Principles of Metalevel Control

by

Nicholas James Hay

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Stuart Russell, Chair
Professor Pieter Abbeel
Professor Rhonda Righter

Fall 2016

Principles of Metalevel Control

Copyright 2016
by
Nicholas James Hay

Abstract

Principles of Metalevel Control

by

Nicholas James Hay

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Stuart Russell, Chair

Intelligent agents perform computations to help them decide how to act. But which computations will best inform these decisions? How long should an agent think before acting? Given the impracticality of performing all possible relevant computations, AI systems need effective strategies for metalevel control—that is, for choosing computations that contribute most to the (object-level) action selection problem.

This thesis presents a theoretical basis for metalevel control problems, situating it within the frameworks of Bayesian selection problems and Markov decision processes. We present some fundamental results concerning the structure of these problems and the nature of their solutions. These results include bounds on the number of computations performed by optimal policies, and an understanding of how the context of multiple actions affects computation choice.

For a more empirical investigation of metalevel control, we apply reinforcement learning techniques to the problem of controlling Monte Carlo tree search. This requires learning a metalevel policy to choose which computation to perform as a function of the internal state of the tree search. To represent an efficient learnable class of policies, we describe how Monte Carlo tree search can be implemented using pointed trees, a recursive data structure that allows efficient evaluation of recursive functions and their derivatives on it by a message-passing algorithm. We propose a concrete class of policies that includes UCT and AlphaGo as special cases, and propose a method for metalevel shaping rewards. We show that by initializing the metalevel policy to UCT, reinforcement learning can find metalevel policies that outperform UCT for smaller number of computations in the domain of computer Hex.

These results show the benefits of a solid theoretical understanding of metalevel control, and the potential for metalevel reinforcement learning to replace hand-crafted algorithm design.

Contents

Contents **i**

List of Figures **iv**

List of Tables **v**

List of Symbols **vi**

1 Introduction **1**
1.1 Metalevel control . . . 2
1.2 Related work . . . 4
1.2.1 Decision-theoretic metareasoning . . . 5
1.2.2 Selection problems . . . 8
1.2.3 Monte Carlo tree search . . . 9
1.3 Contributions . . . 11
1.4 Overview of thesis . . . 12

2 Mathematical background **14**
2.1 Basic notation . . . 14
2.1.1 Sets and sequences . . . 14
2.1.2 Probability theory . . . 15
2.2 Markov decision processes . . . 15
2.3 Factoring and restricting MDPs . . . 17
2.4 Reinforcement learning . . . 19

3 Metalevel control **21**
3.1 Beta-Bernoulli metalevel control problem . . . 22
3.2 Metalevel control problems . . . 25
3.3 Stationary and Markov MCPs . . . 26
3.4 Metalevel MDPs . . . 28
3.5 Factored metalevel MDPs . . . 32
3.6 Deriving the cost of time . . . 35

4 Structure of metalevel policies **38**
4.1 Bounding computation 40
4.1.1 Bounding in expectation 40
4.1.2 Bounding 1-action metalevel MDPs 42
4.1.3 Bounding k-action metalevel MDPs 45
4.2 Context effects 46
4.2.1 No index policies for metalevel decision problems 46
4.2.2 Context is not just a number 48
4.2.3 Context and stopping 52

5 Metalevel reinforcement learning **54**
5.1 Challenges of metalevel reinforcement learning 55
5.2 Pointed trees 56
5.2.1 Definition and recursive construction 56
5.2.2 Recursive functions of pointed trees 58
5.2.3 Local operations 58
5.2.4 Local operations and recursive functions 59
5.2.5 Derivatives of recursive functions 62
5.3 Monte Carlo tree search (MCTS) 63
5.4 MCTS as a metalevel MDP 67
5.5 Metalevel policy class for MCTS 69
5.5.1 UCT and AlphaGo metapolicies 69
5.5.2 Metapolicy class: Recursive component 71
5.5.3 Metapolicy class: Parameterized local component 73
5.6 Metalevel shaping rewards 74

6 Experiments **77**
6.1 Experimental setup 77
6.1.1 Object-level environment: Hex 77
6.1.2 Metalevel environment 79
6.1.3 Calibrating UCT 79
6.1.4 Reinforcement learning 80
6.2 Experimental results 80
6.2.1 Flat architecture 80
6.2.2 Factored architecture 82
6.2.3 Factored architecture initialized to UCT 83
6.2.4 Metalevel reward shaping 84
6.3 Discussion 88

7 Conclusions **89**
7.1 Understanding metalevel control 89
7.1.1 Mechanical model 90

7.1.2 Bayesian model 91
7.1.3 Complementary models 91
7.2 Summary 92
7.3 Future work 93
7.4 Parting thoughts 95

Bibliography **97**

List of Figures

1.1 Two different environments . . . 2
1.2 Hex environment . . . 3
1.3 Metalevel control problem . . . 4

3.1 An agent deciding between actions . . . 22
3.2 The $k = 1$ Beta-Bernoulli metalevel MDP . . . 29
3.3 Illustration of the upper bound of Theorem 3.16 . . . 34

4.1 Structure of the optimal policy of the 1-action Beta-Bernoulli metalevel MDP . 39
4.2 Stopping in the 2-action Beta-Bernoulli metalevel MDP . . . 45
4.3 Traces of the optimal policy for the 3-action Beta-Bernoulli metalevel MDP . . 47
4.4 The value function of a non-indexable metalevel decision problem . . . 49
4.5 Two-action Bernoulli metalevel control problem . . . 50
4.6 Optimal Q-function for the two-action Bernoulli metalevel control problem . . . 51

5.1 Decomposition of a pointed tree . . . 56
5.2 The semantics of message passing over pointed trees . . . 60
5.3 Valid messages . . . 61
5.4 Updating messages after a local operation . . . 62
5.5 A trace of Monte Carlo tree search . . . 65
5.6 Neural network architectures . . . 74

6.1 A game of 3×3 Hex . . . 78
6.2 Learning curve for the flat policy architecture . . . 81
6.3 Learning curve for the factored policy architecture initialized to UCT . . . 83
6.4 Trace of rewards and potential of the maximum estimated action-utility shaping reward estimated by the average reward of rollouts . . . 86
6.5 Trace of rewards and potential of the maximum estimated action-utility shaping reward estimated by the Beta-posterior . . . 87

List of Tables

6.1 Average reward of UCT against itself for varying weight k 80
6.2 Average reward for the flat policy architecture 80
6.3 Average episode length and number of episodes per batch 82
6.4 Average reward for the factored policy architecture 82
6.5 Average reward for the factored policy architecture initialized to UCT 83
6.6 Average reward for the factored policy architecture initialized to UCT, varying test and training number of computations 84
6.7 Average reward for the factored policy architecture initialized to UCT and shaped by maximum estimated action-utility estimated by the average reward of rollouts 86
6.8 Average reward for the factored policy architecture initialized to UCT and shaped by maximum estimated action-utility estimated by the Beta-posterior 87

List of Symbols

$\Diamond$	The unique empty context, page 57
$\lvert h \rvert$	The length of a finite sequence $h \in X^*$ of elements of a set X, page 14
$\perp$	The unique terminal state of an MDP, page 15
π	A policy for an MDP, page 15
A_t	Random variable giving the action taken by policy π in MDP M at time t, page 15
ACT	In a metalevel MDP, stop and take the action of maximum posterior expected utility, page 32
$A(s)$	Set of actions possible in a state $s \in S$ of an MDP, page 15
`Context`	Either the set of all contexts, or one $\texttt{Context}(C, N, i, T_1, \ldots, T_k)$ constructed out of a context C, a node N, an index $i \in \{0, \ldots, k\}$ and $k \geq 0$ trees T_i, page 57
$\texttt{down}(T, i)$	Return the pointed tree T with the point moved down to the ith child of the current point or T if this is impossible, page 59
$\mathbb{E}[\cdot]$	The expectation of a random quantity, page 15
$\mathbb{E}_M^\pi[\cdot]$	The expectation of a random quantity, where the random variables S_t, A_t, N are a realization of MDP M controlled by policy π, page 16
ϵ	The empty sequence, page 14
$h_{i:j}$	The subsequence of $h \in X^*$ from the ith to the jth elements, inclusive of its endpoints, page 14
$\texttt{insert}(T, i, T')$	Return the pointed tree T with T' inserted before the ith child of the current node or T if this is impossible, page 59
M	A Markov decision process (MDP), equal to a tuple (S, A, T, R), page 15

M_u^{const}	The constant metalevel MDP of value $u \in \mathbb{R}$, also denoted simply u, page 32
$M_1 + M_2$	The composition of two MDPs, page 18
$\texttt{make_tree}(N)$	Construct a `Tree` from a `Node` N, page 59
$\texttt{modify}(T, N')$	Return the pointed tree T with the point replaced by N', page 59
$\mu(s)$	The action-utility estimator of a metalevel MDP, mapping a state $s \in \mathcal{S}$ to a k-dimensional real vector $\mu(s) \in \mathbb{R}^k$ giving estimated utilities of all k actions at that state, page 28
μ^*(s)	A coherent upper bound on a metalevel MDP, i.e., such that $\mu^*(s) \in \mathbb{R}$ is an upper bound on $\max_i \mu_i(s)$, and forms a martingale when composed with a realization of the MDP as a Markov chain, page 40
$\mu_i(s)$	The action-utility estimator for the ith action of a metalevel MDP, mapping a state $s \in \mathcal{S}$ to a real value $\mu(s) \in \mathbb{R}$ giving the estimated utility of that action at that state, page 28
N	Random stopping time giving the number of actions taken by policy π in MDP M before termination, page 15
`Context`	The set of nodes, page 57
$\mathbb{P}[\cdot]$	The prior probability of an event, page 15
$\mathbb{P}_M^\pi[\cdot]$	The prior probability of an event, where the random variables S_t, A_t, N are a realization of MDP M controlled by policy π, page 16
$\pi_M^*(s)$	An optimal policy for MDP M, evaluated at state $s \in S$, page 16
$\texttt{point}(T)$	Denotes the `Node` the pointed tree is currently pointing to, page 58
`PointedTree`	Either the set of all pointed trees, or one $\texttt{PointedTree}(C, N, T_1, \ldots, T_k)$ constructed out of a context C, a node N and trees T_i, page 57
Ψ_u^{const}	The constant MCP of value $u \in \mathbb{R}$, page 32
$Q_M^\pi(s, a)$	Q-value function of policy π in the MDP M given initial state $s \in S$ and action $a \in A$, page 16
$Q_M^*(s, a)$	Q-value function of the optimal policy in MDP M given initial state $s \in S$ and action $a \in A$, page 16
$R(s, a, s')$	Reward function of an MDP, giving the reward received upon transitioning from state s by action a to state s', page 15

range(X)	Set of possible values for the random variable X, page 15
S	Set of states in an MDP, page 15
S_t	Random variable giving the state of MDP M under policy π at time t, page 15
$T(s, a, s')$	Transition function of an MDP, giving the probability of transitioning to state $s' \in S \cup \perp$ from state $s \in S$ after action $a \in A(s)$, page 15
`Tree`	Either the set of all trees or one $\texttt{Tree}(N, T_1, \ldots, T_k)$ constructed out of a node N and $k \geq 0$ trees T_i, page 57
$\texttt{up}(T)$	Return the pointed tree T with the point moved up to the parent of the point, or T if this is impossible, page 59
$V_M^\pi(s)$	Value function of policy π in the MDP M given initial state $s \in S$, page 16
$V_M^*(s)$	Value function of the optimal policy in MDP M given initial state $s \in S$, page 16
X^*	The set of finite sequences of elements from a set X, including the empty sequence ϵ, page 14

Acknowledgments

My deepest thanks go to my advisor Stuart Russell for his help in every aspect of this work. His broad perspective on AI and its neighboring disciplines, deep history with metareasoning and ability to clearly and succinctly state the key issues and insights have had a profound influence on me. His exacting scholarly standards and ability to ask the right questions and point me in helpful directions have consistently pushed me to achieve more than I'd thought was possible, even (especially) when I thought I'd done enough.

I'm grateful to my committee members, Pieter Abbeel and Rhonda Righter, for their insights and helpful feedback, and especially for their encouragement during the final dissertation stages.

Many others have guided me through earlier stages of this journey. I'm honored to thank Cris Calude, my supervisor at the University of Auckland, who set me on the path of research, taught me LaTeX and was always ready with sage advice. I'm also grateful for early encouragement and mentorship from James Goodman, who almost tempted me into computer architecture research.

I have Joe Halpern to thank for encouraging me to pursue a PhD in the US while I was a visiting student at Cornell, and for teaching me that apparently obvious foundations always have a variety of compelling alternatives if only you look for them.

Thanks to Eric Horvitz, for taking the time to talk to an eager young graduate student about metareasoning and the future of AI.

Berkeley has served as the ideal setting for my graduate studies. Thanks to my fellow RUGS members for their patience, comments and questions as I slowly came to understand the metalevel: Gregory Lawrence, Jason Wolfe, Shaunak Chatterjee, Nimar Arora, Kevin Canini, Norm Aleks, Rodrigo de Salvo Braz, Erik Sudderth, Emma Brunskill, Siddharth Srivastava, David Moore, Lei Li, Yusuf Erol, Will Cushing, Daniel Duckworth, Dylan Hadfield-Menell, Constantin Berzan and Yi Wu.

Many other Berkeley colleagues have made my graduate studies significantly more enjoyable. Thanks to my Soda Hall AI batchmates: Dave Golland, Mohit Bansal, Jon Barron, Jeremy Maitin-Shepard, Aditi Muralidharan, Ahn Pham and Jie Tang. I have fond memories of adjusting to our new graduate student life together in that windowless basement office. Special thanks to Dave for the Thanksgiving meals and wide-ranging conversations over Thai food; John DeNero, for teaching me how to teach (and introducing me to Samosa); and John Schulman, for all the discussions about brains and AI, future and present.

I've had the great fortune of finishing this chapter of my life while in the supportive and perhaps too stimulating environment of Vicarious. I'm especially appreciative for the seemingly infinite patience of Dileep George, Scott Phoenix, Michael Stark, Charlotte Bowell and Devin Gribbons.

I cannot imagine having gotten through this without the support of friends and family both back home in New Zealand and here in the Bay Area.

I'm thankful for my friends back home, James & Judy Ting-Edwards, Jamie & Emma Robertson, Tess & Jared Mason, Nathan & Catrien (Fick) Kilpatrick, Sam Mockridge. I'm always looking forward to my next Christmas visit, and fondly back on my last.

My internship at Google was life-changing in several respects. Special thanks to Moshe Looks, for his mentorship and for persistently asking "How many pages?"; to Jonni Kanerva, for all the advice and linguistic fun; and to Praveen "Aphorist" Paritosh, for all his questions.

Thanks to Adam Safron, for setting my sights high and always being there; to Gopal Sarma, for keeping me going and reminding me this is only the beginning; to David Andre, for support and advice from one who's been there before; and to Michael Ellsworth & Julia Bernd, for all the late night writing/cheese visits.

I've managed to gain two California families during the last few years. Thanks to the Alvidrezim—Marc & Jill & Zev & Ari & Talia—for providing a home away from home, and to my newest relatives, I-Cheng, Phoebe, Carrie and Sand. I'm looking forward to spending more time with you!

Profound thanks to my family: Mum and Dad (Wendy and David), Jane and Andrew. I leave home to spend a semester overseas, and look what happens? Thanks for your constant (if sometimes teasing) support—I could feel it all the way from across the Pacific! And yes, finally, really this time, I'm not even joking: I'm done!

Finally, Nancy, I'll never thank you for everything you are, for everything you do for me, for everything you mean to me... enough! I couldn't imagine a better partner.

athaa ham-frashwaa thwaa
xrathwaa swanishtaa
aramataish hudaanu warshwa
Consider thus with your own intelligence! With this generous man, choose the holiest things of the Well-regulated Mind!

Zarathustra, 1500 BC

You better think! (Think!) Think about what you're trying to do....

Aretha Franklin, 1968 AD

Chapter 1

Introduction

1.1 Metalevel control **2**
1.2 Related work **4**
1.2.1 Decision-theoretic metareasoning 5
1.2.2 Selection problems 8
1.2.3 Monte Carlo tree search 9
1.3 Contributions **11**
1.4 Overview of thesis **12**

It's a commonplace that you should think before you act, and we sometimes do. Thinking, at least in theory, can help us figure out how to act. We can consider our options, evaluate their consequences and choose what we believe to be the best course of action. Likewise, artificial agents exhibiting this aspect of intelligence perform computations, for the same reasons and in a similar fashion.

How do humans come to be able to think? As with all skills, practice plays a central role. By trying to think, paying attention to the consequences and making unconscious adjustments to improve matters over time, we learn how to think better. We learn that thinking can lead us to new options, or can tell us more about known options; we learn how long to think before acting. In short, we learn how to get better at controlling our own decision-making processes. Can AI systems learn how to compute, and compute better, in the same way?

This thesis explores this question from two angles: one theoretical, the other practical.

To formalize our learning problem, we adopt and extend a theoretical framework in which computation is characterized as a special metalevel action, one that helps an agent (or system) gather information about the value of other possible actions. Different specific computations yield different information; thus computing better, for these purposes, means choosing more informative computations. The agent's goal, in this casting, is to develop

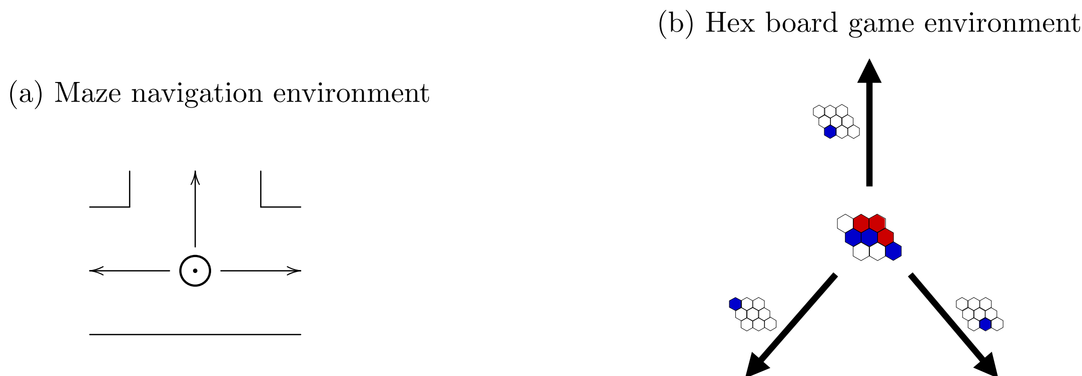


Figure 1.1: Two different environments, each with three possible actions: in the maze, move left, up or right; in the game, place a blue piece in one of the unoccupied (white) locations. In each case, an agent must choose one of these actions and may perform computations in order to determine which one is best.

increasingly effective policies (or strategies) for choosing computations—that is, to learn optimal algorithms for **metalevel control**.

We also approach our learning question in a more practical vein. Since metalevel control problems can be analyzed as a kind of (metalevel) Markov decision problem, we can investigate how techniques for learning can be applied to learn metalevel policies. In particular, we apply reinforcement learning techniques experimentally in the context of AI game-playing, specifically for the game Hex, and find that learned metalevel policies can outperform fixed ones.

Before embarking on these formal and experimental explorations, we provide in this chapter a few guideposts the reader may find useful along the way. We first give an intuitive introduction to these theoretical concepts and their practical application using a simple example (Section 1.1). We then situate the current work with respect to related lines of research (Section 1.2) and the contributions it makes to these (Section 1.3). Finally, we give an overview of the rest of the thesis (Section 1.4).

1.1 Metalevel control

Imagine an agent facing three possible actions in an uncertain environment (Figure 1.1). Each action may lead to a new situation that may be better or worse, perhaps with a different set of options.

If the agent has the computational means to reason about all three actions and explore each of the (expected) consequences, it might gain insights that help it obtain a better outcome. But assuming time and computation may be costly, it must decide whether to spend any computational resources on any of the actions, and if so, how much, what kind and

in what order. In short, besides the original set of three actions in its external environment (the **object level**), it also faces a host of other decisions about those actions (the **metalevel**).

A standard way to formalize the object-level decision, especially for game-playing and other AI problems, is as an **agent-environment interaction** (Russell and Norvig, 2010). The agent takes an observation as input and returns an action as output, while the environment takes an action as input and returns an observation, reward and termination signal. Together they form a dynamic system, exchanging alternating actions and observations, where the objective is to maximize the total reward the agent receives from the environment before the end of the interaction. Choosing the best policy for this search through the state space of possible computations is the **metalevel control problem**.

But in fact, the metalevel decision can be characterized the same way—though with the set of actions expanded appropriately to include metalevel computations. In both cases, the agent needs a policy for which actions to choose. If the state space can be formalized as a tree, then tree-search algorithms correspond to policies for state-space exploration.

Figure 1.2 illustrates the metalevel control problem for tree search, where the object-level actions are moves in a game-playing situation. In this case, computations expand leaves (A, B or C) of the tree to further explore possible future consequences. The choice of which leaf to expand is the choice of which computation to perform. Figure 1.3 shows how the metalevel state of the search procedure continues after expanding leaf B (Figure 1.3a) and then leaf E (Figure 1.3b). Each takes the metalevel system to a new state with new metalevel actions.

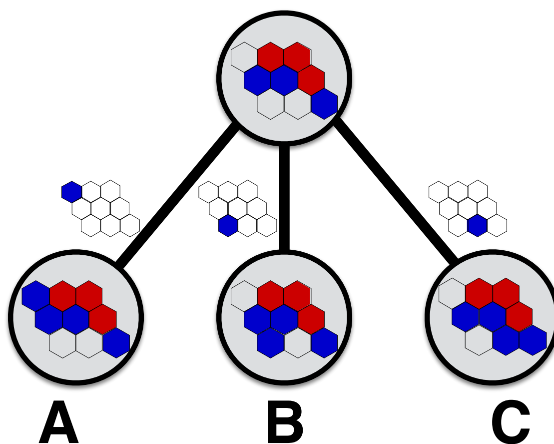
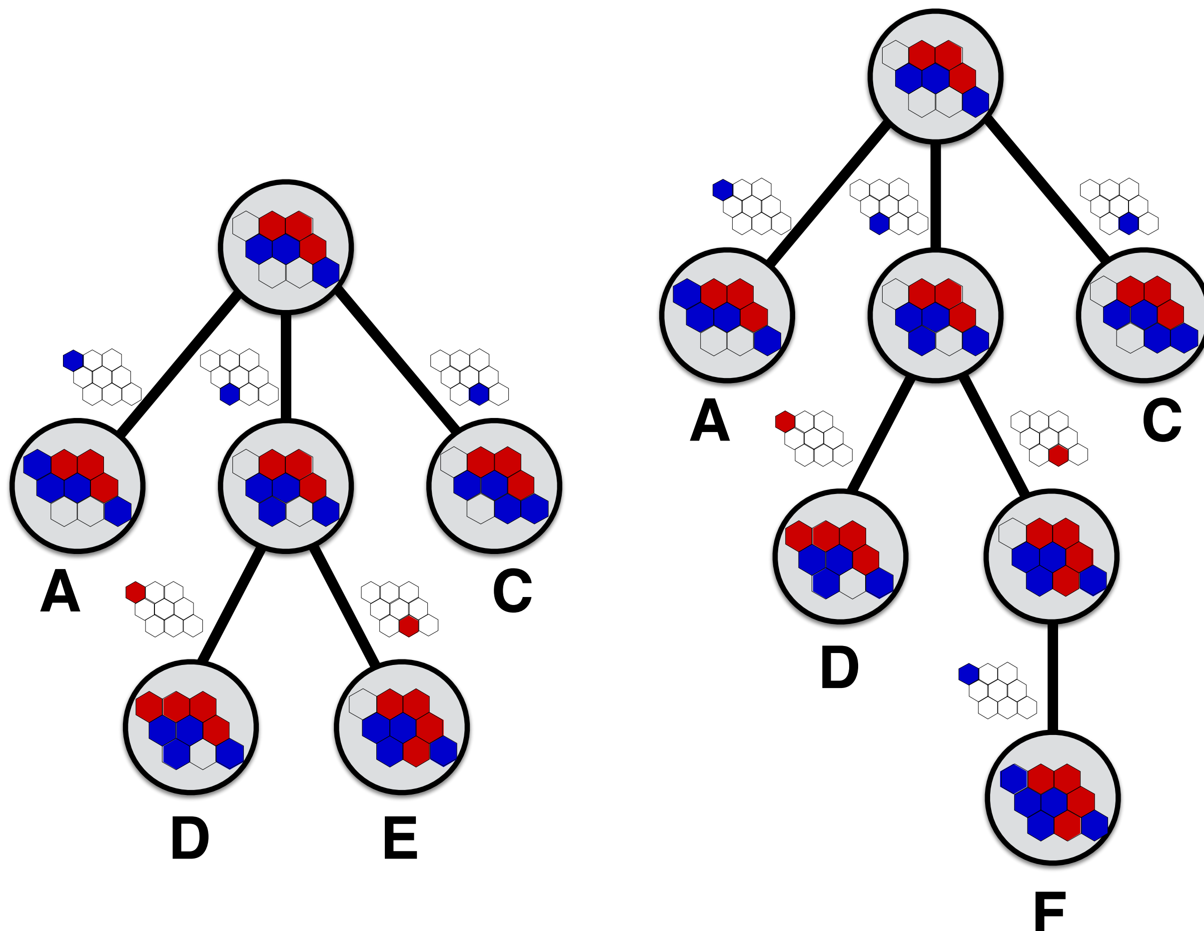


Figure 1.2: Three actions in the game of Hex, which can be explored at A, B or C.

In this thesis I explore and formalize the informal intuition above—that metalevel control problems are direct analogs of object-level decision processes. This yields both theoretical insights and practical benefits. Theoretically, connecting control problems to (metalevel) Markov decision processes allows us to draw on the statistical framework of Bayesian selection problems for richer theoretical analysis. I focus on computations that involve simulating and evaluating possible future action sequences and present some fundamental results concerning the structure of these problems and the nature of their solutions. These results include

Figure 1.3: Metalevel control problem: which computation, if any, to perform next?



(a) After expanding Figure 1.2 at leaf B, the system can now choose between A, C, D and E.

(b) After expanding Figure 1.3a at leaf E, the system can now choose between A, C, D and F.

bounds on the number of computations performed by optimal policies, and an understanding of how the context of multiple actions affects computation choice. Practically, we show how to apply reinforcement learning to a class of metalevel control problems (i.e., Monte Carlo tree search), finding solutions that outperform hand-crafted algorithms (i.e., UCT).

1.2 Related work

Our work relates most closely to three areas of research: decision-theoretic approaches to metareasoning and metalevel control, selection problems and Monte Carlo tree search algorithms. We treat each in turn.

1.2.1 Decision-theoretic metareasoning

Matheson (1968) noted that the metalevel decision problem can be formulated and solved decision-theoretically, borrowing directly from the related concept of **information value theory** (Howard, 1966). Matheson studied the metalevel decision problem in the context of human decision analysis, formulating it decision-theoretically and approximating its solution, observing that computations can be selected according to the expected improvement in decision quality resulting from their execution.

I. J. Good (1968) independently proposed using this idea to control search in chess. Good (1971) later defined "Type II rationality" to refer to agents that maximize expected utility less the cost of the expected time and effort taken to think and do calculations, i.e., agents that optimally solve the metalevel decision problem before acting.

As interest in probabilistic and decision-theoretic approaches in AI grew during the 1980s, several authors explored these ideas further. Horvitz (1987) studied beliefs and actions under resource constraints, presenting classes of approximations and heuristics. Dean and Boddy (1988) introduced, in the context of planning, the notion of **anytime algorithms** (called **flexible algorithms** by Horvitz (1987); see below): algorithms that can be interrupted at any point during computation to return a result whose utility is a function of the computation time. Doyle (1988) formalized and studied metareasoning as rational self-government, where the agent reflects upon itself in order to guide its own reasoning. Fehling and Breese (1988) proposed an architecture for choosing between different problem-solving methods based on their relative costs and benefits, measuring benefits as the value of information provided by the problem-solving method.

The most direct antecedent to this work is that of Russell and Wefald (1988a,b, 1989, 1991a,b), who formulated the **metalevel sequential decision problem**, employing an explicit model of the results of computational actions. In particular, Russell and Wefald (1989) addressed the design of metalevel policies for game tree search, where each computation performed a leaf expansion followed by minimax backup. In order to estimate the value of a computation, they used a probabilistic model of the computation's result, learned from the statistics of a set of training positions. This estimated the value of a computation by the improvement in decision quality after performing that one computation (the single-step assumption), selecting the computation that maximized this (the meta-greedy policy). They applied the resulting algorithm, **MGSS***, to the control of game-playing search in Othello with encouraging results.

More recently, Tolpin and others have applied decision-theoretic metareasoning to define metalevel control policies that optimize analytically derived bounds on the value of computation. Such policies were found to be quite successful at controlling several classes of algorithms: depth-first search in constraint satisfaction problems (Tolpin and Shimony, 2011), Monte Carlo tree search (Hay et al., 2012), A* search (Tolpin et al., 2013) and IDA* search (Tolpin et al., 2014).

Anytime algorithms

Anytime algorithms, called **flexible algorithms** by Horvitz (1987), compute for a variable amount of time before returning a result. A **performance profile** estimates the quality of the algorithm's result as a function of the time given to it. Performance profiles can be used to decide how long to run the algorithm, weighing the improvement in result quality against the cost of time. Russell and Zilberstein (1991) distinguish two kinds of anytime algorithms: **contract algorithms**, which must be given in advance the amount of time they have to compute, and **interruptible algorithms**, which can be interrupted without warning. They show by an elegant doubling construction that a contract algorithm can be converted into an interruptible one that requires at most four times the computation. They study several constructions for forming an anytime algorithm from anytime components, showing how to allocate time among the components and how to determine the performance profile of the composite algorithm from those of its components. Zilberstein and Russell (1996) show that while the problem of allocating computation time optimally to anytime components is in general NP-complete, in certain cases a local compilation approach yields a globally optimal allocation. With online monitoring of the progress of anytime algorithms, time allocation can be continuously adjusted to the actual performance of the algorithm (Horvitz et al., 1989; Horvitz, 1990; Horvitz and Rutledge, 1991; Zilberstein, 1993; Hansen and Zilberstein, 1996). More recent work extends anytime algorithms (Kumar and Zilberstein, 2010) and their monitoring (Carlin and Zilberstein, 2011) to the multiagent setting.

Continual computation (Horvitz, 1997, 2001) extends the anytime setting to the case where a system receives a sequence of problems with interspersed idle time. Horvitz derives optimal policies for using this idle time to perform valuable precomputations in various specific settings, along with presenting a number of applications. Shahaf and Horvitz (2009) further extend this to the setting where problems are composed of subtasks that may be shared, presenting complexity hardness results and polynomial time approximation procedures.

Bounded optimality

Russell (1997, 2014) proposes that achieving the long-term goal of AI, the creation and understanding of intelligence, requires a precise notion of intelligence. Russell considers four possible definitions and finds flaws in all but the last: *perfect rationality*, *calculative rationality*, *metalevel rationality* and *bounded optimality*. **Perfect rationality**, in which agents exhibit perfectly optimal behavior, is impossible in all but trivial environments. **Calculative rationality**, in which agents are perfectly rational given unbounded time to compute, can yield arbitrarily poor behavior given bounded time. **Metalevel rationality** refers to an agent controlling its computations in an optimal manner; it is, sadly, just as impossible as perfect rationality. By contrast, **bounded optimality**, in which agents exhibit the best behavior possible given the limits of the architecture on which the agent is implemented, is suitable as a foundation: it is always possible to achieve; it is theoretically tractable; and

the constrained optimization problem captures a number of intuitions about intelligence.

Simon (1947, 1982) discussed bounded rationality informally, while Russell and Subramanian (1995) put it on a formal footing: an agent is **bounded-optimal** for a given machine architecture in a given environment class if it is of maximal expected utility among all agents implementable on that architecture. A central motivation for metalevel control is as a technique to approximate bounded optimal agent design, by approaching designs that achieve **metalevel bounded optimality**: developing bounded agents whose interaction with the world is not direct, but mediated by controlling the computations of an object level.

Formalizing metalevel control

Russell and Wefald (1991a) used the idea of a **joint-state system**, i.e., one whose state consists of the pair of an environment state and an internal state of the agent, in their formalization of metalevel control. Parr and Russell (1998); Andre and Russell (2002); Marthi et al. (2005) formalized this joint-state formulation using MDPs in the context of **hierarchical reinforcement learning**. This setting is in some sense the opposite of metalevel control: rather than seeing the choice of a single external action as composed of a multitude of metalevel choices (e.g., seeing choosing where to step as composed of many choices of simulations of possible foot trajectories), it composes the single external action itself into a larger plan of behavior (e.g., composing where to step into the larger plan of walking to a destination).

Harada and Russell (1999) consider what is necessary to formalize the value of computation, taking the case of lookahead search in MDPs with application to playing Tetris. They pose this as a reinforcement learning problem in the joint-state MDP, comparing three metalevel controllers: one that uniformly searches to depth 1, one that uniformly searches to depth 2, and one that selects whether to search to depth 1 or 2 in a state-dependent way, learned by reinforcement learning. The third uniformly outperforms the first two no matter how much time is given to the controller. Harada and Russell further argue for the benefits of a formalism capable of handling active interaction between the object level and the metalevel where, for example, the object level can interrupt the metalevel while it is computing if something important arises.

Pearson (2006) illustrates how computation can be viewed as information gathering, arguing that there's no difference between a doctor performing a test on each of a sequence of patients, and the same doctor deducing the test results of a sequence of patients after memorizing their records. Both could be formalized as what we would term a **Beta-Bernoulli metalevel control** problem (see Section 3.1). Following Matheson (1968), this directly relates the value of *information* to the value of *computation*. Pearson applies this perspective to the problem of sampling in influence diagrams, proposing an algorithm that explicitly represents a Dirichlet distribution over the (unknown) distribution of the utility node conditional on actions and observations. This distribution is updated by performing random samples of the influence diagram, which give evidence about this unknown distribution. Pearson proposed several heuristics for selecting computations, including selecting the computation

maximizing the myopic value of information, selecting the computation that has positive k-step value of information for the smallest k, and selecting the computation that is about an action with second-best expected utility. Empirically, Pearson examined only the myopic policy, showing that it outperformed several others on this problem.

1.2.2 Selection problems

Bechhofer (1954) introduced the **ranking and selection problem**, where the agent has a set of alternative actions to decide between, but can repeatedly sample (simulate) their value before making a choice. How should an agent choose samples in order to make the best final choice?

In early applications, the samples were physical information-gathering actions, such as in agriculture, where sampling involves sowing a field with a grain variety, watching the grass grow, then measuring its yield. This was later applied to discrete-event simulation (Swisher et al., 2003), where sampling involves performing a computer simulation of a system. The choice of which simulation to perform is metalevel control in another guise.

Bechhofer (1954) studied the ranking and selection problem in a frequentist setting, finding policies that have good performance on the worst-case probability of selecting the best action.

Decision-theoretic approaches

Raiffa and Schlaifer (1968) first proposed the use of Bayesian decision theory for the ranking and selection problem, studying the problem of maximizing the expected utility of the chosen action given an independent normal prior on the action's utilities.

Most subsequent research has, however, either used the worst-case performance (following Bechhofer) or aimed to maximize the probability of correctness. (See Bechhofer et al. (1995) and Swisher et al. (2003) for reviews.)

We pick up the thread with Frazier and Powell (2007) who consider the setting with k actions whose utilities are distributed according to independent normal priors, and in which n measurements (corrupted by independent normal noise) can be made of the actions' utilities. Frazier and Powell define the **knowledge gradient policy** as that which performs the **myopically optimal** sample at each step, i.e., samples the action that would be optimal to sample if there were only one sample left to perform, which can be computed in closed form in this setting.

Subsequent work extends this basic idea to a number of other settings, deriving and improving upon earlier less principled policies. Frazier and Powell (2008) consider the setting where the prior is a normal-inverse-gamma distribution and there is a cost function $C(n)$ on the number of samples performed allowing early stopping. This gives a principled derivation for a previously proposed policy for choosing samples (LL1 of Chick et al. (2007)) but with a stopping rule that they empirically show to outperform earlier work. Frazier et al. (2009)

consider the case with a correlated normal prior, given an algorithm for computing the knowledge gradient policy exactly in time linear in the number of actions.

Knowledge gradient algorithms are myopic. Chick and Frazier (2012) use a continuous-time approximation to address the non-myopic case. The continuous-time problem can be put into a canonical form by change of variables and solved by numerical methods, yielding two approximate policies ESP_B and ESP_b for the original discrete time problem. ESP_B uses the full numerical solution to decide what to do, while ESP_b approximates the numerical solution by the boundary of its continuation region. In their experiments, ESP_b outperforms a wide selection of prior policies, while their implementation of ESP_B proved too computationally expensive to use in practice.

1.2.3 Monte Carlo tree search

Metalevel control can be applied to algorithms using various kinds of computations. In this thesis we'll focus in particular on those which use **Monte Carlo** methods, in which random sampling is used to estimate the expected value of a distribution based on the average of the samples drawn from it.

Monte Carlo methods for sequential decision problems

There have been several applications of Monte Carlo methods to sequential decision problems, i.e., where the agent will perform not just one but a sequence of actions. In the context of game-tree search, Abramson (1990) introduced the **expected-outcome model** that defines the value of a game position as the expected outcome given random play, proposing that this value be estimated by the Monte Carlo method of simulating random play and averaging its outcome. Bouzy and Helmstetter (2004) used a variant of this method to directly evaluate the value of a move in the context of computer Go, finding it performed well against more knowledge-based approaches.

Separately, Kearns et al. (2002) developed a method, **sparse sampling**, for choosing actions in an MDP given the ability to perform simulations of it. Sparse sampling randomly samples a uniform lookahead tree of possible futures from a given state to a fixed depth, selecting the action based on this tree by alternate maximization and averaging. Kearns et al. show sparse sampling computes near-optimal actions in time independent of the size of the state space but exponential in the lookahead depth.

The straightforward application of Monte Carlo methods to evaluating the quality of actions in the sequential setting has a central weakness: the quality of an action is evaluated assuming future actions are determined randomly, when in fact they will be chosen by the agent. The underlying assumption is that the one decision you are making now is the only sensible one you'll ever make, which yields various biases, e.g., away from reliable strategies that require a sequence of well-chosen actions to work. Sparse sampling explicitly evaluates a tree of future consequences so it doesn't suffer this error. However, it expands the tree uniformly, yielding exponential inefficiency for long time horizons.

UCT

Kocsis and Szepesvári (2006) take a step toward resolving both difficulties with the **UCT** algorithm. UCT operates in the context of **Monte Carlo tree search** (MCTS) algorithms, that progressively build up a lookahead tree from the current state. Computations in MCTS take the form of simulating a randomized sequence of actions leading from a leaf of the current tree to a terminal state. UCT is primarily a method for selecting a leaf from which to conduct the next simulation. UCT chooses a leaf by starting at the root of the tree, which corresponds to the current state, then repeatedly choosing branches until a leaf is reached.

As described in Section 1.1, the choice of which leaves in a search tree to expand is a prototypical example of a metalevel control problem. Any particular strategy for making this choice forms a **metalevel policy**. UCT draws upon the theory of **bandit problems** (Berry and Fristedt, 1985) for its metalevel control policy, using the bandit algorithm UCB1 (Auer et al., 2002) to choose which branch to take within the tree. (See Section 1.2.3 for more on bandit problems.) Kocsis and Szepesvári (2006) show that UCT's estimates of the utility of the best action converges at rate $O(\frac{\log n}{n})$ in the number of simulations n, and that the probability of simulating a suboptimal action at the root converges to zero polynomially.

Variants of UCT form the core for a large family of successful algorithms, with notable successes in computer Go with the programs MoGo (Gelly and Silver, 2011) and its more famous cousin AlphaGo (Silver et al., 2016).

Bandit problems and metalevel control

In bandit problems (Robbins, 1952; Berry and Fristedt, 1985), named after slot machines (multi-armed bandits), the agent faces a set of actions of uncertain utility they can repeatedly take. The object is to find a strategy (also known as a policy) for choosing which action to take based on the results of the actions it has taken so far which maximizes the *total* sum of the utilities of actions it has taken.

Medical trials are a real world example: here the actions are different treatments, and the outcome is whether the patient was cured. Given knowledge of prior treatments and their outcomes, which treatment do we try on the next patient? The one we can learn the most about, to help the patients in the future? Or the one that appears best, to help the patient at hand? These are the two extremes of the **exploration vs. exploitation tradeoff** that any high-value strategy will balance.

Are bandit problems an appropriate model for the metalevel control problem of choosing which branch to explore, as UCT's choice of the bandit algorithm UCB1 may imply? Bubeck et al. (2009) and Hay et al. (2012) observe that it is not: in bandit problems, every trial involves executing a real object-level action with real costs, whereas in the metalevel control problem the trials are simulations whose cost is usually independent of the utility of the action being simulated.

Bubeck et al. (2009) define and study **pure exploration bandits**, where the decision-maker first gets to sample a number of actions, not necessarily known in advance, and is

then asked to choose a single action whose utility it wishes to maximize. This is similar to a selection problem (recall Section 1.2.2 above). Bubeck et al. (2009) give theoretical results comparing UCB to uniformly allocating samples to the actions, showing that for small numbers of samples they are comparable; for moderate numbers of samples, UCB outperforms uniform allocation; but for large numbers of samples, uniform outperforms UCB. The latter result rests on an asymptotic bound showing that *upper* bounds on the **cumulative regret** (i.e., how suboptimal the policy is for the regular bandit problem) give *lower* bounds on the **simple regret** (i.e., how suboptimal the policy is for the pure bandit problem). They give a simple empirical demonstration, noting, however, that for large numbers of samples, the difference in performance between the two policies falls below machine precision, making the difference more theoretical than practical. Audibert et al. (2010) propose and analyze several policies in this setting, although their empirical results do not conclusively find a policy outperforming UCB.

Hay et al. (2012) observe that one consequence of the mismatch between bandit problems and the metalevel control problem is that bandit policies are inappropriately biased away from exploring actions whose current utility estimates are low. Another consequence is the absence of any notion of "stopping" in bandit algorithms, which are designed for infinite sequences of trials. A metalevel policy needs to decide when to stop deliberating and execute a real action. Instead, Hay et al. propose selection problems (Section 1.2.2) form a more appropriate model of metalevel control, particularly in the Monte Carlo case. This is the approach we will adopted here.

1.3 Contributions

The contributions of this thesis are on two fronts.

On the theoretical front, we provide a principled foundation for studying the metalevel control problem. Specifically, we:

- Propose a decision-theoretic formalization of the **metalevel control problem** (Section 3.2).
- Show that under reasonable conditions (Section 3.3) these problems are equivalent to an MDP augmented with the additional structure of an **action-utility estimator**, which we term a **metalevel MDP** (Section 3.4).
- Prove that we can always bound the number of computations by an optimal policy in a metalevel MDP in expectation (Section 4.1.1), but by counter-example (Example 4.4) that in some settings a policy can with some probability perform arbitrarily many computations, even if this ends up outweighing the benefit of the final decision.
- Give settings where we can bound the number of computations (Theorem 4.8), and show that in general we can bound the number of computations performed in a factored

MDP (Definition 2.7) by the sum of bounds on the number of computations performed in its factors (Theorem 4.9).

- Show that metalevel decision problems, unlike bandits (Gittins, 1979), do not in general have index policies (Section 4.2.1) even when the different actions' utilities are independent of each other.

On the practical front, we offer techniques for applying reinforcement learning to the metalevel control problem of controlling Monte Carlo tree search. Specifically, we:

- Show that recursive functions on the functional data structure of **pointed trees** and their derivatives can be efficiently implemented by message passing (Section 5.2), making them a suitable representation for metalevel control policies for tree search in general.
- Show how Monte Carlo tree search algorithms can be defined as a metalevel MDP (Section 3.4) using pointed trees.
- Propose a class of metalevel policies (Section 5.5) that includes UCT and AlphaGo as special cases.
- Propose a metalevel shaping reward (Sections 5.6 and 6.2.4).
- Show that reinforcement learning in the MTCS metalevel MDP using the above recursive policy class can find policies outperforming UCT at least when the bound on the number of allowed computations is small.

An earlier version of the work in Chapters 3 and 4 was published as Hay et al. (2012).

1.4 Overview of thesis

The thesis is organized as follows.

Chapter 2 provides basic notational and mathematical background, including key concepts from probability theory, Markov decision theory and reinforcement learning.

The next two chapters together provide the theoretical foundation for studying metalevel control. Chapter 3 defines the central class of metalevel control problems and metalevel Markov decision processes, relating the two. The analytical tools developed in Chapter 3 are then used in Chapter 4 to give fundamental results about the value of policies and the structure of optimal policies, including whether and how one can bound the amount of computation it is optimal to do (i.e., when to stop thinking), and how performing computations relevant to one action can depend on the context of other actions and what the system knows about them.

Chapters 5 and 6 relate the proposed theoretical models to practical problems. Chapter 5 addresses how reinforcement learning techniques can be used to learn policies for the

metalevel control problem, as exemplified by learning to control Monte Carlo tree search. We show how to pose this problem as a reinforcement learning problem, how to represent a learnable class of tractable policies, and how to apply several techniques for improving learning. We demonstrate this method experimentally (Chapter 6) in the context of AI game-playing, specifically for the game Hex.

We end with a brief summary of contributions and results in Chapter 7.

Chapter 2

Mathematical background

2.1 Basic notation **14**
2.1.1 Sets and sequences 14
2.1.2 Probability theory 15
2.2 Markov decision processes **15**
2.3 Factoring and restricting MDPs **17**
2.4 Reinforcement learning **19**

In this chapter we introduce some theoretical concepts and tools needed in the rest of the thesis. We first provide basic notational conventions for concepts used throughout (Section 2.1). Next, we recall basic definitions and results of Markov decision theory (Section 2.2), following up with some less standard definitions and results for factoring and restricting MDPs (Section 2.3). Finally, Section 2.4 covers reinforcement learning.

2.1 Basic notation

We will make use of standard notation for sets and sequences (Section 2.1.1) and probability theory (Section 2.1.2).

2.1.1 Sets and sequences

Given a set X, let X^* be the set of finite sequences of elements from X. Denote by ϵ the unique empty sequence. Given a sequence $h \in X^*$, let $|h|$ be its length, and note that $|\epsilon| = 0$. If $h = x_1 \dots x_t \in X^*$, for $1 \leq i \leq j \leq t$, let $h_{i:j} = x_i \dots x_j$ denote the subsequence from the ith to the jth elements, inclusive of its endpoints.

2.1.2 Probability theory

We assume knowledge of probability theory (see, e.g., Kallenberg (2006)) and present the following simply to fix notation. A $\mathcal{Y}$-valued random variable X is, formally, a measurable function $X\colon \Omega \to \mathcal{Y}$ from a probability space Ω to a measurable space $\mathcal{Y}$. We assume that all our random variables are defined on a shared probability space. For a random variable X, let $\operatorname{range}(X)$ denote its range as a function (i.e., its set of possible values).

Note, however, that in the following we'll use mostly countable or real-valued random variables, and thus will not usually need the full strength of the measure-theoretic apparatus.

Denote by $\mathbb{P}[\cdot]$ the probability of an event or statement under this shared probability space, and by $\mathbb{E}[X]$ the expectation of a random variable X. Conditional probabilities and expectations are denoted by $\mathbb{P}[\cdot|\cdot]$ and $\mathbb{E}[\cdot|\cdot]$, respectively.

If random variables X and Y are conditionally independent given another Z, we write this $X \perp\!\!\!\perp Y \mid Z$.

2.2 Markov decision processes

Markov decision processes (see, e.g., (Puterman, 1994)) are commonly used to model sequential stochastic decision problems. To fix notation, we present the fundamental definitions and results without proof.

Definition 2.1. A **Markov decision process** (MDP) $M = (S, A, T, R)$ consists of a countable set S of states, a countable set $A(s)$ of actions available in state $s \in S$, a transition distribution $T(s, a, s')$ giving the probability of transitioning to state $s' \in S \cup \perp$ from state $s \in S$ after action $a \in A(s)$, where $\perp$ is the unique terminal state, and a reward function $R(s, a, s')$ giving the (average) reward received upon transitioning from state s by action a to state s'.

Note that we restrict our attention to MDPs that are countable, undiscounted and have a unique terminal state. MDPs in general needn't satisfy any of these conditions, but it's what we'll need in the following.

Definition 2.2. A **policy** π for an MDP $M = (S, A, T, R)$ is a function mapping a state $s \in S$ to an action $\pi(s) \in A(s)$ to perform in that state. A **stochastic policy** π generalizes this, mapping a state $s \in S$ to a probability distribution $\pi(\cdot|s)$ over actions $A(s)$.

An MDP $M = (S, A, T, R)$ with a policy π and an initial state $S_0 \in S$ (which may be a random variable) defines a Markov chain $S_0, S_1, \ldots$ by:

$$\begin{aligned} A_{t+1} &= \pi(S_t), \\ S_{t+1} \mid S_t, A_{t+1} &\sim T(S_t, A_{t+1}, \cdot). \end{aligned} \tag{2.1}$$

In fact, this chain is defined only up to the point that the chain terminates by transitioning to $\perp$. Let $N \in \{0, \ldots, \infty\}$ be the random stopping time defined by

$$N = \min\{t : S_{t+1} = \perp\} \tag{2.2}$$

giving the number of non-terminal actions before termination.

We'll use $\mathbb{P}^{\pi}_{M}[\cdot]$ and $\mathbb{E}^{\pi}_{M}[\cdot]$ for probabilities and expectations whenever the random variables S_t, A_t, N are a realization of an MDP M controlled by policy π (i.e., whenever they are defined and distributed according to Equations 2.1 and 2.2 above).

Definition 2.3. The **value function** V^{π}_{M} for a policy π in an MDP $M = (S, A, T, R)$ gives the expected total reward received when taking actions according to that policy for each possible starting state $s \in S$:

$$V^{\pi}_{M}(s) = \mathbb{E}^{\pi}_{M}\left[\sum_{t=0}^{N} R(S_t, A_t, S_{t+1}) \middle| S_0 = s\right]. \tag{2.3}$$

Similarly, the **Q-function** Q^{π}_{M} gives the total reward received when starting in state $s \in S$ and taking action $a \in A(s)$ then following π:

$$Q^{\pi}_{M}(s, a) = \mathbb{E}^{\pi}_{M}\left[\sum_{t=0}^{N} R(S_t, A_t, S_{t+1}) \middle| S_0 = s, A_0 = a\right]. \tag{2.4}$$

Definition 2.4. An **optimal policy** π^* for an MDP M, if any exist, is one that maximizes the value of every state: $V^{\pi^*}_{M}(s) \geq \sup_{\pi} V^{\pi}_{M}(s)$ for all $s \in S$.

If we define for each $s \in S$:

$$V^{*}_{M}(s) = \sup_{\pi} V^{\pi}_{M}(s),$$
$$Q^{*}_{M}(s, a) = \sup_{\pi} Q^{\pi}_{M}(s, a),$$

then a policy π^* is optimal iff $V^{\pi^*}_{M}(s) = V^{*}_{M}(s)$ for all $s \in S$ and iff $\pi^*(s) \in \operatorname{argmax}_a Q^{*}_{M}(s, a)$.

Theorem 2.5. *Given an MDP M and a policy π, the functions V^π_M, Q^π_M, V^*_M, and Q^*_M can be uniquely defined as fixed points:*

$$V^\pi_M(s) = \sum_{s'} T(s, \pi(s), s')(R(s, \pi(s), s') + V^\pi_M(s'))$$
$$Q^\pi_M(s, a) = \sum_{s'} T(s, a, s')(R(s, a, s') + Q^\pi_M(s', \pi(s')))$$
$$V^*_M(s) = \max_a \sum_{s'} T(s, a, s')(R(s, a, s') + V^*_M(s'))$$
$$Q^*_M(s, a) = \sum_{s'} T(s, a, s')(R(s, a, s') + \max_{a'} Q^*_M(s', a')).$$

Proof. See Puterman (1994). □

2.3 Factoring and restricting MDPs

In Chapters 3 and 4 we'll establish properties of MDPs by establishing properties of their subcomponents. The two kinds of analysis we will use are based on the operations of **factoring** and **restricting** an MDP's state space.

In this thesis, a **factored MDP** (related to but distinct from the factored MDPs of Boutilier et al. (2000)) is an MDP that behaves like two separate MDPs running in parallel with their actions interleaved. To formalize this idea, we'll define the operation of **composing** two MDPs in this interleaved manner. We can then define a factored MDP as one that equals the composition of two other MDPs.

Definition 2.6. The **composition** of MDPs $M_1 = (S_1, A_1, T_1, R_1)$ and $M_2 = (S_2, A_2, T_2, R_2)$ is the MDP $M_1 + M_2 = (S, A, T, R)$ that has:

- States $S = S_1 \times S_2$,
- Actions $A = A_1 \cup A_2$, where we assume A_1 and A_2 are disjoint,
- Transition function:

$$T((s_1, s_2), a, (s'_1, s'_2)) = \begin{cases} T_1(s_1, a, s'_1) & \text{if } a \in A_1 \text{ and } s'_2 = s_2, \\ T_2(s_2, a, s'_2) & \text{if } a \in A_2 \text{ and } s'_1 = s_1, \\ 0 & \text{otherwise, and} \end{cases}$$

- Reward function:

$$R((s_1, s_2), a, (s'_1, s'_2)) = \begin{cases} R_1(s_1, a, s'_1) & \text{if } a \in A_1, \\ R_2(s_2, a, s'_2) & \text{otherwise.} \end{cases}$$

Definition 2.7. A **factored MDP** M is one for which there exists two other MDPs M_1 and M_2 such that M equals[1] their composition $M_1 + M_2$.

Our second way of forming a subcomponent of the MDP is to restrict its state space to a subset of the full state space. We can use this to restrict an MDP to a part of its space that we know something about, prove something there, then lift that result to the original MDP (for an example of this method, see Theorem 3.17 and Corollary 3.18 later, which rely on Lemmas 2.10 and 2.11 below).

However, we cannot restrict the state space to any possible subset and still have an MDP. We can restrict an MDP's state space only to a subset that the MDP's transitions cannot "get out of". Stating this requirement more formally:

Definition 2.8. In an MDP $M = (S, A, T, R)$, a subset of states $\bar{S} \subseteq S$ is **closed under transitions** if whenever $s \in \bar{S}$, $a \in A(s)$, $s' \in S$, and $T(s, a, s') > 0$, we have $s' \in \bar{S}$.

We can then define the **restriction** of an MDP to such a subset:

Definition 2.9. Let $M = (S, A, T, R)$ be an MDP and $\bar{S} \subseteq S$ be a subset of M's states closed under transitions. Then the **restriction of** M **onto** $\bar{S}$ is the MDP $M|_{\bar{S}} = (\bar{S}, A|_{\bar{S}}, T|_{\bar{S}}, R|_{\bar{S}})$ where:

- $A|_{\bar{S}}$ is the action function $A(s)$ restricted to $\bar{S}$.
- $T|_{\bar{S}}(s, a, s') = T(s, a, s')$ for $s, s' \in \bar{S}$ and $a \in A(s)$. Note that $\bar{S}$ being closed under transitions is exactly the condition required for $T|_{\bar{S}}$ to be normalized.
- $R|_{\bar{S}}(s, a, s') = R(s, a, s')$ for $s, s' \in \bar{S}$ and $a \in A(s)$.

The restricted MDP is closely related to the original MDP. In particular, its value function is the restriction of the original MDP's value function:

Lemma 2.10. *Let $M = (S, A, T, R)$ be an MDP and $\bar{S} \subseteq S$ be a subset of M's states closed under transitions. Then for any $s \in \bar{S}$ and $a \in A(s)$: the optimal value function and optimal Q-functions of $M|_{\bar{S}}$ satisfy*

$$V^*_{M|_{\bar{S}}}(s) = V^*_M(s) \tag{2.5}$$

$$Q^*_{M|_{\bar{S}}}(s, a) = Q^*_M(s, a). \tag{2.6}$$

Proof. Let $(V^*_M)|_{\bar{S}}$ be the value function $V^*_M \colon S \to \mathbb{R}$ with domain restricted to the set $\bar{S}$, and $(Q^*_M)|_{\bar{S} \times A}$ be the Q-function $Q^*_M \colon S \times A \to \mathbb{R}$ with domain restricted to the set $\bar{S} \times A$, where $S \times A$ and $\bar{S} \times A$ denote

$$S \times A = \{(s, a) : s \in S \text{ and } a \in A(s)\}$$
$$\bar{S} \times A = \{(s, a) : s \in \bar{S} \text{ and } a \in A(s)\}.$$

[1]Properly speaking, we don't really want to require that M *equals* $M_1 + M_2$, but that they be equivalent in some appropriate sense. As interesting as this line of thought gets (Lawvere and Schanuel, 2009), for our purposes we can use simple equality without any difficulties.

Then, observe that $(V^*_M)|_{\bar{S}}$ and $(Q^*_M)|_{\bar{S}\times A}$ satisfy the Bellman equations for $M|_{\bar{S}}$. □

Further, the optimal policy of a restricted MDP is the restriction of an optimal policy of the original MDP:

Lemma 2.11. *If π^*_M is an optimal policy for the MDP M, then $(\pi^*_M)|_{\bar{S}}$ is an optimal policy for $M|_{\bar{S}}$. Conversely, if $\pi^*_{M|_{\bar{S}}}$ is an optimal policy for $M|_{\bar{S}}$, then there exists an optimal policy π^* of M such that $\pi^*_{M|_{\bar{S}}} = \pi^*|_{\bar{S}}$. Diagrammatically, the following commutes:*

$$\begin{array}{ccc} M & \longrightarrow & M|_{\bar{S}} \\ \downarrow & & \downarrow \\ \pi^*_M & \longrightarrow & (\pi^*_M)|_{\bar{S}} = \pi^*_{M|_{\bar{S}}} \end{array}$$

Proof. Follows from Equation 2.6 of Lemma 2.10. □

2.4 Reinforcement learning

Reinforcement learning (Sutton and Barto, 1998; Szepesvári, 2010) is the problem of learning what to do in an environment in order to maximize a numerical reward signal. This problem can be formalized as an MDP, and reinforcement learning techniques can be seen as optimization methods for finding policies of high value.

Policy optimization methods are a class of reinforcement learning techniques that seek to directly optimize the policy. Let $M = (S, A, T, R)$ be an MDP, S_0 be a random variable giving an initial state of M, and $\pi_\theta(a|s)$ for $\theta \in \mathbb{R}^k$ be a parameterized class of stochastic policies. Let

$$\eta(\theta) = \mathbb{E}\, V^{\pi_\theta}(S_0)$$

be the expected return of the policy π_θ as a function of θ. Policy optimization methods seek to optimize $\eta(\theta)$. This is non-trivial since $\eta(\theta)$ is highly nonlinear in θ and can be estimated only empirically.

Trust-region policy optimization (TRPO) (Schulman et al., 2015) is a recent policy optimization method that has been demonstrated to be robust: little hyperparameter search is needed to adapt it to different domains. It is derived by making several approximations to an approach that is guaranteed to increase $\eta(\theta)$ monotonically iteration by iteration. It is a batch method that collects trajectories produced by the current policy π_{θ_0} and uses them to estimate an approximate lower bound of $\eta(\theta)$. It optimizes this lower bound robustly using a trust-region method that caps the allowed average KL divergence between the probabilities assigned to the trajectories by the old policy π_{θ_0}, and the probabilities assigned by the new policy π_θ. This ensures that the new policy is not wildly dissimilar to the old policy.

Generalized advantage estimate (GAE) (Schulman et al., 2016) is a complementary method that fits a value function $V_\phi(s)$ to the empirical returns of the policy, using this to

reduce the variance in the estimate of the bound on $\eta(\theta)$. This is closely related to the idea of reward shaping (Dorigo and Colombetti, 1994; Ng et al., 1999).

We use TRPO combined with GAE in our experiments in Chapter 6, and refer the reader to the original publications for the details of these algorithms.

Having establishing this background, we now turn to our theoretical investigations.

Chapter 3

Metalevel control

3.1	Beta-Bernoulli metalevel control problem	**22**
3.2	Metalevel control problems	**25**
3.3	Stationary and Markov MCPs	**26**
3.4	Metalevel MDPs	**28**
3.5	Factored metalevel MDPs	**32**
3.6	Deriving the cost of time	**35**

In this chapter we formalize the problem of controlling computation by introducing several new theoretical constructs, starting with the class of **metalevel control problems** (MCPs). We illustrate these first by example in Section 3.1 and define them generally in Section 3.2. These formalize the problem of controlling an agent's computations from a Bayesian perspective. To study policies for controlling computation, it will prove helpful to define an equivalent MDP to which we can apply established MDP results. This motivates defining **stationary** and **Markov** MCPs, since stationary Markov MCPs are exactly those that have an equivalent MDP (Section 3.3). These equivalent MDPs fall into the broader class of **metalevel MDPs**, defined in Section 3.4; later results, particularly in Chapter 4, apply to this larger class of problems, although with a richer interpretation for those metalevel MDPs converted from MCPs.

We end by exploring **factored metalevel MDP**s (Section 3.5): metalevel MDPs that are also equivalent to factored MDPs (Section 2.3)—that is, metalevel MDPs that are composed of other metalevel MDPs. One important special case is that of a metalevel MDP composed with an action of constant known utility. We give results on how the value function transforms as a function of the known utility action, and when in a composition of two metalevel MDPs it is optimal for a policy to compute in only one of the component metalevel MDPs. These general analytical results are applied in Chapter 4 to study the structure of optimal metalevel policies.

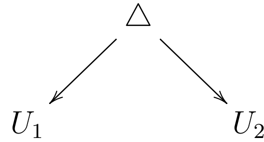


Figure 3.1: The agent is deciding between two actions, the ith action being of unknown utility $U_i \in \{0, 1\}$.

3.1 Beta-Bernoulli metalevel control problem

Consider again the scenario described in Section 1.1, of the agent deciding between k different actions of uncertain utility. In decision-theoretic terms, a good Bayesian agent would do its best to to maximize expected utility: it dutifully represents the utility of each action $i = 1, \ldots, k$ by a random variable U_i, and expresses its uncertainty in its prior. Given no further information, the optimal choice is to simply maximize the prior expected utility: $i^* = \operatorname{argmax}_i \mathbb{E}\, U_i$. But what if the agent has another option: to perform additional computations that may better inform its choice? If, however, these computations aren't free, the agent now faces a new problem: it needs a good policy for choosing which computations to perform and when to stop and act. How does it determine which policy to adopt? Or, in other words, how should the agent decide *what* to think about before acting?

To explore this question, we'll start with a simple fundamental case. Suppose the agent is deciding between two actions (Figure 3.1). Each action $i = 1, 2$ will either succeed or fail, with an unknown probability Θ_i of success. Success is of utility 1, failure of utility 0. The agent is unsure about the probability of success Θ_i, having a uniform prior over it. Thus the agent's prior over the utilities of its actions is for $i = 1, 2$:

$$\begin{aligned} \Theta_i &\sim \text{Beta}(1, 1) \\ U_i \mid \Theta_i &\sim \text{Bernoulli}(\Theta_i) \end{aligned} \tag{3.1}$$

where we recall that Beta$(1, 1)$ is the uniform distribution over $[0, 1]$.

In order to decide which action to select, the agent can choose between two possible computations: it can perform a **Monte Carlo** simulation (Section 1.2.3) of the first or the second action. Each simulation of action i succeeds or fails with its corresponding action's probability Θ_i, each simulation being independent of the others conditional on Θ_i. That the simulations succeed with the same probability as their corresponding real action represents the belief that these are simulations of that action. Denote by $O_{t,i} \in \{0, 1\}$ the result of the Monte Carlo simulation of action i if it is performed at time t, i.e., after t computations of

either action have been performed.[1] Then for $i = 1, 2$:

$$O_{t,i} \,|\, \Theta_i \overset{\text{iid}}{\sim} \text{Bernoulli}(\Theta_i) \qquad \text{for } t \geq 0. \tag{3.2}$$

Suppose the agent has performed a sequence of such computations. Naming the computation of simulating the ith action by the number i, the set of all computations is $\mathcal{C} = \{1, 2\}$. Any particular history of computations h performed is a finite sequence of 1's and 2's, i.e., $h \in \mathcal{C}^*$. Having performed these computations means the agent has observed their corresponding outcomes. In particular, if $h = h_1 \dots h_T \in \mathcal{C}^*$ is the history of performed computations, then the agent has observed O_{t,h_t} for $t = 1, \dots, T$.

How do the results of these computations inform the agent's beliefs about the utility U_i of action i? That is, what is the posterior distribution over U_i given the computational results? Noting that U_i is conditionally independent of $O_{t,i}$ given Θ_i and recalling that the Beta and Bernoulli distributions are conjugate, we have for $i = 1, 2$:

$$\Theta_i \,|\, O_{1,h_1}, \dots, O_{T,h_T} \sim \text{Beta}\left(1 + \sum_{1 \leq t \leq T : h_t = i} O_{t,i}, \quad 1 + \sum_{1 \leq t \leq T : h_t = i} (1 - O_{t,i})\right) \tag{3.3}$$

and thus:

$$U_i \,|\, O_{1,h_1}, \dots, O_{T,h_T} \sim \text{Bernoulli}\left(\frac{1 + \sum_{1 \leq t \leq T : h_t = i} O_{t,i}}{2 + \#\{1 \leq t \leq T : h_t = i\}}\right). \tag{3.4}$$

Now, Equation 3.1 gives the agent's prior belief about the actions' probability of success Θ_i and utility U_i before computation, Equation 3.2 specifies how the agent believes its computations are generated, and Equations 3.3 and 3.4 gives the agent's posterior belief about the actions' probability of success Θ_i and utility U_i after computation.

We can now ask again: how should the agent decide *what* to think about before acting? In other words: what policy for choosing computations will yield the greatest utility?

In general, a policy is a function mapping a state to an action. A computational policy should map the agent's informational state (what it knows) to its decision of what to compute next (or whether it should stop computing and act instead).

What does the agent know after performing computations $h = h_1 \dots h_T \in \mathcal{C}^*$? In one sense, it knows O_{t,h_t} for $t = 1, \dots, T$. But Equations 3.1–3.4 show that this is more than it needs to remember: the posterior over Θ_i determines the distribution over the action's utilities and all future computational outcomes, and this posterior is determined by the pair

[1] We need multiple copies of the random variable containing the outcome of a stochastic simulation of an action because different simulations of the same action will yield different results. Later expressions are simplified by indexing by the total number of computations that have been performed so far rather than by, for instance, the total number of times that particular computation has been performed.

of outcome counts (plus 1):[2]

$$\left(1+\sum_{1\le t\le T:h_t=i} O_{t,i},\quad 1+\sum_{1\le t\le T:h_t=i}(1-O_{t,i})\right). \tag{3.5}$$

Denote by $\tilde{S}_h$ the vector of both pairs (for $i=1$ and $i=2$), with the pair corresponding to action i denoted $(\tilde{S}_h[\alpha_i], \tilde{S}_h[\beta_i])$.[3]

$\tilde{S}_h$ is the random variable giving the agent's full internal state after performing a sequence of computations $h \in \mathcal{C}^*$. A computational policy can now be fully determined by a function $\pi\colon \mathbb{N}^{2\times 2} \to \{1, 2, \textsc{act}_1, \textsc{act}_2\}$, where after performing a sequence of computations $h \in \mathcal{C}^*$, the value of $\pi(\tilde{S}_h)$ determines what the agent does next:

1. If $\pi(\tilde{S}_h) = 1$, it simulates action 1;
2. If $\pi(\tilde{S}_h) = 2$, it simulates action 2;
3. If $\pi(\tilde{S}_h) = \textsc{act}_1$, it stops and takes action 1; and
4. If $\pi(\tilde{S}_h) = \textsc{act}_2$, it stops and takes action 2.

Finally, what is the value of such a policy? We suppose computations have a uniform cost $d > 0$.[4] The value of the policy is the expected utility of the action it eventually takes, less the cost of the computations it performs before doing so. To formalize this, we need to define the trace of the agent's internal state under a policy π:

$$\begin{aligned} H_0^\pi &= \epsilon \\ H_{t+1}^\pi &= H_t^\pi \cdot \pi(S_t^\pi) \\ S_t^\pi &= \tilde{S}_{H_t} \\ N^\pi &= \min\{t : \pi(S_t^\pi) \in \{\textsc{act}_1, \ldots, \textsc{act}_k\}\} \end{aligned} \tag{3.6}$$

where N^π gives the time the agent acts, H_t^π for $0 \le t \le N^\pi$ gives the sequence of the first t computations performed, and S_t^π for $0 \le t \le N^\pi$ gives the agent's internal state at that point. Thus, the value of a policy π is:

$$V^\pi = \mathbb{E}[-d\, N^\pi + U_{\pi(S_N^\pi)}]. \tag{3.7}$$

The above defines the **Beta-Bernoulli metalevel control problem**, whose optimal solution is a policy π^* that maximizes Equation 3.7.

Having come to this point, we can ask:

[2] In Equations 3.3 and 3.5 we directly store the parameters of the posterior Beta distribution over Θ_i, rather than storing the counts of simulated successes and failures. This choice simplifies later expressions. Since the prior is Beta(1, 1), the parameters in the initial state before any computations all have value 1, which requires that we add one to the counts.

[3] This notation is used because (α, β) is the conventional notation for a Beta distribution's parameters.

[4] This assumption of a fixed cost of computation is a simplification; precise conditions for its validity are given by Harada (1997).

1. How generally applicable is the above chain of reasoning? What information is sufficient to specify this problem? Section 3.2 addresses these questions, retracing the above steps more generally and more formally.
2. What can we say about the optimal policy π^* for this problem? Can we find it? Sections 3.3–3.5 and Chapter 4 explore these issues.

3.2 Metalevel control problems

We can now formally define the more general problem exemplified by the Beta-Bernoulli metalevel control problem introduced in Section 3.1.

Definition 3.1. A **metalevel control problem** (MCP) $\Psi = (\mathcal{S}, \mathcal{C}, \{S_h\}_{h \in \mathcal{C}^*}, \{U_i\}_{i=1,\ldots,k}, d)$ consists of:

- a set $\mathcal{S}$ of metalevel states;
- a set $\mathcal{C}$ of possible computations, where $\mathcal{C}^*$ denotes the set of all finite sequences of such computations, and $\epsilon \in \mathcal{C}^*$ denotes the empty sequence consisting of no computations;
- for any finite sequence of computations $h \in \mathcal{C}^*$, an $\mathcal{S}$-valued random variable $\tilde{S}_h$ giving the (uncertain) metalevel state after performing that sequence of computations;
- for each of k possible actions, an $\mathbb{R}$-valued random variable U_i representing its utility; and
- a uniform cost $d \geq 0$ of performing a computation.

Example 3.2. The k-action **Beta-Bernoulli metalevel control problem** with computation cost $d > 0$, as described in Section 3.1, is a metalevel control problem with:

- metalevel states $s \in \mathcal{S} = \mathbb{N}^{2k}$ consisting of a k-dimensional vector of integer pairs, the ith pair of which, denoted $(s[\alpha_i], s[\beta_i])$, holds the number of simulated successes (plus 1; see footnote 2) and the number of simulated failures (plus 1), respectively;
- a set of computations $\mathcal{C} = \{1, \ldots, k\}$, computation $i \in \mathcal{C}$ performing independent well-calibrated stochastic simulations of action i's outcome;
- $\tilde{S}_h$ as defined in Equation 3.5;
- U_i for $i = 1, \ldots, k$ as defined in Equation 3.1; and
- computation cost $d > 0$.

We now define the policy and value function of an MCP.

Definition 3.3. A **policy** π for an MCP $\Psi = (\mathcal{S}, \mathcal{C}, \{S_h\}_{h\in\mathcal{C}^*}, \{U_i\}_{i=1,\ldots,k}, d)$ is a function $\pi\colon \mathcal{S} \to \mathcal{C} \cup \{\textsc{act}_1, \ldots, \textsc{act}_k\}$ mapping states to either a computation $c \in \mathcal{C}$ or the decision $\textsc{act}_i$ to stop and perform the ith action.

Definition 3.4. The **value** V_Ψ^π of a policy π for an MCP $\Psi = (\mathcal{S}, \mathcal{C}, \{S_h\}_{h\in\mathcal{C}^*}, \{U_i\}_{i=1,\ldots,k}, d)$ is given by

$$V^\pi = \mathbb{E}[-d\, N^\pi + U_{\pi(S_N^\pi)}], \tag{3.8}$$

a function of the trace of the policy π in this MCP:

$$\begin{aligned} H_0^\pi &= \epsilon \\ H_{t+1}^\pi &= H_t^\pi \cdot \pi(S_t^\pi) \\ S_t^\pi &= \tilde{S}_{H_t} \\ N^\pi &= \min\{t : \pi(S_t^\pi) \in \{\textsc{act}_1, \ldots, \textsc{act}_k\}\}. \end{aligned}$$

We have now laid the basic theoretical foundation for MCPs and their associated policies and policy values. But which of these policies are of high value, and how can we find them? What can we say about their structure?

To approach these questions, we will convert MCPs into equivalent MDPs, since MDPs are more amenable to certain kinds of analysis. This construction works only for MCPs satisfying specific properties that we define in Section 3.3.

We will see that both the original MCP and its MDP conversion are important. While the MDP allows certain forms of analysis, the MCP defines the semantics of the MDP, giving the underlying random variables $\hat{S}_h$ and U_i that generate the MDP's transitions and rewards. This equivalence will prove important later, particularly in Theorem 4.3.

3.3 Stationary and Markov MCPs

This section defines the two properties of MCPs necessary and sufficient for there being an equivalent MDP.

A **stationary MCP** is one for which the particular sequence of computations it took to reach a metalevel state is irrelevant. Specifically:

Definition 3.5. A **stationary** MCP $(\mathcal{S}, \mathcal{C}, \{\tilde{S}_h\}_{h\in\mathcal{C}^*}, \{U_i\}_{i=1,\ldots,k}, d)$ is one for which for all $s \in \mathcal{S}$ for any $h, h' \in \mathcal{C}^*$ such that $\mathbb{P}[\tilde{S}_h = s] > 0$ and $\mathbb{P}[\tilde{S}_{h'} = s] > 0$, we have

$$\begin{aligned} \mathbb{P}[\tilde{S}_{hc} = s' \mid \tilde{S}_h = s] &= \mathbb{P}[\tilde{S}_{h'c} = s' \mid \tilde{S}_{h'} = s] && \text{for all } c \in \mathcal{C},\ s' \in \mathcal{S}, \\ \mathbb{E}[U_i \mid \tilde{S}_h = s] &= \mathbb{E}[U_i \mid \tilde{S}_{h'} = s] && \text{for all } i = 1, \ldots, k. \end{aligned}$$

To avoid mentioning the irrelevant h, we'll use the notation $\mathbb{E}[U_i \mid \tilde{S} = s]$ where appropriate.

A **Markov MCP** is one whose execution trace forms a Markov chain. Formally:

Definition 3.6. A **Markov** MCP $(\mathcal{S}, \mathcal{C}, \{S_h\}_{h \in \mathcal{C}^*}, \{U_i\}_{i=1,\ldots,k}, d)$ is one in which the distribution of the successor state $\tilde{S}_{hc}$ is conditionally independent of the earlier states $\tilde{S}_\epsilon, \ldots, \tilde{S}_{h_{1:t-1}}$ given the immediately preceding state S_h:

$$\tilde{S}_{hc} \perp\!\!\!\perp \tilde{S}_\epsilon, \ldots, \tilde{S}_{h_{1:t-1}} \mid \tilde{S}_h \tag{3.9}$$

and similarly for the expectation of the utility variables U_i:

$$\mathbb{E}[U_i \mid \tilde{S}_\epsilon, \ldots, \tilde{S}_h] = \mathbb{E}[U_i \mid \tilde{S}_h]. \tag{3.10}$$

Example 3.7. The Beta-Bernoulli MCP (Example 3.2) is both stationary and Markov. To see this, first observe for any $h \in \mathcal{C}^*$ the posterior distribution of Θ_i given the computational trajectory $\tilde{S}_\epsilon, \ldots, \tilde{S}_h$:

$$\begin{aligned}
\Theta_i \mid \tilde{S}_\epsilon, \ldots, \tilde{S}_h &\overset{d}{=} \Theta_i \mid O_{1,h_1}, \ldots, O_{|h|,h_{|h|}} \\
&\sim \text{Beta}(1 + \sum_{t:h_t=i} O_{t,h_t}, 1 + \sum_{t:h_t=i} (1 - O_{t,h_t})) \\
&= \text{Beta}(\tilde{S}_h[\alpha_i], \tilde{S}_h[\beta_i])
\end{aligned} \tag{3.11}$$

with each Θ_i being independent in the posterior, the first equality holding since both sets are deterministic functions of each other, the second by conjugacy of Beta and Bernoulli distributions, and the last by definition of $\tilde{S}_h$. Since U_i, $O_{1:|h|,i}$, and $O_{t+1,i}$ are all conditionally independent of each other given Θ_i, we have by the above that

$$O_{t+1,i} \mid \tilde{S}_\epsilon, \ldots, \tilde{S}_h \sim \text{Bernoulli}(\tilde{S}_h[\alpha_i]/(\tilde{S}_h[\alpha_i] + \tilde{S}_h[\beta_i])) \tag{3.12}$$

$$\mathbb{E}[U_i \mid \tilde{S}_\epsilon, \ldots, \tilde{S}_h] = \tilde{S}_h[\alpha_i]/(\tilde{S}_h[\alpha_i] + \tilde{S}_h[\beta_i]). \tag{3.13}$$

Next note that:

$$\tilde{S}_{hc}[\alpha_i] = \begin{cases} \tilde{S}_h[\alpha_i] + O_{|h|,c} & \text{if } i = c, \\ \tilde{S}_h[\alpha_i] & \text{otherwise.} \end{cases}$$

$$\tilde{S}_{hc}[\beta_i] = \begin{cases} \tilde{S}_h[\beta_i] + (1 - O_{|h|,c}) & \text{if } i = c, \\ \tilde{S}_h[\beta_i] & \text{otherwise.} \end{cases}$$

Letting $s[\alpha_i \leftarrow \alpha_i + 1]$ denote $s \in \mathbb{R}^{2k}$ with the α_i component incremented and similarly for $s[\beta_i \leftarrow \beta_i + 1]$, it follows from the above and Equation 3.12 that:

$$\tilde{S}_{hc} \mid \tilde{S}_\epsilon, \ldots, \tilde{S}_h \sim \begin{cases} \tilde{S}_h[\alpha_i \leftarrow \alpha_i + 1] & \text{with probability } \tilde{S}_h[\alpha_i]/(\tilde{S}_h[\alpha_i] + \tilde{S}_h[\beta_i]), \\ \tilde{S}_h[\beta_i \leftarrow \beta_i + 1] & \text{with probability } \tilde{S}_h[\beta_i]/(\tilde{S}_h[\alpha_i] + \tilde{S}_h[\beta_i]). \end{cases} \tag{3.14}$$

Stationarity follows because Equations 3.14 and 3.13 are functions only of the value of $\tilde{S}_h$ not of h. Markovness follows because Equations 3.14 and 3.13 are functions only of $\tilde{S}_h$ not $\tilde{S}_\epsilon, \ldots, \tilde{S}_{h_{1:|h|-1}}$.

3.4 Metalevel MDPs

We can now formally define **metalevel Markov decision processes** (MMDPs). These extend the familiar class of MDPs (Section 2.2) with extra machinery needed to formalize the metalevel environment. In particular, an MMDP must be able to distinguish metalevel actions (that is, computation) from object-level actions (that is, physical actions), and to evaluate the object-level actions with an **action-utility estimator**. We will later state an equivalence between certain subclasses of MMPDs and MCPs.

Definition 3.8. A **metalevel MDP (MMDP)** $M = (\mathcal{S}, \mathcal{A}, T, \mu, d, R)$ is an undiscounted MDP $(\mathcal{S}, \mathcal{A}, T, R)$ combined with an **action-utility estimator** $\mu\colon S \to \mathbb{R}^k$, the ith component of which at state $s \in S$ gives an estimate $\mu_i(s) \in \mathbb{R}$ of the utility of the ith action, and a cost of computation $d > 0$, whose action set $A(s)$ decomposes into computations $c \in \mathcal{C}(s)$ and physical actions $\text{ACT}_1, \ldots, \text{ACT}_k$:

$$A(s) = \mathcal{C}(s) \cup \{\text{ACT}_i\}_{i=1,\ldots,k},$$

where physical actions transition to the unique terminal state $\perp \notin \mathcal{S}$, and whose reward function is of the form:

$$\begin{aligned} R(s, c, s') &= -d, && \text{for } s, s' \in S \text{ and } c \in \mathcal{C}, \\ R(s, \text{ACT}_i, \perp) &= \mu_i(s), && \text{for } s \in S,\ i = 1, \ldots, k. \end{aligned}$$

Note that μ, d are sufficient to reconstruct R.

Definition 3.9. The **metalevel MDP of a stationary Markov metalevel control problem** $(\mathcal{S}, \mathcal{C}, \{S_h\}_{h \in \mathcal{C}^*}, \{U_i\}_{i=1,\ldots,k}, d)$ is the tuple $(\mathcal{S}, \mathcal{A}, T, \mu, d, R)$ where:

- the states $\mathcal{S}$ of the MDP are those of the MCP, as is the computational cost d,
- action set $A(s) = \mathcal{C} \cup \{\text{ACT}_i\}_{i=1,\ldots,k}$,
- transition function

  $$T(s, c, s') = \mathbb{P}[S_{hc} = s' \mid S_h = s]$$

  for any h such that $\mathbb{P}[\tilde{S}_h = s] > 0$,
- action-utility estimator

  $$\mu_i(s) = \mathbb{E}[U_i \mid S_h = s]$$

  for any h such that $\mathbb{P}[\tilde{S}_h = s] > 0$,
- R is defined from μ and d as in Definition 3.8.

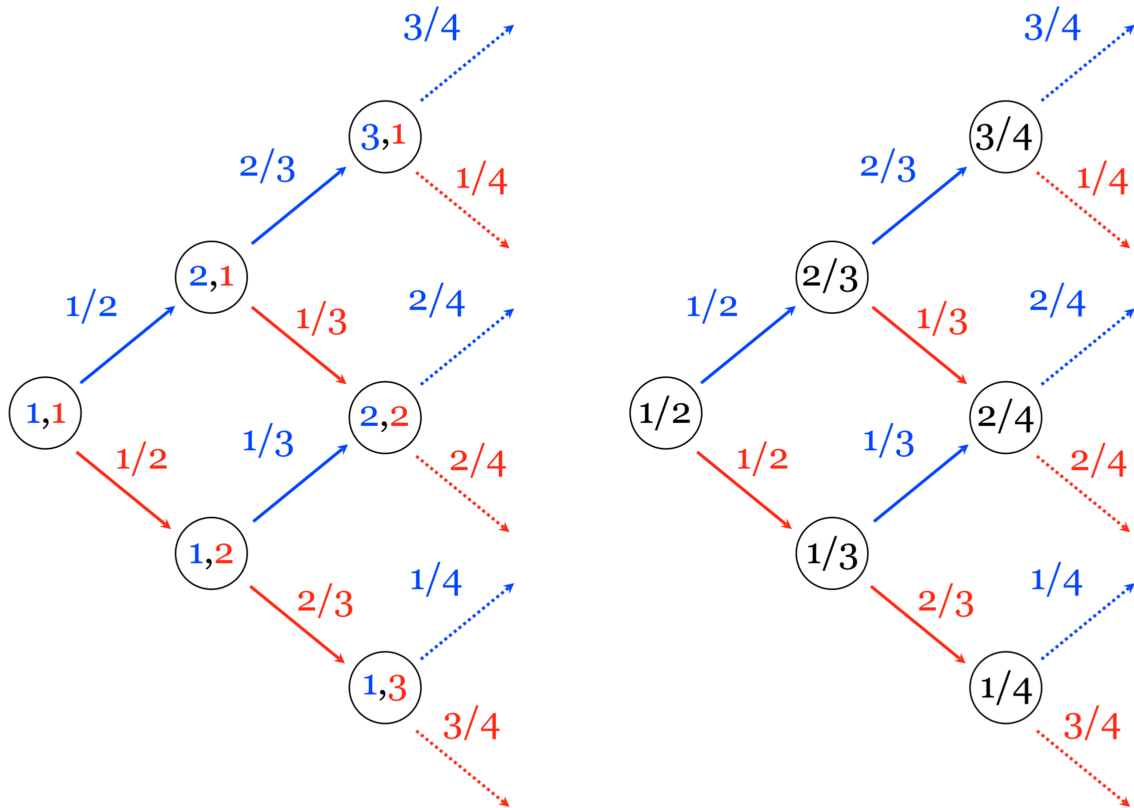


Figure 3.2: Illustration of the $k = 1$ Beta-Bernoulli metalevel MDP. Blue means success, red means failure. Each node corresponds to a state, and edges are labeled by the probability of transitioning along that edge after performing a simulation of the unknown action (the only computation). On the left, nodes are labeled with their corresponding state s, the first component being $s[\alpha_1]$ and the second $s[\beta_1]$. On the right, nodes are labeled with $\mu_1(s)$, the posterior expected utility of action 1 in that state.

Example 3.10. The **metalevel MDP of the k-action Beta-Bernoulli MCP** is $M = (\mathcal{S}, \mathcal{A}, \mu, d, T, R)$ where:

- The states $\mathcal{S}$ (see Figure 3.2) of the MDP are those of the MCP, namely vectors $s \in \mathbb{N}^{2k}$ of k pairs of natural numbers, the ith of which, denoted $(s[\alpha_i], s[\beta_i])$, holds the number of observed successes (plus 1) and the number of observed failures (plus 1), respectively. Equivalently, these form the parameters of the $\text{Beta}(s[\alpha_i], s[\beta_i])$ posterior over the utility U_i of action i.

- Action set $A(s) = \{1, \ldots, k\} \cup \{\textsc{act}_i\}_{i=1,\ldots,k}$.

- Transition function (see Figure 3.2) for $c \in \{1, \ldots, k\}$ follows from Equation 3.14:

$$T(s, i, s') = \begin{cases} s[\alpha_i]/(s[\alpha_i] + s[\beta_i]) & \text{if } s' = s[\alpha_i \leftarrow \alpha_i + 1], \\ s[\beta_i]/(s[\alpha_i] + s[\beta_i]) & \text{if } s' = s[\beta_i \leftarrow \beta_i + 1], \\ 0 & \text{otherwise.} \end{cases}$$

- Action-utility estimator (see Figure 3.2) follows from Equation 3.13:

$$\mu_i(s) = s[\alpha_i]/(s[\alpha_i] + s[\beta_i]).$$

- R is defined from μ and d as in Definition 3.8.

The following theorem shows that optimal policies for the MMDP are optimal policies for the MCP. This means we can solve the MCP using MDP methods.

Theorem 3.11. *Let Ψ be a stationary Markov MCP and M_Ψ its corresponding MMDP. Then:*

$$V^\pi_\Psi = \mathbb{E}\, V^\pi_{M_\Psi}(\tilde{S}_\epsilon). \tag{3.15}$$

Proof. Recall Definition 3.4 of the value of the policy π in MCP Ψ:

$$\begin{aligned} V^\pi &= \mathbb{E}[-d\, N^\pi + U_{\pi(S^\pi_N)}] \\ H^\pi_0 &= \epsilon \\ H^\pi_{t+1} &= H^\pi_t \cdot \pi(S^\pi_t) \\ S^\pi_t &= \tilde{S}_{H_t}. \end{aligned}$$

First, observe that since the MCP is Markov, S^π_t forms a Markov chain. By Definition 3.9, the transition dynamics of this chain are exactly those of M_Ψ. Next, recalling stationarity of the MCP and Definition 3.9, we have $\mu_i(S^\pi_N) = \mathbb{E}[U_i \mid S^\pi_N]$. Then by Definition 2.3 and

Definition 3.9:

$$
\begin{aligned}
\mathbb{E}\, V^{\pi}_{M_\Psi}(\tilde{S}_\epsilon) &= \mathbb{E}\left[\mathbb{E}\left[\sum_{t=0}^{N^\pi} R_{M_\Psi}(S^\pi_t, \pi(S^\pi_t), S^\pi_{t+1}) \mid S^\pi_0 = \tilde{S}_\epsilon\right]\right] \\
&= \mathbb{E}\left[\sum_{t=0}^{N^\pi - 1} R_{M_\Psi}(S^\pi_t, \pi(S^\pi_t), S^\pi_{t+1})\right] \\
&= \mathbb{E}\left[-d\, N^\pi + \mu_{\pi(S^\pi_N)}(S^\pi_N)\right] \\
&= \mathbb{E}\left[-d\, N^\pi + \mathbb{E}[U_{\pi(S^\pi_N)} \mid S^\pi_N]\right] \\
&= \mathbb{E}\left[-d\, N^\pi + U_{\pi(S^\pi_N)}\right] \\
&= V^\pi. \qquad \square
\end{aligned}
$$

The next two results simplify our analysis in the following. Lemma 3.12 shows that when optimal policies stop to act, they always take the action of maximum posterior expected utility. Lemma 3.13 expresses the value of a metalevel MDP as the expected utility of the action chosen on stopping, less the expected cost of computation.

Lemma 3.12. *Given a metalevel policy π for a metalevel MDP M, the policy*

$$
\pi'(s) = \begin{cases} \text{ACT}_{\operatorname{argmax}_i \mu_i(s)} & \text{if } \pi(s) = \text{ACT}_j \text{ for some } j, \\ \pi(s) & \text{otherwise,} \end{cases}
$$

is never worse than π: $V^\pi_M(s) \le V^{\pi'}_M(s)$ for all $s \in S$.

Proof. Intuitively, π' behaves the same as π except that whenever π takes a terminal action, π' takes the best terminal action. Formally, let S_{ACT} be the set of states where $\pi(s) \in \{\text{ACT}_i\}_{i=1,\ldots,k}$. Let $v\colon S \to \mathbb{R}$ be an arbitrary function on states such that $v(\perp) = 0$ and observe that:

$$
\begin{aligned}
(T^\pi_M v)(s) &= \sum_{s' \in S_{\text{ACT}}} T(s'|s, \pi(s)(R(s,a,s') + v(s')) + \sum_{s' \in S \setminus S_{\text{ACT}}} T(s'|s, \pi(s))(R(s,a,s') + v(s')) \\
&= \sum_{s' \in S_{\text{ACT}}} T(s'|s, \pi(s))\mu_{\pi(s)}(s) + \sum_{s' \in S \setminus S_{\text{ACT}}} T(s'|s, \pi(s))(R(s,a,s') + v(s')) \qquad (3.16) \\
&\le \sum_{s' \in S_{\text{ACT}}} T(s'|s, \pi(s))\mu_{\pi'(s)}(s) + \sum_{s' \in S \setminus S_{\text{ACT}}} T(s'|s, \pi'(s))(R(s,a,s') + v(s')) \\
&= \sum_{s' \in S_{\text{ACT}}} T(s'|s, \pi'(s))(R(s,a,s') + v(s')) + \sum_{s' \in S \setminus S_{\text{ACT}}} T(s'|s, \pi'(s))(R(s,a,s') + v(s')) \\
&= (T^{\pi'}_M v)(s),
\end{aligned}
$$

where Equation 3.16 holds because $\mu_{\pi(s)}(s) \le \max_i \mu_i(s) = \mu_{\pi'(s)}(s)$ for all $s \in S_{\text{ACT}}$ and $\pi(s) = \pi'(s)$ for all $s \in S \setminus S_{\text{ACT}}$. Thus, $V^\pi_M(s) \le V^{\pi'}(s)$ for all $s \in S$. $\square$

Given the above, we'll assume for all policies that whenever $\pi(s) = \textsc{act}_i$ for some $i = 1, \ldots, k$, then $i = \text{argmax}_i\, \mu_i(s)$, breaking ties to the action with smaller index. In this case we'll use the shorthand $\pi(s) = \textsc{act}$ to denote this action.[5]

Lemma 3.13. *For any metalevel MDP $M = (S, A, T, R, \mu, d)$ and policy π:*

$$V_M^\pi(s) = \mathbb{E}_M^\pi[-d\,N + \max_i \mu_i(S_N) | S_0 = s]. \tag{3.17}$$

Proof. Recalling Definition 2.3 of the value function, we have:

$$\begin{aligned}
V_M^\pi(s) &= \mathbb{E}_M^\pi \left[\sum_{t=0}^{N} R(S_t, A_t, S_{t+1}) \middle| S_0 = s \right] \\
&= \mathbb{E}_M^\pi \left[\left(\sum_{t=0}^{N-1} -d \right) + R(S_N, A_N, S_{N+1}) \middle| S_0 = s \right] \\
&= \mathbb{E}_M^\pi \left[-d\,N + \max_i \mu_i(S_N) \middle| S_0 = s \right],
\end{aligned}$$

where for the chain to terminate we must have $A_N = \textsc{act}$ and so $R(S_N, A_N, S_{N+1}) = \max_i \mu_i(S_N)$. □

3.5 Factored metalevel MDPs

Finally, we consider metalevel MDPs that have compositional structure, which we call **factored metalevel MDPs**. We first study the simple case of an MMDP M composed with a *constant MMDP* u, defined below (Definition 3.14), studying in particular the properties of the value function of their composition $M + u$. These results are central to the study of context in Section 4.2. We then apply this to characterizing when it's optimal to compute in only one metalevel MDP of a composition $M_1 + M_2$. This result is used in Section 4.1 to bound the number of computations.

Definition 3.14. The **constant metalevel control problem** is $(\{\star\}, \emptyset, \{S_\epsilon\}, \{u\}, 0)$, denoted by Ψ_u^{const}. It has a unique state $\star$, no computations, a single-state random variable $S_\epsilon = \star$ that is constant, one action of constant utility u and no cost of computation. The **constant metalevel MDP** M_u^{const} **of value** $u \in \mathbb{R}$ is the one corresponding to this MCP. Where it causes no confusion we'll denote M_u^{const} simply by u.

We will commonly form the composition $M + u$ of a metalevel MDP M and a constant metalevel MDP u. This metalevel MDP has states $(s, \star)$ for $s \in \mathcal{S}$, which we'll denote by s for simplicity.[6]

[5]One might ask why we defined the metalevel MDP to have seemingly redundant actions $\textsc{act}_1, \ldots, \textsc{act}_k$ when the single action $\textsc{act}$ seems to suffice. The main reason is that our approach simplifies the treatment of factored metalevel MDPs; see Section 3.5.

[6]This is justified by an equivalence of MDPs relying on the isomorphism $\mathcal{S} \times \{\star\} \cong \mathcal{S}$.

Theorem 3.15. *Given a metalevel MDP M, the value function $V^\pi_{M+u}(s)$ of $M+u$ is for any s non-decreasing and convex in $u \in \mathbb{R}$. Further, the subdifferential at $u_0 \in \mathbb{R}$, i.e., the set of all $c \in \mathbb{R}$ that are the gradient of a linear lower bound*

$$c(u - u_0) \leq V^\pi_{M+u}(s) - V^\pi_{M+u_0}(s) \quad \textit{for all } u,$$

is

$$\left[\mathbb{P}^\pi_M\left(\max_i \mu_i(S_N) < u_0 \,|\, S_0 = s\right), \mathbb{P}^\pi_M\left(\max_i \mu_i(S_N) \leq u_0 \,|\, S_0 = s\right)\right] \subseteq [0,1].$$

Proof. From Equation 3.17:

$$\begin{aligned} V^\pi_{M+u}(s) &= \mathbb{E}^\pi_{M+u}[-d\,N + \max(u, \max_i \mu_i(S_N)) \,|\, S_0 = s] \\ &= \mathbb{E}^\pi_M[-d\,N + \max(u, \max_i \mu_i(S_N)) \,|\, S_0 = s] \\ &= \mathbb{E}^\pi_M[-d\,N \,|\, S_0 = s] + \mathbb{E}^\pi_M[\max(u, \max_i \mu_i(S_N)) \,|\, S_0 = s], \end{aligned}$$

where the second equality is justified by the fact that the distributions of N and S_N under a policy π in $M+u$ equal those in M. Observing that $\max(u, \max_i \mu_i(S_N))$ is non-decreasing and convex in u, and that these properties are closed under expectation, we see that $V^\pi_{M+u}(s)$ is non-decreasing and convex in u.

For the subdifferential it suffices to compute (see Rockafellar (1979)):

$$\begin{aligned} \lim_{u \uparrow u_0} \frac{V^\pi_{M+u}(s) - V^\pi_{M+u_0}(s)}{u - u_0} &= \lim_{u \uparrow u_0} \mathbb{E}^\pi_M\left[\frac{\max(u, \max_i \mu_i(S_N)) - \max(u_0, \max_i \mu_i(S_N))}{u - u_0}\middle| S_0 = s\right] \\ &= \mathbb{E}^\pi_M\left[\lim_{u \uparrow u_0} \frac{\max(u, \max_i \mu_i(S_N)) - \max(u_0, \max_i \mu_i(S_N))}{u - u_0}\middle| S_0 = s\right] \\ &= \mathbb{E}^\pi_M[1(\max_i \mu_i(S_N) < u_0) | S_0 = s] \\ &= \mathbb{P}^\pi_M[\max_i \mu_i(S_N) < u_0 | S_0 = s] \\ \lim_{u \downarrow u_0} \frac{V^\pi_{M+u}(s) - V^\pi_{M+u_0}(s)}{u - u_0} &= \mathbb{P}^\pi_M[\max_i \mu_i(S_N) \leq u_0 | S_0 = s], \end{aligned}$$

where the first equality holds by linearity of expectation, and the second by the monotone convergence theorem (Kallenberg, 2006). □

Theorem 3.16. *Given a metalevel MDP M, for any integrable random variable U and any real $u_0 \in \mathbb{R}$, we have:*

$$\mathbb{E}[V^\pi_{M+U}(s)] \leq V^\pi_{M+u_0}(s) + \mathbb{E}\max(U - u_0, 0). \tag{3.18}$$

Proof. Recall that $V^\pi_{M+u}(s)$ is convex in u with every subgradient within $[0,1]$. For any $u, u_0 \in \mathbb{R}$, the following uppper bound follows (illustrated in Figure 3.3):

$$V^\pi_{M+u}(s) \leq V^\pi_{M+u_0}(s) + \max(u - u_0, 0) \tag{3.19}$$

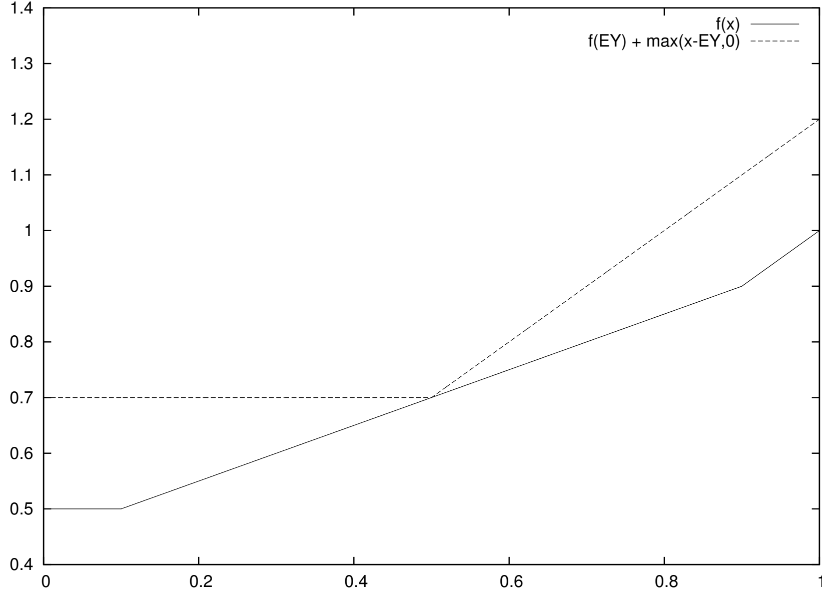


Figure 3.3: Illustration of the upper bound in Equation 3.19 of Theorem 3.16.

since for $u = u_0$ both sides are equal, for $u < u_0$ since there is no subgradient below 0 (i.e., $V^{\pi}_{M+u}(s)$ is non-increasing in u), and for $u > u_0$ since there is no subgradient above 1. Substituting $u = U$ then taking expectations gives the result. □

Theorem 3.17. *Let M_1 and M_2 be two metalevel MDPs. If it's optimal to* ACT *in all states s^1 of $M_1 + u$ for all $u \in \mathbb{R}$, then for any M_2 it's optimal not to perform M_1's computations in state (s^1, s^2) of $M_1 + M_2$ for any $s^2 \in \mathcal{S}_2$.*

Proof. First define $\mu^x_{\max}(s^1) = \max_i \mu^x_i(s^1)$ for $x = 1, 2$. It suffices to show for all $s^2 \in \mathcal{S}_2$ that

$$V^*_{M_1+M_2}(s^1, s^2) = V^*_{\mu^1_{\max}(s^1)+M_2}(s^2), \tag{3.20}$$

where the left-hand side is the optimal value function of $M_1 + M_2$ and the right-hand side is the optimal value function of $\mu^1_{\max}(s^1) + M_2$, since optimal policies for the latter will then be optimal policies for the former. We'll establish Equation 3.20 by showing that the right-hand side satisfies the Bellman equation for the optimal policy of $M_1 + M_2$, which by Theorem 2.5 establishes the equality.

In particular, if we establish

$$V^*_{\mu^1_{\max}(s^1)+M_2}(s^2) = \max\left(\mu^1_{\max}(s^1), \mu^2_{\max}(s^2), \max_{c_2 \in \mathcal{A}_2(s_2)} \mathbb{E}V^*_{\mu^1_{\max}(s^1)+M_2}(S^2_{c_2}) - d\right) \tag{3.21}$$

$$V^*_{\mu^1_{\max}(s^1)+M_2}(s^2) \geq \max_{c_1 \in \mathcal{A}_1(s_1)} \mathbb{E}[V^*_{\mu^1_{\max}(S^1_{c_1})+M_2}(s^2)] - d, \tag{3.22}$$

then by applying these (in)equalities in turn we show that $V^*_{\mu^1_{\max}(s^1)+M_2}(s^2)$ satisfies the Bellman equation for $M_1 + M_2$:

$$\begin{aligned}
&\max\left(\mu^1_{\max}(s^1), \mu^2_{\max}(s^2), \max_{c_1 \in \mathcal{A}_1(s_1)} \mathbb{E}V^*_{\mu^1_{\max}(S^1_{c_1})+M_2}(s^2) - d, \max_{c_2 \in \mathcal{A}_2(s_2)} \mathbb{E}V^*_{\mu^1_{\max}(s^1)+M_2}(S^2_{c_2}) - d\right) \\
&= \max\left(V^*_{\mu^1_{\max}(s^1)+M_2}(s^2), \max_{c_1 \in \mathcal{A}_1(s_1)} \mathbb{E}V^*_{\mu^1_{\max}(S^1_{c_1})+M_2}(s^2) - d\right) \\
&= V^*_{\mu^1_{\max}(s^1)+M_2}(s^2).
\end{aligned}$$

Equation 3.21 is simply the Bellman equation for $\mu^1_{\max}(s^1) + M_2$. For Equation 3.22, let $S^1_{c_1} \sim T(\cdot|s^1, c_1)$ be a random variable distributed according to the state reached by starting in state s^1 and performing computation $c_1 \in \mathcal{C}_1$. Observe that because it is optimal to stop in $M_1 + u$ for any u, it holds in particular for $u = \mu^1_{\max}(s^1)$ and so:

$$\begin{aligned}
-d + \mathbb{E}\max(\mu^1_{\max}(S^1_{c_1}), \mu^1_{\max}(s^1)) &= Q^*_{M_1+\mu^1_{\max}(s^1)}(s_1), c_1) \\
&\leq Q^*_{M_1+\mu^1_{\max}(s^1)}(s_1), \text{ACT}) \\
&= \mu^1_{\max}(s^1).
\end{aligned}$$

Combining this result with Theorem 3.16, we see that:

$$\begin{aligned}
\mathbb{E}[V^*_{\mu^1_{\max}(S^1_{c_1})+M_2}(s^2)] &\leq V^*_{\mu^1_{\max}(s^1)+M_2}(s^2) + \mathbb{E}\max(\mu^1_{\max}(S^1_{c_1}) - \mu^1_{\max}(s^1), 0) \\
&\leq V^*_{\mu^1_{\max}(s^1)+M_2}(s^2) + d.
\end{aligned}$$

Rearranging and maximizing over $c_1 \in \mathcal{C}_1$ we get Equation 3.22 and thus our result.

□

Corollary 3.18. *Let M_1 and M_2 be two metalevel MDPs, and let $\bar{\mathcal{S}}^1 \subseteq \mathcal{S}^1$ be a set of states of M_1 that is closed under transitions of M_1. If it's optimal to* ACT *in all states $s^1 \in \bar{\mathcal{S}}^1$ of $M_1 + u$ for all $u \in \mathbb{R}$, then for any M_2 it's optimal not to perform M_1's computations in states $\bar{\mathcal{S}}^1 \times \mathcal{S}^2$.*

Proof. This follows by applying Lemma 2.11 to Theorem 3.17. □

Before continuing to Chapter 4, we first revisit an assumption we made in passing when defining the metalevel control problem: that computations incur a fixed cost. How strong an assumption is this? When does it apply? Can we derive it from something more reasonable?

3.6 Deriving the cost of time

A central effect of performing a computation rather than acting is time delay: what we would have done now, we'll now do later. In some cases delay has no effect, up to a point: Strictly

speaking, it doesn't matter how close you get to a deadline so long as you don't cross it. In other cases, the effect of time is arbitarily complex: wait long enough and the situation you face will change completely.

Russell and Wefald (1991a, pp. 67–68) observe that if there is a well-defined **cost of time**, i.e., if the effect that time delay has on the utility of an action is a constant cost independent of the action, then the choice of the best action is unchanged over time. This means that although delay incurs a cost, it doesn't change the nature of the object-level decision the agent is making, and so the agent doesn't have to change what it is computing about. We show below that the converse holds: if the effect of a delay doesn't change the object-level decision problem, and if the effect of a delay doesn't itself change over time, then there is a well-defined cost of time.

Let T be a set of times, O be a set of outcomes, and $U\colon O \times T \to \mathbb{R}$ be a utility function of an outcome occurring at a given time. Fix a time $t \in T$ and consider the object-level decision problem of choosing between actions at that time. An action is characterized by a probability distribution $p_a\colon O \to \mathbb{R}$ that it induces over outcomes, and the best action is the one that maximizes the expectation of U under p_a. Von Neumann and Morgenstern (1944) show that a utility function is uniquely characterized, up to a translation and positive scaling, by the ordering it induces over such distributions p_a (termed **lotteries**).

If the object-level decision problem is unchanged by time, then the ordering induced over distributions p_a will be unchanged, and there exist functions $\alpha\colon T \to \mathbb{R}^+$ and $\beta\colon T \to \mathbb{R}$ such that:

$$U(o,t) = \alpha(t)\, U(o) + \beta(t), \tag{3.23}$$

where $U(o) = U(o,0)$ is the utility of receiving the outcome o now.[7]

Suppose, further, that the problem is stationary, i.e., that the problem of deciding between actions at different times isn't affected by time advancing by a time interval $\Delta t \in T$. Then, there exist functions $\alpha'\colon T \to \mathbb{R}^+$ and $\beta'\colon T \to \mathbb{R}$ such that

$$U(o, t+\Delta t) = \alpha'(\Delta t)\, U(o,t) + \beta'(\Delta t). \tag{3.24}$$

Comparing Equations 3.23 and 3.24, we find that for all t and o:

$$\begin{aligned}
\alpha(t+1)\, U(o) + \beta(t+1) &= U(o,t+1) \\
&= \alpha'(1)\, U(o,t) + \beta'(1) \\
&= \alpha'(1)(\alpha(t)\, U(o) + \beta(t)) + \beta'(1) \\
&= \alpha'(1)\alpha(t)\, U(o) + (\alpha'(1)\beta(t) + \beta'(1))\,.
\end{aligned}$$

[7]Note that $\alpha(t)$ and $\beta(t)$ just describe the effect of delaying the action; they don't capture any intrinsic costs of computation, such as resources consumed, which might vary for different kinds of computations.

Thus, denoting $a = \alpha'(1)$ and $b = \beta'(1)$, we have:

$$\begin{aligned} \alpha(t+1) &= a\,\alpha(t) \\ &= a^{t+1} \end{aligned} \tag{3.25}$$

$$\begin{aligned} \beta(t+1) &= b + a\,\beta(t) \\ &= b\,(a^t + a^{t-1} + \cdots + 1) \\ &= b\,\frac{1 - a^{t+1}}{1 - a}. \end{aligned} \tag{3.26}$$

Observe also that if delaying is a cost rather than a benefit, then $0 \leq a \leq 1$ and $b \leq 0$. Combined with Equation 3.23, we see that each unit of time delay incurs (potentially) two costs: a discount a, devaluing the future, and a cost b of spending time.

In this chapter, we have defined a number of key theoretical concepts and established a variety of their fundamental properties:

- metalevel control problems and their associated policies and value, illustrated first with the Beta-Bernoulli example and then defined for the general case,
- two subclasses of MCPs that have useful properties: stationary MCPs and and Markov MCPs,
- metalevel MDPs and their correspondences with stationary and Markov MCPs,
- factored MMDPs, which are composed of other MMDPs,
- the cost of time, and its derivation.

These definitions and properties equip us to analyze metalevel policies in Chapter 4, in particular exploiting the structure of MCPs to establish bounds on metalevel computation and understand the effect of context.

Chapter 4

Structure of metalevel policies

4.1 Bounding computation **40**
4.1.1 Bounding in expectation 40
4.1.2 Bounding 1-action metalevel MDPs 42
4.1.3 Bounding k-action metalevel MDPs 45
4.2 Context effects **46**
4.2.1 No index policies for metalevel decision problems 46
4.2.2 Context is not just a number 48
4.2.3 Context and stopping 52

This chapter investigates the structure of optimal policies for the metalevel MDPs defined in Chapter 3. It addresses two main questions:

1. Can you **bound** the amount of computation an optimal policy performs?
2. How can the optimal strategy for computing about an action depend on its **context**, i.e., by the other actions and the agent's state of knowledge about them?

These questions may best be understood by visualizing the behavior of the optimal policy. Figure 4.1 shows that state space of the 1-action Beta-Bernoulli metalevel MDP of computation cost d composed with the constant metalevel MDP of known utility u. Each diagram of the figure plots two policies (the optimal one and the *myopic* one defined in Definition 4.5), while varying values for d and u.

Note in particular the filled colored nodes in the diagrams, which represent the states in which these policies compute. Call this set of nodes the **continuation region**.

Observe first that in this problem, the continuation region is bounded, that is, the optimal policy always eventually stops computing. Further, this bound increases with the inverse cost $1/d$. Section 4.1 presents results on when and how one can bound the number of computations performed by the optimal policy.

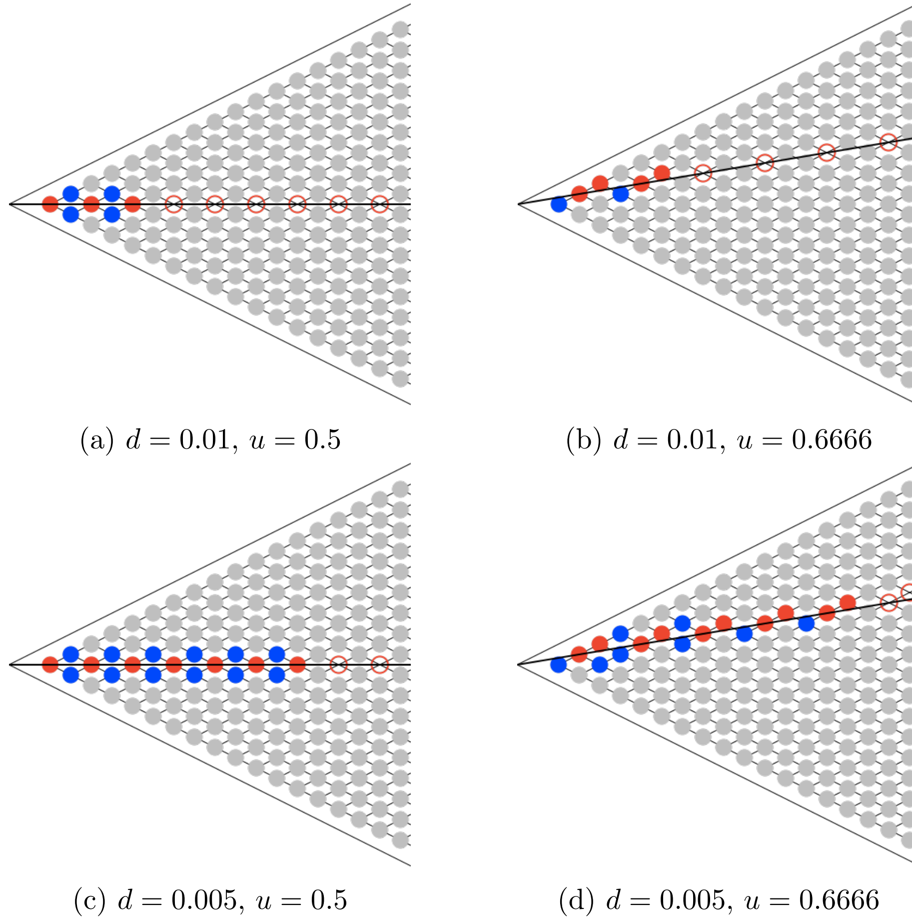


Figure 4.1: The above depicts the state space of the 1-action Beta-Bernoulli metalevel MDP of cost d composed with an action of known utility u. The cost d varies vertically and the action of known utility u varies horizontally. Its triangular structure is similar to that of Figure 3.2 but zoomed out. The black line is the line of posterior utility equal to u, i.e., it covers all (fractional) states s such that $\mu_1(s) = u$. The color of the nodes indicates the decisions made by the optimal and myopic policies (defined in Definition 4.5). Both policies stop in gray nodes, and both compute in red nodes. In blue nodes, the optimal policy computes, while the myopic one stops. Filled colored nodes are reachable from the initial state at the apex of the triangle. Open nodes are not.

Observe second that the continuation region clusters around the black line in the figure that depicts the states s whose posterior expected utility $\mu_1(s)$ is near that of the utility u of the constant alternative. Section 4.2 presents results on the effect that the context of an action, such as u in Figure 4.1, can have on the optimal policy for that action.

4.1 Bounding computation

What bounds can we establish on the number of computations performed by the optimal policy? Section 4.1.1 shows that although there is a fully general bound on the *expected* number of computations proportional to the inverse cost (Theorem 4.3), there are cases where the *actual* number of computations an optimal policy performs is *unbounded* (such as Example 4.4). Sections 4.1.2 and 4.1.3 then show general cases where one *can* find such a bound.

4.1.1 Bounding in expectation

In order to bound the amount of computation performed from a given state s of a metalevel MDP, we'll need an upper bound on the utility the agent can expect to achieve with *unbounded* computation. Metalevel MDPs that come from MCPs automatically have such a bound: $\mathbb{E}[\max_i U_i \mid \tilde{S} = s]$. The following definition gives the extension required for metalevel MDPs in general:

Definition 4.1. A **coherent upper bound** on a metalevel MDP $M = (\mathcal{S}, \mathcal{A}, \mu, d, T, R)$ is a function $\mu^* \colon \mathcal{S} \to \mathbb{R}$ such that for all $s \in \mathcal{S}$ and $c \in \mathcal{C}(s)$

$$\mu^*(s) \geq \max_i \mu_i(s),$$
$$\mu^*(s) = \sum_{s'} T(s, c, s')\mu^*(s').$$

Lemma 4.2. *Given a metalevel MDP derived from an MCP*

$$\mu^*(s) = \mathbb{E}[\max_i U_i \mid \tilde{S} = s]$$

is a coherent upper bound.

Proof. This is clearly an upper bound, and it is coherent by the tower property of expectations. □

Theorem 4.3. *In a metalevel MDP M with a coherent upper bound μ^*, the expected number of computations performed by an optimal policy π^* is bounded:*

$$\mathbb{E}_M^{\pi^*}[N \mid S_0 = s] \leq \frac{1}{d}\left(\mu^*(s) - \max_i \mu_i(s)\right). \tag{4.1}$$

If the metalevel MDP is derived from a metalevel control problem, this can be interpreted as the value of perfect information divided by the cost of computation:

$$\mathbb{E}_M^{\pi^*}[N \mid S_0 = s] \leq \frac{1}{d}\left(\mathbb{E}[\max_i U_i \mid \tilde{S} = s] - \max_i \mathbb{E}[U_i \mid \tilde{S} = s]\right).$$

Further, this holds for any policy π that outperforms the policy that does no computation.

Proof. We prove the more general case. Recalling Lemma 3.13, we have for any π that

$$\begin{aligned}
V_M^\pi(s) &= \mathbb{E}_M^\pi[-d\,N + \max_i \mu_i(S_N) \mid S_0 = s] \\
&= -d\,\mathbb{E}_M^\pi[N \mid S_0 = s] + \mathbb{E}_m^\pi[\max_i \mu_i(S_N) \mid S_0 = s] \\
&\leq -d\,\mathbb{E}_M^\pi[N \mid S_0 = s] + \mathbb{E}_m^\pi[\mu^*(S_N) \mid S_0 = s] \\
&= -d\,\mathbb{E}_M^\pi[N \mid S_0 = s] + \mu^*(s)
\end{aligned}$$

where the inequality in line 3 holds as $\max_i \mu_i(s) \leq \mu^*(s)$ and the equality in line 4 as $\mu^*(S_t)$ is a martingale. Rearranging, we find:

$$\mathbb{E}_M^\pi[N \mid S_0 = s] \leq \frac{1}{d}\left(\mu^*(s) - V_M^\pi(s)\right).$$

The result then follows because by hypothesis $V^\pi(s) \geq \max_i \mu_i(s)$, since $\max_i \mu_i(s)$ is the value function for the policy $\pi_{\text{ACT}}(s) = \text{ACT}$ that does no computation. If the metalevel MDP is derived from a metalevel control problem, the second part of the theorem follows immediately from Definition 4.2. □

As noted above, although the *expected* number of computations is always bounded, there are important cases in which the *actual* number is not, such as the following inspired by the sequential probability ratio test (Wald, 1945):

Example 4.4. Consider the 1-action Beta-Bernoulli model but with different prior: Θ_1 is $1/3$ or $2/3$ with equal probability. After performing computations with s simulated successes and f simulated failures, the posterior odds ratio is

$$\frac{\mathbb{P}(\Theta_1 = 2/3 | s, f)}{\mathbb{P}(\Theta_1 = 1/3 | s, f)} = \frac{(2/3)^s (1/3)^f}{(1/3)^s (2/3)^f} = 2^{s-f}.$$

Note that this ratio completely determines the posterior distribution of Θ_1:

$$\mathbb{P}(\Theta_1 = u) = \begin{cases} 1/(2^{f-s}+1) & \text{if } u = 2/3, \\ 1/(2^{s-f}+1) & \text{if } u = 1/3, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, whether it is optimal to stop or compute is a function only of this ratio and thus of $s-f$. For sufficiently low computation cost, the optimal policy computes when $s - f \in \{-1, 0, 1\}$. But with probability $1/3$, a state with $s - f = 0$ transitions to another with $s - f = 0$ after two samples, giving a finite, although exponentially decreasing, probability to arbitrary long sequences of computations. In particular, for all $n \geq 0$:

$$(1/3)^n \leq \mathbb{P}[N \geq 2n].$$

Note that the cost of computation can far exceed the value of computing, even for an optimal policy; there is no sense in which the policy says "I've already spent more on that than it's worth." This may seem strange, but it is the *correct* choice in this situation: the amount already spent on computation is a sunk cost, so it should not influence your future decisions.

4.1.2 Bounding 1-action metalevel MDPs

In a number of settings, including the original Beta-Bernoulli example, we can give a concrete upper bound on the number of computations performed by the optimal policy. To do this, we first need to get an analytical handle on the optimal policy. The key insight comes from considering two natural *suboptimal* policies.

The first is π^{ACT} which always acts immediately: $\pi^{\text{ACT}}(s) = \text{ACT}$ for all $s \in S$. Its value and Q-value functions are a lower bound on the optimal value and Q-value functions:

$$V_M^*(s) \geq V_M^{\pi^{\text{ACT}}}(s) = \max_i \mu_i(s) \tag{4.2}$$

$$Q_M^*(s,a) \geq Q_M^{\pi^{\text{ACT}}}(s,a) = \begin{cases} \max_i \mu_i(s) & \text{if } a = \text{ACT}, \\ -d + \sum_{s'} T(s'|s,a) \max_i \mu_i(s') & \text{otherwise.} \end{cases} \tag{4.3}$$

The **myopic policy** π^m (known as the metagreedy approximation with single-step assumption in (Russell and Wefald, 1991a)) makes the best decision, to either stop and ACT or perform a computation, assuming (perhaps wrongly) that it has at most one computation left to do. This is equivalent to believing that its future decisions are made by π^{ACT}, i.e., to optimizing $Q_M^{\pi^{\text{ACT}}}(s,a)$:

Definition 4.5. The **myopic policy** optimizes the lower bound in Equation 4.3:

$$\pi^m(s) = \operatorname*{argmax}_a Q^{\pi^{\text{ACT}}}(s,a), \tag{4.4}$$

breaking ties towards ACT.

The myopic policy has a tendency to stop too early, because changing one's mind about which object-level action to take often takes more than one computation. Concretely, the myopic policy stops in a state $s \in S$ iff

$$-d + \max_{c \in \mathcal{C}} \sum_{s'} T(s'|s,c) \max_i \mu_i(s') \leq \max_i \mu_i(s)$$

$$\max_{c \in \mathcal{C}} \left[\sum_{s'} T(s'|s,c) \max_i \mu_i(s') \right] - \max_i \mu_i(s) \leq d, \tag{4.5}$$

otherwise performing the computation achieving the maximum in Equation 4.5.

However, if the myopic policy *doesn't stop*, then neither does the optimal policy:

Lemma 4.6. *In a metalevel MDP $M = (S, A, T, R, \mu, d)$, if the myopic policy performs a computation in state $s \in S$, then the optimal policy does too, i.e., if $\pi^m(s) \neq$ ACT then $\pi^*(s) \neq$ ACT.*

Proof. Let s be a state in which the myopic policy doesn't stop. Then:

$$\begin{aligned} Q^*_M(s, \text{ACT}) &= \max_i \mu_i(s) \\ &= Q^m_M(s, \text{ACT}) \\ &\leq \max_{c \in A(s)} Q^m_M(s, c) \\ &\leq \max_{c \in A(s)} Q^*_M(s, c). \end{aligned}$$

□

There is a partial converse:

Theorem 4.7. *Given a metalevel MDP $M = (S, A, T, R, \mu, d)$ that has a subset of states $\bar{S} \subseteq S$ closed under transitions, if the myopic policy stops on all states in $\bar{S}$ then so does the optimal policy.*

Proof. Take any state $s \in \bar{S}$ and note that all states to which the chain can transition are also in $\bar{S}$, by closure under transitions. If we initialize the chain with $S_0 = s$ then the myopic policy will stop in *all* states reachable under *any* policy π, i.e., that for all $t > 0$:

$$\mathbb{E}^\pi_M[1(t < N)(\max_i \mu_i(S_{t+1}) - d) \,|\, S_0 = s] \leq \mathbb{E}^\pi_M[1(t < N) \max_i \mu_i(S_t) \,|\, S_0 = s]$$

where $1(t < N)$ is a random variable equal to 1 if $t < N$ and 0 otherwise. Therefore:

$$\begin{aligned} V^\pi_M(s) &= \mathbb{E}^\pi_M[-d\,N + \max_i \mu_i(S_N) | S_0 = s] \\ &= \mathbb{E}^\pi_M[\max_i \mu_i(S_0) + \sum_{t=0}^{N-1} (\max_i \mu_i(S_{t+1}) - d - \max_i \mu_i(S_t)) \,|\, S_0 = s] \\ &\leq \mathbb{E}^\pi_M[\max_i \mu_i(S_0) \,|\, S_0 = s] \\ &= \max_i \mu_i(s). \end{aligned}$$

Thus, stopping maximizes expected utility in state s, so the optimal policy stops. □

As a concrete corollary:

Theorem 4.8. *The number of computations performed in the 1-action Beta-Bernoulli decision problem with constant action $u \in [0, 1]$ is at most:*

$$\frac{u(1-u)}{d} - 3 \leq \frac{1}{4d} - 3.$$

Proof. States of this MDP are of the form $s = (\alpha, \beta)$. Fixing one particular $s = (\alpha, \beta)$, let $n = \alpha + \beta$, and let $s^+ = (\alpha + 1, \beta)$ and $s^- = (\alpha, \beta + 1)$ be the states reached from s after a simulation that succeeds or fails, respectively. First note that with $n = \alpha + \beta$ we have:

$$\mu_1(s^+) = \frac{\alpha + 1}{n + 1} = \frac{n}{n+1}\mu_1(s) + \frac{1}{n+1},$$
$$\mu_1(s^-) = \frac{\alpha}{n + 1} = \frac{n}{n+1}\mu_1(s).$$

Next, observe that the myopic policy stops in this state s if

$$\begin{aligned} d &\geq \mu_1(s)\max(\mu_1(s^+), u) + (1 - \mu_1(s))\max(\mu_1(s^-), u) - \max(\mu_1(s), u) \\ &= \mu_1(s)\max\left(\frac{n}{n+1}\mu_1(s) + \frac{1}{n+1}, u\right) + (1 - \mu_1(s))\max\left(\frac{n}{n+1}\mu_1(s), u\right) \\ &\quad - \max(\mu_1(s), u). \end{aligned}$$

Note that $\mu_1(s) \in [0, 1]$ and consider the above function of $\mu_1(s)$. Breaking into cases to remove the maximum operators, we can restate the inequality piecewise as:

$$d \geq \begin{cases} \mu_1(s)u + (1 - \mu_1(s))u - u & \text{if } \mu_1(s) \in [0, u - (1-u)/n] \\ \mu_1(s)\left(\frac{n}{n+1}\mu_1(s) + \frac{1}{n+1}\right) + (1 - \mu_1(s))u - u & \text{if } \mu_1(s) \in [u - (1-u)/n, u] \\ \mu_1(s)\left(\frac{n}{n+1}\mu_1(s) + \frac{1}{n+1}\right) + (1 - \mu_1(s))u - \mu_1(s) & \text{if } \mu_1(s) \in [u, u + u/n] \\ \mu_1(s)\left(\frac{n}{n+1}\mu_1(s) + \frac{1}{n+1}\right) + (1 - \mu_1(s))\frac{n}{n+1}\mu_1(s) - \mu_1(s) & \text{if } \mu_1(s) \in [u + u/n, 1] \end{cases}$$

$$d \geq \begin{cases} 0 & \text{if } \mu_1(s) \in [0, u - (1-u)/n] \\ \frac{n}{n+1}\mu_1(s)^2 + \left(\frac{1}{n+1} - u\right)\mu_1(s) & \text{if } \mu_1(s) \in [u - (1-u)/n, u] \\ \frac{n}{n+1}\mu_1(s)^2 + \left(\frac{1}{n+1} - u - 1\right)\mu_1(s) + u & \text{if } \mu_1(s) \in [u, u + u/n] \\ 0 & \text{if } \mu_1(s) \in [u + u/n, 1]. \end{cases}$$

Observe that the second piece is increasing and the third decreasing, so the maximum of the function is at their junction, $\mu_1(s) = u$, and so the following inequality is sufficient to ensure myopic stops in s:

$$d \geq \frac{u(1-u)}{n+1}$$
$$n \geq \frac{u(1-u)}{d} - 1.$$

Finally, observe that the initial state in this problem is $(1, 1)$, which has $n = 2$, so the number of computations is bounded above by $\frac{u(1-u)}{d} - 3$. Optimizing over u gives the weaker bound of $\frac{1}{4d} - 3$. The result then follows from Theorem 4.7. □

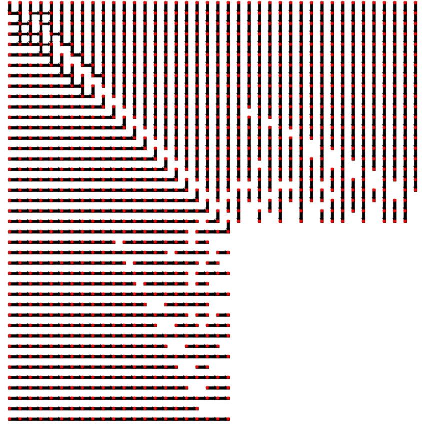


Figure 4.2: All states $(\alpha_1, \beta_1, \alpha_2, \beta_2)$ with $n_1 = \alpha_1 + \beta_1 - 2 \leq 40$ and $n_2 = \alpha_2 + \beta_2 - 2 \leq 40$ which are reachable by the optimal policy from initial state $(1, 1, 1, 1)$; such a state is plotted at the coordinates (n_1, n_2). A downward line leaving a state means it is optimal to sample the first action, a rightward line that it is optimal to sample the second, and no lines that it is optimal to stop (ties towards stopping then sampling the first action). Notice the L shape, where the arms of the L are 1-action problems. The arms extend off to infinity.

4.1.3 Bounding k-action metalevel MDPs

Bounds for 1-action metalevel MDPs, such as those established in the previous section, can be extended to bounds in the k-action setting by utilizing the results of Section 3.5 on factored MMDPs.

Theorem 4.9. *For $i = 1, \ldots, l$ let $M_i = (\mathcal{S}_i, \mathcal{A}_i, \mu^i, d, T_i, R_i)$ be a metalevel MDP in which it is optimal to perform at most n_i computations in $M_i + u$ for all $u \in \overline{\mathbb{R}}$ from any $s^i \in \mathcal{S}_i$. Then it is optimal to perform at most $\sum_{i=1}^{l} n_i$ computations in the metalevel MDPs $\left(\sum_{i=1}^{l} M_i\right) + u$ and $\left(\sum_{i=1}^{l} M_i\right)$ from any state.*

Proof. It suffices to establish the result for $\left(\sum_{i=1}^{l} M_l\right) + u$, for in general $M + (-\infty)$ is equivalent to M. Further, we can assume $l = 2$ since the $l = 1$ case is immediate and $l > 2$ follows by induction.

For each i and $s^i \in \mathcal{S}_i$, let $n_i^*(s^i)$ be the maximal number of computations the optimal policy performs starting in s^i in $M_i + u$ for any $u \in \overline{\mathbb{R}}$, which is well-defined and finite since it is at most n_i. Note that if s_i' is reachable from s_i in a single transition, then $n_i^*(s_i') \leq n_i^*(s_i) - 1$. Let $\mathcal{S}_i^0 = \{s_i \in \mathcal{S}_i : n_i^*(s_i) = 0\} \subseteq \mathcal{S}_i$ be the set of states in which it is optimal to stop in M_i, and observe that this set is closed under transitions.

Define $n^*(s^1, s^2) = (n_1^*(s^1), n_2^*(s^2))$ for states (s^1, s^2) of $M_1 + M_2 + u$. We'll show that computations must decrease one of the two components, and that it is optimal not to perform

M_i's computations if the ith component is zero. The result then follows as $n_1^*(s^1)+n_2^*(s^2) \leq n_1+n_2$.

The first follows because performing M_i's computation changes only the ith component of the state, and we've already observed that if s_i' is reachable from s_i, then $n_i^*(s_i') \leq n_i^*(s_i)-1$.

The second follows by applying Corollary 3.18 twice to the hypothesis, once to M_1 and (M_2+u), the other to M_2 and (M_1+u). We see it is optimal in M_1+M_2+u not to perform M_1's computations in the states $\mathcal{S}_1^0 \times \mathcal{S}_2$, and not to perform M_2's computations in the states $\mathcal{S}_1 \times \mathcal{S}_2^0$ (see Figure 4.2).

As a result, from a state (s^1, s^2) at most $n_1^*(s^1)$ computations of M_1 will be performed and at most $n_2^*(s^2)$ computations of M_2 will be performed, thus at most $n_1^*(s^1)+n_2^*(s^2) \leq n_1+n_2$ computations will be performed. □

Corollary 4.10. *The number of computations performed in the k-arm Beta-Bernoulli decision problem is at most:*

$$k\left(\frac{1}{4d}-3\right).$$

Proof. Combine the results of Theorem 4.8 and Theorem 4.9. □

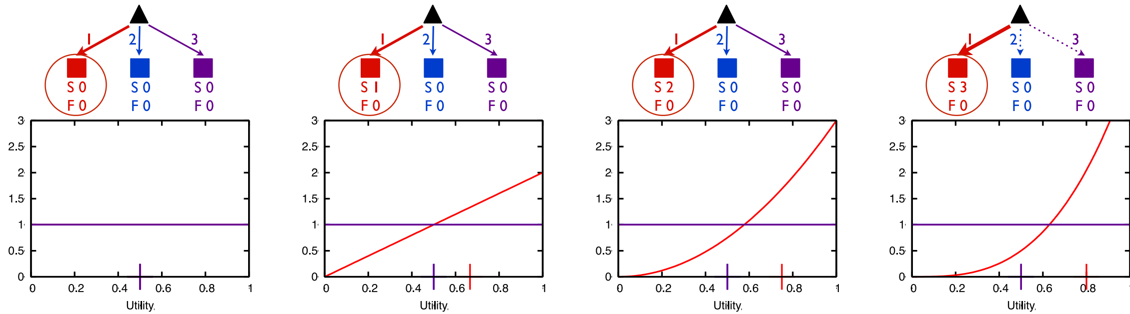
One key implication is that the *optimal* policy can be computed in time $O(1/d^2)$. This is particularly appropriate when the cost of computation is relatively high, such as in simulation experiments (Swisher et al., 2003), or when the decision to be made is critical. See Figure 4.3 for a trace of the optimal policy for the 3-action Beta-Bernoulli problem, which can be computed by using the above bounds to make the state space finite. The choice of which action to compute depends on the posterior, but there is a bias towards uncertain actions and towards actions which might be best, although the exact tradeoff is subtle.

4.2 Context effects

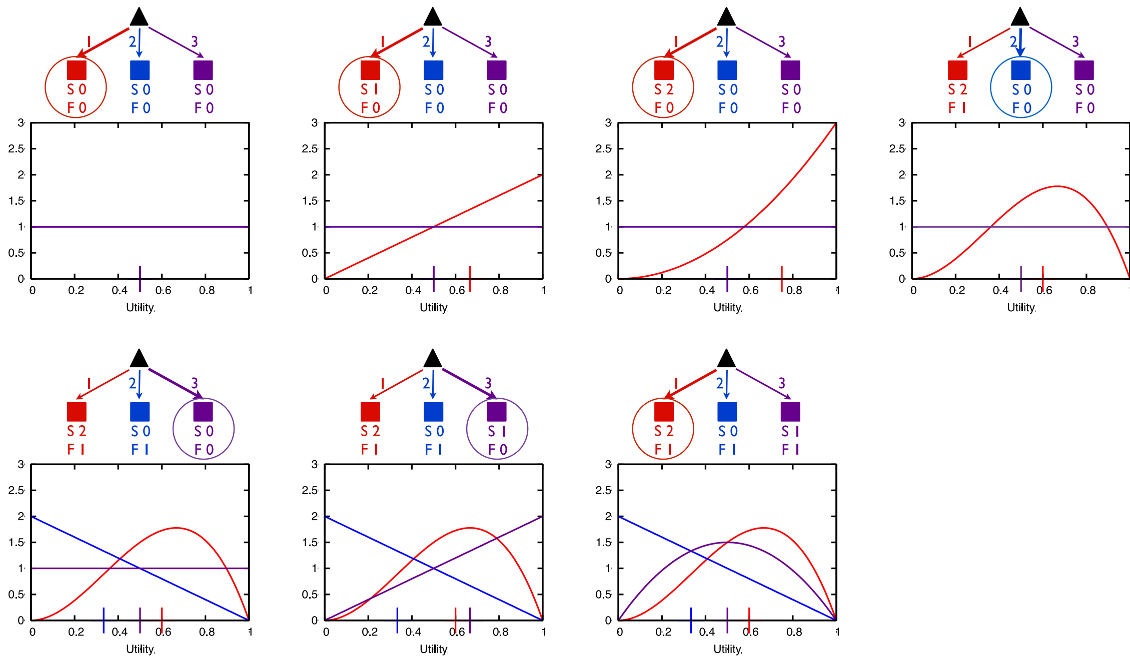
We now consider a different aspect of the structure of optimal policies: the effect of context. Section 4.2.1 gives a counter-example where the context has a significant influence on the optimal computation, while Section 4.2.2 shows that an action's context cannot be simply summarized. Section 4.2.3 provides a counterbalance, showing that the pattern observed in Figure 4.1 of the continuation region clustering around a constant context u holds in general for these contexts.

4.2.1 No index policies for metalevel decision problems

The Gittins index theorem (Gittins, 1979) is a famous structural result for bandit problems (Section 1.2.3). It states that in bandit problems with independent reward distribution for each arm and geometric discounting, the optimal policy is an **index policy**: each arm is assigned a real-valued index based on its state only, such that it is optimal to sample the arm with the greatest index.



(a) Continual success makes for a quick decision. Trace: simulate action 1, succeed; simulate action 1, succeed; simulate action 1, succeed; stop computing and take action 1.



(b) Mixed results merit prolonged thought. Trace: simulate action 1, succeed; simulate action 1, succeed; simulate action 1, fail; simulate action 2, fail; simulate action 3, succeed; simulate action 3, fail; stop computing and take action 1.

Figure 4.3: Two traces of the optimal policy for the 3-action Beta-Bernoulli metalevel control problem with cost $d = 0.01$. This was solved exactly using the results of Section 4.1. Each trace consists of a number of states s depicted in two ways. The triangle-rooted trees (above the graphs) provide counts of simulated successes (top row; equal to $s[\alpha_i] - 1$) and simulated failures (bottom row; equal to $s[\beta_i] - 1$), where edges correspond to the three possible actions and the circle indicates the action computed or taken. Each graph in the figure overlays the posterior probability densities for each action, color-coded to match action edges, with a vertical line on the x-axis marking the posterior mean of each density.

The analogous result does *not* hold for metalevel decision problems, even when the action's values are independent, as the following example shows:

Example 4.11 (Non-indexability)**.** Consider a metalevel probability model with three actions. U_1 is equally likely to be -1.5 or 1.5 (low mean, high variance), U_2 is equally likely to be 0.25 or 1.75 (high mean, low variance), and $U_3 = u$ has a known value (the context). The two computations are to observe exactly U_1 and U_2, respectively, each with cost 0.2.

The corresponding metalevel MDP has $3 \times 3 = 9$ states, which can be denoted (u_1, u_2) for $u_i \in \{0, +, -\}$: either of u_1 or u_2 is unobserved (0), observed to have a large value (+), or observed to have a small value ($-$).

This metalevel MDP can be exactly solved, as a function of u, by considering all policies that might be optimal. Note that if both variables are observed, it is optimal to stop, and if $U_2 = 1.75$, then it is optimal to stop regardless of whether U_1 is measured: it's known that $U_1 < U_2$, and after measuring U_2 it is known whether $u > U_2$ or $u \leq U_2$. Thus in the five states $(+,+)$, $(+,-)$, $(-,+)$, $(-,-)$, and $(0,+)$, an optimal policy must act. This leaves only the four states in which potentially optimal policies can vary in their choice: $(0,0)$, in which a policy can either act, observe U_1, or observe U_2; $(-,0)$ and $(+,0)$, in which a policy can either act, or observe U_2; and $(0,-)$, in which a policy can either act or observe U_1. If a state is unreachable under the policy, it doesn't matter which action is taken.

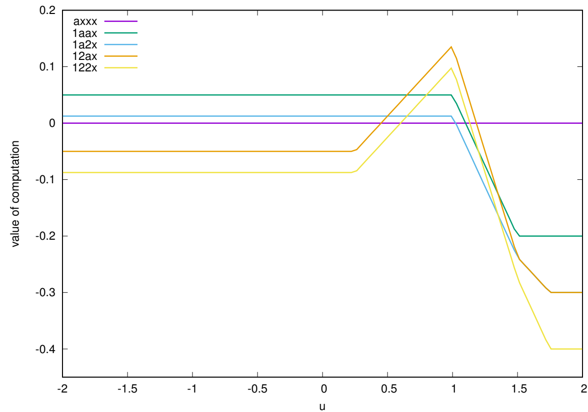
Notate a policy by a string giving its action in these four states in order, with 1 for observing U_1, 2 for observing U_2, a for acting, and x for unreachable. Figure 4.4a gives the expected utility of all policies that observe U_1 in state $(0,0)$, minus the utility of the policy that acts immediately, and Figure 4.4b does the same for policies observing U_2 first. Figure 4.4c compares just the two best policies from the previous figures, giving the optimal Q-values for observing U_1, observing U_2, or acting.

An index policy would assign a value of observing U_1 depending only on what is known about U_1, and similarly for observing U_2. In particular, this is independent of the value of u which affects only U_3, so in state $(0,0)$ there's a value λ_1 of observing U_1 and λ_2 of observing U_2. However, note in Figure 4.4c that if $u = 0$ then it's better to observe U_1, so we must have $\lambda_1 > \lambda_2$, but if $u = 1$ it is better to observe U_2, so we must have $\lambda_1 < \lambda_2$, a contradiction. Therefore the optimal policy is not an index policy, unlike in bandit problems.

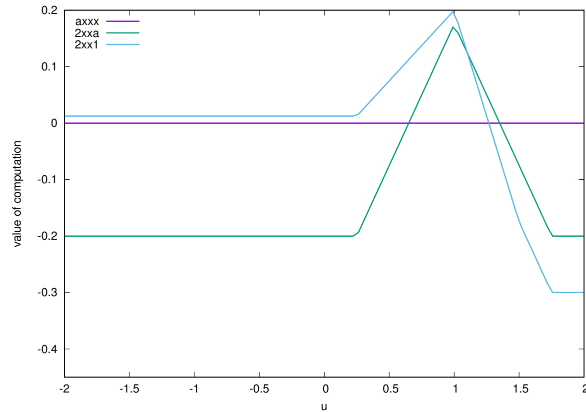
4.2.2 Context is not just a number

The previous example used a constant context $u \in \mathbb{R}$. In deciding what to compute about the ith action, is it sufficient to use a 1-dimensional summary of the other actions like their expected utility? No, as the following analysis of the Bernoulli metalevel MDP establishes.

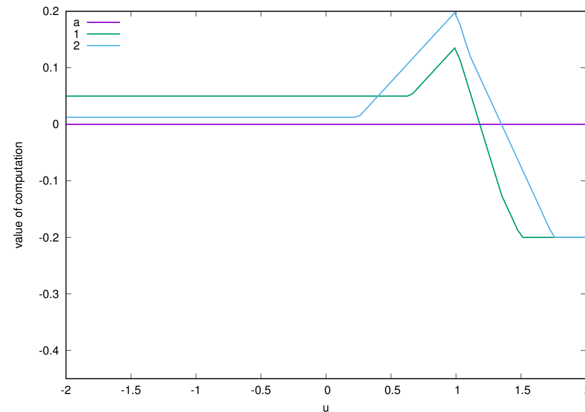
In a Bernoulli selection problem, each action's utility is independently distributed according to a Bernoulli distribution. The agent can compute, at cost d, the exact value of



(a) The expected utility of all the policies that observe U_1 in state $(0,0)$, minus the utility of the policy that acts immediately.



(b) The expected utility of all the policies that observe U_2 in state $(0,0)$, minus the utility of the policy that acts immediately.



(c) The expected utility of the maximum over the policies from (a) and (b), giving the optimal Q-values for observing U_1 or U_2 relative to stopping and acting.

Figure 4.4: The value function for Example 4.11.

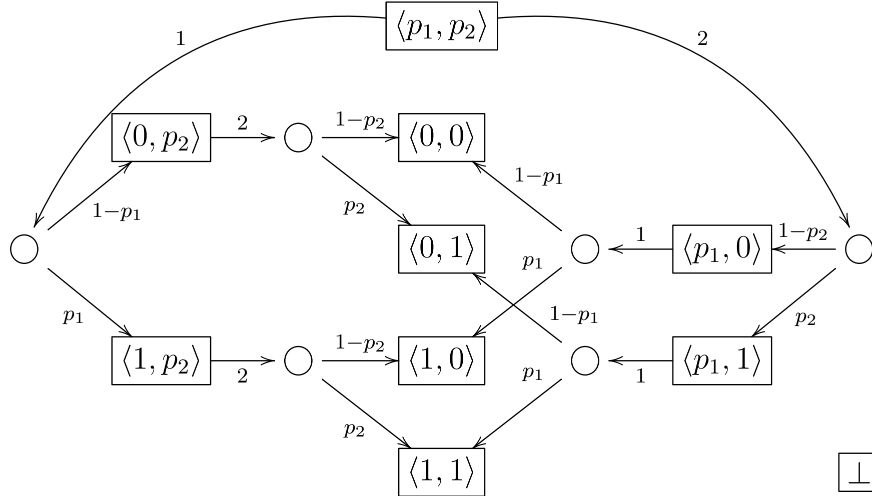


Figure 4.5: Metalevel states and transitions in a Bernoulli selection problem with two actions. Rectangular nodes are metalevel states, with outgoing arrows labeled by computations (computations not depicted are those that have already been performed so repeats won't change the state). Circular nodes are chance nodes whose outgoing edges label possible successor states with their probability. Choosing to take an action transitions from any state to the unique terminal state $\perp$ (at bottom right with incoming arrows elided).

any action. This corresponds to the metalevel probability model

$$\begin{aligned} U_i &\sim \text{Bernoulli}(p_a) && \text{for } i = 1, \ldots, k, \\ M_{\text{ber}} &= (\{U_i\}_{i=1,\ldots,k}, \{U_i\}_{i=1,\ldots,k}), \end{aligned}$$

where we assume without loss of generality that the p_i are non-increasing.

There are 3^k possible metalevel states (not including the unique terminal state), that can each be denoted by vectors giving for each of the k actions i the marginal probability that $U_i = 1$ (either 0, 1, or p_i) e.g. if $k = 5$ we denote metalevel state $\{(2, 0), (3, 0), (5, 1)\}$ by $\langle p_1, 0, 0, p_4, 1\rangle$.

For $k = 2$ this metalevel MDP is small enough to illustrate (Figure 4.5), and small enough to solve by hand (Figure 4.6). Observe that its optimal to ACT when any action is observed to have utility 1.

It's likewise optimal for general k to ACT if any action is observed to have utility 1: Since the maximum utility is 1, performing any computation yields utility at most $1 - d$. This implies that after performing a computation there's at most one successor state that performs a computation (the one where the previous computational outcome was 0), and so the policy observes variables in a fixed order $a_1, \ldots, a_k$ given by:

$$a_i = \pi(\{(a_j, 0) : \text{for } j < i\})$$

where m is the least index such that $a_{m+1} = \text{ACT}$. In fact, we can say more:

m	$Q^*(m, \text{ACT})$	$Q^*(m,1)$	$Q^*(m,2)$
$\langle 0,0\rangle$	0		
$\langle 0,1\rangle$	1		
$\langle 1,0\rangle$	1		
$\langle 1,1\rangle$	1		
$\langle p_1,0\rangle$	p_1	$-d+p_1$	
$\langle p_1,1\rangle$	1	$-d+1$	
$\langle 0,p_2\rangle$	p_2		$-d+p_2$
$\langle 1,p_2\rangle$	1		$-d+1$
$\langle p_1,p_2\rangle$	p_1	$-d+p_1+(1-p_1)p_2$	$-d+p_2+(1-p_2)p_1$

Figure 4.6: Optimal Q-function for two-action Bernoulli selection problem computed by backwards induction. Repeated computations are elided since the value of repeating a computation in state m is $V^*(m) - d$: always suboptimal.

Theorem 4.12. *In the Bernoulli selection problem there is some $0 \leq m < k$ such that it is optimal to observe action utilities $1, 2, \ldots, m$ in sequence,* ACT*ing immediately if any has utility 1, otherwise* ACT*ing after the full sequence.*

Proof. The above discussion shows that we can restrict our search for optimal policies to those that observe variables $a_1, \ldots, a_m$ in a fixed order, stopping early upon observing a 1. We can further assume that $m < k$, as it is optimal to ACT when there's only one unknown action left: the other $k-1$ actions will have been observed to have value 0, so the final action's utility cannot be worse.

Thus, it suffices to show for any $0 \leq m < k$ that the value of any policy $a_1, \ldots, a_m$ is no more than the value of the policy $1, \ldots, m$. We'll give a closed-form expression for the value of such a policy and show we can transform any policy $a_1, \ldots, a_m$ into $1, \ldots, m$ using transformations that only increase policy's value, namely sorting $a_1, \ldots, a_m$ in increasing order (i.e., such that if $i < j$ then $a_i < a_j$) and replacing actions a_i by smaller actions (i.e., ones with high prior probability of one).

Fix a policy $a_1, \ldots, a_m$. Let a_{m+1} be the action of least index (i.e., that with the highest prior probability of being one) not in the set $\{a_1, \ldots, a_m\}$, and observe that this policy is guaranteed to take an action of utility 1 if there is one within the set $\{a_1, \ldots, a_{m+1}\}$, receiving utility 0 otherwise. Thus, the policy's value equals:

$$V^{[a_1,\ldots,a_m]} = -d \sum_{i=1}^{m} \prod_{j=1}^{i-1} (1 - p_{a_j}) + \left(1 - \prod_{j=1}^{m+1} (1 - p_{a_j})\right), \tag{4.6}$$

where the first term is the expected cost of performing at most m computations factoring in the chance of stopping early, and the second the expected utility of the action taken after computation.

First observe that the right term of Equation 4.6 is independent of the ordering of the a_i's, but the left term is maximized when they are in decreasing order, for suppose they weren't. Then there would be an inversion $p_{a_{l+1}} > p_{a_l}$. Now, the only two terms of the left summation that would be affected by the swap are:

$$-d\left(\prod_{j=1}^{l}(1-p_{a_j}) + \prod_{j=1}^{l+1}(1-p_{a_j})\right) = -d\prod_{j=1}^{l-1}(1-p_{a_l})(1+(1-p_{a_{l+1}}))$$
$$= -d\prod_{j=1}^{l-1}(2-2p_{a_l}-p_{a_{l+1}}+p_{a_l}p_{a_{l+1}})$$

that is clearly maximized by the swap.

Finally observe that increasing the value of any individual p_{a_i} would increase both terms of Equation 4.6, so replacing a_i with a smaller action would increase the policy's value, completing the proof. □

Theorem 4.12 reduces the k-action Bernoulli problem to a stopping problem that by Equation 4.6 can be solved by a recurrence:

$$V^{[]} = p_1$$
$$V^{[1,\ldots,m-1,m]} = V^{[1,\ldots,m-1]} + (p_{m+1}(1-p_m) - d)\prod_{j=1}^{m-1}(1-p_j)$$

Consider now the case of $d = 0.05$ and $p_1, p_2 = 0.9, 0.4$ with two values of p_3:

	$p_3 = 0.3$	$p_3 = 0.1$
$V^{[]}$	0.9	0.9
$V^{[1]}$	$0.89 = 0.9 + (0.4(1-0.9) - 0.05)$	$0.89 = 0.9 + (0.4(1-0.9) - 0.05)$
$V^{[1,2]}$	$0.903 = 0.89 + (0.3(1-0.4) - 0.05)0.1$	$0.891 = 0.89 + (0.1(1-0.4) - 0.05)0.1$

When $p_3 = 0.3$ it is optimal to compute in the initial state, and when $p_3 = 0.1$ it is optimal to stop. But in both cases the state p_1 of the action it is optimal to compute, and the value of the best alternative, p_2, are the same. That is, the context of the action we are sampling cannot be summarized as simply the expected value of the best alternative. In general, all of the alternative actions can have an influence.

4.2.3 Context and stopping

Despite the negative results of the previous sections, there is a restriction on what kind of influence the context can have, generalization the observation we made in the opening of this chapter about the 1-action Beta-Bernoulli metalevel MDP with constant context u.

Theorem 4.13. *Given a metalevel MDP M, for each $s \in S$, there is a closed interval $I(s) \subseteq \mathbb{R}$ such that it is optimal to* ACT *in state s of $M + u$ iff $u \in I(s)$. Further,* $\max_i \mu_i(s) \in I(s)$ *if $I(s)$ is nonempty.*

Proof. Consider the advantage of following π versus ACTing as a function of $u \in \mathbb{R}$:

$$\phi_s(u) = V^{\pi}_{M+u}(s) - \max(\max_i \mu_i(s), u)$$

and note that

$$I(s) = \{u : \phi_s(u) \geq 0\}.$$

Observe that by Theorem 3.15 and one-sided derivatives of max we have:

$$\lim_{u \uparrow u_0} \frac{\phi_s(u) - \phi_s(u_0)}{u - u_0} = \mathbb{P}^{\pi}_{M}\left(\max_i \mu_i(S_N) < u_0 \,|\, S_0 = s\right) - 1(u_0 > \max_i \mu_i(s))$$
$$\lim_{u \downarrow u_0} \frac{\phi_s(u) - \phi_s(u_0)}{u - u_0} = \mathbb{P}^{\pi}_{M}\left(\max_i \mu_i(S_N) \leq u_0 \,|\, S_0 = s\right) - 1(u_0 \geq \max_i \mu_i(s)),$$

so $\phi_s(u)$ is non-decreasing on $(-\infty, \max_i \mu_i(s)]$ and non-increasing on $[\max_i \mu_i(s), \infty)$. □

This brings us to the end of our theoretical investigations. The last two chapters have established a formal basis for the general problem of metalevel control, defining an equivalent class of metalevel MDPs and exploiting structural properties of both to yield results on computational bounds and the effect of context.

We turn next to our practical investigations: applying reinforcement learning to the metalevel control problem, with specific application to Monte Carlo tree search.

Chapter 5

Metalevel reinforcement learning

5.1 Challenges of metalevel reinforcement learning **55**
5.2 Pointed trees **56**
5.2.1 Definition and recursive construction 56
5.2.2 Recursive functions of pointed trees 58
5.2.3 Local operations 58
5.2.4 Local operations and recursive functions 59
5.2.5 Derivatives of recursive functions 62
5.3 Monte Carlo tree search (MCTS) **63**
5.4 MCTS as a metalevel MDP **67**
5.5 Metalevel policy class for MCTS **69**
5.5.1 UCT and AlphaGo metapolicies 69
5.5.2 Metapolicy class: Recursive component 71
5.5.3 Metapolicy class: Parameterized local component 73
5.6 Metalevel shaping rewards **74**

How can the principles of metalevel control be put into practice? The theoretical framework defined in Chapters 3 and 4 provides formal tools for understanding and reasoning about the problem of controlling computation, cast as a class of metalevel MDPs. It does not, however, address what specific (metalevel) policy should be followed by the metalevel agent.

The natural choice for MDPs is to exploit reinforcement learning techniques to find good policies. In this chapter we describe how **metalevel reinforcement learning** can be used to find good policies for controlling computation. After laying out some of the challenges that arise in applying reinforcement learning to metalevel control problems (Section 5.1), we describe **pointed trees** (Section 5.2), a recursive data structure that affords efficient learnable functions. Sections 5.3–5.6 then show how to apply this in the context of controlling

Monte Carlo tree search (MCTS; Section 1.2.3). Experiments applying these techniques are given in Chapter 6.

5.1 Challenges of metalevel reinforcement learning

We focus for concreteness on the specific application of controlling Monte Carlo tree search (MCTS, described in Section 5.3). Several challenges arise in applying reinforcement learning (Section 2.4) to the problem of controlling computations in this context:

1. We must precisely define the metalevel MDP for which we'll be finding policies. The states of the metalevel MDP are possible internal states of the algorithm. For MCTS, this internal state is a tree, along with a pointer to the node in the tree the algorithm is currently processing. Metalevel actions control search, centrally by navigating through and manipulating this search tree.

2. Reinforcement learning uses and modifies various functions of state and state-action pairs: policies, value functions and Q-functions. In typical reinforcement learning applications, state spaces can be encoded as arrays of fixed shape (e.g. 2d or 3d arrays for images, 2d arrays for audio sequences). But for controlling search, the elements of the state space include trees, which have complex recursive structure. Naively, functions of such trees (such as those used in MCTS) would take time proportional to the tree's size to evaluate, which would quickly become intractable. How can we tractably represent policies and value functions on this space?

3. There is no intrinsic value to computing: a computation is useful only inasmuch as it leads to a better action choice down the line. There may be hundreds, thousands or millions of computations before acting. This means that rewards, which can come only by acting in the environment, will be very rare. How should we handle this reward sparsity?

The rest of the chapter addresses these issues as follows:

Section 5.2 addresses (1) above by defining the **pointed tree**, a representation with properties useful for modeling and implementing Monte Carlo tree search (Section 1.2.3). Section 5.3 describes Monte Carlo tree search with pointed trees as internal states, including the specific case of UCT. MCTS is then cast as a metalevel MDP in Section 5.4.

Section 5.5 addresses (2) above by describing a tractable metalevel policy class suitable for learning to control MCTS.

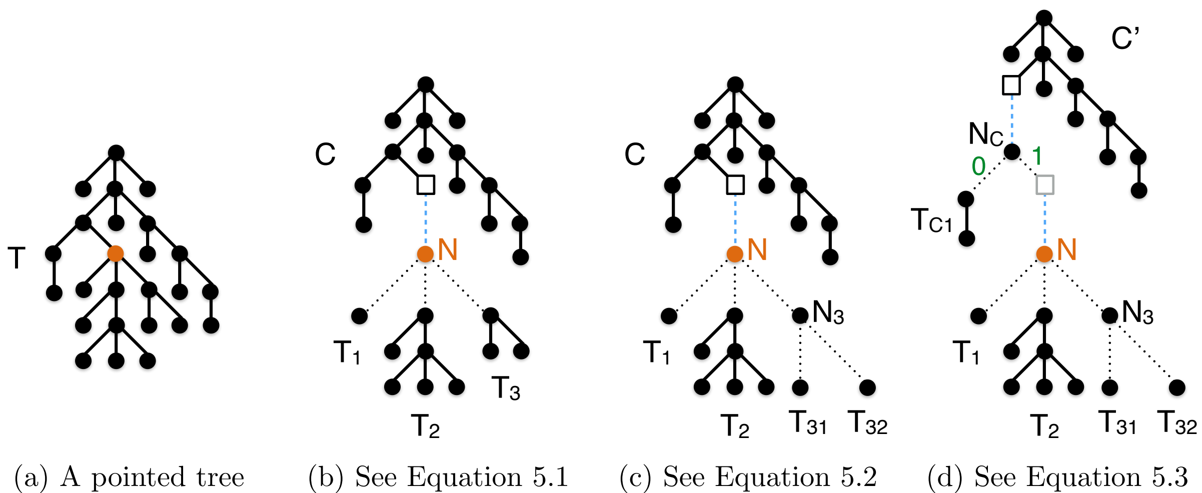
Finally, Section 5.6 addresses (3) above by describing a method for applying **shaping rewards** to reduce reward sparsity based on the estimated best value of the next physical action. This method has an elegant theoretical interpretation related to the value of computation.

5.2 Pointed trees

We begin with a substantial theoretical prelude needed for our later discussion of Monte Carlo tree search: we define **pointed trees** (Section 5.2.1), which will represent the internal state of MCTS, as well as **recursive functions** on pointed trees (Section 5.2.2), which will represent policies. In subsequent sections we discuss local operations on pointed trees (Section 5.2.3), how recursive functions on pointed trees can be efficiently maintained after local operations (Section 5.2.4), and how derivatives of these functions can be efficiently computed (Section 5.2.5). These discussion provide the basis for understanding how efficient algorithms that exploit complex internal state can be represented and learned.

5.2.1 Definition and recursive construction

A **pointed tree** is a tree with one selected node, which we will call the **point**. Pointed trees can be constructed in a recursive fashion, following Huet (1997).[1]



(a) A pointed tree (b) See Equation 5.1 (c) See Equation 5.2 (d) See Equation 5.3

Figure 5.1: A pointed tree T and its recursive decomposition into trees, contexts and nodes. Dots correspond to T's `Nodes`, and T's point is the node colored orange. An open square marks the context's hole. Dashed lines connect structures that have been decomposed. See text for details.

By way of example, consider the pointed tree T in Figure 5.1a with point N (colored orange). This can be decomposed (Figure 5.1b) into the node N, N's subtrees (T_1, T_2 and T_3), and N's context C, where C effectively has one leaf with a "hole" corresponding to the

[1] Huet termed the structure a zipper; our `Tree` is his `tree`, our `Context` is his `path`, and our `PointedTree` is his `path` combined with his `location`. Note also that his zippers are binary trees, but the generalization is natural.

original tree's point (N). Denote the fact of this decomposition by the equality:

$$T = \texttt{PointedTree}(C, N, T_1, T_2, T_3). \tag{5.1}$$

The tree T_3 can in turn be decomposed (Figure 5.1c) into its root node N_3 and its two subtrees T_{31} and T_{32}:

$$T_3 = \texttt{Tree}(N_3, T_{31}, T_{32}). \tag{5.2}$$

Finally, denote the parent of C's hole by N_C, and note that C's hole is the second child of N_C (where N_C's children are labeled with their indices in the figure). We can decompose C into a new context C' by removing the tree rooted at N_C, and splitting that tree into its subtrees (Figure 5.1d). Effectively, in going from the lower context C to the upper context C', the original hole can be seen as "moving" up to its parent. In sum, the context C is decomposed into the context C' of the parent node N_C of C's hole, the parent node N_C of C's hole, the number 1 indicating the index in N_C's (zero-indexed) list of children of C's hole, and the other subtrees of N_C, in this case only the one T_{C1}:

$$C = \texttt{Context}(C', N_C, 1, T_{C1}). \tag{5.3}$$

Since each subcomponent is smaller than the original structure, this decomposition will eventually bottom out into a set of `Node`s and the unique empty context, which we denote by $\Diamond$.

A specific class of pointed trees is defined by specifying the `Node` type: what fields does it have? What do they store?

The above equations are instances of the more general constructors:

$\texttt{PointedTree}(C, N, T_1, \ldots, T_k)$	for a context C, a node N and $k \geq 0$ trees T_i
$\texttt{Tree}(N, T_1, \ldots, T_k)$	for a node N and $k \geq 0$ trees T_i
$\texttt{Context}(C, N, i, T_1, \ldots, T_k)$	for a context C, a node N, an index $i \in \{0, \ldots, k\}$ and $k \geq 0$ trees T_i
$\Diamond$	the unique empty context (a constant)

These constructors correspond to the following recursive set definitions for the set of all trees, contexts and pointed trees:

$$\texttt{Tree} = \texttt{Node} \times \bigcup_{k=0}^{\infty} \texttt{Tree}^k$$

$$\texttt{Context} = \{\Diamond\} \cup \texttt{Context} \times \texttt{Node} \times \bigcup_{k=1}^{\infty} k \times \texttt{Tree}^{k-1}$$

$$\texttt{PointedTree} = \texttt{Context} \times \texttt{Node} \times \bigcup_{k=0}^{\infty} \texttt{Tree}^k.$$

For a pointed tree, denote by $\texttt{point}(T)$ the node the pointed tree is currently pointing to. More precisely, this is defined by:

$$\texttt{point}(\texttt{PointedTree}(C, N, T_1, \ldots, T_k)) = N.$$

To make use of pointed trees as a representation useful for metalevel control, we need recursive functions (Section 5.2.2) of pointed trees and local operations on pointed trees (Section 5.2.2). We define these next.

5.2.2 Recursive functions of pointed trees

An X-valued function $f\colon \texttt{PointedTree} \to X$ on pointed trees is **recursive** if its value $f(T)$ at $T \in \texttt{PointedTree}$ is defined by structural recursion on T. This means that f is extendable to a function on pointed trees, trees and contexts:

$$f\colon \texttt{PointedTree} \cup \texttt{Tree} \cup \texttt{Context} \to X \tag{5.4}$$

for which there exist functions:

- $f_{\texttt{Tree}}\colon \texttt{Node} \times X^* \to X$,
- $f_{\texttt{Context}}\colon X \times \texttt{Node} \times \mathbb{N} \times X^* \to X$,
- $f_{\Diamond} \in X$,
- $f_{\texttt{PointedTree}}\colon X \times \texttt{Node} \times X^* \to X$,

which satisfy the recursive equations:

$$\begin{aligned} f(\texttt{Tree}(N, T_1, \ldots, T_k)) &= f_{\texttt{Tree}}(N, f(T_1), \ldots, f(T_k)) \\ f(\texttt{Context}(C, N, i, T_1, \ldots, T_{k-1})) &= f_{\texttt{Context}}(f(C), N, i, f(T_1), \ldots, f(T_k)) \\ f(\Diamond) &= f_{\Diamond} \\ f(\texttt{PointedTree}(C, N, T_1, \ldots, T_k)) &= f_{\texttt{PointedTree}}(f(C), N, f(T_1), \ldots, f(T_k)) \end{aligned}$$

In fact, by structural induction, the functions $f_{\texttt{PointedTree}}$, $f_{\texttt{Tree}}$, $f_{\texttt{Context}}$, $f_{\Diamond}$ suffice to exactly characterize any recursive f.

See Section 5.5.1 for examples of specific recursive functions.

5.2.3 Local operations

Local operations make a local change to a pointed tree, either modifying its nodes or structure or moving its pointer. The following local operations on a pointed tree T will suffice:

- up(T): return the pointed tree with the point moved up to the parent of the point, or T if this is impossible.
- down(T, i) for $i \in \mathbb{N}$: return the pointed tree with the point moved down to the ith child of the current point, or T if there is no ith child.
- modify(T, N') for $N' \in$ Node: return the pointed tree with the node at the point replaced by a new node N'.
- insert(T, i, T') for $i \in \mathbb{N}$ and $T' \in$ Tree: return the pointed tree with T' inserted before the ith child of the current node, or T if i is out of bounds. Note this operation preserves all existing subtrees of the point, and T' becomes the new ith child.

Note that when combined with the method Tree(N) to construct a tree out of a single node N and the empty context $\Diamond$, the above operations can locally construct any pointed tree starting from any one-element pointed tree PointedTree($\Diamond, N$) for $N \in$ Node. (Our implementation of MCTS in Section 5.3 does exactly this, for example.) It will also be convenient to have a method PointedTree(N) that constructs a pointed tree out of a single node N.

Notice that every node in a pointed tree T can be reached by performing a sequence of up and down operations. In fact, the set of all pointed trees T' that can be reached by these operations is isomorphic to the set of nodes of the tree, this isomorphism being witnessed by the function point: PointedTree $\to$ Node.

All these functions have an efficient immutable implementation.[2]

5.2.4 Local operations and recursive functions

Our implementation of Monte Carlo tree search (Section 5.3) will maintain a pointed tree to which it will apply local operations to figure out what to do. It will use functions f of the current value of the pointed tree to decide what to compute. It is imperative that these functions be tractably computable. In particular, after performing a local operation, the time required to compute the next value of the function must be independent of the size of the tree.

Fortunately, the value of recursive functions on pointed trees can be maintained by caching the intermediate values of the recursive computation as messages that are lazily propagated in time independent of the size of the tree. We sketch such a message-passing algorithm below.

Denote the current pointed tree by T, and let f be the X-valued recursive function we wish to incrementally compute. The algorithm extends the Node type to include two fields that store **messages** (see Figure 5.2). Specifically, for a node N:

[2] And an even more efficient mutable implementation.

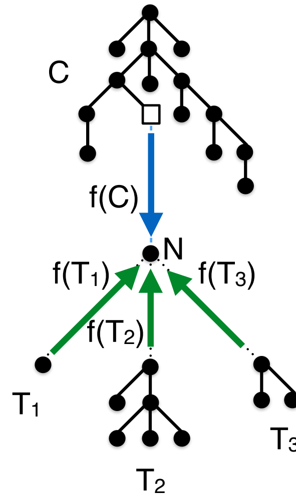


Figure 5.2: Any node N of a pointed tree (not only the point) has a context C surrounding it and subtrees $T_1, \ldots, T_k$ (compare Figure 5.1b). The **down message** (blue) to N is $f(C)$, the function f evaluated at the context surrounding N, and the **up messages** (green) to N are $f(T_1), \ldots, f(T_n)$, the value of function f evaluated at N's child subtrees T_i. The message-passing algorithm extends the `Node` type to include a field `down_msg` storing the (not necessarily valid) down message to the node, and a field `up_msgs` storing the (not necessarily valid) up messages to the node.

- N.`down_msg` stores an X-valued **down message** to N, which when valid contains the value of f evaluated at the context surrounding N, and
- N.`up_msgs` stores a list of X-valued **up messages** to N, the ith entry of which, denoted N.`up_msgs`$[i]$, contains the value of f evaluated at the subtree rooted at the ith child of N, when valid.

Not all of these messages will be valid: this algorithm will lazily propagate them just in time. In fact, it is not possible for all of these messages be valid: local changes cannot be propagated to the rest of the tree in constant time.

However, for every node in the tree, at least one message coming *from* it will be valid, namely the one heading toward the point (Figure 5.3). More formally:

- For every node N of the tree that is not an ancestor of the node `point`(T), excluding the point itself, the up message *from* N to its parent is valid. If N' is the parent of N, i is the index of N in the child list of N, and `Tree`(T, N) is the tree rooted at N in T, then:

$$N'.\texttt{up_msgs}[i] = f(\texttt{Tree}(T, N)). \tag{5.5}$$

- For all the ancestors N of $\texttt{point}(T)$, including the point itself, the down message *to* N from its parent is valid, i.e.:

$$N.\texttt{down_msg} = f(\texttt{Context}(T, N)), \tag{5.6}$$

  where $\texttt{Context}(T, N)$ denotes the context surrounding a node N of T.

Note that no messages from the point itself are required to be valid, and that we are implictly using the extension of recursive functions on pointed trees to recursive functions on pointed trees, trees and contexts (Equation 5.4).

If this invariant holds, the value of f at the current pointed tree is locally computable:

$$\begin{aligned} f(T) &= f(\texttt{PointedTree}(C, N, T_1, \ldots, T_k)) \\ &= f_{\text{PointedTree}}(f(C), N, f(T_1), \ldots, f(T_k)) \\ &= f_{\text{PointedTree}}(N.\texttt{down_msg}, N, N.\texttt{up_msgs}). \end{aligned}$$

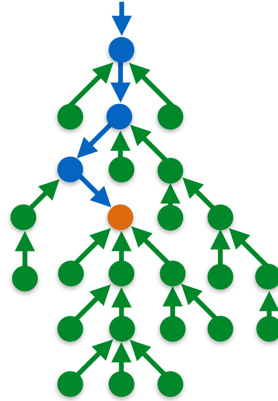
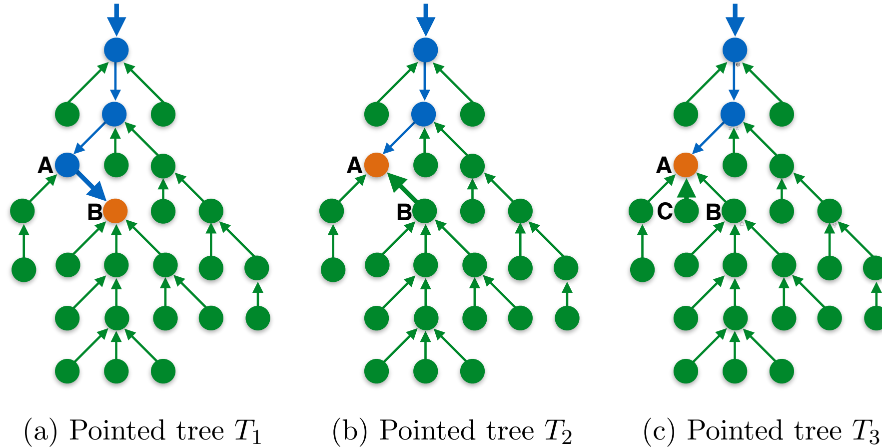


Figure 5.3: A pointed tree T with valid up and down messages indicated as arrows between nodes. (The point node is orange, ancestors are blue and non-ancestors are green.) For every node N of the tree T that is an ancestor of the node $\texttt{point}(T)$ (those nodes in blue), including the point itself, the down message $N.\texttt{down_msg}$ to N (blue arrows) is valid, i.e., equals $f(\texttt{Context}(T, N))$. For every node N of the tree that is not an ancestor of the node $\texttt{point}(T)$ (those nodes in green), including the point itself (if it's not the root), the up message from N (green arrows) is valid, i.e., equals $f(\texttt{Tree}(T, N))$.

To initialize the algorithm for a pointed tree T, the above messages can be recursively computed; since intermediate computations can be shared, it can be seen that the messages are the only values of f that need to be evaluated. In total, one of $f_{\texttt{Tree}}$ and $f_{\texttt{Context}}$ needs to be invoked for each node of the tree (and none for the point). If these can be computed in time linear in the number of their arguments, initialization can be done in time linear in the number of nodes in T.[3]

[3]Recall that the total sum of degrees of nodes in any graph is twice the number of nodes in the graph.



(a) Pointed tree T_1 (b) Pointed tree T_2 (c) Pointed tree T_3

Figure 5.4: When the point in a pointed tree is moved either `up` (e.g., taking T_1 to T_2) or `down` (e.g., taking T_2 to T_1) there is only one message that needs to be computed. To see this, note that T_1 has the down message from node A to node B (the thick blue down arrow) while T_2 does not, and T_2 has the up message from node B to node A (the thick green up arrow) while T_1 does not. If `modify` is used to change the node at the point this affects no messages. If `insert` is used to add a subtree (marked C in figure (c)) already satisfying the invariant to the point (taking T_2 to T_3), the messages in the subtree are all already valid (none of the nodes of the subtree are ancestors of the point) and no other message is invalid.

Suppose a given pointed tree T is annotated with messages satisfying this invariant. If a local operation is applied to yield a new pointed tree T', the invariant can be restored by generating only one new message (see Figure 5.4).

The messages are best maintained by modifying the local operations to transparently update the fields `up_msgs` and `down_msg`, so that algorithms using the pointed tree can have the value of $f(T)$ always efficiently available.

This algorithm can be further refined to track which messages are invalid and update only those that are changed.

Note that many familiar tree algorithms can be reinterpreted, in line with the foregoing discussion, as effectively maintaining the value of a recursive pointed tree function. This is the case with UCT (Kocsis and Szepesvári, 2006), to be discussed in Section 5.3 below.

5.2.5 Derivatives of recursive functions

In order to learn a recursive function, it's important to be able to compute derivatives. Specifically, consider a parameteric family f_θ of recursive functions on pointed trees, for θ a real-valued vector. Let its recursive component functions $f_{\texttt{PointedTree}}$, $f_{\texttt{Tree}}$, $f_{\texttt{Context}}$ and $f_\Diamond$

take θ in as their first argument:

$$\begin{aligned}
f_\theta(\texttt{Tree}(N, T_1, \ldots, T_k)) &= f_{\texttt{Tree}}(\theta, N, f_\theta(T_1), \ldots, f_\theta(T_k)) \\
f_\theta(\texttt{Context}(C, N, i, T_1, \ldots, T_{k-1})) &= f_{\texttt{Context}}(\theta, f_\theta(C), N, i, f_\theta(T_1), \ldots, f_\theta(T_k)) \\
f_\theta(\texttt{PointedTree}(C, N, T_1, \ldots, T_k)) &= f_{\texttt{PointedTree}}(\theta, f_\theta(C), N, f_\theta(T_1), \ldots, f_\theta(T_k)).
\end{aligned}$$

Invoking the chain rule for partial derivatives, we see that if f_θ is a recursive function on pointed trees, contexts and trees, then so too is the combination of f_θ and its derivative $\partial_\theta f_\theta$.[4] Using subscript $\cdot_{,i}$ to denote the partial derivative of a function with respect to its ith parameter:

$$\begin{aligned}
&\partial_\theta f_\theta(\texttt{Tree}(N, T_1, \ldots, T_k)) \\
&\qquad = \partial_\theta f_{\texttt{Tree}}(\theta, N, f_\theta(T_1), \ldots, f_\theta(T_k)) \\
&\qquad = f_{\texttt{Tree},1}(\theta, N, f_\theta(T_1), \ldots, f_\theta(T_k)) \\
&\qquad\quad + \sum_{i=1}^{k} f_{\texttt{Tree},2+i}(\theta, N, f_\theta(T_1), \ldots, f_\theta(T_k))\, \partial_\theta f_\theta(T_i) \qquad (5.7) \\
&\partial_\theta f_\theta(\texttt{Context}(C, N, i, T_1, \ldots, T_{k-1})) \\
&\qquad = \partial_\theta f_{\texttt{Context}}(\theta, f_\theta(C), N, i, f_\theta(T_1), \ldots, f_\theta(T_k)) \\
&\qquad = f_{\texttt{Context},1}(\theta, f_\theta(C), N, i, f_\theta(T_1), \ldots, f_\theta(T_k)) \\
&\qquad\quad + f_{\texttt{Context},2}(\theta, f_\theta(C), N, i, f_\theta(T_1), \ldots, f_\theta(T_k))\, \partial_\theta f_\theta(C) \\
&\qquad\quad + \sum_{i=1}^{k-1} f_{\texttt{Context},4+i}(\theta, f_\theta(C), N, i, f_\theta(T_1), \ldots, f_\theta(T_k))\, \partial_\theta f_\theta(T_i) \qquad (5.8) \\
&\partial_\theta f_\theta(\texttt{PointedTree}(C, T_1, \ldots, T_{k-1})) \\
&\qquad = \partial_\theta f_{\texttt{PointedTree}}(\theta, f_\theta(C), N, f_\theta(T_1), \ldots, f_\theta(T_k)) \\
&\qquad = f_{\texttt{PointedTree},1}(\theta, f_\theta(C), N, f_\theta(T_1), \ldots, f_\theta(T_k)) \\
&\qquad\quad + f_{\texttt{PointedTree},2}(\theta, f_\theta(C), N, f_\theta(T_1), \ldots, f_\theta(T_k))\, \partial_\theta f_\theta(C) \\
&\qquad\quad + \sum_{i=1}^{k-1} f_{\texttt{PointedTree},3+i}(\theta, f_\theta(C), N, f_\theta(T_1), \ldots, f_\theta(T_k))\, \partial_\theta f_\theta(T_i). \qquad (5.9)
\end{aligned}$$

5.3 Monte Carlo tree search (MCTS)

We can now turn to the problem that motivated our introduction of pointed trees: representing the complex internal state of Monte Carlo tree search (MCTS).

[4] The derivative by itself is not a recursive function, since its recursive evaluation requires the evaluation of f. The combination of a function and its first derivative forms what is called a (1-)jet, and this result can be seen as taking advantage of the fact that the composition of a 1-jet is the 1-jet of its composition.

Recall the framework of interacting agents and environments described in Chapter 1, and consider an agent that has, in addition to its input observations and potential output actions, access to a stochastic simulator of the environment. As before, the agent's objective is to maximize the total reward received from the environment before the interaction terminates.

Perhaps the simplest thing to try is the **Monte Carlo approach**: simulate the reward received by taking each action a number of times, and choose the action with greatest average reward. By the law of large numbers, the empiricial average of simulated rewards tends toward the expected reward as the number of simulations increases.

In situations where multiple actions must be taken before receiving the final reward, as is typically the case, this simulation process requires a particular policy for choosing future actions. The easiest approach is to use a fixed stochastic policy, termed a **rollout policy**, the simplest of which takes actions uniformly at random. However, the rollout policy is inaccurate: it doesn't necessarily represent what the agent would actually do in those scenarios. Nor does it improve in any way as the algorithm computes more about the problem it is facing.

Monte Carlo tree search (Section 1.2.3) algorithms address these issues by using the results of simulations to form a lookahead tree to improve the rollout policy near the current state. Approaches vary in how they decide which Monte Carlo simulations to perform, what additional information (if any) they have, and how to use the results of those simulations to decide which action to take.

To specify MCTS, we first describe how we represent interacting agents and environments, and then define the tree-search algorithm, using the pointed trees introduced in Section 5.2 as a means of representing complex internal state.

An agent (for example, the Monte Carlo tree search agent of Algorithm 1) implements two methods:[5]

- `agent.init()`, which initializes any persistent internal state, and
- `agent.act(observations)`, which updates internal state and returns an action.

An environment (for example, Algorithm 2 below) implements two methods:

- `environment.reset()`, which initializes any persistent internal state and returns the first observation to get the alternating interaction started, and
- `environment.step(actions)`, which advances the environment's state and returns a tuple of an observation, reward and termination signal.

MCTSAgent (Algorithm 1) gives a general framework for Monte Carlo tree search using pointed trees for internal state.[6] Recall that the function `point` is defined in Section 5.2.1

[5] Function names and signatures in the following are inspired by those of the OpenAI Gym (Brockman et al., 2016).

[6] MCTS algorithms don't all fit into exactly this framework: in particular, they often add only a single node after a rollout, and might not expand a leaf every time. These are easy extensions.

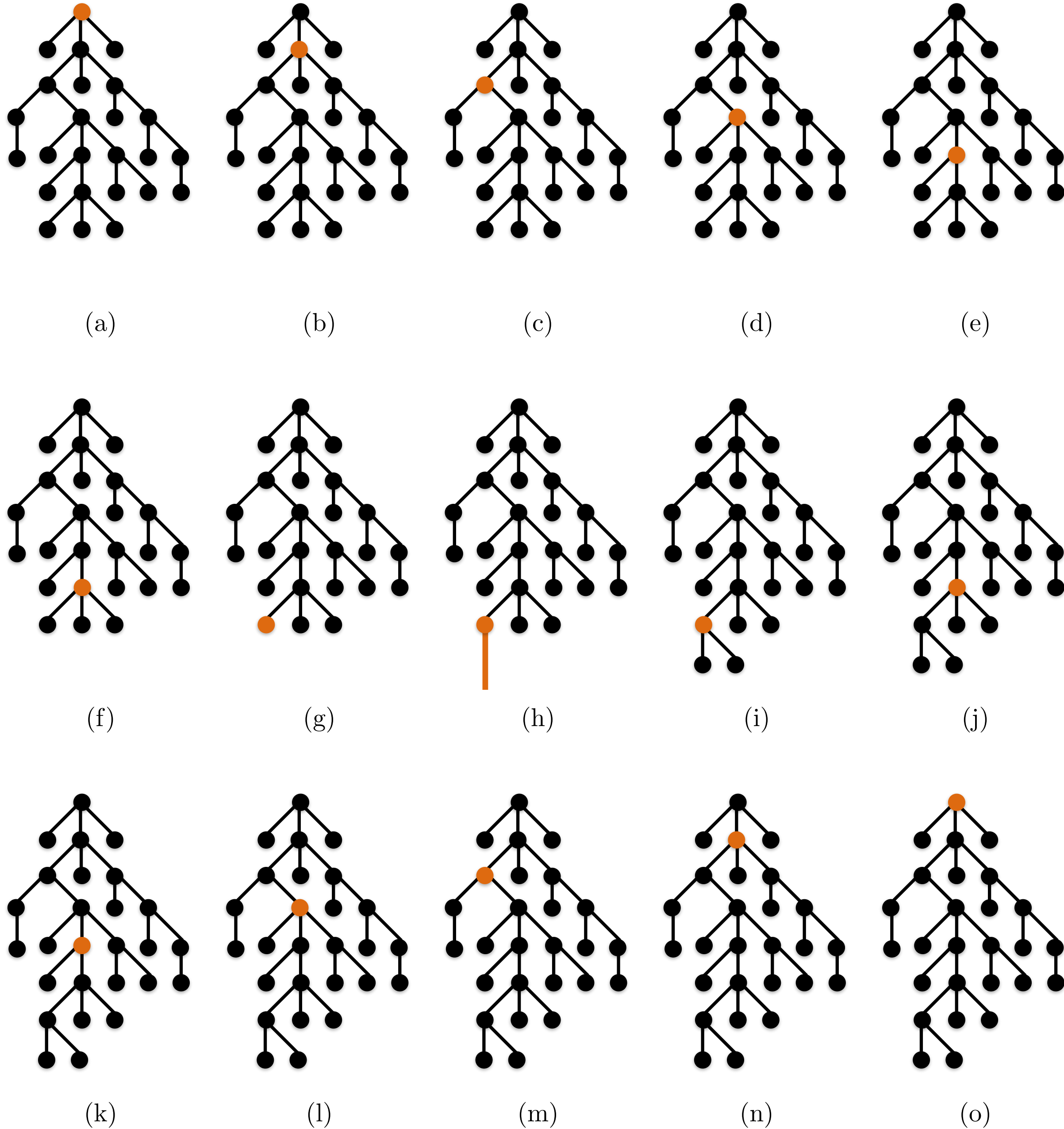


Figure 5.5: A trace of Monte Carlo tree search (Algorithm 1) divided into stages, where the point is colored in orange: (a)-(g) rolling down the tree to a leaf (algorithm line 21), (h) performing a rollout (lines 11-12), (i) expanding a leaf (lines 13-14), and (j)-(o) backing up to the root (lines 15-16). Notice that this takes 15 steps to perform one rollout, of which the first six, (a)-(f), require a choice of a branch into which to descend.

Algorithm 1 Monte Carlo Tree Search Agent

```
 1  MCTSAgent.init()
 2      simulator <- new_simulator()
 3      last_action <- None
 4      last_state <- None

 5  MCTSAgent.act(state)
 6      T <- PointedTree(new_node(last_state, last_action, state))
 7      simulator.reset(state)
 8      last_simulated_action <- None
 9      while not finished(T)
10          if at_leaf(T)
11              score <- simulator.rollout()
12              T <- modify(T, update_node(point(T), last_simulated_action,
                                          simulator.state(), score))
13              for action, next_state in simulator.successors()
14                  T <- insert(T, 0, Tree(new_node(simulator.state(),
                                                   action, next_state)))
15              while not at_root(T)
16                  T <- move_up()
17              simulator.reset(state)
18          else
19              last_simulated_action <- choose_branch(T)
20              simulator.act(action)
21              T <- down(T, action)
22      last_action <- best_action(T)
23      last_state <- state
24      return last_action
```

and the functions `up`, `down`, `modify`, `insert` and `PointedTree` are defined in Section 5.2.3. (See Figure 5.5 for an example execution trace.)

The following functions must be specified by any specific class of MCTS algorithm:

- `new_node(last_state, action, state)`: Create a `Node` corresponding to the given `state`, which was reached by performing `action` in state `last_state`. (In the inital state, both `last_state` and `last_action` are given stub value `None`.)
- `update_node(node, action, state, score)`: Update the value of the given `Node`, which corresponds to the given `state` (which was reached by the given `action`), with the result `score` of a rollout started at `node`.

- `choose_branch(T)`: Choose a branch of `point(T)` to roll down into.
- `finished(T)`: Decide whether to stop and act.
- `best_action(T)`: Decide which action to take.

UCT (Kocsis and Szepesvári, 2006) is a special case of Algorithm 1 where:

- `new_node(last_state, action, state)` and `update_node(node, action, state, score)` maintain for each node `N` the number of times $n_{\texttt{N}}$ that node has been visited by the algorithm, the number of times $n_{\texttt{N},i}$ the ith branch has been taken by the algorithm, and the average future reward $r_{\texttt{N},i}$ received in those simulations.
- `choose_branch(T)` implements the UCB algorithm (Auer et al., 2002) from the theory of multi-armed bandits (see Section 1.2.3): if there are any untried branches, it selects one at random, otherwise selecting the branch i that maximizes

$$r_{\texttt{point(T)},i} + k\sqrt{\frac{\log n_{\texttt{point(T)}}}{n_{\texttt{point(T)},i}}}$$

  for some fixed constant k. The first term favors actions that have performed well so far, the second term favors actions that have been tried fewer times (so their value is more uncertain), and the constant k tunes the balance between these two goals.
- `finished(T)` terminates either when a time limit is reached, or after a given number of simulations.
- `best_action(T)` returns the branch i at the root node `point`(T) of T (note this function is called only when T's point is at its root) of maximal $r_{\texttt{point(T)},i}$.

5.4 MCTS as a metalevel MDP

Observe that the definition of MCTS in `MCTSAgent` (Algorithm 1) can be viewed from the familiar perspective of an agent's actions in the object-level—i.e., its interactions with the external environment. With respect to its internal state representation (the pointed tree), the node manipulation functions `new_node(action, state)` and `update_node(node, action, state, score)`, along with the `simulator`, determine what tree the agent internally searches, while the functions `finished(T)`, `choose_branch(T)` and `best_action(T)` control the agent's computations and actions.

But our overarching goal here is to learn how to control computations, that is, to learn the functions `finished(T)`, `choose_branch(T)` and `best_action(T)`. That is, in the framing of Chapter 1, it is the metalevel agent's (metalevel) actions that we wish to learn. Hence, the particulars of the `MCTSAgent`'s search are not of direct use for the metalevel agent; rather, this metalevel agent interacts with a metalevel environment, which provides as observations

Algorithm 2 Monte Carlo Tree Search Metalevel Environment

```
MCTSMetaEnvironment.reset()
    state <- objectlevel_environment.reset()
    simulator <- new simulator
    simulator.reset(state)
    T <- PointedTree(new_node(None, None, state))
    return T

MCTSMetaEnvironment.step(meta_action)
    if meta_action is a computation
        simulator.step(meta_action)
        T <- down(T, meta_action)
        if at_leaf(T)
            score <- simulator.rollout()
            T <- modify(T, update_node(point(T), meta_action,
                                       simulator.state(), score))
            for action, next_state in simulator.successors()
                T <- insert(T, 0, Tree(new_node(simulator.state(), action,
                                                next_state)))
            simulator.reset(state)
            while not at_root(T)
                T <- move_up()
        return T, 0, False
    else meta_action is an action
        next_state, score, done <- objectlevel_environment.step(meta_action)
        T <- PointedTree(new_node(state, meta_action, next_state))
        simulator.reset(next_state)
        state <- next_state
        if done
            return T, score, done
        else
            return T, 0, False
```

its metalevel state (i.e., the pointed tree T). It also expects to receive (from the metalevel agent) a metalevel action that specifies either a computation to perform (i.e., a branch to select) or an action to take in the underlying object-level environment.

The metalevel environment, defined in `MCTSMetaEnvironment` (Algorithm 2), thus wraps the object-level environment and provides as observation its metalevel state, the pointed tree T, and expects to receive a metalevel action that specifies either a computation to perform

Algorithm 3 Monte Carlo Tree Search Metalevel Agent

```
MCTSMetaAgent.act(T)
    if finished(T)
        return choose_branch(T)
    else
        return best_action(T)
```

(i.e., a branch to select) or an action to take in the underlying object-level environment.

`MCTSMetaAgent` (Algorithm 3) shows how we can construct a metalevel agent from the functions `finished(T)`, `choose_branch(T)` and `best_action(T)`.

In general, a metalevel agent can be described by a single metalevel policy $\pi(T)$ that maps a pointed tree T to the choice of either a branch to roll down or an external action to take.

5.5 Metalevel policy class for MCTS

This section describes UCT and AlphaGo Silver et al. (2016) as metalevel policies for MCTS, then presents the class of metalevel policies we'll use in Chapter 6 to learn how to control MCTS.

5.5.1 UCT and AlphaGo metapolicies

UCT can be specified as a metalevel policy for MCTS in the following fashion. UCT's `Node` has the fields:

- `num_rollouts`: the number of rollouts started from this node. `new_node(last_state, action, state)` initializes this to zero, and `update_node(node, action, state, score)` increments it by one.
- `sum_rollouts`: the sum of the rewards of these rollouts. `new_node(last_state, action, state)` initializes this to zero, and `update_node(node, action, state, score)` increments it by `score`.

Then UCT's policy is defined at the pointed tree $T = \texttt{PointedTree}(C, N, T_1, \dots, T_k)$ by:

$$w_r(T_i) = \sum_{N \in T_i} N.\texttt{sum_rollouts}$$
$$n_r(T_i) = \sum_{N \in T_i} N.\texttt{num_rollouts}$$
$$\pi_{\text{UCT}}(T) = \operatorname*{argmax}_i \frac{w_r(T_i)}{n_r(T_i)} + k\sqrt{\frac{\log \sum_i n_r(T_i)}{n_r(T_i)}}.$$

This is a recursive function on pointed trees, where the up message is $[w_r(T_i), n_r(T_i)]$ and the down message is empty (context is ignored).

Note that the infinities go in the right direction: if nothing has been done yet, pick an action at random; prefer an action that's not yet been selected.

AlphaGo (Silver et al., 2016) has at its core an MCTS algorithm that uses information gained both from rollouts using a carefully trained rollout policy and by evaluating a prior policy and value function.[7]

Let $\pi_0(a|s)$ be the probability distribution of the prior policy, assigning a probability to taking action a in state s, and $V_0(s)$ its value function. AlphaGo's `Node` will store, in addition to UCT's fields, the following:

- `num_value`: the number of times the prior value function is evaluated. `new_node(last_state, action, state)` initializes this to zero, and `update_node(node, action, state, score)` increments it by one.

- `sum_value`: the sum of these evaluations. `new_node(last_state, action, state)` initializes this to zero, and `update_node(node, action, state, score)` increments it by $V_0(\texttt{state})$.

- `prior_prob`: the probability the prior policy would take the action. `new_node(last_state, action, state)` initializes this to $\pi_0(\texttt{action}|\texttt{last_state})$, and `update_node(node, action, state, score)` does nothing.

Then AlphaGo's policy is defined at the pointed tree $T = \texttt{PointedTree}(C, N, T_1, \dots, T_k)$

[7] We bypass the complexities needed to have a parallel implementation of this algorithm: our work here is serial.

by:

$$w_r(T_i) = \sum_{N \in T_i} N.\texttt{sum_rollouts}$$
$$n_r(T_i) = \sum_{N \in T_i} N.\texttt{num_rollouts}$$
$$w_v(T_i) = \sum_{N \in T_i} N.\texttt{sum_value}$$
$$n_v(T_i) = \sum_{N \in T_i} N.\texttt{num_value}$$
$$\pi_{\text{AlphaGo}}(T) = \operatorname*{argmax}_i \ (1-\lambda)\frac{w_v(T_i)}{n_v(T_i)} + \lambda\frac{w_r(T_i)}{n_r(T_i)} + c_{\text{puct}}\,\texttt{point}(T_i).\texttt{prior_prob}\,\frac{\sqrt{\sum_j n_r(T_j)}}{1+n_r(T_i)}$$

for constants λ and c_{puct}. This is a recursive function on pointed trees, where the up message is $[w_r(T_i), n_r(T_i), w_r(T_i), n_r(T_i), \texttt{prior_prob}]$ and the down message is empty (context is ignored).

UCT and AlphaGo are special cases of the metapolicy representation we will use, presented in the next section.

5.5.2 Metapolicy class: Recursive component

Our policies are the composition of two components: a fixed recursive function on trees $f_{\text{fixed}}(T)$ and a parameterized function class g_θ on top of this:

$$\pi_\theta(T) = g_\theta(f_{\text{fixed}}(T)). \tag{5.10}$$

We describe f_{fixed} in this section and g_θ in the next. The major reason for making this decomposition is technical: we use batch reinforcement learning methods that compute and store execution paths and optimize θ over those paths. If we used a general recursive function on trees, we'd have to keep the trees around and perform computation linear in the size of the tree every time the optimization algorithm adjusted θ.

Throughout we assume a deterministic adversarial environment where players move in alternation. By using negamax scoring we treat each player symmetrically.

Similar to the previous section, we first describe the fields our `Node`s are annotated with, then the recursive functions on pointed trees we use. We follow by describing how the messages are computed.

Our `Node`s have the following fields:

- AlphaGo fields: `num_value`, `sum_value`, `prior_prob`, `num_rollouts`, `sum_rollouts`.

- `expanded?`: whether this node has been expanded, 1 if it has, 0 if it hasn't.

We incrementally maintain the following functions of `Tree`s T:

- $\texttt{num_visits}(T)$: total number of simulations that passed through the root of T.
- $\texttt{all_done}(T)$: whether all leaves of T are expanded, 1 if true, 0 if false.
- $\texttt{avg_r}(T)$: the average value to the current player of rollouts resulting from simulations passing through T.
- $\texttt{avg_v}(T)$: the average evaluation to the current player of leaves expanded by simulations passing through T.
- $\texttt{p_over_n+1}(T)$: the prior probability of the root of T divided by the number of visits incremented by one.
- $\texttt{negamax}(T)$: the value to the current player derived from evaluations of leaves in T, evaluated by negamax.

which satisfy the following recurrences for a tree $T = \texttt{Tree}(N, T_1, \ldots, T_k)$:

$$\texttt{num_visits}(T) = N.\texttt{num_rollouts} + \sum_i \texttt{num_visits}(T_i)$$

$$\texttt{all_done}(T) = \begin{cases} N.\texttt{expanded?} & \text{if } k = 0, \\ \max_i \texttt{all_done}(T_i) & \text{otherwise.} \end{cases}$$

$$\texttt{avg_r}(T) = \begin{cases} \dfrac{N.\texttt{sum_rollouts} - \sum_i \texttt{num_visits}(T_i)\,\texttt{avg_r}(T_i)}{N.\texttt{num_visits}} & \text{if } k > 0, \\ 0 & \text{otherwise.} \end{cases}$$

$$\texttt{avg_v}(T) = \begin{cases} \dfrac{N.\texttt{prior_value} - \sum_i \texttt{num_visits}(T_i)\,\texttt{avg_v}(T_i)}{N.\texttt{num_visits}} & \text{if } k > 0, \\ 0 & \text{otherwise.} \end{cases}$$

$$\texttt{p_over_n+1}(T) = \frac{N.\texttt{prior_probability}}{N.\texttt{num_visits} + 1}$$

$$\texttt{negamax}(T) = \begin{cases} N.\texttt{prior_value} & \text{if } k = 0, \\ \max_i -\texttt{negamax}(T_i) & \text{otherwise.} \end{cases}$$

We also incrementally maintain the following functions of `Context`s C:

- $\texttt{depth}(C)$: the depth of the context's hole.
- $\alpha(C)$: the alpha value of this context, a lower bound on the utility we can achieve: we can achieve at least this much somewhere else in the tree.

- $\beta(C)$: the beta value of this context, an upper bound on the utility we can achieve: our opponent can achieve at most this much somewhere else in the tree.

which satisfy the following recurrences:

$$\begin{aligned}
\texttt{depth}(\Diamond) &= 0 \\
\texttt{depth}(\texttt{Context}(C, N, i, T_1, \ldots, T_k)) &= 1 + \texttt{depth}(C) \\
\alpha(\Diamond) &= -1 \\
\beta(\Diamond) &= 1 \\
\alpha(\texttt{Context}(C, N, i, T_1, \ldots, T_k)) &= -\beta(C) \\
\beta(\texttt{Context}(C, N, i, T_1, \ldots, T_k)) &= -\max(\alpha(C), \max_i -\texttt{negamax}(T_i))
\end{aligned}$$

At a pointed tree $T = \text{PointedTree}(C, N, T_1, \ldots, T_k)$, the fixed component $f_{\text{fixed}}(T)$ packages the value of the above functions together into a vector (see the red vector at the top of Figure 5.6a):

- `node`: various features of the node N to be elaborated in Section 6.1.2.
- `down`: the down message from the context: $\texttt{depth}(C)$, $\alpha(C)$, and $\beta(C)$.
- `up-i`: for each subtree T_i the up message from it: $\texttt{num_visits}(T)$, $\texttt{all_done}(T)$, $\texttt{avg_r}(T)$, $\texttt{avg_v}(T)$, $\texttt{p_over_n+1}(T)$, $\texttt{negamax}(T)$.

In practice, we maintain the value of $f_{\text{fixed}}(T)$ by further wrapping the MCTS metalevel environment; see Section 6.1.2 for details.

5.5.3 Metapolicy class: Parameterized local component

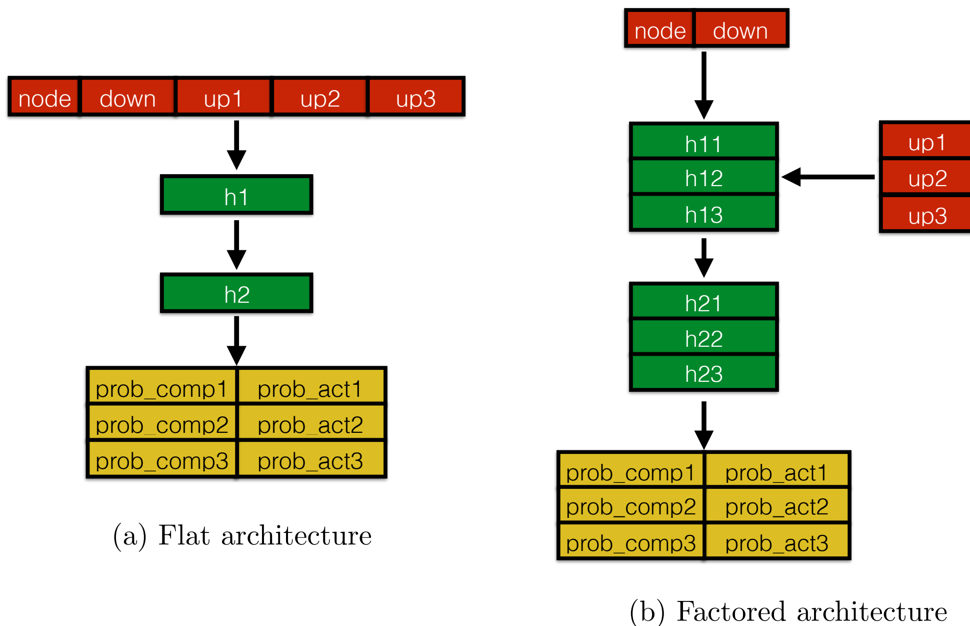
Our second component, the parameterized local component g_θ, maps the output of the fixed recursive component, $f_{\text{fixed}}(T)$ to a probability distribution over meta-actions, i.e., over actions and computations. Concretely, the output distribution is a $\texttt{max_branching} \times 2$ matrix of probabilities. We compare two different neural network architectures for implementing g_θ (Figure 5.6). The first is a straightfoward architecture, and the second makes use of symmetries in the metalevel MDP (namely, repeated up messages coming from different branches) to improve scaling.

1. **Flat** (Figure 5.6a). This maps the flattened $f_{\text{fixed}}(T)$ to the distribution array without reshaping. We use dense linear mappings with tanh activation into the hidden layers, and a dense linear mapping with softmax to the output layer.
2. **Factored** (Figure 5.6b). This utilizes structure in $f_{\text{fixed}}(T)$ by splitting it into two components:
    - The context: node vector, and the down message.

- The up messages: reshaped into a `max_branching` × `up_msg_size` array.

The up messages are linearly mapped to form the first hidden layer, a `max_branching` × `h1_size` dimension array, the ith up message mapping to the ith row, all rows being mapped by the same linear map. The context is mapped linearly into an `h1_size` dimensional array which is added to each component of the second hidden layer. A tanh activation is then applied.

The rest of the layers are mapped in a similarly uniform manner, applying tanh activation at hidden layers and softmax at the output.



(a) Flat architecture

(b) Factored architecture

Figure 5.6: Neural network architectures for the local component of the metalevel policy (see Section 5.5.3). Input layer(s) in red, output layer in yellow, hidden layers in green. See the text for descriptions of the specific transformations between layers.

5.6 Metalevel shaping rewards

Reward shaping (Dorigo and Colombetti, 1994; Ng et al., 1999) is a method for addressing reward sparsity by giving the learning process intermediate rewards. Rewards are especially sparse in metalevel MDPs, where computations yield no intrinsic reward, and there may be millions of computations before any external action.

In general, applying shaping rewards runs the risk of distorting the objective and changing which policies are optimal, but potential-based shaping rewards do not (Ng et al., 1999). As

their name suggests, they are defined from a potential ϕ function over states. They give as intermediate rewards the change in the state's potential after a transition.[8] Ng et al. (1999) show that the optimal shaping potential for an MDP is the true value function, and suggest that approximations of the value function will be good shaping rewards.

The metalevel environment described in this chapter can be seen as an instance of a **joint-state MDP** (Russell and Wefald, 1991a; Parr and Russell, 1998; Andre and Russell, 2002): one whose state has two components: the state of the object-level environment and the internal state of the agent. Denote the former by w (the world state) and the latter by s. The full state of the metalevel environment is then given by the **joint state** (w, s).

One lower bound on the value of a joint state is the value of acting immediately without further computation. The value of this is not known exactly, but algorithms often maintain an approximation, such as `avg_r`(T) and `avg_v`(T) from Section 5.5.2.

Denoting by $\hat{Q}(w, a|s)$ such an approximation, our shaping reward is

$$\phi(w, s) = \max_a \hat{Q}(w, a|s). \tag{5.11}$$

Denote the unshaped reward function R by:

$$\begin{aligned} R(w, s; a; w', s) &= R_{\mathcal{W}}(w; a; w') \\ R(w, s; c; w, s') &= 0. \end{aligned}$$

Then the shaped reward function R_ϕ is:

$$\begin{aligned} R_\phi(w, s; a; w', s) &= \left[R_{\mathcal{W}}(w; a; w') + \max_{a'} \hat{Q}(w', a'|s')\right] - \max_a \hat{Q}(w, a|s) \\ R_\phi(w, s; c; w, s') &= \max_a \hat{Q}(w, a|s') - \max_a \hat{Q}(w, a|s) \end{aligned}$$

This shaped reward has an interesting interpretation. The shaped reward after a physical action a is the difference between how much reward the system expected to get before knowing the next reward and how much reward the system expects to get after knowing the next reward. Letting $a^*(w|s) = \operatorname{argmax} \hat{Q}(w, a|s)$, the expected shaped reward under computations can be decomposed into two terms:

$$\begin{aligned} &\sum_{s'} T_S(s, c, s') R_\phi(w, s; c; w, s') \\ &\quad = \sum_{s'} T_S(s, c, s') \left[\hat{Q}(w, a^*(w|s')|s') - \hat{Q}(w, a^*(w|s)|s)\right] \\ &\quad = \sum_{s'} T_S(s, c, s') \left[\hat{Q}(w, a^*(w|s')|s') - \hat{Q}(w, a^*(w|s)|s')\right] \\ &\qquad - \left[\hat{Q}(w, a^*(w|s)|s) - \sum_{s'} T_S(s, c, s') \hat{Q}(w, a^*(w|s)|s')\right] \end{aligned}$$

[8] Note that if there are terminal states, the potential must have the same value on all terminal states, otherwise what is optimal can be changed.

The first component measures the (necessarily non-negative) expected improvement in decision quality resulting from performing computation c: it equals the estimated utility difference between the action $a^*(w|s')$ that would be chosen having performed the computation c and the action $a^*(w|s)$ that would be chosen having not performed it, both measured using post-computation value estimates $\hat{Q}(w, a|s')$. This is closely related to the value of information. The second component measures the failure of $\hat{Q}(w, a|s')$ to behave like a conditional expectation.

Having established how Monte Carlo tree search, along with an appropriate representation of the metalevel state, can be used to define a metalevel MDP, we can now investigate how metalevel reinforcement learning can be used to learn to control MCTS computations. Experiments applying these techniques are given in Chapter 6.

Chapter 6

Experiments

6.1 Experimental setup **77**
6.1.1 Object-level environment: Hex 77
6.1.2 Metalevel environment 79
6.1.3 Calibrating UCT 79
6.1.4 Reinforcement learning 80
6.2 Experimental results **80**
6.2.1 Flat architecture 80
6.2.2 Factored architecture 82
6.2.3 Factored architecture initialized to UCT 83
6.2.4 Metalevel reward shaping 84
6.3 Discussion **88**

We set out to investigate whether metalevel reinforcement learning can successfully learn to control Monte Carlo tree search. Section 6.1 describes the practical details for our experiments and elaborates how the approach outlined in Chapter 5 can be applied to the board game Hex. Section 6.2 presents the experiments and analyzes their results, and Section 6.3 ends with discussion.

6.1 Experimental setup

6.1.1 Object-level environment: Hex

We used as our object-level environment for MCTS the board game Hex (Figure 6.1). In Hex, the players alternately fill in empty tiles, trying to connect opposite sides with tiles of their color. The first player wins by connecting the top side to the bottom side, while the second player wins by connecting the left side to the right side. Full-size Hex is 11×11,

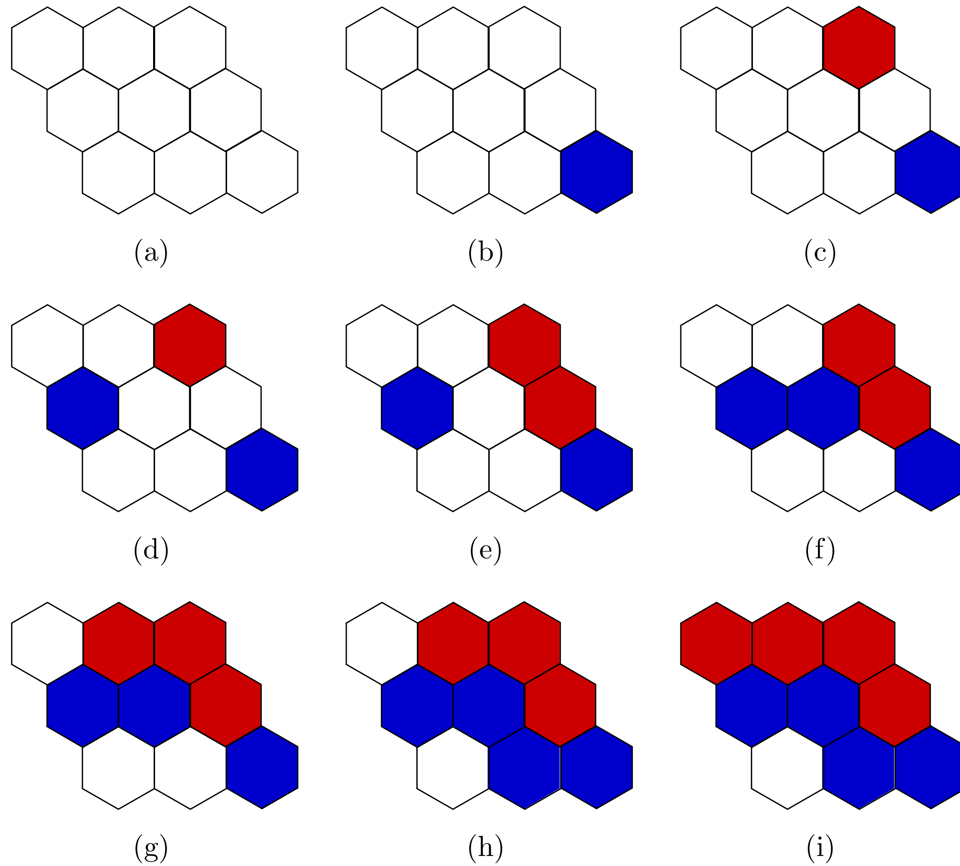


Figure 6.1: A game of 3×3 Hex. Players alternate filling empty (hexagonal) tiles, with the first player colored in blue, the second in red. The goal of both players is to create a path of contiguous tiles in their respective colors between two of the edges: top and bottom for the first player (blue), left and right for the second (red). In this game the second player (red) wins.

but we use odd-sized boards from 3×3 upwards to control the difficulty of the object-level problem, denoting the board size by L.

In our experiments, the state of the Hex environment is not directly observable to the metalevel policy.

Hex has a first-player advantage in practice, and can be shown to have a winning strategy in theory. For this reason, we randomize which player gets to play first in order to remove this asymmetry and normalize our results: an agent playing against itself will receive zero average reward (see Table 6.1 for illustration).

6.1.2 Metalevel environment

Our metalevel environment, which we name `MCTSEnv`, wraps `MCTSMetaEnvironment` (Algorithm 2 of Section 5.5) to compute the recursive functions $f_{\text{fixed}}(T)$ defined in Equation 5.10 of Section 5.5.2. The local parameterized policy g_θ is a function of the observation vector defined at the end of Section 5.5.3. Specifically, we used the following:

- `can_act`: 1 if it's valid to act, 0 otherwise. It is valid to act when at the root and after having performed the minimum number of computations.

- `can_comp`: 1 if it's valid to compute, 0 otherwise. It is valid to compute when not at the root or when it has not yet performed the maximum number of computations.

- `node_vector`: vector of derived quantities of the node. In our case, this equals $[\sqrt{\texttt{num_visits}}, 1, \texttt{is_root?}]$ (this is in order to represent AlphaGo, and p-UCT specifically, within this class).

- `messages`: flattened $(1+\texttt{max_actions}) \times (1+\texttt{message_size})$ array. This constant-sized array stores a variable number of messages. The first row is the down message, the rest are up messages. The first column is a validity bit marking whether that message exists: 1 if it exists, 0 otherwise. This validity bit is needed to pass a fixed-sized vector into the neural network of the metalevel policy even while the branching factor is variable. Invalid messages are zeroed out.

The metalevel policy masks its action distribution to exclude invalid choices: performing a physical action in a non-root state before the preset number of computations is performed, performing a computation after the preset number of computations is performed, or choosing an action or a branch that does not exist (an artifact of mapping into an observation vector of uniform size).

6.1.3 Calibrating UCT

We use UCT (Section 5.5.1) as the opponent for the following experiments, and also as a baseline to measure the performance of our learned policies. UCT has a free parameter k (see Section 5.5.1), and its performance may be sensitive to its value.

To investigate UCT's sensitivity to varying this parameter, we conducted a preliminary experiment in which games of Hex were played using UCT as the metalevel policy for both players—effectively allowing UCT to play against itself. We varied k between 0 and 100 for both players independently.

Table 6.1 shows the average reward for these pairings, along with the reward against an opponent following a random policy. Positive valued entries are bolded, and indicate cases where the player outperforms its opponent. In the later tables of this section, bold entries will show cases where our learned policy outperforms the baseline. The more bold entries, the better the learning method.

	k=0	k=0.1	k=0.2	k=0.5	k=1	k=2	k=5	k=10	k=100	random
k=0	**0.03**	-0.12	-0.16	-0.30	-0.31	-0.24	-0.04	-0.15	**0.01**	**0.44**
k=0.1	**0.14**	**0.02**	-0.03	-0.11	-0.12	-0.06	**0.01**	**0.06**	**0.08**	**0.51**
k=0.2	**0.14**	**0.01**	-0.03	-0.12	-0.09	-0.07	**0.02**	**0.06**	**0.07**	**0.58**
k=0.5	**0.25**	**0.06**	**0.07**	-0.02	-0.05	-0.01	**0.13**	**0.08**	**0.17**	**0.57**
k=1	**0.29**	**0.13**	**0.08**	**0.02**	**0.02**	**0.01**	**0.17**	**0.17**	**0.23**	**0.66**
k=2	**0.21**	**0.07**	**0.07**	-0.02	-0.04	-0.03	**0.11**	**0.10**	**0.16**	**0.61**
k=5	**0.15**	-0.08	-0.09	-0.14	-0.14	-0.13	-0.02	-0.01	**0.06**	**0.53**
k=10	**0.09**	-0.06	-0.05	-0.16	-0.21	-0.12	-0.01	-0.01	**0.04**	**0.56**
k=100	**0.01**	-0.12	-0.13	-0.16	-0.24	-0.17	-0.06	-0.04	-0.01	**0.52**
random	-0.46	-0.52	-0.55	-0.63	-0.65	-0.58	-0.55	-0.53	-0.52	**0.02**

Table 6.1: Average reward of UCT vs. UCT varying the weight k of the upper confidence term, with $L = 3$ board size, $n = 20$ computations, averaged over 1000 random games. The final column/row give the performance against random game play for calibration. Observe that $k = 1$ performs no worse than the rest.

6.1.4 Reinforcement learning

In our experiments we applied TRPO with GAE (Section 2.4) for reinforcement learning,[1] using settings that have been demonstrated to be robust: $\gamma = 0.995$, $\lambda = 0.97$, $\max_{KL} = 0.01$, and conjugate gradient damping factor of 0.1. We used a batch size of $50,000$ steps throughout. (See Schulman et al. (2015, 2016) for the semantics of these parameters.)

For the flat policy architecture (Figure 5.6a), we used 64 hidden units, while for the factored policy architecture (Figure 5.6b), we used 8 hidden units.

6.2 Experimental results

6.2.1 Flat architecture

	n=10	n=20	n=50	n=100
L=3	**0.69**	**0.31**	-0.03	-0.29
L=5	**0.68**	-0.03	-0.80	-0.79
L=7	**0.31**	-0.07	-0.42	-0.17

Table 6.2: Average reward for the flat policy architecture (Figure 5.6a) as a function of Hex board size L and number of computations n. The performance of UCT against itself averages 0.0, so the learned metalevel policy outperforms UCT for small number of computations.

[1]We used the reference implementation available online at `https://github.com/joschu/modular_rl`.

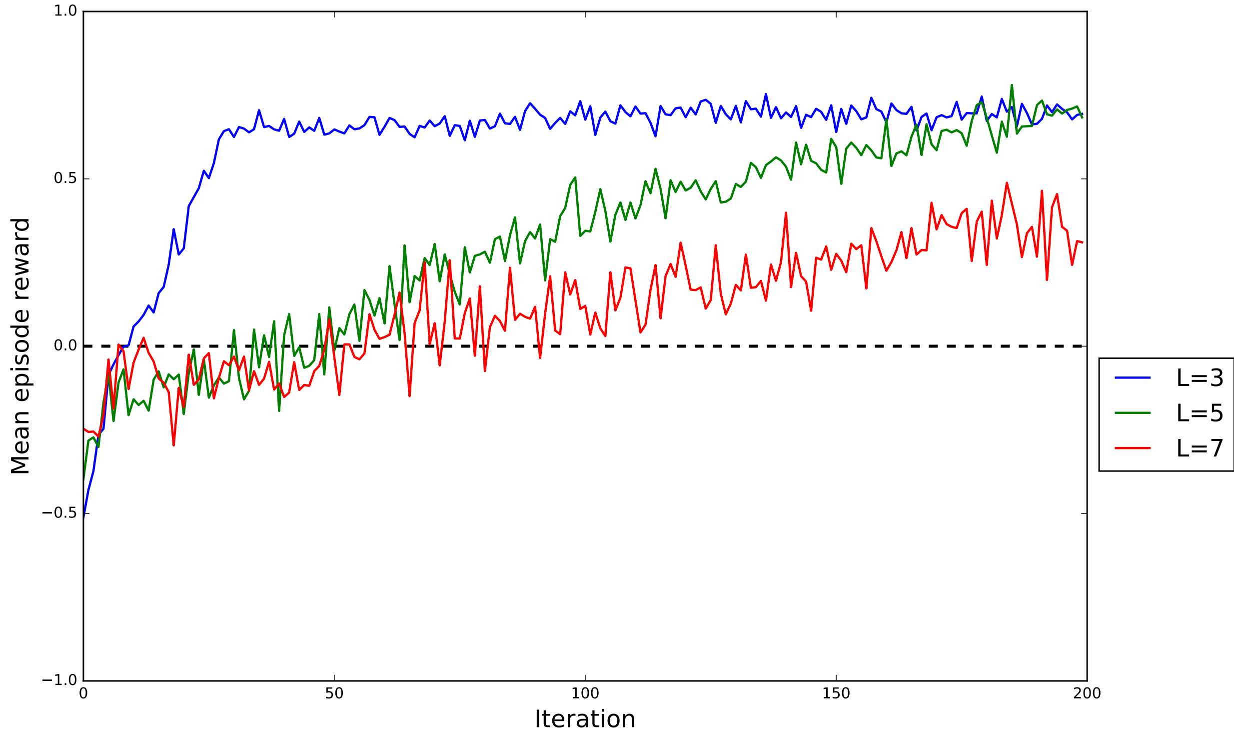


Figure 6.2: Average reward as a function of learning iteration for the flat policy architecture (see Figure 5.6a), varying the board size L for constant number of computations $n = 10$. The metalevel policy outperforms UCT for all the board sizes, but learning is slower for larger boards (see text).

We first investigated reinforcement learning in the metalevel environment `MCTSEnv` (Section 6.1.2) using the flat policy class architecture (Figure 5.6a) to determine what quality metalevel policies it would learn. Table 6.2 shows the results for varying board size and number of computations. RL successfully learns metalevel policies outperforming UCT for varying board size if the number of computations is small, outperforming UCT for $n = 10$ (Figure 6.2), and at least matching for $n = 20$. The performance drops off for larger number of computations and larger board size.

We considered three potential reasons for this performance decrease:

1. The number of parameters of the flat architecture scales with the square of the board size L: the observation vector (Section 6.1.2) is of size $12 + 7L^2$ and the first layer of the flat architecture (Figure 5.6a) is a dense layer with $(12 + 7L^2) \times 64$ parameters.

2. A randomly initialized policy acts effectively at random. As the number of computations available to the opponent increases, and as a larger board size reduces the chance of blind luck, the initial performance of the learned policy decreases. If the policy starts out losing every game, it has no information to learn from.

3. Increasing the board size or number of computations increases the average length of each episode and decreases the number of episodes per batch (Table 6.3). This makes learning more difficult by increasing the chance of overfitting to peculiarities of those particular episodes, by making the influence of any given computation on overall reward more indirect, and by decreasing the information available to the learner (there is only one non-zero reward: at the end of an episode indicating a win or a loss).

	n=10	n=20	n=50	n=100
L=3	49.02	131.49	540.77	996.12
L=5	127.88	370.30	1239.29	2690.58
L=7	288.23	714.04	2146.38	4389.08

	n=10	n=20	n=50	n=100
L=3	1019.94	380.27	92.46	50.19
L=5	390.98	135.03	40.35	18.58
L=7	173.47	70.02	23.30	11.39

Table 6.3: Average length of episode (above) and number of episodes per batch (below) as a function of board size L and number of computations n. Notice the two orders of magnitude difference between $L = 3, n = 10$ and $L = 7, n = 100$, and the absolutely few number of episodes per batch in the latter case. These numbers are for the learned flat policies (see Table 6.2), but the numbers are consistent across metalevel policies.

The following sections propose improvements to help address these issues: Section 6.2.2 addresses policy parameters scaling with the square of the board size; Section 6.2.3 addresses the initial policy's poor performance; Section 6.2.4 addresses sparsity of reward.

6.2.2 Factored architecture

	n=10	n=20	n=50	n=100
L=3	**0.60**	**0.44**	-0.59	-0.60
L=5	**0.50**	**0.48**	-0.42	-0.91
L=7	**0.35**	**0.50**	-0.63	-0.54

Table 6.4: Average reward for the factored policy architecture (Figure 5.6b) as a function of Hex board size L and number of computations n. Note the wider range of settings relative to the flat policy class in which the learned metalevel policy outperforms UCT.

The factored architecture (Figure 5.6b) maps each up message through the same network, making the number of weights invariant to the board size. Table 6.4 shows the performance of RL using the factored architecture.

Compared to the flat architecture (Table 6.2), the factored architecture now uniformly outperforms UCT for $n = 20$ computations. However, it still fails to find a good policy for larger numbers of computations ($n = 50, 100$).

The following section improves upon this situation by jump-starting the learned policy.

6.2.3 Factored architecture initialized to UCT

	n=10	n=20	n=50	n=100
L=3	**0.58**	**0.47**	**0.47**	**0.33**
L=5	**0.70**	**0.60**	**0.51**	-0.89
L=7	**0.37**	**0.47**	**0.36**	-0.45

Table 6.5: Average reward for the factored policy architecture initialized to UCT as a function of Hex board size L and number of computations n. The learned policy outperforms UCT for up to 50 computations per move, on all board sizes.

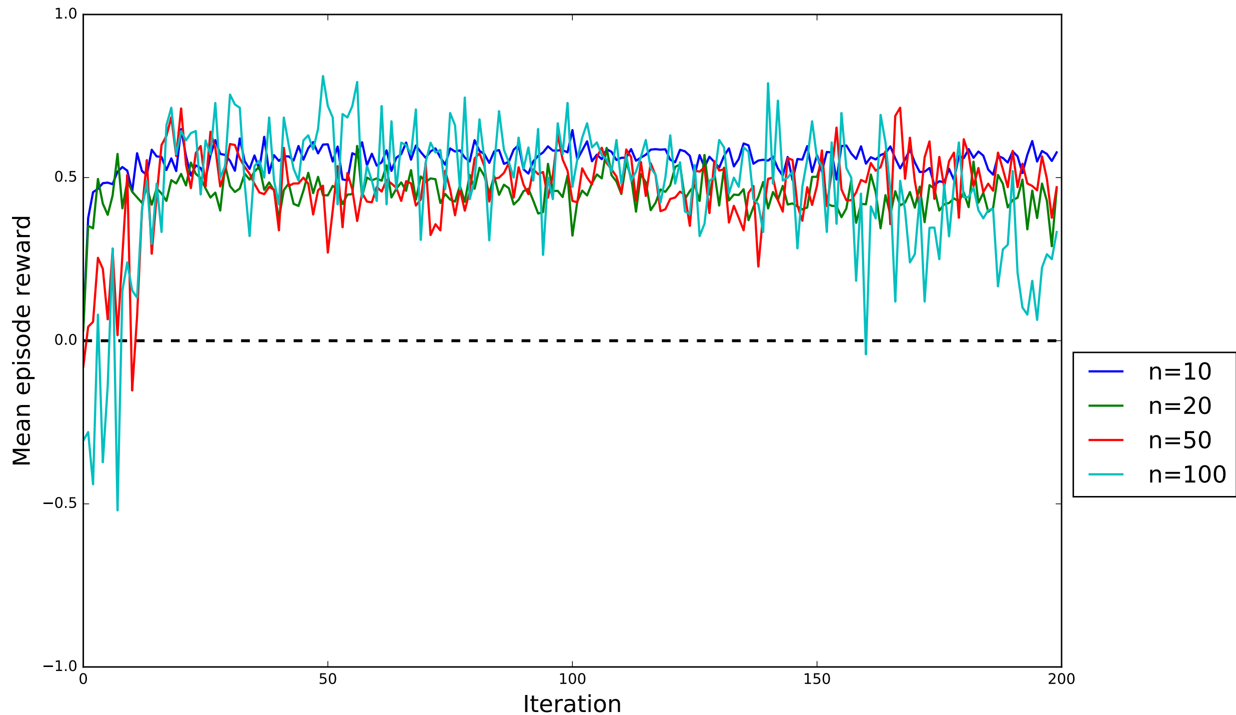


Figure 6.3: Learning curve for the factored policy architecture initialized to UCT for board size $L = 3$ and varying number of computations n. Learning succeeds in all cases, although the $n = 100$ curve is less robust than the others.

UCT falls within the factored policy class by design. This means we can initialize the weights of the network to implement UCT, then continue learning. Since UCT already performs well against itself, with an expected performance of 0, this should significantly improve learning.

Table 6.5 shows what results. The learned policy now outperforms UCT for up to 50 computations per move, on all board sizes. (See Figure 6.3 for learning curves.) Learning fails in two settings, $n = 100$ with $L = 5, 7$, finding a policy worse than its initialization. Table 6.3 suggests a reason: in these settings there are fewer than 20 episodes per batch, yielding at most 20 bits of information to the learning algorithm per batch, not enough to even maintain the initially good weight assignment.

L=3	n_{test}=10	n_{test}=20	n_{test}=50	n_{test}=100
n_{train}=10	0.46	0.02	-0.52	-0.96
n_{train}=20	0.42	0.34	-0.98	-1
n_{train}=50	0.4	0.4	0.48	0.68
n_{train}=100	0.42	0.42	0.24	0.22

L=5	n_{test}=10	n_{test}=20	n_{test}=50	n_{test}=100
n_{train}=10	0.6	0.46	0.22	-0.1
n_{train}=20	0.6	0.58	0.56	0.22
n_{train}=50	0.58	0.62	0.64	0.54
n_{train}=100	-0.52	-0.82	-0.98	-0.84

Table 6.6: Average reward for the factored policy architecture initialized to UCT, systematically varying the number of computations used in training (n_{train}) and the number of computations used in testing (n_{test}). See text for discussion.

How robust is the policy found by RL? Does a policy learned for $n = 10$ computations generalize to one for $n = 50$ computations, and vice versa? Table 6.6 shows the results of an experiment run to investigate this question, in which we systematically vary the number of computations at training and test time. We find that policies trained for larger numbers of computations generalize to smaller number of computations but not the reverse (as indicated by performance increasing as we move along a row to the left). This makes sense, as the process of constructing a good $n = 50$ tree, for example, produces a good $n = 10$ tree along the way, but a policy constructing an $n = 10$ tree has never seen a larger tree.

6.2.4 Metalevel reward shaping

The third issue identified in Section 6.2.1 was that of reward sparsity. Recall from Section 5.6 that *reward shaping* (Dorigo and Colombetti, 1994; Ng et al., 1999) is a method for addressing reward sparsity by giving the learning process intermediate rewards.

We investigated using the shaping reward proposed in Section 5.6. This shaping reward shapes by an estimate of the utility to be gained by stopping and acting immediately, equal

to the maximum over the estimated utilities of the actions available in the current world state. We estimate the utility of a particular action by the average reward avg_r of all the rollouts that begin with that action. Concretely, if w is the current world state, s is the current metalevel state (the pointed tree), $n_a^{+1}(s)$ the number of rollouts starting with action a at the root yielding reward +1, and $n_a^{-1}(s)$ the number of rollouts starting with action a at the root yielding reward -1, avg_r estimates the value of the action by:

$$\hat{Q}_1(w, a|s) = \begin{cases} \dfrac{n_a^{+1}(s) - n_a^{-1}(s)}{n_a^{+1}(s) + n_a^{-1}(s)} & \text{if } n_a^{+1}(s) + n_a^{-1}(s) > 0, \\ 0 & \text{otherwise.} \end{cases}$$

This gives the shaping potential:

$$\phi_1(w, s) = \max_a \hat{Q}_1(w, a|s).$$

Table 6.7 shows the results. In fact, the performance for this shaped cases is *worse* than that in the unshaped case (Table 6.7). Examining the shaping rewards before and after training (Figure 6.4) shows the reason for this failure. Because the average rollout after a single success has value 1, the maximum possible reward, the shaping reward quickly hits its maximum, giving no further signal. The metalevel policy learns to hit this maximum as soon as possible. This, for example, gives it a bias toward not resampling an action that has succeeded once, because its estimated utility can only go down.

A more accurate estimate of an action's utility, such as one that regressed toward the mean for estimates based on few samples, would not have this weakness because it would be roughly equally likely to go up as down upon further computation. The Beta-Bernoulli metalevel control problem (recall Section 3.1) suggests such an estimate: model the samples gained from rollout as Bernoulli samples of an underlying probability of success (i.e., reward +1 rather than -1) with a Beta prior, and use the posterior mean. Concretely:

$$\hat{Q}_2(w, a|s) = \frac{n_a^{+1}(s) - n_a^{-1}(s)}{n_a^{+1}(s) + n_a^{-1}(s) + 2}$$
$$\phi_2(w, s) = \max_a \hat{Q}_2(w, a|s).$$

The shaping rewards and potentials before and after training (Figure 6.5) are much more reasonable, with no pathological clamping. The results in Table 6.8 suggest that for smaller numbers of computations, the shaped reward scales better with number of actions.

These results are not yet as good as we'd like. Ng et al. (1999) argues that good shaping potentials closely approximate the value function, and the above approximations are still quite crude. Better approximations of the value function of Beta-Bernoulli model of Section 3.1 may well yield more powerful shaping rewards.

	n=10	n=20	n=50	n=100
L=3	**0.54**	**0.31**	-0.10	-0.77
L=5	**0.71**	**0.66**	**0.58**	-0.33
L=7	**0.61**	**0.55**	-0.42	-0.43

Table 6.7: Average reward for the factored policy architecture initialized to UCT and shaped by the maximum estimated action-utility estimated by the average reward of roll-outs ($\phi_1(w, s)$; see text) as a function of Hex board size L and number of computations n. Compared with the unshaped case (Table 6.5) this shows *worse* performance. See text for explanation.

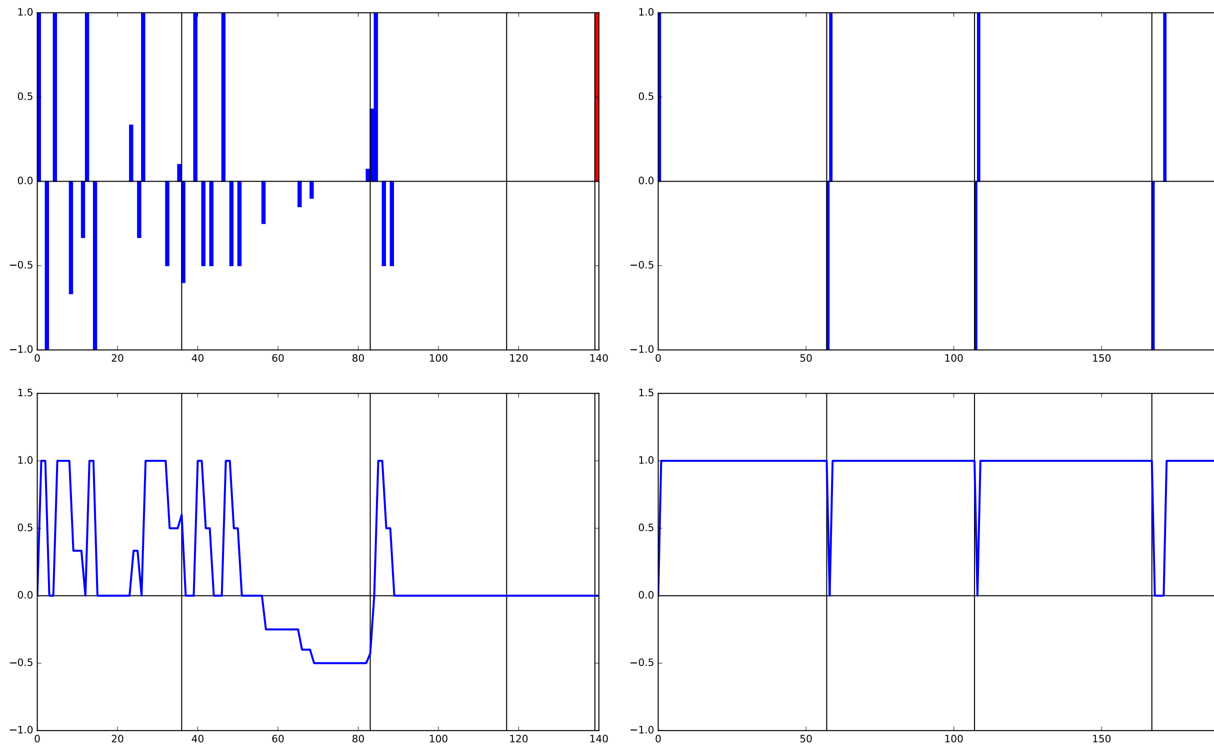


Figure 6.4: Traces over an episode for the factored policy architecture, with board size $L = 3$ and number of computations $n = 20$. The top plots the shaping reward at each time step (blue bars) along with the single final reward (red bar). The bottom plots the shaping potential $\phi_1(w, s)$ over time: the shaping rewards equal the change in potential. The left side is before training, the right side is after training. Note that the maximum value of the shaping potential is 1, corresponding to a certain win. See text for discussion.

	n=10	n=20	n=50	n=100
L=3	**0.55**	**0.45**	**0.34**	-0.74
L=5	**0.73**	**0.77**	0.00	0.00
L=7	**0.61**	**0.92**	-0.67	-0.43

Table 6.8: Average reward for the factored policy architecture initialized to UCT and shaped by the maximum estimated action-utility estimated by the Beta-posterior ($\phi_2(w, s)$; see text) as a function of Hex board size L and number of computations n. Compared with the unshaped case (Table 6.5) this shows *worse* performance. See text for explanation.

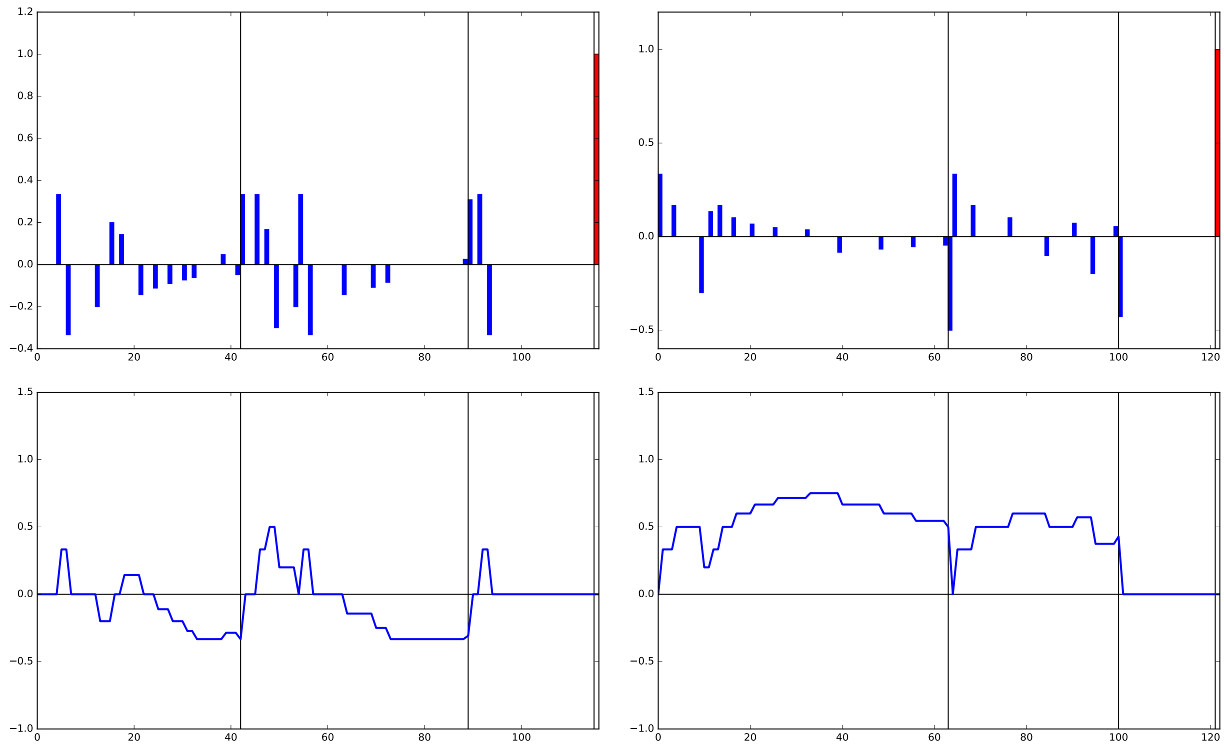


Figure 6.5: Traces over an episode for the factored policy architecture, with board size $L = 3$ and number of computations $n = 20$. The top plots the Beta-posterior shaping reward at each time step (blue bars) along with the single final reward (red bar). The bottom plots the shaping potential $\phi_2(w, s)$ over time: the shaping rewards equal the change in potential. The left side is before training, the right side is after training. Note that that unlike the trace for ϕ_1, this does not immediately clamp at +1. See text for further discussion.

6.3 Discussion

Section 6.2.1 demonstrated that reinforcement learning in the metalevel environment can successfully learn metalevel policies that outperform hand-crafted policies like UCT. However, straightforward approaches have issues scaling to larger domains and greater numbers of computations. We identified three broad issues: the number of parameters in a flat architecture scales with the maximum number of actions, in an adversarial setting poor initial policies may greatly hinder learning, and the reward sparsity of the metalevel MDP.

The first two issues were addressed with the factored policy architecture, which includes UCT within its parameter space, allowing learning to start from UCT and explore a continuous space of variants.

For the third issue, our use of reward shaping was not successful, but we outlined a direction for improvement through shaping potentials based on approximate value functions of simpler metalevel MDPs, such as the Beta-Bernoulli metalevel MDP.

Further directions for resolving the issue of reward sparsity include exploring environments in which the agent selects computations for only a single real-world action, with either an oracle policy (a high-quality policy for the domain) that determines other actions or an oracle value function (an accurate object-level value function) that evaluates the quality of the decision. Such an approach would provide feedback after every object-level action, increasing the information available to the learning process. A similar improvement may result simply from shaping the object-level environment with an accurate object-level value function.

In sum, the above experiments show that metalevel reinforcement learning can find metalevel policies that outperform hand-crafted metalevel policies. The reward sparsity and recursive structure of computations makes this an exceptionally difficult learning problem, providing scope and promise for future work.

Chapter 7

Conclusions

7.1 Understanding metalevel control **89**
7.1.1 Mechanical model 90
7.1.2 Bayesian model 91
7.1.3 Complementary models 91
7.2 Summary **92**
7.3 Future work **93**
7.4 Parting thoughts **95**

We have come some way toward a principled foundation for addressing the questions we began with: How can an agent learn to control its own decision-making process? How can it know when it's better to think a bit more before acting, and for how long? How much more computation does it need to perform—or information does it need to gather—before choosing a course of action?

These questions, and the answers offered in the previous chapters, reflect different ways of conceptualizing the complex problems at hand. We conclude our investigations by stepping back to consider the conceptual models we have used to understand the metalevel control problem (Section 7.1). We then revisit our contributions in light of these distinctions (Section 7.2) and suggest potentially fruitful directions for continued progress (Section 7.3). We end with a few remarks on the broader context of the ideas explored here (Section 7.4).

7.1 Understanding metalevel control

Discussions of metalevel agents, metalevel control, metalevel decision-making and metareasoning are bound to invite confusion. We talk about, and think about, such abstract

concepts in a variety of ways that are not always consistent.[1] The notion of metalevel control constructed here also depends on many other equally complex concepts, which can be organized in different ways for different modeling goals and approaches.

This section makes explicit the conceptual models adopted in this work, in the hope of dispelling potential terminological and conceptual confusion. Broadly speaking, the theoretical and practical studies reflect two conceptual models of metalevel control, which we will term the *mechanical* model and the *Bayesian* model. We discuss each of these in turn (Sections 7.1.1 and 7.1.2) and compare them (Section 7.1.3) below.

7.1.1 Mechanical model

The **mechanical model** views computations as actions that change the *internal* state of the agent. These contrast with *external* actions that change the state of the environment, but they are not fundamentally different. From this perspective, the metalevel control problem is likewise not fundamentally different from the (non-metalevel) control problem of choosing among possible actions: choose the computation-actions that maximize the overall reward received.

The mechanical model is exemplified in our treatment of MCTS as a metalevel environment (recall Sections 5.3 and 5.4 and Figure 5.5). The internal state of the MCTS agent is a pointed tree, i.e., a tree with a pointer at a particular node. Computations are actions that move this pointer up and down the tree, propagating information along with it, and performing random rollouts at the leaves.

The mechanical model has two broad variants. One takes an *internal* perspective, adding computations to the existing set of external actions available to an agent. This relates to the everyday experience of choosing to either do something or think about doing something. The other takes an *external* perspective, in which the agent is limited to its existing set of external actions, while a separate meta-agent is responsible for the computations that direct the agent. (On this view, the agent is part of the environment, relative to the meta-agent.) This view is akin to separating ourselves into a part that acts and thinks (the agent), and a part that chooses *what* to think (the meta-agent).

Formally, these two variants are closely related. The mechanical model can be formalized as a joint-state MDP (Section 5.6). A **joint policy** for this joint-state MDP is a function $\pi\colon \mathcal{W} \times \mathcal{S} \to \mathcal{A} \cup \mathcal{C}$ taking joint states (w, s) to either an action $a \in \mathcal{A}(w)$ or a computation $c \in \mathcal{C}(s)$. This joint policy can be subdivided into three sub-policies:

- a **stopping policy** $\pi_{\text{STOP}}\colon \mathcal{W} \times \mathcal{S} \to \{\text{COMP}, \text{ACT}\}$ that decides whether to compute more or stop and act,
- a **computational policy** $\pi_{\text{COMP}}\colon \mathcal{W} \times \mathcal{S} \to \mathcal{C}$ that decides what specific computation to perform, and

[1] They are in this regard much like other abstract concepts that are understood as metaphorical extensions of more concrete domains (Lakoff and Johnson, 1980).

- an **action policy** $\pi_{\text{ACT}}\colon \mathcal{W} \times \mathcal{S} \to \mathcal{A}$ that decides what specific action to take.

In the internal variant, a single agent chooses actions and computations, and so specifies the joint policy π. In the external variant, the action policy π_{ACT} is part of the agent, while the stopping π_{STOP} and computational policies π_{COMP} are part of the meta-agent.

7.1.2 Bayesian model

The **Bayesian model** views computations as providing Bayesian evidence relevant for decision-making. The state of the agent records the evidence it has accumulated so far. The metalevel control problem, in this framing, is a sequential information-gathering problem: choose the computations that yield the most relevant information at the least cost.

The formal problems defined in Chapter 3—the Beta-Bernoulli metalevel control problem (Section 3.1) and metalevel control problems (MCP) in general (Section 3.2)—directly reflect this Bayesian perspective. In MCPs, computations provide evidence about the utility of different actions. The result of a computation and the utility of an action are both uncertain, and are thus both modeled as random variables distributed according to some prior. In the Beta-Bernoulli metalevel control problem, the state of the agent is a representation of the posterior distribution over the action's utilities conditional on the simulations observed so far. This is the minimal record of evidence provided by these simulations.

The Bayesian model builds upon the mechanical model by giving the agent's states and computations a Bayesian semantics. This semantics makes it clear what use computations have: they provide information relevant to making the decision.

7.1.3 Complementary models

The models serve different and complementary purposes.

The Bayesian model is analytically tractable, as Chapters 3 and 4 illustrate. It allows us to formalize important intuitions about computation, such as: the results of a computation are not known before it's performed; computations provide information relevant to deciding how to act, including especially information about the relative utility of actions; the agent's state integrates the results of the computations performed.

The mechanical model is more general: any algorithm class can be understood in terms of the mechanical model; the metalevel control problem can thus be posed and good policies can be learned, as Chapter 5 and 6 demonstrate. Further, any Bayesian model can be seen as a mechanical model by integrating out the probabilities, as illustrated in the conversion of an MCP to a metalevel MDP (Definition 3.9).

Results and understanding gained through the Bayesian model can be transferred to the mechanical model. For example, the Bayesian model allows analytic approximations to the value function of a problem; such value functions are useful candidates for shaping rewards (see Section 6.2.4).

While the different perspectives demand different analytical tools and practical techniques, a clear understanding of their conceptual compatibility enables us to benefit from the insights of both views without confusion.

7.2 Summary

We can now summarize the contributions of the thesis with respect to the different conceptual bases for metalevel control just delineated.

As noted above, the **Bayesian perspective** provided an ideal setting for the theoretical analysis of metalevel control, as pursued in Chapters 3 and 4.

- We formalized the problem of controlling computations by defining metalevel control problems (MCPs), which captures the view of computation as information-gathering.
- We characterized the conditions under which these MCPs have an equivalent MDP that we can then analyze using the theory of MDPs.
- We combined the notions of factored and metalevel MDPs, and showed how one can derive properties of the factored MDP from properties of its component factors, in particular bounding the number of computations of the optimal policy of the factored MDP by the sum of the bounds on the number of computations of optimal policies in its factors.
- We showed that the context of an action can centrally influence what it is optimal to compute, but we demonstrated limits on what this influence can be.

For the practical experiments described in Chapter 5 and 6, the more general **mechanical perspective** proved more appropriate. This perspective allows a wider range of applications, since an explicit Bayesian model is not required.

- We showed how to represent Monte Carlo tree search as a metalevel MDP, and how to represent metalevel policies for controlling such MDPs.
- We observed that using the formalism of pointed trees to represent the internal state of Monte Carlo tree search allows us to define an efficient class of policies. Specifically, we showed that recursive functions on pointed trees can be efficiently computed in an incremental fashion by message-passing. UCT, among other MCTS algorithms, can be seen from this perspective as implementing this message-passing algorithm to compute a particular recursive function on pointed trees.
- We proposed concrete classes of parameterized recursive functions on pointed trees and demonstrated that reinforcement learning by policy optimization can learn metalevel policies that outperform fixed algorithms like UCT.

- We introduced two classes of recursive functions on trees, flat and factored, and further showed that the factored representation outperforms the flat representation.
- The factored representation includes UCT as a special case, and we showed that initializing the metalevel policy to UCT allows metalevel reinforcement learning to learn faster and scale to a larger number of computations.
- Finally, we began an exploration of how metalevel shaping rewards can help address the reward sparsity problem, although further work is required to make this approach a uniform success.

7.3 Future work

This thesis opens up a number of avenues for exploration, from both the mechanical and Bayesian perspectives. Some concrete next steps suggested by the foregoing:

1. Let a **coherent** metalevel MDP $M = (\mathcal{S}, \mathcal{A}, T, \mu, d, R)$ be one whose utility estimates are coherent (cf. Definition 4.1), i.e., such that for all $i = 1, \ldots, k$, $s \in \mathcal{S}$ and $c \in \mathcal{C}$:

$$\mu_i(s) = \sum_{s'} T(s, c, s')\mu_i(s'). \tag{7.1}$$

   One can show that if S_t is a Markov chain realizing the above MDP, then $\mu_i(S_t)$ is a martingale. Under reasonable conditions, e.g., if μ is bounded, standard results of martingale theory (Kallenberg, 2006) can be used to show there exists a random variable U_i such that $\mu_i(S_t) = \mathbb{E}[U_i|S_t]$. Further, one can make this construction uniform, defining U_i independently of the metalevel policy used. Can this be extended to show that coherent metalevel MDPs correspond exactly to Markov stationary MCPs, or are further conditions required?

2. The treatment of factored MDPs (Sections 2.3 and 3.5) may be made more precise by defining a factored MDP as one that is **equivalent** to the composition of other MDPs, as alluded to in Definition 2.7. It would then be a theorem that the k-action Beta-Bernoulli MMDP is equivalent to the composition of k copies of the 1-action Beta-Bernoulli MMDP. This theorem could strengthen the analysis of the previous point, by showing, for instance, that converting a coherent MMDP into an MCP and back again yields an equivalent MDP. There are a number of ways to define equivalent MDPs, but the key properties will likely be analogs of Lemmas 2.10 and 2.11.

3. Theorem 4.8, bounding the number of computations performed in the 1-action Beta-Bernoulli MMDP by the inverse cost $O(1/d)$, can be straightforwardly extended to normal distributions. How generally does it hold? Can it be proven for exponential families with conjugate priors, under suitable conditions to exclude Example 4.4? The

posterior in this case can be parameterized by two terms: the weight ($\alpha + \beta$ in the Beta-Bernoulli case) and the sum of observations (α in the Beta-Bernoulli case). The weight increases with the number of samples observed. Can the weight play the role of $\alpha + \beta$ in an exponential-family version of Theorem 4.8? The result of Yu (2011, Theorem 2), that the value of a two-armed bandit with arms distributed according to an exponential family decreases in the weight, may be relevant here.

4. Metalevel control problems and MDPs are defined for the one-shot case, where there is only a single external action to take. These concepts can be extended to the sequential case in a number of ways. Which way is most theoretically tractable, allowing analogous results to those in Chapters 3 and 4? One approach replaces the U_i of Definition 3.1 with random variables $R_{w,a}$ and $\hat{Q}_{w,a}$ for world states $w \in \mathcal{W}$ and external actions $a \in \mathcal{A}(w)$ applicable in them. These variables each take up one of the two roles that U_i plays in the one-shot theory. The cumulative discounted sum of $R_{w,a}$ defines the actual object-level reward the agent received, using the definition of the value of the policy (cf. Equation 3.8). The estimated value $\hat{Q}_{w,a}$ (similar to the estimated utility model of Russell and Wefald (1991a)) specifies what the agent uses to select external actions once it decides to stop computing. The motivation for separating these two cases is similar to Russell and Wefald (1991a): ensuring the agent always has a well-defined external action to take when computation stops.

   One might want to consider, were it tractable, defining $\hat{Q}_{w,a}$ as a Bellman-like fixed point:
$$\hat{Q}_{w,a} = R_{w,a} + \gamma \sum_{w' \in \mathcal{W}} T_{\mathcal{W}}(w, a, w') \max_{a'} \hat{Q}_{w',a'}, \tag{7.2}$$
   for discount ratio γ. Note, however, that this assumes that the future agent will act *optimally*, selecting the action that does in fact maximize the cumulative reward. For a bounded agent this is false, so a more tractable model of future actions may be required.

5. Section 5.2 shows how the derivative of a recursive function on pointed trees can be computed efficiently. It would be very interesting to extend the policy class given in Section 5.5 to a fully recursive class and extend the results of Chapter 6. Note that when the parameters θ of a fully recursive function f_θ are changed, all its messages need to be recomputed, which requires time proportional to the size of the tree. However, when a function f_θ is evaluated over a sequence of trees generated by applying a succession of local operations to an empty tree, this function can be computed efficiently (recall 5.2.4).

6. Metalevel reinforcement learning (Chapters 5 and 6) can in principle be applied to any metalevel control problem, not just that of controlling Monte Carlo tree search. Metalevel reinforcement learning may be especially powerful where the agent has a range of computations whose value may depend on subtle conditions that are best

learned. Particularly interesting applications include the control of sampling in influence diagrams (Pearson, 2006), hierarchical lookahead search (Marthi et al., 2009) and it may be possible to apply metalevel reinforcement learning more generally to any partially specified agent program, perhaps by adapting programming languages designed for specifying partial policies in hierarchical reinforcement learning (Parr and Russell, 1998; Andre and Russell, 2002; Marthi et al., 2005).

7.4 Parting thoughts

Spending time thinking *meta-* encourages one to reflect, and to reflect upon one's reflections. And to reflect upon how long and in what manner one should reflect upon whatever it is one should be reflecting on.[2] (This might take a while.)

One observes, in this vein, a general transformation that takes a *something* at the object-level to a *something* at the metalevel. Object-level control of an environment to maximize utility transforms into metalevel control of an agent to maximize utility. Object-level uncertainty about the environment's response to an action transforms into metalevel uncertainty about the result of a computation. Object-level learning to control action transforms into metalevel learning to control thought.

This thesis is an application of the same pattern of transformation. The theoretical portion repurposes Bayesian decision theory to model and understand metalevel control. The practical portion repurposes reinforcement learning to learn metalevel control. There's a lot more object-level work ripe for being transformed, and a lot more work to do at the metalevel.

As for the (metametalevel) choice of potential paths that warrant exploration, our collective attention may do well to look to biological instances of metalevel control for inspiration. For example, Swanson (2000, 2012) describes how the non-cerebral parts of the brain form what he terms the **behavioral control column** that can sense and act on its own. In many non-mammalian vertebrates it largely does. The cerebral hemispheres (the metalevel) systematically map onto the behavioral control column (the object-level) and modulate its activity. Examining exactly how the cerebral hemispheres modulate and control the behavioral control column may yield insight into metalevel control: evolution may have stumbled upon tricks that we've yet to discover.

Reflecting more broadly, the impact of AI on society has increasingly become a subject of attention, both within the field and in the public at large. In the near term, autonomous vehicles, armed (Russell, 2016b) or otherwise, and the economic and social impact of increased job automation loom large. In the long term, the possibility of AI systems meeting and exceeding human intelligence has become a topic of serious consideration (Good, 1965; Yudkowsky, 2008; Bostrom, 2014; Russell et al., 2015b; Russell, 2016a). Although such a possibility remains distant and speculative, the potential magnitude of its effects—beneficial and otherwise—warn against procrastination, and a number of tractable research directions

[2]This sort of thinking risks infinite regress, which for the sake of the reader we've carefully sidestepped.[1]

are being actively pursued (Russell et al., 2015a; Hadfield-Menell et al., 2017; Amodei et al., 2016; Taylor et al., 2016). With such work we can help ensure that the benefits of AI continue to robustly outweigh its costs.

Returning to the present, which is where we always already are, we hope the reader has correctly calculated how long to spend with this thesis, and has gained maximal benefit at reasonable cost.

Bibliography

Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE transactions on pattern analysis and machine intelligence*, 12(2):182–193, 1990.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Man. Concrete Problems in AI Safety, 2016.

David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, pages 119–125, 2002.

J.Y. Audibert, S. Bubeck, and R. Munos. Best arm identification in multi-armed bandits. In *COLT*, 2010.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 2002.

R.E. Bechhofer. A single-sample multiple decision procedure for ranking means of normal populations with known variances. *The Annals of Mathematical Statistics*, 25(1):16–39, 1954.

Robert E. Bechhofer, Thomas J. Santner, and David M. Goldsman. *Design and analysis of experiment for statistical selection, screening, and multiple comparisons*. Wiley, 1995.

D.A. Berry and B. Fristedt. *Bandit problems: sequential allocation of experiments*. Chapman and Hall London, 1985.

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

Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1):49–107, 2000.

Bruno Bouzy and Bernard Helmstetter. Monte Carlo Go developments. In *Advances in Computer Games*, pages 159–174. Springer, 2004.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016. URL `http://arxiv.org/abs/1606.01540`.

Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in multi-armed bandits problems. In *International conference on Algorithmic learning theory*, pages 23–37. Springer, 2009.

Alan Carlin and Shlomo Zilberstein. Decentralized Monitoring of Distributed Anytime Algorithms. In *Proceedings of the Tenth International Conference on Autonomous Agents and Multiagent Systems*, pages 157–164, Taipei, Taiwan, 2011. URL `http://rbr.cs.umass.edu/shlomo/papers/CZaamas11.html`.

S.E. Chick and P. Frazier. Sequential Sampling with Economics of Selection Procedures. *Management Science*, 2012.

Stephen E Chick, Jurgen Branke, and Christian Schmidt. New greedy myopic and existing asymptotic sequential selection procedures: preliminary empirical results. In *2007 Winter Simulation Conference*, pages 289–296. IEEE, 2007.

T. Dean and Mark Boddy. An analysis of time-dependent planning. In *AAAI-88*, pages 49–54, 1988.

M. Dorigo and M. Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence*, 71(2):321–370, 1994.

J. Doyle. Artificial intelligence and rational self-government. Technical Report CMU-CS-88-124, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1988.

M.R. Fehling and J.S. Breese. A computational model for the decision-theoretic control of problem solving under uncertainty. In *Proceedings of the Fourth Workshop on Uncertainty in Artificial Intelligence, Minneapolis, MN: AAAI*, 1988.

Peter Frazier and Warren Powell. The knowledge gradient policy for offline learning with independent normal rewards. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, 2007.

Peter Frazier and Warren B Powell. The knowledge-gradient stopping rule for ranking and selection. In *2008 Winter Simulation Conference*, pages 305–312. IEEE, 2008.

Peter Frazier, Warren Powell, and Savas Dayanik. The knowledge-gradient policy for correlated normal beliefs. *INFORMS journal on Computing*, 21(4):599–613, 2009.

Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 2011.

J.C. Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 148–177, 1979.

I. J. Good. A five-year plan for automatic chess. In Ella Dale and Donald Michie, editors, *Machine Intelligence 2*, pages 89–118. Oliver and Boyd, 1968.

I. J. Good. Twenty-seven principles of rationality. In V. P. Godambe and D. A. Sprott, editors, *Foundations of Statistical Inference*, pages 108–141. Holt, Rinehart, Winston, 1971.

Irving John Good. Speculations Concerning the First Ultraintelligent Machine. *Advances In Computers*, 6(99):31–83, 1965.

Dylan Hadfield-Menell, Anca Dragan, Pieter Abbeel, and Stuart Russell. Cooperative Inverse Reinforcement Learning. In *Advances in Neural Information Processing Systems 25*. MIT Press, 2017.

Eric A. Hansen and Shlomo Zilberstein. Monitoring Anytime Algorithms. *SIGART Bulletin Special Issue on Anytime Algorithms and Deliberation Scheduling*, 7(2):28–33, 1996. URL `http://rbr.cs.umass.edu/shlomo/papers/HZsigart96.html`.

Daishi Harada. Reinforcement Learning with Time. In *Proc. Fourteenth National Conference on Artificial Intelligence*, pages 577–582. AAAI Press, 1997.

Daishi Harada and Stuart Russell. Learning search strategies. In *AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*, 1999.

Nicholas Hay, Stuart Russell, Solomon Eyal Shimony, and David Tolpin. Selecting Computations: Theory and Applications. In *Proc. UAI-12*, 2012.

Eric Horvitz. Reasoning about Beliefs and Actions under Computational Resource Constraints. In *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence*, pages 429–444, 1987.

Eric Horvitz. Models of continual computation. In *AAAI/IAAI*, pages 286–293, 1997.

Eric Horvitz. Principles and applications of continual computation. *Artificial Intelligence*, 126(1):159–196, 2001.

Eric Horvitz and Geoffrey Rutledge. Time-dependent utility and action under uncertainty. In *Proceedings of the Seventh Conference on Uncertainty in Artificial Intelligence*, pages 151–158. Morgan Kaufmann Publishers Inc., 1991.

Eric J Horvitz, Gregory F Cooper, and David E Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *IJCAI*, pages 1121–1127, 1989.

Eric Joel Horvitz. *Computation and action under bounded resources.* PhD thesis, stanford university, 1990.

Ronald A Howard. Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, 1966.

Gérard Huet. The zipper. *Journal of functional programming*, 7(05):549–554, 1997.

Olav Kallenberg. *Foundations of modern probability*. Springer Science & Business Media, 2006.

Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 2002.

Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. *ECML*, 2006.

Akshat Kumar and Shlomo Zilberstein. Anytime Planning for Decentralized POMDPs using Expectation Maximization. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 294–301, Catalina Island, California, 2010. URL `http://rbr.cs.umass.edu/shlomo/papers/KZuai10.html`.

George Lakoff and Mark Johnson. *Metaphors we live by*. University of Chicago Press, 1980.

F William Lawvere and Stephen H Schanuel. *Conceptual mathematics: a first introduction to categories*. Cambridge University Press, 2009.

Bhaskara Marthi, Stuart Russell, and David Latham. Writing Stratagus-playing agents in concurrent ALisp. *IJCAI-05 Workshop on Reasoning, Representation, and Learning in Computer Games*, 2005.

Bhaskara Marthi, Stuart Russell, and Jason Wolfe. Angelic hierarchical planning: Optimal and online algorithms (revised). Technical Report UCB/EECS-2008-150, EECS Department, University of California, Berkeley, 2009.

James E Matheson. The economic value of analysis and computation. *Systems Science and Cybernetics*, 4:325–332, 1968.

Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proc. Sixteenth International Conference on Machine Learning*. Morgan Kaufmann, 1999.

Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, pages 1043–1049, 1998.

Mark Pearson. Utility-Directed Sampling in Influence Diagrams. Master's thesis, UC Berkeley, 2006.

M.L. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, Inc., 1994.

Howard Raiffa and Robert Schlaifer. *Applied Statistical Decision Theory*. M.I.T. Press, 1968.

Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527–535, 1952.

R Tyrrell Rockafellar. *Convex Analysis*. Princeton University Press, 1979.

Stuart Russell. Rationality and intelligence. *Artificial intelligence*, 94(1-2):57–77, 1997.

Stuart Russell. Rationality and intelligence: A brief update. In *Fundamental Issues of Artificial Intelligence*, pages 7–28. Springer, 2014.

Stuart Russell. Should We Fear Supersmart Robots? *Scientific American*, 314(6):58–59, 2016a.

Stuart Russell. Robots in war: the next weapons of mass destruction? *World Economic Forum*, January 2016b.

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

Stuart Russell and Eric Wefald. *Do The Right Thing*. The MIT Press, 1991a.

Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 1991b.

Stuart Russell, Daniel Dewey, and Max Tegmark. Research Priorities for Robust and Beneficial Artificial Intelligence. *AI Magazine*, 36(4), 2015a.

Stuart Russell, Tom Dietterich, Eric Horvitz, Bart Selman, Francesca Rossi, Demis Hassabis, Shane Legg, Mustafa Suleyman, Dileep George, and Scott Phoenix. Research Priorities for Robust and Beneficial Artificial Intelligence: An Open Letter. *AI Magazine*, 36(4), 2015b.

Stuart J Russell and Devika Subramanian. Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, 2:575–609, 1995.

Stuart J. Russell and Eric H. Wefald. Decision-theoretic control of search: General theory and an application to game-playing. Technical Report UCB/CSD 88/435, Computer Science Division, University of California, Berkeley, 1988a.

Stuart J. Russell and Eric H. Wefald. Multi-Level Decision-Theoretic Search. In *Proceedings of the AAAI Spring Symposium Series on Computer Game-Playing*, Stanford, California, 1988b. AAAI.

Stuart J. Russell and Eric H. Wefald. On optimal game-tree search using rational metareasoning. In *Proc. Eleventh International Joint Conference on Artificial Intelligence*, pages 334–340, Detroit, 1989. Morgan Kaufmann.

Stuart J Russell and Shlomo Zilberstein. Composing real-time systems. In *IJCAI*, volume 91, pages 212–217, 1991.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization. *ICML*, 2015.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *ICLR*, 2016.

Dafna Shahaf and Eric Horvitz. Investigations of Continual Computation. In *IJCAI*, pages 285–291, 2009.

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.

Herbert A. Simon. *Administrative Behavior: a Study of Decision-Making Processes in Administrative Organization*. Macmillan, 1947.

Herbert A. Simon. *Models of bounded rationality*, volume 2. MIT Press, 1982.

R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*. Cambridge University Press, 1998.

Larry W Swanson. Cerebral hemisphere regulation of motivated behavior. *Brain research*, 886(1):113–164, 2000.

Larry W Swanson. *Brain Architecture: Understanding the Basic Plan*. Oxford University Press, 2012.

James R. Swisher, Sheldon H. Jacobson, and Enver Yücesan. Discrete-event simulation optimization using ranking, selection, and multiple comparison procedures: A survey. *ACM Transactions on Modeling and Computer Simulation*, 2003.

Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

Jessica Taylor, Eliezer Yudkowsky, Patrick LaVictoire, and Andrew Critch. Alignment for Advanced Machine Learning Systems. Technical Report 20161, MIRI, 2016.

David Tolpin and Solomon Eyal Shimony. Rational Deployment of CSP Heuristics. IJCAI, 2011.

David Tolpin, Tal Beja, Solomon Eyal Shimony, Ariel Felner, and Erez Karpas. Towards Rational Deployment of Multiple Heuristics in A*. In *IJCAI 2013*, 2013.

David Tolpin, Oded Betzalel, Ariel Felner, and Solomon Eyal Shimony. Rational Deployment of Multiple Heuristics in IDA*. In *ECAI*, 2014.

John Von Neumann and Oskar Morgenstern. *Theory of games and economic behavior*. Princeton university press, 1944.

A. Wald. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics*, 16: 117–186, 1945.

Y. Yu. Structural Properties of Bayesian Bandits with Exponential Family Distributions. *Arxiv preprint arXiv:1103.3089*, 2011.

Eliezer Yudkowsky. Artificial Intelligence as a Positive and Negative Factor in Global Risk. In Nick Bostrom and Milan Cirkovic, editor, *Global Catastrophic Risks*, page 303. Oxford University Press Oxford, UK, 2008.

Shlomo Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms*. PhD thesis, Computer Science Division, University of California Berkeley, 1993. URL http://rbr.cs.umass.edu/shlomo/papers/Zshort93.html.

Shlomo Zilberstein and Stuart J. Russell. Optimal Composition of Real-Time Systems. *Artificial Intelligence*, 82(1-2):181–213, 1996. URL http://rbr.cs.umass.edu/shlomo/papers/ZRaij96.html.