

AN ABSTRACT OF THE DISSERTATION OF

Jervis Pinto for the degree of Doctor of Philosophy in Computer Science presented on
June 12, 2015.

Title: Incorporating and Learning Behavior Constraints
for Sequential Decision Making

Abstract approved: _____

Alan P. Fern

Writing a program that performs well in a complex environment is a challenging task. In such problems, a method of deterministic programming combined with reinforcement learning (RL) can be helpful. However, current systems either force developers to encode knowledge in very specific forms (e.g., state-action features), or assume advanced RL knowledge (e.g., ALISP).

This thesis explores techniques that make it easier for developers, who may not be RL experts, to encode their knowledge in the form of behavior constraints. We begin with the framework of adaptation-based programming (ABP) for writing self-optimizing programs. Next, we show how a certain type of conditional independency called "influence information" arises naturally in ABP programs. We propose two algorithms for learning reactive policies that are capable of leveraging this knowledge. Using influence information to simplify the credit assignment problem produces significant performance improvements.

Next, we turn our attention to problems in which a simulator allows us to replace reactive decision-making with time-bounded search, which often outperforms purely reactive decision-making at significant computational cost. We propose a new type of behavior constraint in the form of partial policies, which restricts behavior to a subset of good actions. Using a partial policy to prune sub-optimal actions reduces the action branching factor, thereby speeding up search. We propose three algorithms for learning partial policies offline, based on reducing the learning problem to i.i.d. supervised learning and we give a reduction-style analysis for each one. We give concrete implementations using the popular framework of Monte-Carlo tree

search. Experiments on challenging problems demonstrates large performance improvements in search-based decision-making generated by the learned partial policies.

Taken together, this thesis outlines a programming framework for injecting different forms of developer knowledge into reactive policy learning algorithms and search-based online planning algorithms. It represents a few small steps towards a programming paradigm that makes it easy to write programs that learn to perform well.

©Copyright by Jervis Pinto
June 12, 2015
All Rights Reserved

Incorporating and Learning Behavior Constraints
for Sequential Decision Making

by

Jervis Pinto

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 12, 2015
Commencement June 2015

Doctor of Philosophy dissertation of Jervis Pinto presented on June 12, 2015.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Jervis Pinto, Author

ACKNOWLEDGEMENTS

This thesis would not have been possible without my advisor, Prof. Alan Fern. It has been a great privilege for me to have been given the opportunity to closely observe Alan's principled approach to solving problems. Our one-on-one research meetings are easily my favorite learning experience of my graduate work. I stand in awe of Alan's talents: His insightful analysis, clarity of thought and communication, focus, persistence, professionalism and work ethic. I'm also extremely grateful for his incredible patience with me as I tried to imitate his methods. His incredible ability to design, plan and perform research has easily been the most enjoyable and fruitful learning experience of my career. Alan was also very supportive of my efforts in exploring problems in automated testing, for which I'm very grateful.

My wife, Pallavi, cheered, supported and coaxed me through every step of this journey. There is little I could say that can encompass my gratitude for her tireless love and support. This is also true of my parents and Jen and Lester who have encouraged and supported me unconditionally. I'm keenly aware of my luck to be able to call such stellar individuals, family.

I've been fortunate to have a number of amazing collaborators from other CS areas. Special thanks to Prof. Alex Groce, who took the ABP prototype places its creators never imagined. I consider myself lucky to have been given a chance by my advisor to dip my toes in this very interesting and pragmatic field. I have greatly enjoyed learning about software engineering and automated testing from Alex and have gained invaluable experience at the intersection of ML and automated testing during my collaboration with him.

I'd also like to thank the ABP group: Prof. Martin Erwig, Prof. Thanh Nguyen, Tim Bauer, Patrick Zhu and Thuan Dzung. The group meetings provided me additional viewpoints of ML which continues to shape and inform my work. I'd also like to thank Tim for allowing me to look over his shoulder as he expertly crafted code.

The wonderful atmosphere of learning and research in the ML lab created by Profs. Tom Dietterich, Alan Fern, Prasad Tadepalli, Weng-Keen Wong, Xiaoli Fern and Raviv Raich made my time at OSU extremely enjoyable. Interacting with these inspiring individuals during seminars, reading groups and classes has been my favorite learning experience that I will greatly cherish. Their enthusiasm for learning, kindness and humility makes the OSU ML group a wonderful place to be.

Special thanks to the stalwarts of engineering support, Todd Schechter and Mike Sanders. They are invaluable to our efforts! I'd personally like to thank Mike for his epic patience restarting compute nodes brought down by my experiments on the cluster. I'd also like to thank the graduate co-ordinators (Ferne Simendinger and Nicole Thompson) and the office staff for always being available to assist the graduate students.

The discussions with friends, peers and lab-mates have been one of my favorite experiences at OSU. I'm very grateful to the occupants of KEC 2048 over the years: Neville Mehta, Ethan Dereszynski, Aaron Wilson, Rob Hess, Jun Yu, Dharin Maniar, RK Balla, Javad Azimi, Jesse Hofstedler, Aswin Nadamuna, Saikat Roy, Kshitij Judah, Sriram Natarajan and Ronnie Bjaranason. Also included are collaborators from neighboring domains: Vikram Iyer, Madan Thangavelu, Amin Alipour, Nadia Payet, William Brendel and Forrest Briggs. Each provided wise counsel and often served as excellent pruning functions in my search through the space of ideas. Many continue to provide valuable advice from their current positions and I'm lucky to have a network of such talented colleagues.

Outside the technical realm, I'd like to thank Shireen Hyrapiet, Kunal Kate, Rohan Madtha, Megan Knight, the chemistry gang (Adam, Cory, Somnath and Jessica) and the folks in Seattle for their enjoyable company and valuable perspectives from other disciplines.

Special gratitude is reserved for Janardhan Doppa, who provided priceless guidance at key junctures! I also thank Prof. Prasad Tadepalli, Padma and their family for welcoming Pallavi and me into their home on numerous occasions. I'm grateful for his guidance and teaching in each of his many roles as researcher, committee member, academic advisor and course instructor.

Finally, my eternal gratitude to Prof. Vivek Borkar, who, through a chance encounter, introduced machine learning to me and patiently showed me the possibilities it held. Without his initial guidance and advice, I may have taken a very different path.

A journey this long implies that this list can include only a tiny fraction of the people involved. I apologize to anyone not mentioned. Each of you made the journey memorable in your own special way.

To my family, friends, mentors and teachers: I dedicate this thesis to you.

CONTRIBUTION OF AUTHORS

Martin Erwig and Tim Bauer contributed to the research in chapter 2 and chapter 3.

TABLE OF CONTENTS

	<u>Page</u>
1 Introduction	1
1.1 Hand-coded vs Learned Behavior	1
1.2 Sequential Decision Problems	2
1.2.1 Markov Decision Problems (MDPs)	2
1.2.2 Learning from the Environment	3
1.2.3 Time-bounded Lookahead (Tree) Search	5
1.3 Contributions of this thesis	7
2 Robust Learning for Adaptive Programs by Leveraging Program Structure	8
2.1 Introduction	8
2.1.1 Motivating Example	8
2.2 Adaptation-Based Programming	9
2.2.1 ABP in Java	9
2.2.2 ABP Learning Problem	10
2.3 Basic Program Analysis for ABP	10
2.4 Informing the Decision Process	11
2.4.1 Standard Decision Process	12
2.4.2 Deficiencies with the Standard Decision Process	13
2.4.3 The Informed Decision Process	13
2.4.4 Properties of the Informed Decision Process	14
2.5 Informed SARSA(λ)	14
2.5.1 SARSA(λ)	15
2.5.2 Deficiencies of SARSA(λ)	15
2.5.3 iSARSA(λ)	16
2.5.4 Properties	17
2.6 Experiments	17
2.6.1 Results	18
2.7 Related Work	21
2.8 Summary	21
3 Improving Policy Gradient Estimates with Influence Information	23
3.1 Introduction	23
3.2 MDPs and Policy Gradient Methods	24
3.2.1 Policy Gradient RL	25

TABLE OF CONTENTS (Continued)

	<u>Page</u>
3.3 Context-Specific Independence	26
3.4 Policy Gradient with Influence Functions	29
3.4.1 Incorporating Discounting	30
3.5 Algorithm Properties	31
3.5.1 Variance Reduction	34
3.6 Adaptation-Based Programming (ABP)	35
3.7 Experiments	38
3.8 Related Work	43
3.9 Summary	43
 4 Learning Partial Policies to Speedup MDP Tree Search	 45
4.1 Introduction	45
4.2 Problem Setup	47
4.2.1 Online Tree Search	47
4.2.2 Search with a Partial Policy	48
4.2.3 Learning Problem	50
4.3 Learning Partial Policies	51
4.3.1 OPI : Optimal Partial Imitation	52
4.3.2 FT-OPI : Forward Training OPI	54
4.3.3 FT-QCM: Forward Training Q-Cost Minimization	55
4.4 Implementation Details	57
4.4.1 UCT	57
4.4.2 Partial Policy Representation and Learning	59
4.4.3 Generating Training States	60
4.5 Related Work	61
4.6 Experiments	63
4.6.1 Experimental Domains	64
4.6.2 Setup	67
4.6.3 Comparing FT-QCM, FT-OPI and OPI	68
4.6.4 Relating search and regret	68
4.6.5 Selecting σ_d	70
4.6.6 Comparing FT-QCM to baselines	73
4.6.7 The cost of knowledge	77
4.7 Summary and Future Work	79

TABLE OF CONTENTS (Continued)

	<u>Page</u>
5 Lessons Learned and Future Work	80
5.1 Summary	83
Bibliography	83

LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
2.1	Illustrative adaptive programs. <code>test</code> , <code>randomContext</code> , and <code>payoff</code> are non-adaptive methods. <code>reward</code> and <code>suggest</code> are part of our ABP library (section 2.2).	9
2.2	(a-e) are performance graphs of programs Yahtzee, SeqL (training set), S1,S2,S3 for $\lambda = 0.75$. (f) SeqL (test set) for varying λ and algorithms.	20
3.1	The conditional influence graph for “press” decision variable D_1 is shown after the state at $t = 1$ has been observed. Note that D_1 has only a single path of influence $\{D_1, P_2, R_2\}$, as shown by the thick arcs. Thus “press” decisions can only influence the immediate reward whereas move decisions like D_2 have at least one path to every subsequent reward (not shown) via the state variables X_3, Y_3 and B_3	28
3.2	Illustrative adaptive programs reproduced from Figure 2.1. <code>test</code> , <code>randomContext</code> , and <code>payoff</code> are non-adaptive methods. <code>reward</code> and <code>suggest</code> are part of the ABP library [10].	36
3.3	Effect of discarding irrelevant rewards. The figure shows the performance graphs of adaptive programs without any discounting. The vertical axis is the total reward over a single execution. The horizontal axis shows the number of program executions.	41
3.4	Effect of discounting. The figure shows the performance graphs of representative adaptive programs for two moderately high values of β and a subset of adaptive programs. The remaining programs are either invariant to moderately high values of β (S1, S2) or qualitatively similar to the ones shown (NetTalk \approx SeqLabel). For SeqLabel, both methods perform identically and the graphs overlap.	42
3.5	Label prediction accuracy (as percentage) for phoneme and stress labels in the NetTalk datasets. Shown are accuracies before and after learning on the training set and the final evaluation on the test set.	42
4.1	Unpruned and pruned expectimax trees with depth $D = 2$ for an MDP with $ A = 3$ and two possible next states.	48

LIST OF FIGURES (Continued)

<u>Figure</u>		<u>Page</u>
4.2	Training data generation and ranking reduction for each algorithm. Start by sampling s_0 from the same initial state distribution μ_0 . OPI uses the optimal policy at each depth whereas FT-OPI and FT-QCM use the most recently learned complete policy (at depth $d - 1$). OPI and FT-OPI use 0-1 costs. Only FT-QCM uses the Q cost function which turns out to be critical for good performance. . .	61
4.3	A visualization of a Galcon game in progress. The agent must direct variable-sized fleets to either reinforce its controlled planets or to take over enemy planets or unoccupied ones. The decision space is very large and consists of hundreds of actions.	65
4.4	A typical scoring sheet for Yahtzee. In each game, all 13 categories must be scored, one per stage. Each stage consists of at most three dice rolls (original roll + two more controlled by the player). The objective is to maximize the total score, under significant stochasticity introduced by the dice rolls. Our simulator implements the variant with a single Yahtzee.	66
4.5	The search performance of UCT using partial policies learned by FT-QCM, FT-OPI and OPI as a function of the search budget, measured in seconds. The results for Galcon are in the first column and those for Yahtzee in the second. Each row corresponds to a different set of pruning thresholds σ_d with maximum pruning in the top row and least pruning in the third row. FT-QCM easily outperforms FT-OPI and OPI in all but one case (where they are tied). As mentioned in section 4.6.1, the reward is a linear function of the raw score, measured at the end of the game.	69
4.6	Average regret and pruning error for the learned partial policies ψ_d , as a function of the pruning ratio on the Galcon domain. The metrics are evaluated with respect to a held-out test set of states sampled from μ_0 . FT-QCM has the least error and regret at the root. At depths $D > 0$, performance degrades towards random.	71
4.7	Average regret and pruning error for the learned partial policies ψ_d , as a function of the pruning ratio on the Yahtzee domain. The metrics are evaluated with respect to a held-out test set of states sampled from μ_0 . FT-QCM has the least error and regret at the root. At depths $D > 0$, FT-QCM continues to perform better than FT-OPI and OPI due to the relatively higher quality of training data at deeper states compared to Galcon.	72

LIST OF FIGURES (Continued)

<u>Figure</u>		<u>Page</u>
4.8	The key empirical result of this paper shows that the search performance of UCT injected with FT-QCM significantly outperforms all other baselines at small budgets. At larger budgets, it either continues to win or achieves parity.	76
4.9	Setting the cost of knowledge to zero by using simulations instead of time. UCT with biased rollouts (IR) now appears to perform much better. However, UCT with FT-QCM continues to perform the best across significant portions of the anytime curve.	78

LIST OF TABLES

<u>Table</u>	<u>Page</u>
4.1	Instantiations for OPI, FT-OPI and FT-QCM in terms of the template in Algorithm 3. Note that ψ_d is a partial policy while π_d^* , ψ_d^+ and ψ_d^* are complete policies.
	52

LIST OF ALGORITHMS

<u>Algorithm</u>		<u>Page</u>
1	iSARSA(λ). All Q, η values are initially zero. The current program location ℓ' is at choice (A', a') . The last choice is (A, a) with reward r seen in between. .	17
2	PGRI.	31
3	A template for learning a partial policy $\psi = (\psi_0, \dots, \psi_{D-1})$. The template is instantiated by specifying the pairs of distributions and cost functions (μ_d, C_d) for $d \in \{0, \dots, D - 1\}$. LEARN is an i.i.d. supervised learning algorithm that aims to minimize the expected cost of each ψ_d relative to C_d and μ_d	51

Chapter 1: Introduction

Consider the problem of writing a computer controller that can play a real-time strategy (RTS) game which is a popular video game genre involving building and controlling large armies. Writing such controllers is extremely complicated and involves managing many complex sub-tasks simultaneously. For example, in RTS games (e.g., Wargus, StarCraft), the player must manage collecting resources, building units, advancing technology trees, fighting battles, etc. All of this must be done under significant uncertainty about the map and the actions of other players in the game. In such situations, the standard approach involves large teams of developers encoding scripted behaviors. This is time-consuming and costly since every aspect of the controller must be fully-specified. From the designer’s perspective, this approach produces easily maintainable and perhaps more importantly, predictable behavior. However, in terms of performance (i.e., the user’s perspective), a hand-coded solution can often perform very poorly.

The key issue is that human players are particularly good at detecting and exploiting flaws in the controller. For example, a common exploit of scripted behaviors is via “kiting” which involves finding ways to induce enemy units into chasing a friendly unit without suffering any damage [67]. One particularly egregious kiting attack against game “AI” involves moving a fast cavalry unit across the front line of an enemy formation, inducing the whole formation to rotate by 90 degrees, thereby exposing its flanks.

Note that in the above example, even a novice human player would instantly spot the trap and would be unlikely to move an entire army out of formation just to pursue a single unit. More importantly, a human is unlikely to repeat such a mistake. Scripted behaviors regularly make such mistakes and worse, are doomed to repeat them since they are incapable of learning.

1.1 Hand-coded vs Learned Behavior

The above example highlights the following fact: It is very difficult, even for experts, to correctly specify complex behaviors. In the RTS example, the hand-coded behavior was sub-optimal despite massive, costly efforts undertaken by expert game designers with advanced knowledge about the domain (i.e., the game’s internal scoring rules, movement dynamics, etc.) unknown

to the typical user. Despite these advantages, under similar game conditions, hand-coded game strategies routinely perform very poorly against humans.

In situations where the expert or designer has limited knowledge or significant uncertainty about future operating conditions, the task of hand-coding an optimal controller becomes even more challenging. For example, consider the task of designing a complex network protocol like 802.11 without having precise knowledge about the expected network traffic. The current approach involves simulating different types of network traffic to find a controller that works well, on average. However, networking protocols that learn have been shown to perform significantly better in a number of situations [16, 75, 2].

Thus, fully specified strategies are predictable and relatively easy to analyze but they can perform very poorly. An alternative strategy is to replace hand-coded behaviors with a learning algorithm that improves its behavior with experience. This framework, called sequential decision-making under uncertainty, requires the designer to leave some choices “open” in exchange for (hopefully) improved decision-making. In this thesis, we study sequential decision problems in the framework of discrete Markov Decision Processes (MDPs), reviewed next.

1.2 Sequential Decision Problems

We begin with a high-level description of a very popular framework for formulating sequential decision problems.

1.2.1 Markov Decision Problems (MDPs)

In the setting considered here, an MDP [52] is a tuple (S, A, P, R) , where S is a finite set of states, A is a finite set of actions, $P(s'|s, a)$ is the probability of transitioning into state s' after executing action a in state s , and $R(s, a) \in \mathbb{R}$ is the reward function specifying the immediate utility of taking action a in state s . The typical goal in MDP planning and learning is to compute a policy for selecting an action in any state, such that following the policy (approximately) maximizes some measure of long-term expected reward. For example, two common choices involve maximizing the discounted or undiscounted sum of rewards.

We now briefly describe the main algorithmic classes that can be used to solve MDPs. Throughout the discussion, we will highlight the ways in which a designer’s knowledge can be encoded as different types of behavior constraints. We will review existing techniques and

propose new types of behavior constraints and learning algorithms that can leverage them.

1.2.2 Learning from the Environment

Methods for learning directly from the environment typically require very little of the designer. In fact, they can be viewed as the opposite of fully-specified, hand-coded controllers. We start with reinforcement learning, discuss its difficulties and then describe how knowledge in the form of behavior constraints can be used to accelerate learning.

1.2.2.1 Reinforcement Learning (RL)

In RL [63], the goal is to learn a policy through direct interaction with the environment. The RL loop operates as follows: The learner is given an “observation” (environment state) and uses its internal state to choose an action to execute next. The environment responds with a numeric reward as feedback and the next observation, which the learner uses to update its internal state. In the episodic problems considered here (e.g., games), these steps repeat until a terminal state is reached (e.g., the game ends) at which point, the environment is “reset” to some starting state¹. Typically, the learner’s performance improves as it gains more experience.

RL algorithms and hand-coded controllers are the two extremes on the spectrum of solutions for encoding behavior. In RL, the designer specifies little else about the problem besides a representation in terms of the states, actions, rewards and policy class. The agent must learn to select actions through trial-and-error, receiving only rewards as feedback. Since actions may have long-term effects, the induced optimization problem must operate over many time steps which makes it very challenging to solve. This credit assignment problem is central to RL and techniques that make it easier for the agent to assign credit typically perform well. Unfortunately, this also implies that in the absence of additional knowledge or structural assumptions, pure RL algorithms are often very costly to use, either in terms of how much experience is needed (“sample complexity”) or in terms of their computational complexity (e.g., model-based planning or search)

¹Note that in non-episodic problems, the sum of rewards may not converge. A common strategy here is to discount future rewards, allowing the sum to converge.

1.2.2.2 Hierarchical RL (HRL)

One technique to accelerate the learning of complex behavior is via a task hierarchy which decomposes a large task (e.g., win RTS game) into a hierarchy of smaller tasks (e.g., collect gold, collect stone, build barracks, etc.). The agent is now restricted to learning a policy that is consistent with the task hierarchy. HRL frameworks like MAXQ [21] and ALISP [4, 5] permit the designer to declare (or more recently, learn [42]) the task hierarchy. Given the task hierarchy, an appropriate algorithm can then be used for learning a good “flat” policy consistent with the hierarchy.

The main observation here is that the task hierarchy serves to constrain the set of behaviors that the learner may consider during its search. That is, the task hierarchy “prunes” the space of behaviors, ideally only eliminating obviously incorrect strategies while preserving good or optimal behaviors. It may occur that the optimal policy is no longer admissible under the imposed constraints. However, given the difficulty of learning good behavior in large MDPs, let alone optimal, the tradeoff made in HRL seems reasonable. That is, in return for risking the inadmissibility of optimal behavior, the learner gets to consider a reduced set of behaviors, quickly identifying near-optimal behaviors instead.

1.2.2.3 Adaptation-based Programming (ABP)

HRL systems are primarily intended for designers who are RL experts. However, we have discussed how decision problems occur naturally whenever a complex behavior needs to be encoded. We focus on the use case where the designer has significant domain knowledge but perhaps is less familiar with algorithms for sequential decision problems. This is the primary use case for adaptation-based programming (ABP) which is a programming paradigm that allows end-users to leverage the benefits of behavior learning algorithms without requiring sophisticated knowledge of RL [10, 9]. ABP allows the designer to specify as much of the controller as desired, leaving as “open”, decisions in the program where there is significant uncertainty. Instead, the designer only specifies the relevant context and the set of candidate actions at these “choice points” and numeric feedback indicating the program’s performance. The ABP program induces a MDP which is then optimized using standard RL.

1.2.2.4 Behavior Constraints via Influence Information

The next two chapters will discuss ABP in more detail. Here, we only observe that the ABP paradigm makes it particularly easy to encode knowledge as behavior constraints. Given the large performance improvements produced by existing types of behavior constraints, it is natural to ask if there are other forms that can be leveraged to accelerate learning and improve performance.

The first contribution of this thesis is to show how the ABP framework naturally induces a new type of behavior constraint in the form of conditional independencies between states, actions and rewards. We call this knowledge “influence information”. Our second contribution, contained in [chapter 2](#), shows how influence information can be leveraged inside a value-function learning algorithm in order to significantly simplify the credit assignment problem induced during the optimization of ABP programs. Our third contribution is to obtain stronger theoretical guarantees by replacing value-function learning with a policy gradient method in [chapter 3](#).

These policy learning algorithms constitute the first part of the thesis. For the second part, we replace policy learning with lookahead tree search, a fundamentally different type of decision-making. However, the central idea remains the same: We show how the quality of decisions can be significantly improved by leveraging behavior constraints, in this case, by learning partial policies. We begin with a high-level description of planning algorithms for lookahead tree search.

1.2.3 Time-bounded Lookahead (Tree) Search

This type of sequential decision making does not involve any policy learning at all. Rather, given a state at which a decision must be made, in problems where we have access to a model or simulator of the environment, we can replace reactive decision making with time-bounded search. Using the simulator, search-based or planning algorithms explicitly consider the space of future executions from the root state. After a certain amount of time, the best available decision is returned which the agent executes in the real environment. A particularly popular choice of search-based algorithm is lookahead tree search (LTS). Examples of LTS methods include classical heuristic search techniques like A* [55], AO* [45], etc. Recently, simulation-based methods like UCT [25] have become very popular [17] after their strong performance in many challenging domains like Computer Go [26].

In a typical LTS application, performance usually improves with increasing search budgets. The reason for this is intuitively clear: The estimates at interior nodes, and therefore the decisions, improve with more experience which translates into higher quality simulations. Thus, methods that improve the quality of tree construction often produce superior performance and have attracted significant attention in the planning community. The typical approach to improve search take place as follows: The designer has significant amounts of problem-specific or domain-specific knowledge (i.e., search-control knowledge). This knowledge is encoded into a convenient representation (e.g., leaf evaluation heuristic, informed rollout policy, etc.). Finally, the search procedure must be modified to use this representation, resulting in biased tree construction and hopefully, improved performance.

The large performance improvements that can often result from the use of search-control knowledge has led to a large body of work for injecting “side information” into practically every aspect of tree search. For example, in MCTS alone, techniques are available for node initialization [26], move ordering [19], leaf evaluation [69], generalizing between nodes (RAVE [26]), classical techniques like opening books, transposition tables, etc. [17].

1.2.3.1 Behavior Constraints via (Learned) Partial Policies

As in the previous sections, we focus on knowledge that can be represented as constraints on the behavior. An intuitively appealing technique to utilize behavior constraints inside a search procedure is to identify and remove obviously sub-optimal actions from consideration. That is, we are interested in using behavior constraints to prune actions during search.

For lookahead tree search, a natural way to encode behavior constraints is via a partial policy. As opposed to complete policies which return exactly one action in a given state, a partial policy is permitted to return a subset of actions. Intuitively, specifying a partial policy is similar in principle to writing a program in the ABP framework since the designer is free to restrict the set of actions that can be considered at any point in the program. However, here, we are interested in learning partial policies. This places a much smaller knowledge requirement on the designer. In [chapter 4](#), we consider the learning of partial policies for speeding up MDP tree search. We will give a family of algorithms, based on imitation learning, for learning partial policies. Each learning algorithm comes with good theoretical properties in terms of bounds on the expected regret at the root which are a key contribution of this work. We show how the learned partial policies can be easily incorporated into existing simulation-based tree search algorithms

like UCT. A large-scale experimental evaluations reveals significant real-time improvements in search quality on two challenging MDPs.

1.3 Contributions of this thesis

In [chapter 2](#), we show how adaptive programs that may seem correct to the user actually induce very challenging learning problems. We introduce influence information and show how it can arise naturally in ABP programs. The main contribution of this chapter is a learning algorithm that can leverage the influence information. Specifically, we modify the SARSA(λ) learning algorithm to obtain an informed variant iSARSA(λ). The results on synthetic and real-world adaptive programs show significantly improved performance.

In [chapter 3](#), we continue to focus on leveraging influence information. We give an algorithm for incorporating influence information into a policy gradient learning algorithm. The main benefit from switching from the value function method to a policy gradient method is the strong theoretical guarantees under function approximation. We show how our proposed algorithm, PGRI, maintains its good theoretical properties, under both function approximation and the use of influence information. For the experimental evaluation, we show strong performance improvements obtained by simplifying the credit assignment problem in adaptive programs.

In [chapter 4](#), we introduce partial policies. This constitutes the second novel form of behavior constraints. Next, we investigate methods for learning partial policies. We give three algorithms, based on imitation learning, and give formal bounds on their expected regret. We use the learned partial policies to prune actions during tree search. Large-scale experiments on two challenging real-world problems show that using the learned partial policies to prune actions produces large performance gains, significantly speeding up MDP tree search.

Taken together, this thesis proposes a number of techniques and learning algorithms that make it easier to solve large sequential decision problems by leveraging the designer’s knowledge in the form of constraints on the agent’s behavior. Our methods are primarily intended to assist designers who may have significant problem-specific knowledge but may not have much experience with RL or planning algorithms. Experimental results show significant performance gains produced by our techniques on real-world problems. Our open-source implementations make these methods for sequential decision making more accessible to developers, hopefully making it easier to write intuitively clear, self-optimizing programs that learn to perform well.

Chapter 2: Robust Learning for Adaptive Programs by Leveraging Program Structure

2.1 Introduction

Deterministic programming languages are not well suited for problems where a programmer has significant uncertainty about what the program should do at certain points. To address this, we study *adaptation-based programming* (ABP) where *adaptive programs* allow for specific decisions to be left open. Instead of specifying those decisions the programmer provides a “reward” signal. The idea then is for the program to automatically learn to make choices at the open decision points in order to maximize reward.

As detailed in [section 2.7](#), prior work has studied reinforcement learning (RL) for optimizing adaptive programs. Here we identify a shortcoming of this prior work, which is a significant obstacle to allowing non-RL-experts to benefit from ABP. Mainly, the success of prior RL approaches depends on subtle details of an adaptive program’s structure, which is difficult to predict for non-RL-experts¹.

2.1.1 Motivating Example

Consider the two very simple adaptive programs P1 and P2 in [Figure 2.1](#) written using our ABP Java library (see [section 2.2](#)). The objects A and B are *adaptives*, and are used to encode the programmer’s uncertainty. Program P1 is a conditional structure with rewards at the leaves. Program P2 is a trivial transformation of P1 and to a typical programmer P1 and P2 appear functionally equivalent. However, as will be detailed in [section 2.4](#), these programs induce very different learning problems, apparent when prior RL approaches easily solve P1, but can fail for P2. Such sensitivity to trivial program changes is likely to be counter-intuitive and frustrating for a typical programmer.

As detailed in [section 2.4](#) and [section 2.5](#), the fundamental problem with prior work is that they ignore much information about the structure of the program being optimized. Our primary

¹A version of this work appeared at ICMLA 2010 [[49](#)].

```

a = A.suggest();
b = B.suggest();

if (test()) {
  if (A.suggest()) {
    reward(1);
  } else {
    reward(2);
  }
} else {
  if (B.suggest()) {
    reward(2);
  } else {
    reward(1);
  }
}
Program P1

if (test()) {
  if (a) {
    reward(1);
  } else {
    reward(2);
  }
} else {
  if (b) {
    reward(2);
  } else {
    reward(1);
  }
}
Program P2

for (i=1; i<N; i++) {
  c = randomContext();
  m = move.suggest(c);
  reward(payoff(c,m));
}
Program P3

```

Figure 2.1: Illustrative adaptive programs. `test`, `randomContext`, and `payoff` are non-adaptive methods. `reward` and `suggest` are part of our ABP library (section 2.2).

contribution is to develop a general learning approach that takes program structure into account. To our knowledge this is the first work that attempts to exploit such program structure for faster and more robust learning. We show empirically that these ideas lead to significant improvement in learning on a set of adaptive Java programs.

2.2 Adaptation-Based Programming

We review our Java ABP library and the learning problem.

2.2.1 ABP in Java

The key object in our ABP library is an *adaptive*, which is the construct for specifying uncertainty. An adaptive has a *context type* and *action type* and can be created anywhere in a Java program where it can be viewed and used as a function from the context type to the action type, which changes over time via learning. This function is called via an adaptive’s `suggest` method and we refer to any call to `suggest` as a *choice point*. In P1 and P2 of Figure 2.1, the action type of adaptives A and B is boolean, and the context is null. As another example, an adaptive program to control a grid world agent might include an adaptive `move` with action type `{N, E, S, W}` for the desired movement direction and context type `GridCell`, enumerating the grid locations. The call `move.suggest(gridCell)` would return a movement direction for

gridCell.

The second key ABP construct is the `reward` method, which has a numeric argument and can be called anywhere in a Java program. In P1 and P2 the `reward` calls give the desirability of each leaf of the conditional. The goal of learning is to select actions at choice points to maximize the reward during program runs. In a grid world, the reward function might assert a small negative reward for each step and a positive reward for reaching a goal.

2.2.2 ABP Learning Problem

A *choice function* for an adaptive A , is a function from A 's context type to action type. A *policy* π for adaptive program P gives a choice function for each adaptive in P . The execution of P on input x with respect to π is an execution of P where calls to an adaptive A 's `suggest` method are serviced by A 's choice function in π . Each such execution yields a deterministic sequence of `reward` calls and we denote the sum of rewards by $R(P, \pi, x)$. In a grid world, an input x might be the initial grid position and $R(P, \pi, x)$ might give the number of steps to the goal when following π .

Let D be a distribution over possible inputs x . The learning goal is to find a policy π that maximizes the expected value of $R(P, \pi, x)$ where x is distributed according to D .² Typically, finding the optimal π analytically is intractable. Thus, the ABP framework attempts to “learn” a good, or optimal, π based on experience gathered through repeated executions of the program on inputs drawn from D .

2.3 Basic Program Analysis for ABP

Our proposed learning approach (section 2.4 and section 2.5) is based on leveraging the structure of adaptive programs and thus assume the ability to compute the following two properties of an adaptive program during its execution. v_ℓ denotes the value suggested by a `suggest` call at program location ℓ .

1. $\bar{P}(\ell, \ell')$ says that v_ℓ is definitely irrelevant for reaching the location ℓ' , that is, ℓ' would have been reached no matter what the value of v_ℓ is. In this case we say that ℓ' is *path-independent* of the choice point ℓ .

²In order to make this a well defined objective we assume that all program executions for any π and x terminate in a finite number of steps so that $R(P, \pi, x)$ is always finite. If this is not the case then we can easily formulate the optimization problem in terms of discounted infinite reward.

2. $E(\ell, \ell')$ says that v_ℓ is potentially relevant for the computation of any value in a statement at position ℓ' . We say that ℓ' is *value-dependent* on ℓ .

To understanding our learning algorithm, it is sufficient to assume an oracle for these properties. We do not require that the oracle be complete for these properties, which are formally undecidable. Rather, our learning approach only requires that the oracle be correct whenever it asserts that one of the properties is `true`—the oracle can be incorrect when asserting `false`. The design goal of our approach is for a more complete oracle to lead to faster learning.

We briefly outline a relatively simple approach for implementing the oracle. The details are not important for understanding our proposed learning approach and can be safely skipped. The oracle can be implemented as a pre-processor that instruments an adaptive program with code that generates information for evaluating the above predicates. We will use the notation $V(\ell)$ for the set of variables used in the statement at position ℓ . The key to implementing the predicates is to track the data flow of suggested values as well as their use in control-flow statements, which can be done via standard data-flow analysis techniques [1]. This gives for each program block B and each choice point ℓ the set $V_B(\ell)$ of variables visible in B whose values are potentially derived from v_ℓ .

Second, we instrument the program by inserting commands after each call to `suggest` and before each branching statement to report suggested value generation and usage. Moreover, before each branching statement at a position ℓ' with $V(\ell') \cap V_B(\ell) \neq \emptyset$ (where B is the current block), we insert a command that, during the program run, will produce a *use* event for v_ℓ . Through this instrumentation it is easy to compute the predicates $\bar{P}(\ell, \ell')$ and $E(\ell, \ell')$. Our experimental results are based on instrumenting the adaptive programs by hand in the same way the above pre-processor would. Note that developing a fully-automated processor is a large but straightforward engineering task that is not relevant to the main contribution of this paper (using information about program structure).

2.4 Informing the Decision Process

Prior work on learning for ABP follows two steps: 1) Define a decision process, which we will call the *standard decision process* for the adaptive program, 2) Apply standard RL to this decision process. Below we describe the first step, a general deficiency with it and our proposed improvement. For ease of description and without loss of generality, we focus our discussion on adaptive programs where the context type of all adaptives is null, i.e. the adaptives do not depend

on any context.³

We first define the notion of a general *decision process*, which is defined over a set of *observations* and *actions*. We consider episodic decision processes, where each episode begins with an initial observation o_1 , drawn from an initial observation distribution I . There is a decision point at each observation, where a controller must select one of the available actions. After the i 'th action a_i , the decision process generates a numeric reward r_i and a new observation o_{i+1} according to a transition distribution $T(r_i, o_{i+1} | H_i, a_i)$, where $H_i = (o_1, a_1, r_1, o_2, a_2, r_2, \dots, o_i)$ is the observation history. Note that in general the transition distribution can depend arbitrarily on the history. In the special case when this distribution only depends on the o_i and a_i , then the process is a Markov decision process (with states corresponding to observations). The goal of RL is to interact with a decision process in order to find an action-selection policy that maximizes the expected total episodic reward.

2.4.1 Standard Decision Process

Similar to prior ABP work [46, 4] we can define a decision process corresponding to an adaptive program P and a distribution D over its inputs. We will refer to this as the *standard decision process* to reflect the fact that prior work on ABP define a similar process to which RL is applied. The observations of the standard process are the names of the adaptives in P and the process actions are the actions for those adaptives. Each episode of the process corresponds to an execution of P on an input drawn from D . Decision points correspond exactly to choice points during the execution, and at each decision point the observation generated is the name of the adaptive. It is easy to show that there is a well-defined but implicit initial observation distribution and transition function for the standard process, both of which will be unknown to the learning algorithm. Further, there is a one-to-one correspondence between policies for P and for the standard process and the expected total reward for a policy is the same for the adaptive program and associated process. Thus, we can directly apply RL algorithms to the process and arrive at a policy for the program.

³Since we have assumed earlier that all context types are finite sets, it is easy to treat an adaptive A that has a non-null context type over values $\{c_1, \dots, c_n\}$ as a collection of adaptives $\{A(c_1), \dots, A(c_n)\}$, each one having a null context type and the same action type as A . Any suggest call $A.suggest(c)$ can be replaced by a conditional statement that checks the value of the context argument c and makes a call to $A(c).suggest()$ instead.

2.4.2 Deficiencies with the Standard Decision Process

Consider now the standard process corresponding to P1 in [Figure 2.1](#). Each run of the program will generate exactly one observation, either A or B. The reward after the observation is equal to the appropriate leaf reward, which depends on the selected action. It is easy to verify that this process is Markovian and accordingly it can be easily solved by nearly all RL algorithms. In contrast, P2 induces a standard process that is non-Markovian. To see this, notice that the observation sequence for each run of the program is always A, B with zero reward between those observations and a non-zero reward at the end depending on the actions. This process is non-Markovian since the final reward depends not just on B and the action selected, but also on the actions of A (in the case that the left branch of the top level IF statement is selected). Because of this, applying algorithms that strongly rely on the Markov property, such as Q-learning or SARSA(0) can fail quite badly as our experiments will show. Furthermore, the credit-assignment problem is more difficult for P2 than P1 since the rewards arrive further from the decision points responsible for them. This example shows how even the simplest of program transformations can result in an adaptive program of very different difficulty for RL.

2.4.3 The Informed Decision Process

The general problem highlighted by the above example is that for the standard process, the sequence of observed decision points and rewards is highly dependent on details of the program structure, and some sequences can be much more difficult to learn from than others. Here, we partially address the problem by introducing the informed decision process, which is similar to the standard decision process, but with the addition of *pseudo decision points*. The new decision points will occur during a program execution whenever an instruction “depends” on the action of a previous choice point. Intuitively this encodes information into the process about when, during a program execution, the action at a choice point is actually used which is not available in the standard process.

More formally, given an adaptive program P , the corresponding informed decision process has the same observation and action space as the standard process (names of adaptives and their actions respectively). Also like the standard process, the informed process has a decision point whenever, during the execution of P , a choice point involving an adaptive A is encountered, which generates an observation A and allows for the selection of one of A ’s actions. In addition, at each non-choice-point location ℓ' of P , the oracle is asked for each prior choice point location

ℓ , whether ℓ' is value-dependent on ℓ . If the answer is yes, then a pseudo choice point is inserted into the informed decision process, which generates observation A , where A is the adaptive at ℓ . Further, the only action allowed at this pseudo decision point is the action a that was previously selected by A , meaning that the controller has no real choice and must select a . The only effect of adding the pseudo decision points is for the informed process to generate an observation-action pair (A, a) at times in P 's execution where A 's choice of a is potentially influential on the immediate future.

2.4.4 Properties of the Informed Decision Process

It is easy to show that there is a one-to-one correspondence between policies of the informed process and of the adaptive program and that policy values are preserved. Thus, solving the informed decision process is a justified proxy for solving the adaptive program. Consider again P2 in Figure 2.1 on an execution where A and B both select `true`. If the variable `test()` method evaluates to `true`, then the informed decision process will generate the following sequence of observations, actions, and rewards,

`A, true, 0, B, true, 0, A, true, 1`

where the final observation A is a pseudo decision point, which was inserted into the process when it was detected that the condition in the IF statement was dependent on it via the variable `a`. The insertion of the pseudo decision point has intuitively made the sequence easier to learn from since the choice (A, true) , which was ultimately responsible for the final reward, is now seen before this reward.

2.5 Informed SARSA(λ)

While the informed decision process will typically be better suited to standard RL algorithms, it will still often be non-Markovian and not capture all of the potentially useful information about program structure. Prior work has studied the problem of learning memoryless policies for non-Markovian processes [61, 47, 40], showing that when good memoryless policies exist, learning algorithms based on eligibility traces such as SARSA(λ) [63] can often find optimal or very good policies [40]. Thus, we take SARSA(λ) as a starting point and later introduce the informed SARSA(λ) (or simply, iSARSA(λ)) algorithm that leverages information about program structure not captured by the informed process.

2.5.1 SARSA(λ)

The SARSA(λ) algorithm interacts with a decision process in order to learn a Q-function $Q(A, a)$. While SARSA(λ) was originally developed for MDPs, it can be applied to non-Markovian processes, though certain guarantees are lost. The key idea of SARSA(λ) is to maintain an eligibility trace function $\eta_t(A, a)$ and to update the Q-function after each transition for each (A, a) according to their eligibilities. Intuitively, recently observed pairs are more eligible for update based on a new transition.

At the start of each program run we set $\eta(A, a) = 0$ for all pairs. The behavior of SARSA(λ) can now be described by what it does at each decision point and transition: Given a current observation A , SARSA(λ) first selects an action a according to an exploration policy, ϵ -greedy here. This action causes a transition, which produces a reward r and a new observation A' upon which it again uses the exploration policy to select an action a' . The algorithm next updates the eligibility trace and Q-values for all pairs as follows:

$$\eta(A, a) \leftarrow 1 \quad (2.1)$$

$$\eta(B, b) \leftarrow \lambda \eta(B, b), \quad \text{for all } (B, b) \neq (A, a) \quad (2.2)$$

$$\delta \leftarrow r + Q(A', a') - Q(A, a) \quad (2.3)$$

$$Q(B, b) \leftarrow Q(B, b) + \alpha \cdot \delta \cdot \eta(B, b), \quad \text{for all } (B, b) \quad (2.4)$$

where $0 \leq \lambda \leq 1$ is the eligibility parameter, and $0 < \alpha < 1$ is the learning rate. For larger values of λ , decisions are more eligible for update based on temporally distant rewards, with $\lambda = 1$ corresponding to learning from Monte-Carlo Q-value estimates. The selection of λ is largely empirical, but for non-Markovian processes, larger values are preferred.

2.5.2 Deficiencies of SARSA(λ)

Consider the observation-action-reward sequence of the informed process for an execution of program P2 described at the end of [section 2.4](#),

`A, true, 0, B, true, 0, A, true, 1,`

where the final observation was a pseudo decision point inserted by the informed process. When the final reward of 1 is observed, the choice (B, true) will be eligible for update. However,

a simple analysis of the program reveals that in this trace the decision of adaptive B had no influence on whether the reward was received or not. Thus, intuitively the decision involving B does not deserve credit for that reward, but this fact is missed by the generic SARSA(λ) algorithm.

A more severe example is program P3 in [Figure 2.1](#), which contains a loop where each iteration corresponds to the complete play of a simple game. The action, selected by the adaptive move based on the current context influences the reward payoff. An execution of P3 iterates through the loop generating a sequence of choices made by the adaptive and rewards. A simple analysis reveals that the choice made at iteration i has no influence on the reward observed in future iterations and thus should not be given credit for those rewards. However, SARSA(λ) again does not capture this information and will update the choice made at i based on some combination of all future rewards, making the learning problem very difficult since the independence across iterations must be learned.

Exploiting this type of simple program analysis to improve credit assignment is the design goal of iSARSA(λ). The main idea is to reset the eligibility of such irrelevant choices to zero, making them ineligible for updates based on rewards they provably have no influence on.

2.5.3 iSARSA(λ)

We now describe the working of our algorithm listed in [Algorithm 1](#). iSARSA(λ) gets invoked at every observation-action pair (A', a') (henceforth simply ‘choice’) in the informed process. For each such (A', a') , let (A, a) denote the immediately preceding choice and r be the reward between (A, a) and (A', a') . Like SARSA(λ), we first set the eligibility of choice (A, a) to 1 and decay the eligibility of all prior choices by λ (Lines 1-4). Furthermore, iSARSA(λ) will exploit the program analysis and ask for every previous choice (B, b) , whether it is irrelevant to getting to A' (Line 8). In this case $Q(B, b)$ is updated based on only the immediate reward r (rather than also on the estimate of $Q(A', a')$) and its eligibility is then reset to zero (Lines 9-10) so that it will not receive credit for further rewards. Otherwise if irrelevance is not detected for (B, b) the usual SARSA(λ) update is performed (Line 12).

Algorithm 1 iSARSA(λ). All Q, η values are initially zero. The current program location ℓ' is at choice (A', a') . The last choice is (A, a) with reward r seen in between.

```

1: for each choice  $(B, b)$  do
2:    $\eta(B, b) = \eta(B, b) * \lambda$ 
3: end for
4:  $\eta(A, a) = 1$ 
5:  $\delta_e = r - Q(A, a)$  ▷ terminal delta
6:  $\delta = r + Q(A', a') - Q(A, a)$  ▷ regular delta
7: for each previous choice  $(B, b)$  at location  $\ell$  do
8:   if  $\ell'$  is path-independent of  $\ell$  then
9:      $Q(B, b) = Q(B, b) + \alpha \eta(B, b) \delta_e$ 
10:     $\eta(B, b) = 0$ 
11:   else
12:      $Q(B, b) = Q(B, b) + \alpha \eta(B, b) \delta$ 
13:   end if
14: end for

```

2.5.4 Properties

The convergence of SARSA(λ) for arbitrary lambda is an open problem even for Markovian processes, though the algorithm is widely used for its well documented empirical benefits. However, for $\lambda = 1$, iSARSA(λ) when run in the informed decision process has a useful interpretation as a Monte Carlo algorithm. In particular, the Q-value of program choice (A, a) will be updated toward the sum of future program rewards, excluding those rewards for which (A, a) is provably irrelevant. This is an intuitively appealing property and as we will show can lead to very large empirical benefits. As a simple illustration, for program P3 it is easily verified that iSARSA(λ) will update the choice at iteration i based on only the reward observed at iteration i , rather than future iterations, as appropriate.

2.6 Experiments

We experiment with adaptive programs that include a Yahtzee playing program, a sequence labeling program, and 3 synthetic programs containing a variety of program structures.

Yahtzee: Yahtzee is a complex, stochastic dice game played over 13 rounds or cycles where each round typically requires three high-level decisions [27]. We wrote an adaptive program that learns to play Yahtzee over repeated games. It includes two adaptives to make the two

most difficult sub-decisions while other easily-made decisions were hard-coded. The program structure is such that certain choices made by the adaptives may get ignored depending on the game context making for a difficult credit assignment problem.

Sequence Labeling (SeqL): This is an important problem in machine learning, where the goal is to accurately label a sequence of input symbols by a sequence of output labels. The programmer will often have significant domain knowledge about proper labelings, and ABP provides a tool for encoding both, the knowledge and the uncertainty. We wrote an adaptive program that repeatedly cycles through a set of labeled training sequences, and for each sequence makes a left to right pass assigning labels. The reward statements reflect whether the selected labels are equal to the true labels. At each sequence position the choice of label is made using a conditional structure that contains a combination of the programmer’s knowledge and a single adaptive, which ends up making the credit assignment problem quite difficult for standard RL. We generated a synthetic data set based on a Hidden Markov Model (HMM) involving 10 input and 10 output symbols. The training set contains 500 sequences, while the test set has 200 sequences. The test set is hidden during training and used to evaluate the learned program.

Synthetic Programs: These are composed of adaptives and `reward` statements scattered across branches and loops. All adaptives have binary actions and possibly non-null context types. *Program S1* is based on P3 from [Figure 2.1](#), where the adaptive has 10 possible contexts. This program captures the common adaptive-programming pattern of repeated runs which can cause failure when using standard RL. *Program S2* contains an IF-THEN-ELSE tree followed by another tree. The reward generated in the first tree depends only on the choices made in there and the reward seen in the lower tree is independent of the first. This type of program might be written when the input is used to solve two independent sub-tasks, each with its own reward. *Program S3* contains a sequence of independent trees. The upper tree has a loop with a tree inside it. The lower tree has a similar structure but the reward depends on the context stochastically.

2.6.1 Results

We evaluate three algorithms: 1) SARSA(λ) applied to the standard decision process, which is representative of prior work, 2) SARSA(λ) applied to the informed decision process, and 3) iSARSA(λ) applied to the informed decision process. We used a constant learning rate of $\alpha = 0.01$ and ϵ -greedy exploration with $\epsilon = 0.1$. For each synthetic adaptive program we

conducted 20 learning runs of each learning algorithm and averaged the resulting learning curves. Each learning run consisted of repeated program executions during which the learning algorithm adapts the policy. After every 10 program executions, learning was turned off and the current policy was evaluated by averaging the total reward of another 10 program executions, which are the values we report. For Yahtzee we evaluate every 1000 games by averaging the score of 100 games and average over 20 learning runs. For the SeqL program, we use 1000 passes over the training set, evaluating performance (i.e. number of correct labels) on the *training* set after every 10 passes. At the end of the program run, we evaluate performance on the test set. This is done over 5 learning runs, each one using different datasets generated by the HMM. [Figure 2.2\(a-e\)](#) show the averaged learning curves when using $\lambda = 0.75$ for all algorithms on all benchmarks. [Figure 2.2\(f\)](#) shows the average number of correct labels per sequence over the test set for SeqL.

Benefit of Informed Decision Process. For all example programs, we see a small improvement in learning efficiency for SARSA(λ) when learning in the informed process versus the standard process. This shows that the additional information in the informed process is useful, however, the relatively small improvement indicates that the credit assignment problem in the informed process is still difficult.

Benefit of iSARSA(λ). Now compare learning in the informed process with SARSA(λ) versus iSARSA(λ). For each of our programs there is a substantial boost in learning efficiency when using iSARSA(λ). Furthermore, for the Yahtzee, SeqL and S1(loop) programs, it appears that iSARSA(λ) leads to substantially better steady state performance in the informed process than SARSA(λ). All of these programs contain loops with independent iterations, which makes credit assignment difficult with standard eligibility traces, but much easier with the informed traces of iSARSA(λ). These results give strong evidence that iSARSA(λ) is able to effectively leverage information about the program structure for faster learning. Particularly impressive is its performance on the test set in SeqL, in [Figure 2.2\(f\)](#) where on average, it predicts 40% more labels correctly for $\lambda = 0.75$.

Varying λ . We ran similar experiments for $\lambda = 0$ (pure TD-learning) and $\lambda = 1$ (pure Monte Carlo). The general observation of these experiments is that the relative performance of the methods is similar to $\lambda = 0.75$. Further, iSARSA(λ) is much less sensitive to the value of λ than SARSA(λ) in the informed process, which was less sensitive to λ than SARSA(λ) in the standard process. This shows that the additional information used by iSARSA and contained in the informed process have a large benefit in terms of addressing the credit assignment problem, which is traditionally dictated by the precise value of λ .

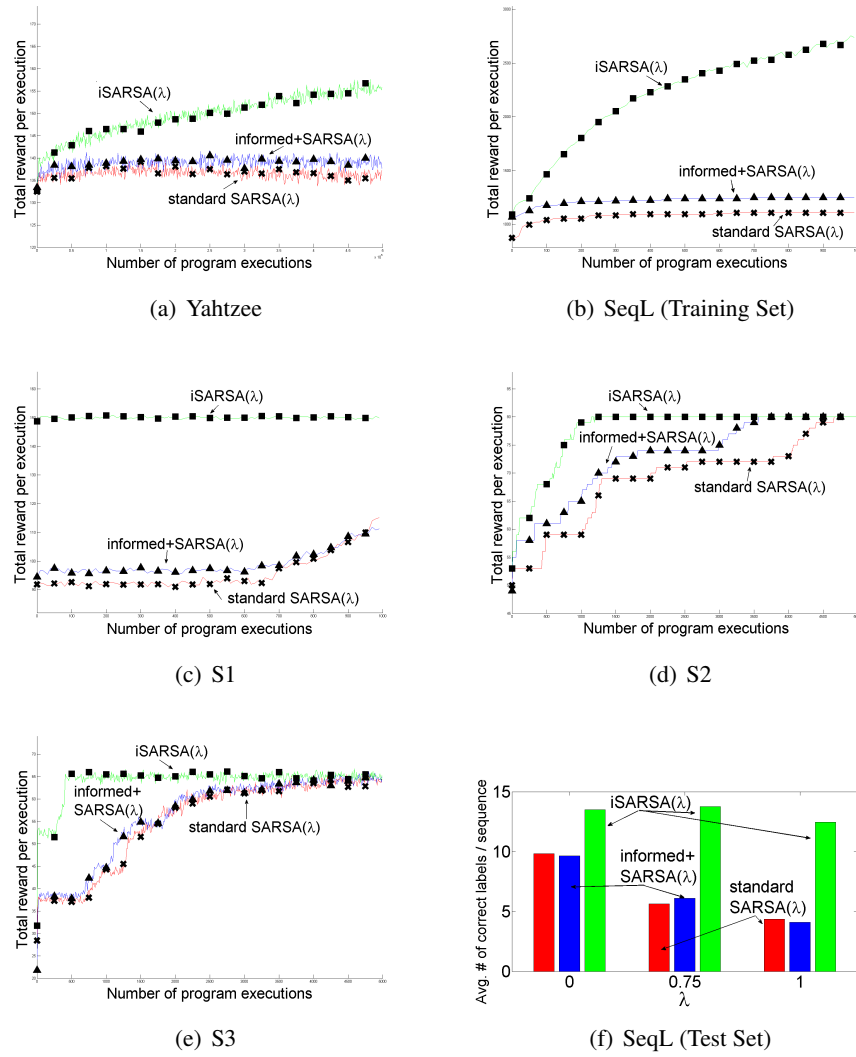


Figure 2.2: (a-e) are performance graphs of programs Yahtzee, SeqL (training set), S1,S2,S3 for $\lambda = 0.75$. (f) SeqL (test set) for varying λ and algorithms.

Overall our results give strong evidence that: 1) Learning in the informed process is beneficial compared to learning in the standard process, and 2) iSARSA(λ) is able to leverage program structure to dramatically improve learning compared to pure SARSA(λ) in the informed process.

2.7 Related Work

There have been several prior and similar ABP efforts, sometimes under the name partial programming [46, 4, 5, 60]. Most notably ALISP [5] extends LISP with choice structures, which are similar to adaptives in our library. There is an important semantic distinction between our work and prior work such as ALISP. The semantics of ALISP are tied to an interface to an external MDP (e.g. which produces reward). This coupling to MDP theory is a hurdle for non-RL expert programmers. Rather, our ABP semantics are not tied to the notion of an external MDP, but defined completely in terms of a program and a distribution over its inputs. This makes our library immediately applicable in any context that Java programs might be written.

Prior learning algorithms for partial programming make very limited use of the program structure, which is the key contribution of our work. The only exception is work on ALISP that leverages program structure in order to decompose the value function according to subroutines [5]. While this is a powerful mechanism it is orthogonal to the type of structure exploited in our work. In particular, the program structure that we exploit can be present within a single sub-routine and does not require a sub-routine decomposition. Combining our work with decomposition based on sub-routines is an interesting direction.

ABP should not be confused with work on RL frameworks/libraries (e.g. RL-Glue[66]). Such libraries are intended for *RL experts* to develop, test and evaluate RL algorithms and provide a language-independent, standardized test harness for RL experiments. Rather, ABP is intended to be a more thorough integration of learning into a programming language, allowing arbitrary programmers to benefit from learning algorithms by simply including adaptive constructs into their programs in a natural way.

2.8 Summary

We have highlighted the fact that subtle differences in adaptive programs can lead to large differences in the performance of standard RL. For non-RL experts, this seriously impedes their ability to use the ABP paradigm. To address this issue we showed how to leverage program structure

to both define a new induced decision process and inform the SARSA(λ) RL algorithm. The results show that this approach leads to good performance on complicated adaptive programs, both real and synthetic.

Chapter 3: Improving Policy Gradient Estimates with Influence Information

3.1 Introduction

Standard off-the-shelf reinforcement learning (RL) algorithms often do not scale well for large, complex sequential decision making problems. This has led to research that extends RL algorithms to leverage various types of additional inputs, or *side information*, in order to accelerate learning. For example, such side information has included shaping rewards [44] and hierarchical task structures in the form of MAXQ hierarchies [21] or ALISP programs [4]. While often effective, such information may require significant human expertise and only captures a subset of the potentially useful side information that might be available in a domain.

In this paper, we consider a form of side information that has not yet been exploited for RL, yet is often available at little to no cost to a designer. In particular, we consider side information in the form of assertions about context-specific influences and independencies among state variables, decisions, and rewards. Such assertions are often easily elicited from a domain expert or can be automatically computed given a partial, qualitative domain model. Further, in our motivating application of Adaptation-Based Programming (ABP), where RL is used to optimize the performance of programs, the independencies can be automatically extracted using standard program analysis techniques with no extra effort from the user (see [section 3.6](#)). As a simpler example to help motivate this form of side information, consider the following illustrative RL problem, which will be used as a running example throughout the paper.

Illustrative Example. Consider a robot moving in a grid world where certain grid cells have buttons that can be pressed to receive rewards. On even numbered time-steps, the robot may either move one step in any cardinal direction or “do-nothing”, whereas at odd numbered time-steps the robot may either choose to execute a “press” action or “do-nothing”. The state of the process is given by the robot’s current location (X, Y) , a binary variable B indicating whether there is a button at the current location, and a binary flag O which is 1 at odd time steps and 0 otherwise, indicating which type of action is applicable. The press action results in a fixed negative reward in a button-less cell, and a reward of $x + y$ in a cell (x, y) that has a button. In all other cases, zero reward is obtained. The objective is to maximize the sum of rewards over T

steps.

Now suppose that reward r_t is obtained after a button press at an odd time step t . A traditional RL algorithm must solve the credit assignment problem of deciding which previous decisions had an “influence” on the value of r_t . However, it is straightforward from the above description to identify which decisions are provably irrelevant to the value of r_t . In particular, all previous “move” decisions can potentially change the location of the robot which affects the reward so all decisions at even-numbered time-steps are potentially relevant as well as the “push” decision at time t . However, the remaining decisions at prior odd time steps (“press” or “do-nothing”) cannot influence the value of r_t . Intuitively, providing such influence information to an RL algorithm has the potential to significantly simplify the credit assignment problem and hence speedup learning. In the absence of such information, standard RL will need to learn through trial and error that button presses deserve no credit for rewards at later time steps. Unfortunately, no existing RL algorithms are able to leverage side information about such influences, or lack of influences, when available.

Contributions. Our main contributions in this work are: 1) A formalization of the above type of side information, 2) A modified policy gradient estimator that uses this information to speedup policy-gradient RL, 3) A proof that the new estimator is unbiased and can have reduced variance, and 4) A demonstration of the approach in the context of Adaptation-Based Programming, showing significant benefits compared to standard policy-gradient without side information.

In what follows, [section 3.2](#) describes the standard model for sequential decision processes and reviews a well-known policy gradient learning algorithm, which is the basis of our approach. In [section 3.3](#), we formalize the notion of “influence” and then in [section 3.4](#) describe a simple modified policy gradient algorithm that takes this information into account. In [section 3.5](#), we analyze the bias and variance of the new algorithm. Next we describe the application of the new approach in the context of adaptation-based programming ([section 3.6](#)) followed by experimental results in this domain ([section 3.7](#)). Finally, we discuss related work and conclude¹.

3.2 MDPs and Policy Gradient Methods

We consider sequential decision making in the framework of Markov Decision Processes (MDPs) [[12](#), [63](#)]. An MDP is a tuple $\{X, A, P, R, P_0\}$ where X is a set of possible states, A is the set

¹A version of this work appeared at ACML 2011 [[50](#)].

of actions, and P is a transition probability function giving the probability $\Pr(x_{t+1}|x_t, a_t)$ of a transition to state x_{t+1} when action a_t is taken in state x_t . Finally, R is a reward function that maps states to real-valued rewards and P_0 is a distribution over initial states. In this work, we are interested in solving large MDPs where the states are represented in terms of M state variables. Thus, the state at time t will be denoted by an M -dimensional vector $x_t = [x_t^1, x_t^2, \dots, x_t^M]$. We will also denote the decision made at time t as d_t , which can take the value of any action. For the rest of the paper, we use upper case symbols to denote random variables (e.g. X_t or D_t) and lower-case for concrete values (e.g. x_t or d_t).

Each decision D_t is made by a policy π_θ , which is a stochastic function from X to A that is parameterized by a vector $\theta \in \mathbb{R}^K$. Our approach will not make any assumptions about π_θ other than that it is a differentiable function of θ . For example, it might be a log-linear probability model or a neural network over a set of state features. Executing π_θ for a horizon of T steps starting in initial state x_0 drawn from P_0 produces a random trajectory of states, decisions, and rewards

$$H = \{X_0, R_0, D_0, X_1, R_1, \dots, D_T, X_T, R_T\} \quad (3.1)$$

and we will denote the expected sum of rewards on such trajectories as $\eta(\theta) = \mathbf{E}(\sum_t R_t)$. The learning goal we consider in this work is to find policy parameters that maximize the value of $\eta(\theta)$. We do this in an RL setting where the MDP model is unknown and the algorithm must learn via direct interaction with the environment.

3.2.1 Policy Gradient RL

We consider a policy-gradient RL approach [72, 11], where the main idea is to estimate the gradient $\eta(\theta)$ with respect to the parameters θ and to then adjust the parameters in the direction of the gradient. The primary difficulty lies in estimating the gradient via direct interaction with the environment for which there is a significant literature. We now review the well-known likelihood ratio gradient estimator, which is the basis of our approach [72].

Given any trajectory $h = \{x_0, r_0, d_0, x_1, \dots, d_T, x_T, r_T\}$ of states x_i , decisions d_i , and rewards r_i we let $r(h) = \sum_{t=0}^T r_t$ denote the sum of rewards along the trajectory and $q_\theta(h)$ denote the probability of generating h using policy π_θ . The policy value can now be written as,

$$\eta(\theta) = \sum_h q_\theta(h) r(h)$$

where the summation is over all length T trajectories. Taking the gradient of this expression and applying the likelihood ratio method we get the standard form for $\nabla\eta(\theta)$ [72].

$$\nabla\eta(\theta) = \mathbf{E} \left[\sum_{t=0}^{T-1} \frac{\nabla\pi_\theta(D_t|X_t)}{\pi_\theta(D_t|X_t)} \sum_{j=t+1}^T R_j \right] \quad (3.2)$$

where X_t , D_t , and $R_t = R(X_t)$ are random variables that respectively denote the state, decision, and reward at time t when following π_θ . The utility of this expression is that it allows for a simple unbiased estimator of the gradient to be calculated without knowledge of the model. This is done by estimating the expectation with an average taken over N independent trajectories $\{h(1), h(2), \dots, h(N)\}$ obtained by following π_θ in the environment as follows,

$$\hat{\nabla}\eta(\theta) = \frac{1}{N} \sum_{n=1}^N \left(\sum_{t=0}^{T-1} \frac{\nabla\pi_\theta(d_{n,t}|x_{n,t})}{\pi_\theta(d_{n,t}|x_{n,t})} \sum_{j=t+1}^T r_{n,j} \right) \quad (3.3)$$

where $d_{n,t}$, $x_{n,t}$, and $r_{n,t}$ are respectively the t 'th decision, state, and reward of trajectory h_n . For large horizons T this gradient estimate can have large variance, which is commonly addressed by using a discount factor $\beta \in [0, 1)$ in order to reduce the credit given to decisions for distant future rewards, yielding the following modified estimator.

$$\hat{\nabla}\eta(\theta) = \frac{1}{N} \sum_{n=1}^N \left(\sum_{t=0}^{T-1} \frac{\nabla\pi_\theta(d_{n,t}|x_{n,t})}{\pi_\theta(d_{n,t}|x_{n,t})} \sum_{j=t+1}^T \beta^{j-t-1} r_{n,j} \right) \quad (3.4)$$

For $\beta = 1$ this estimator is identical to the above and hence unbiased. However, as β decreases, the variance is reduced but bias can grow, which often has an overall positive impact on computing a useful gradient direction.

3.3 Context-Specific Independence

In this section, we introduce the type of side information that we wish to incorporate into a policy gradient estimator. At a high level, this side information consists of context-specific conditional independencies between the state, decision, and reward variables in the domain. Graphical models such as tree-based dynamic Bayesian networks (DBNs) [15] or acyclic decision diagrams (ADDs) [32] have been commonly employed to compactly represent MDP models by directly encoding context-specific independencies among variables. The common idea behind

these models is that given the context of a state x_{t-1} and decision d_{t-1} at some time $t - 1$, for any state variable V_t , the models identify a set of *parent variables* $\text{Pa}(V_t|x_{t-1}, d_{t-1})$, which can be any variables from time-step $t - 1$.² For reward or decision variable V_t , the context is simply x_t and the parents are denoted by $\text{Pa}(V_t|x_t)$. The intention is that the parents given a context are the only variables that matter with respect to specifying the distribution over values of V_t , with all other variables being conditionally independent. For instance, for our example in [section 3.1](#), the parents of the location variable Y_t change depending on the value of O_{t-1} , which indicates whether the time step is odd or even. If $O_{t-1} = 1$ then no action can change the location (it is a “press” time step) and the parents are $\{Y_{t-1}, O_{t-1}\}$ indicating that these are the only variables that the value of Y_t depends on. Otherwise, when $O_{t-1} = 0$ a move action may change the location yielding a parent set $\{Y_{t-1}, O_{t-1}, D_{t-1}\}$, indicating that the decision at $t - 1$ can influence the value of Y_t .

Since our approach is not tied to a particular type of model such as DBN or ADD, we formulate the notion of independence in terms of a generic parent function, which must satisfy certain general properties with respect to the underlying MDP.

Definition 1 *For state variable V_t , a parent function $\text{Pa}(V_t|x_{t-1}, d_{t-1})$ is consistent with an MDP and parameterized policy representation π_θ if the following two conditions hold:*

1. *For any context (x_{t-1}, d_{t-1}) , there are no cycles in the graph of variables where an arc is drawn to V_t from every variable in $\text{Pa}(V_t|x_{t-1}, d_{t-1})$.*
2. *For any context (x_{t-1}, d_{t-1}) , $\Pr(V_t|x_{t-1}, d_{t-1}) = \Pr(V_t|\text{Pa}(V_t|x_{t-1}, d_{t-1}))$, where the probability is with respect to the MDP transition function.*

Corresponding conditions exist when V_t is a decision or reward variable where the context is now x_t and the parent function is given by $\text{Pa}(V_t|x_t)$. Also, in the second condition, when V_t is a decision variable, the probability is with respect to the policy representation.

Importantly, our approach will not require explicit access to a consistent parent function, but rather only access to a function related to it, which we now formalize. Consider the set of all possible state, decision, and reward variables from time $t = 0$ to $t = T$ and assume that we have

²In most prior work, e.g. [15, 32], decision variables are not explicitly included in the graphical model, meaning that the set of their parents is implicitly assumed to be all state variable (i.e. the policy considers all state variables). Rather, we allow decision variables to have context specific parent functions, where the parents can depend on the specific values of state variables. This allows for policy architectures that need not look at all state variables and where this set (the parents) can vary from state to state.

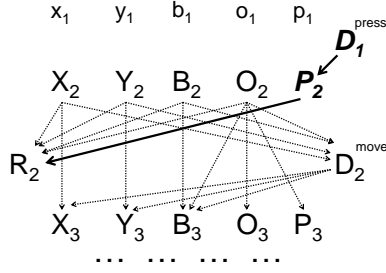


Figure 3.1: The conditional influence graph for “press” decision variable D_1 is shown after the state at $t = 1$ has been observed. Note that D_1 has only a single path of influence $\{D_1, P_2, R_2\}$, as shown by the thick arcs. Thus “press” decisions can only influence the immediate reward whereas move decisions like D_2 have at least one path to every subsequent reward (not shown) via the state variables X_3, Y_3 and B_3 .

observed a trajectory up to time t resulting in state x_t . We are interested in defining the set of future variables that can possibly be “influenced” by the decision D_t at time t given x_t . For this purpose, we construct the following *conditional influence graph*. Draw an arc from variable D_t to any state or reward variable V_{t+1} at time $t + 1$ such that $D_t \in \text{Pa}(V_{t+1}|x_t, D_t = a)$ for some value a of D_t . These are all the variables at time $t + 1$ whose value can possibly be influenced by the choice of D_t . Next for all future times $t' > t + 1$ and for any state, decision, or reward variable $V_{t'}$ we draw an arc to $V_{t'}$ from any variable in $\bigcup_{x,a} \text{Pa}(V_{t'}|X_{t'-1} = x, D_{t'-1} = a)$. Thus, there is an arc from a parent to a child for times $t' > t + 1$ if the parent can potentially influence the child, i.e. there is some way of setting the previous state and decision so that the parent is influential. [Figure 3.1](#) shows a part of the conditional influence graph for the example problem given earlier³.

Using this graph, we now define the *reward influence function* $I(t, x_t, t')$ for a parent function $\text{Pa}()$ as follows:

- $I(t, x_t, t') = 0$, if given x_t there is *no path* in the influence graph for $\text{Pa}()$ from D_t to $R_{t'}$
- $I(t, x_t, t') = 1$, otherwise

From this definition, it is clear that if $I(t, x_t, t') = 0$ then the choice of D_t at time t given x_t can have no influence on the value of the reward at time t' . Our algorithm only requires access to such an influence function, and correctness can be guaranteed even when given access to only an approximation. In particular, we say that an approximate reward influence function $\hat{I}(t, x_t, t')$ is *influence complete* for an MDP if there exists a consistent parent function for the MDP such

³State variable P is introduced so that the rewards are state-dependent only whereas the original description in [section 3.1](#) had state-action dependent rewards. This is done only to maintain consistency with the policy gradient RL formulation in [section 3.2](#). P is 1 only in the time-step immediately after a button is pressed and 0 in all other cases.

that if $I(t, x_t, t') = 1$ then $\hat{I}(t, x_t, t') = 1$. That is, the approximate influence function will always indicate a potential influence when one exists. Stated differently, if an influence function \hat{I} is influence complete then it soundly detects when a reward at time t' is independent of a decision at time t , which is the key property that our algorithm will require. Importantly, while computing the true influence function I may often be difficult or even undecidable, obtaining an influence complete approximation is often straightforward. For example, in our application of adaptation-based programming, an influence function can be computed via program analysis, or if a qualitative model is available that provides an estimate of the parent function, influence can be computed via path finding.

3.4 Policy Gradient with Influence Functions

Suppose that we are given an influence function $\hat{I}(t, x_t, t')$ as a form of side information. Intuitively, such a function can be used to help alleviate the credit assignment problem faced by a policy gradient estimator. To see how this might be done, consider [Equation 3.2](#) which shows that the exact policy gradient can be expressed as an expectation over a sum of terms, one for each decision from $t = 0$ to $t = T - 1$. Each term is simply the likelihood-scaled gradient of the policy's decision at time t weighted by the sum of rewards after that decision is made. Essentially, this weighting strategy assumes that a decision is equally responsible for the value of any future reward. Given this view, if our influence function tells us that the decision at time t is independent of a future reward, then it is plausible that we can soundly drop that reward from the weighting factor of term t . This results in the following modified gradient expression.

$$\nabla \eta_{SI}(\theta) = \mathbf{E} \left[\sum_{t=0}^{T-1} \frac{\nabla \pi_{\theta}(D_t | X_t)}{\pi_{\theta}(D_t | X_t)} \sum_{j=t+1}^T \hat{I}(t, X_t, j) R_j \right] \quad (3.5)$$

As proven in the next section, it turns out that when the influence function is influence complete this expression is equal to the true gradient. Thus, we can obtain an unbiased estimator by sampling trajectories and averaging, arriving at the estimator.

$$\hat{\nabla} \eta_{SI}(\theta) = \frac{1}{N} \sum_{n=1}^N \left(\sum_{t=0}^{T-1} \frac{\nabla \pi_{\theta}(d_{n,t} | x_{n,t})}{\pi_{\theta}(d_{n,t} | x_{n,t})} \sum_{j=t+1}^T \hat{I}(t, x_{n,t}, j) r_{n,j} \right) \quad (3.6)$$

The new estimator $\hat{\nabla} \eta_{SI}(\theta)$ simply disregards the contribution to the gradient estimate of

decision-reward pairs that are judged as being independent according to \hat{I} . When this judgement is correct, the new estimator can significantly simplify credit assignment. When such terms are not ignored, as is the case for standard gradient estimates, it is much more difficult to efficiently and accurately estimate the gradient due to the “noise” introduced by those terms. Overcoming such noise necessitates the use of many more trajectories.

3.4.1 Incorporating Discounting

To help reduce variance we now consider an analogue of the biased gradient estimate from [Equation 3.4](#) that takes the influence function into account. [Equation 3.5](#) reduces variance by discounting rewards with respect to a decision based on the duration between the decision and reward. This is based on the assumption that decisions nearer to a reward are likely to be more influential. However, this assumption can be strongly violated when there are many non-influential actions for certain rewards. For instance, in our running example, move actions which always influence future rewards are discounted more due to the intermediate button presses, all but one of which are irrelevant to a given reward. In general, if there are t time-steps between a relevant decision d and a reward r , then d sees a discounted reward of $\beta^t r$ but if n of those decisions are irrelevant, an intuitively better discount factor would be $\beta^{t-n} r$. That is, a more appealing assumption is to discount based on the number of potentially relevant decisions between a decision and a reward, rather than the total number of decisions as in [Equation 3.5](#).

Using the conditional influence graph described in the previous section, we can define a biased estimator that captures the above intuition. Let $L(t, x_t, j)$ be the number of decisions on the shortest path between D_t and R_j in the influence graph conditioned on x_t and let $\beta(t, j) = \beta^{L(t, x_t, j)}$ be a modified discounting factor. Using this factor in place of the standard discounting approach of [Equation 3.4](#) results in a biased gradient estimate, a necessary consequence of discounting. The intention is that the reduction in variance will result in a net benefit with respect to estimating a useful gradient direction.

Our complete learning algorithm, which incorporates discounting, denoted as PGRI (policy gradient with reward influences) is given in [Algorithm 2](#). This algorithm is an incremental implementation of the gradient estimator and when $\beta = 1$, it corresponds to the unbiased version of the estimator ([Equation 3.6](#)) since updates are only performed when $\hat{I}(i, x_i, j) = 1$. The algorithm updates parameters at every time-step rather than after N trajectories. This is a common practice in policy gradient RL since the number of parameter updates for the episodic update

is quite small compared to the amount of accumulated experience, leading to slow learning, particularly for long trajectories. However, while the sum of the updates made by these online algorithms between recurrent states is in the gradient direction, the individual updates are not in expectation (see discussion in [11]). Nevertheless, while the theoretical understanding of such updates are less well understood, in practice they very often yield superior performance.

Algorithm 2 PGRI.

```

1: Inputs: At time step  $t$ , observed reward  $r_t$ , prior trajectory  $h$ , influence-complete  $\hat{I}$ , current
   param. vector  $\theta$ , discount factor  $\beta$ , learning rate  $\alpha$ 
2: Output: Updated parameter vector
3:  $\Delta\theta = 0$ 
4: for  $i = 1 : t$  do
5:   if  $\hat{I}(i, x_i, j) = 1$  then
6:      $\Delta\theta = \Delta\theta + \frac{\nabla\pi_t(d_i)}{\pi(d_i)} r_t \beta^{L(i, x_i, j)}$   $\triangleright d_i$  is the  $i$ 'th decision in  $h$ 
7:      $\triangleright L(i, x_i, j)$  is the number of decisions on the shortest path between  $d_i$  and  $r_j$ 
8:   end if
9: end for
10:  $\theta = \theta + \alpha\Delta\theta$ 

```

3.5 Algorithm Properties

We now consider the properties of the new estimator when $\beta = 1$, which corresponds to $\hat{\nabla}\eta_{SI}(\theta)$. First, we show that this estimator is unbiased, a property which follows from the result given next.

Theorem 2 *For any reward influence function \hat{I} that is influence complete with respect to the underlying MDP, the following holds: $\nabla\eta_{SI}(\theta) = \nabla\eta(\theta)$.*

Proof Let \mathbf{pa} denote a consistent parent function for which \hat{I} is influence complete with respect to the underlying MDP (see section 3.3). Let $\eta_t(\theta)$ be the expected value of the single reward R_t at time t when following policy π_θ . Thus $\eta(\theta) = \sum_{t=1}^T \eta_t(\theta)$. The sum over length t trajectories h_t can be written as a sequence of sums over the individual state and decision variables,

$$\begin{aligned}
\eta_t(\theta) &= \mathbf{E}[R_t] = \sum_{h_t} q_\theta(h_t) r_t \\
&= \sum_{x_0^1} q(x_0^1) \sum_{x_0^2} q(x_0^2) \dots \sum_{d_0} \pi_\theta(d_0|x_0) \sum_{x_1^1} q(x_1^1|x_0, d_0) \dots \sum_{x_t^M} q(x_t^M|x_{t-1}, a_{t-1}) r_t(x_t) \\
&= \sum_{x_0^1} q(x_0^1) \sum_{x_0^2} q(x_0^2) \dots \sum_{d_0} \pi_\theta(d_0|\mathbf{pa}) \sum_{x_1^1} q(x_1^1|\mathbf{pa}) \dots \sum_{x_t^M} q(x_t^M|\mathbf{pa}) r_t(\mathbf{pa})
\end{aligned} \tag{3.7}$$

$$\tag{3.8}$$

where $q(x_i^m|\mathbf{pa}) = \Pr(x_i^m|\mathbf{pa})$ and for clarity we have not shown the arguments to the parent function. Equation 3.8 is the result of applying the properties of consistent parent functions to Equation 3.7. Taking the gradient of Equation 3.8 w.r.t. θ gives us $(t+1) \cdot M$ sums for $t+1$ states and another t sums for the decision variables. Since the probabilities of state transitions are not parameterized by θ , $\nabla q(x_i^m|\mathbf{pa})$ is zero for all i and m and we are left with a sum of t terms where a single $\pi_\theta(d_i|x_i)$ appears differentiated in each term,

$$\nabla \eta_t(\theta) = \sum_{i=0}^{t-1} \sum_{x_0^1} q(x_0^1) \sum_{x_0^2} q(x_0^2) \dots \sum_{d_i} \nabla \pi_\theta(d_i|\mathbf{pa}) \sum_{x_1^1} q(x_1^1|\mathbf{pa}) \dots \sum_{x_t^M} q(x_t^M|\mathbf{pa}) r_t(\mathbf{pa}) \tag{3.9}$$

The crucial step in this proof is based on the observation that given history upto the decision D_i , we may re-order the sums over the remaining terms as long as we use a topological ordering of the variables w.r.t. the conditional influence graph. Using the arc-drawing procedure from section 3.3 that was used to define the influence graph and reward influence function, we can partition the variables after D_i into two sets: U which includes those variables that are unreachable from D_i and $\neg U$ for the rest. Denote by $\nabla \eta_t^i(\theta)$ the i 'th component of $\nabla \eta_t(\theta)$ where $\nabla \eta_t(\theta) = \sum_{i=0}^{T-1} \nabla \eta_t^i(\theta)$.

Given the above definitions we can derive that the variables in U denoted by u_1, u_2, \dots can be summed over before the variables in $\neg U$ denoted by $\neg u_1, \neg u_2, \dots$ provided the variables in each set are topologically ordered w.r.t. the conditional influence graph. That is,

$$\nabla \eta_t^i(\theta) = \sum_{h_i} q(h_i) \sum_{u_1} q(u_1|\mathbf{pa}) \sum_{u_2} \dots \sum_{d_i} \nabla \pi_\theta(d_i|\mathbf{pa}) \sum_{\neg u_1} q(\neg u_1|\mathbf{pa}) \sum_{\neg u_2} q(\neg u_2|\mathbf{pa}) \dots r_t(\mathbf{pa})$$

where the first sum is over the history h_i up to time i . This holds based on the following argument. Since the conditional influence graph is acyclic, there must exist a topological ordering of the variables in U which we denote by $\{U_1, U_2, \dots\}$. Similarly, we have a topological ordering of the variables in $\neg U$. For any variable $U_i \in U$, the probability distribution $q(U_i|\mathbf{pa})$ becomes well defined given h_i and values for the variables in U that precede U_i in the topological ordering⁴ and correspondingly for the variables in $\neg U$. Since the reordering of the summation terms satisfy such an ordering and the parent function is consistent with the MDP, all the probability terms remain well-defined (and unchanged) which is all that is needed for the equation to hold.

Now consider the case where the influence complete influence function asserts $\hat{I}(i, x_i, t) = 0$. This implies r_t is unreachable from D_i in the influence graph given x_i . Also, by the definition of $\neg U$, r_t is unreachable from every variable in $\neg U$. Therefore, when $\hat{I}(i, x_i, t) = 0$, r_t is conditionally independent of D_i and every variable in U given h_i and values of the variables in $\neg U$. This allows us to move r_t to the left of d_i in the summation as shown,

$$\nabla \eta_t^i(\theta) = \sum_{h_i} q(h_i) \sum_{u_1} q(u_1|\mathbf{pa}) \sum_{u_2} \dots r_t(\mathbf{pa}) \sum_{d_i} \nabla \pi_\theta(d_i|\mathbf{pa}) \sum_{\neg u_1} q(\neg u_1|\mathbf{pa}) \sum_{\neg u_2} \dots \quad (3.10)$$

The summations over the $\neg u_i$ now simply become 1. Further, since π_θ is a probability distribution over decisions we have that,

$$\sum_{d_i} \nabla \pi_\theta(d_i|\mathbf{pa}) = \nabla \sum_{d_i} \pi_\theta(d_i|\mathbf{pa}) = \nabla 1 = 0.$$

This shows that the entire sum must be zero. Thus, $\hat{I}(i, x_i, t) = 0$ implies $\nabla \eta_t^i(\theta) = 0$ and hence,

$$\nabla \eta_t(\theta) = \mathbf{E} \left[\sum_{t=0}^{T-1} \frac{\nabla \pi_\theta(D_t|X_t)}{\pi_\theta(D_t|X_t)} \hat{I}(t, X_t, t) R_t \right] \quad (3.11)$$

where we have re-written Equation 3.9 in the likelihood ratio form. Finally, it is a straightforward manipulation to combine the gradient estimates for the reward at each time-step into the full gradient.

⁴For the decision variables in U , $\pi_\theta(d_i|\mathbf{pa})$ becomes well-defined.

$$\nabla_{\eta}(\theta) = \sum_{t=1}^T \nabla_{\eta_t}(\theta) = \nabla_{\eta_{SI}}(\theta). \quad (3.12)$$

■

Corollary 3 *For any reward influence function \hat{I} that is influence complete with respect to the underlying MDP, the estimator in Equation 3.6 is an unbiased estimate of $\nabla_{\eta}(\theta)$.*

3.5.1 Variance Reduction

We now consider the variance of the new estimator for $\beta = 1$ compared to the standard estimator that ignores the influence function. From Equation 3.3 and Equation 3.6, we see that for any set N of independently sampled trajectories H , the gradient estimate $\hat{\nabla}_{\eta}(\theta)$ can be written as the sum of $\hat{\nabla}_{\eta_{SI}}(\theta)$ and a residual term $\text{Res}(\theta)$ which is the sum of all terms in $\hat{\nabla}_{\eta}(\theta)$ that are ignored by our new estimator (i.e. when $\hat{I} = 0$). Note that the dependence on H is left implicit. From this we get the following relationship between the variances of the two estimators.

$$\begin{aligned} \text{Var} \left(\hat{\nabla}_{\eta}(\theta) \right) &= \text{Var} \left(\hat{\nabla}_{\eta_{SI}}(\theta) + \text{Res}(\theta) \right) \\ &= \text{Var} \left(\hat{\nabla}_{\eta_{SI}}(\theta) \right) + \text{Var} (\text{Res}(\theta)) + 2\text{Cov} \left(\hat{\nabla}_{\eta_{SI}}(\theta), \text{Res}(\theta) \right) \end{aligned}$$

The variance of $\nabla_{\eta_{SI}}(\theta)$ will be less than that of the standard estimator $\nabla_{\eta}(\theta)$ as long as,

$$\text{Var} (\text{Res}(\theta)) > -2\text{Cov} \left(\hat{\nabla}_{\eta_{SI}}(\theta), \text{Res}(\theta) \right)$$

Since the variance of the residual is always positive this will only be violated when there is a strong negative correlation between the residual and $\nabla_{\eta_{SI}}(\theta)$. While it is possible to construct pathological MDPs where this can occur, it is very difficult to find a realistic MDP where this occurs. In typical domains that we have considered it is the case that the magnitude of the residual variance dominates the magnitude of the correlation term, which implies reduced variance for the new estimator. For example, in all of our applications domains it is not hard to verify that dominating negative correlation does not occur.

3.6 Adaptation-Based Programming (ABP)

The potential benefits of incorporating influence information are clearly visible in the application domain of Adaptation-Based Programming (ABP), which is therefore the focus of our experimental results. ABP (sometimes called “partial programming”) integrates reinforcement learning (RL) into a programming language with the intention of allowing programmers to represent uncertainty in their programs. ABP permits specific decisions in the *adaptive program* to be left unspecified. Instead, the programmer provides a reward signal as a measure of program performance, which the ABP system uses to learn a policy for the open decision points. Our work is built on an existing ABP Java library [10], where the main construct is the `adaptive` object, which represents an open decision point. Given some information (context) representing the current program state, the adaptive can suggest an action, chosen from a set of specified actions. Another way to view an adaptive is a mapping from a context to an action which needs to be learned over time in order to optimize an objective. Figure 3.2 contains simple examples of adaptive programs reproduced from [49]. P1 is a conditional structure where `A`, `B` are adaptives representing uncertainty in how to pick a branch. `A` and `B` have fixed context and boolean actions. The `suggest` method returns either `true` or `false`. The notion of optimizing a performance objective motivates the second important construct: `reward(r)`. In order to measure program performance, ABP allows the programmer to place reward statements anywhere in the program, where numerically larger rewards indicate better performance. For example, in P1, the reward statements measure the desirability of each path from root to leaf. [10] contains a detailed description of the syntax and semantics of each ABP construct and examples of realistic adaptive programs.

Learning Problem: Given an adaptive program P , we want to learn a *choice function* for each adaptive in P , which is a mapping from an adaptive’s context type to its action type. A *policy* π for P is a set of choice functions, one for each adaptive in P . Executing P on input x using policy π results in a sequence of `suggest` calls where each call to adaptive A is handled by its choice function in π . Also encountered are a sequence of deterministic `reward` statements. Let $R(P, \pi, x)$ denote the sum of these rewards. Assuming input x is drawn from some probability distribution D , the goal of learning is to find a policy π that maximizes $\mathbf{E}_D[R(P, \pi, x)]$, the expected sum of rewards⁵ over a single execution of P . Computing the

⁵This assumes that each execution terminates in a finite number of steps resulting in a finite reward sum. If this is not the case, we can instead maximize the *discounted* infinite sum of rewards.

```

a = A.suggest();
b = B.suggest();

if (test()) {
  if (A.suggest()) {
    reward(1);
  } else {
    reward(2);
  }
} else {
  if (B.suggest()) {
    reward(2);
  } else {
    reward(1);
  }
}
Program P1

if (test()) {
  if (a) {
    reward(1);
  } else {
    reward(2);
  }
} else {
  if (b) {
    reward(2);
  } else {
    reward(1);
  }
}
Program P2

for (i=1; i<N; i++) {
  c = randomContext();
  m = move.suggest(c);
  reward(payoff(c,m));
}
Program P3

```

Figure 3.2: Illustrative adaptive programs reproduced from Figure 2.1. `test`, `randomContext`, and `payoff` are non-adaptive methods. `reward` and `suggest` are part of the ABP library [10].

optimal value of π analytically is infeasible for non-trivial adaptive programs. Instead, the ABP system attempts to learn a good, possibly optimal, policy through multiple executions of P with inputs drawn from D . We now illustrate the potential for leveraging influence information in ABP and review recent work [49] on the topic.

Motivation and Prior Work: Consider the adaptive programs in Figure 3.2. P1 is a conditional structure with two adaptives (A and B) at the inner nodes and rewards at the leaves. This is an easy problem to solve optimally for even simple RL algorithms, since on each execution a single `suggest` call is made to one of the adaptives and a single reward observed. Now consider P2 which is identical except the choices are made at the very top of the program and stored in `a` and `b` which are used later. On every execution, the learner now sees two choices, one of which is spurious. This is a harder credit assignment problem since the learner cannot see which branch was actually taken but only the sequence of adaptive calls and rewards. However, P1 and P2, both valid ABP programs, are equivalent from the *programmer's point of view*. A very different learning problem is induced by P3 which is a simple bounded loop. On each iteration, it makes a single choice based on a randomly generated context, producing a reward. Each reward is generated independently and only depends on the most recent decision. However, a standard learning algorithm has no way of detecting this and instead sees a very noisy reward signal containing the sum of future rewards, which dramatically complicates the credit assignment problem.

The solution proposed by [49] is based on the observation that there is information useful for simplifying credit assignment in the structure of the adaptive program itself. Their method relies on existing program analysis techniques that extract program structure information, abstracted as an oracle that can compute two properties, *value dependence* and *path irrelevance*, of an adaptive program during its execution. In order to leverage this analysis as *side information* for an RL process, the SARSA(λ) RL algorithm [63] was modified so that the eligibility parameters were sensitive to the side information. As one example, the eligibility of a decision was reset to zero whenever it becomes path independent of the current execution point of the program. While this approach was shown to significantly outperform SARSA(λ) without the use of side information, the semantics of the algorithm are unclear. Further, even when restricted to just policy evaluation, the updates to eligibility values fail to satisfy a condition necessary for convergence [12]. In general, the algorithm lacks theoretical characterization and is a somewhat ad-hoc solution to the problem of using side information about program structure in RL.

A more fundamental issue concerns the use of value function-based learning algorithms in ABP systems. Previous work such as ALISP [4, 5] provides convergence guarantees, but only by making strong assumptions about the abstractions used to represent value functions at choice points. In practice, for programs written by non-RL experts, it will be rare to have such conditions satisfied and there will typically be substantial partial observability for the learner. In such cases, value, or TD-based methods, can be problematic. Thus, in this work we consider policy gradient-based methods, which are arguably a more principled way for dealing with partial observability, which is inevitable in ABP.

Policy Gradient for ABP: Since we assume that our adaptive programs always terminate, it is straightforward to apply the policy gradient algorithms presented in [section 3.2](#) and [section 3.4](#). The process being controlled corresponds to the adaptive program with decision points occurring whenever an adaptive is reached. The underlying process state corresponds to the entire program state (stack, memory, counter, etc) and the observations correspond to the context made available by the programmer at each adaptive. We assume that the context c of each adaptive A is a user defined feature vector that is computed each time the adaptive is encountered. For example, if the decision corresponds to making a particular choice in a game, then the context would encode game features relevant to that choice.

As discussed previously, a policy π for an adaptive program is the set of choice functions and as such we parameterize our policies by a parameter vector θ which is the concatenation of parameter vectors for each choice function. The parameters corresponding to adaptive A are

denoted by θ_A , which are further assumed to have components specific to each action a , denoted by $\theta_{A,a}$. We use a standard log-linear stochastic policy $\pi_\theta(a, A, c)$ which defines the probability of selecting action a at adaptive A given context c and parameters θ as,

$$\pi_\theta(a, A, c) = \frac{\exp(c \cdot \theta_{A,a})}{\sum_{a'} \exp(c \cdot \theta_{A,a'})} \quad (3.13)$$

Given this functional form, we can directly apply the likelihood-ratio approach for estimating the performance gradient $\eta(\theta) = \mathbf{E}_D[R(P, \pi_\theta, x)]$ based on a set of trajectories generated by executing the program using parameters θ . These estimates can then be used to conduct stochastic gradient descent by taking steps in the estimated gradient directions.

We are yet to discuss how we construct an influence complete approximation of the reward influence function for the PGRI algorithm. This is actually the easiest part since it turns out that we can use the same program analysis from [section 2.3](#) and [\[49\]](#) to define a suitable approximation. The easy transformation of program analysis information into an (approximate) influence function is not a coincidence as a close correspondence seems to exist between the conditional influence graph over an MDP and the control-flow/ data-flow graph structure computed during program analysis. As an example of the program analysis capabilities, it is able to infer that in program P2 the decision made by adaptive B does not influence the final reward in program trajectories where `test()` evaluates to true as desired. Following the approach in [\[49\]](#), we annotate the adaptive programs by hand with the same outputs that the analysis would produce.

Combining the program analysis with the PGRI algorithm results in an ABP system for non-ML-experts that is both provably convergent and leverages program structure for faster learning. Our experiments on large adaptive programs demonstrate the utility of taking influence information into account.

3.7 Experiments

We compare the performance of the PGRI algorithm against the baseline approach in [Equation 3.4](#) on a set of adaptive programs. We evaluate β -dependence, convergence rate and the performance of the learned policy. Each adaptive program is executed multiple times during which it learns. After every 10 executions, we switch off learning and evaluate the learned behavior on 10 executions (for programs with stochastic rewards) and report the total reward obtained during evaluation, during which the standard practice of executing the highest proba-

bility action is used. Finally, all results are averaged across 10 runs to eliminate random effects and initialization bias. The learning rate for both algorithms is set to 10^{-5} and kept constant throughout. We now briefly describe the adaptive programs and associated credit assignment issues.

Sequence Labeling (SeqLabel): An important problem in ML, sequence labeling involves the labeling of each character of an input string. Here, we use input sequences of length 20, generated by a Hidden Markov Model. Both character and label sets have cardinality 10. The training set contains 500 labeled sequences while the test set contains 200. The adaptive program uses a single adaptive to suggest a label for each input in each sequence in the dataset, using the character and previous label as context. A reward of +1 is given after each correct prediction. Each program run consists of one pass through the dataset. Ideally, the programmer would indicate an independency between labels for consecutive sequences via an ABP library method. However, our informal studies have shown that programmers routinely fail to identify such independencies or forget to indicate them resulting in a noisy reward signal. Our algorithm is unaffected by this since program analysis detects that all the choices made during the labeling of one sequence either fall out of scope or get over-written at the end of a sequence, terminating their influence on future decisions.

Multi-label sequence labeling (NetTalk): Here, each input sequence has multiple label sequences. We use a subset of the NetTalk dataset [58] which consists of English words and label sequences for 1) the phonemes used to pronounce each word and, 2) the stresses placed on it. There are 50 phonemes and 5 stresses. Instead of predicting labels directly, the program uses output coding [30] as follows: Every label symbol is encoded using 10 bits and a different adaptive predicts the bit at each position giving a total of 20 adaptives. The context for the adaptive at bit position i consists of two sliding windows of length five, one over the input sequence and the second over the previous predictions at position i . A reward of +1 is given for each correct bit prediction. Both training and test sets contain 500 sequences with some overlap. This program poses a harder credit assignment problem since the two label predictions are now inter-leaved besides the inter-sequence noise.

Resource gathering (ResGather): We consider the resource gathering problem in a Real-Time Strategy (RTS) game where a number of peasants must collect and deposit as much gold as possible within a fixed amount of time. The map consists of a grid with a single base and mine at fixed locations. Each peasant must move to the mine and dig there and return to the base to deposit the gold. A single adaptive controls all the peasants. After all the peasants have acted, the

“world” returns a vector of the amounts of gold deposited by each peasant. Although the optimal policy in this scenario is obvious, the learning problem is challenging even for a small number of peasants and map since the rewards are zero almost everywhere and the decision sequences are interleaved which makes credit assignment very hard. Program analysis can associate a peasant with the deposited gold, which our algorithm is able to exploit to perform better credit assignment.

Synthetic Programs: We also experiment with three synthetic adaptive programs adapted from [49]. The first program S1 is the loop program P3 where N is set to 1000 and the adaptive can have one of 10 contexts. The second program S2 consists of two decision trees in sequence with choices at the root and internal nodes and stochastic rewards at the leaves. All three decisions are made at the top leading to partial observability. The third synthetic program S3 is more complex and consists of multiple nested trees and loops. The outer structure is a tree containing decision trees in some branches and loops in others. Stochastic rewards of different magnitudes are scattered all over the program. Taken together, these programs represent common usage patterns in adaptive programs.

Effect of discarding irrelevant rewards: Figure 3.3 shows the performance of PGRI compared to the baseline method (PG) without discounting for all six programs. Our algorithm clearly does better than the baseline on all programs both in terms of convergence rate and the eventual learned performance, indicating the usefulness of side information. On problems where the primary source of noise in the reward signal comes from the independence between reward streams (SeqLabel, NetTalk), the baseline fails completely. While S1 and S3 also present a similar challenge, they are somewhat easier learning problems and the baseline is able to learn, though significantly more slowly than PGRI. The remaining programs (S2, ResGather) do not contain this kind of noise and the baseline eventually learns a policy as good as PGRI.

Effect of discounting: Next we evaluate the effect of varying β for a number of values. The results are shown in Figure 3.4 for two representative values of β and three programs that show interesting trends to avoid cluttering the figures. The remaining programs are relatively robust to values of β that are not too high. The figures clearly show the baseline’s sensitivity to the value of β . This is expected since the different programs pose “contrary” credit assignment problems where a large discount factor helps for some programs and hurts for other programs. For example, in the NetTalk programs, updating choices based on a large number of future rewards makes the problem harder while the opposite is true for the resource gathering problem. The baseline quickly becomes invariant on the SeqLabel (and NetTalk) programs as β decreases

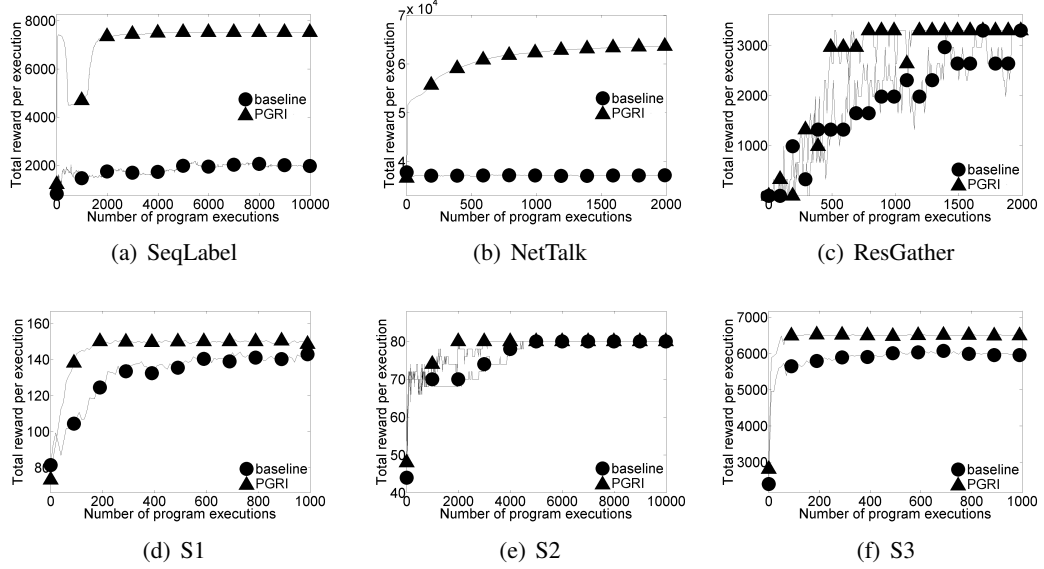


Figure 3.3: Effect of discarding irrelevant rewards. The figure shows the performance graphs of adaptive programs without any discounting. The vertical axis is the total reward over a single execution. The horizontal axis shows the number of program executions.

from one. However, it fails completely on ResGather even for the moderately high value of $\beta = 0.75$. Across all programs, the PGRI algorithm is more robust to β than the baseline, a desirable quality since users of ABP systems (non-experts in RL) are not expected to be able to tune the learning algorithm.

Finally, [Figure 3.5](#) shows the classification accuracy before and after learning on the NetTalk training set and the final evaluation on the test set since the real objective in this domain is to obtain accurate sequence labels, not binary labels. As expected, the baseline fails on $\beta = 1$ but learns as well as our algorithm for $\beta = 0.75$. Our method is practically invariant to β on this problem. While stresses are eventually predicted accurately, predicting phonemes is a much harder problem due to the large number of possible labels and learning is very slow. Note that structured prediction algorithms (e.g., HC-search [22]) are likely to be better choices for this particular problem. However, the goal of these experiments is to demonstrate that a relatively simple adaptive program is able to do reasonably well on an important and challenging problem.

Overall, these results clearly demonstrate the benefits of our method over a standard policy gradient method. Although we do not show it, the PGRI method demonstrates considerably

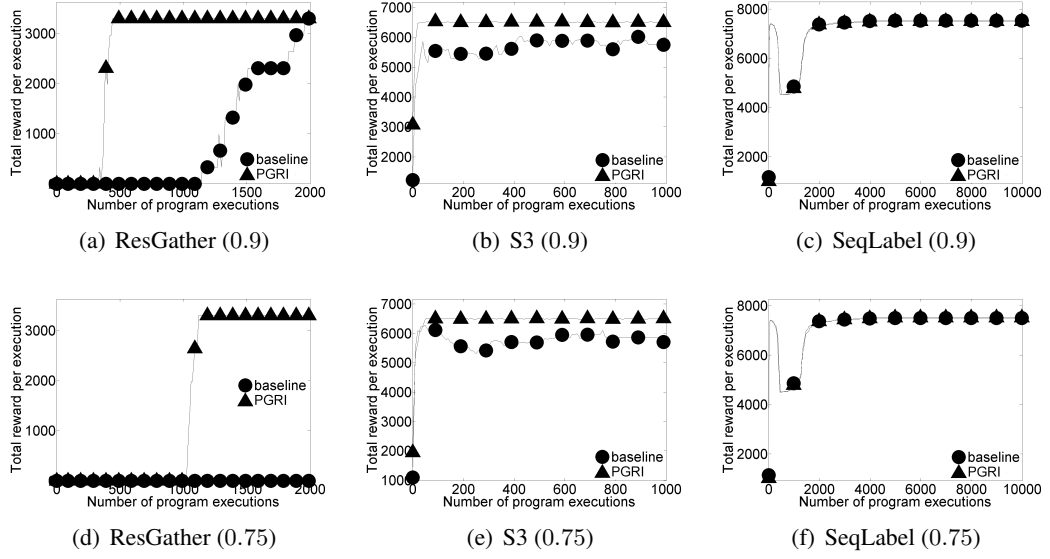


Figure 3.4: Effect of discounting. The figure shows the performance graphs of representative adaptive programs for two moderately high values of β and a subset of adaptive programs. The remaining programs are either invariant to moderately high values of β (S1, S2) or qualitatively similar to the ones shown (NetTalk \approx SeqLabel). For SeqLabel, both methods perform identically and the graphs overlap.

	Before	After	Test-Set
baseline	16.2	15.1	14.0
PGRI	17.4	55.1	54.2

(a) Phoneme ($\beta = 1$)

	Before	After	Test-Set
baseline	18.7	45.0	44.6
PGRI	17.1	49.0	47.4

(b) Phoneme ($\beta = 0.75$)

	Before	After	Test-Set
baseline	48.2	33.2	32.5
PGRI	42.3	80.4	80.1

(c) Stress ($\beta = 1$)

	Before	After	Test-Set
baseline	49.6	81.0	80.9
PGRI	45.36	81.0	80.2

(d) Stress ($\beta = 0.75$)

Figure 3.5: Label prediction accuracy (as percentage) for phoneme and stress labels in the NetTalk datasets. Shown are accuracies before and after learning on the training set and the final evaluation on the test set.

lower variance across learning runs which is related to smaller variance in its estimates.

3.8 Related Work

The notion of leveraging structure in domain variables is not new. Much work has been done in the planning community towards planning algorithms for compact representations of large MDPs [15, 32]. However, these approaches are generally intractable in practice, since the underlying planning problems are computationally hard. There has been very limited work on incorporating information about variable and reward influences into RL algorithms. Recent work by [65] assumes partial knowledge of the transition function (rather than just influence information) and shows how to incorporate that information into policy-gradient RL yielding strong guarantees of reduced variance. The influence information that we use is a very different way of representing partial information about a model. An interesting future direction is to combine these different types of knowledge into one estimator.

The most closely related work is the TD-based iSARSA(λ) algorithm from chapter 2, which also used ABP as the motivating application. There, we incorporated influence information into the SARSA(λ) algorithm in a somewhat ad-hoc way. A direct comparison between PGRI and iSARSA is only possible on small test problems since iSARSA uses Q-tables without any function approximation. In these cases, both methods perform comparably. While it is possible that some modified version of iSARSA would be competitive on larger programs (e.g., via linear function approximation), iSARSA has no guarantees of convergence, unlike the policy-gradient formulation in PGRI. Further, we are able to provide such guarantees while making few assumptions about the architecture of the parameterized policy, which in the case of ABP, corresponds to arbitrary adaptive programs. In general, there is no clear understanding about the conditions under which TD methods such as SARSA might empirically outperform policy gradient methods. A common approach [40, 11] is to study improvements within each class independently, as we have done here.

3.9 Summary

This chapter introduced the idea of incorporating side information about influences among variables and rewards into policy-gradient estimation. We formalized the notion of influence and provided an algorithm for using the information. The algorithm was proven to be unbiased

and conditions were given for when the new estimator will have reduced variance. We demonstrated our new algorithm in the substantial application domain of adaptation-based programming, where influence information can be automatically computed via standard program analysis techniques. The results showed significant speedups in learning compared to standard policy gradient.

Chapter 4: Learning Partial Policies to Speedup MDP Tree Search

4.1 Introduction

Online sequential decision making involves selecting actions in order to optimize a long-term performance objective while meeting practical decision-time constraints. A common solution approach is to use model-free learning techniques such as reinforcement learning [63] or imitation learning [51, 56] in order to learn reactive policies, which can be used to quickly map any state to an action. While reactive policies support fast decision making, for many complex problems, it can be difficult to represent and learn high-quality reactive policies. For example, in games such as Chess, Go, and other complex problems, the best learned reactive policies are significantly outperformed by even modest amounts of look-ahead search [26, 69]. As another example, recent work has shown that effective reactive policies can be learned for playing a variety of Atari video games using both RL [43] and imitation learning [29]. However, the performance of these reactive policies are easily outperformed or equaled by a vanilla lookahead planning agent [29]. The trade-off, of course, is that the planning agents requires significantly more time per decisions, which may not be practical for certain applications, including real-time Atari game play.

This paper attempts to make progress in the region between the above extremes of fast reactive decision making and slow lookahead search. In particular, we study algorithms for learning to improve the anytime performance of lookahead search, allow for more effective deliberative decisions to be made within practical time bounds. Lookahead tree search constructs a finite-horizon lookahead tree using a possibly stochastic world model or simulator in order to estimate action values at the current state. A variety of algorithms are available for building such trees, including instances of Monte-Carlo Tree Search (MCTS) such as UCT [37], Sparse Sampling [34], and FSSS [71], along with model-based search approaches such as RTDP [8] and AO* [14]. However, under practical time constraints (e.g., one second per root decision), the performance of these approaches strongly depends on the action branching factor. In non-trivial MDPs, the number of possible actions at each state is often considerable which greatly limits the feasible search depth. An obvious way to address this problem is to provide prior knowledge for explicitly pruning bad actions from consideration. In this paper, we consider the offline learning

of such prior knowledge in the form of partial policies ¹.

A partial policy is a function that quickly returns an action subset for each state and can be integrated into search by pruning away actions not included in the subsets. Thus, a partial policy can significantly speedup search if it returns small action subsets, provided that the overhead for applying the partial policy is small enough. If those subsets typically include high-quality actions, then we might expect little decrease in decision-making quality. Although learning partial policies to guide tree search is a natural idea, it has received surprisingly little attention, both in theory and practice. In this paper we formalize this learning problem from the perspective of “speedup learning”. We are provided with a distribution over search problems in the form of a root state distribution and a search depth bound. The goal is to learn partial policies that significantly speedup depth D search, while bounding the expected regret of selecting actions using the pruned search versus the original search method that does not prune.

In order to solve this learning problem, there are at least two key choices that must be made: 1) Selecting a training distribution over states arising in lookahead trees, and 2) Selecting a loss function that the partial policy is trained to minimize with respect to the chosen distribution. The key contribution of our work is to consider a family of reduction-style algorithms that answer these questions in different ways. In particular, we consider three algorithms that reduce partial policy learning to i.i.d. supervised learning problems characterized by choices for (1) and (2). Our main results bound the sub-optimality of tree search using the learned partial policies in terms of the expected loss of the supervised learning problems. Interestingly, these results for learning partial policies mirror similar reduction-style results for learning (complete) policies via imitation, e.g. [53, 64, 54].

We empirically evaluate our algorithms in the context of learning partial policies to speedup MCTS in two challenging domains with large action branching factors: 1) a real-time strategy game, Galcon and 2) a classic dice game, Yahtzee. The results show that using the learned partial policies to guide MCTS leads to significantly improved anytime performance in both domains. Furthermore, we show that several other existing approaches for injecting knowledge into MCTS are not as effective as using partial policies for action pruning and can often hurt search performance rather than help.

¹A version of this work appeared in UAI 2014 [48].

4.2 Problem Setup

We consider sequential decision making in the framework of Markov Decision Processes (MDPs). A MDP is a tuple (S, A, P, R) , where S is a finite set of states, A is a finite set of actions, $P(s'|s, a)$ is the transition probability of arriving at state s' after executing action a in state s , and $R(s, a) \in [0, 1]$ is the reward function giving the reward of taking action a in state s . The typical goal in MDP planning and learning is to compute a policy for selecting an action in any state, such that following the policy (approximately) maximizes some measure of long-term expected reward. For example, two popular choices are maximizing the expected finite-horizon total reward or expected infinite horizon discounted reward.

In practice, regardless of the long-term reward measure, for large MDPs, the offline computation of high-quality policies over all environment states is impractical. In such cases, a popular action-selection approach is online tree search, where at each encountered environment state, a time-bounded search is carried out in order to estimate action values. Note that this approach requires the availability of either an MDP model or an MDP simulator in order to construct search trees. In this paper, we assume that a model or simulator is available and that online tree search has been chosen as the action selection mechanism. Next, we formally describe the paradigm of online tree search, introduce the notion of partial policies for pruning tree search, and then formulate the problem of offline learning of such partial policies.

4.2.1 Online Tree Search

We will focus on depth-bounded search assuming a search depth bound of D throughout which bounds the length of future action sequences to be considered. Given a state s , we denote by $T(s)$ the depth D expectimax tree rooted at s . $T(s)$ alternates between layers of state nodes and action nodes, labeled by states and actions respectively. The children of each state node are action nodes for each action in A . The children of an action node a with parent labeled by s are all states s' such that $P(s'|s, a) > 0$. [Figure 4.1\(a\)](#) shows an example of a depth two expectimax tree. The depth of a state node is the number of action nodes traversed from the root to reach it, noting that leaves of $T(s)$ will always be action nodes.

The optimal value of a state node s at depth d , denoted by $V_d^*(s)$, is equal to the maximum value of its child action nodes, which we denote by $Q_d^*(s, a)$ for child a . Formally,

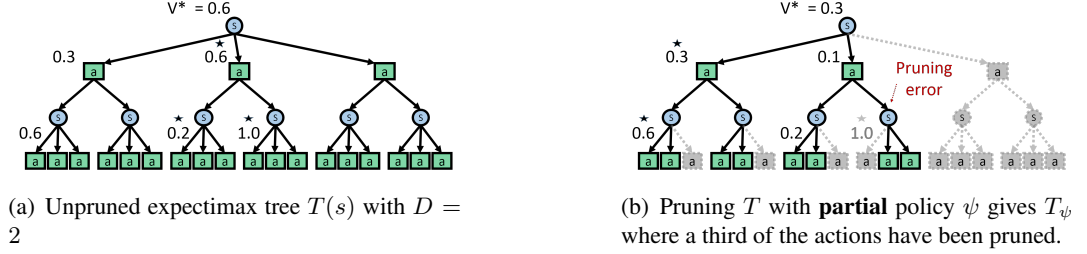


Figure 4.1: Unpruned and pruned expectimax trees with depth $D = 2$ for an MDP with $|A| = 3$ and two possible next states.

$$V_d^*(s) = \arg \max_a Q_d^*(s, a)$$

$$Q_d^*(s, a) = R(s, a) \quad \text{if } d = D - 1 \quad (4.1)$$

$$= R(s, a) + E_{s' \sim P(\cdot | s, a)} [V_{d+1}^*(s')] \quad \text{otherwise} \quad (4.2)$$

Equation 4.1 corresponds to leaf action nodes. In Equation 4.2 $s' \sim P(\cdot | s, a)$ ranges over the children of a . Given these value functions, the optimal action policy for state s at depth d is,

$$\pi_d^*(s) = \arg \max_a Q_d^*(s, a)$$

Given an environment state s , online search algorithms such as UCT or RTDP attempt to completely or partially search the tree $T(s)$ in order to approximate the root action values $Q_0^*(s, a)$ well enough to identify the optimal action $\pi_0^*(s)$. It is important to note that optimality in our context is with respect to the specified search depth D , which may be significantly smaller than the expected planning horizon in the actual environment. This is a practical necessity that is often referred to as receding-horizon control. Here we simply assume that an appropriate search depth D has been specified and our goal is to speedup planning within that depth.

4.2.2 Search with a Partial Policy

One way to speedup depth D search is to prune actions from $T(s)$. In particular, if a fixed fraction σ of actions are removed from each state node, then the size of the tree would decrease by a factor of $(1 - \sigma)^D$, potentially resulting in significant computational savings. In this paper,

we will utilize partial policies for pruning actions. We define a partial policy as follows: A depth D (non-stationary) partial policy ψ is a sequence $(\psi_0, \dots, \psi_{D-1})$ where each ψ_d maps a state to an action subset. Given a partial policy ψ and root state s , we can define a pruned tree $T_\psi(s)$ that is identical to $T(s)$, except that each state s at depth d only has subtrees for actions in $\psi_d(s)$, pruning away subtrees for any child $a \notin \psi_d(s)$. **Figure 4.1(b)** shows a pruned tree $T_\psi(s)$, where ψ prunes away one action at each state. It is straightforward to incorporate ψ into a search algorithm by only expanding actions at state nodes that are consistent with ψ .

We define the state and action values relative to $T_\psi(s)$ the same as before and let $V_d^\psi(s)$ and $Q_d^\psi(s, a)$ denote the depth d state and action value functions, as follows.

$$\begin{aligned} V_d^\psi(s, a) &= \arg \max_{a \in \psi_d(s)} Q_d^\psi(s, a) \\ Q_d^\psi(s, a) &= R(s, a) && \text{if } d = D - 1 \\ &= R(s, a) + E_{s' \sim P(\cdot | s, a)} [V_{d+1}^\psi(s')] && \text{otherwise} \end{aligned}$$

We will denote the highest-valued, or greedy, root action of $T_\psi(s)$ as,

$$G_\psi(s) = \arg \max_{a \in \psi_0(s)} Q_0^\psi(s, a)$$

This is the root action that a depth D search procedure would attempt to return in the context of ψ . We say that a partial policy ψ subsumes a partial policy ψ' if for each s and d , $\psi'_d(s) \subset \psi_d(s)$. In this case, it is straightforward to show that for any s and d , $V_d^{\psi'}(s) \leq V_d^\psi(s)$. This is simply because planning under ψ allows for strictly more action choices than planning under ψ' . Note that a special case of partial policies is when $|\psi_d(s)| = 1$ for all s and d , which means that ψ defines a traditional (complete) deterministic MDP policy. In this case, V_d^ψ and Q_d^ψ represent the traditional depth d state value and action value functions for policies. The second special case occurs at the other extreme where $\psi_d = A$ for all d . This corresponds to full-width search and we have $V^* = V^\psi$ and $Q^* = Q^\psi$.

Clearly, a partial policy can reduce the complexity of search. However, we are also concerned with the quality of decision making using $T_\psi(s)$ versus $T(s)$, which we will quantify in terms of expected regret. The regret of selecting action a at state s relative to $T(s)$ is equal to $V_0^*(s) - Q_0^*(s, a)$. Note that the regret of the optimal action $\pi_0^*(s)$ is zero. We prefer partial policies that result in root decisions with small regret over the root state distribution that we expect to encounter, while also supporting significant pruning. For this purpose, if μ_0 is a distribution over

root states, we define the expected regret of ψ with respect to μ_0 as,

$$\text{REG}(\mu_0, \psi) = E[V_0^*(s_0) - Q_0^*(s_0, G_\psi(s_0))], \quad \text{where } s_0 \sim \mu_0. \quad (4.3)$$

4.2.3 Learning Problem

We consider an offline learning setting where we are provided with a model or simulator of the MDP in order to train a partial policy that will be used later for online decision making. More specifically, the learning problem provides us with a distribution μ_0 over root states (or a representative sample from μ_0) and a depth bound D . The intention is for μ_0 to be representative of the states that will be encountered during online use. In this work, we are agnostic about how μ_0 is defined for an application. However, a typical example and one used in our experiments, is for μ_0 to correspond to the distribution of states encountered along trajectories of a receding horizon controller that makes decisions based on unpruned depth D search. This setup is closely related to imitation learning [64, 53, 54] since we are essentially treating unpruned search as the expert. Additionally, the initial state distribution μ_0 might also correspond to the state distributions that arise from more complex imitation learning algorithms such as DAGGER [54].

Given μ_0 and D , our “speedup learning” goal is to learn a partial policy ψ with small expected regret $\text{REG}(\mu_0, \psi)$, while providing significant pruning. That is, we want to approximately imitate the decisions of depth D unpruned search via a much less expensive depth D pruned search. In general, there will be a tradeoff between the potential speedup and expected regret. At one extreme, it is always possible to achieve zero expected regret by selecting a partial policy that does no pruning and hence no speedup. At the other extreme, we can remove the need for any search by learning a partial policy that always returns a single action (i.e., a complete policy). However, for many complex MDPs, it can be difficult to learn computationally efficient, or reactive, policies that achieve small regret. Rather, it may be much easier to learn partial policies that prune away many, but not all actions, yet still retain high-quality actions. While such partial policies lead to more search than a reactive policy, the regret can be much less.

In practice, we seek a good tradeoff between the two extremes, which will depend on the application. Instead of specifying a particular trade-off point as our learning objective, we develop learning algorithms in the next section that provide some ability to explore different points. In particular, the algorithms are associated with regret bounds in terms of supervised learning objectives that measurably vary with different amounts of pruning.

4.3 Learning Partial Policies

Given μ_0 and D , we now develop reduction-style algorithms for learning partial policies. The algorithms reduce partial policy learning to a sequence of D i.i.d. supervised learning problems, each producing one partial policy component ψ_d . The supervised learning problem for ψ_d will be characterized by a pair (μ_d, C_d) , where μ_d is a distribution over states, and C_d is a cost function that, for any state s and action subset $A' \subseteq A$ assigns a prediction cost $C_d(s, A')$. The cost function is intended to measure the quality of A' with respect to including actions that are high quality for s . Typically the cost function will be a monotonically decreasing function of $|A'|$ and $C_d(s, A) = 0$.

In this section, we assume the availability of an i.i.d. supervised learner **LEARN** that takes as input a set of states drawn from μ_d , along with C_d , and returns a partial policy component ψ_d that is intended to minimize the expected cost of ψ_d on μ_d , that is, minimize $E[C_d(s, \psi_d(s))]$ for $s \sim \mu_d$. Note that the expected cost that can be achieved by **LEARN** is related to the amount of pruning that it produces. In our experiments, the desired amount of pruning will be specified by the designer, which will be related to the time constraints of the decision problem. In practice, the specific learner will depend on the cost function and we describe our particular implementations in [section 4.4](#). Rather, in this section, we focus on defining reductions that allow us to bound the expected regret of ψ by the expected costs of the ψ_d returned by **LEARN**. The generic template for our reduction algorithms is shown in [Algorithm 3](#).

Algorithm 3 A template for learning a partial policy $\psi = (\psi_0, \dots, \psi_{D-1})$. The template is instantiated by specifying the pairs of distributions and cost functions (μ_d, C_d) for $d \in \{0, \dots, D-1\}$. **LEARN** is an i.i.d. supervised learning algorithm that aims to minimize the expected cost of each ψ_d relative to C_d and μ_d .

```

1: procedure PARTIALPOLICYLEARNER( $\{(\mu_d, C_d)\}$ )
2:   for  $d = 0, 1, \dots, D-1$  do
3:     Sample a training set of states  $S_d$  from  $\mu_d$ 
4:      $\psi_d \leftarrow \text{LEARN}(S_d, C_d)$ 
5:   end for
6:   return  $\psi = (\psi_0, \psi_1, \dots, \psi_{D-1})$ 
7: end procedure

```

In the following, our state distributions will be specified in terms of distributions induced by (complete) policies. In particular, given a policy π , we let $\mu_d(\pi)$ denote the state distribution induced by drawing an initial state from μ_0 and then executing π for d steps and returning the

OPI	$\mu_d = \mu_d(\pi^*)$ $C_d(s, \psi_d(s)) = 0 \text{ if } \pi_d^*(s) \in \psi_d(s), \text{ otherwise } 1$
FT-OPI	$\mu_d = \mu_d(\psi_{0:d-1}^+)$ $C_d(s, \psi_d(s)) = 0 \text{ if } \pi_d^*(s) \in \psi_d(s), \text{ otherwise } 1$
FT-QCM	$\mu_d = \mu_d(\psi^*)$ $C_d(s, \psi_d(s)) = Q(s, \pi_d^*(s)) - Q(s, \psi_d^*(s))$

Table 4.1: Instantiations for OPI, FT-OPI and FT-QCM in terms of the template in [Algorithm 3](#). Note that ψ_d is a partial policy while π_d^* , ψ_d^+ and ψ_d^* are complete policies.

final state. Since we have assumed an MDP model or simulator, it is straightforward to sample from $\mu_d(\pi)$ for any provided π . Before proceeding we state a simple proposition that will be used to prove our regret bounds.

Proposition 4 *If a complete policy π is subsumed by partial policy ψ , then for any initial state distribution μ_0 , $REG(\mu_0, \psi) \leq E[V_0^*(s_0)] - E[V_0^\pi(s_0)]$, for $s_0 \sim \mu_0$.*

Proof Since π is subsumed by ψ , we know that $Q_0^\psi(s, G_\psi(s)) = V_0^\psi(s) \geq V_0^\pi(s)$. Since for any a , $Q_0^*(s, a) \geq Q_0^\psi(s, a)$, we have for any state s , $Q_0^*(s, G_\psi(s)) \geq V_0^\pi(s)$. The result follows by negating each side of the inequality, followed by adding $V_0^*(s)$, and taking expectations. ■

Thus, we can bound the regret of a learned ψ if we can guarantee that it subsumes a policy whose expected value has bounded sub-optimality. Next we present three reductions for doing this, each having different requirements and assumptions, summarized in [Table 4.1](#).

4.3.1 OPI : Optimal Partial Imitation

Perhaps the most straightforward idea for learning a partial policy is to attempt to find a partial policy that is usually consistent with trajectories of the optimal policy π^* . That is, each ψ_d should be learned so as to maximize the probability of containing actions selected by π_d^* with respect

to the optimal state distribution $\mu_d(\pi^*)$. This approach is followed by our first algorithm called Optimal Partial Imitation (OPI). In particular, [Algorithm 3](#) is instantiated with $\mu_d = \mu_d(\pi^*)$ (noting that $\mu_0(\pi^*)$ is equal to μ_0 as specified by the learning problem) and C_d equal to zero-one cost. Here $C_d(s, A') = 0$ if $\pi_d^*(s) \in A'$ and $C_d(s, A') = 1$ otherwise. Note that the expected cost of ψ_d in this case is equal to the probability that ψ_d does not contain the optimal action. We refer to this as the pruning error and denote it by,

$$e_d^*(\psi) = \Pr_{s \sim \mu_d(\pi^*)} (\pi_d^*(s) \notin \psi_d(s)) \quad (4.4)$$

A naive implementation of OPI is straightforward. We can generate length D trajectories by drawing an initial state from μ_0 and then selecting actions (approximately) according to π_d^* using standard unpruned search. Defined like this, OPI has the nice property that it only requires the ability to reliably compute actions of π_d^* , rather than requiring that we also estimate action values accurately. This allows us to exploit the fact that search algorithms such as UCT often quickly identify optimal actions, or sets of near-optimal actions, well before the action values have converged.

Intuitively we should expect that if the expected cost e_d^* is small for all d , then the regret of ψ should be bounded, since the pruned search trees will generally contain optimal actions for state nodes. The following clarifies this dependence. For the proof, given a partial policy ψ , it is useful to define a corresponding complete policy ψ^+ such that $\psi_d^+(s) = \pi_d^*(s)$ whenever $\pi_d^*(s) \in \psi_d(s)$ and otherwise $\psi_d^+(s)$ is the lexicographically least action in $\psi_d(s)$. Note that ψ^+ is subsumed by ψ .

Theorem 5 *For any initial state distribution μ_0 and partial policy ψ , if for each $d \in \{0, \dots, D-1\}$, $e_d^*(\psi) \leq \epsilon$, then $REG(\mu_0, \psi) \leq \epsilon D^2$.*

Proof Given the assumption that $e_d^*(\psi) \leq \epsilon$ and that ψ^+ selects the optimal action whenever ψ contains it, we know that $e_d^*(\psi^+) \leq \epsilon$ for each $d \in \{0, \dots, D\}$. Given this constraint on ψ^+ we can apply Lemma 3² from [\[64\]](#) which implies $E[V_0^{\psi^+}(s_0)] \geq E[V_0^*(s_0)] - \epsilon D^2$, where $s_0 \sim \mu_0$. The result follows by combining this with Proposition [4](#). ■

This result mirrors work on reducing imitation learning to supervised classification [\[53, 64\]](#), showing the same dependence on the planning horizon. It is straightforward to construct an

²The main result of [\[64\]](#) holds for stochastic policies and requires a more complicated analysis that results in a looser bound. Lemma 3 is strong enough for deterministic policies.

example problem where the above regret bound is shown to be tight as is also the case for imitation learning. This result motivates a learning approach where we have `LEARN` attempt to return ψ_d that each maximizes pruning (returns small action sets) while maintaining a small expected cost.

4.3.2 FT-OPI : Forward Training OPI

OPI has a potential weakness, similar in nature to issues identified in prior work on imitation learning [53, 54]. In short, OPI does not train ψ to recover from its own pruning mistakes. Consider a node n in the optimal subtree of a tree $T(s_0)$ and suppose that the learned ψ erroneously prunes the optimal child action of n . This means that the optimal subtree under n will be pruned from $T_\psi(s)$, increasing the potential regret. Ideally, we would like the pruned search in $T_\psi(s)$ to recover from the error gracefully and return an answer based on the best remaining subtree under n . Unfortunately, the distribution used to train ψ by OPI was not necessarily representative of this alternate subtree under n , since it was not an optimal subtree of $T(s)$. Thus, no guarantees about the pruning accuracy of ψ can be made under n .

In imitation learning, this type of problem has been dealt with via “forward training” of non-stationary policies [54] and a similar idea works for us. The Forward Training OPI (FT-OPI) algorithm differs from OPI only in the state distributions used for training. The key idea is to learn the partial policy components ψ_d in sequential order from $d = 0$ to $d = D - 1$ and then training ψ_d on a distribution induced by $\psi_{0:d-1} = (\psi_0, \dots, \psi_{d-1})$, which will account for pruning errors made by $\psi_{0:d-1}$. Specifically, recall that for a partial policy ψ , we defined ψ^+ to be a complete policy that selects the optimal action if it is consistent with ψ and otherwise the lexicographically least action. The state distributions used to instantiate FT-OPI is $\mu_d = \mu_d(\psi_{0:d-1}^+)$ and the cost function remains zero-one cost as for OPI. We will denote the expected cost of ψ_d in this case by $e_d^+(\psi) = \Pr_{s \sim \mu_d(\psi_{0:d-1}^+)} (\pi_d^*(s) \notin \psi_d(s))$, which gives the probability of pruning the optimal action with respect to the state distribution of $\psi_{0:d-1}^+$.

Note that as for OPI, we only require the ability to compute π^* in order to sample from $\mu_d(\psi_{0:d-1}^+)$. In particular, note that when learning ψ_d , we have $\psi_{0:d-1}$ available. Hence, we can sample a state for μ_d by executing a trajectory of $\psi_{0:d-1}^+$. Actions for ψ_d^+ can be selected by first computing π_d^* and selecting it if it is in ψ_d and otherwise selecting the lexicographically least action.

As shown for the forward training algorithm for imitation learning [54], we give below an

improved regret bound for FT-OPI under an assumption on the maximum sub-optimality of any action. The intuition is that if it is possible to discover high-quality subtrees, even under sub-optimal action choices, then FT-OPI can learn on those trees and recover from errors.

Theorem 6 *Assume that for any state s , depth d , and action a , we have $V_d^*(s) - Q_d^*(s, a) \leq \Delta$. For any initial state distribution μ_0 and partial policy ψ , if for each $d \in \{0, \dots, D-1\}$, $e_d^+(\psi) \leq \epsilon$, then $REG(\mu_0, \psi) \leq \epsilon \Delta D$.*

Proof The theorem’s assumptions imply that ψ^+ and the search tree $T(s)$ satisfies the conditions of Theorem 2.2 of [54]. Thus we can infer that $E[V_0^{\pi^+}(s_0)] \geq E[V_0^*(s_0)] - \epsilon \Delta D$, where $s_0 \sim \mu_0$. The result follows by combining this with Proposition 4. ■

Thus, when Δ is significantly smaller than D , FT-OPI has the potential to outperform OPI given the same bound on zero-one cost. In the worst case $\Delta = D$ and the bound will equal to that of OPI.

4.3.3 FT-QCM: Forward Training Q-Cost Minimization

While FT-OPI addressed one potential problem with OPI, they are both based on zero-one cost, which raises other potential issues. The primary weakness of using zero-one cost is its inability to distinguish between highly sub-optimal and slightly sub-optimal pruning mistakes. It was for this reason that FT-OPI required the assumption that all action values had sub-optimality bounded by Δ . However, in many problems, including those in our experiments, that assumption is unrealistic, since there can be many highly sub-optimal actions (e.g. ones that result in losing a game). This motivates using a cost function that is sensitive to the sub-optimality of pruning decisions.

In addition, it can often be difficult to learn a ψ that achieves both, a small zero-one cost and also provides significant pruning. For example, in many domains, in some states there will often be many near-optimal actions that are difficult to distinguish from the slightly better optimal action. In such cases, achieving low zero-one cost may require producing large action sets. However, learning a ψ that provides significant pruning while reliably retaining at least one near-optimal action may be easily accomplished. This again motivates using a cost function that is sensitive to the sub-optimality of pruning decisions, which is accomplished via our third algorithm, Forward Training Q-Cost Minimization (FT-QCM)

The cost function of FT-QCM is the minimum sub-optimality, or Q-cost, over unpruned actions. In particular, we use $C_d(s, A') = V_d^*(s) - \max_{a \in A'} Q_d^*(s, a)$. Our state distribu-

tion will be defined similarly to that of FT-OPI, only we will use a different reference policy. Given a partial policy ψ , define a new complete policy $\psi^* = (\psi_0^*, \dots, \psi_{D-1}^*)$ where $\psi_d^*(s) = \arg \max_{a \in \psi_d(s)} Q_d^*(s, a)$, so that ψ^* always selects the best unpruned action. We define the state distributions for FT-QCM as the state distribution induced by ψ^* , i.e. $\mu_d = \mu_d(\psi_{0:d-1}^*)$. We will denote the expected Q-cost of ψ at depth d to be $\Delta_d(\psi) = E[V_d^*(s_d) - \max_{a \in \psi_d(s_d)} Q_d^*(s_d, a)]$, where $s_d \sim \mu_d(\psi_{0:d-1}^*)$.

Unlike OPI and FT-OPI, this algorithm requires the ability to estimate action values of sub-optimal actions in order to sample from μ_d . That is, sampling from μ_d requires generating trajectories of ψ_d^* , which means we must be able to accurately detect the action in $\psi_d(s)$ that has maximum value, even if it is a sub-optimal action. The additional overhead for doing this during training depends on the search algorithm being used. For many algorithms, near-optimal actions will tend to receive more attention than clearly sub-optimal actions. In those cases, as long as $\psi_d(s)$ includes reasonably good actions, there may be little additional regret. The following regret bound motivates the FT-QCM algorithm.

Theorem 7 *For any initial state distribution μ_0 and partial policy ψ , if for each $d \in \{0, \dots, D-1\}$, $\Delta_d(\psi) \leq \Delta$, then $REG(\mu_0, \psi) \leq 2D^{\frac{3}{2}}\sqrt{\Delta}$.*

Proof Consider any pair of non-negative real numbers (ϵ, δ) such that for any d , the probability is no more than ϵ of drawing a state from $\mu_d(\psi^*)$ with Q-cost (wrt ψ) greater than δ . That is, $\Pr(C_d(s_d, \psi_d(s_d)) \geq \delta) \leq \epsilon$. We will first bound $REG(\mu_0, \psi)$ in terms of ϵ and δ .

Let Π_δ be the set of policies for which all selected actions have regret bounded by δ . It can be shown by induction on the depth that for any $\pi \in \Pi_\delta$ and any state s , $V_0^\pi(s) \geq V_0^*(s) - \delta D$. For the chosen pair (ϵ, δ) we have that for a random trajectory t of ψ^* starting in a state drawn from μ_0 there is at most an ϵD probability that the Q-cost of ψ^* on any state of t is greater than δ . Thus, compared to a policy that always has Q-cost bounded by δ , the expected reduction in total reward of ψ for initial state distribution μ_0 is no more than ϵD^2 . This shows that,

$$\begin{aligned} E[V_0^{\psi^*}(s_0)] &\geq \min_{\pi \in \Pi_\delta} E[V_0^\pi(s_0)] - \epsilon D^2 \\ &\geq E[V_0^*(s_0)] - \delta D - \epsilon D^2 \end{aligned}$$

Since ψ^* is subsumed by ψ , Proposition 4 implies that $REG(\mu_0, \psi) \leq \delta D + \epsilon D^2$.

We now reformulate the above bound in terms of the bound on expected Q-cost Δ assumed by the theorem. Since Q-costs are non-negative, we can apply the Markov inequality to conclude

that $\Pr(C_d(s_d, \psi_d(s_d)) \geq \delta) \leq \frac{\Delta_d(\psi)}{\delta} \leq \frac{\Delta}{\delta}$. The pair $(\frac{\Delta}{\delta}, \delta)$ then satisfies the above condition for ψ^* . Thus, we get $\text{REG}(\mu_0, \psi) \leq \delta D + \frac{\Delta}{\delta} D^2$. The bound is minimized at $\delta = \sqrt{D\Delta}$, which yields the result. ■

FT-QCM tries to minimize this regret bound by minimizing $\Delta_d(\psi)$ via supervised learning at each step. Importantly, as we will show in our experiments, it is often possible to maintain small expected Q-cost with significant pruning, while the same amount of pruning would result in a much larger zero-one cost. It is an open problem as to whether this bound is tight in the worst case.

4.4 Implementation Details

This section first describes the UCT algorithm which is the base planner used in all the experiments. We then describe the partial policy representation and the learning algorithm. Finally, we specify how we generate the training data.

4.4.1 UCT

UCT [37] is an online planning algorithm, which, given the current state s , selects an actions by building a sparse lookahead tree over the state space reachable from s . Thus, s is at the root of the tree and edges correspond to actions and their outcomes so that the tree consists of alternating layers of state and action nodes. Finally, leaf nodes correspond to terminal states. Each state node in the tree stores Q estimates for each of the available actions which are used to select the next action to be executed.

This algorithm, described below, became famous for advancing the state-of-the-art in Computer Go [25, 26]. Since then, however, many additional successful applications have been reported, including but not limited to the IPCC planning competitions [35], general game playing [41], Klondike Solitaire [13], tactical battles in real-time strategy games [7], feature selection [24], etc. See [17] for a comprehensive survey.

UCT is unique in the way that it constructs the tree and estimates action values. Unlike standard minimax search or sparse sampling [34], which build depth-bounded trees and apply evaluation functions at the leaves, UCT neither imposes a depth bound nor does it require an evaluation function. Rather, UCT incrementally constructs a tree and updates action values by

carrying out a sequence of Monte-Carlo rollouts of entire trajectories from the root to the terminal state. The key idea in UCT is to bias the rollout trajectories towards promising ones, indicated by prior trajectories while continuing to explore. The outcome is that the most promising parts of the tree are grown first while still guaranteeing that an optimal decision will be made given sufficient time and memory.

There are two key algorithmic choices in UCT. First, the policy used to conduct each rollout. Second, the method for updating the tree in response to the outcome produced by the rollout. A popular technique requires storing at each node: a) the number of times the node has been visited in previous rollouts $n(s)$ and $n(s, a)$ for state and action nodes respectively, b) the current estimate of the action value $Q(s, a)$. Given these two pieces of information at each state node, UCT performs a rollout as follows. Each rollout begins at the root. If there exist actions that have yet to be tried, then a random choice is made over these untried actions. Otherwise, when all actions have been tried at least once, we select the action that maximizes an upper confidence bound given by,

$$Q_{\text{UCB}}(s, a) = Q(s, a) + c \sqrt{\frac{\log n(s)}{n(s, a)}} \quad (4.5)$$

where c is a constant that controls the exploration bonus. In practice, c is often tuned separately on a per-domain basis. The selected action is then executed and the resulting state is added to the tree if it is not already present. This action selection rule is based on the UCB rule [6] for multi-armed bandits which seeks to balance exploration and exploitation. The first term $Q(s, a)$ rewards actions with high Q value estimates (exploitation) while the second term is large for actions that have been relatively less visited compared to the parent state. Note that the exploration bonus goes to zero as the action is visited more often.

Having described a procedure to generate trajectories, we now describe the method used to update the tree. After the trajectory reaches a terminal state and receives a reward R , the action values and counts of each state along the trajectory are updated. In particular, for any state action pair (s, a) , the update rules are,

$$n(s) \leftarrow n(s) + 1 \quad (4.6)$$

$$n(s, a) \leftarrow n(s, a) + 1 \quad (4.7)$$

$$Q(s, a) \leftarrow Q(s, a) + \frac{1}{n(s, a)}(R - Q(s, a)) \quad (4.8)$$

The update is simply the average reward of rollout trajectories that pass through the node (s, a) . The algorithm is anytime, which means that a decision can be obtained by querying the root at any time. A common practice is to run the algorithm for a fixed number of trajectories and then select the root action which has the largest Q value. Thus, the algorithm described has two parameters: the exploration constant c and the number of rollout trajectories.

The above algorithm is the most basic form of UCT, which has received a very large amount of attention in the game playing and planning community. See [17] for a comprehensive survey of the MCTS family of algorithms. We defer the discussion of the MCTS family and relevant UCT variants to [section 4.5](#). For now, we proceed to describe the partial policy representation and learning framework used in our experiments.

4.4.2 Partial Policy Representation and Learning

Our partial policies operate by simply ranking the actions at a state and then pruning a percentage of the bottom actions. In this work, we focus on learning linear ranking functions, but non-linear approaches can also be used given an appropriate learner. We use partial policy components that have the form $\psi_d(s \mid w_d, \sigma_d)$, parameterized by an n -dimensional weight vector w_d and pruning fraction $\sigma_d \in [0, 1)$. Given a state s , each action a is ranked according to the linear ranking function $f_d(s, a) = w_d^T \phi(s, a)$, where $\phi(s, a)$ is a user provided n -dimensional feature vector that describes salient properties of state-action pairs. Using f_d we can define a total order over actions, breaking ties lexicographically. $\psi_d(s \mid w_d, \sigma_d)$ is then equal to the set of the $\lceil (1 - \sigma_d) \times |A| \rceil$ highest ranked actions. This representation is useful as it allows us to separate the training of f_d from the selection of the pruning fraction.

Next, we specify how we implement the LEARN procedure. Each training set will consist of pairs $\{(s_i, c_i)\}$ where s_i is a state and c_i is a vector that assigns a cost to each action. For OPI and FT-OPI the cost vector assigns 0 to the optimal action and 1 to all other actions. For FT-QCM the c_i give the Q-costs of each action. We learn the partial policy by first learning the

ranking function in a way that attempts to rank the optimal action as highly as possible and then select an appropriate pruning percentage based on the learned ranker.

For rank learning, we follow a common approach of converting the problem to cost-sensitive binary classification. In particular, for a given example (s, c) with optimal action a^* we create a cost-sensitive classification example for each action $a_j \neq a^*$ of the form,

$$(s, c) \longrightarrow \{(\phi(s, a^*) - \phi(s, a_j), c(a^*) - c(a_j)) \mid a_j \neq a^*\}$$

Learning a linear classifier for such an example will attempt to rank a^* above a_j according to the cost difference. We apply an existing cost-sensitive learner [38] to learn a weight vector based on the pairwise data. We also experimented with other methods of generating ranking constraints (e.g., all pairs, unpruned vs pruned). The simplest technique involving the optimal action performed best in this experimental setup.

Finally, after learning the ranking function for a particular ψ_d , we must select an appropriate pruning fraction σ_d . In practice, we do this by analyzing the expected cost of ψ_d for a range of pruning values and select a pruning value that yields reasonably small costs. In [section 4.6](#), we give details of the selections used in our experiments.

4.4.3 Generating Training States

Each of our algorithms requires sampling training states from trajectories of particular policies that either require approximately computing π_d^* (OPI and FT-OPI) or also action values for sub-optimal actions (FT-QCM). Our implementation of this is to first generate a set of trees using substantial search, which provides us with the required policy or action values and then to sample trajectories from those trees.

More specifically, our learning algorithm is provided with a set of root states S_0 sampled from the target root distribution μ_0 . For each $s_0 \in S_0$ we run UCT for a specified time bound, which is intended to be significantly longer than the eventual online time bound. Note that the resulting trees will typically have a large number of nodes on the tree fringe that have been explored very little and hence will not be useful for learning. Because of this we select a depth bound D for training such that nodes at that depth or less have been sufficiently explored and have meaningful action values. The trees are then pruned until depth D .

Given this set of trees we can then generate trajectories using the MDP simulator of the policies specified for each algorithm. [Figure 4.2](#) shows how the trajectories are generated by

Algorithm	Action cost vector	Sample trajectory
OPI	$[\dots, I_{a^*=a_i}, \dots]$	$s_0 \xrightarrow{\pi_0^*(s_0)} s_1 \xrightarrow{\pi_1^*(s_1)} s_2 \dots$
FT-OPI	$[\dots, I_{a^*=a_i}, \dots]$	$s_0 \xrightarrow{\psi_0^+(s_0)} s_1 \xrightarrow{\psi_1^+(s_1)} s_2 \dots$
FT-QCM	$[\dots, Q_d^*(s, a^*) - Q_d^*(s, a_i), \dots]$	$s_0 \xrightarrow{\psi_0^*(s_0)} s_1 \xrightarrow{\psi_1^*(s_1)} s_2 \dots$

Figure 4.2: Training data generation and ranking reduction for each algorithm. Start by sampling s_0 from the same initial state distribution μ_0 . OPI uses the optimal policy at each depth whereas FT-OPI and FT-QCM use the most recently learned complete policy (at depth $d - 1$). OPI and FT-OPI use 0-1 costs. Only FT-QCM uses the Q cost function which turns out to be critical for good performance.

each algorithm with respect to the algorithmic template given in [Algorithm 3](#). For example, OPI simply requires running trajectories through the trees based on selecting actions according to the optimal action estimates at each tree node. The state at depth d along each trajectory is added to the data set for training ψ_d . FT-QCM samples states for training ψ_d by generating length d trajectories of $\psi_{0:d-1}^*$.

Each such action selection requires referring to the estimated action values and returning the best one that is not pruned. The final state on the trajectory is then added to the training set for ψ_d . Note that since our approach assumes i.i.d. training sets for each ψ_d , we only sample a single trajectory from each tree. However, experimental evidence suggests that multiple trajectories can be safely sampled from the tree without significant performance loss (although the guarantee is lost). Doing so allows many additional examples to be generated per tree avoiding the need for additional expensive tree construction.

4.5 Related Work

While there is a large body of work on integrating learning and planning, to the best of our knowledge, we do not know of any work on learning partial policies for speeding up online MDP planning.

There are a number of efforts that study model-based reinforcement learning (RL) for large MDPs that utilize tree search methods for planning with the learned model, for example, RL using FSSS [71], Monte-Carlo AIXI [68], and TEXPLORE [31]. However, these methods focus on model/simulator learning and do not attempt to learn to speedup tree search using the learned models, which is the focus of our work.

A more related body of work is on learning search control knowledge in deterministic planning and games. One thread of work has been on learning knowledge for STRIPS-style deterministic planners, for example, learning heuristics and policies for guiding best-first search [74] or state ranking functions [73]. The problem of learning improved leaf evaluation heuristics has also been studied in the context of deterministic real-time heuristic search [18]. As another example, evaluation functions for game tree search have been learned from the “principle variations” of deep searches [69].

Perhaps the most closely related work is the recent work in [29] where deep UCT search is used to generate high-quality state-action training pairs. The high quality training data is used to train a stationary reactive policy using a deep neural network (DNN) policy. This is standard imitation learning where the goal is to learn a reactive policy. The resulting DNN policy is the state-of-the-art on a number of ATARI games. However, even this sophisticated, but fast, reactive policy is unable to compete with deep, but slower, UCT search. This work provides a clear example of how even sophisticated reactive controllers may not be competitive with sufficiently deep lookahead tree search in some domains. Our work attempts to provide a rigorous middle-ground between reactive decision-making and expensive unpruned search.

There have been a number of efforts for utilizing domain-specific knowledge in order to improve/speedup MCTS, many of which are covered in a recent survey [17]. A popular technique is to use a bias term $f(s, a)$ for guiding action selection during search. f is hand-provided in [19, 20] and learned in [26, 62]. Generally there are a number of parameters that dictate how strongly $f(s, a)$ influences search and how that influence decays as search progresses. In our experience, tuning the parameters for a particular problem can be tedious and difficult. Similar issues hold for the approach in [26] which attempts to learn an approximation of Q^* and then initializes search nodes with the estimate. In [62], control knowledge is learned via policy-gradient techniques in the form of a reward function and used to guide MCTS with the intention of better performance given a time budget. So far, however, the approach has not been analyzed formally and has not been demonstrated on large MDPs. Experiments in small MDPs have also not demonstrated improvement in terms of wall clock time over vanilla MCTS.

Finally, MCTS methods often utilize hand-coded or learned “default policies” (e.g., MoGo [25]) to improve anytime performance. While this has shown some promise in specific domains such as Go, where the policies can be highly engineered for efficiency, we have found that the computational overhead of using learned default policies is often too high a price to pay. In particular, most such learned policies require the evaluation of a feature vector at each state encoun-

tered, or for each state-action pair. This may cause orders of magnitude fewer trajectories to be executed compared to vanilla MCTS. In our experience, this can easily lead to degraded performance per unit time. Furthermore, there is little formal understanding about how to learn such rollout policies in principled ways, with straightforward approaches often yielding decreased performance [59].

4.6 Experiments

We perform a large-scale empirical evaluation of the partial policy learning algorithms on two challenging MDP problems. The first, Galcon, is a variant of a popular real-time strategy game while the second is Yahtzee, a classic dice game. Both problems pose a serious challenge to MCTS algorithms for entirely different reasons described below. However, in both cases, UCT, using our learned partial policies, is able to significantly improve real-time search performance over all the baselines. The key experimental findings are summarized next.

1. The search performance of UCT using a partial policy learned by the FT-QCM algorithm is significantly better than that of FT-OPI and OPI in both domains.
2. UCT using partial policies learned by FT-QCM performs significantly better than vanilla UCT and other “informed” UCT variants at small search budgets. At larger search budgets, it either wins (in Galcon) or achieves parity (in Yahtzee).
3. The average regret of the pruning policies on a supervised dataset of states is strongly correlated with the search performance, which is in line with our formal results. FT-QCM has significantly lower regret than FT-OPI and OPI.
4. The quality of the training data deteriorates quickly as depth increases. Regret increases with increasing depth.
5. It is critical to incorporate the computational cost of using control knowledge into the search budget during evaluation since not doing so can significantly change the perceived relative performance of different algorithms.

4.6.1 Experimental Domains

Galcon: The first domain is a variant of a popular two-player real-time strategy game, illustrated in [Figure 4.3](#). The agent seeks to maximize its population by launching variable-sized population fleets from one of the planets currently under its control. The intention may be to either reinforce one’s own planets or to attack an enemy planet or to capture a neutral (unclaimed) planet. This produces a very large action space ($O(|\text{planets}|^2)$) with hundreds of actions in maps containing 20 planets. State transitions are typically deterministic unless a battle between enemy populations occurs since the outcome of battles between similarly sized armies is random. The problems caused by the large action branching factor are exacerbated by the relatively slow simulator and the length of the game. In our setting, the problem horizon H is at most 90. Given these conditions, UCT can only perform a small number of rollouts even when the search budget is as large as eight seconds per move. In fact, UCT only sparsely expands the tree beyond depths two or three, even with larger budgets. Note however, that the rollout component of UCT is run to the end of the game and hence provides information well beyond those depths. Finally, the second player or adversary is fixed to be a UCT agent with a budget of one second per move. All evaluations are performed against this fixed adversary. The agent’s performance can be measured in terms of the final population difference, ($\text{score} = \text{agent.population} - \text{adversary.population}$). This value is zero when the game is tied, a large positive value when the agent wins convincingly and a negative value when the agent is much weaker than the adversary.

Yahtzee: The second domain is a classic dice game which consists of thirteen stages. In each stage, the agent may roll any subset of the dice at most twice, after which the player must assign the dice to one of the empty categories. A scored category may not be re-scored in later stages. This produces a total of $13 * 3 = 39$ decisions that must be made per game, where the last action trivially assigns the only remaining category. The categories are divided into upper and lower categories with each one scored differently. The upper categories roughly correspond to poker-style conditions (full house, three of a kind, straight, etc.) while the lower categories seek to maximize the individual dice faces (ones, twos, ..., sixes). A bonus is awarded if the sum of the lower category scores exceed a certain value for a maximum score of 375. This version is the simplest version of the game, restricted to a single “Yahtzee” without any jokers. Our implementation for the simulator will be made available upon request³. [Figure 4.4](#) shows a summary of the rules of the game.

³We are also in the process of open-sourcing the entire code base.

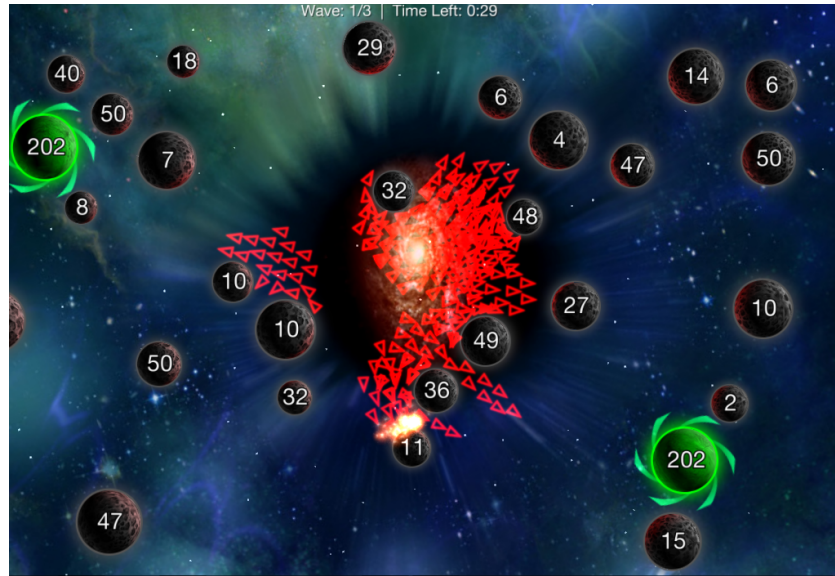








Figure 4.3: A visualization of a Galcon game in progress. The agent must direct variable-sized fleets to either reinforce its controlled planets or to take over enemy planets or unoccupied ones. The decision space is very large and consists of hundreds of actions.

The trees generated by UCT differ significantly in the two domains. Compared to Galcon, Yahtzee has a relatively small, but still significant, action branching factor, smaller time horizon and a fast simulator. Furthermore, the available actions and the branching factor in Yahtzee decrease as categories are filled, whereas in Galcon, the winning agent will have many additional actions to choose from since it will occupy more planets. Thus, we can play many more games of Yahtzee per unit time compared to Galcon. However, Yahtzee poses a severe challenge for MCTS algorithms due to the massive state branching caused by the dice rolls which makes it very hard to correctly evaluate actions. Consequently, even deep search with the UCT algorithm makes many obvious errors and struggles to improve performance. As in Galcon, the search quality of UCT degrades significantly at depths beyond two or three. Note however, that the performance degradation here is primarily caused by the transition stochasticity from the dice rolls although the action branching factor remains significant. Again, as in Galcon, UCT rolls out trajectories to the end of the game and provides information much deeper than these shallow depths.

UPPER SECTION:

MINIMUM REQUIRED FOR BONUS	HOW TO SCORE	GAME #1	GAME #2	GAME #3	GAME #4	GAME #5	GAME#6
Aces  = 3	COUNT AND ADD ONLY ACES						
Twos  = 6	COUNT AND ADD ONLY TWOS						
Threes  = 9	COUNT AND ADD ONLY THREES						
Fours  = 12	COUNT AND ADD ONLY FOURS						
Fives  = 15	COUNT AND ADD ONLY FIVES						
Sixes  = 18	COUNT AND ADD ONLY SIXES						
TOTAL = 63	➡						
Bonus IF 63 OR OVER	SCORE 35						
TOTAL OF UPPER HALF	➡						

LOWER SECTION:

3 of a kind	ADD TOTAL OF ALL DICE						
4 of a kind	ADD TOTAL OF ALL DICE						
Full house	SCORE 25						
Sm. Straight Sequence of 4	SCORE 30						
Lg. Straight Sequence of 5	SCORE 40						
YAHTZEE 5 OF A KIND	SCORE 50						
Chance	SCORE TOTAL OF ALL 5 DICE						
TOTAL OF LOWER HALF	➡						
TOTAL OF UPPER HALF	➡						
GRAND TOTAL	➡						

Figure 4.4: A typical scoring sheet for Yahtzee. In each game, all 13 categories must be scored, one per stage. Each stage consists of at most three dice rolls (original roll + two more controlled by the player). The objective is to maximize the total score, under significant stochasticity introduced by the dice rolls. Our simulator implements the variant with a single Yahtzee.

Normalizing rewards

The final score is always non-negative for Yahtzee but may be negative for Galcon if the agent loses the game. To achieve a consistent comparison between the supervised metrics and search performance, we perform a linear rescaling of the score so that the performance measure now lies in the range $[0, 1]$. In our experiments, we provide this measure as reward to the agent at the end of each trajectory. Note that a reward above 0.5 indicates a win in Galcon. In Yahtzee, a value of one is the theoretical upper limit achievable in the best possible outcome. This rarely occurs in practice. Furthermore, even a small increase in the reward (e.g., 0.1) is a significant improvement in Yahtzee.

4.6.2 Setup

We now describe the procedure used to generate training data for the three learning algorithms, OPI, FT-OPI and FT-QCM. Each algorithm is provided with root states generated by playing approximately 200 full games allowing UCT 600 seconds per move per tree for Galcon and 120 seconds per move per tree for Yahtzee. Each game results in a trajectory of states visited during the game. All of those states across all games constitute the set of root states used for training as described in [section 4.4](#). Due to the large action and state branching factors, despite an enormous search budget, depths beyond two or three are typically only sparsely expanded. We therefore select $D = 2$ (i.e., three pruning policies) for learning since the value estimates produced by UCT for deeper nodes are not accurate, as noted above. When UCT generates tree nodes at depths greater than D , we prune using ψ_D . As previously mentioned, we do not limit the rollouts of UCT to depth D . Rather, the rollouts proceed until terminal states, which provide a longer term heuristic evaluation for tree nodes. However, we have confirmed experimentally that typically such nodes are not visited frequently enough for pruning to have a significant impact.

We report the search budget in real-time (seconds), varying the budget up to eight seconds per move. Each point on the anytime curve shows the average score at the end of 1000 full games along with 95% confidence intervals. This large-scale evaluation has massive computational cost and requires access to a cluster. The use of time-based search budgets in a cluster environment adds variance, requiring additional trials. However, an important result of the following experiments is that the use of time to measure budgets is essential for a fair comparison of "knowledge-injected" search algorithms.

4.6.3 Comparing FT-QCM, FT-OPI and OPI

In order to compare OPI, FT-OPI and FT-QCM, we trained the algorithms using different sets of pruning thresholds σ_i and used them to prune actions during search. [Figure 4.5](#) shows the impact on search performance using UCT as the fixed base search algorithm. The first row shows the results for a large amount of pruning ($\sigma_i = 0.90$) while the second and third rows have progressively smaller pruning thresholds.

The first observation is that FT-QCM is clearly better than FT-OPI and OPI in both domains. FT-OPI and OPI are only competitive with FT-QCM in Galcon if the pruning is reduced significantly (bottom left). In Yahtzee, FT-QCM is significantly better even when the pruning is reduced to 50%.

Second, we observe that all search algorithms perform better as the search budget is increased. However, the pruning thresholds significantly impact the performance gain across the anytime curve. In Galcon, this is clearly seen by the steep curves for $\sigma_d = 0.5$ compared to the relatively flat curves for larger pruning thresholds. In Yahtzee, performance *appears* to increase very slowly. However, this small increase corresponds to significant improvement in playing strength.

The clear superiority of FT-QCM merits further investigation. Recall that FT-QCM differs from FT-OPI only in terms of the cost function. Also, forward training seems less relevant given the near-identical performance of FT-OPI and OPI. Clearly, incorporating the Q-cost into learning instead of zero-one is a dominating factor. Next, we analyze the learned partial policies themselves to gain additional insight into the impact of Q-cost learning compared to zero-one cost and forward training.

4.6.4 Relating search and regret

In order to better understand how search performance is related to the quality of the pruning policies, we analyze the learned partial policies on a held-out dataset of states. In particular, for each linear partial policy, we compute its average pruning error and average regret as the level of pruning is increased from none ($\sigma = 0$) to maximum ($\sigma = 1.0$). Recall the definition of pruning error from [Equation 4.4](#), which is the probability of ψ pruning away the optimal action (w.r.t. the expert policy). The definition for regret is given in [Equation 4.3](#). The average values for each of these is computed by averaging over a held-out dataset of states. [Figure 4.6](#) shows these supervised metrics on Galcon for each of the three depths starting with $D = 0$ in the first

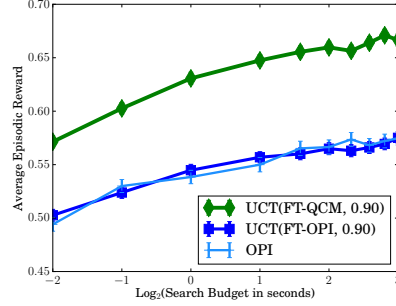
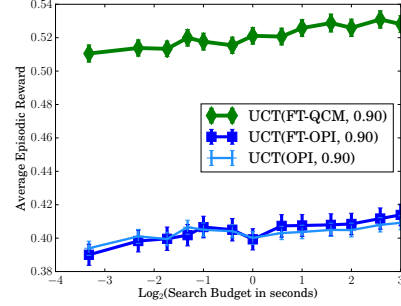
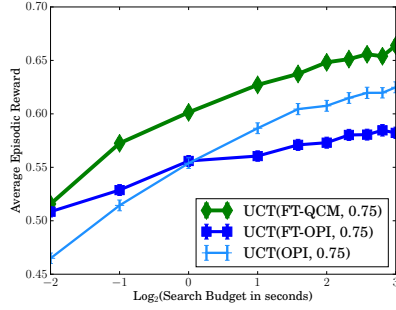
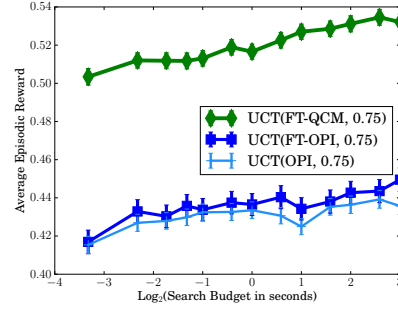
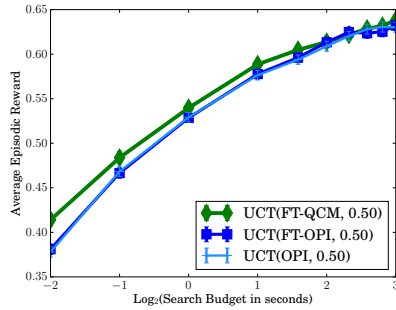
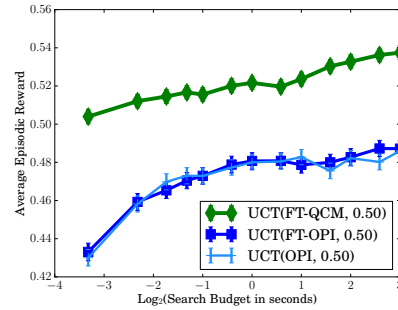
(a) Galcon, $\sigma_d = 0.90$ (b) Yahtzee, $\sigma_d = 0.90$ (c) Galcon, $\sigma_d = 0.75$ (d) Yahtzee, $\sigma_d = 0.75$ (e) Galcon, $\sigma_d = 0.50$ (f) Yahtzee, $\sigma_d = 0.50$

Figure 4.5: The search performance of UCT using partial policies learned by FT-QCM, FT-OPI and OPI as a function of the search budget, measured in seconds. The results for Galcon are in the first column and those for Yahtzee in the second. Each row corresponds to a different set of pruning thresholds σ_d with maximum pruning in the top row and least pruning in the third row. FT-QCM easily outperforms FT-OPI and OPI in all but one case (where they are tied). As mentioned in [section 4.6.1](#), the reward is a linear function of the raw score, measured at the end of the game.

row. The left column shows the average regret while the right column is the average pruning error rate. The result of pruning randomly is also shown since it corresponds to pruning with random action subsets.

We start at the root which is the first row in [Figure 4.6](#) for Galcon and [Figure 4.7](#) for Yahtzee. As expected, regret and error increase as more actions are pruned for all three algorithms. The key result here is that FT-QCM has significantly lower regret than FT-OPI and OPI for both domains. The lower regret is unsurprising given that FT-QCM uses the regret-based cost function while FT-OPI and OPI use 0/1 costs. However, FT-QCM also demonstrates smaller average pruning error in both problems than OPI and FT-OPI. Given that the same state distribution is used at the root, this observation merits additional analysis.

There are two main reasons why FT-OPI and OPI do not learn good policies. First, the linear policy space has very limited representational capacity. Second, the expert trajectories are noisy. The poor quality of the policy space is seen by the spike in the regret at maximum pruning, clearly indicating that all three learners have difficulty imitating the expert. We also observed this during training where the error remains high at the end for extreme levels of pruning. The expert’s shortcomings lie in the difficulty of the problems themselves. For instance, in Galcon, very few actions at lower depths receive more than a few visits. In Yahtzee, the stochasticity in dice rolls makes it very difficult to estimate actions accurately. Furthermore, in states where many actions are qualitatively similar (e.g., roll actions with nearly identical Q values), the expert’s choice is effectively random among those actions. This adds noise to the training examples which makes it hard to imitate accurately. However, FT-QCM’s use of regret-based costs allows it to work well in Yahtzee at all depths considered but only at the root in Galcon. On Galcon, the expert deteriorates rapidly away from the root policy and the training dataset becomes very noisy. Given the clear superiority of FT-QCM in every measurable way, for the remainder of the paper, we focus on FT-QCM.

4.6.5 Selecting σ_d

We will now describe a simple technique for selecting “good” values for σ_d . Recall that a good pruning threshold must “pay for itself”. That is, in order to justify the time spent evaluating knowledge, a sufficient number of sub-optimal actions must be pruned so that the improvement produced is larger than what would have been achieved by simply searching more. On one hand, too much pruning incurs large regret and little performance improvement with increasing search

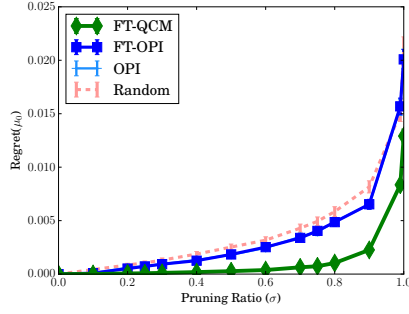
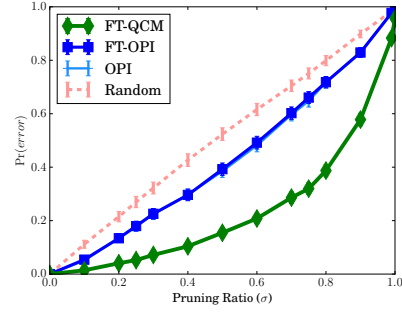
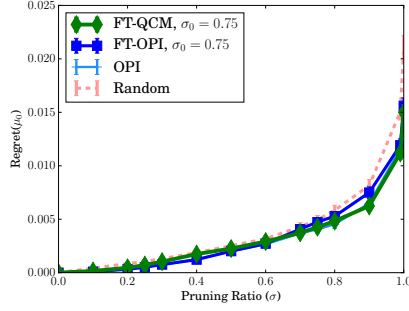
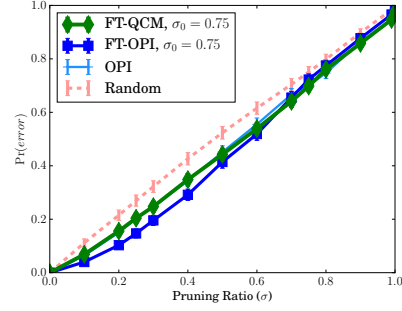
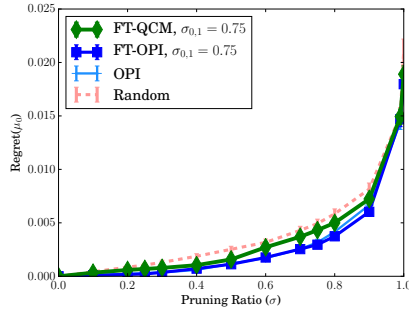
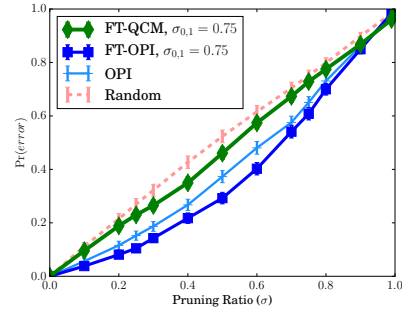
(a) Galcon: Regret at the root ($D=0$)(b) Galcon: Pruning error rate at the root ($D=0$)(c) Galcon: Regret at $D=1$ (d) Galcon: Pruning error rate at $D=1$ (e) Galcon: Regret at $D=2$ (f) Galcon: Pruning error rate at $D=2$

Figure 4.6: Average regret and pruning error for the learned partial policies ψ_d , as a function of the pruning ratio on the Galcon domain. The metrics are evaluated with respect to a held-out test set of states sampled from μ_0 . FT-QCM has the least error and regret at the root. At depths $D > 0$, performance degrades towards random.

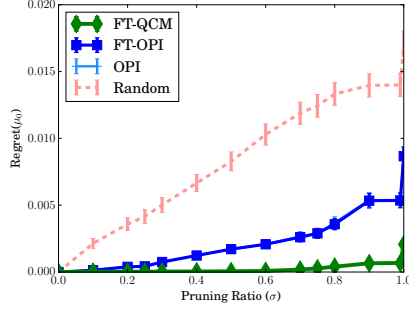
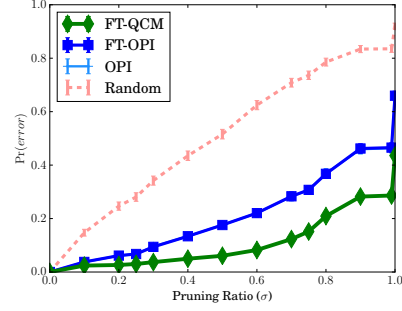
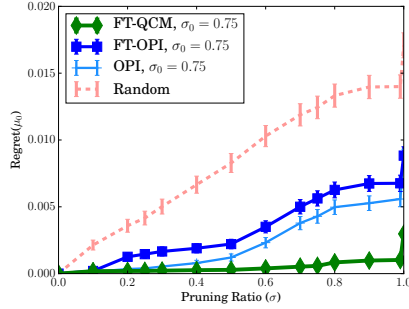
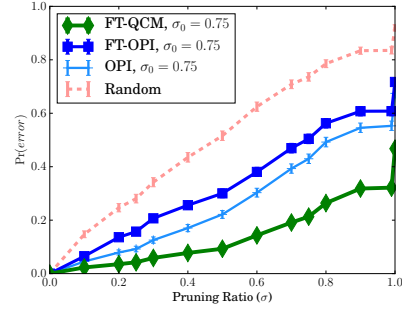
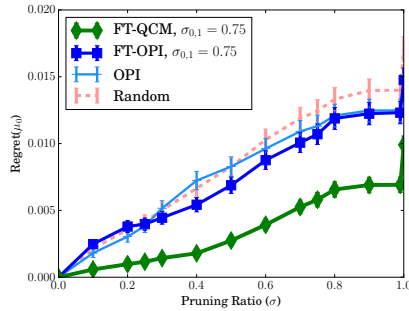
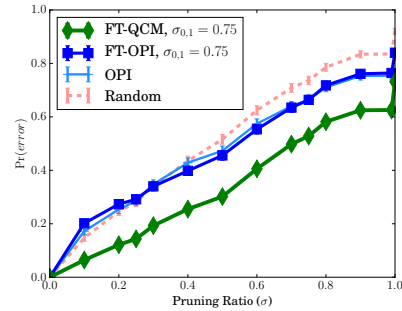
(a) Yahtzee: Regret at the root ($D=0$)(b) Yahtzee: Pruning error rate at the root ($D=0$)(c) Yahtzee: Regret at $D=1$ (d) Yahtzee: Pruning error rate at $D=1$ (e) Yahtzee: Regret at $D=2$ (f) Yahtzee: Pruning error rate at $D=2$

Figure 4.7: Average regret and pruning error for the learned partial policies ψ_d , as a function of the pruning ratio on the Yahtzee domain. The metrics are evaluated with respect to a held-out test set of states sampled from μ_0 . FT-QCM has the least error and regret at the root. At depths $D > 0$, FT-QCM continues to perform better than FT-OPI and OPI due to the relatively higher quality of training data at deeper states compared to Galcon.

budgets. On the other hand, if insufficient actions are pruned, search performance may be worse than uninformed search, due to the additional expense of computing state-action features.

The key to selecting good values lies in the supervised metrics discussed above where the regret and search graphs show a good correlation. Thus one simple technique for selecting σ_d is to simply use the largest value that has sufficiently small regret. For instance, in Galcon, we see that the regret is near-zero for $\sigma_0 < 0.75$ and increases sharply thereafter. The regret is practically zero for Yahtzee until the pruning threshold exceeds 0.5 and only seems to increase significantly around 0.75. For simplicity, we choose a constant pruning threshold value of $\sigma_d = 0.75$ at all depths in both domains. As we will see next, using FT-QCM partial policies with $\sigma_d = 0.75$ results in large performance gains across the anytime curve. A more computationally expensive way to select σ_d would be to evaluate a number of thresholds for actual planning performance and select the best.

4.6.6 Comparing FT-QCM to baselines

We will now compare FT-QCM against other search-based and reactive agents. Most of the baselines require control knowledge in the form of either a policy or a heuristic evaluation function. Since the partial policy learned by FT-QCM at the root has the least regret, we will use it in every technique⁴. We denote this partial policy as ψ_G and its linear scoring function as $h_G(s, a) = w_G^T \phi(s, a)$.

Heuristic Bias (HB): Our first baseline is a popular technique for injecting a heuristic into the MCTS framework. HB first adds a bias to an action node’s score and then slowly reduces it as the node is visited more often. This has the effect of favoring high-scoring actions early on and slowly switching to unbiased search for large budgets. This technique can be viewed as a form of progressive widening [19] with biased action selection and encompasses a large body of search variants. For example, one method uses the visit counts to explicitly control the branching factor at each node [20]. However, none of these methods come with formal regret bounds and are agnostic to the choice of heuristic, if one is used. From a practical standpoint, bias adjustment is highly domain-dependent and requires significant tuning. In our experiments, we observed that search performance is extremely sensitive to the interaction between the reward estimate, non-stationary exploration bonus and the non-stationary bias term. In our domains, the

⁴We also tried regression and RL techniques to learn an evaluation function but none of the approaches worked as well as FT-QCM.

following heuristic worked better than the other variations considered.

$$\begin{aligned} q(s, a) &= Kh_G(s, a) && \text{if } n_a = 0 \\ &= \text{UCB}(s, a) + (1 - k_n)Kh_G(s, a) && n_a > 0 \end{aligned}$$

where K is a domain-specific constant and $k_n = n(s, a)/(n(s, a) + 1)$ is zero for an unvisited action and increases towards one as the action is visited more. Thus, the exploration scaling term $(1 - k_n)$ is maximum (one) initially and decreases towards zero with each subsequent visit. Next, $\text{ucb}(s, a)$ is the UCB score of an action node. The value combines the Q estimate with an exploration bonus for infrequently visited actions. The best values of K and the UCB constant were obtained through grid search for each domain separately. This is computationally expensive, requiring many thousands of games to be played for each choice. Consequently, tuning HB is a significant task.

Informed Rollout (IR): Another popular technique for using knowledge in UCT involves using an “informed” or biased rollout policy. The underlying idea is that a biased rollout policy can produce more accurate state evaluations than uniform random action selection. Typically, this policy is hand-engineered using expert knowledge (e.g., MoGo [25]) but it can also be learned offline using domain knowledge [59] or online in a domain-independent manner [23]. Here, we use a MoGo-like strategy of selecting randomly from the partial policy ψ_G using a pruning ratio of 0.75. That is, at each state during a rollout from a leaf node, an action is randomly sampled from the top 25% of the action set with respect to the scoring function h_G . From the experiments in [section 4.6.4](#), we expect this policy to have very small regret. However, “informed” rollout policies have significantly larger computational expense compared to random action selection since each rollout requires knowledge to be evaluated a very large number of times. We experimented with other informed rollout policies including a complete policy that prunes all but one action, softmax policies, etc. However, none of these approaches were able to perform significantly better.

UCT with Random pruning (URP): A simple yet interesting baseline is the UCT algorithm which prunes randomly. We use 75% pruning for consistency. This search algorithm avoids the cost of feature computation while simultaneously reducing the number of actions under consideration. It is motivated by the insight obtained in the previous section, where performance at small search budgets improved with additional pruning. An important observation is that search-

ing with random subsets corresponds to evaluating the search performance of a random partial policy (i.e., red curve in [Figure 4.6](#) and [Figure 4.7](#)).

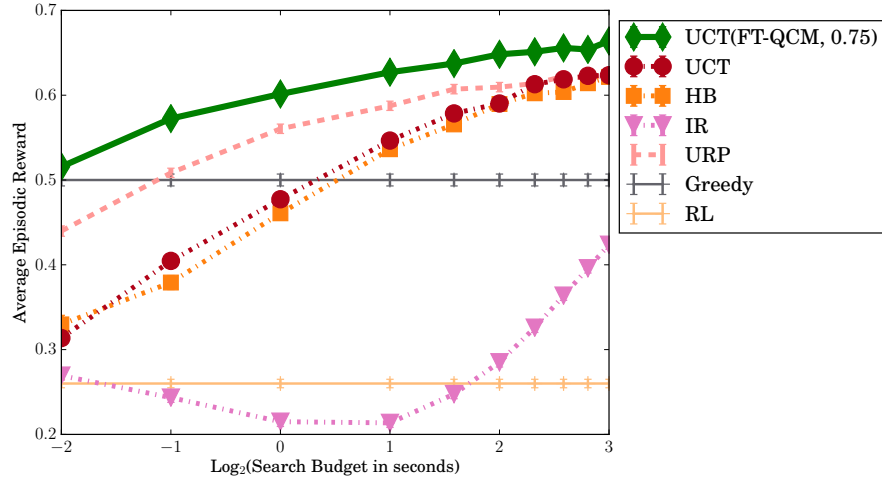
Reactive policies: Finally, we consider policies that do not search. The first, “Greedy”, is simply $\arg \max_a h_G(s, a)$, acting greedily w.r.t. ψ_G . The second is a linear policy learned using the SARSA(λ) RL algorithm [8] over the same features used to learn ψ_G . We experimented with different values of λ , exploration and learning rates and report the best settings for each domain.

Results

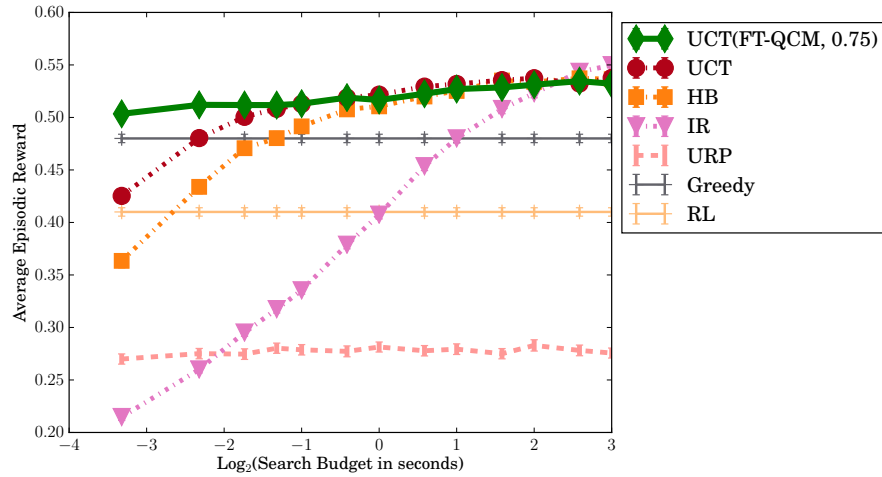
[Figure 4.8\(a\)](#) and [Figure 4.8\(b\)](#) show the results for Galcon and Yahtzee respectively. They constitute the main experimental result of this paper. We discuss the results for each domain separately below.

Galcon: The key observation in [Figure 4.8\(a\)](#) is that FT-QCM injected into UCT outperforms all other baselines at small to moderate search budgets, as measured in real time. At larger time scales, the FT-QCM injection continues to show slow improvement whereas uninformed UCT (red curve with circles) fails to match performance even at eight seconds per move. We were unable to get HB and IR to be competitive, which is surprising given prior success with these methods. IR performs particularly poorly and is worse than even the reactive policies (Greedy, SARSA). The main reason for IR’s poor performance is the massive computational overhead of its “informed” rollouts. In Galcon, games (and therefore rollouts) are long and computing features for every legal state-action pair is exorbitant. In contrast, random rollout policies are extremely fast and provide non-trivial value estimates. However, we observe that IR appears to be improving rapidly and may eventually outperform UCT but only at exorbitant search budgets (e.g., ten minutes per decision).

In contrast, heuristic bias (HB) is relatively inexpensive since it must only be computed once for each state added to the tree. That is, HB has the same overhead as partial policy injection which allows it to do better than IR. However, HB is unable to improve over uninformed UCT. Again, this is surprising since prior work has described successful injections with HB variants. In our experiments, we observe that HB is very sensitive to the particular choice of bias decay. Getting this injection to work well across the anytime curve is challenging and time-consuming. Next, we consider uninformed UCT which has zero overhead. As suspected, UCT is near-random at small time scales, getting outperformed by the variant which prunes randomly (URP). The difficulty of planning in large action spaces is highlighted by the strong performance of



(a) Galcon, FT-QCM v baselines



(b) Yahtzee, FT-QCM v baselines

Figure 4.8: The key empirical result of this paper shows that the search performance of UCT injected with FT-QCM significantly outperforms all other baselines at small budgets. At larger budgets, it either continues to win or achieves parity.

URP, which is second only to FT-QCM across the anytime curve. Finally, the reactive policies (Greedy, SARSA) perform reasonably well but are incapable of utilizing larger time budgets. Greedy, in particular, appears to perform as well as UCT with a search budget of one second per

move. This is much better than the policy learned by the RL algorithm. Finally, observe that Greedy outperforms all other baselines and is second only to FT-QCM.

Yahtzee: On this domain as well, FT-QCM easily outperforms all baselines. However, compared to Galcon, the baselines are far more competitive at larger time scales and a number of methods achieve parity with FT-QCM. Vanilla UCT is particularly good and quickly matches FT-QCM as the search budget is increased. However, the key result is that FT-QCM is able to significantly outperform UCT (and other baselines) for small decision times and does not lose to UCT at larger budgets, despite pruning 75% of the actions. This result also confirms that small supervised regret (Figure 4.6) translates into good performance as demonstrated by the strong performance of Greedy. However, as in Galcon, both reactive policies are outperformed by search-based techniques given sufficient time.

We observe that HB and IR fail to be competitive once again. IR performs very poorly at first for reasons discussed previously. However, IR eventually overtakes all methods and promises further improvements given additional time. On the other hand, HB is worse than UCT and only achieves parity at large time scales. The challenges of planning in Yahtzee are clearly visible in the relatively flat anytime curves for each search variant and the high residual variance despite averaging over more than one thousand games at each time budget. Given the relatively small action branching factor compared to Galcon, it is notable that FT-QCM is able to perform significantly better than all competing knowledge injections and reactive policies without requiring much tuning.

Finally the performance of URP is again illustrative in this domain. Unlike the case of Galcon, URP performs very poorly in Yahtzee. This is due to the fact that in Yahtzee, there are a relatively small percentage of actions that are reasonably good in most states. Rather, in Galcon, this percentage was higher. Thus, random pruning in Yahtzee, as done by URP, will often result in pruning away all reasonable actions.

4.6.7 The cost of knowledge

Given that many successful applications of biased initialization (e.g., HB) and biased rollout (e.g., IR) have exhibited improved search performance in the literature, we were surprised when HB and IR failed to improve over *vanilla* MCTS in our domains. Much tuning was required for HB to balance the effect of different score components. In contrast, it is relatively simple to bias rollouts and the results indicate that state-of-the-art performance is achievable if one is prepared

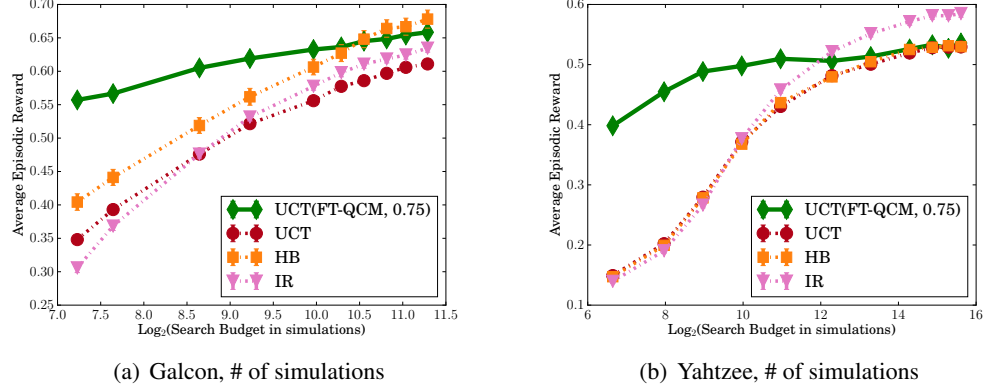


Figure 4.9: Setting the cost of knowledge to zero by using simulations instead of time. UCT with biased rollouts (IR) now appears to perform much better. However, UCT with FT-QCM continues to perform the best across significant portions of the anytime curve.

to wait for very long periods of time.

The key issue seems to be the high computational cost of knowledge compared to conducting additional search. In order to test whether the cost of knowledge is the key issue for IR and HB, we conducted further evaluations, where we measure performance in terms of the number of simulated trajectories rather than wall-clock time. This evaluation approach ignores the time-cost of applying either the evaluation heuristic or the biased rollout policy. This reporting method has the advantage of being neutral to the architecture and implementation and is widely used in the MCTS literature. Note that we cannot re-compute these results by simply translating the time-based evaluations since each rollout can take a variable amount of time to finish. For example, rollouts from states at the start of a game take more time compared to rollouts from near-terminal states. We therefore re-evaluate the UCT injections, where each search is now allowed to perform a fixed number of simulations from the root. The impact of measuring budgets using simulations instead of time are shown in [Figure 4.9\(a\)](#) and [Figure 4.9\(b\)](#).

FT-QCM injected search continues to win easily at small to moderate search budgets. However, HB and IR now appear to perform far better. HB outperforms UCT in Galcon and achieves parity in Yahtzee. IR shows huge improvement in both domains. We can summarize these experiments as follows: Using biased rollouts can improve search performance but only at exorbitant and unpredictable runtimes, varying with the particular state at the root of the tree. Given these

results, it seems very important to correctly incorporate the cost of knowledge when comparing different injections, which has not always been the case in the MCTS literature.

4.7 Summary and Future Work

We have shown algorithms for offline learning of partial policies for reducing the action branching factor in time-bounded tree search. The algorithms leverage a reduction to i.i.d. supervised learning and are shown to have bounds on the expected regret. Experiments in two challenging domains show significantly improved anytime performance in Monte-Carlo Tree Search.

There are a number of promising directions to explore next. Perhaps the most obvious extension is to replace the fixed-width forward pruning with a theoretically justified progressive widening (or selective unpruning) method. Doing so would allow the search to smoothly transition from reactive decision-making to full-width search at larger time scales. It would also eliminate the need to specify σ_d upfront. Ideally, the pruning level should depend on the remaining search budget. Such a time-aware search procedure has the potential to produce state-of-the-art anytime search performance. We are unaware of any method that leverages control knowledge for performing time-aware search on a per-decision basis in a principled manner.

Next, we would like to use the learned control knowledge to improve the expert policy, which was responsible for generating the deep trees used for training. The learned partial policy is only as good as the expert so any improvements to the expert may produce large improvements in search quality, particularly at small time scales. For instance, in Yahtzee, we observe that the expert makes serious mistakes in category selection, from which there can be no recovery. Thus, avoiding expert mistakes is key to further improvement. An iterative scheme that alternates between learning control knowledge and using it to improve the expert is likely to produce good results. However, FT-QCM can only obtain good value estimates for unpruned actions. Thus, it is important to analyze the regret bounds and performance from training only on a subset of actions.

Finally, we are interested in exploring principled learning algorithms for other types of control knowledge that can be shown to have guarantees similar to those in this paper. It can be argued that a huge part of the success of MCTS methods is their flexibility to incorporate different forms of knowledge, which are typically engineered for each application. Automating these processes in a principled way seems to be an extremely promising direction for improving planning systems.

Chapter 5: Lessons Learned and Future Work

In this thesis, we have proposed a number of techniques for leveraging behavior constraints, either specified or learned, in order to significantly improve the quality of decision-making. We have shown that these techniques are particularly applicable in self-optimizing programming frameworks like ABP. At the end of each chapter, we proposed future work in the context of the ideas presented there. Here, we take a much broader view of the work, discussing the lessons learned and outline a few long-term directions in which this work may be extended.

Tooling in ABP. We begin by focusing on the process of writing ABP programs. The current ABP framework, by design, provides only a minimal interface to the developer. In our informal interactions with ABP users, we observed that developers face significant uncertainty in all aspects of writing ABP programs: Identifying the correct choice points and the relevant context for each adaptive, identifying appropriate rewards, etc. This problem is exacerbated by the long time required to train each ABP program. In practice, this entails a complicated workflow where the impact of modifications to the source program are difficult to understand and predict.

This is a significant departure from modern development workflows, where the emphasis is on rapid product iterations. Note that these issues are not specific to ABP. Rather, they occur frequently during the development of many learning systems (e.g., search and recommendation engines). The design and implementation of learning programs is naturally iterative and even developers with experience in ML can experience significant task uncertainty. This has lead to custom tools. For example, Brainwash [3] is an IDE that attempts to simplify feature engineering, which is a critical task in most learning systems. Similar software tools are required in ABP for explaining and visualizing the adaptive program during all phases in the development of self-optimizing programs.

Behavioral Test Suites. We have argued that ABP systems need to provide more feedback to the developer. One simple way to do so is to allow the developer to specify a test suite, consisting of tuples, (c, a^*) specifying the correct action at each (adaptive, context) program location. Intuitively, this is related to unit testing where the developer specifies the expected output to a set of program inputs for small code units. Methods to specify such a test suite include

explicit enumeration (i.e., manual testing), which, although tedious, remains a very popular technique for testing software. More sophisticated techniques may include using a domain-specific language for declaratively specifying test configurations, constraints, invariants, etc. [28]. Having such a test suite allows the developer to verify learned behavior, study failing cases, and in general, obtain a better understanding of the program’s behavior.

Of course, such a test suite can be viewed as yet another set of expert demonstrations of program behavior. A key question here, is how to enforce the behavior constraints induced by the test suite on the learned policy? Perhaps more importantly, how can we do so when the test suite is very small, since the developer is unlikely to specify a large number of examples of target behavior. A similar problem has been studied in [36], an example of the large interest in systems that combine expert guidance and RL in a principled manner [33].

As previously mentioned, a critical requirement in modern development workflows is that developers receive feedback on their code quickly. This has lead to the development of programming paradigms that provide instantaneous visual feedback when the source program changes [70]. How to apply such techniques to self-optimizing programs that have significant computational resource requirements is currently unclear.

The Scale and Spectrum of Optimization. The above discussion was developer-centric. Now we switch our view to that of the learning algorithm. The flexibility of ABP admits a very large spectrum of optimization problems, from simple bandits at one end to problems with non-stationary transition dynamics and rewards at the other end. Note that the latter are outside the MDP framework but occur widely in practice. Furthermore, the computational environment can also vary drastically in terms of how much data can be generated and stored, the amount of time available before a decision must be returned, the cost of exploratory decisions, etc. The key point is that the learner is not given knowledge of any of these problem-specific parameters. The need to adjust representations and algorithmic settings is well understood by the experts who develop learning systems. However, in ABP, this tuning needs to occur automatically. The current method of using a fixed representation and learning algorithm is unlikely to be robust across a wide spectrum of tasks.

Model Learning. In chapter 4, we gave algorithms for “compiling” deep search into near-reactive decision-making via learned partial policies and simulation-based lookahead tree search. However, we assumed the simulator was provided. However, simulators are large, complex pieces of code and are costly to develop and maintain. In many problems, the simulator is simply unavailable. In such cases, model learning methods can be used to learn inexpensive,

approximate models online. However, the impact of simulator errors on the learned partial policies and the lookahead tree is unclear. A key issue in model learning is the tradeoffs between the representational capacity of the model representation and the scalability. Tabular methods do not scale to large problems while specifying good, state-action feature representations can be challenging and tedious for developers.

Learning Contexts. The representation issue becomes particularly severe when it comes to specifying the context. From the developer’s perspective, the relevant information needed to make a decision is already present in the full program state at each choice point. Thus, the requirement to design and specify a context may seem unnecessary to the developer. Being asked to do so in a particular form (e.g., as fixed-length, state-action features) is particularly confusing to developers, based on our informal interactions with end-users.

However, from the learner’s perspective, a good representation of state is critical to performance, since it abstracts away irrelevant details. The ability to learn a compact task-specific representation directly from the data is the holy grail in machine learning. Recent years have seen a resurgence of neural network (NN) architectures [57]. NN-based methods have been successfully used on increasingly complex “end-to-end” sequential tasks like playing ATARI games [43, 29] and robotic manipulation tasks [39]. The advantages of NN-based policies are that they require very little feature engineering besides the raw input encoded as fixed-length vectors and can represent very complex non-linear relationships.

However, there are serious practical difficulties that need to be resolved before we can use these architectures in ABP systems. Chief among these is the assumption of fixed-length input and output vectors. It is unclear how to convert arbitrary code objects (e.g., Python dictionaries) into fixed-length vector encodings in a principled manner. Standard techniques like “one-hot” encodings do not scale to large, discrete state spaces without additional information. Furthermore, training NN architectures typically entails significant tuning in terms of the network size, regularization, etc.

However, ABP systems that can leverage NN-representations for learning and planning would be intuitively appealing since they would form a natural bridge between the two extreme forms of behaviour programming: Hand-coded behaviors at one end and fully automated, black-box neural networks at the other.

5.1 Summary

In this thesis, we have proposed a number of techniques for incorporating and learning behavior constraints for sequential decision problems and experimentally demonstrated significant performance improvements. We have shown how many of these techniques are naturally applicable in the framework of adaptation-based programming. It represents a few small steps towards the goal of making self-optimizing programs ubiquitous.

Bibliography

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [2] Omar Alsaleh, Bechir Hamdaoui, and Alan Fern. Q-learning for opportunistic spectrum access. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, pages 220–224. ACM, 2010.
- [3] Michael Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher R, and Ce Zhang. Brainwash: A Data System for Feature Engineering. In *Conference on Innovative Data Systems Research*, 2013.
- [4] David Andre and Stuart J. Russell. Programmable Reinforcement Learning Agents. In *Advances in Neural Information Processing Systems*, pages 1019–1025, 2000.
- [5] David Andre and Stuart J. Russell. State Abstraction for Programmable Reinforcement Learning Agents. In *AAAI Conference on Artificial Intelligence*, pages 119–125, 2002.
- [6] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 27:235–256, 2002.
- [7] Radha-Krishna Balla and Alan Fern. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *International Joint Conference on Artificial Intelligence*, pages 40–45, 2009.
- [8] Andrew G Barto, Steven J Bradtke, and Satinder P Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.
- [9] Tim Bauer. Adaptation-based programming. *PhD Dissertation*, 2013. <http://hdl.handle.net/1957/37365>.
- [10] Tim Bauer, Martin Erwig, Alan Fern, and Jervis Pinto. Adaptation-Based Programming in Java. *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 81–90, 2011.
- [11] Jonathan Baxter and Peter L. Bartlett. Infinite-Horizon Policy-Gradient Estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

- [12] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.
- [13] R. Bjarnason, A. Fern, and P. Tadepalli. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In *International Conference on Automated Planning and Scheduling*, 2009.
- [14] Bonet Blai and Geffner Hector. Action Selection for MDPs: Anytime AO* Versus UCT. In *AAAI Conference on Artificial Intelligence*, 2012.
- [15] C. Boutilier, R. Dearden, and M. Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1-2):49–107, 2000.
- [16] Justin A Boyan and Michael L Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in Neural Information Processing Systems*, pages 671–671, 1994.
- [17] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.
- [18] Vadim Bulitko and Greg Lee. Learning in Real-Time Search: A Unifying Framework. *Journal of Artificial Intelligence Research*, 25:119–157, 2006.
- [19] Guillaume Chaslot, Mark Winands, Jaap H van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. In *Joint Conference on Information Sciences*, pages 655–661, 2007.
- [20] Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *Learning and Intelligent Optimization*, pages 433–445. 2011.
- [21] Thomas G. Dietterich. The MAXQ Method for Hierarchical Reinforcement Learning. In *International Conference on Machine Learning*, pages 118–126, 1998.
- [22] Janardhan Rao Doppa, Alan Fern, and Prasad Tadepalli. HC-Search: Learning heuristics and cost functions for structured prediction. In *AAAI Conference on Artificial Intelligence*, 2013.
- [23] Hilmar Finnsson and Yngvi Björnsson. Learning Simulation Control in General Game-Playing Agents. In *AAAI Conference on Artificial Intelligence*, pages 954–959, 2010.
- [24] Romaric Gaudel and Michele Sebag. Feature selection as a one-player game. In *International Conference on Machine Learning*, pages 359–366, 2010.

- [25] S. Gelly and Y. Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *Conference on Neural Information Processing Systems*, 2006.
- [26] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *International Conference on Machine Learning*, pages 273–280, 2007.
- [27] James Glenn. An Optimal Strategy for Yahtzee. Technical Report CS-TR-0002, Loyola College, 2006.
- [28] Alex Groce and Jervis Pinto. A Little Language for Testing. In *NASA Formal Methods*, volume 9058 of *Lecture Notes in Computer Science*, pages 204–218. 2015.
- [29] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. In *Advances in Neural Information Processing Systems*, 2014.
- [30] Guohua Hao and Alan Fern. Revisiting output coding for sequential supervised learning. In *International Joint Conference on Artificial Intelligence*, 2007.
- [31] Todd Hester and Peter Stone. TEXPLORE: real-time sample-efficient reinforcement learning for robots. *Machine Learning Journal*, 90(3):385–429, 2013.
- [32] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier. Spudd: Stochastic planning using decision diagrams. In *Conference on Uncertainty in Artificial Intelligence*, pages 279–288, 1999.
- [33] Kshitij Judah, Alan P Fern, Thomas G Dietterich, and Prasad Tadepalli. Active Imitation Learning: Formal and Practical Reductions to IID Learning. *Journal of Machine Learning Research*, 15:3925–3963, 2014.
- [34] Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Machine Learning*, 49(2–3):193–208, 2002.
- [35] Thomas Keller and Malte Helmert. Trial-based heuristic tree search for finite horizon mdps. In *International Conference on Automated Planning and Scheduling*, 2013.
- [36] Beomjoon Kim, Amir massoud Farahmand, Joelle Pineau, and Doina Precup. Learning from Limited Demonstrations. In *Advances in Neural Information Processing Systems*, pages 2859–2867. 2013.
- [37] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pages 282–293, 2006.
- [38] John Langford. Vowpal Wabbit, 2011. Repository located at https://github.com/JohnLangford/vowpal_wabbit/wiki. Accessed December 14, 2014.

- [39] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *CoRR*, abs/1504.00702, 2015.
- [40] J. Loch and S. Singh. Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *International Conference on Machine Learning*, 1998.
- [41] Jean Méhat and Tristan Cazenave. Combining UCT and nested Monte Carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, 2010.
- [42] Neville Mehta. Hierarchical structure discovery and transfer in sequential decision problems. *PhD Dissertation*, 2011. <http://hdl.handle.net/1957/25199>.
- [43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602, 2013.
- [44] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning*, pages 278–287, 1999.
- [45] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1980.
- [46] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, pages 1043–1049, 1997.
- [47] M.D. Pendrith and M.J. McGarity. An analysis of direct reinforcement learning in non-Markovian domains. In *International Conference on Machine Learning*, 1998.
- [48] Jervis Pinto and Alan Fern. Learning Partial Policies to Speedup MDP Tree Search. In *Conference on Uncertainty in Artificial Intelligence*, 2014.
- [49] Jervis Pinto, Alan Fern, Tim Bauer, and Martin Erwig. Robust learning for adaptive programs by leveraging program structure. In *International Conference on Machine Learning and Applications*, pages 943–948, 2010.
- [50] Jervis Pinto, Alan Fern, Tim Bauer, and Martin Erwig. Improving Policy Gradient Estimates with Influence Information. In *Asian Conference on Machine Learning*, pages 1–18, 2011.
- [51] D. Pomerleau. ALVINN: an autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems*, 1989.

- [52] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994.
- [53] Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *International Conference on Artificial Intelligence and Statistics*, pages 661–668, 2010.
- [54] Stéphane Ross, Geoffrey J Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *International Conference on Artificial Intelligence and Statistics*, pages 627–635, 2011.
- [55] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.
- [56] Claude Sammut, Scott Hurst, Dana Kedzier, and Donald Michie. Learning to fly. In *International Workshop on Machine Learning*, 1992.
- [57] Jürgen Schmidhuber. Deep Learning in Neural Networks: An Overview. *CoRR*, abs/1404.7828, 2014.
- [58] Terrence J Sejnowski and Charles R Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1(1):145–168, 1987.
- [59] David Silver and Gerald Tesauro. Monte-Carlo simulation balancing. In *International Conference on Machine Learning*, pages 945–952, 2009.
- [60] Christopher Simpkins, Sooraj Bhat, Charles Lee Isbell Jr., and Michael Mateas. Towards adaptive programming: integrating reinforcement learning into a programming language. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 603–614, 2008.
- [61] S.P. Singh, T. Jaakkola, and M.I. Jordan. Learning without state-estimation in partially observable Markovian decision processes. In *International Conference on Machine Learning*, 1994.
- [62] Jonathan Sorg, Satinder P Singh, and Richard L Lewis. Optimal Rewards versus Leaf-Evaluation Heuristics in Planning Agents. In *AAAI Conference on Artificial Intelligence*, 2011.
- [63] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.
- [64] Umar Syed and Robert E Schapire. A Reduction from Apprenticeship Learning to Classification. In *Advances in Neural Information Processing Systems*, pages 2253–2261, 2010.

- [65] Aviv Tamar, Dotan Di Castro, and Ron Meir. Integrating Partial Model Knowledge in Model Free RL Algorithms. In *International Conference on Machine Learning*, pages 305–312, 2011.
- [66] Brian Tanner and Adam White. RL-Glue : Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10:2133–2136, September 2009.
- [67] Alberto Uriarte and Santiago Ontañón. Kiting in RTS Games Using Influence Maps. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.
- [68] Joel Veness, Kee Siong Ng, Marcus Hutter, William Uther, and David Silver. A Monte-Carlo AIXI approximation. *Journal of Artificial Intelligence Research*, 40(1):95–142, 2011.
- [69] Joel Veness, David Silver, Alan Blair, and William W Cohen. Bootstrapping from game tree search. In *Advances in Neural Information Processing Systems*, pages 1937–1945, 2009.
- [70] Bret Victor. Learnable Programming : Designing a programming system for understanding programs. 2012.
- [71] Thomas J Walsh, Sergiu Goschin, and Michael L Littman. Integrating Sample-Based Planning and Model-Based Reinforcement Learning. In *AAAI Conference on Artificial Intelligence*, 2010.
- [72] Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8:229–256, May 1992.
- [73] Yuehua Xu, Alan Fern, and Sungwook Yoon. Learning linear ranking functions for beam search with application to planning. *The Journal of Machine Learning Research*, 10:1571–1610, December 2009.
- [74] Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *The Journal of Machine Learning Research*, 9:683–718, 2008.
- [75] Pingan Zhu, Jervis Pinto, Thanh Nguyen, and Alan Fern. Achieving Quality of Service with Adaptation-based Programming for medium access protocols. In *Global Communications Conference*, pages 1932–1937. IEEE, 2012.

