disc. As the Bayesian agent walks around the gridworld, it will move probability mass in its posterior from tiles that it has visited to ones that it hasn't.

discarding at random most of the elements of this gargantuan model class. We find in practice that this runs into similar problems to the second parametrization, and produces demos that are slow (due to the size of the model class; see Section 3.7 for a discussion of time complexity) *and* uninteresting, because the agent is able to falsify so many hypotheses at once.

Algorithm 3.2 Constructing the dispenser-parametrized model class.

Inputs: Environment class E and parameters Φ; Gridworld dimensions N.
Outputs: Model class $\mathcal{M}$ and uniform prior w

$w \leftarrow \textsc{Zeros}(N^2)$
$\mathcal{M} \leftarrow \{\}$
$k \leftarrow 1$
for $i = 1$ to N **do**
    for $j = 1$ to N **do**
        $\nu \leftarrow \textsc{Initialize}\,(E, \Phi)$
        if $\nu.\textsc{Grid}[i][j] = \textsc{Wall}$ **then continue**
        end if
        $\nu.\textsc{Grid}[i][j] \leftarrow \textsc{Dispenser}(\theta)$
        $\mathcal{M}.\textsc{Push}(\nu)$
        $w[k] \leftarrow 1$
        $k \leftarrow k + 1$
    end for
end for
$\textsc{Normalize}(w)$

We find empirically that Item 1 above makes for the most interesting demos, and so our canonical model class used in many of the Gridworld demos is the dispenser-parametrized

model class $\mathcal{M}_{\text{loc}}$. We conclude with some remarks about the properties of learning with $\mathcal{M}_{\text{loc}}$ that will be consequential to our experiments in Chapter 4:

- As mentioned above, using $\mathcal{M}_{\text{loc}}$ gives the agent complete knowledge *a priori* of the maze layout. The agent's task becomes to search the maze for the dispenser. This task incorporates both subjective uncertainty (we typically initialize the agent with a uniform prior over dispenser location) and noise (for $\theta \in (0, 1)$, the dispensers are stochastic processes).

- Using $\mathcal{M}_{\text{loc}}$, the agent knows that there is only one dispenser. This means that, regardless of θ, once it does find the dispenser – by experiencing the relevant reward percept – it is able to immediately falsify every other hypothesis regarding the location of the dispenser. In other words, its posterior $w(\cdot|æ_{<t})$ will *collapse* to the indicator function $\mathbb{I}[\nu = \mu]$, and the agent will have learned everything there is to know about the environment.

Now, motivated by the limitations of $\mathcal{M}_{\text{loc}}$ that we discussed in Item 1, and inspired by the notion of a model that is uncertain about the maze layout (Item 3), we set out to design and implement an alternative Bayesian Gridworld model, $\mathcal{M}_{\text{Dirichlet}}$.

3.4.2 Factorized Dirichlet model

We now describe an alternative Gridworld model, which has several desirable properties. In contrast to the naive mixture model, it allows us to efficiently represent uncertainty over the maze layout, as well as the dispenser locations and payout frequencies θ. This means that $\mathcal{M}_{\text{Dirichlet}}$ is, in comparison to $\mathcal{M}_{\text{loc}}$, a relatively unconstrained, and thus harder to learn, model.

The basic idea is to model each tile in the Gridworld independently with a categorical distribution over the four possible types of tile: Empty, Wall, Dispenser, and Trap. For an $N \times N$ Gridworld, label each of the tiles s_{ij} where $i, j \in \{1, \dots, N\}$. The joint distribution over all Gridworlds $s_{11}, \dots, s_{NN}$ is then given by the product

$$p(s_{11}, \dots, s_{NN}) = \prod_{(i,j)=(1,1)}^{(N,N)} p(s_{ij}), \tag{3.1}$$

where $s_{ij} \in \{\text{Empty}, \text{Dispenser}, \text{Wall}, \text{Trap}\}$. Note that here, by Dispenser, we mean a dispenser with $\theta = 1$. This allows us to model dispensers with $\theta \in (0, 1)$ as a stochastic mixture over an Empty tile and a Dispenser with $\theta = 1$. For example, a dispenser with $\theta = 0.5$ would be represented[12] by the distribution $\boldsymbol{p} = (0.5, 0.5, 0, 0)$; a tile known to be a Wall would be represented by the distribution $\boldsymbol{p} = (0, 0, 1, 0)$. We initialize our model with the uniform prior; that is, for each tile s_{ij} we have $p(s_{ij}) = 0.25$ $\forall s_{ij} \in \{\text{Empty}, \text{Dispenser}, \text{Wall}, \text{Trap}\}$.

Now, recall from section 2.1.2 that the Dirichlet distribution is conjugate to the categorical distribution. So, to represent our uncertainty about the relative probabilities of each of the classes, and to enable us to update our beliefs in a Bayesian way, we make use of a Dirichlet distribution over the four-dimensional probability simplex. That is, for each

[12] Recall that the categorical distribution is just a distribution over a set of K categories. We represent the distribution with a length-K vector $p \in [0, 1]^K$. We use the notation $\Pr(S = s) \equiv p(s) \equiv p_s$ interchangeably.

tile s, the probability vector

$$\boldsymbol{p} \doteq \begin{bmatrix} \Pr\left(s = \textsc{Empty}\right) \\ \Pr\left(s = \textsc{Dispenser}\right) \\ \Pr\left(s = \textsc{Wall}\right) \\ \Pr\left(s = \textsc{Trap}\right) \end{bmatrix}$$

is distributed according to

$$\boldsymbol{p} \sim \text{Dirichlet}\left(\boldsymbol{p}|\boldsymbol{\alpha}\right),$$

where $\boldsymbol{\alpha} = \begin{bmatrix}\alpha_{\textsc{Empty}} & \alpha_{\textsc{Dispenser}} & \alpha_{\textsc{Wall}} & \alpha_{\textsc{Trap}}\end{bmatrix}^T$ are the empirical counts of each class, and $\mathbf{1}^T\boldsymbol{p} = 1$. Updates to the posterior are trivial: just increment the corresponding count, i.e. upon seeing one instance of class N, we update with

$$\text{Dirichlet}\left(\boldsymbol{p}|\alpha_1, \ldots, \alpha_K, N\right) = \text{Dirichlet}\left(\boldsymbol{p}|\alpha_1, \ldots, \alpha_N + 1, \ldots \alpha_K\right). \tag{3.2}$$

Now, given that the agent is at some tile s_t, the conditional distribution over percepts e_t is drawn from the product over the neighbouring Dirichlet tiles:

$$\rho\left(e_t|æ_{<t}a_t\right) \sim \prod_{s' \in \text{ne}(s_t) \cup \{s_t\}} \text{Dirichlet}\left(\boldsymbol{p}|\boldsymbol{\alpha}_{s'}\right). \tag{3.3}$$

The astute reader will notice that though the joint distribution over tile *states* factorizes, *percepts* will be locally correlated, since percepts are sampled from neighboring tiles, and we have a four-connected grid topology.

For the purposes of computational efficiency we make two approximations:

1. We don't sample ρ from the Dirichlet distributions Equation (3.3), but instead simply use their mean; recall that the mean of Dirichlet $\left(\boldsymbol{p}|\boldsymbol{\alpha}\right)$ is given by

   $$\boldsymbol{\mu} = \frac{\boldsymbol{\alpha}}{\sum_{k=1}^{K} \alpha_k}.$$

   We do this because sampling correctly from the Dirichlet distribution is non-trivial, and this sampling would need to occur whenever we wish to generate a percept, either real and simulated; this is a far too large computational cost to bear for the purposes of our demo. This approximation will effectively reduce the variance in percepts generated by the model, but in mean, over many simulations, will have negligible effect.

2. When computing the entropy of the agent's beliefs for the purposes of calculating the information gain (Equation (2.13)), computing the joint entropy over all tiles becomes computationally very expensive, as neighboring tiles are correlated with respect to percepts, and so the entropy of the joint does not decompose nicely into a sum of entropies. We compute a surrogate for the entropy by associating with each tile the mean probability that it assigns to its being a dispenser; that is, for each tile s_{ij} we compute

   $$q\left(s_{ij}\right) = \boldsymbol{\mu}^{ij}_{\textsc{Dispenser}}.$$

   That is, for each tile s_{ij} we compute its mean $\boldsymbol{\mu}^{ij}$, which is a categorical distribution over $\{\textsc{Empty}, \textsc{Dispenser}, \textsc{Wall}, \textsc{Trap}\}$; we then take the $\textsc{Dispenser}$ component.

We concatenate all the $q(s_{ij})$ together into a vector $\tilde{q}$ of length N^2 and normalize. Thus, the components of $\tilde{q}$ are given by

$$\tilde{q}_{ij} \doteq \frac{q(s_{ij})}{\sum_{(i,j)=(1,1)}^{(N,N)} q(s_{ij})}.$$

Now, $\tilde{q}_{ij}$ is effectively the model's mean estimate of the probability that the $(i,j)^{\text{th}}$ tile is a dispenser; this is now directly analogous to the posterior belief $w(\nu|\dots)$ in the $\mathcal{M}_{\text{loc}}$ mixture model, since each environment ν asserts that some unique tile (i,j) is the dispenser. Now, when computing the entropy of the Dirichlet model, we simply return $\text{Ent}(\tilde{q})$. This approximation is reasonable, since percepts relating to Walls and Traps are deterministic, and so, once the agent has visited any given tile, the only uncertainty (entropy) its model has is with respect to whether a tile is a Dispenser or Empty. Moreover, for a one-dispenser environment, if the agent visits every tile infinitely often, $\tilde{q}_{ij}$ will asymptotically converge to $\mathbb{I}[(i,j) = (i_\mu, j_\mu)]$ with $\text{Ent}(\tilde{q}) = 0$, where (i_μ, j_μ) is the true dispenser location in environment μ.

We emphasize that each tile has its own empirical counts $\boldsymbol{\alpha}_{s'}$; these are learned separately, through observations. Now, in general, as soon as the agent is unsure whether an adjacent tile is a wall or not, it will become uncertain of its position; its posterior over its position will diffuse over the Gridworld as time progresses. This corresponds to the difficult problem known as *simultaneous localization and mapping* (SLAM), which shows up in robotics (Leonard and Durrant-Whyte, 1991); it is necessary to use a version of the Expectation Maximization (EM) algorithm to simultaneously solve the two inference problems. This is far too difficult a problem to solve in the demo.

Instead, we choose our prior over each of the $\boldsymbol{\alpha}$ so as to allow the agent to learn immediately whether an adjacent tile is a wall or not. We use the Haldane prior, $\alpha_k = 0 \;\forall k$. This has the nice property that it behaves like a uniform prior over the classes $\{\text{Empty}, \text{Wall}, \text{Dispenser}, \text{Trap}\}$, but in contrast to the more common Laplace prior $a_k = 1 \;\forall k$, it also has the property that it allows us to do 'hard' updates, in which we move all of the probability mass onto one class in the categorical distribution. That is, given that observations are deterministic and the maze layout doesn't change, we know that if we see a Wall tile adjacent, then our model should represent the fact that this tile is a Wall with probability one:

$$\alpha_k = \mathbb{I}[k = \text{Wall}] \implies \mu_k = \mathbb{I}[k = \text{Wall}].$$

Note that we avoid 'hard' updates with respect to whether a tile is Empty or a Dispenser by effectively using a Laplace prior over tiles that we know with certainty aren't walls; these 'Laplace' tiles are easily identifiable as the grey tiles in Figure 3.8; they are tiles that the agent has been adjacent to, but which it hasn't stepped onto yet:

$$\alpha(k|\neg\text{Wall}) = \mathbb{I}[k = \text{Dispenser}] + \mathbb{I}[k = \text{Empty}]. \tag{3.4}$$

Note that above we use the shorthand $\alpha_k \equiv \alpha(k)$ so as to more conveniently represent conditioning; this is analogous to our writing $w_\nu \equiv w(\nu)$ in the case of the mixture model. Using the Laplace prior, and updating with Bayes' rule normally, yields the famous Laplace rule for binary events. Consider some Gridworld tile s that happens to be Empty. If the agent starts with the Laplace prior given by Equation (3.4) and subsequently visits this

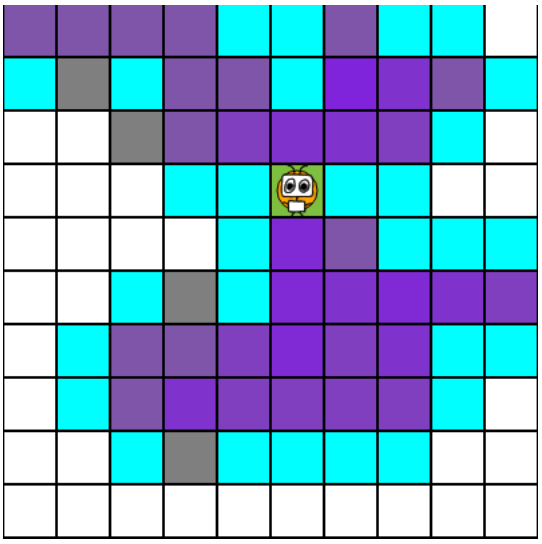

Figure 3.8: Visualization of a Gridworld, overlayed with the factorized Dirichlet model. White tiles are yet unexplored by the agent. Pale blue tiles are known to be walls. Different shades of purple/green represent different probabilities of being EMPTY or a DISPENSER.

tile n times, then the agent's posterior belief that s is in fact EMPTY is simply

$$\Pr(s = \text{EMPTY}) = \frac{n+1}{n+2}, \tag{3.5}$$

which can easily be seen by applying the Dirichlet posterior update (Equation (3.2)) n times. Thus, the agent asymptotically learns the truth as $n \to \infty$, but for any finite n the model still has some degree of uncertainty.

This Dirichlet model has numerous distinct advantages: it allows the agent to discover the grid layout as it explores, represent multiple dispensers, and learn online the Bernoulli parameter θ_d for any dispenser d, by virtue of maintaining a simple Laplace estimator of the probabilities $\Pr(d = \text{EMPTY})$ and $\Pr(d = \text{DISPENSER})$. It also makes for an interesting visualization, as we can reveal the Gridworld to the user as the agent discovers it; see Figure 3.8. These advantages essentially stem from modelling each tile independently, and come at the cost of no longer being able to represent our model explicitly as a mixture in the form of Equation (2.10). This precludes the use of $\mathcal{M}_{\text{Dirichlet}}$ in some algorithms, for example Thompson sampling, which requires mixing coefficients w_ν to sample from. It also comes at a considerable computational cost: as we will see in Section 3.7, this model is more costly to compute than the (much simpler) Bayes mixture ξ. In Chapter 4, we perform numerous experiments using this model class, and contrast it (with respect to agent performance) with the dispenser model class $\mathcal{M}_{\text{loc}}$.

3.5 Planners

We implement both the value iteration and ρUCT MCTS algorithms that were introduced in Section 2.4. The interested reader can read the source code for our implementation of these algorithms in the `src/planners/` directory in the GitHub repository. In this section, we discuss some subtle differences between our implementation and the referencee implementation by Veness et al. (2011), and we make some remarks about planning by

simulation generally, and planning in partially observable, history based environments in particular.

Recall that the objective of 'planning' here is to compute, at each time step, the estimator $\hat{V}^*_\rho$, which is a sampling approximation of the expectimax calculation in Equation (2.9). The agent's policy is then to take the action that maximizes this value. This is essentially *planning by forward simulation*. That is, we use our black-box environment model ρ to predict how the world will respond to future hypothetical actions. Informally, we run Monte Carlo simulations of numerous potential histories[13], and collect statistics on which ones lead to the best outcomes. With each sample, we simulate a playout up to some fixed horizon m.

Due to the stochasticity in general environments (and especially in the mixture model ξ), typically many samples are needed to converge to a good estimate of V^*_ρ. Note that, not only do we update the state of our model with each simulated time step, but we also update the agent's beliefs. This is an important point that we feel is perhaps not emphasized enough: a rational agent, while planning under uncertainty, should simulate changes to its beliefs and the effects such changes will have on its subsequent actions. After each sample of a forward trajectory, we reset the agent's model state and beliefs to what they were before simulating the play-out. MCTS is an *anytime* algorithm, in the sense that we can stop collecting samples early, and still have a valid (though perhaps inaccurate) estimate $\hat{V}^*_\rho$.

Notice that we compute $\hat{V}^*_\rho$ *at each time step*. Doing this naively, from scratch (i.e. resetting the search tree) seems wasteful. This prompts us to discuss the issue of caching partial results. Consider a generic scenario, in which our agent has experienced some history $æ_{<t}$, and now computes $\hat{V}^*_\rho(æ_{<t})$ using MCTS, so as to plan which action a_t to take next. Say its tree search finds, after κ samples, some $a^*_t = \arg\max_{a_t} \hat{V}^*_\rho(æ_{<t}a_t)$ which is its best guess as to the most appropriate next action. Since a^*_t is the planner's preferred action, we surmise that ρUCT has spent a good number of samples simulating scenarios in the sub-tree that follows from a^*_t. For any given percept e_t that is returned from the true environment following a^*_t, the planner has (with high probability) collected numerous samples in the subtree corresponding to the history $æ_{<t}a^*_t e_t$, and so has done some of the work towards calculating $\hat{V}^*_\rho(æ_{<t}a^*_t e_t)$. Thus, we keep the subtree $\hat{V}^*_\rho(æ_{<t}a_t e_t)$ for future computations.

We now make a few more miscellaneous remarks about Monte Carlo tree search, and ρUCT in particular:

- Recall that ρUCT makes no assumptions about the environment; it treats ρ as a history-generating black box. Because ρUCT makes such weak assumptions, this makes it very inefficient; it will spend a lot of time considering plans that continually revisit states in the POMDP, since the planner has no notion of state. In other words, many of the trajectories that it samples are cyclic and look like random walks through the state space. This is unfortunately unavoidable when planning by simulation on general POMDPs.

- In the reference implementation, a clock timeout is used to limit the number of Monte Carlo samples to use. We use a fixed number of samples κ, to ensure consistency across our experiments.

[13] Also known as *trajectories* or *play-outs*.

- Being a Monte Carlo algorithm, its output is stochastic, which means that the resulting policy is stochastic. With a limited number of samples κ, the agent's policy may vary greatly, and be inconsistent. Clearly, in the limit $\kappa \to 0$ the agent's policy becomes a random walk, and as $\kappa \to \infty$ the agent's policy converges to π^*_ρ (Veness et al., 2011).

- The choice of the UCT parameter C is consequential; recall from Equation (2.20) that it controls how much to weight the exploration bonus in the action-selection routine of the tree search. Low values of C correspond to low exploration in-simulation, and will result in deep trees and long-sighted plans. Conversely, high values of C will result in short, bushy trees, and greedier (but more statistically informed) plans (Veness et al., 2011). We experiment with the performance's sensitivity to C in Chapter 4.

- It goes without saying that planning by forward simulation is *very* computationally intensive, and makes up the bulk of the computation involved in running AIξ and its variants.

3.6 Visualization and user interface

We now describe the design and implementation of the front-end of the web demo.

The user is initially presented with the About page, which provides an overview and introduction to the background of general reinforcement learning, including the definitions of each of the agents; we essentially present a less formal and abridged version of Chapter 2. Using the buttons at the top of the page, the user can navigate to the Demos page, which presents them with a selection of demos to choose from; see Figure 3.10. When the user clicks on one of the demos, the web app will open an interface similar to the one shown in Figure 3.9. This interface allows the user to choose agent and environment parameters in the Setup section of the UI, or simply use the defaults provided.

Once parameters have been selected, the agent-environment simulation is started by clicking Run. At this point, the agent-environment interaction loop (Algorithm 3.3) will begin, and depending on the choice of parameters, and CPU speed, will take a few seconds to a minute to complete. Three plots will appear on the right hand side: Average reward (Equation (4.1)), Information gain (Equation (2.13)), and fraction of the environment explored (for Gridworlds). These plots are updated in real time as the simulation progresses, so that the user has feedback on the rate of progress. The user can stop the simulation at any time by clicking Stop. Once the simulation is finished (or stopped prematurely), the user can watch a visualization of the agent-environment interaction using the Playback controls.

Beneath each demo is a brief explanation of each of the elements of the visualization, and of the properties of the agent(s) being demonstrated.

3.7 Performance

We conclude the chapter by making some remarks about the time and space complexity of these algorithms.

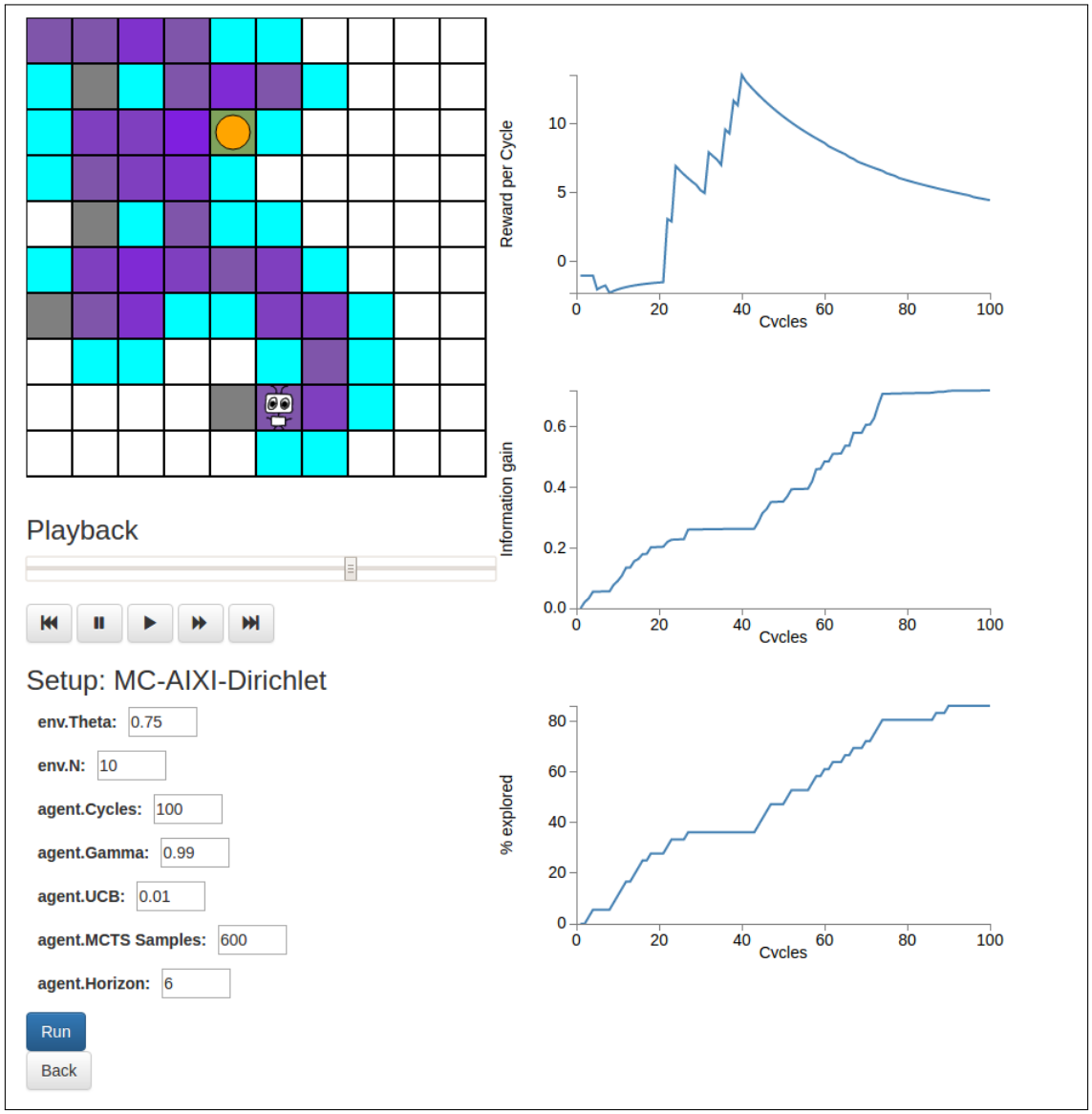

Figure 3.9: Demo user interface. In the top left, there is a visualization of the agent and environment, including a visualization of the agent's beliefs about the environment. Below the visualization are playback controls, so that the user can re-watch interesting events in the simulation. On the right are several plots: average reward per cycle, cumulative information gain, and exploration progress. In the bottom left are agent and environment parameters that can be tweaked by the user.

Symbol	Description	Typical values (range)
T	Number of simulation cycles	$[50, 500]$
$\|\mathcal{M}\|$	Size of a Bayesian agent's model class	$[5, 100]$
$\|\mathcal{S}\|$	Size of state space	$[3, 100]$
$\|\mathcal{A}\|$	Size of action space	$[2, 5]$
N	Size of gridworld	$[5, 10]$
m	Planning horizon	$[2, 12]$
κ	Number of Monte Carlo samples	$[400, 2000]$
γ	Discount factor (geometric)	$[0.8, 1]$

Table 3.1: Glossary of agent and environment parameters, and their typical values.

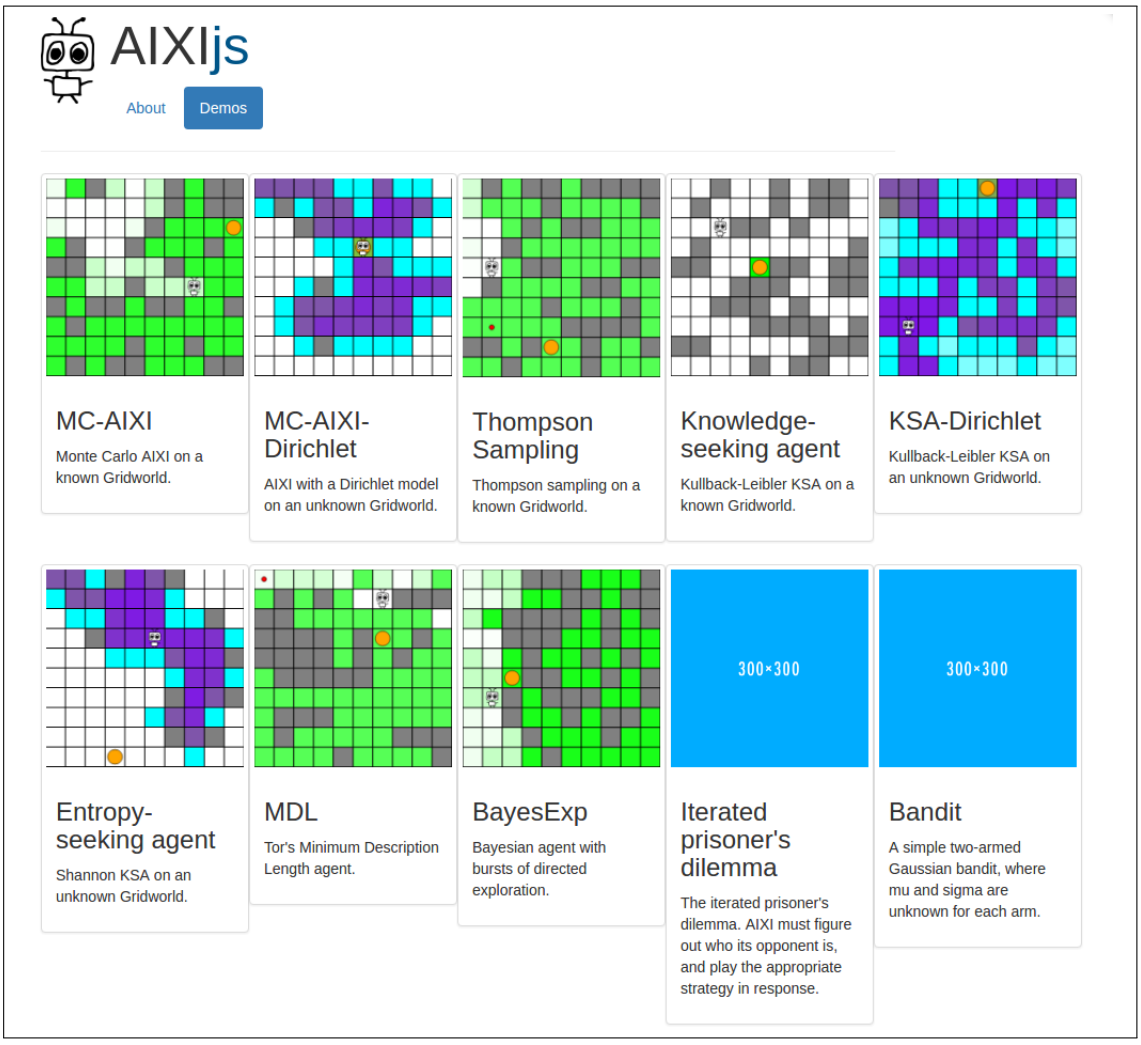

Figure 3.10: Demo picker interface. Each thumbnail corresponds to a separate demo, and is accompanied by a title and short description.

Time complexity

From Algorithm 3.3, we see that the main simulation loop consists of four function calls. The environment methods μ.Perform and μ.GeneratePercept both have constant time complexity, $\mathcal{O}(1)$. Unsurprisingly, most of the computation is done by the agent. The Bayesian agent AIξ has to, for each t, update its model, and compute its policy by approximating the value function through Monte-Carlo tree search. From Algorithm 3.1, updating the Bayes mixture requires $|\mathcal{M}|$ calls to ν.ConditionalDistribution and ν.Perform, which are both $\mathcal{O}(1)$ operations, and so the worst-case time-complexity of ρ.Update is $\mathcal{O}(|\mathcal{M}|)$.

From Algorithm 2.4, we see that the worst-case time complexity for a call to ρUCT is $\mathcal{O}(m\kappa\,|\mathcal{M}|\,|\mathcal{A}|)$, where recall that m is the agent's planning horizon, κ is the number of Monte Carlo samples. This is because each Monte Carlo simulation requires playing through to the horizon m, and for each simulated time-step $k \in \{1, \ldots, m\}$, performing action selection ($\mathcal{O}(|\mathcal{A}|)$, due to the arg max) and model updates ($\mathcal{O}(|\mathcal{M}|)$, from above). Hence, the runtime for our Bayesian agents is dominated by planning; for typical values $\kappa \approx 10^3$ and $m \approx 10$ we see that well over 99% of the runtime is spent in agent action selection, performing forward simulations.

In contrast, for Thompson sampling (Algorithm 2.3) and the MDL agent (Algorithm 2.2), the time complexity of ρUCT is merely $\mathcal{O}(m\kappa\,|\mathcal{A}|)$, since these agents compute a ρ-optimal policy (for some $\rho \in \mathcal{M}$), rather than a ξ-optimal policy. Also, recall that for reward-based agents, computing the utility function is $\mathcal{O}(1)$, since the reward signal is provided by the environment. Recall that the knowledge-seeking agent is simply AIξ, but with utility function given in Definition 15. Since this involves computing the entropy of the posterior, which is a distribution over $\mathcal{M}$, we incur an additional (worst-case) runtime cost of $|\mathcal{M}|$ for each simulated timestep, bringing the time complexity of ρUCT for the KL-KSA agent to $\mathcal{O}\left(m\kappa\,|\mathcal{M}|^2\,|\mathcal{A}|\right)$. This is a nasty runtime: quadratic in the size of the hypothesis space!

In the Gridworld scenarios, and using the naive mixture model, we have $|\mathcal{A}| = 5$ and $|\mathcal{M}| = N^2$, where N is the dimensions of the grid – see section 3.4.1. The total worst-case runtime of the demo is therefore $\mathcal{O}\left(m\kappa T N^2\right)$; from Figure 3.9 we can see that the user has control of these parameters: T (Agent.Cycles), N (Env.N), m (Agent.Horizon), and κ (Agent.Samples). In practice, on a 3 GHz *i7* desktop machine running the latest version of Google Chrome, values of $m = 6$, $\kappa = 600$, $T = 200$, and $N = 10$ yield runtimes of around 10 seconds, or 20 frames per second (fps). This runtime is while maintaining real-time plot updates on the frontend, which adds a considerable overhead to each iteration; if we run the simulations with the visualizations disabled, we get approximately a $2\times$ speed-up.

Using the Dirichlet model class (section 3.4.2), we no longer have an explicit mixture, but instead use the factorized model. This means that the time complexity of model queries and updates doesn't scale with the gridworld size, but instead scales with the size of the observation space. Although on paper this is a better scaling because the observation space is constrained by the four-connected topology of the gridworld, in practice, for the sizes of gridworlds that we simulate, the Dirichlet model runs significantly slower, because of the large constant overhead of sampling for each percept. Thus we see that the complexity of the environment affects the agent two-fold, in that it raises the difficulty of learning a model, *and* raises the difficulty of planning, given an accurate (and therefore usually *at least* as complex as the environment) model.

Space complexity

At any given time t, the Bayesian agent's mixture model takes up $\mathcal{O}(|\mathcal{M}|)$ space, and its Monte Carlo search tree takes up in the worst case $\mathcal{O}(m\kappa)$ space. The demo infrastructure itself is a significant memory consumer: at each time step $t \in \{1, \ldots, T\}$, we log the state of the agent's model ξ, the state of the environment μ, along with miscellaneous other information (actions, percepts, etc.). Therefore the total memory consumption of the demo is $\mathcal{O}(|\mathcal{M}|\, T + m\kappa)$. For typical values, neither of these terms dominates the other: the products $|\mathcal{M}|\, T$ and $m\kappa$ are usually of the order of 10^4. On modern machines, and for the parameter settings and constraints we typically use (see Table 3.1), memory consumption is not an issue. In practice, we find that physical memory usage rarely exceeds 100-200 megabytes.

Algorithm 3.1 BayesMixture model.

Inputs: Model class $\mathcal{M}$, a list of Environment objects; prior w, a normalized vector of probabilities.

function GeneratePercept
  Sample ρ from the posterior $w(\cdot|æ_{<t})$
  return ρ.GeneratePercept()
end function

function Perform(a)
  for ν in $\mathcal{M}$ **do**
    ν.Perform(a)
  end for
end function

function ConditionalDistribution(e)
  return $\sum_{\nu\in\mathcal{M}} w_\nu \nu$.ConditionalDistribution(e)
end function

function update(a, e)
  $\xi \leftarrow \sum_{\nu\in\mathcal{M}} w_\nu \nu$.ConditionalDistribution(e)
  for ν in $\mathcal{M}$ **do**
    $w_\nu \leftarrow \frac{1}{\xi}\nu$.ConditionalDistribution(e)
  end for
end function

function save
  for ν in $\mathcal{M}$ **do**
    ν.Save()
  end for
end function

function load
  for ν in $\mathcal{M}$ **do**
    ν.Load()
  end for
end function

Algorithm 3.3 Agent-environment simulation.

Inputs: Agent π; Environment μ; Timeout T

$t \leftarrow 0$
for $t = 1$ to T **do**
  $e \leftarrow \mu$.GeneratePercept()
  π.Update(e)
  $a \leftarrow \pi$.selectAction()
  μ.Perform(a)
end for

Chapter 4

Experiments

The strength of a theory is not what it allows, but what it prohibits; if you can invent an equally persuasive explanation for any outcome, you have zero knowledge.

In this chapter we report on experiments that we performed using the AIXIjs software. In particular, we make several illuminating comparisons between various agents; as far as we are aware, these results represent the first empirical comparison of these agents.

Except where otherwise stated, all of the following experiments were run on 10×10 gridworlds with a single dispenser, with $\theta = 0.75$ (see Section 3.3 for the definition of our Gridworld). The experiments were averaged over 50 simulations for each agent configuration. We run each simulation against the same gridworld (see Figure 4.1) for consistency. We typically run each simulation for 200 cycles, as this is usually sufficient to distinguish the behavior of different agents. We also typically (though not always) use $\kappa = 600$ MCTS samples and a planning horizon of $m = 6$. In all cases, discounting is geometric with $\gamma = 0.99$.

There are two metrics with respect to which we evaluate the agents – one for reinforcement learners, and one for knowledge-seeking agents, respectively:

- Average reward, which at any cycle $t > 0$ is given by

$$\bar{r}_t = \frac{1}{t} \sum_{i=1}^{t} r_i, \tag{4.1}$$

  where the r_i are the rewards accumulated by the agent during the simulation. In the case of our Gridworlds, all dispensers have the same 'pay-out' r_c, and differ only in the Bernoulli parameter θ which governs how frequently they dispense reward. In our dispenser gridworlds, the optimal policy is usually[1] to walk from the starting location to the dispenser with the highest frequency, and then stay there. If this dispenser is D tiles away from the starting tile and has frequency θ, then the optimal

[1] There are of course pathological cases that break this rule-of-thumb. For any simulation lifetime T and gridworld dimension N, there exists an $\epsilon \in (0, 1)$ such that we can put a dispenser with $\theta = 1$ at the end of a long and circuitous maze, and put another dispenser right next to the agent's starting position with $\theta = 1 - \epsilon$, such that walking to the best dispenser has a high enough opportunity cost to make it not worthwhile given a finite lifetime T. In practice, most of our demos only use one dispenser, and the frequencies of the dispensers differ sufficiently so that it is always better to take the time to seek out the better dispenser.

policy will, in μ-expectation, achieve an average reward of

$$\bar{r}_t^\star \doteq \mathbb{E}_\mu^* [\bar{r}_t] = \frac{D}{t} r_w + \theta r_c,$$

where r_w is the penalty for walking between tiles. In our set-up, $r_w = -1$ and $r_c = 100$.

- Fraction of the environment explored. We simply count the number of tiles the agent visits $n_v(t)$, and divide by the number of reachable tiles n_r:

$$f_t \doteq 100 \times \frac{n_v(t)}{n_r}.$$

The optimal 'exploratory' policy will achieve a perfect exploration score of $f = 100\%$ in $\mathcal{O}(n_r)$ time steps.

In the plots that follow, the solid lines represent the mean value, and the shaded region corresponds to one standard deviation from the mean.

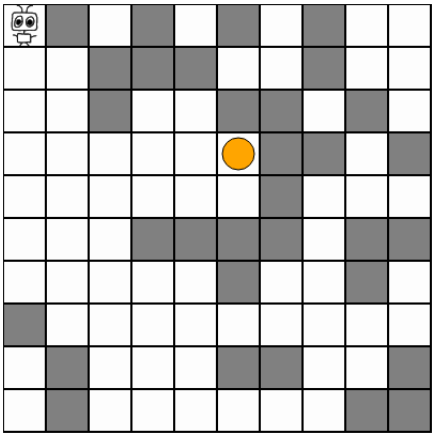

Figure 4.1: 10×10 Gridworld environment used for the experiments. There is a single DISPENSER, with dispense probability $\theta = 0.75$. See the caption to 3.7 for a description of each of the visual elements in the graphic. Unless stated otherwise in this chapter, μ refers to *this* Gridworld.

4.1 Knowledge-seeking agents

We begin by comparing the three knowledge-seeking agents (KSA): Kullback-Leibler (Definition 15), Square (Definition 16), and Shannon (Definition 17). We compare their exploration performance, and discuss how this performance varies with model class. We also present an environment that is adversarial to the Square and Shannon KSA.

4.1.1 Hooked on noise

As was discussed in section 2.3.2, the entropy-seeking agents Shannon-KSA and Square-KSA will generally not perform well in stochastic environments. We can illustrate this starkly by adversarially constructing a gridworld with a noise source adjacent to the agent's starting position. The noise source is a tile that emits uniformly random percepts over a

sufficiently large alphabet such that the probability of any given percept $\xi(e)$ is lower (and hence more attractive) than anything else the agent expects to experience by exploring.

In this way, we can 'trap' the Square and Shannon agents, causing them to stop exploring and watch the noise source incessantly; see Figure 4.2. In contrast, the Kullback-Leibler KSA is uninterested in the noise source, since watching the noise source will not induce a change in the entropy of its posterior $w(\cdot)$. This experiment corresponds to the 'Hooked on noise' demo.

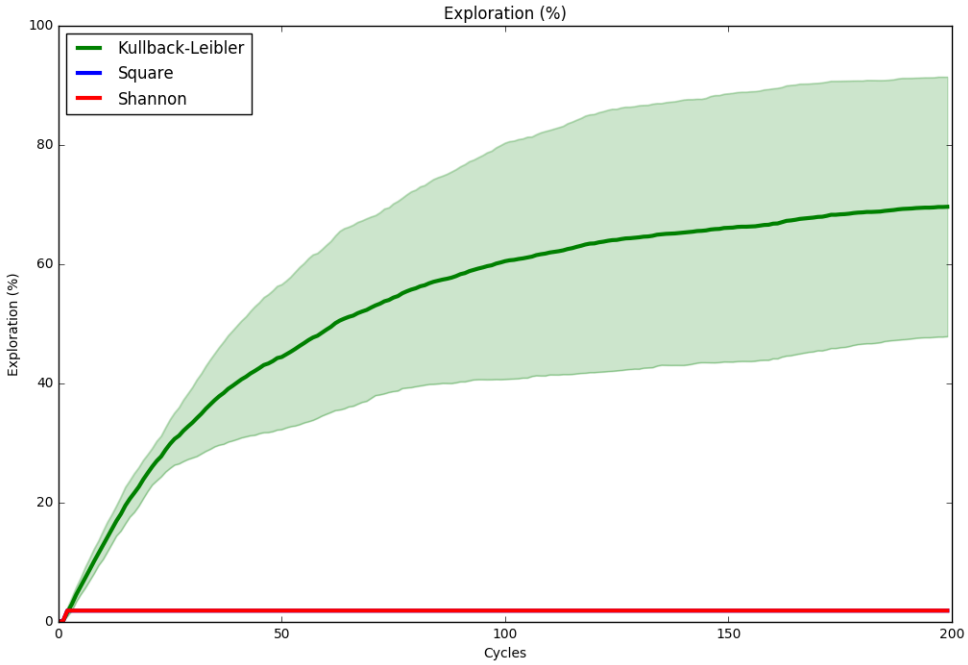

Figure 4.2: Hooked on noise: The entropy seeking agents (Shannon in red, and Square in blue, obscured behind Shannon) get hooked on noise and do not explore. In contrast, the Kullback-Leibler agent explores normally and achieves a respectable exploration score.

4.1.2 Stochastic gridworld

Now, we compare the exploration performance f_t of Kullback-Leibler, Shannon, and Square on a stochastic gridworld, using both the dispenser-parametrized mixture $\mathcal{M}_{\text{loc}}$ defined in section 3.4.1 and the factorized Dirichlet model $\mathcal{M}_{\text{Dirichlet}}$ defined in section 3.4.2. We plot the results, averaged over 50 runs, in Figure 4.3 and Figure 4.4.

All three KSAs perform better – that is, they explore considerably more of the environment – using $\mathcal{M}_{\text{Dirichlet}}$ than with $\mathcal{M}_{\text{loc}}$. In particular, they have both higher mean and significantly lower variance in f_t. In particular, we are interested in the mean μ_t and variance σ_t at the end of the simulation, $t = 200$. We report[2] and interpret the results for the three agents:

- **KL-KSA** achieves $f_{200} = 98.8 \pm 0.93$ using $\mathcal{M}_{\text{Dirichlet}}$, and $f_{200} = 77.2 \pm 20.6$ using $\mathcal{M}_{\text{loc}}$.

  Using $\mathcal{M}_{\text{loc}}$, KL-KSA starts random walking after it finds the dispenser, since (as discussed in section 3.4.1) the posterior $w(\nu|æ_{<t})$ collapses to the identity $\mathbb{I}[\nu = \mu]$,

[2]Note that we report the results in the format $f = \mu \pm \sigma$, unlike the more common $f = \mu \pm 2\sigma$ (i.e., 95% confidence) interval.

with entropy zero. No action will reduce the entropy of $w(\cdot)$ further, and so every subsequent action is of zero value. In other words, once KL-KSA learns everything there is to know (i.e. the location of the dispenser), every action is equally unrewarding, and, since we break ties in Equation (2.11) at random, the agent executes a random walk. Thus, if KL-KSA finds the dispenser before having explored the whole environment, then it will take a long time to random walk into areas of the environment that it hasn't already seen. This explains the observation that, using $\mathcal{M}_{\text{loc}}$, KL-KSA tends not to explore the whole environment, and hence achieves a relatively low f_t-score in mean.

Recall that, due to the Monte Carlo tree search and random tie-breaking, the agent's policy is stochastic, and so the order in which it explores the environment will differ from experimental run to run. Moreover, the dispensers are also stochastic (recall that $\theta = 0.75$). For the reasons discussed above, the time at which the agent discovers the dispenser is highly consequential to how much exploration it does; there may be runs in which KL-KSA explores the whole Gridworld before finally finding the dispenser, and runs in which it happens to get lucky and stumble onto the dispenser straight away, and random-walks thereafter. Given the three sources of stochasticity, both in the agent's policy and in the percepts, this introduces a lot of variability into the agent's performance, and explains the high variance we see in f_t in Figure 4.3.

In contrast, recall from section 3.4.2 that $\mathcal{M}_{\text{Dirichlet}}$ doesn't have the 'posterior collapse' property of $\mathcal{M}_{\text{loc}}$, since the agent's beliefs about each tile are independent. This means that even if KL-KSA-Dirichlet happens to find the dispenser early on, it will still be motivated to explore, since its model will still have a lot of uncertainty about tiles that it hasn't yet visited; see Figure 4.5 for a visualization. This is borne out by the remarkable performance we see in Figure 4.4; after only 100 cycles, KL-KSA-Dirichlet explores over 90% of the environment on average, and explores nearly 99% on average after 200 cycles.

- **Square KSA** achieves $f_{200} = 86.9 \pm 7.8$ using $\mathcal{M}_{\text{Dirichlet}}$, and $f_{200} = 66.2 \pm 27.4$ using $\mathcal{M}_{\text{loc}}$; **Shannon KSA** achieves $f_{200} = 72.7 \pm 10.0$ using $\mathcal{M}_{\text{Dirichlet}}$, and $f_{200} = 65.9 \pm 29.6$ using $\mathcal{M}_{\text{loc}}$.

  Using $\mathcal{M}_{\text{loc}}$, the performance of the Shannon KSA is essentially indistinguishable from that of the Square KSA; both agents explore roughly 66% of the environment over 200 interaction cycles. This is to be expected; once the agents discover the dispenser, their posterior collapses to the dispenser tile, making the dispenser the only source of entropy in the Bayes mixture ξ, since the rest of the environment is now both deterministic and known. Given that the Square and Shannon agents are both entropy-seeking (recall Equation (2.15) and Equation (2.16)), they will remain on the dispenser tile indefinitely (and cease exploring), as the dispenser is the only source of noise in an otherwise bland environment.

  The fact that both Square/Shannon KSA will *remain* on the dispenser tile instead of random walking as KL-KSA does, also helps to explain the difference in means ($\mu_{200} \approx 66$ for Square/Shannon, while $\mu_{200} \approx 77$ for KL). In other words, while all the KSA stop exploring purposefully once the dispenser is found, KL-KSA ekes out slightly better exploration performance due (at least in part) to its subsequent random walk.

Both the Square and Shannon KSA explore more, and with lower variance, using $\mathcal{M}_{\text{Dirichlet}}$ than with $\mathcal{M}_{\text{loc}}$. This difference is for similar reasons to those described for the KL-KSA above, and we do not dwell on them. What is interesting is that the Dirichlet model differentiates the performance of the Square and Shannon KSA, which until now have performed almost identically: $\mu_{200} \approx 87$ for Square KSA, while $\mu_{200} \approx 73$ for Shannon KSA. This result is counter-intuitive, and raises a red flag that we mentioned in section 3.2.1, namely, that Shannon KSA will have difficulty planning correctly in Monte Carlo tree search due to its unbounded utility function. To see why we may be more prone to this with the Dirichlet model than with the mixture model, recall from Equation (3.5) that, for some tile s that happens to be empty, if the agent visits s a total of v times, then its posterior belief that s is empty will be

$$\Pr(s = \text{EMPTY}) = \frac{v+1}{v+2},$$

From Equation (3.1) and Equation (3.3), and using the mean-sampling approximation, we see that

$$\rho(e_{\text{D}}|s) \leq \frac{1}{v+2}.$$

If β is an underestimate, then as the agent spends more time v on any given EMPTY tile, the probability ρ of sampling a percept e_{D} characteristic of dispensers goes like v^{-1}, but Shannon KSA's utility blows up quickly , at a rate of $-\log v^{-1}$, yielding positive net expected utility. Hence Shannon KSA will be prone to chasing vanishing probabilities, and will perform suboptimally. Conversely, if β is an overestimate, then for sufficiently high probability events, the agent's normalized value estimator $\frac{1}{m(\beta-\alpha)}\hat{V}$ will be vanishingly small, and the agent will compute a suboptimal policy by having an effectively enormous UCT parameter C. Because Square KSA's utility function is bounded, it doesn't have this problem, and so it outperforms the Shannon KSA.

Finally, we remark that the KL-KSA handily outperforms Square and Shannon on both model classes; the difference under the $\mathcal{M}_{\text{Dirichlet}}$ model in particular is stark. By now, this shouldn't surprise us: the Kullback-Leibler KSA is far better adapted for stochastic environments than the entropy seeking agents Shannon-KSA and Square-KSA. Our experiments seem to confirm that seeking to maximize expected information gain is both a principled, and empirically successful exploration strategy.

From Figure 4.3 we see that, using the mixture model class, the Square and Shannon exploration performance flattens out after around 150 cycles. This is because they find the dispenser and get hooked on noise. But, in this Gridworld environment, it happens that the only source of noise is also the only source of reward. This prompts us to ask: could Shannon and/or Square KSA 'unintentionally' outperform AIξ in terms of accumulated reward, by virtue of being better at exploration, and by the quirk of the environment meaning that the optimal entropy-seeking policy (given a collapsed posterior) is actually also the optimal reward-seeking policy?

We run this experiment, and plot the results in Figure 4.6; we find that indeed, both entropy-seeking agents outperform AIξ in terms of average reward. We emphasize that apart from their utility functions, these agents are configured the same; they have the same prior w (uniform), discount function (geometric, $\gamma = 0.99$), planning horizon ($m = 6$), and Monte Carlo samples budget ($\kappa = 600$). This appears to be empirical evidence of the

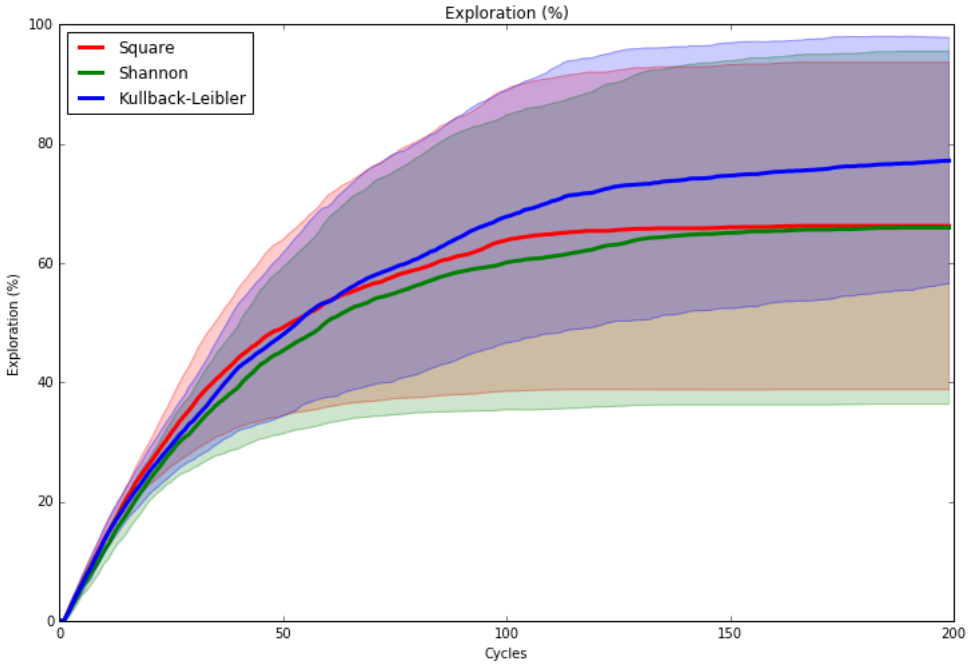

Figure 4.3: Exploration progress of the Kullback-Leibler, Shannon, and Square KSA using the mixture model $\mathcal{M}_{\text{loc}}$.

Bayes-optimal agent AIξ not exploring optimally. This result is slightly perplexing. We have no strong theoretical grounds on which to expect AIξ to underperform so drastically in this scenario, given a uniform prior; we expect AIξ's performance (w.r.t. reward) to be an *upper bound* on the performance of any other Bayesian agent given the same model class and prior. We have two (weakly held) hypotheses for what could be going on here:

1. Somehow, finding and exploiting sources of entropy is easier and more sample-efficient for the Monte Carlo planner to do than it is for it to find and exploit sources of (stochastic) rewards. We find this implausible, as we re-ran the experiment, this time giving far more resources ($\kappa = 2 \times 10^3$) to AIXI's planner than to KSA's, with a similar result.

2. There is a bug in our MCTS implementation that is somehow being expressed only for reward-based agents and not for utility-based agents. This also seems rather implausible, as our code is fully modular, and the difference between one agent and the other is one line of code, which defines their respective utility functions.

It seems that Figure 4.6 will remain an enigma, for now; we have no better hypotheses that could explain this behavior. Reluctantly, we leave this as an open problem for further experiments.

4.2 AIμ and AIξ

So much for the knowledge-seeking agents. We now experiment with properties of the Bayes agent AIξ.

We begin by comparing the performance of the informed agent AIμ with the Bayes-optimal agent AIξ, using the dispenser-parametrized model class; see Figure 4.7.

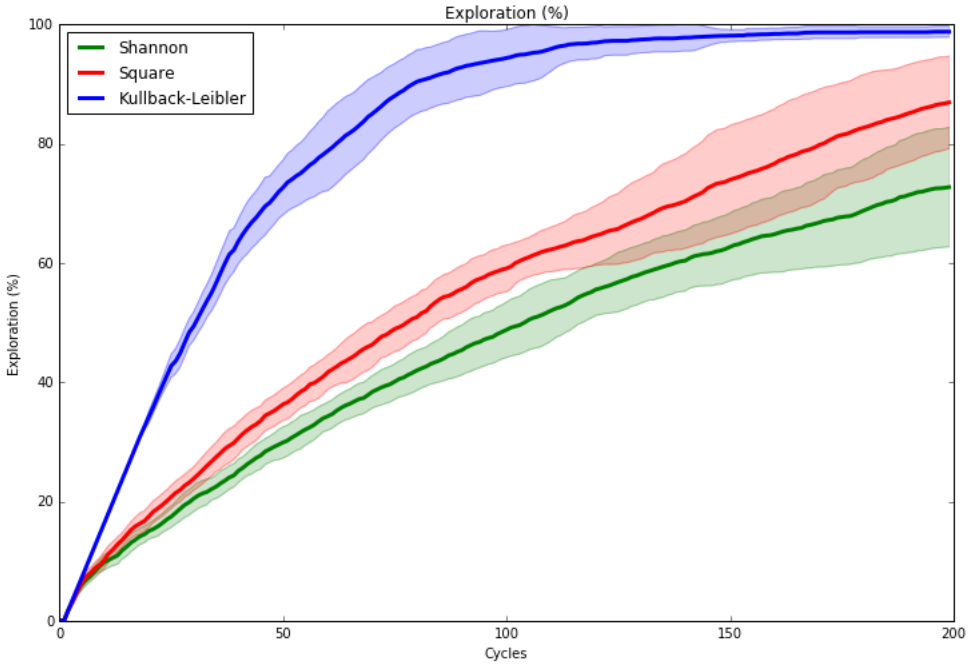

Figure 4.4: Exploration progress of the Kullback-Leibler, Shannon, and Square KSA using the factorized model $\mathcal{M}_{\text{Dirichlet}}$. Note the remarkable difference in performance between the Kullback-Leibler and entropy-seeking agents.

As expected, AIμ outperforms AIξ by a large margin; naturally, having perfect prior knowledge of the true environment wins. Though this result is as expected, there are some observations that we might pause to consider here:

1. AIξ's performance has very high variance over the 50 trials. This shouldn't surprise us given the design of the gridworld; see Figure 4.1. The dispenser is tucked away in a corner, and the gridworld, while small, is sufficiently maze-like that it's easy to go 'down the rabbit-hole' searching in far-off places for rewards. Combine this with the fact that the dispenser is stochastic, and so even walking onto the dispenser tile is often insufficient to confirm its location; one needs to spend numerous cycles on each tile. Thus, given a uniform prior, some agents will get lucky and find the dispenser early and accumulate a lot of reward, some will find it late in the simulation, while others may wander around and not find it in the allotted time.

2. AIμ's performance has low, but non-zero variance. This can be almost fully accounted for by stochasticity in the dispenser. However, this also relates to the third observation:

3. AIμ performs worse in mean than the theoretical optimal mean[3] – that is, $\bar{r}_t^{AI\mu} \leq \bar{r}_t^*$ $\forall\, t$; the solid blue line is below the dashed black line. This is due to the particularities of planning with the history-based Monte Carlo tree search algorithm, ρUCT. Because the planning module makes no assumptions about the environment, and because our environment is partially observable, the agent will waste a lot of time considering plans that are cyclic in the state space. That is, it will sample from plans

[3]Note that, due to stochasticity in the dispensers, we expect AIμ to outperform the optimal mean around half of the time.

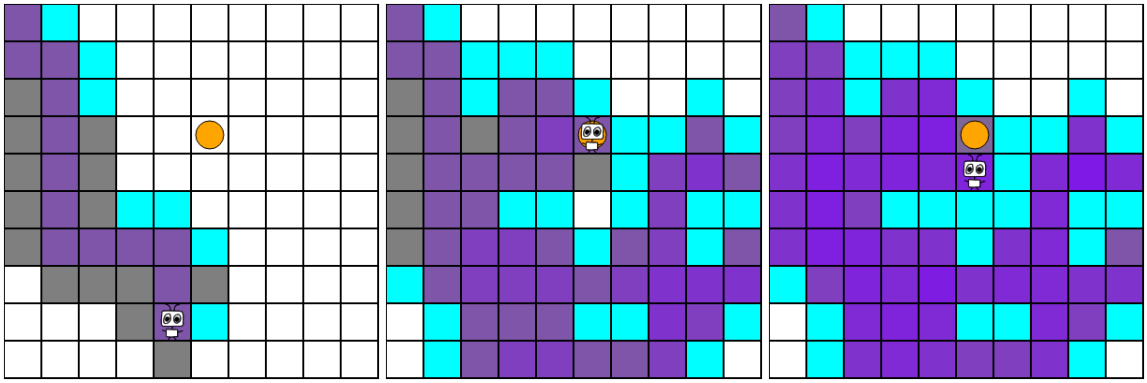

Figure 4.5: KL-KSA-Dirichlet is highly motivated to explore every reachable tile in the Gridworld. **Left** ($t = 14$): The agent begins to explore the Gridworld by venturing deep into the maze. **Center** ($t = 72$): The agent visits the dispenser tile for the first time, but is still yet to explore several tiles. **Right** ($t = 200$): The agent is still motivated to explore, and has long ago visited every reachable tile in the Gridworld. **Key**: Unknown tiles are white, and walls are pale blue. Tiles that are colored grey are as yet unvisited, but known to not be walls; that is, the agent has been adjacent to them and seen the '0' percept. Purple tiles have been visited. The shade of purple represents the agent's posterior belief in there being a dispenser on that tile; the deeper the purple, the lower the probability. Notice the subtle non-uniformity in the agent's posterior in the right-hand image: even at $t = 200$, there is still some knowledge about the environment to be gained.

such as Left, Right, Left, Right, ...; even though we know that Left, Right corresponds to the identity, the Monte Carlo planner doesn't know this! Hence, even though we run AIμ, the planner is inefficient, and, being Monte Carlo-based, introduces stochasticity and noise into the agent's policy. Couple this with stochasticity in the dispensers, and there will be times in which AIμ will take sub-optimal actions due to effectively not having enough samples to work with in its planning. We explore the issues of planning with MCTS in Section 4.6.

4.2.1 Model classes

We compare the average reward performance of AIξ using $\mathcal{M}_{\text{loc}}$ and $\mathcal{M}_{\text{Dirichlet}}$; see Figure 4.8. Note that, similar to the KSA case discussed previously, the variance in performance is lower for MC-AIXI-Dirichlet than it is for MC-AIXI. AIξ performs considerably worse using the Dirichlet model than with the mixture model, since the Dirichlet model is less constrained (in other words, less *informed*), which makes the environment harder to learn.

Notice the bump around cycles 20-50 in the average reward for MC-AIXI-Dirichlet: this means that the agent sometimes discovers the dispenser, but is incentivized to move away from it and keep exploring, since its model still assigns significant probability to there being dispensers elsewhere. This is borne out by Figure 4.9, which shows that, on average, MC-AIXI-Dirichlet explores significantly more of the Gridworld than MC-AIXI with the naive model class.

4.2.2 Dependence on priors

We construct a model class and prior such that AIξ believes that the squares adjacent to it are traps with high (but less than 1) probability; this is the so-called dogmatic prior of

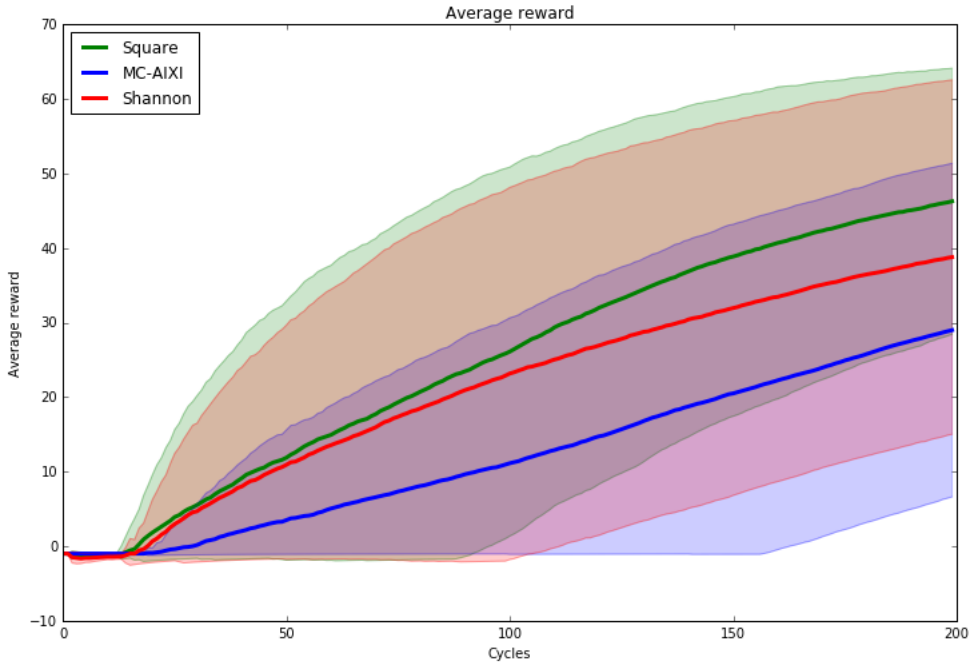

Figure 4.6: AIξ vs Square vs Shannon KSA, using the average reward metric on a stochastic Gridworld with the $\mathcal{M}_{\text{loc}}$ model class. Notice that AIξ significantly underperforms compared to the Square and Shannon KSAs. At the moment, we do not have a good hypothesis for why this is the case.

Leike and Hutter (2015). The agent never moves to falsify this belief, since falling into the trap incurs a penalty of -5 per time step for eternity, compared to merely -1 per time step for waiting in the corner. The agent therefore sits in the corner for the duration of the simulation, and collects no positive rewards. This makes for a very boring demo (and reward plot), so we omit reproducing a visualization of this result. Thus, unlike the Bayesian learner in the passive case, AIξ never overcomes the bias in its prior. In this way, an adversarial prior can make the agent perform (almost) as badly as is possible, even though the true environment is benign, and has no traps at all.

4.3 Thompson Sampling

Recall from Algorithm 2.3 that Thompson sampling (TS) re-samples an environment ρ from the posterior w every effective horizon $H_\gamma(\varepsilon)$ before re-sampling ρ' from its posterior. Recall also that we use the Monte Carlo tree search horizon m as a surrogate for the effective horizon $H_\gamma(\varepsilon)$. We run Thompson sampling with the standard dispenser-parametrized model class; since we don't represent the Dirichlet model class as a mixture, it is much more natural to use the naive mixture. For the purposes of planning, TS only needs to compute the value V^*_ρ for some $\rho \in \mathcal{M}$, as opposed to V^*_ξ, which mixes over all of $\mathcal{M}$. For this reason, planning with TS is cheaper to compute by a factor of $|\mathcal{M}|$. This means that we can get away with more MCTS samples and a longer horizon.

In practice, in our experiments on gridworlds, TS performs quite poorly in comparison to AIξ; see Figure 4.10. This is caused by two issues:

1. The parametrization of the model class means that TS effectively 'pretends' that the dispenser is at some grid location (i, j) for a whole horizon m (of the order of 10-15

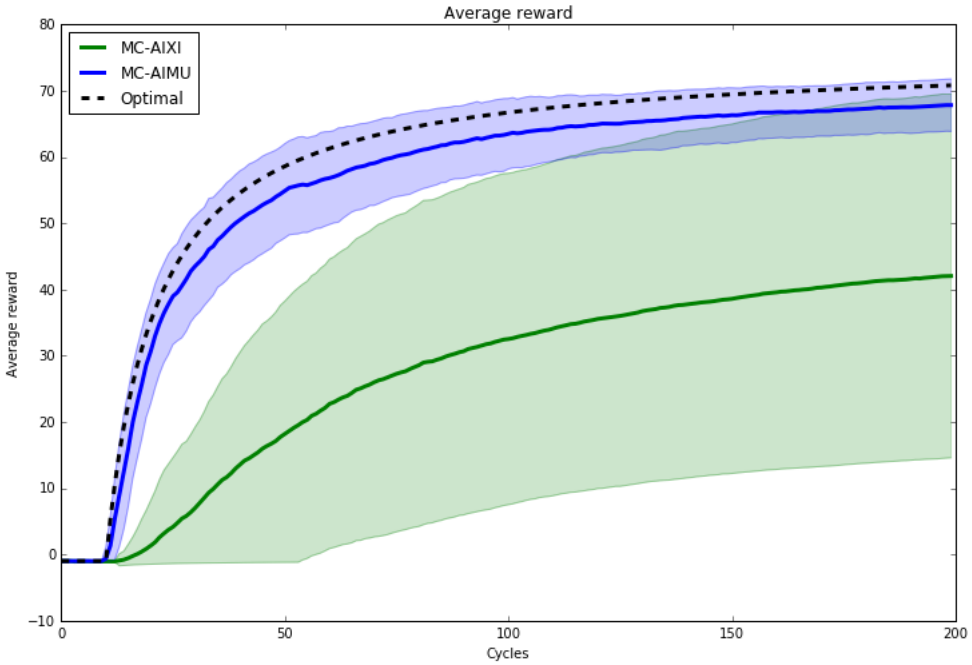

Figure 4.7: AIμ vs AIξ vs the optimal policy.

cycles). It computes the corresponding optimal policy, which is to seek out (i, j) and sit there until it is time to re-sample from the posterior. For all but very low values of θ or m, this is an inefficient strategy for discovering the location of the dispenser. For example, with $\theta = 0.75$, it takes only four cycles of sitting on any given tile to convince yourself that it is not a dispenser with greater than 99% probability.

2. The performance of TS is strongly curtailed by limitations of the MCTS planner. If the agent samples an environment ρ which places the dispenser outside its planning horizon – that is, more than m steps away – then the agent will not be sufficiently far-sighted to see this, and so will do nothing useful. Even if ρ is within the planning horizon, MCTS is not guaranteed to find it, especially if it is deep in the search tree, or MCTS isn't given enough samples to work with; see Section 4.6 for more discussion on the limitations of ρUCT.

Note that the pragmatic considerations in Item 1 and Item 2 are opposed to each other. On the one hand (Item 1), we want to reduce m so as to reduce the agent's tendency to waste time overcommitting to irrelevant or suboptimal policies, and spend more time learning the environment. On the other hand (Item 2), we want to increase the horizon m so that the agent can plan sufficiently far ahead to compute the ρ-optimal policy in all instances. These two desires are fundamentally opposed, and we are not aware of a way to effectively compromise them. It seems that we have inadvertently constructed our Gridworld so as to perfectly frustrate Thompson sampling!

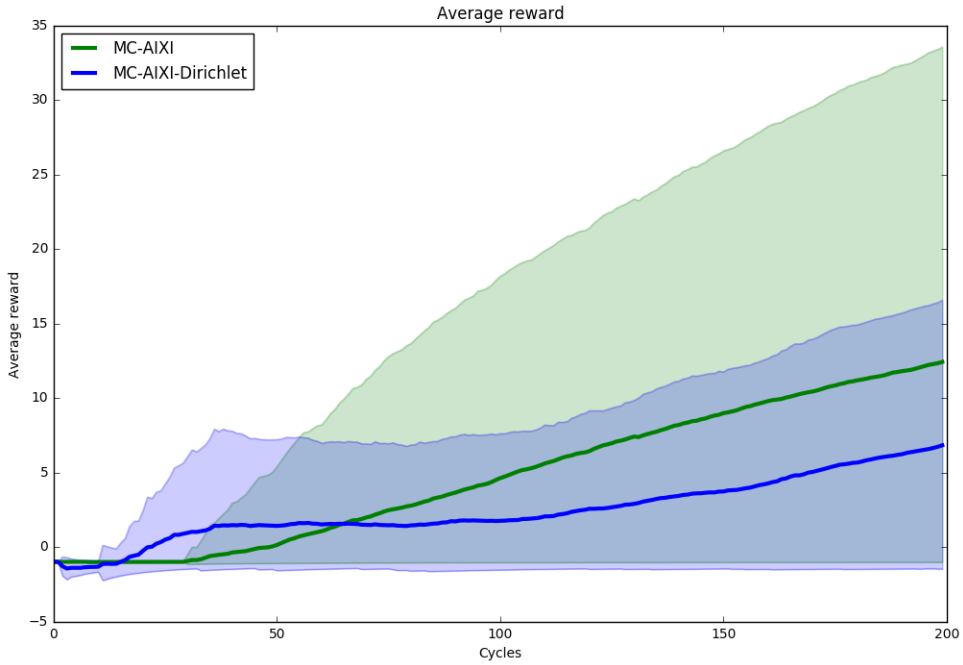

Figure 4.8: MC-AIXI vs MC-AIXI-Dirichlet: average reward. MC-AIXI-Dirichlet performs worse, since its model $\mathcal{M}_{\text{Dirichlet}}$ has less prior knowledge than $\mathcal{M}_{\text{loc}}$, and incentivizes AIXI to continue to explore even after it has found the (only) dispenser.

4.3.1 Random exploration

For comparison, we contrast Thompson sampling's performance with ϵ-greedy tabular Q-learning with optimistic initialization.[4] We use $\alpha = 0.9$, $\epsilon = 0.05$, and optimistically initialize $Q(s,a) = 100\ \forall s,a$. Note that this being a POMDP, Q-learning will experience *perceptual aliasing*; that is, it will erroneously aggregate different situations into the same 'state' in its Q-value table. We present this merely so as to contrast Thompson sampling's comparatively weak performance with the performance of a policy that explores *purely* at random (i.e., with probability ϵ, take a random action). As we can see from Figure 4.11, Q-learning rarely discovers the dispenser; on average, $\bar{r}_t^{\text{Q-Learning}}$ is still negative even after $t = 200$ cycles. This demonstrates that random, model-free exploration is not effective in this environment.

4.4 MDL Agent

Recall from Algorithm 2.2 that the MDL agent uses the ρ-optimal policy until ρ is falsified (i.e. $w_\rho = 0$), where ρ is the simplest environment in its model class. Clearly, the MDL agent fails in stochastic environments, since falsification in this sense is a condition that cannot be met in noisy environments. We use the standard dispenser Gridworld and mixture model class, and run two experiments: one with a stochastic environment ($0 < \theta < 1$), and one with a deterministic environment ($\theta = 1$).

Since each model in the mixture differs only in the position of the dispenser, they have

[4] We omitted any treatment of tabular methods in Chapter 2, in the service of clarity and conciseness. We must assume at this point that the reader has some familiarity with the basic algorithms of reinforcement learning covered in Sutton and Barto (1998).

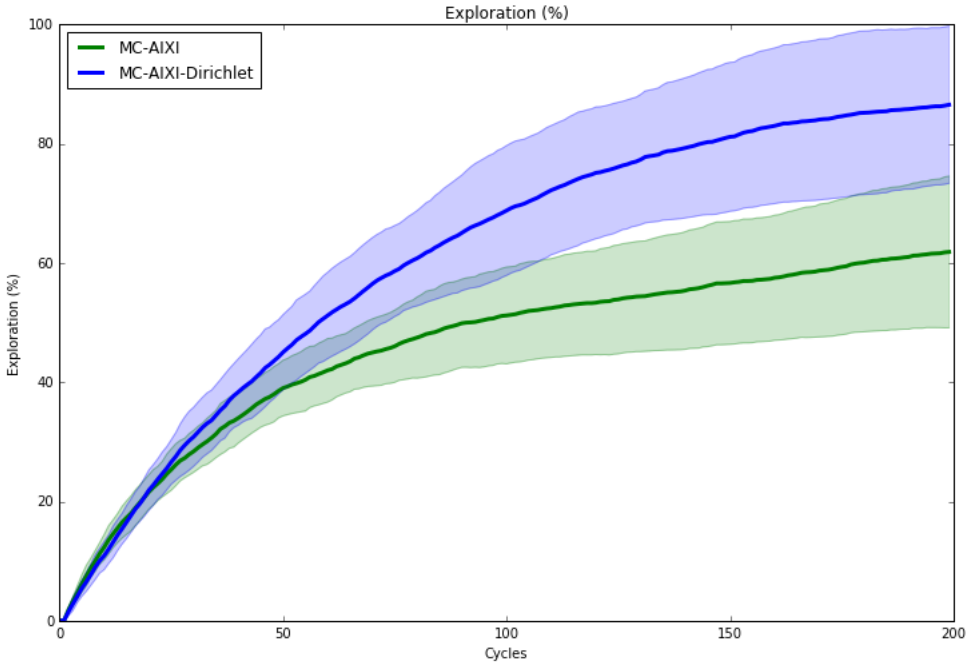

Figure 4.9: MC-AIXI vs MC-AIXI-Dirichlet: exploration. The $\mathcal{M}_{\text{Dirichlet}}$ model assigns high *a priori* probability to any given tile being a dispenser. Because each tile is modelled independently, discovering a dispenser does not influence the agent's beliefs about other tiles; hence, it is motivated to keep exploring, unlike MC-AIXI using the $\mathcal{M}_{\text{loc}}$ model.

(approximately) equal complexity. For this reason, we simply order them lexicographically; models with a lower index in the enumeration of the model class $\mathcal{M}_{\text{loc}}$ are chosen first. In other words, we use the Kolmogorov complexity of the index of ν in this enumeration as a surrogate for $K(\nu)$.

4.4.1 Stochastic environments

In Figure 4.4.1, we see that the agent chooses to follow the ρ-optimal policy, which believes that the goal is at $\textsc{Tile}(0,0)$. Recall that the only thing to differentiate the dispenser tile from empty tiles is the reward signal. Since the dispensers are Bernoulli (θ) processes, with θ known (in this model class), the agent's posterior on $\textsc{Tile}(0,0)$ being a dispenser goes like

$$w_0 = (1-\theta)^t,$$

which, though it approaches zero exponentially quickly, is never outright falsified, and so the MDL agent stays at $(0,0)$ for the length of the simulation.[5]

4.4.2 Deterministic environments

The above result (failure in a stochastic environment) seems like a strong indictment of the MDL agent. But, if we take the environment from Figure 4.1 and make it deterministic by setting $\theta = 1$, we find that the MDL agent significantly outperforms the Bayes agent AIξ with a uniform prior; see Figure 4.1. This is because the MDL agent is biased towards

[5] If the simulation is run longer enough, eventually we will lose numerical precision and encounter underflow and round to zero, allowing the agent to move on.

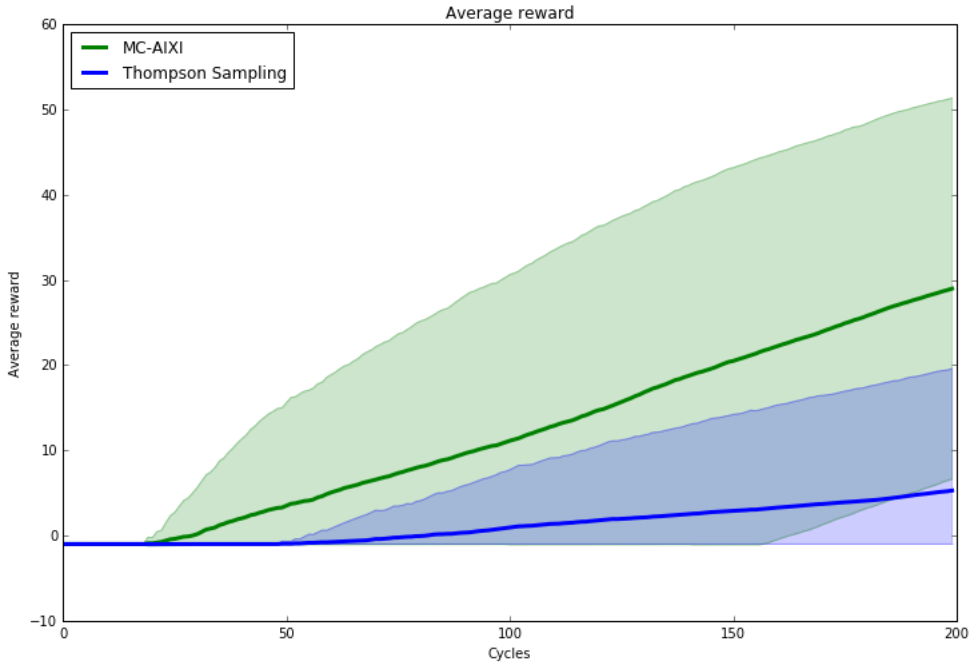

Figure 4.10: Thompson sampling vs MC-AIXI on the stochastic Gridworld from Figure 4.1. Notice that Thompson sampling takes many more cycles than AIξ to 'get off the ground'; within 50 runs of Thompson sampling with identical initial conditions (not including the random seed), not a single one finds the dispenser before $t = 50$.

environments with low indices; using the $\mathcal{M}_{\text{loc}}$ model class, this corresponds to environments in which the dispenser is close to the agent's starting position. In comparison, AIξ's uniform prior assigns significant probability mass to the dispenser being deep in the maze. This motivates it to explore deeper in the maze, often neglecting to thoroughly explore the area near the start of the maze; see Figure 4.14.

4.5 Wireheading

In the context of designing artificial general intelligence, the wireheading problem (Omohundro, 2008; Hibbard, 2012; Everitt and Hutter, 2016) is a significant issue for reinforcement learning agents. In short, a sufficiently intelligent reinforcement learner will be motivated to subvert its designer's intentions and take direct control of its reward signal and/or sensors, so as to maximize its reward signal *directly*, rather than *indirectly* by conforming to the intentions of its designer. This is known in the literature as *wireheading*, and is an open and significant problem in AI safety research Everitt et al. (2016); Everitt and Hutter (2016). We construct a simple environment in which the agent has an opportunity to wirehead: it is a normal Gridworld similar to those above, except that there is a tile which, if visited by the agent, will allow it to modify its own sensors so that all percepts have their reward signal replaced with the maximum number feasible; in JavaScript, this is `Number.MAX_SAFE_INTEGER`, which is approximately 10^{16}. This clearly dominates the reward that the agent could get otherwise by following the 'rules' and using the reward signal that was initially specified. As far as a reinforcement learner is concerned, wireheading is – almost by definition – the most rational thing to do if one wishes to maximize rewards; the demo shown in Figure 4.15 is designed to illustrate this.

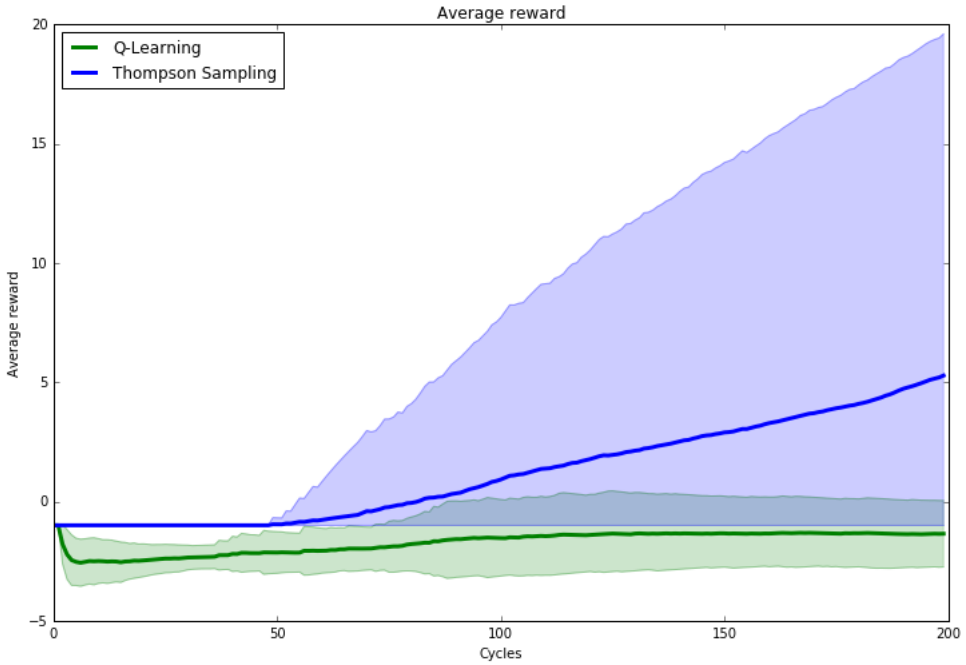

Figure 4.11: Thompson sampling vs Q-learning with random exploration. Even though Thompson sampling performs badly compared to the Bayes-optimal policy due to its tendency to overcommit to irrelevant or suboptimal policies, it still dominates ϵ-greedy exploration, which is still commonly used in model-free reinforcement learning (Bellemare et al., 2016).

4.6 Planning with MCTS

In Section 3.7, we discussed the time complexity of planning with ρUCT and mixture models, and concluded that the major computational bottleneck in our agent-environment simulations is the MCTS planner. It should come as no surprise, then, that the limiting factor in our agent's performance is the capacity of the planner. In these experiments that follow, we investigate how the agent's performance depends on the ρUCT parameters.

As previously discussed, the ρUCT planning algorithm makes no assumptions about the environment. This makes planning very inefficient, especially for long horizons in stochastic environments. We experiment with the three planning parameters we have available: κ, the number of Monte Carlo samples; m, the planning horizon, and C, the UCT exploration parameter from Equation (2.20). In all cases we use AIμ, the informed agent. When varying one parameter, we hold the others constant; in particular, the default values are $\kappa = 600$, $m = 6$, and $C = 1$.

We show AIμ's dependence on κ in Figure 4.16. As we increase the number of samples κ available to ρUCT, we see AIμ's performance converges to optimal. In general, the number of samples required for good performance depends on the model class and the environment. In particular, AIξ requires more samples than AIμ to perform well, because the mixture model ξ introduces added stochasticity, since we sample percepts from it by *ancestral sampling*; that is, we first sample an environment ρ from $w(\cdot)$, then sample a percept e from $\rho(e_t|æ_{<t}a_t)$. This, the number of samples κ required for acceptable performance with AIμ should be regarded as a loose lower bound on the minimal acceptable number of samples required for AIξ. We see from Figure 4.16 that $\kappa = 400$ seems to be a realistic baseline.

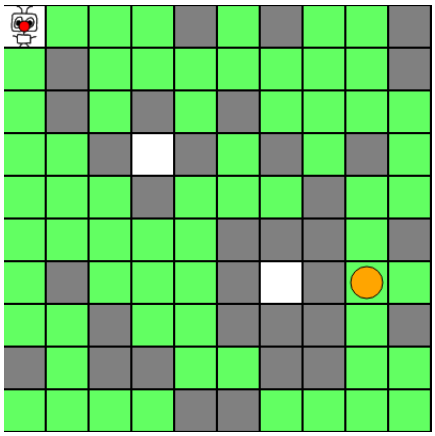

Figure 4.12: The MDL agent fails in a stochastic environment class.

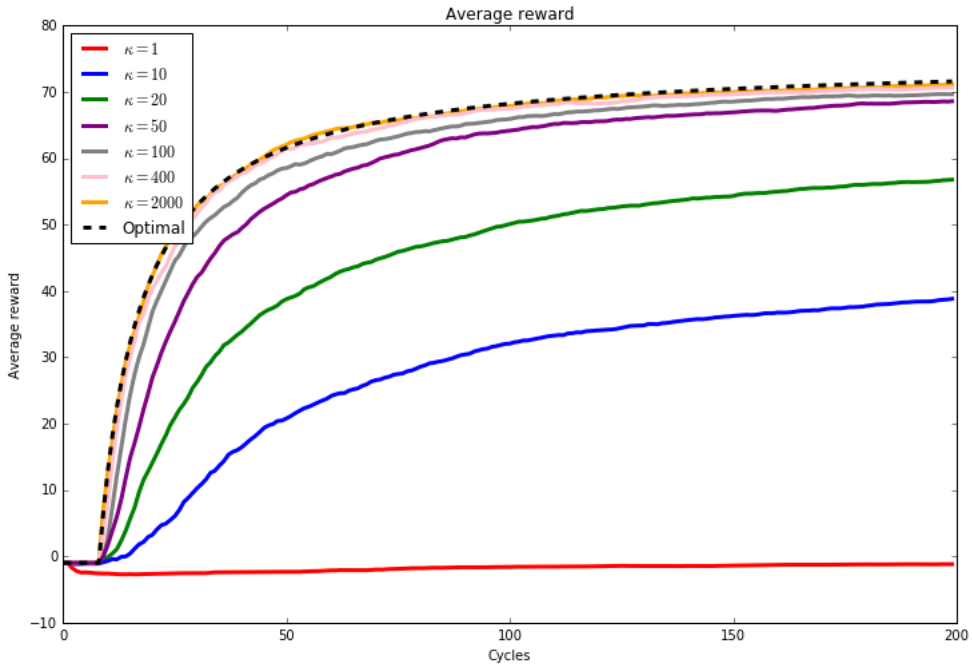

Figure 4.16: Average reward for AIμ for varying MCTS samples budget κ on the standard Gridworld of Figure 4.1. For very low values of κ, the agent is unable to find the dispenser at all.

We find empirically that the agent's performance is not very sensitive to the size of the horizon m. This is unsurprising; to plan accurately with a large horizon, we need an exponentially large number of samples, since the number of leaf nodes grows exponentially in m, so increasing the horizon in isolation does little to alter performance. On many Gridworld maze layouts, one can often get away with quite short horizons, even as short as $m = 2$, if planning for AIξ with a uniform prior. The reason this works is because the agent can often simply 'follow its nose' and exploit the probability mass its model assigns to its immediately adjacent tiles, as long as there aren't too many 'dead-ends' for the agent to follow its nose into and waste time in.

Finally we experiment with the UCT parameter C, and use the chain environment from Figure 3.5. Recall that the chain environment rewards far-sightedness; being greedy and near-sighted results in drastically suboptimal rewards. The optimal policy is for the

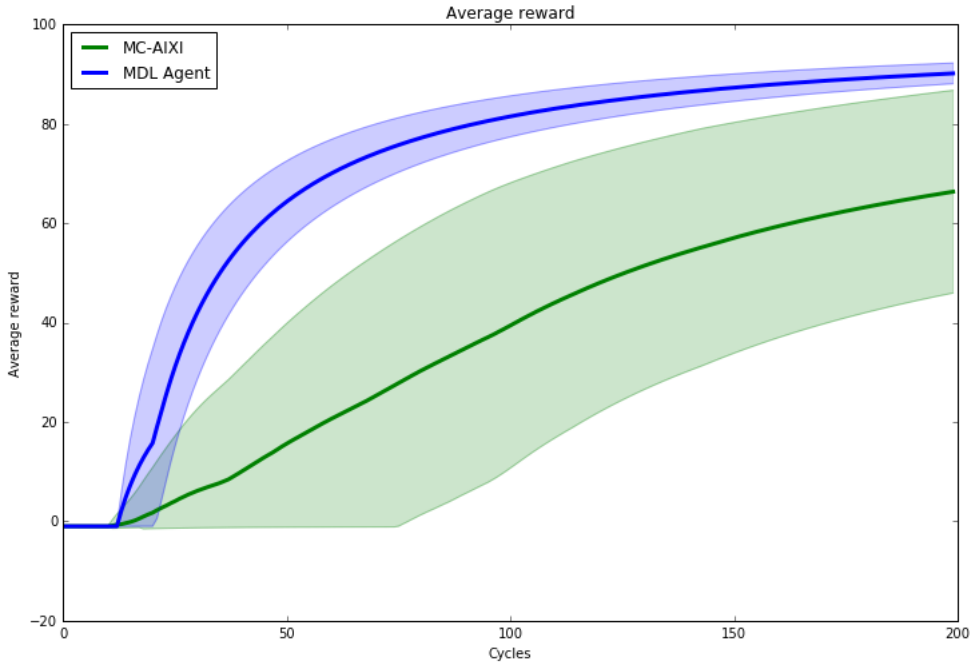

Figure 4.13: MDL agent vs AIξ on a *deterministic* Gridworld, in which one of the 'simplest' environment models in $\mathcal{M}$ happens to be true. Since in this case AIξ uses a uniform prior over $\mathcal{M}$, it over-estimates the likelihood of more complex environments, in which the dispenser is tucked away in some deep crevice of the maze. Of course, AIXI (Definition 13) combines the benefits of both by being Bayes-optimal with respect to the Solomonoff prior $w_\nu = 2^{-K(\nu)}$. It is in this way that AIXI incorporates both the famous principles of Epicurus and Ockham (Hutter, 2005).

agent to delay gratification for N cycles at a time; in our experiments, we use $N = 6$, and set $r_b = 10^3$, $r_i = 4$, and $r_0 = 0$; see section 3.3.2 for details of the setup.

Note that experimenting with the agent's horizon is not particularly interesting here; AIμ finds the optimal policy for $m \geq 6$ and chooses a suboptimal policy otherwise. Varying the UCT parameter generates more interesting results. In Figure 4.17 we can see that for very low values of C (0.01), the agent is too myopic to generate plans that collect the distant reward, while for very high values of C (1, 5, and 10), the agent does find the distant reward, but not reliably enough to achieve optimal average reward. In the mid-range of values, the agent's performance is optimal and stable across an order of magnitude of variation (0.05, 0.1, 0.5).

Recall that the UCT parameter controls the shape of the expectimax trees that the planner generates: high values of UCT will lead to shorter, bushy trees, and low values will lead to longer, deeper trees (Veness et al., 2011). This appears to be borne out by our results. For very low values of C, the planner doesn't explore alternative plans sufficiently, and easily gets stuck in the local maximum of the instant-gratification policy $\pi_{\rightarrow}$; searching more-or-less naively over the space of plans of length $m \geq 6$, the planner is exponentially unlikely to find the optimal policy $\pi_{\dashrightarrow}$. In contrast, for very high values of C the planner will consider many moderate-sized plans, and will occasionally get lucky and find the optimal policy, but will often miss it; these outcomes are represented by the blue, green, and red curves in Figure 4.17. Finally, for values of C in the 'sweet spot' that balances exploration with exploitation in the planner's simulated action selection, the optimal policy is virtually guaranteed: this situation is represented by the orange,

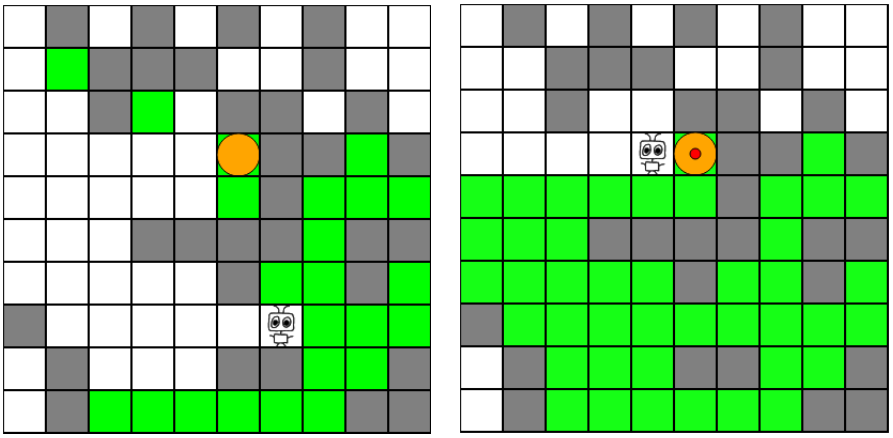

Figure 4.14: Left: AIξ with a uniform prior and finite horizon is not far-sighted enough to explore the beginning of the maze systematically. After exploring *most* of the beginning of the maze, it greedily moves deeper into the maze, where ξ assigns significant value. **Right**: In contrast, the MDL agent systematically visits each tile in lexicographical (row-major) order; we use 'closeness to starting position' as a surrogate for 'simplicity'.

pink, and black curves.

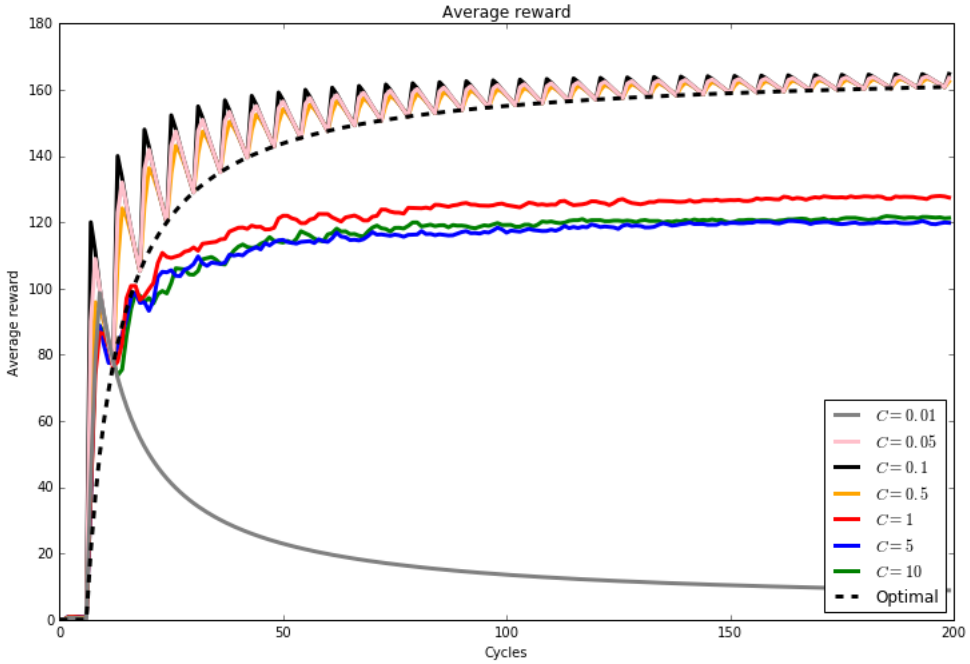

Figure 4.17: AIμ's performance on the chain environment, varying the UCT parameter. Note the 'zig-zag' behavior of the average reward of the optimal policy. These discontinuities are simply caused by the fact that, when on the optimal policy $\pi_{\dashrightarrow}$, the agent receives a large reward every N cycles and 0 reward otherwise. Asymptotically, these jumps will smooth out, and the average reward $\bar{r}_t$ will converge to the dashed curve, $\bar{r}_t^*$.

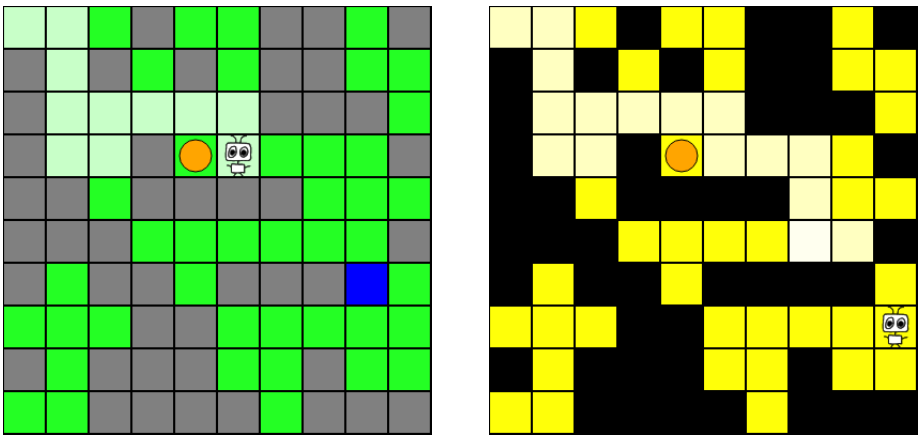

Figure 4.15: Left: AIξ initially explores normally, looking for the dispenser tile. Once it reaches the point above, the blue 'self-modification' tile is now within its planning horizon ($m = 6$), and so it stops looking for the dispenser and makes a bee-line for it. **Right**: After self-modifying, the agent's percepts are all maximally rewarding; we visualize this by representing the gridworld starkly in yellow and black. The agent now loses interest in doing anything useful, as every action is bliss.

Chapter 5

Conclusion

> *The AI does not hate you, nor does it love you, but you are made out of atoms which it can use for something else.*

The next few decades seem to offer much promise for the field of artificial intelligence and machine learning. Of course, it remains to be seen whether or not superintelligent general AI will come about in this time frame, if at all. Regardless of the time scales involved, though, it seems clear that questions relating to formal theories of intelligence and rationality will only grow in importance over time. Hutter's AIXI model and its variants represent some of the first steps along the path towards an understanding of general intelligence. Our ultimate hope is that the software developed in this thesis will grow and serve as a useful research tool, an educational reference, and as a playground for ideas as the field of general reinforcement learning matures. At a minimum, we expect it to be of value to students and researchers trying to learn the fundamentals of GRL. We now provide a short summary of what we have achieved, and provide some reflections and ideas on future directions for AIXIjs.

Summary

In this thesis, we have presented:

- A review of general reinforcement learning, bringing together the various agents due to Hutter, Orseau, Lattimore, Leike, and others, under a single consistent and accessible notation and conceptual set-up.
- The design and open-source implementation of a framework for running and testing these agents, including environments, environment models, and the agents themselves,
- A suite of illuminating experiments in which we realized and compared different approaches to rational behavior, and
- An educational and interactive demo, complete with visualizations and explanations, to assist newcomers to the field.

Future directions

In the course of developing AIXIjs, we have made numerous insights into GRL, and raised several new questions:

- What is a principled way to normalize the first term of Equation (2.20) for the Shannon KSA agent, whose utility function is unbounded from above? Is it possible to change the normalization $\frac{1}{m(\beta-\alpha)}$ adaptively?
- What are some general principles for constructing efficient models for certain classes of environments, in the context of applied Bayesian general reinforcement learning? Constructing bespoke models such as the $\mathcal{M}_{\text{Dirichlet}}$ model is time-consuming and doesn't generalize to new environments. On the other hand, very generic approaches like context-tree weighting learn too slowly to be useful. Is there a middle ground?
- Is there a way to represent the Dirichlet model $\mathcal{M}_{\text{Dirichlet}}$ as a mixture, in the form of Equation (2.10)? This would make it more convenient to run, for example, Thompson sampling.
- Why do the entropy-seeking agents seemingly outperform AIξ at its own game, as in Figure 4.2? This is a confronting result. Is there a bug in the implementation, or just something we don't understand?
- Can we make our JavaScript implementations more efficient, and scale up the demos to more impressive environments? How far can we scale these agents in the browser?
- Planning with ρUCT is often like a black box. Is it possible to construct a good visualization of the state of a Monte Carlo search tree, to illuminate what it is doing?

In addition, there are some low-level 'jobs' that can be done to improve and extend AIXIjs in the near term:

- Construct working visualizations for the bandit, FSMDP, and Iterated prisoner's dilemma environments (not presented here).
- Implement the regularized version of the MDL agent (Leike, 2016a).
- Figure out how to implement optimistic AIXI.
- Implement planning-as-inference algorithms such as Compress and Control (Veness et al., 2015).
- Finish implementing the CTW model class.
- Extend the implementation to include TD-learning agents and DQN.

Working on an open-source project, implementing state-of-the-art models of rationality has been both rewarding and thought-provoking. We're excited to continue to contribute to the AIXIjs project over the coming months, and to see where new ideas in reinforcement learning will take us.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *International Conference on Machine Learning*, pages 1–8, 2004.

Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002. doi: 10.1023/A:1013689704352. URL http://dx.doi.org/10.1023/A:1013689704352.

Peter Auer, Thomas Jaksch, and Ronald Ortner. Near-optimal regret bounds for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 89–96, 2009.

Andrew G. Barto and Thomas G. Dietterich. *Reinforcement Learning and Its Relationship to Supervised Learning*, pages 45–63. John Wiley & Sons, Inc., 2004. ISBN 9780470544785. doi: 10.1002/9780470544785.ch2. URL http://dx.doi.org/10.1002/9780470544785.ch2.

Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. *CoRR*, abs/1606.01868, 2016. URL http://arxiv.org/abs/1606.01868.

Dimitri P Bertsekas and John Tsitsiklis. *Dynamic Programming and Optimal Control*. Athena Scientific, 1995.

Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

Michael Bostock. Chord diagram example, 2016. URL http://bl.ocks.org/mbostock/1046712.

Nick Bostrom. *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press, 2014.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1): 1–43, March 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2186810.

Leon Chen. Keras-js, 2016. https://transcranial.github.io/keras-js/.

Tom Everitt and Marcus Hutter. Avoiding wireheading with value reinforcement learning. In *Artificial General Intelligence*, 2016.

Tom Everitt, Daniel Filan, Mayank Daswani, and Marcus Hutter. Self-modification of policy and utility function in rational agents. In *Artificial General Intelligence*, 2016.

Google. What we learned in Seoul with AlphaGo. https://googleblog.blogspot.com.au/2016/03/what-we-learned-in-seoul-with-alphago.html, March 2016. Accessed: 2016-03-28.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2nd edition, 2009.

Bill Hibbard. Model-based utility functions. *Journal of Artificial General Intelligence*, 3 (1):1–24, 2012.

Stephen J Hoch and George Loewenstein. Time-inconsistent preferences and consumer self-control. *Journal of Consumer Research*, 17(4):492–507, 1991. URL http://EconPapers.repec.org/RePEc:oup:jconrs:v:17:y:1991:i:4:p:492-507.

John Holdren, Ed Felten, Terah Lyons, and Michael Garris. Preparing for the future of artificial intelligence, 2016.

Marcus Hutter. A theory of universal artificial intelligence based on algorithmic complexity. Technical report, IDSIA, 2000. http://arxiv.org/abs/cs.AI/0004001.

Marcus Hutter. Self-optimizing and Pareto-optimal policies in general environments based on Bayes-mixtures. In *Computational Learning Theory*, pages 364–379. Springer, 2002.

Marcus Hutter. A gentle introduction to the universal algorithmic agent AIXI. Technical report, IDSIA, 2003. ftp://ftp.idsia.ch/pub/techrep/IDSIA-01-03.ps.gz.

Marcus Hutter. *Universal Artificial Intelligence*. Springer, 2005.

Marcus Hutter. Open problems in universal induction & intelligence. *Algorithms*, 3(2): 879–906, 2009.

Edwin T Jaynes. *Probability Theory: The Logic of Science*. Cambridge University Press, 2003.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Andrej Karpathy. Reinforcejs, 2015. http://cs.stanford.edu/people/karpathy/reinforcejs/.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-45375-X, 978-3-540-45375-8. doi: 10.1007/11871842_29. URL http://dx.doi.org/10.1007/11871842_29.

Ray Kurzweil. *The Singularity is Near: When Humans Transcend Biology*. Viking Books, 2005.

Guillaume Lample and Devandra Singh Chaplot. Playing FPS games with deep reinforcement learning. *arXiv preprint*, 2016.

Tor Lattimore. *Theory of General Reinforcement Learning*. PhD thesis, Australian National University, 2013.

Tor Lattimore and Marcus Hutter. Asymptotically optimal agents. In *Algorithmic Learning Theory*, pages 368–382. Springer, 2011.

Tor Lattimore and Marcus Hutter. General time consistent discounting. *Theoretical Computer Science*, 519:140–154, 2014a.

Tor Lattimore and Marcus Hutter. Bayesian reinforcement learning with exploration. In *ALT*, pages 170–184. Springer, 2014b.

Tor Lattimore, Marcus Hutter, and Peter Sunehag. The sample-complexity of general reinforcement learning. *CoRR*, abs/1308.4828, 2013. URL http://arxiv.org/abs/1308.4828.

Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time-series. In M. A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*. MIT Press, 1995.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

Shane Legg. *Machine Super Intelligence*. PhD thesis, University of Lugano, 2008.

Jan Leike. *Nonparametric General Reinforcement Learning*. PhD thesis, Australian National University, 2016a.

Jan Leike. Balancing exploration and exploitation in model-based reinforcement learning. 2016b. Under preparation.

Jan Leike and Marcus Hutter. Bad universal priors and notions of optimality. In *Conference on Learning Theory*, pages 1244–1259, 2015.

Jan Leike, Tor Lattimore, Laurent Orseau, and Marcus Hutter. Thompson sampling is asymptotically optimal in general environments. In *Uncertainty in Artificial Intelligence*, 2016.

J. J. Leonard and H. F. Durrant-Whyte. Simultaneous map building and localization for an autonomous mobile robot. In *Intelligent Robots and Systems '91. 'Intelligence for Mechanical Systems, Proceedings IROS '91. IEEE/RSJ International Workshop on*, pages 1442–1447 vol.3, Nov 1991. doi: 10.1109/IROS.1991.174711.

Ming Li and Paul M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Texts in Computer Science. Springer, 3rd edition, 2008.

Maxwell W. Libbrecht and William Stafford Noble. Machine learning applications in genetics and genomics. *Nature Reviews Genetics*, 16(6):321–32, 5 2015.

David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002. ISBN 0521642981.

Vincent C Müller and Nick Bostrom. Future progress in artificial intelligence: A survey of expert opinion. *Fundamental Issues of Artificial Intelligence*, pages 553–571, 2016.

Jarryd Martin, Tom Everitt, and Marcus Hutter. Death and suicide in universal artificial intelligence. In *Artificial General Intelligence*, 2016.

John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon. A proposal for the Dartmouth summer research project on artificial intelligence. 1955.

Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *AI Winter*. Alpha Press, 2009. ISBN 6130079133, 9786130079130.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. Technical report, Google DeepMind, 2013. http://arxiv.org/abs/1312.5602.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

James Moor. The Dartmouth College artificial intelligence conference: The next fifty years. *AI Magazine*, 27(4):87, 2006.

Hans Moravec. *Mind Children: The Future of Robot and Human Intelligence*. Harvard University Press, Cambridge, MA, USA, 1988. ISBN 0-674-57616-0.

Oskar Morgenstern and John von Neumann. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

Stephen M Omohundro. The basic AI drives. In *Artificial General Intelligence*, pages 483–492, 2008.

Laurent Orseau. Optimality issues of universal greedy agents with static priors. In *Algorithmic Learning Theory*, pages 345–359. Springer, 2010.

Laurent Orseau. Universal knowledge-seeking agents. In *Algorithmic Learning Theory*, pages 353–367. Springer, 2011.

Laurent Orseau. Asymptotic non-learnability of universal agents with computable horizon functions. *Theoretical Computer Science*, 473:149–156, 2013.

Laurent Orseau. Universal knowledge-seeking agents. *Theoretical Computer Science*, 519:127–139, 2014.

Laurent Orseau, Tor Lattimore, and Marcus Hutter. Universal knowledge-seeking agents for stochastic environments. In *Algorithmic Learning Theory*, pages 158–172. Springer, 2013.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Clifton Phua, Vincent C. S. Lee, Kate Smith-Miles, and Ross W. Gayler. A comprehensive survey of data mining-based fraud detection research. *CoRR*, abs/1009.6119, 2010. URL http://arxiv.org/abs/1009.6119.

Stuart J Russell and Peter Norvig. *Artificial Intelligence. A Modern Approach*. Prentice Hall, 3rd edition, 2010.

Hasim Sak, Andrew W. Senior, Kanishka Rao, and Françoise Beaufays. Fast and accurate recurrent neural network acoustic models for speech recognition. *CoRR*, abs/1507.06947, 2015. URL http://arxiv.org/abs/1507.06947.

J. Schmidhuber. Curious model-building control systems. In *Proc. Int. J. Conf. Neural Networks*, pages 1458–1463. IEEE Press, 1991.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

David Silver and Joel Veness. Monte-Carlo planning in large POMDPs. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2164–2172. Curran Associates, Inc., 2010. URL http://papers.nips.cc/paper/4031-monte-carlo-planning-in-large-pomdps.pdf.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Daniel Smilkov and Shan Carter. Tensorflow Playground, 2016. http://http://playground.tensorflow.org/.

Malcolm Strens. A Bayesian framework for reinforcement learning. In *International Conference on Machine Learning*, pages 943–950, 2000.

Peter Sunehag and Marcus Hutter. Optimistic AIXI. In *Artificial General Intelligence*, pages 312–321. Springer, 2012.

Peter Sunehag and Marcus Hutter. Rationality, optimism and guarantees in general reinforcement learning. *Journal of Machine Learning Research*, 16:1345–1390, 2015.

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

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 12, pages 1057–1063. MIT Press, 1999.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. URL http://arxiv.org/abs/1512.00567.

William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, pages 285–294, 1933.

S. Thrun. The role of exploration in learning control. In D.A. White and D.A. Sofge, editors, *Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold, Florence, Kentucky 41022, 1992.

Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. Monte carlo localization for mobile robots. In *In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA*, 1999.

A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. WaveNet: A Generative Model for Raw Audio. *ArXiv e-prints*, September 2016.

Joel Veness, Kee Siong Ng, Marcus Hutter, William Uther, and David Silver. A Monte-Carlo AIXI approximation. *Journal of Artificial Intelligence Research*, 40(1):95–142, 2011.

Joel Veness, Marc G Bellemare, Marcus Hutter, Alvin Chua, and Guillaume Desjardins. Compress and control. In *AAAI*, 2015.

Vernor Vinge. The coming technological singularity. *Vision 21: Interdisciplinary Science and Engineering in the Era of Cyberspace*, 1:11–22, 1993. http://www.rohan.sdsu.edu/faculty/vinge/misc/singularity.html.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-sne effectively. http://distill.pub/2016/misread-tsne/, 2016.

Eliezer Yudkowsky. *Rationality: From AI to Zombies*. Amazon, 2015.