Robust Planning in Domains with Stochastic Outcomes, Adversaries, and Partial Observability

Hugh Brendan McMahan

CMU-CS-06-166

December 2006

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Avrim Blum, Co-Chair
Geoffrey Gordon, Co-Chair
Jeff Schneider
Andrew Ng, Stanford University

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under contract nos. F30602-01-C-0219 and HR0011-06-1-0023, and by the National Science Foundation (NSF) under grants CCR-0105488, NSF-ITR CCR-0122581, and NSF-ITR IIS-0312814. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the NSF, the U.S. government, or any other entity.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Robust Planning in Domains with Stochastic Outcomes, Adversaries, and Partial Observability	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Real-world planning problems often feature multiple sources of uncertainty including randomness in outcomes, the presence of adversarial agents and lack of complete knowledge of the world state. This thesis describes algorithms for four related formal models that can address multiple types of uncertainty Markov decision processes, MDPs with adversarial costs, extensiveform games, and a new class of games that includes both extensive-form games and MDPs as special cases. Markov decision processes can represent problems where actions have stochastic outcomes. We describe several new algorithms for MDPs, and then show how MDPs can be generalized to model the presence of an adversary that has some control over costs. Extensive-form games can model games with random events and partial observability. In the zero-sum perfect-recall case a minimax solution can be found in time polynomial in the size of the game tree. However, the game tree must ?remember? all past actions and random outcomes, and so the size of the game tree grows exponentially in the length of the game. This thesis introduces a new generalization of extensive-form games that relaxes this need to remember all past actions exactly, producing exponentially smaller representations for interesting problems. Further, this formulation unifies extensive-form games with MDP planning. We present a new class of fast anytime algorithms for the off-line computation of minimax equilibria in both traditional and generalized extensive-form games. Experimental results demonstrate their effectiveness on an adversarial MDP problem and on a large abstracted poker game. We also present a new algorithm for playing repeated extensive-form games that can be used when only the total payoff of the game is observed on each round.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**207**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Keywords: Planning, Game Theory, Markov Decision Processes, Extensive-form Games, Convex Games, Algorithms.

Abstract

Real-world planning problems often feature multiple sources of uncertainty, including randomness in outcomes, the presence of adversarial agents, and lack of complete knowledge of the world state. This thesis describes algorithms for four related formal models that can address multiple types of uncertainty: Markov decision processes, MDPs with adversarial costs, extensive-form games, and a new class of games that includes both extensive-form games and MDPs as special cases.

Markov decision processes can represent problems where actions have stochastic outcomes. We describe several new algorithms for MDPs, and then show how MDPs can be generalized to model the presence of an adversary that has some control over costs. Extensive-form games can model games with random events and partial observability. In the zero-sum perfect-recall case, a minimax solution can be found in time polynomial in the size of the game tree. However, the game tree must "remember" all past actions and random outcomes, and so the size of the game tree grows exponentially in the length of the game. This thesis introduces a new generalization of extensive-form games that relaxes this need to remember all past actions exactly, producing exponentially smaller representations for interesting problems. Further, this formulation unifies extensive-form games with MDP planning.

We present a new class of fast anytime algorithms for the off-line computation of minimax equilibria in both traditional and generalized extensive-form games. Experimental results demonstrate their effectiveness on an adversarial MDP problem and on a large abstracted poker game. We also present a new algorithm for playing repeated extensive-form games that can be used when only the total payoff of the game is observed on each round.

Acknowledgments

A great many people have helped me along the path to this thesis. My parents inspired my creativity and provided the education that started the journey. Laura Schueller deserves special credit for introducing me to the world of scary (I mean, higher) mathematics and teaching me how to think rigorously. Andrzej Proskurowski helped me see how to bridge the gap between mathematics and computer science.

The support of my advisors, Geoff Gordon and Avrim Blum, has been invaluable. They have been diligent guides through the research process and reliable sources of enthusiasm and fresh ideas. Many other people have helped shape and inspire this work; I am particularly grateful for the support of my other committee members, Andrew Ng and Jeff Schneider.

My time in Pittsburgh has been blessed with wonderful new friendships and the opportunity to build on old ones. You know who you are. Thank you! And last but not least, I'm especially grateful for the love and support of my wife, Amy. She knows how to get me to relax and how to help me stay focused, and she can always tell which of those I need. I couldn't ask for more.

To everyone who has aided my efforts, both those named here and those not, let me simply say: thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of this thesis is to design powerful modeling frameworks and general purpose, efficient algorithms for reasoning about uncertain environments. We draw upon techniques from the theory of Markov decision processes (MDPs) and partially observable MDPs (POMDPs), reinforcement learning, online learning (experts and bandit algorithms), and game theory in purist of this goal. This section introduces the major topics and results of the thesis, and broadly places the work in the context of other research. Detailed discussions of related work as well as most citations will be deferred to the relevant chapters. Figure (1.1) summarizes the principal problem models we will consider.

Our investigation into planning takes root in a fertile body of previous work, for planning problems have inspired a rich tradition in both AI and operations research. Simple problems like the shortest path problem gave way to the broad field of (deterministic) AI planning, which includes general-purpose search algorithms like A^* as well as algorithms for specialized but higher dimensional STRIPS-style problems [Russell and Norvig, 2003, Blum and Furst, 1997, Weld, 1999]. Researchers realized early on that purely deterministic planning would not be sufficient for many problems of real-word interest. Markov decision processes (MDPs) were one of the earliest formulations of planning problems with uncertainty. Books by Bellman [1957] and Howard [1960] brought greater exposure to the framework, and MDPs continue to make regular appearances in the AI and planning literature.

Markov decision processes MDPs can be used to represent a wide array of planning problems. Generally, they model uncertainty by describing the effect of a particular action as a distribution over possible outcomes, rather than assuming a single deterministic out-

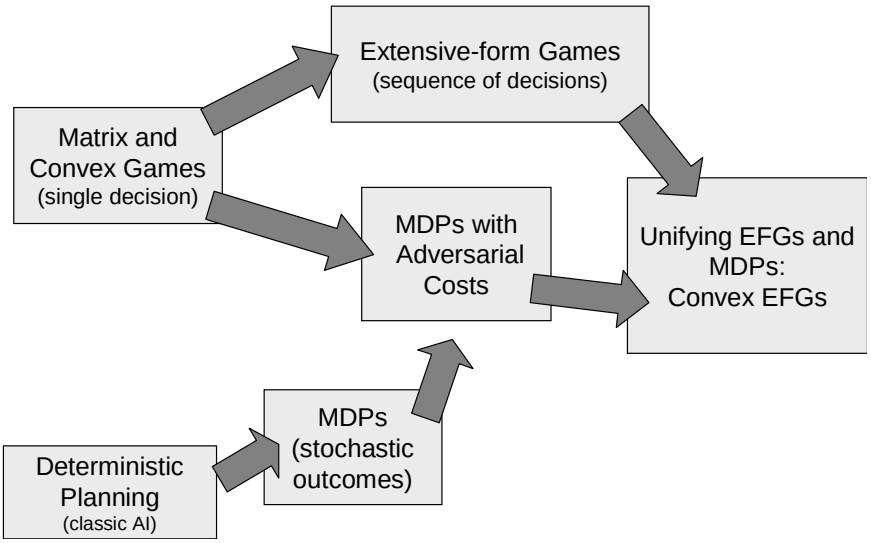


Figure 1.1: Relationships between problem models considered in this thesis. Arrows point to more general models; MDPs with adversarial costs and Convex EFGs are introduced in this thesis.

come. We refer to this type of uncertainty as *outcome uncertainty* (sometimes also called action uncertainty). In Chapter 2, we describe the MDP model more thoroughly, and discuss several new algorithms that offer advantages over previously known techniques. These algorithms are particularly effective for solving problems that have relatively few stochastic states and where only a small fraction of the state-space is relevant. Both of these are common characteristics of real-world problems.

Zero-sum game theory The MDP model assumes that the uncertainty in the world is stochastic, that is, that the uncertainty over outcomes is defined by some probability distribution, even if that distribution is not initially known. This approach is valid in many cases: stochastic models can effectively describe sensor noise, wheel-slippage on a mobile robot, the arrival of jobs for a computing cluster—in fact, the list could go on for pages, as modern research in AI is full of clever uses of stochastic modeling and probability theory. But, consider a game of chess: as we contemplate our move, we will have uncertainty about what response our opponent will make. Accurately modeling this response with a probability distribution would essentially require a complete model of our opponent's thought process, something that is not typically available! Instead, a more plausible approach is to attempt to select a move so that no matter what our opponent does we win (assuming such a move exists).

Game theory offers a variety of zero-sum models that capture this type of worst-case reasoning. The term zero-sum implies there are two players and that at the end of the game the payoff to one player is the negative of the payoff to the other player—that is, the payoffs sum to zero. This captures a purely adversarial model of interaction, since we can imagine the payoff as a value that one player has to pay to another player. Most common two-player games played by people are in fact zero-sum. However, it is worth emphasizing that the value of considering zero-sum games comes not from the ability to model parlor games. Rather, our interest is in using the tools of zero-sum game theory to reason about types of uncertainty where stochastic models are not available. In particular, we can reason about the worst-case over a set of potential eventualities, rather than reasoning in expectation about these possibilities given a fixed probability distribution.

This kind of strictly worse-case analysis can be overly pessimistic—in many situations there are extremely unlikely events[1] that may dramatically sway our plans if we assume an adversary is free to force one of these events to happen. One of the advantages of the models of uncertainty introduced in this thesis is that they provide a great deal of flexibility to interpolate between stochastic and adversarial models of uncertainty. As a first example of this approach, we consider MDPs where an opponent has some control over the costs of actions taken by the player planning in the MDP.

MDPs with adversary-controlled costs While MDPs can model outcome uncertainty, they cannot directly model domains with only partially observable state, unknown dynamics, or the presence of adversarial or cooperative agents. We describe several new algorithms for planning in more general environments without sacrificing the relative computational tractability of standard MDPs. These algorithms efficiently find plans that minimize the expected total worst-case cost of reaching a goal state when an adversary may choose any of a number of cost models. For example, we can find a solution to a stochastic shortest path problem that minimizes the maximum expected cost over a set of different edge-cost scenarios. A novel formulation as a linear program (LP) is sufficient to show that these problems can be solved in polynomial time. However, experiments demonstrate that directly solving the linear program is too slow for realistically-sized problems, even when using state-of-the-art commercial solvers. To address this, we present a transformation that allows the use of any MDP solver as a subroutine, producing over an order of magnitude speedup. We describe this model and its LP formulation in Chapter 3, and present our faster algorithms in Chapter 5.

[1] Even if we have no way of probabilistically quantifying "extremely unlikely."

Extensive-form games Partially observable stochastic games (POSGs) are the gold standard for modeling uncertain planning problems. They can directly handle most types of uncertainty: partial observability, noisy observations, random events, uncertain outcomes, and other agents. It is extremely difficult to come up with a planning problem that can not be modeled in some way as a POSG; unfortunately, it is equally challenging to find realistic problems where solving any POSG representation is computationally feasible.

For this reason, we do not consider fully general POSGs in this thesis work; however, we will be quite concerned with the closely related class of extensive-form games. EFGs can model all of the types of uncertainty that can be modeled by POSGs, with an important caveat: while POSGs can have a general state model where cycles are possible (states can be revisited), states in an EFG are always structured in a directed tree, and so cycles are not possible (no state is ever visited twice in the course of a game). Intuitively, a state in an EFG corresponds to a complete history of past actions and events in a POSG. Thus, the size of the game-tree for an EFG can be exponential in the size of a POSG representation of the same game.

Why constrain ourselves to a formulation that may entail an exponential penalty in representation size? Because EFGs can be solved in polynomial time in the size of the game tree (in the perfect-recall, zero-sum case—we fully introduce these restrictions in Chapter 3). Rather than attack the provably hard problem of solving general POSGs, we instead look to leverage the relative computational tractability of EFGs. We do this in two ways: we generalize the model in a way that allows many interesting games to be modeled exponentially more compactly than was previously possible, and we design fast algorithms that solve both standard EFGs and our generalization very quickly.

Convex extensive-form games Convex extensive-form games (CEFGs) generalize EFGs by replacing the usual small set of discrete actions with arbitrary one-shot two-player convex games—we fully describe the formulation in Chapter 4. Under some reasonable assumptions, CEFGs can be solved as a single convex optimization problem of polynomial-size in the representation of the game tree. This model can represent traditional extensive-form and matrix (normal form) games as well as MDPs and MDPs with adversary-controlled costs. (In fact, an MDP with adversary controlled costs can be modeled as a single node in the game tree of a CEFG, as can an arbitrary matrix game). The central advantage of the CEFG framework is that it can provide exponentially smaller representations of many interesting planning problems and sequential games. In addition to developing the theory necessary for efficient computation with CEFGs, we motivate the work by providing a high-level view of some interesting games that can be solved or approximated using CEFGs.

Game Class	Abbreviation	Reference
matrix (normal form) games	-	Section 3.1
convex games	CG	Section 3.1
stochastic (Markov) games	SG	Section 3.5
extensive-form games	EFG	Section 3.2
convex stochastic games	CSG	Section 3.5
convex extensive-form games	CEFG	Section 4.1

Table 1.1: Game models considered.

Convex games Matrix games, extensive-form games, convex extensive-form games, MDPs, and MDPs with adversary-controlled costs are all instances of the deceptively simple yet extremely expressive class of convex games. Though the concept of convex games is not at all new [see Dresher and Karlin, 1953], hopefully this thesis will help highlight theoretical and algorithmic properties of convex games that make the framework a powerful tool for modern computer science. We will fully consider the class of convex games in Chapter 3.

In addition to the examples listed above, we show that the the normal-form stage games in a stochastic game can be replaced with general convex games; these *convex stochastic games* (CSGs) can be solved in the discounted case by minimax value iteration. By using the convex game representation of an extensive-form game, we can embed EFGs as the stage games of a CSG, creating a class of tractable partially observable stochastic games. The key feature of this class is that the periods of partial observability are of bounded duration. Figure (1.1) summarizes the types of games considered in this thesis, along with the abbreviations used and the section where the class is first discussed.

Algorithms for convex games For convex games, it is typical to have fast best-response oracles that find an optimal response to a fixed opponent strategy. This is the case for EFGs, where a linear-time dynamic programming algorithm can calculate a best response; and for MDPs with adversary-controlled costs, where standard MDP algorithms can be used to calculate a best response.

In Chapter 5, we present new anytime algorithms for solving convex games that leverage such fast best-response oracles. Our principal approach is to use oracles for both players to build a model of the overall game that is used to identify search directions; the algorithm then does an exact minimization in this direction via a specialized line search.

We test our algorithms on both a simplified version of Texas Hold'em poker repre-

sented as an extensive-form game, and a sensor-placement / observation-avoidance game modeled as an MDP with adversary controlled costs. For the poker game, our algorithm approximated the exact value of this game within $\$0.30$ (the maximum pot size is $\$310.00$) in a little over 2 hours, using less than 1.5GB of memory; finding a solution with comparable bounds using a state-of-the-art interior-point linear programming algorithm took over 4 days and 25GB of memory. Our algorithms also demonstrate several orders of magnitude better performance on the adversarial MDP problem.

The online problem MDPs and extensive-form game models are useful when we have at least a partial model (either adversarial or stochastic) of the environment in which we wish to plan. However, such models are not easily available in many cases of interest. For problems where such a model is not available, no-regret algorithms can provide a reasonable framework for decision making.

Chapter 6 presents an algorithm for a general online (repeated) decision problem, where on each timestep a strategy from a convex set must be chosen without knowledge of the current cost (objective) function. Our algorithm guarantees performance almost as good as the best fixed solution in hindsight, while making no assumptions about the nature of the costs in the world and receiving information only about the outcomes of the decisions actually made, not potential alternatives. Previous results for this problem were limited to oblivious adversaries that make all decisions in advance; the algorithm we discuss was the first to also handle the case of an adaptive adversary.

This algorithm can be applied to a wide range of problems, for example, it can be used to play repeated convex games. Against a fully rational opponent, our no-regret algorithm will asymptotically perform as well as the minimax strategy, and against an arbitrary adversary the algorithm will do at least as well as the best fixed strategy in hindsight—which can be much better than the minimax value of the game if the opponent is not, in fact, fully adversarial.

Chapter 2

Algorithms for Planning in Markov Decision Processes

2.1 Introduction

Markov decision processes (MDPs) provide a framework for planning in domains where actions have uncertain outcomes. MDPs generalize deterministic planning problems like the shortest path problem that can be solved with Dijkstra's algorithm or A^* as well as the more structured deterministic problems tackled by AI planning [Cormen et al., 1990, Russell and Norvig, 2003, Weld, 1999, Blum and Furst, 1997]. After briefly introducing the MDP framework, we present two lines of research that lead to fast new algorithms for solving MDPs.

In the first, we study the problem of computing the optimal value function for a Markov decision process with positive costs. Computing this function quickly and accurately is a basic step in many schemes for deciding how to act in stochastic environments. There are efficient algorithms which compute value functions for special types of MDPs: for deterministic MDPs with S states and A actions, Dijkstra's algorithm runs in time $O(AS \log S)$. And, in single-action MDPs (Markov chains), standard linear-algebraic algorithms find the value function in time $O(S^3)$, or faster by taking advantage of sparsity or good conditioning. Algorithms for solving general MDPs can take much longer: we are not aware of any speed guarantees better than those for comparably-sized linear programs. We present a family of algorithms which reduce to Dijkstra's algorithm when applied to deterministic MDPs, and to standard techniques for solving linear equations when applied to Markov chains. More importantly, we demonstrate experimentally that these algorithms perform

well when applied to MDPs which "almost" have the required special structure. This work was originally presented in [McMahan and Gordon, 2005a,b].

MDPs for real-world problems often have intractably large state spaces. In the second line of work presented in this chapter, we consider solving such large problems when only a partial policy to get from a fixed start state to a goal is needed. In this situation, restricting computation to states relevant to this task can make much larger problems tractable. We introduce a new algorithm, Bounded Real-Time Dynamic Programming (BRTDP), which can produce partial policies with strong performance guarantees while only touching a fraction of the state space, even on problems where other algorithms would have to visit the full state space. To do this, Bounded RTDP maintains both upper and lower bounds on the optimal value function. The performance of Bounded RTDP is greatly aided by the introduction of a new technique to efficiently find suitable upper (pessimistic) bounds on the value function; this technique can also be used to provide informed initialization to a wide range of other planning algorithms. This is an extended treatment of the research originally described in [McMahan et al., 2005].

2.2 Markov Decision Processes

We briefly review the MDP formulation and introduce the notation we will use for the work presented in this chapter. The problem of finding an optimal policy in an MDP can be formulated with respect to several possible objective functions: expected total reward, expected discounted reward, and average reward [Puterman, 1994]. In this chapter, we restrict ourselves to the expected total reward criteria; we assume non-negative costs and the existence of an absorbing goal state to ensure a finite optimal value function. This formulation is sometimes called the stochastic shortest path problem.

We represent a stochastic shortest path problem with a fixed start state as a tuple $\mathcal{M} = (S, A, P, c, \mathsf{s}, \mathsf{g})$, where S is a finite set of states, $\mathsf{s} \in S$ is the start state, $\mathsf{g} \in S$ is the goal state, A is a finite action set, $c : S \times A \to \mathbb{R}_+$ is a cost function, and P gives the dynamics; we write P^a_{xy} for the probability of reaching state y when executing action a from state x. Since g is an absorbing goal state we have $c(\mathsf{g}, a) = 0$ and $P^a_{\mathsf{g},\mathsf{g}} = 1$ for all actions a. The set $\mathsf{succ}(x, a) = \{y \in S \mid P^a_{xy} > 0, y \neq \mathsf{g}\}$ contains all possible possible successors of state x under action a, except that the goal state is always excluded. Similarly, $\mathsf{pred}(x)$ is the set of all state-action pairs (y, b) such that taking action b from state y has a positive chance of reaching state x.

A stationary policy is a function $\pi : S \to A$. A policy is proper if an agent following it from any state will eventually reach the goal with probability 1. We make the standard

assumption that at least one proper policy exists for $\mathcal{M}$, and that all improper policies have infinite expected total cost at some state [Bertsekas and Tsitsiklis, 1996]. For a proper policy π, we define the value function of π as the solution to the set of linear equations:

$$v_\pi(x) = c(x, \pi(x)) + \sum_{y \in S} P_{xy}^{\pi(x)} v_\pi(y).$$

It is straightforward to verify that $v_\pi(x)$ is exactly the expected cost of reaching the goal by following π.

If $v \in \mathbb{R}^{|S|}$ is an arbitrary assignment of values to states, we define Q values with respect to v by

$$Q_v(x, a) = c(x, a) + \sum_{y \in S} P_{xy}^a v(y).$$

It is well-known that there exists an optimal (minimal) value function $v^\star$, and it satisfies *Bellman's equations* at all non-goal states x and for all actions a:

$$\begin{aligned} v^*(x) &= \min_{a \in A} Q^*(x, a) \\ Q^*(x, a) &= c(x, a) + \sum_{y \in \mathsf{succ}(x,a)} P_{xy}^a v^*(y). \end{aligned}$$

We can write these equations more compactly as $v^\star(x) = \min_{a \in A} Q_{v^\star}(x, a)$. For an arbitrary v, we define the (signed) *Bellman error* of v at x by $\mathsf{be}_v(x) = v(x) - \min_{a \in A} Q_v(x, a)$, the difference between the left-hand-side and right-hand-side of the Bellman equations. A greedy policy with respect to some value function v, $\mathsf{greedy}(v)$, is a policy that satisfies

$$\mathsf{greedy}(v, x) \in \operatorname*{argmin}_{a \in A} Q_v(x, a).$$

We say a policy π is optimal if $v_\pi(x) = v^*(x)$ for all x. It follows that a greedy policy with respect to v^* is optimal. In this way, the problem of finding an optimal policy reduces to that of finding the optimal value function.

To simplify notation, we have omitted the possibility of discounting. A discount factor γ can be introduced indirectly, however, by reducing P_{xy}^a by a factor of γ for all $y \neq \mathsf{g}$ and increasing $P_{x\mathsf{g}}^a$ accordingly, where g is the absorbing goal state.

For more details on the MDP formulation, consult Puterman [1994] or Bertsekas [1995]. We now turn to algorithms for the stochastic shortest path problem. Note that we do not assume the existence of a fixed start state s in Section 2.3, but this assumption will be central to Bounded Real-Time Dynamic Programming, introduced in Section 2.4.

2.3 Approaches based on Prioritization and Policy Evaluation

Many algorithms for planning in Markov decision processes work by maintaining estimates v and Q of v^* and Q^*, and repeatedly updating the estimates to reduce the difference between the two sides of the Bellman equations (called the *Bellman error*). For example, value iteration (VI) repeatedly loops through all states x performing *backup* operations at each one:

for all actions a **do**
  $Q(x,a) \leftarrow c(x,a) + \sum_{y \in \mathsf{succ}(x,a)} P^a_{xy} v(y)$
end for
$v(x) \leftarrow \min_{a \in A} Q(x,a)$

On the other hand, Dijkstra's algorithm carefully schedules[1] *expansion* operations at each state x instead:

$v(x) \leftarrow \min_{a \in A} Q(x,a)$
for all $(y,b) \in \mathsf{pred}(x)$ **do**
  $Q(y,b) \leftarrow c(y,b) + \sum_{x' \in \mathsf{succ}(y,b)} P^b_{yx'} v(x')$
end for

For good recent references on value iteration and Dijkstra's algorithm, see [Bertsekas, 1995] and [Cormen et al., 1990].

Any sequence of backups or expansions is guaranteed to make v and Q converge to the optimal v^* and Q^* so long as we visit each state infinitely often. Of course, some sequences will converge much more quickly than others. A wide variety of algorithms have attempted to find good state-visitation orders to ensure fast convergence. For example, Dijkstra's algorithm is guaranteed to find an optimal ordering for a deterministic positive-cost MDP; for general MDPs, algorithms like prioritized sweeping [Moore and Atkeson, 1993], generalized prioritized sweeping [Andre et al., 1998], RTDP [Barto et al., 1995], LRTDP [Bonet and Geffner, 2003b], Focussed Dynamic Programming [Ferguson and Stentz, 2004], and HDP [Bonet and Geffner, 2003a] all attempt to compute good orderings.

Algorithms based on backups or expansions have an important disadvantage, though: they can be slow at policy evaluation in MDPs with even a few stochastic transitions. For

[1] We deffer a full discussion of Dijkstra's algorithm to Section 2.3.1.

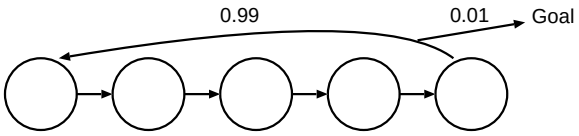


Figure 2.1: A Markov chain for which backup-based methods converge slowly. Each action costs 1.

example, in the Markov chain of Figure 2.1 (which has only one stochastic transition), the best possible ordering for value iteration will only reduce Bellman error by 1% with each five backups. To find the optimal value function quickly for this chain (or for an MDP which contains it), we turn instead to methods which solve systems of linear equations.

The policy iteration algorithm alternates between steps of *policy evaluation* and *policy improvement*. If we fix an arbitrary policy and temporarily ignore all off-policy actions, the Bellman equations become linear. We can solve this set of linear equations to evaluate our policy, and set v to be the resulting value function. Given v, we compute a greedy policy $\pi = \mathsf{greedy}(v)$. Fixing this greedy policy gives another set of linear equations, which can be solved to compute an improved policy. Policy iteration is guaranteed to converge so long as the initial policy has a finite value function. Within the policy evaluation step of policy iteration methods, we can choose any of several ways to solve our set of linear equations [Press et al., 1992]. For example, we can use Gaussian elimination, sparse Gaussian elimination, or biconjugate gradients with any of a variety of preconditioners. We can even use value iteration, although as mentioned above value iteration may be a slow way to solve the Bellman equations when we are evaluating a fixed policy.

Of the algorithms discussed above, no single one is fast at solving all types of Markov decision process. Backup-based and expansion-based methods work well when the MDP has short or nearly deterministic paths with little chance of cycling, but can converge slowly in the presence of noise and cycles. On the other hand, policy iteration evaluates each policy quickly, but may spend work evaluating a policy even after it has become obvious that another policy is better.

This section describes three new algorithms which blend features of Dijkstra's algorithm, value iteration, and policy iteration. In Section 2.3.1, we describe Improved Prioritized Sweeping. IPS reduces to Dijkstra's algorithm when given a deterministic MDP, but also works well on MDPs with stochastic outcomes. In Section 2.3.2, we develop Prioritized Policy Iteration, by extending IPS by incorporating policy evaluation steps. Section 2.3.3 describes Gauss-Dijkstra Elimination (GDE), which interleaves policy evaluation and prioritized scheduling more tightly. GDE reduces to Dijkstra's algorithm for

main():
queue.clear()
$(\forall x)$ closed$(x) \leftarrow$ false
$(\forall x)\ v(x) \leftarrow M$
$(\forall x, a)\ Q(x, a) \leftarrow M$
$(\forall a)\ Q(\text{goal}, a) \leftarrow 0$
closed(goal) $\leftarrow$ true
$(\forall x)\ \pi(x) \leftarrow$ undefined
π(goal) = arbitrary
update(goal)
while (not queue.isempty()) **do**
  $x \leftarrow$ queue.pop()
  closed$(x) \leftarrow$ true
  update(x)
end while

update(x)**:**
$v(x) \leftarrow Q(x, \pi(x))$
for all $(y, b) \in \mathsf{pred}(x)$ **do**
  $Q_{\text{old}} \leftarrow Q(y, \pi(y))$ (or M if $\pi(y)$ undefined)
  $Q(y, b) \leftarrow c(y, b) + \sum_{x' \in \mathsf{succ}(y,b)} P^b_{yx'} v(x')$
  if ((not closed(y)) and $Q(y, b) < Q_{\text{old}}$) **then**
    $\text{pri} \leftarrow Q(y, b)$ $\quad (*)$
    $\pi(y) \leftarrow b$
    queue.decreasepriority(y, pri)
  end if
end for

Figure 2.2: Dijkstra's algorithm, in a notation which will allow us to generalize it to stochastic MDPs. The variable "queue" is a priority queue which returns the smallest of its elements each time it is popped. The constant M is an upper bound on the value of (distance to) any state.

deterministic MDPs, and to Gaussian elimination for policy evaluation. In Section 2.3.5, we experimentally demonstrate that these algorithms extend the advantages of Dijkstra's algorithm to "mostly" deterministic MDPs, and that the policy evaluation performed by PPI and GDE speeds convergence on problems where backups alone would be slow.

2.3.1 Improved Prioritized Sweeping

Dijkstra's algorithm is shown in Figure 2.2. Its basic idea is to keep states on a priority queue, sorted by how urgent it is to expand them. The priority queue is assumed to support operations queue.pop(), which removes and returns the queue element with numerically lowest priority; queue.decreasepriority(x, p), which puts x on the queue if it wasn't there, or if it was there with priority $> p$ sets its priority to p, or if it was there with priority $< p$ does nothing; and queue.clear(), which empties the queue.

In deterministic Markov decision processes with positive costs, it is always possible to find a new state x to expand whose value we can set to $v^*(x)$ immediately. So, in

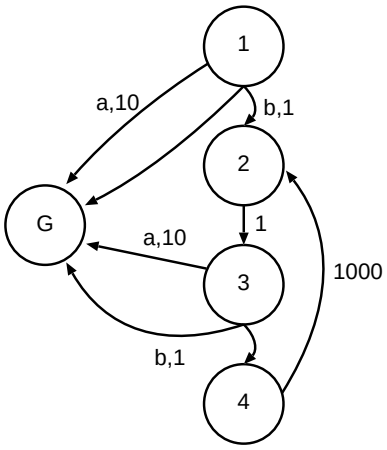


Figure 2.3: An MDP whose best state ordering is impossible to determine using only local properties of the states. Arcs which split correspond to actions with stochastic outcomes; for example, taking action b from state 1 reaches G with probability 0.5 and 2 with probability 0.5.

these MDPs, Dijkstra's algorithm touches each state only once while computing v^*, and is therefore by far the fastest way to find a complete policy.

An *optimal ordering* for backups or expansions is an ordering of the states such that for all states x, the value $v^*(x)$ can be determined using only $v^*(y)$ for states y which come before x in the ordering. In MDPs with stochastic outcomes, there need not exist an optimal ordering. Even if there exists such an ordering (*i.e.*, if there is an acyclic optimal policy), we might need to look at non-local properties of states to find it: Figure 2.3 shows an MDP with four non-goal states (numbered 1–4) and two actions (a and b). In this MDP, the optimal policy is acyclic with ordering $G3214$. But, after expanding the goal state, there is no way to tell which of states 1 and 3 to expand next: both have one deterministic action which reaches the goal, and one stochastic action that reaches the goal half the time and an unexplored state half the time. If we expand either one we will set its policy to action a and its value to 10; if we happen to choose state 3 we will be correct, but the optimal action from state 1 is b and $v^*(1) = 13/2 < 10$.

Several algorithms, most notably prioritized sweeping [Moore and Atkeson, 1993] and generalized prioritized sweeping [Andre et al., 1998], have attempted to extend the priority queue idea to MDPs with stochastic outcomes. These algorithms give up the property of visiting each state only once in exchange for solving a larger class of MDPs. However, neither of these algorithms reduce to Dijkstra's algorithm if the input MDP happens to be deterministic. Therefore, they potentially take far longer to solve a deterministic or nearly-

deterministic MDP than they need to. In the next section, we discuss what properties an expansion-scheduling algorithm needs to have to reduce to Dijkstra's algorithm on deterministic MDPs.

Generalizing Dijkstra

We will consider algorithms which replace the line $(*)$ in Figure 2.2 by other priority calculations that maintain the property that when the input MDP is deterministic with positive edge costs an optimal ordering is produced. If the input MDP is stochastic, a single pass of a generalized Dijkstra algorithm generally will not compute v^*, so we will have to run multiple passes. Each subsequent pass can start from the value function computed by the previous pass (instead of from $v(x) = M$ like the first pass), so multiple passes will cause v to converge to v^*. (Likewise, we can save Q values from pass to pass.) We now consider several priority calculations that satisfy the desired property.

Large Change in Value The simplest statistic which allows us to identify completely-determined states, and the one most similar in spirit to prioritized sweeping, is how much the state's value will change when we expand it. In line $(*)$ of Figure 2.2, suppose that we set

$$\text{pri} \leftarrow d(v(y) - Q(y, b)) \tag{2.1}$$

for some monotone decreasing function $d : \mathbb{R} \to \mathbb{R}$. Any state y with $\text{closed}(y) = \text{false}$ (called an open state) will have $v(y) = M$ in the first pass, while closed states will have lower values of $v(y)$. So, any deterministic action leading to a closed state will have lower $Q(y, b)$ than any action which might lead to an open state. And, any open state y which has a deterministic action b leading to a closed state will be on our queue with priority at most $d(v(y) - Q(y, b)) = d(M - Q(y, b))$. So, if our MDP contains only deterministic actions, the state at the head of the queue will the open state with the smallest $Q(y, b)$—identical to Dijkstra's algorithm.

Note that prioritized sweeping and generalized prioritized sweeping perform backups rather than expansions, and use a different estimates of how much a state's value will change when updated. Namely, they keep track of how much a state's successors' values have changed and base their priorities on these changes weighted by the corresponding transition probabilities. This approach, while in the spirit of Dijkstra's algorithm, does not reduce to Dijkstra's algorithm when applied to deterministic MDPs. Wiering [1999] discusses the priority function (2.1), but he does not prescribe the uniform pessimistic initialization of the value function which is given in Figure 2.2. This pessimistic initialization

is necessary to make (2.1) reduce to Dijkstra's algorithm. Other authors (for example Dietterich and Flann [1995]) have discussed pessimistic initialization for prioritized sweeping, but only in the context of the original non-Dijkstra priority scheme for that algorithm.

One problem with the priority scheme of equation (2.1) is that it only reduces to Dijkstra's algorithm if we uniformly initialize $v(x) \leftarrow M$ for all x. If instead we pass in some nonuniform $v(x) \geq v^*(x)$ (such as one which we computed in a previous pass of our algorithm, or one we got by evaluating a policy provided by a domain expert), we may not expand states in the correct order in a deterministic MDP.[2] This property is somewhat unfortunate: by providing stronger initial bounds, we may cause our algorithm to run longer. So, in the next few subsections we will investigate additional priority schemes which can help alleviate this problem.

Low Upper Bound on Value Another statistic which allows us to identify completely-determined states x in Dijkstra's algorithm is an upper bound on $v^*(x)$. If, in line $(*)$ of Figure 2.2, we set

$$\text{pri} \leftarrow m(Q(y, b)) \tag{2.2}$$

for some monotone increasing function $m(\cdot)$, then any open state y which has a deterministic action b leading to a closed state will be on our queue with priority at most $m(Q(y, b))$. (Note that $Q(y, b)$ is an upper bound on $v^*(y)$ because we have initialized $v(x) \leftarrow M$ for all x.) As before, in a deterministic MDP, the head of the queue will be the open state with smallest $Q(y, b)$. But, unlike before, this fact holds no matter how we initialize v (so long as $v(x) > v^*(x)$): in a deterministic positive-cost MDP, it is always safe to expand the open state with the lowest upper bound on its value.

High Probability of Reaching Goal Dijkstra's algorithm can also be viewed as building a set of closed states, whose v^* values are completely known, by starting from the goal state and expanding outward. According to this intuition, we should consider maintaining an estimate of how well-known the values of our states are, and adding the best-known states to our closed set first.

[2]We need to be careful passing in arbitrary $v(x)$ vectors for initialization: if there are any optimal but underconsistent states (states whose $v(x)$ is already equal to $v^*(x)$, but whose $v(x)$ is less than the right-hand side of the Bellman equation), then the check $Q(y, b) < v(y)$ will prevent us from pushing them on the queue even though their predecessors may be inconsistent. So, such an initialization for v may cause our algorithm to terminate prematurely before $v = v^*$ everywhere. Fortunately, if we initialize using a v computed from a previous pass of our algorithm, or set v to the value of some policy, then there will be no optimal but underconsistent states, so this problem will not arise.

For this purpose, we can add extra variables $p_{\text{goal}}(x, a)$ for all states x and actions a, initialized to 0 if x is a non-goal state and 1 if x is a goal state. Let us also add variables $p_{\text{goal}}(x)$ for all states x, again initialized to 0 if x is a non-goal state and 1 if x is a goal state.

To maintain the p_{goal} variables, each time we update $Q(y, b)$ we can set

$$p_{\text{goal}}(y, b) \leftarrow \sum_{x' \in \mathsf{succ}(y,b)} P^b_{yx'} p_{\text{goal}}(x')$$

And, when we assign $v(x) \leftarrow Q(x, a)$ we can set

$$p_{\text{goal}}(x) \leftarrow p_{\text{goal}}(x, a)$$

(in this case, we will call a the *selected action* from x). With these definitions, $p_{\text{goal}}(x)$ will always remain equal to the probability of reaching the goal from x by following selected actions and at each step moving from a state expanded later to one expanded earlier (we call such a path a *decreasing path*). In other words, $p_{\text{goal}}(x)$ tells us what fraction of our current estimate $v(x)$ is based on fully-examined paths which reach the goal.

In a deterministic MDP, p_{goal} will always be either 0 or 1: it will be 0 for open states, and 1 for closed states. Since Dijkstra's algorithm never expands a closed state, we can combine any decreasing function of $p_{\text{goal}}(x)$ with any of the above priority functions without losing our equivalence to Dijkstra. For example, we could use

$$\text{pri} \leftarrow m(Q(y, b), 1 - p_{\text{goal}}(y)) \tag{2.3}$$

where m is a two-argument monotone function.[3]

In the first sweep after we initialize $v(x) \leftarrow M$, priority scheme (2.3) is essentially equivalent to schemes (2.1) and (2.2): the value $Q(x, a)$ can be split up as

$$p_{\text{goal}}(x, a) Q_D(x, a) + (1 - p_{\text{goal}}(x, a))M$$

where $Q_D(x, a)$ is the expected cost to reach the goal assuming that we follow a decreasing path. That means that a fraction $1 - p_{\text{goal}}(x, a)$ of the value $Q(x, a)$ will be determined by the large constant M, so state-action pairs with higher $p_{\text{goal}}(x, a)$ values will almost always have lower $Q(x, a)$ values. However, if we have initialized $v(x)$ in some other way, then equation (2.1) no longer reduces to Dijkstra's algorithm, while equations (2.2) and (2.3) are different but both reduce to Dijkstra's algorithm on deterministic MDPs.

[3] A monotone function with multiple arguments is one which always increases when we increase one of the arguments while holding the others fixed.

This general technique can be thought of as tracking the probability of reaching the goal (versus reaching a history where no action is specified) under a particular non-stationary partial policy. In addition to providing a method for scheduling in our generalizations of Dijkstra's algorithm, we will use a similar approach to help schedule row-elimination operations in an application of Gaussian elimination to solving MDPs (Section (2.3.3)), as well as to produce high-quality upper bounds on the optimal value function in order to initialize the Bounded RTDP algorithm (Section (2.4.2)).

All of the Above Instead of restricting ourselves to just one of the priority functions mentioned above, we can combine all of them: since the best states to expand in a deterministic MDP will win on any one of the above criteria, we can use any monotone function of all of the criteria and still behave like Dijkstra in deterministic MDPs. For example, we can take the sum of two of the priority functions, or the product of two positive priority functions; or, we can use one of the priorities as the primary sort key and break ties according to a different one.

We have experimented with several different combinations of priority functions; the experimental results we report use the priority functions

$$\mathrm{pri}_1(x,a) = \frac{Q(x,a) - v(x)}{Q(x,a) + 1} \tag{2.4}$$

and

$$\mathrm{pri}_2(x,a) = \langle 1 - p_{\mathrm{goal}}(x),\ \mathrm{pri}_1(x,a) \rangle \tag{2.5}$$

The pri_1 function combines the value change criterion (2.1) with the upper bound criterion (2.2). It is always negative or zero, since $0 < Q(x,a) \leq v(x)$. It decreases when the value change increases (since $1/Q(x,a)$ is positive), and it increases as the upper bound increases (since $1/x$ is a monotone decreasing function when $x > 0$, and since $Q(x,a) - v(x) \leq 0$).

The pri_2 function uses p_{goal} as a primary sort key and breaks ties according to pri_1. That is, pri_2 returns a vector in $\mathbb{R}^2$ which should be compared according to lexical ordering (*e.g.*, $(3,3) < (4,2) < (4,3)$).

Sweeps vs. Multiple Updates

The algorithms we have described so far in this section must update every state once before updating any state twice. We can also consider a version of the algorithm which does not enforce this restriction; this multiple-update algorithm simply skips the check "if

not closed(y)" which ensures that we don't push a previously-closed state onto the priority queue. The multiple-update algorithm still reduces to Dijkstra's algorithm when applied to a deterministic MDP: any state which is already closed will fail the check $Q(y, b) < v(y)$ for all subsequent attempts to place it on the priority queue.

Experimentally, the multiple-update algorithm is faster than the algorithm which must sweep through every state once before revisiting any state. Intuitively, the sweeping algorithm can waste a lot of work at states far from the goal before it determines the optimal values of states near the goal.

In the multiple-update algorithm we are always effectively in our "first sweep," and so since we initialize uniformly to a large constant M we can reduce to Dijkstra's algorithm by using priority pri_1 from equation (2.4). The resulting algorithm is called Improved Prioritized Sweeping; its update method is listed in Figure 2.4.

As is typical for value-function based methods, we declare convergence when the maximum Bellman error (over all states) drops below some preset limit ϵ. This is implemented in IPS by an extra check that ensures all states on the priority queue have Bellman error at least ϵ; when the queue is empty it is easy to show that no such states remain. Similar methods are used for our other algorithms.

2.3.2 Prioritized Policy Iteration

The Improved Prioritized Sweeping algorithm works well on MDPs which are moderately close to being deterministic. Once we start to see large groups of states with strongly interdependent values, there will be no expansion order which will allow us to find a good approximation to v^* in a small number of visits to each state. The MDP of Figure 2.1 is an example of this problem: because there is a cycle which has high probability and visits a significant fraction of the states, the values of the states along the cycle depend strongly on each other.

To avoid having to expand states repeatedly to incorporate the effect of cycles, we will turn to algorithms that occasionally do some work to evaluate the current policy. When they do so, they will temporarily fix the current actions to make the value determination problem linear. The simplest such algorithm is policy iteration, which alternates between complete policy evaluation (which solves an $S \times S$ system of linear equations in an S-state MDP) and greedy policy improvement (which picks the action which achieves the minimum on the right-hand side of Bellman's equation at each state).

We will describe two algorithms which build on policy iteration. The first algorithm, called Prioritized Policy Iteration, is the subject of the current section. PPI attempts to

update(x)**:**
$v(x) \leftarrow Q(x, \pi(x))$
for all $(y, b) \in \mathsf{pred}(x)$ **do**
  $Q_{\text{old}} \leftarrow Q(y, \pi(y))$ (or M if $\pi(y)$ undefined)
  $Q(y, b) \leftarrow c(y, b) + \sum_{x' \in \mathsf{succ}(y,b)} P^b_{yx'} Q(x', \pi(x'))$
  if ($Q(y, b) < Q_{\text{old}}$) **then**
    $\text{pri} \leftarrow (Q(y, b) - v(y))/(Q(y, b) + 1)$
    $\pi(y) \leftarrow b$
    if ($|v(y) - Q(y, b)| > \epsilon$) **then**
      queue.decreasepriority(y, pri)
    end if
  end if
end for

Figure 2.4: The **update** function for the Improved Prioritized Sweeping algorithm. The **main** function is the same as for Dijkstra's algorithm. As before, "queue" is a priority min-queue and M is a very large positive number.

improve on policy iteration's greedy policy improvement step, doing a small amount of extra work during this step to try to reduce the number of policy evaluation steps. Since policy evaluation is usually much more expensive than policy improvement, any reduction in the number of evaluation steps will usually result in a better total planning time. The second algorithm, which we will describe in the Section 2.3.3, tries to interleave policy evaluation and policy improvement on a finer scale to provide more accurate Q and p_{goal} estimates for picking actions and calculating priorities on the fringe.

Pseudo-code for PPI is given in Figure 2.5. The main loop is identical to regular policy iteration, except for a call to `sweep()` rather than to a greedy policy improvement routine. The policy evaluation step can be implemented efficiently by a call to a sophisticated linear solver; such a solver can take advantage of sparsity in the transition dynamics by constructing an explicit LU factorization [Duff et al., 1986], or it can take advantage of good conditioning by using an iterative method such as stabilized biconjugate gradients [Barrett et al., 1994]. In either case, we can expect to be able to evaluate policies efficiently even in large Markov decision processes.

The policy improvement step is where we hope to beat policy iteration. By performing

main():
  $(\forall x)\ v(x) \leftarrow M,\ v_{\text{old}}(x) \leftarrow M$
  $v(\text{goal}) \leftarrow 0,\ v_{\text{old}}(\text{goal}) \leftarrow 0$
  while (true) **do**
    $(\forall x)\ \pi(x) \leftarrow$ undefined
    $\Delta \leftarrow 0$
    sweep()
    if ($\Delta <$ tolerance) **then**
      declare convergence
    end if
    $(\forall x)\ v_{\text{old}}(x) \leftarrow v(x)$
    $v \leftarrow$ evaluate policy $\pi(x)$
  end while

sweep():
  $(\forall x)\ \text{closed}(x) \leftarrow$ false
  $(\forall x)\ p_{\text{goal}}(x) \leftarrow 0$
  $\text{closed}(\text{goal}) \leftarrow$ true
  update(goal)
  while (not queue.isempty()) **do**
    $x \leftarrow$ queue.pop()
    $\text{closed}(x) \leftarrow$ true
    update(x)
  end while

update(x):
  for (all $((y, a) \in \mathsf{pred}(x))$ **do**
    if ($\text{closed}(y)$) **then**
      $Q(y, a) \leftarrow c(y, a) + \sum_{x' \in \mathsf{succ}(y,a)} P^a_{yx'} v(x')$
      $\Delta \leftarrow \max(\Delta, v(y) - Q(y, a))$
    else
      for all actions b **do**
        $Q_{\text{old}} \leftarrow Q(y, \pi(y))$ (or M if $\pi(y)$ undefined)
        $Q(y, b) \leftarrow c(y, b) + \sum_{x' \in \mathsf{succ}(y,b)} P^b_{yx'} v(x')$
        $p_{\text{goal}}(y, b) \leftarrow \sum_{x' \in \mathsf{succ}(y,b)} P^b_{yx'} p_{\text{goal}}(x') + P^b_{y\mathsf{g}}$
        if ($Q(y, b) < Q_{\text{old}}$) **then**
          $v(y) \leftarrow Q(y, b)$
          $\pi(y) \leftarrow b$
          $p_{\text{goal}}(y) \leftarrow p_{\text{goal}}(y, b)$
          $\text{pri} \leftarrow \langle 1 - p_{\text{goal}}(x),\ (v(y) - v_{\text{old}}(y))/v(y) \rangle$
          queue.decreasepriority(y, pri)
        end if
      end for
    end if
  end for

Figure 2.5: The Prioritized Policy Iteration algorithm. As before, "queue" is a priority min-queue and M is a very large positive number.

a prioritized sweep through state space, so that we examine states near the goal before states farther away, we can base many of our policy decisions on multiple steps of look-ahead. Scheduling the expansions in our sweep according to one of the priority functions previously discussed insures PPI reduces to Dijkstra's algorithm: when we run it on a deterministic MDP, the first sweep will compute an optimal policy and value function, and will never encounter a Bellman error in a closed state. So Δ will be 0 at the end of the sweep, and we will pass the convergence test before evaluating a single policy. On the other hand, if there are no action choices then PPI will not be much more expensive than solving a single set of linear equations: the only additional expense will be the cost

of the sweep. If B is a bound on the number of outcomes of any action, then this cost is $O((BA)^2 S \log S)$, typically much less expensive than solving the linear equations (assuming $B, A << S$). For PPI, we chose to use the pri_2 schedule from equation (2.5). Unlike pri_1 (equation (2.4)), pri_2 forces us to expand states with high p_{goal} first, even when we have initialized v to the value of a near-optimal policy.

In order to guarantee convergence, we need to set $\pi(x)$ to a greedy action with respect to v before each policy evaluation. Thus in the **update**(x) method of PPI, for each state y for which there exists some action that reaches x, we re-calculate $Q(y, b)$ values for all actions b. In IPS, we only calculated $Q(y, b)$ for actions b that reach x. The extra work is necessary in PPI because the stored Q values may be unrelated to the current v (which was updated by policy evaluation), and so otherwise $\pi(x)$ might not be set to a greedy action. Other Q-value update schemes are possible,[4] and will lead to convergence as long as they fix a greedy policy. Note also that extra work is done if the loops in **update** are structured as in Figure 2.5; with a slight reduction in clarity, they can be arranged so that each predecessor state y is backed up only once.

One important additional tweak to PPI is to perform multiple sweeps between policy evaluation steps. Since policy evaluation tends to be more expensive, this allows a better tradeoff to be made between evaluation and improvement via expansions.

A potentially important optimization is to restrict the policy evaluations to a subset of the states. By fixing a reduced set of states (called an envelope [Dean et al., 1995]) which contains mostly "important" states, we can hope to gain most of the benefits of policy evaluation at a fraction of the cost. There are many ways to pick an envelope; for example, the LAO* algorithm [Hansen and Zilberstein, 2001] is one popular approach.

2.3.3 Gauss-Dijkstra Elimination

The Gauss-Dijkstra Elimination algorithm continues the theme of taking advantage of both Dijkstra's algorithm and efficient policy evaluation, but it interleaves them at a deeper level.

Gaussian Elimination and MDPs Fixing a policy π for an MDP produces a Markov chain and a vector of costs c. If our MDP has S states (not including the goal state), let P^π be the $S \times S$ matrix with entries $(P^\pi)_{xy} = P_{xy}^{\pi(x)}$ for all $x, y \neq \mathrm{goal}$. Finding the values of

[4] For example, we experimented with only updating $Q(y, b)$ when $P_{yx}^b > 0$ in **update** and then doing a single full backup of each state after popping it from the queue, ensuring a greedy policy. This approach was on average slower than the one presented above.

the MDP under the given policy reduces to solving the linear equations

$$(I - P^\pi)v = c$$

To solve these equations, we can run Gaussian elimination and backsubstitution on the matrix $(I - P^\pi)$. Gaussian elimination calls **rowEliminate**(x) (defined in Figure 2.6, where Θ is initialized to P^π and w to c) for all x from 1 to S in order,[5] zeroing out the subdiagonal elements of $(I - P^\pi)$. Backsubstitution calls **backsubstitute**(x) for all x from S down to 1 to compute $(I - P^\pi)^{-1}c$. In Figure 2.6, $\Theta_{x\cdot}$ denotes the x'th row of Θ, and $\Theta_{y\cdot}$ denotes the y'th row. We show updates to $p_{\text{goal}}(x)$ explicitly, but it is easy to implement these updates as an extra dense column in Θ.

To see why Gaussian elimination works faster than Bellman backups in MDPs with cycles, consider again the Markov chain of Figure 2.1. While value iteration reduces Bellman error by only 1% per sweep on this chain, Gaussian elimination solves it exactly in a single sweep. The starting $(I - P^\pi)$ matrix and c vector are:

$$\left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & -0.99 & -0.01 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \end{array}\right], \begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix}$$

(for clarity, we have shown $-p_{\text{goal}}(x)$ as an additional column separated by a bar). The first call to **rowEliminate** changes row 2 to:

$$\left[\begin{array}{ccccc|c} 0 & 1 & 0 & 0 & -0.99 & -0.01 \end{array}\right], \begin{bmatrix}2\end{bmatrix}$$

We can interpret this modified row 2 as a macro-action: we start from state 2 and execute our policy until we reach a state other than 1 or 2. (In this case, we will end up at the goal with probability 0.01 and in state 5 with probability 0.99.) Each subsequent call to **rowEliminate** zeros out one of the -1s below the diagonal and defines another macro-action of the form "start in state i and execute until we reach a state other than 1 through i." After four calls we are left with

$$\left[\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & -0.99 & -0.01 \\ 0 & 1 & 0 & 0 & -0.99 & -0.01 \\ 0 & 0 & 1 & 0 & -0.99 & -0.01 \\ 0 & 0 & 0 & 1 & -0.99 & -0.01 \\ 0 & 0 & 0 & -1 & 1 & 0 \end{array}\right], \begin{bmatrix}1\\2\\3\\4\\1\end{bmatrix}$$

[5]Using the Θ representation causes a few minor changes to the Gaussian elimination code, but it has the advantage that (Θ, w) can always be interpreted as a Markov chain which is has the same value function as the original (P^π, c). Also, for simplicity we will not consider pivoting; if π is a proper policy then $(I - \Theta)$ will always have a nonzero entry on the diagonal.

The last call to **rowEliminate** zeros out the last subdiagonal element (in line (1)), setting row 5 to:

$$\left[\begin{array}{ccccc|c} 0 & 0 & 0 & 0 & 0.01 & -0.01 \end{array}\right], \left[5\right] \tag{2.6}$$

Then it divides the whole row by 0.01 (line (2)) to get:

$$\left[\begin{array}{ccccc|c} 0 & 0 & 0 & 0 & 1 & -1 \end{array}\right], \left[500\right] \tag{2.7}$$

The division accounts for the fact that we may visit state 5 multiple times before our macro-action terminates: equation (2.6) describes a macro-action which has a 99% chance of self-looping and ending up back in state 5, while equation (2.7) describes the macro-action which keeps going after a self-loop (an average of 100 times) and only stops when it reaches the goal.

At this point we have defined a macro-action for each state which is guaranteed to reach either a higher-numbered state or the goal. We can immediately determine that $v^*(5) = 500$, since its macro-action always reaches the goal directly. Knowing the value of state 5 lets us determine $v^*(4)$, and so forth: each call to **backsubstitute** tells us the value of at least one additional state.

Note that there are several possible ways to arrange the elimination computations in Gaussian elimination. Our example shows *row Gaussian elimination*,[6] in which we eliminate the first $k-1$ elements of row k by using rows 1 through $k-1$; the advantage of using this ordering for GDE is that we need not fix an action for state x until we pop it from the priority queue and eliminate its row.

Gauss-Dijkstra Elimination Gauss-Dijkstra elimination combines the above Gaussian elimination process with a Dijkstra-style priority queue that determines the order in which states are selected for elimination. The main loop is the same as the one for PPI, except that the policy evaluation call is removed and **sweep**() is replaced by **GaussDijkstraSweep**(). Pseudo-code for **GaussDijkstraSweep**() is given in Figure 2.6.

When x is popped from the queue, its action is fixed to a greedy action. The outcome distribution for this action is used to initialize $\Theta_{x\cdot}$, and row elimination transforms $\Theta_{x\cdot}$ and $w(x)$ into a macro-action as described above. If $\Theta_{x,\text{goal}} = 1$, then we fully know the state's value; this will always happen for the $|S|$th state, but may also happen earlier. We do immediate backsubstitution when this occurs, which eliminates some non-zeros above the diagonal and possibly causes other states' values to become known. Immediate backsubstitution ensures that $v(x)$ and $p_{\text{goal}}(x)$ are updated with the latest information, improving

[6]This sequence is called the Doolittle ordering when used to compute a LU factorization.

main():
  $(\forall x)\ v(x) \leftarrow M$
  $v(\text{goal}) \leftarrow 0$
  while (true) **do**
    $(\forall x)\ \pi(x) \leftarrow$ undefined
    GaussDijkstraSweep()
    if ((max L_1 bellman error) $<$ tolerance) **then**
      declare convergence
    end if
  end while

GaussDijkstraSweep():
  while (not queue.empty()) **do**
    $x \leftarrow$ queue.pop()
    $\pi(x) \leftarrow \arg\min_a Q(x, a)$
    $(\forall y)\ \Theta_{xy} \leftarrow P_{xy}^{\pi(x)}$
    $w(x) \leftarrow c(x, \pi(x))$
    rowEliminate(x)
    $v(x) \leftarrow (\Theta_{x\cdot}) \cdot v + w(x)$
    $F = \{x\}$
    if ($\Theta_{x,\text{goal}} = 1$) **then**
      backsubstitute(x)
    end if
    $(\forall y \in F)$ update(y)
  end while

backsubstitute(x):
  for each y such that $\Theta_{yx} > 0$ **do**
    $p_{\text{goal}}(x) \leftarrow p_{\text{goal}}(x) + \Theta_{yx}$
    $w(y) \leftarrow w(y) + \Theta_{yx} v(x)$
    $\Theta_{yx} \leftarrow 0$
    if ($p_{\text{goal}}(y) = 1$) **then**
      backsubstitute(y)
      $F \leftarrow F \cup \{\text{y}\}$
    end if
  end for

rowEliminate(x):
  for (y from 1 to x-1) **do**
    $w(x) \leftarrow w(x) + \Theta_{xy} w(y)$
    $\Theta_{x\cdot} \leftarrow \Theta_{x\cdot} + \Theta_{xy}\Theta_{y\cdot}$ (1)
    $p_{\text{goal}}(x) \leftarrow p_{\text{goal}}(x) + \Theta_{xy} p_{\text{goal}}(y)$
    $\Theta_{xy} \leftarrow 0$
  end for
  $w(x) \leftarrow w(x)/(1 - \Theta_{xx})$
  $\Theta_{x\cdot} \leftarrow \Theta_{x\cdot}/(1 - \Theta_{xx})$ (2)
  $\Theta_{xx} \leftarrow 0$
  $p_{\text{goal}}(x) \leftarrow p_{\text{goal}}(x)/(1 - \Theta_{xx})$

Figure 2.6: Gauss-Dijkstra Elimination. The update(y) method is the same one used for PPI, but with the pri_2 priority function.

our priority estimates for states on the queue and possibly saving us work later (for example, in the case when our transition matrix is block lower triangular, we automatically discover that we only need to factor the blocks on the diagonal). Finally, all predecessors of the state popped and any states whose values became known are updated using the **update**() routine for PPI (in Figure 2.5). However, for GDE we use the pri_2 priority function.

Since S can be large, Θ will usually need to be represented sparsely. Assuming Θ is

stored sparsely, GDE reduces to Dijkstra's algorithm in the deterministic case; it is easy to verify the additional matrix updates require only $O(S)$ work. In a general MDP, initially it takes no more memory to represent Θ than it does to store the dynamics of the MDP, but the elimination steps can introduce many additional non-zeros. The number of such new non-zeros is greatly affected by the order in which the eliminations are performed. There is a vast literature on techniques for finding such orderings; Duff et al. [1986] provides a good introduction. One of the main advantages of GDE seems to be that for practical problems, the prioritization criteria we present produce good elimination orders as well as effective policy improvement.

Our primary interest in GDE stems from the wide range of possibilities for enhancing its performance; even in the naive form outlined it is usually competitive with PPI. We anticipate that doing "early" backsubstitution when states' values are mostly known (high $p_{\text{goal}}(x)$) will produce even better policies and hence fewer iterations. Further, the interpretation of rows of Θ as macro-actions suggests that caching these actions may yield dramatic speed-ups when evaluating the MDP with a different goal state. The usefulness of macro-actions for this purpose was demonstrated by Dean and Lin [1995]. A convergence-checking mechanism such as those used by LRTDP and HDP [Bonet and Geffner, 2003a,b] could also be used between iterations to avoid repeating work on portions of the state space where an optimal policy and value function are already known. The key to making GDE widely applicable, however, probably lies in appropriate thresholding of values in Θ, so that transition probabilities near zero are thrown out when their contribution to the Bellman error is negligible. Our current implementation does not do this, so while its performance is good on many problems, it can perform poorly on problems that generate lots of fill-in.

2.3.4 Incremental Expansions

In describing IPS, PPI, and GDE we have touched on a number of methods of updating v and Q values. In summary: Value iteration iteration repeatedly backs up states in an arbitrary order. Prioritized sweeping backs up states in an order determined by a priority queue. PPI and GDE also pop states from a priority queue, but rather than backing up the popped state, they backup up all of its predecessors. IPS pops states from a priority queue, but instead of fully backing up the predecessors of the popped state x, it only recomputes Q values for actions that might reach x.

Here we provide a more thorough accounting of the expansion mechanism used by

IPS. Suppose we are given an initial upper bound v_{old} on v^*. Then, we can define Q by

$$Q(x,a) = c(x,a) + \sum_y P^a_{xy} v_{\text{old}}(y)$$

and then v_{new} by $v_{\text{new}}(x) = \min_a Q(x,a)$. Note that rather than storing v_{new} we can simply store Q and $\pi(x)$, the greedy policy with respect to v_{old}. Our goal in an expansion operation is to set $v_{\text{old}}(x) \leftarrow v_{\text{new}}(x)$, and then update Q so it reflects this change, and then update v_{new} so that again $v_{\text{new}}(x) = \min_a Q(x,a)$. Perhaps the easiest way to ensure this property is via a *full expansion* of the state x:

$v_{\text{old}}(x) \leftarrow Q(x, \pi(x))$
for all $(y,b) \in \mathsf{pred}(x)$ **do**
  $Q(y,b) \leftarrow c(y,b) + \sum_{x' \in \mathsf{succ}(y,b)} P^b_{yx'} v_{\text{old}}(x')$
  if $(Q(y,b) < Q(y,\pi(y)))$ **then**
    $\pi(y) \leftarrow b$
  end if
end for

Doing such a full expansion requires $O(B)$ work per predecessor state-action pair. We can accomplish the same task with $O(1)$ work if we assume without loss of generality[7] $(\forall x, a)$ $P^a_{xx} = 0$, and perform an *incremental expansion*:

$\Delta(x) \leftarrow Q(x, \pi(x)) - v_{\text{old}}(x)$
for all $(y,b) \in \mathsf{pred}(x)$ **do**
  $Q(y,b) \leftarrow Q(y,b) + P^b_{yx} \Delta(x)$
  if $(Q(y,b) < Q(y,\pi(y)))$ **then**
    $\pi(y) \leftarrow b$
  end if
end for
$v_{\text{old}}(x) \leftarrow Q(x, \pi(x))$

However, when doing a full expansion, we have a better option for calculating $Q(y,b)$ than the one given above. We can update $Q(y,b)$ using $Q(x', \pi(x'))$ in place of $v_{\text{old}}(x')$, and

[7]Suppose $P^a_{xx} > 0$. There exists an optimal stationary policy, so if a is selected and a self-loop occurs, it is safe to assume that action a is selected again, until a new state is reached. In expectation this will take $1/(1 - P^a_{xx})$ trials, so in a pre-processing step we replace a with action a', which is equivalent to taking a until a new state is reached: we have $c(x,a') = c(x,a)/(1 - P^a_{xx})$, with transition probabilities given by setting $P^{a'}_{xx} = 0$, and normalizing P^a_{xy} for all $y \neq x$.

```
update(x):
 for all (y, b) ∈ pred(x) do
   Q_temp ← c(y, b) + Σ_{x'∈succ(y,b)} P^b_{yx'} v(x')
   if (Q_temp < v(y)) then
     pri ← (Q_temp − v(y))/(Q_temp + 1)
     π(y) ← b
     if (|v(y) − Q_temp| > ε) then
       queue.decreasepriority(y, pri)
     end if
     v(y) ← Q_temp
   end if
 end for
```

Figure 2.7: The **update** function for the Improved Prioritized Sweeping algorithm, implemented with a single value-function array v and a temporary variable Q_{temp}.

this may offer a tighter upper bound because $Q(x', \pi(x')) \leq v(x')$ when we pessimistically initialize. In our experiments, this method proved superior to doing incremental expansions, and it is the method used by Improved Prioritized Sweeping (see Figure 2.4 for the code). However, on certain problems incremental expansions may give superior performance. IPS based on incremental expansions tends to do more updates (at lower cost) and so priority queue operations account for a larger fraction of its running times. Thus, fast approximate priority queues might offer a significant advantage to incremental IPS implementations.

One final implementation note. Our pseudocode for IPS and PPI indicates that Q values for all actions are stored. While this is necessary if incremental expansions are performed, we do full expansions so the extra storage is not required. It is sufficient to store a single value for each state, which takes the place of Q_{old} and v in the pseudocode; newly calculated $Q(y, b)$ values can be replaced by a temporary variable Qt; the value is only relevant if it causes $v(y)$ to change, in which case we immediately assign $v(y)$ the value of the temporary for $Q(y, b)$ rather than waiting until y is popped from the queue. Figure (2.7) shows this modification to the original IPS update method given in Figure (2.4).

2.3.5 Experimental Results

We implemented IPS, PPI, and GDE and compared them to VI, Prioritized Sweeping, and LRTDP. All algorithms were implemented in Java 1.5.0 and tested on a 3Ghz Intel machine with 2GB of main memory under Linux.

Our PPI implementation uses a stabilized biconjugate gradient solver with an incomplete LU preconditioners as implemented in the Matrix Toolkit for Java [Heimsund, 2004]. No native or optimized code was used; using architecture-tuned implementations of the underlying linear algebraic routines could give a significant speedup.

For LRTDP we specified a few reasonable start states for each problem. Typically LRTDP converged after labeling only a small fraction of the the state space as solved, up to about 25% on some problems.

Experimental Domain

We describe experiments in a discrete 4-dimensional planning problem that captures many important issues in mobile robot path planning. Our domain generalizes the racetrack domain described previously in [Barto et al., 1995, Bonet and Geffner, 2003b,a, Hansen and Zilberstein, 2001]. A state in this problem is described by a 4-tuple, $s = (x, y, dx, dy)$, where (x, y) gives the location in a 2D occupancy map, and (dx, dy) gives the robot's current velocity in each dimension. On each time step, the agent selects an acceleration $a = (ax, ay) \in \{-1, 0, 1\}^2$ and hopes to transition to state $(x + dx, y + dy, dx + ax, dy + ay)$. However, noise and obstacles can affect the actual result state. If the line from (x, y) to $(x+dx, y+dy)$ in the occupancy grid crosses an occupied cell, then the robot "crashes," moving to the cell just prior to the obstacle and losing all velocity. (The robot does not reset to the start state as in some racetrack models.) Additionally, the robot may be affected by several types of noise:

- **Action Failure** With probability f_p, the requested acceleration fails and the next state is $(x + dx, y + dy, dx, dy)$.
- **Local Noise** To model the fact that some parts of the world are more stochastic than others, we mark certain cells in the occupancy grid as "noisy," along with a designated direction. When the robot crosses such a cell, it has a probability f_ℓ of experiencing an acceleration of magnitude 1 or 2 in the designated direction.
- **One-way passages** Cells marked as "one-way" have a specified direction (north, south, east, or west), and can only be crossed if the agent is moving in the indicated

	$\|S\|$	f_p	f_ℓ	% determ	O	notes
A	59,780	0.00	0.00	100.0%	1.00	deterministic
B	96,736	0.05	0.10	17.2%	2.17	$\|A\| = 1$
C	11,932	0.20	0.00	25.1%	4.10	$f_h = 0.05$
D	10,072	0.10	0.25	39.0%	2.15	cycle
E	96,736	0.00	0.20	90.8%	2.41	
F	21,559	0.20	0.00	34.5%	2.00	`large-b`
G	27,482	0.10	0.00	90.4%	3.00	

Table 2.1: Test problems sizes and parameters.

direction. Any non-zero velocity in another direction results in a crash, leaving the agent in the one-way state with zero velocity.

- **High-velocity noise** If the robot's velocity surpasses an L_2 threshold, it incurs a random acceleration on each time step with probability f_h. This acceleration is chosen uniformly from $\{-1, 0, 1\}^2$, excluding the $(0, 0)$ acceleration.

These additions to the domains allow us to capture a wider variety of planning problems. In particular, kinodynamic path planning for mobile robots generally has more noise (more possible outcomes of a given action as well as higher probability of departure from the nominal command) than the original racetrack domain allows. Action failure and high-velocity noise can be caused by wheels slipping, delays in the control loop, bumpy terrain, and so on. One-way passages can be used to model low curbs or other map features that can be passed in only one direction by a wheeled robot. And, local noise can model a robot driving across sloped terrain: downhill accelerations are easier than uphill ones.

Table 2.1 summarizes the parameters of the test problems we used. The "% determ" column indicates the percentage of (s, a) pairs with deterministic outcomes; our implementation uses a deterministic transition to apply the collision cost, so all problems have some deterministic transitions. The O column gives the average number of outcomes for non-deterministic transitions. All problems have 9 actions except for (B), which is a policy evaluation problem. Problem (C) has high velocity noise, with a threshold of $\sqrt{2} + \epsilon$. Figure 2.8 shows the 2D world maps for most of the problems.

To construct larger problems for some of our experiments, we consider linking copies of an MDP in series by making the goal state of the ith copy transitions to the start state of the $(i + 1)$st copy. We indicate k serial copies of an MDP M by M^k, so for example 22 copies of problem (G) is denoted (G^{22}).

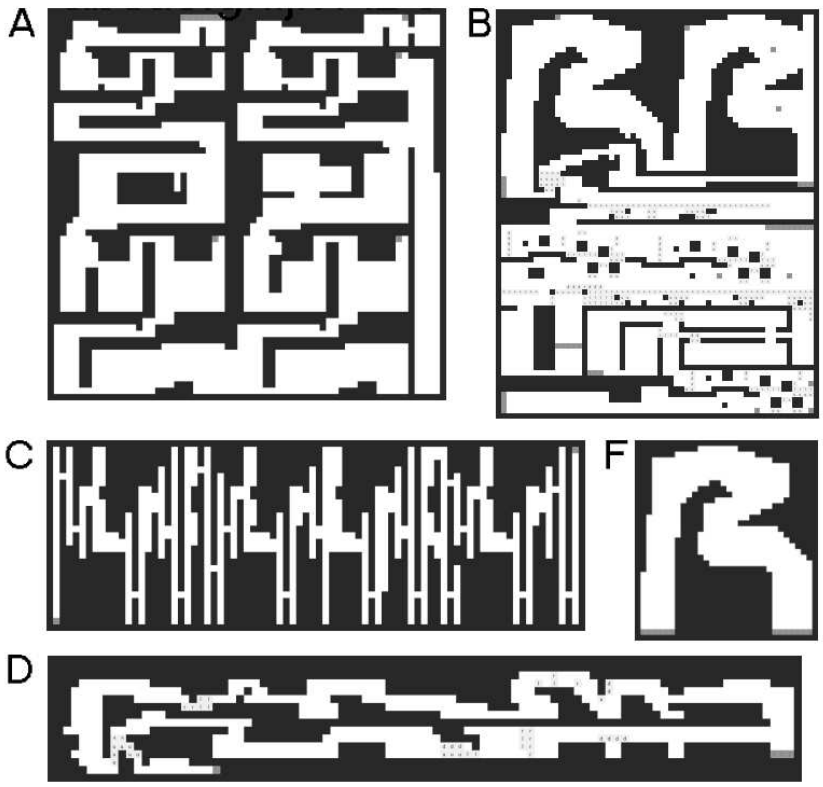


Figure 2.8: Some maps used for test experiments; maps are not drawn to the same scale. Problem (E) uses the same map as (B). Problem (G) uses a smaller version of map (B). Special states (one-way passages, local noise) are indicated by light grey symbols; contact the authors for full map specifications.

Experimental Results

Effects of Local Noise First, we considered the effect of increasing the randomness f_ℓ and f_p for the fixed map (G), a smaller version of (B). One-way passages give this complex map the possibility for cycles. Figure 2.9 shows the run times (y-axis) of several algorithms plotted against f_p. The parameter f_ℓ was set to $0.5 f_p$ for each trial.

These results demonstrate the catastrophic effect increased noise can have on the performance of VI. For low-noise problems, VI converges reasonably quickly, but as noise is increased the expected length of trajectories to the goal grows, and VI's performance

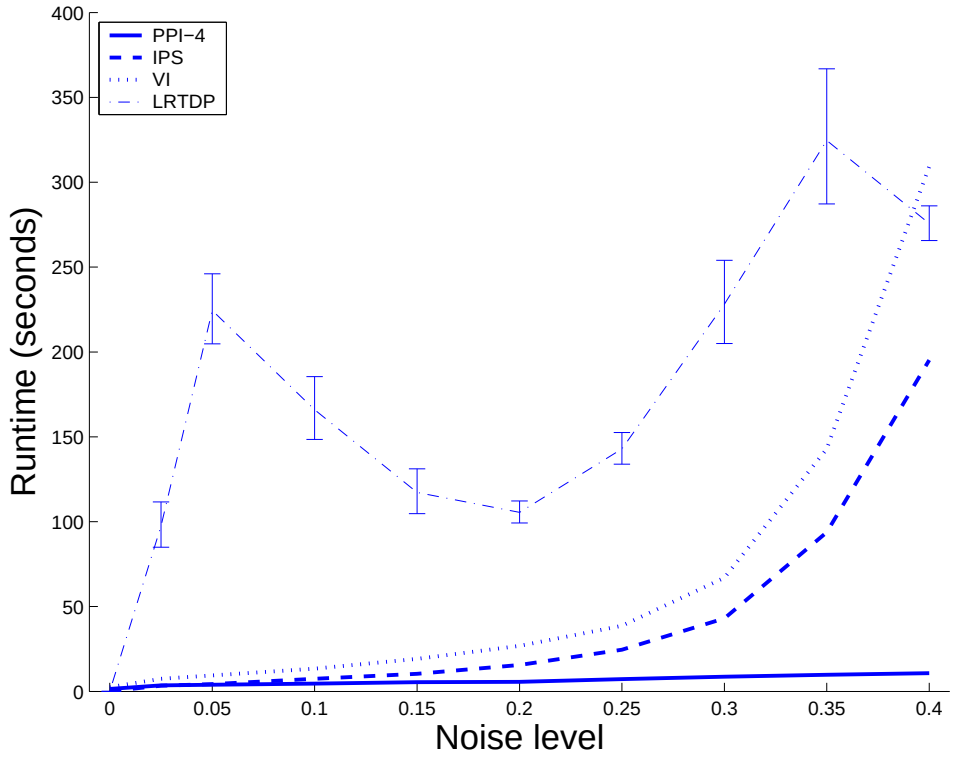


Figure 2.9: Effect of local noise on solution time. The leftmost data point is for the deterministic problem. Note that PPI-4 exhibits almost constant runtime even as noise is increased.

degrades accordingly. IPS performs somewhat better overall, but it suffers from this same problem as the noise increases. However, PPI's use of policy evaluation steps quickly propagates values through these cycles, and so its performance is almost totally unaffected by the additional noise. PPI-4 beats VI on all trials. It wins by a factor of 2.4 with $f_p = 0.05$, and with $f_p = 0.4$ PPI-4 is 29 times faster than VI.

The dip in runtimes for LRTDP is probably due to changes in the optimal policy, and the number and order in which states are converged. Confidence intervals are given for LRTDP only, as it is a randomized algorithm. The deterministic algorithms were run multiple times, and deviations in runtimes were negligible.

Number of Policy Evaluation Steps Policy iteration is an attractive algorithm for MDPs where policy evaluation via backups or expansions is likely to be slow. It is well known

that policy iteration typically converges in few iterations. However, Figure 2.10 shows that our algorithms can greatly reduce the number of iterations required. In problems where policy evaluation is expensive, this can provide a significant overall savings in computation time.

The number of iterations that standard policy iteration takes to converge depends on the initial policy. We experimented with initializing to the uniform stochastic policy,[8] random policies that at least give all states finite value, and an optimal policy for the deterministic relaxation of the problem.[9] The choice of initial policy rarely changed the number of iterations by more than 2 or 3, and in almost all cases initializing with the policy from the deterministic relaxation gave the best performance. Policy iteration was initialized in this way for the results in Figure 2.10.

We compare policy iteration to PPI, where we use either 1,2, or 4 sweeps of Dijkstra policy improvement between iterations. We also ran GDE on these problems. Typically it required the same number of iterations as PPI, but we hope to improve upon this performance in future work.

Q-value Computations Our implementation are optimized not for speed but for ease of use, instrumentation, and modification. We expect our algorithms to benefit much more from tuning than value iteration. To show this potential, we compare IPS, PS, and VI on the number of Q-value computations (Q-comps) they perform. A single Q-comp means iterating over all the outcomes for a given (s, a) pair to calculate the current Q value. A backup takes $|A|$ Q-comps, for example. We do not compare PPI-4, GDE, and LRTDP based on this measure, as they also perform other types of computation.

IPS typically needed substantially fewer Q-comps than VI. On the deterministic problem (A), VI required 255 times as many Q-comps as IPS, due to IPS's reduction to Dijkstra's algorithm; VI made 7.3 times as many Q-comps as PS. On problems (B) through (F), VI on average needed 15.29 times as many Q-comps as IPS, and 5.16 times as many as PS. On (G^{22}) it needed 36 times as many Q-comps as IPS. However, these large wins in number of Q-comps are offset by value iteration's higher throughput: for example, on problems (B) through (F) VI averaged 27,630 Q-comps per millisecond, while PS averaged 4,033

[8]This is a poor initialization not only because it is an ill-advised policy, but also because it often produces a poorly-conditioned linear system that is difficult to solve

[9]This is the policy chosen by an agent who can choose the outcome of each action, rather than having an outcome sampled from the problem dynamics. This policy and its value function can be computed by any shortest path algorithm or A^* if a heuristic is available. Note that this policy is different than the greedy policy with respect to the value function of the deterministic relaxation, which need not even be a proper policy. We will discuss this issue in greater depth in Section (2.4.1).

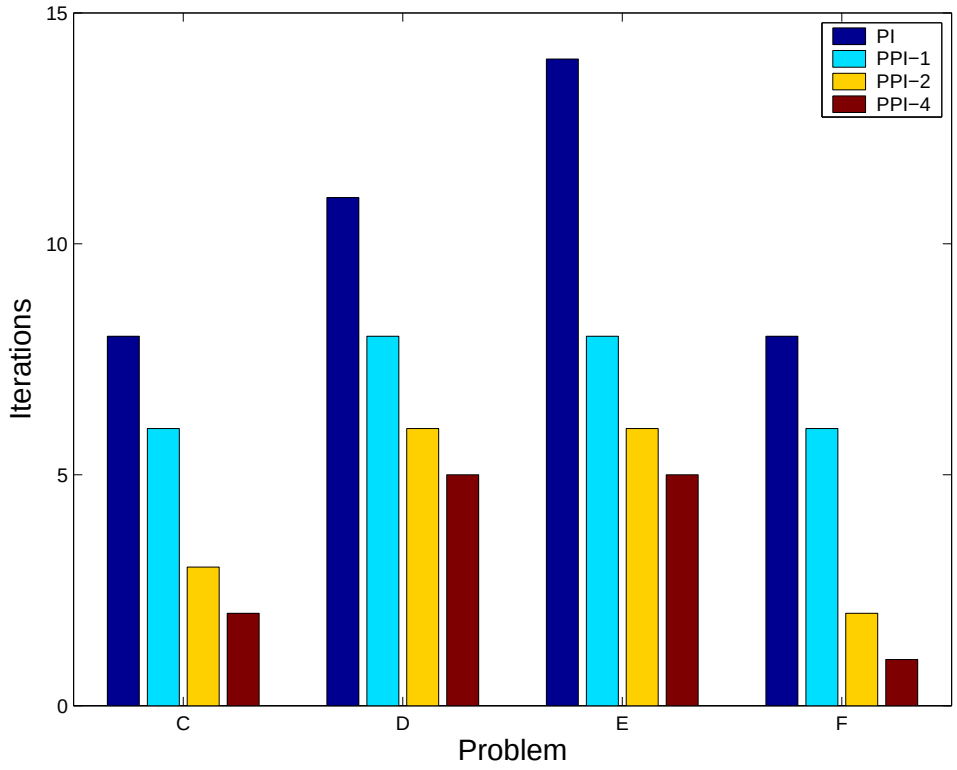


Figure 2.10: Number of policy evaluation steps.

and IPS averaged 3,393. PS and IPS will always have somewhat more overhead per Q-comp than VI. However, replacing the standard binary heap we implemented with a more sophisticated algorithm or with an approximate queuing strategy could greatly reduce this overhead, possibly leading to significantly improved performance.

Figure 2.11 compares the number of Q-comps required to solve serially linked copies of problem (D): the x-axis indicates the number of copies, from (D^1) to (D^8). VI still has competitive run-times because it performs Q-comps much faster. On (D^8) it averages 41,360 Q-comps per millisecond, while PS performs only 4,453 and IPS only 3,871.

Overall Performance of Solvers Figure 2.12 shows a comparison of the run-times of our solvers on the various test problems. Problem (G^{22}) has 623,964 states, showing that our approaches can scale to large problems.[10] On (G^{22}), the stabilized biconjugate gra-

[10]This experiment was run on a different (though similar) machine than the other experiments, a 3.4GHz Pentium under Linux with 1GB of memory.

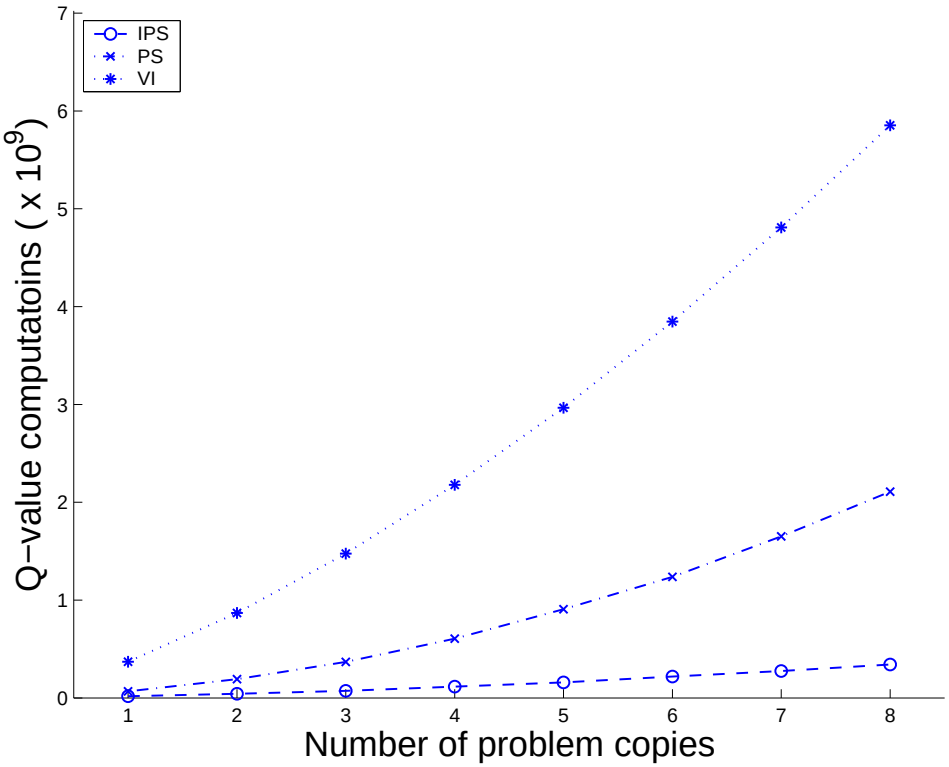


Figure 2.11: Comparison of number of Q-computations performed by IPS, PS, and VI to solve serially-linked copies of problem (D).

dient algorithm failed to converge on the initial linear systems produced by PPI-4, so we instead used PPI where 28 initial sweeps were made (so that there was a reasonable policy to be evaluated initially), and then 7 sweeps were made between subsequent evaluations. We also found that adding a pass of standard greedy policy improvement after the sweeps improved performance. These changes roughly balanced the time spent on sweeping and policy improvement. In future work we hope to develop more principled and automatic methods for determining how to split computation time between sweeps and policy evaluation. We did not run PS, LRTDP, or GDE on this problem.

Generally, our algorithms do best on problems that are sparsely stochastic (only have randomness at a few states) and also on domains where typical trajectories are long relative to the size of the state space. These long trajectories cause serious difficulties for methods that do not use an efficient form of policy evaluation. For similar reasons, our algorithms do better on long, narrow domains rather than wide open ones; the key factor is again the

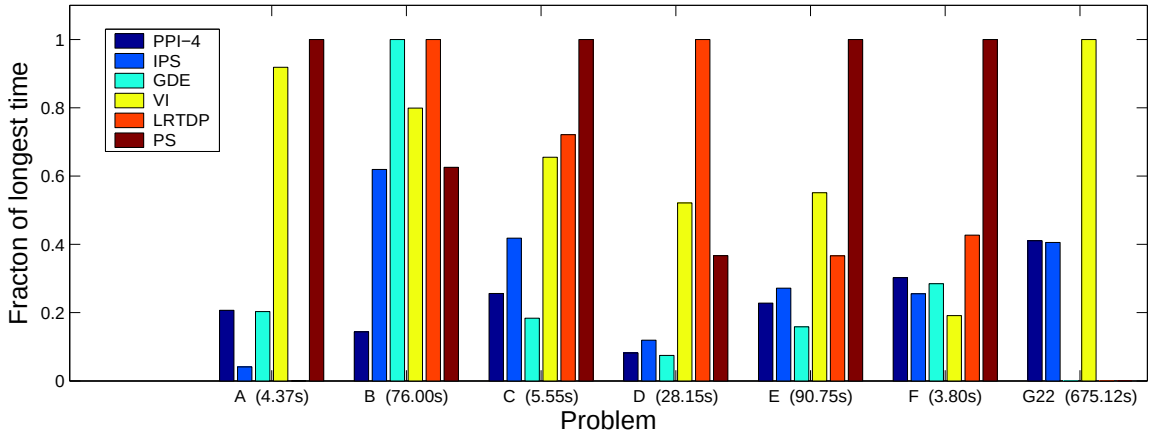


Figure 2.12: Comparison of a selection of algorithms on representative problems. Problem (A) is deterministic, and Problem (B) requires only policy evaluation. Results are normalized to show the fraction of the longest solution time taken by each algorithm. On problems (B) and (E), the slowest algorithms were stopped before they had converged. LRTDP is not charged for time spent calculating its heuristic, which is negligible in all problems except (A).

expected length of the trajectories versus the size of the state space.

Value iteration backed up states in the order in which states were indexed in the internal representation; this order was generated by a breadth-first search from the start state to find all reachable states. While this ordering provides better cache performance than a random ordering, we ran a minimal set of experiments and observed that the natural ordering performs somewhat worse (up to 20% in our limited experiments) than random orderings. Despite this, we observed better than expected performance for value iteration, especially as it compares to LRTDP and Prioritized Sweeping. For example, on the `large-b` problem (F), [Bonet and Geffner, 2003a] reports a slight win for LRTDP over VI, but our experiments show VI being faster.

Also, GDE's performance is typically close to or better than that of PPI-4, except on problem (B), where GDE fails due to moderately high fill in. These results are encouraging because GDE already sometimes performs better than PPI-4, and currently GDE is based on a naive implementation of Gaussian elimination and sparse matrix code. The literature in the numerical analysis community shows that more advanced techniques can yield dramatic speedups (see, for example, [Gupta, 2002]).

2.3.6 Discussion

The success of Dijkstra's algorithm has inspired many algorithms for MDP planning to use a priority queue to try to schedule when to visit each state. However, none of these algorithms reduce to Dijkstra's algorithm if the input happens to be deterministic. And, more importantly, they are not robust to the presence of noise and cycles in the MDP. For MDPs with significant randomness and cycles, no algorithm based on backups or expansions can hope to remain efficient. Instead, we turn to algorithms which explicitly solve systems of linear equations to evaluate policies or pieces of policies.

We have introduced a family of algorithms—Improved Prioritized Sweeping, Prioritized Policy Iteration, and Gauss-Dijkstra Elimination—which retain some of the best features of Dijkstra's algorithm while integrating varying amounts of policy evaluation. We have evaluated these algorithms in a series of experiments, comparing them to other well-known MDP planning algorithms on a variety of MDPs. Our experiments show that the new algorithms can be robust to noise and cycles, and that they are able to solve many types of problems more efficiently than previous algorithms could.

For problems which are fairly close to deterministic or with only moderate noise and cycles, we recommend Improved Prioritized Sweeping. For problems with fast mixing times or short average path lengths, value iteration is hard to beat and is probably the simplest of all of the algorithms to implement. For general use, we recommend the Prioritized Policy Iteration algorithm. It is simple to implement, and can take advantage of fast, vendor-supplied linear algebra routines to speed policy evaluation. All of these approaches are most appropriate when the agent may visit a large fraction of the state space, either because the agent's start state is unknown or because reaching the goal requires visiting much of the state space. In the next section, we consider problems where a fixed start state is known, and it is possible (with high probability) to reach the goal while only visiting a small fraction of the state space.

2.4 Bounded Real-Time Dynamic Programming

In this section we consider the problem of finding a policy in a Markov decision process with a fixed start state s, a fixed zero-cost absorbing goal state g, and non-negative costs. An arbitrary distribution over initial states can be modeled by adding an imaginary start state with a single action that produces the desired distribution. Perhaps the simplest algorithm for this problem is value iteration, which solves for an optimal policy on the full state space. Many realistic problems, however, are too large for such an approach and often

only a small fraction of the the state space is relevant to the problem of reaching g from s. This fact has inspired the development of a number of algorithms that focus computation on states that seem to be most relevant to finding an optimal policy from s. Such algorithms include Real-Time Dynamic Programming (RTDP) [Barto et al., 1995], Labeled-RTDP (LRTDP) [Bonet and Geffner, 2003b], LAO* [Hansen and Zilberstein, 2001], Heuristic Search/DP (HDP) [Bonet and Geffner, 2003a], Envelope Propagation (EP) [Dean et al., 1995], and Focused Dynamic Programming (FP) [Ferguson and Stentz, 2004].

Many of these algorithms use heuristics (lower bounds on the optimal value function) and/or sampled greedy trajectories to focus computation. In this section, we introduce Bounded RTDP (BRTDP), which is based on RTDP and uses both a lower bound and sampled trajectories. Unlike RTDP, however, it also maintains an upper bound on the optimal value function, which allows it to focus on states that are both relevant (frequently reached under the current policy) and poorly understood (large gap between upper and lower bound). Further, acting greedily with respect to an appropriate upper bound allows BRTDP to make anytime performance guarantees.

Finding an appropriate upper bound to initialize BRTDP can greatly impact its performance. One of the contributions of this work is an efficient algorithm for finding such an upper bound. Nevertheless, our experiments show that BRTDP performs well even when initialized naively.

We evaluate BRTDP on two criteria: *off-line convergence*, the time required to find an approximately optimal partial policy before any actions are taken in the real world; and *anytime performance*, the ability to produce a reasonable partial policy at any time after computation is started.

Our experiments show that when run off-line, BRTDP often converges much more quickly than LRTDP and HDP, which are known to have good off-line convergence properties. In fact, the gap in offline performance between BRTDP and competing algorithms can be arbitrarily large because of differences in how they check convergence. HDP, LRTDP, and LAO* (and most other algorithms of which we are aware[11]) have convergence guarantees based on achieving small Bellman residual on *all states* reachable under the current policy, while BRTDP only requires a small residual on states reachable *with significant probability*. Let $f_\pi(y)$ be the expected number of visits to state y given that the agent starts at s and executes policy π. We say an MDP has *dense noise* if all policies have many nonzero entries in f_π. For example, planning problems with action errors have $f_\pi > 0$

[11] After the initial submission of this work, it was pointed out that our exploration strategy is similar to that of the HSVI algorithm Smith and Simmons [2004]; since HSVI is designed for POMDPs rather than MDPs, the forms of the bounds that it maintains are different from ours, and its backup operations are much more expensive.

for all reachable states. (Action errors mean that, with some small probability, we take a random action rather than the desired one.) Dense noise is fairly common, particularly in domains from robotics. For example, Gaussian errors in movement will make every state have positive probability of being visited. Gaussian motion-error models are widespread, e.g. Ng et al. [2004]. Unpredictable motion of another agent can also cause large numbers of states to have positive visitation probability; an example of this sort of model is described by Roy et al. [2004].

For HDP or LRTDP to converge on problems with dense noise, they must do work that is at least linear in the number of nonzero entries in f_π, even if most of those entries are almost zero. BRTDP's bounds allow it to make performance guarantees on MDPs with dense noise without doing work linear in the number of states reachable under its greedy policy, potentially making it *arbitrarily* faster than HDP, LRTDP, and LAO*.

When used as an anytime algorithm, a suitably-initialized BRTDP consistently outperforms a similarly initialized RTDP (which is known to have good anytime properties). Without any initialization information, BRTDP is competitive with RTDP and sometimes better. Furthermore, given reasonable initialization assumptions, BRTDP will always return a policy with provable performance bounds. We know of no other MDP algorithms with this property.

In the next section we establish notation and define concepts needed to describe our algorithm. We then propose an algorithm for finding a monotone upper bound in time linear in the size of the state space. Section 2.4.3 explains BRTDP in detail and Section 2.4.4 describes different initialization scenarios and associated guarantees. In section 2.4.5 we formalize our notions of off-line convergence and anytime performance, and demonstrate that BRTDP can outperform existing algorithms on both of these tasks.

2.4.1 Basic Results

We will again work with value functions. Recall that the Bellman error for a value function v at a state x is given by $\mathsf{be}_v(x) = v(x) - \min_{a \in A} Q_v(x, a)$. We are particularly interested in monotone value functions: v is monotone optimistic (a monotone lower bound) if $\forall x,\ \mathsf{be}_v(x) \leq 0$ and monotone pessimistic (monotone upper bound) if $\forall x,\ \mathsf{be}_v(x) \geq 0$. We use the following two theorems, which can be proved using techniques from [Bertsekas and Tsitsiklis, 1996, Sec. 2.2].

Theorem 2.4.1. *If v is monotone pessimistic, then v is an upper bound on $v^\star$. Similarly, if v is monotone optimistic, then v is a lower bound on $v^\star$.*

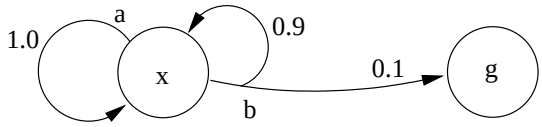


Figure 2.13: An MDP where the greedy policy with respect to v_d, the values from the deterministic relaxation, is improper. Costs are $c(x, a) = 1$ and $c(x, b) = 10$.

Theorem 2.4.2. *Suppose v_u is a monotone upper bound on $v^\star$. If π is the greedy policy with respect to v_u, for all x, $v_\pi(x) \leq v_u(x)$.*

No analog to Theorem 2.4.2 exists for a greedy policy based on a lower bound v_ℓ, monotone or otherwise: such a policy may be arbitrarily bad. Consider an arbitrary stochastic shortest path problem $\mathcal{M} = (S, A, P, c, \mathsf{s}, \mathsf{g})$, and consider the values found by solving $\mathcal{M}_d$, the deterministic relaxation of $\mathcal{M}$. That is, in $\mathcal{M}_d$ the planner can choose any outcome of any action from each state, rather than choosing an action and then facing a stochastic outcome. It is easy to show that the optimal values v_d for $\mathcal{M}_d$ are a monotonic lower bound on $v^\star$. Further, $\mathcal{M}_d$ is deterministic, so it can be solved via A^* or Dijkstra's algorithm. However, greedy(v_d) need not even be proper. Consider the MDP shown in Figure 2.13, and suppose $c(x, a) = 1$ and $c(x, b) = 10$. Then, $v_d(x) = 10$, and so $Q_{v_d}(x, a) = 11$ and $Q_{v_d}(x, b) = 19$ and the greedy policy with respect to v_d thus always selects action a. This observation is important because RTDP, LRTDP, HDP, and LAO* are often initialized to v_d, and they select actions greedily with respect to their value functions. Thus, initially these algorithms may produce arbitrarily bad stationary policies.

A proper policy, however, can always be extracted from v_d. In order for x to get a value $v_d(x)$ there must exist some $y \in S$ and $a \in A$ satisfying $P^a_{xy} > 0$ and $v_d(x) = c(x, a) + v_d(y)$. Then, it is not hard to show that we can construct a proper policy π_d by setting $\pi_d(x) = a$ for any such a; if there are multiple such actions, it is natural to pick the one with highest P^a_{xy}.

In summary, monotone lower bounds can have arbitrarily bad greedy policies, but greedy policies for monotone upper bounds do at least as well as the bound. Thus, we believe (and our experimental results demonstrate) that there is significant advantage to having an anytime algorithm that returns a policy that is greedy with respect to a monotone upper bound on the value function. The intuition is that if we have not finished planning and must return some non-optimal plan to be executed, it is wise to be pessimistic about regions of the state space where we haven't done much work. However, during the planning phase (that is, in simulation), being optimistic about relatively unexplored regions is beneficial. Thus, BRTDP gains an advantage by maintaining both an upper and lower bound on

the value function. The question of how to efficiently compute an initial monotone upper bounds is addressed in the next section.

2.4.2 Monotonic Upper Bounds in Linear Time

Our planning algorithm, BRTDP, is described below in Section 2.4. It can be initialized with any upper and lower bounds v_u and v_ℓ on $v^\star$, and provides performance guarantees if v_u and v_ℓ are monotone. So, we need to compute monotone bounds v_u and v_ℓ efficiently. This section describes how to do so assuming we can afford to visit every state a small number of times; Section 2.4.4 describes looser bounds which don't require visiting all of S. As noted above, we can initialize v_ℓ to the value of the deterministic relaxation $\mathcal{M}_d$; so, the remainder of this section deals with v_u.

For any proper policy π, the value function v_π is a monotone upper bound. A proper policy can be found reasonably quickly, for example by computing π_d from the deterministic relaxation. Unfortunately, directly solving the linear system to evaluate π requires about $\mathcal{O}(|S|^3)$ time in the worst case.[12] This is the fastest technique currently in the literature of which we are aware. We introduce a new algorithm, which we call Dijkstra Sweep for Monotone Pessimistic Initialization (DS-MPI), which can compute a monotone upper bound in $\mathcal{O}(|S| \log |S|)$ time.

Suppose we are given a policy π along with $p_{\text{goal}}, w \in \mathbb{R}_+^{|S|}$ that satisfy the following property: if we execute π from x until some fixed but arbitrary condition[13] is met, then $w(x)$ is an upper bound of the expected cost of the execution from x until the stopping condition is met, and $p_{\text{goal}}(x)$ is a lower bound on the probability the current state is the goal when execution is stopped. If $p_{\text{goal}}(x) > 0$ and $w(x)$ is finite for all x, we can use this information to derive an upper bound on v^*. We first informally motivate this derivation; then, we present a theorem that shows that with some additional conditions on w and p_{goal} we can derive a *monotone* upper bound. Finally, we give an efficient algorithm for finding the necessary w and p_{goal}.

Imagine executing π, starting from some state x, up until the stopping condition is met. This costs at most $w(x)$ in expectation, and with probability at least $p_{\text{goal}}(x)$ we reach the goal. But, suppose we don't reach the goal, and instead arrive at some other state y. We have made no assumptions about π, and so y might be the "worst" state in the

[12] For some particular problems, sparse linear solvers or iterative methods may offer better performance.

[13] For example, we might execute π for t steps; or execute π until we reach a state in some subset of S. Formally, π can be an arbitrary (history-dependent) policy, and the stopping condition can be an arbitrary function Θ. If H is the set of all possible histories (trajectories), then $\Theta : H \to \{0, 1\}$, where $\theta(h) = 1$ implies stopping; Θ need only ensure that every trajectory stops after a finite number of steps

MDP. Nevertheless, we can re-start our execution of π from y, paying an additional $w(y)$ in expectation, and reaching the goal with probability at least $p_{\text{goal}}(y)$. We can continue repeating this process, and because $(\forall x)\ p_{\text{goal}}(x) > 0$, we will eventually reach the goal.

We can upper bound the expected cost in this process by explicitly considering an adversary that after each execution of π (up to the stopping condition) gets to teleport the planning agent to an arbitrary state y with probability $1-p_{\text{goal}}(x)$; with probability $p_{\text{goal}}(x)$, the process ends. We model this interactions as an MDP *for the adversary*: there is a single non-goal state, and one action for each state in the original problem, corresponding to the destination of the teleportation. We can solve this single-state MDP by computing the optimal value

$$\lambda_1 = \max_{x \in S} \frac{w(x)}{p_{\text{goal}}(x)}.$$

It follows that in the original MDP $\mathcal{M}$, for all x, $v^\star(x) \leq \lambda_1$. For any particular x, we also have $v^\star(x) \leq w(x) + (1 - p_{\text{goal}}(x))\lambda_1$: the value of a state can't be any worse than following π until the stopping condition is met (and paying $w(x)$), and then ending up at the worst state in the MDP with probability $1 - p_{\text{goal}}(x)$, which has value no greater than λ_1. Further, if we are given some $Z \in \mathbb{R}$ such that $v^\star(x) \leq Z$ for all x, then we can use Z in place of λ_1 and still have an upper bound by the same argument. However, without further assumptions,v_u (based on λ_1 or Z) need not be monotonic.

The next theorem generalizes this idea by showing how it can also be used to find a monotone upper bound, if w and p_{goal} each satisfy certain (monotonicity-like) conditions.

Theorem 2.4.3. *Suppose p_{goal} and w satisfy the conditions given above for some policy π. Further, suppose for all x, there exists an action a such that either*

$$\textit{(I)} \qquad p_{\text{goal}}(x) < \sum_{y \in S} P^a_{xy} p_{\text{goal}}(y)$$

or

$$\textit{(II)} \qquad w(x) \geq c(x,a) + \sum_{y \in S} P^a_{xy} w(y) \qquad \textit{and}$$
$$p_{\text{goal}}(x) = \sum_{y \in S} P^a_{xy} p_{\text{goal}}(y).$$

Define $\lambda(x,a)$ by

$$\lambda(x,a) = \frac{c(x,a) + \sum_{y \in S} P^a_{xy} w(y) - w(x)}{\sum_{y \in S} P^a_{xy} p_{\text{goal}}(y) - p_{\text{goal}}(x)}$$

when case (I) applies, and let $\lambda(x,a) = 0$ *when case (II) applies, and* $\lambda(x,a) = \infty$ *otherwise. Then, if we choose* $\lambda = \max_{x \in S} \min_{a \in A} \lambda(x,a)$ *the value function* $v_u(x) = w(x) + (1 - p_{\text{goal}}(x))\lambda$ *is a finite monotonic upper bound on* $v^\star$.

Proof. It is sufficient to show that all Bellman errors for v_u are positive, that is,

$$(\forall x) \qquad v_u(x) - \min_{a \in A} \left[c(x,a) + \sum_{y \in S} P^a_{xy} v_u(y) \right] \geq 0.$$

Plugging in the definition of v_u from above gives

$$(\forall x) \qquad w(x) + (1 - p_{\text{goal}}(x))\lambda \geq \min_{a \in A} \left[c(x,a) + \sum_{y \in S} P^a_{xy} \Big(w(y) + (1 - p_{\text{goal}}(y))\lambda \Big) \right].$$

We need to show that this inequality holds for all x given λ as defined by the theorem. It is sufficient to show the inequality holds for at least one action, so fix some $a' \in \operatorname{argmin}_a \lambda(x,a)$. By assumption $\lambda(x,a')$ is finite and hence either condition (I) or (II) applies. For case (I), we can solve the above inequality for λ, and arrive at $\lambda \geq \lambda(x,a')$. This condition is satisfied given our choice a' and the definition of λ from the theorem. If case (II) holds, any $\lambda \geq 0$ will satisfy the above inequality. It follows that v_u monotone. □

Now we will show how to construct the necessary w, p_{goal}, and corresponding π. The idea is simple: suppose state x_1 has an action a such that $P^a_{x_1 \mathtt{g}} > 0$. Then, we can set $w(x_1) = c(x_1, a)$ and $p_{\text{goal}}(x_1) = P^a_{x_1 \mathtt{g}}$. Now, consider some state x_2 that has an action a_2 such that $p = (P^{a_2}_{x_2 \mathtt{g}} + P^{a_2}_{x_2 x_1}) p_{\text{goal}}(x_1) > 0$. Then, we can set $p_{\text{goal}}(x_2)$ equal to p, and $w(x_2) = c(x_2, a_2) + P^{a_2}_{x_2 \mathtt{g}} 0 + P^{a_2}_{x_2 x_1} w(x_1)$. We can now select x_3 to be any state with an action that has positive probability of reaching g, x_1, or x_2, and we will be able to assign it a positive p_{goal}. The policy π corresponding to p_{goal} and w is given by $\pi(x_i) = a_i$, and the stopping condition ends a trajectory whenever a transition from x_i to x_j occurs with $j \geq i$. The p_{goal} and w values we compute are exact values, not bounds, for this policy and stopping condition.

To complete the algorithm, it remains to give a method for determining what state to select next when there are multiple possible states. We propose the greedy maximization of $p_{\text{goal}}(x_k)$: having fixed $x_1, \ldots, x_{k-1}$, select (x_k, a_k) to maximize $\sum_{i<k} P^{a_k}_{x_k x_i} p_{\text{goal}}(x_i)$. If there are multiple states that achieve the same $p_{\text{goal}}(x_k)$, we choose the one that minimizes $\sum_{i<k} P^{a_k}_{x_k x_i} w(x_i)$. Figure (2.14) gives the pseudocode for calculating p_{goal} and w; the queue is a min priority queue (with priorities in $\mathbb{R}^2$ which are compared according to

Initialization:

$\forall(x,a),\ \hat{p}_{\text{goal}}(x,a) \leftarrow 0;\ \ \forall a,\ \hat{p}_{\text{goal}}(\mathsf{g},a) \leftarrow 1$
$\hat{w}(x,a) \leftarrow c(x,a)$
p_{goal}, w initialized arbitrarily
$\forall x,\ \ \pi(x) \leftarrow$ undefined; $\ \pi(\mathsf{g}) \leftarrow$ (arbitrary action)
$\forall x,\ \ \mathsf{pri}(x) \leftarrow \infty$, $\mathsf{closed}(x) \leftarrow$ false

Sweep:

queue.enqueue(goal, 0)
while not queue.empty() **do**
  $x \leftarrow$ queue.pop()
  $\mathsf{closed}(x) \leftarrow$ true
  $w(x) \leftarrow \hat{w}(x, \pi(x))$
  $p_{\text{goal}}(x) \leftarrow p_{\text{goal}}(x, \pi(x))$
  for all (y,a) s.t.$(P^a_{yx} > 0)$ and (not $\mathsf{closed}(y)$) **do**
    $\hat{w}(y,a) \leftarrow \hat{w}(y,a) + P^a_{yx} w(x)$
    $\hat{p}_{\text{goal}}(y,a) \leftarrow \hat{p}_{\text{goal}}(y,a) + P^a_{yx} p_{\text{goal}}(x)$
    $\mathsf{pri} \leftarrow \langle 1 - \hat{p}_{\text{goal}}(y,a), \hat{w}(y,a) \rangle$
    if $\mathsf{pri} < \mathsf{pri}(y)$ **then**
      $\mathsf{pri}(y) \leftarrow \mathsf{pri}$
      $\pi(y) \leftarrow a$
      queue.decreaseKey$(y, \mathsf{pri}(y))$
    end if
  end for
end while

Figure 2.14: The DS-MPI procedure.

lexical order), and $\hat{p}_{\text{goal}}$ and $\hat{w}$ are analogous to the Q values for v. After applying the sweep procedure, one can apply Theorem 2.4.3 to construct v_u.

In fact, condition (I) or (II) will always hold for action a_k, and so it is sufficient to set $\lambda = \max_{x_i \in S} \lambda(x_i, a_i)$. To see this, consider the (x_k, a_k) selected by DS-MPI, after $x_1, \ldots, x_{k-1}$ have already been popped (i.e., fin(x_i) = true, $i < k$). Then, $p_{\text{goal}}(x_k) = \sum_{i<k} P^{a_k}_{x_k x_i} p_{\text{goal}}(x_i)$. At completion, all states x have $p_{\text{goal}}(x) > 0$, and so the only way $p_{\text{goal}}(x_k)$ can equal $\sum_{y \in S} P^{a_k}_{x_k y} p_{\text{goal}}(y)$ is if all outcomes $y \in \mathsf{succ}(x_k, a_k)$ were closed when $p_{\text{goal}}(x_k)$ was set. This implies that $\sum_{i<k} P^{a_k}_{x_k x_i} w(x_i) = \sum_{y \in S} P^{a_k}_{x_k y} w(y)$, and so

$w(y) = c(x, a) + \sum_{y \in S} P^{x_k}_{x_k y} w(y)$ and condition (II) holds. Otherwise, condition (I) must hold for (x_k, a_k). Additional backups of w and p_{goal} will preserve these properties, so if extra computation time is available or the p_{goal} values calculated initially are too small, additional sweeps will tighten the upper bound.

If the dynamics are deterministic, then we can always pick (x_k, a_k) so $p_{\text{goal}}(x_k) = 1$, and so our scheduling corresponds to that of Dijkstra's algorithm. This sweep is similar to the policy improvement sweeps done by the Prioritized Policy Iteration (PPI) algorithm described in Section 2.3. The primary differences are that the PPI version assumes it is already initialized to some upper bound and performs full Q updates, while this version performs incremental updates.

The running time of DS-MPI is $\mathcal{O}(|S| \log |S|)$ (assuming a constant number of actions and outcomes) if a standard binary heap is used to implement the queue. However, an unscheduled version of the algorithm will still produce a finite (though possibly much looser) upper bound, so this technique can be run in $\mathcal{O}(|S|)$ time. If no additional information is available, then this performance is the best possible for arbitrary MDPs: in general it is impossible to produce an upper bound on any state without doing $\mathcal{O}(|S|)$ work, since we must consider the cost at each reachable state.

2.4.3 Bounded RTDP

The pseudocode for Bounded RTDP is given in Algorithm 2.15. BRTDP has four primary differences from RTDP.

- It maintains upper and lower bounds v_u and v_ℓ of $v^\star$, rather than just a lower bound. When a policy is requested in an anytime manner (i.e., before convergence), the policy $\mathsf{greedy}(v_u)$ is returned; v_ℓ helps guide exploration in simulation.
- When trajectories are sampled in simulation, the outcome distribution is biased to prefer transitions to states with a large gap $(v_u(x) - v_\ell(x))$.
- BRTDP maintains a list of the states on the current trajectory, and when the trajectory terminates backups are done in reverse order along the stored trajectory.
- Trajectories terminate when they reach a state that has a "well-known" value, rather than when they reach the goal.

We assume BRTDP is initialized so that v_u is an upper bound, and v_ℓ is a lower bound. We defer a justification of this assumption to the next section.

Main loop:
while $(v_u(\mathsf{s}) - v_\ell(\mathsf{s})) > \alpha$ **do**
runSampleTrial()
end while

runSampleTrial:
$x \leftarrow \mathsf{s}$
traj $\leftarrow$ (empty stack)
while true **do**
traj.push(x)
$v_u(x) \leftarrow \min_a Q_{v_u}(x, a)$
$a \leftarrow \text{argmin}_a Q_{v_\ell}(x, a)$
$v_\ell(x) \leftarrow Q(x, a)$
$\forall y,\ \ b(y) \leftarrow P^a_{xy}(v_u(y) - v_\ell(y))$
$B \leftarrow \sum_y b(y)$
if$(B < (v_u(\mathsf{s}) - v_\ell(\mathsf{s}))/\tau)$ **then** break
$x \leftarrow$ sample from distribution $b(y)/B$
end while
while not traj.empty() **do**
$x \leftarrow$ traj.pop()
$v_u(x) \leftarrow \min_a Q_{v_u}(x, a)$
$v_\ell(x) \leftarrow \min_a Q_{v_\ell}(x, a)$
end while

Figure 2.15: The bounded RTDP algorithm.

Like RTDP, BRTDP performs backups along sampled trajectories that begin from s. From an arbitrary state x on the trajectory a greedy action a is selected with respect to v_ℓ. Let $b(y) = P^a_{xy}(v_u(y) - v_\ell(y))$, and let $B = \sum_{y \in S} b(y)$. Then, BRTDP samples the next state on the trajectory according to the distribution that assigns prob$(y) = b(y)/B$.

The value of the goal state is known to be zero, and so we assume $v_u(\mathsf{g}) = v_\ell(\mathsf{g}) = 0$ initially (and hence always). This implies that $b(\mathsf{g}) = 0$, and so our trajectories will never actually reach the goal. It is natural to end trajectories when a "known" state is reached. For BRTDP, "known" corresponds to states with small gap. However, the smaller the gap the less likely we are to reach the state, so we instead look at the expected gap under the greedy action with respect to the unbiased transition probabilities. The normaliz-

ing constant B has exactly this interpretation, so we terminate the trajectory when B is small. We experimented with both constant thresholds and dynamic ones, and the different choices have relatively minor impacts on performance. We found the adaptive criterion $B \leq (v_u(\mathsf{s}) - v_\ell(\mathsf{s}))/\tau$, where $\tau > 1$ is a constant (say from 10 to 100), to be as good as anything. Figure 2.15 gives the complete pseudocode for the algorithm.

A convergence proof for BRTDP must be very different from the standard one for RTDP. Proving the convergence of RTDP typically relies on the claim that all states reachable under the greedy policy are backed up infinitely often in the limit [Bertsekas and Tsitsiklis, 1996]. In the face of dense noise, this implies convergence will require visiting the full state space. We take convergence to mean $v_u(\mathsf{s}) - v_\ell(\mathsf{s}) \leq \alpha$ for some error tolerance α, and BRTDP can achieve this (given a good initialization) without visiting the whole state space even with dense noise. A detailed proof can be formed by establishing that: (1) v_u and v_ℓ remain upper and lower bounds on $v^\star$, (2) trajectories have finite expected lengths, and (3) every trajectory has a positive probability of increasing v_ℓ or decreasing v_u.

2.4.4 Initialization Assumptions and Performance Guarantees

We assume that at the beginning of planning, the algorithm is given $\mathcal{M}$, including s. As mentioned in Section 2.4.2, if this is the only information available, then on arbitrary problems it may be necessary to consider the whole state space to prove any performance guarantee.

LRTDP, HDP, and LAO* can converge on some problems without visiting the whole state space. This is possible if there exists an $E \subset S$ such that some approximately optimal policy π has $f_\pi(y) > 0$ only for $y \in E$, and further, a tight lower bound on s can be proved by only considering states inside E and possibly a lower bound provided at initialization. While some realistic problems have this property, many do not, including those with dense noise. The question, then, is what is the minimal amount of additional information that might allow convergence guarantees while only visiting a small fraction of S on arbitrary problems. We propose that the appropriate assumption is that an achievable upper bound (v_u^0, π^0) is known; here v_u^0 is some upper bound (it need not be monotone) on v_{π^0} (and hence $v^\star$), where π^0 is known. Such a pair is almost always available trivially, for example, by letting $v_u^0(x) \leftarrow Z$ where Z is some worst-case cost of system failure, and letting $\pi^0(x)$ be the sit-and-wait-for-help action, or something similar. Even such trivial information may be enough to allow convergence while visiting a small fraction of the state space.[14].

[14]This raises the issue of risk-sensitivity in planning. A one in a million chance of a million dollar failure

	S	$v_{\pi_d}(\mathsf{s})$	$v_u(\mathsf{s})$	$v_{\pi'}(\mathsf{s})$	$v^\star(\mathsf{s})$	$v_d(\mathsf{s})$
A	21559	29	63	32	23	19
B	21559	1.3e10	26.9	20.1	19.9	19.0
C	6834	15283	1642	485	176	52
D	6834	7662	182.1	117.1	116.7	63.0

Table 2.2: Test problems sizes and start-state values.

It is easiest to see how to use (v_u^0, π^0) via a transformation. Consider $M' = (S, A \cup \{\phi\}, \tilde{P}, \tilde{c}, \mathsf{s}, \mathsf{g})$, where ϕ is a new action that corresponds to switching to π^0 and following it indefinitely. This action has $\tilde{P}^{\phi}_{x\mathsf{g}} = 1.0$ and costs $\tilde{c}(x, \phi) = v_u^0(x)$; for all other actions, $\tilde{P} = P$ and $\tilde{c} = c$. We know $v_u^0 \geq v_{\pi^0} \geq v^\star$, and so adding the action ϕ does not change the optimal values, so solving $\mathcal{M}'$ is equivalent to solving $\mathcal{M}$. Suppose we run BRTDP on $\mathcal{M}'$, but extract the current upper bound v_u^t before convergence; then, v_u^t need not be monotone for $\mathcal{M}$, though it will be for $\mathcal{M}'$. We show how to construct a policy for $\mathcal{M}$ using only v_u^t that achieves the values v_u^t. At a state where $v_u^t(x) \geq \min_{a \in A} Q_{v_u^t}(x, a)$, we play the greedy action, and the performance guarantee follows from the standard monotonicity argument. Suppose, however, we reach a state x where $v_u^t(x) < \min_{a \in A} Q_{v_u^t}(x, a)$. Then, it is not immediately clear how to achieve this value. However, we show that in this case $v_u^t(x) = v_u^0(x)$, and so we can switch to π^0 to achieve the value. Suppose v_u^{t-1} was the value function just before BRTDP backed up x most recently. Then, BRTDP assigned $v_u^t(x) \leftarrow \min_{a \in A \cup \phi} Q_{v_u^{t-1}}(x, a)$. Since v_u^0 is monotone (for $\mathcal{M}'$, on which BRTDP is running), $Q_{v_u^{t-1}}(x, a) \geq Q_{v_u^t}(x, a)$, and so the only way we could have $v_u^t(x) < \min_{a \in A} Q_{v_u^t}(x, a)$ is if the auxiliary action ϕ achieved the minimum, implying $v_u^t(x) = v_u^0(x)$.

Thus, we conclude that via this transformation it is reasonable to assume BRTDP is initialized with monotone upper bound, implying that at any point in time BRTDP can return a stationary policy with provable performance guarantees. This policy will be greedy in $\mathcal{M}'$, but may be non-stationary on $\mathcal{M}$ as it may fall back on π^0. This potential non-stationary behavior is critical to providing a robust suboptimal policy.

2.4.5 Experimental Results

We test BRTDP on two discrete domains. The first is the 4-dimensional racetrack domain, described in [Barto et al., 1995, Bonet and Geffner, 2003b,a, Hansen and Zilber-

might be acceptable in an expected sense, but it might be preferable to pay $2 for insurance.

stein, 2001]. Problems (A) and (B) are from this domain, and use the `large-b` racetrack map [Bonet and Geffner, 2003a]. Problem (A) fixes a 0.2 probability of getting the zero acceleration rather than the chosen control, similar to test problems from the above references. Problem (B) uses the same map, but uses a dense noise model where with a 0.01 probability a random acceleration occurs. Problems (C) and (D) are from a 2D gridworld problem, where actions correspond to selecting target cells within a Euclidean distance of two (giving 13 actions). Both instances use the same map. In (C), the agent accidentally targets a state up to a distance 2 from the desired target state, with probability 0.2. In (D), however, a random target state (within distance 2) is selected with probability 0.01. Note that problems (A) and (C) have fairly sparse noise, while (B) and (D) have dense noise.

Figure 2.2 summarizes the sizes of S for the test problems. The other columns provide information to enable an evaluation of the DS-MPI. The $v_{\pi_d}(\mathsf{s})$ column gives the value of the start state under a policy π_d derived from solving the deterministic relaxation (see Section 2.4.1). The next column, $v_u(\mathsf{s})$, gives the value computed via DS-MPI. We let $\pi' = \mathsf{greedy}(v_u)$, and give $v_{\pi'}(\mathsf{s})$. This data shows that DS-MPI can produce high-quality upper bounds that have high-quality greedy policies, despite running in $\mathcal{O}(|S| \log |S|)$ time rather than the $\mathcal{O}(|S|^3)$ needed to compute v_{π_d}. The final column gives $v_{\pi_d}(\mathsf{s})$, the value of the start-state under the value function of the deterministic relaxation.

Anytime Performance

We compare the anytime performance of BRTDP to RTDP on the test domains listed in Figure 2.2, considering both *informed* initialization and *uninformed* initialization for both algorithms. Informed initialization means RTDP has its value function initially set to v_d, and BRTDP has v_ℓ set to v_d and v_u set by running DS-MPI. For uninformed initialization, RTDP has v_u set uniformly to zero, and BRTDP has v_ℓ set to zero and v_u set to 10^6.

Figure 2.16 gives anytime performance curves for the algorithms on each of the test problems. There are many possible models of online interaction between a *planner* that produces *policies* and an *actor* that executes them. In general, this interaction can be quite complex. We adopt the precursor deliberation [Dean et al., 1995] or anytime model, wherein we interrupt each algorithm at fixed intervals to consider the quality of the policy available at that time. Rather than simply evaluating the current greedy policy, we assume the executive agent has some limited computational power and can itself run RTDP on a given initial value function received from the planner. (This assumption results in a fairer comparison for RTDP, since that algorithm's greedy policy may be improper.) To evaluate a value function v, we place an agent at the start state, initialize its value function to v, run RTDP until we reach the goal, and record the cost of the resulting trajectory. The curves in

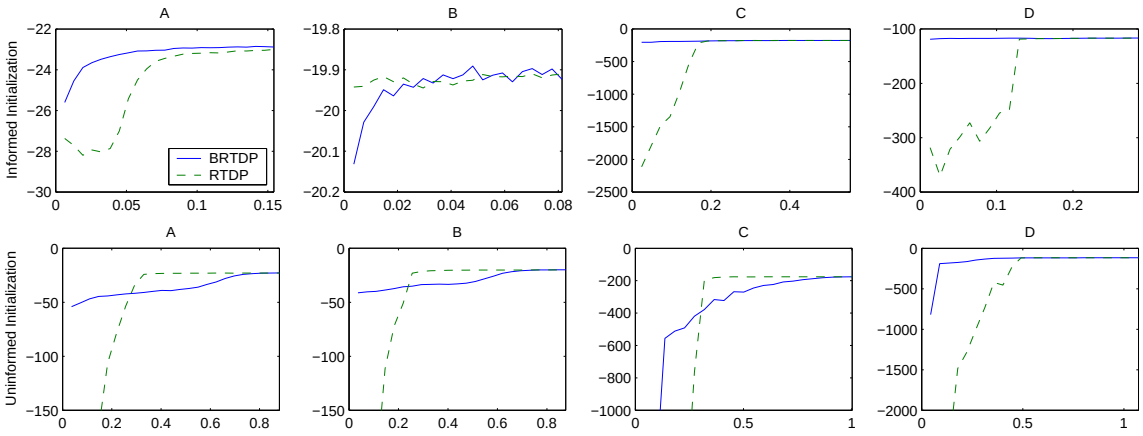


Figure 2.16: Anytime performance of informed and uninformed RTDP and BRTDP: the first row is for the informed initialization, and the second for uninformed. The X-axis gives number of backups ($\times 10^5$), and the Y axis indicates the current value of the policy; Y-axis labels are negative costs, so higher numbers are better. Note the differences in scales.

Figure 2.16 are the result of 100 separate runs of each algorithm, with each value function evaluated using 200 repetitions of the above testing procedure.

BRTDP performs 4 backups for each state on the trajectories it simulates: one each on v_u and v_ℓ during forward simulation, and one each while traversing the trajectory in reverse order. RTDP performs only one backup per sampled state. This gives BRTDP lower overhead and better cache performance per backup, and on the test problems we observed it *computed 1.5 to 3 times more backups per unit of runtime* than RTDP. Thus, if Figure 2.16 was re-plotted with time as the X-axis, the performance of BRTDP would appear even stronger. So, in interpreting the results from this section one should realize we have handicapped BRTDP in two ways: we compare it to RTDP in terms of number of updates rather than CPU time, and we evaluate RTDP-trajectories rather than stationary policies, even though stationary policies taken from BRTDP have provable guarantees.

Several conclusions can be drawn the results in Figure 2.16. First, appropriate initialization provides significant help to both RTDP and BRTDP. Second, under both types of initialization, BRTDP often provides much higher-reward policies than RTDP for a given number of backups (especially with a small number of backups, and especially with informative initialization), and we never observed its policies to be much worse than RTDP. In particular, observe that on problems (C) and (D) BRTDP is nearly optimal from the very beginning. This, combined with the fact that BRTDP provides performance bounds even

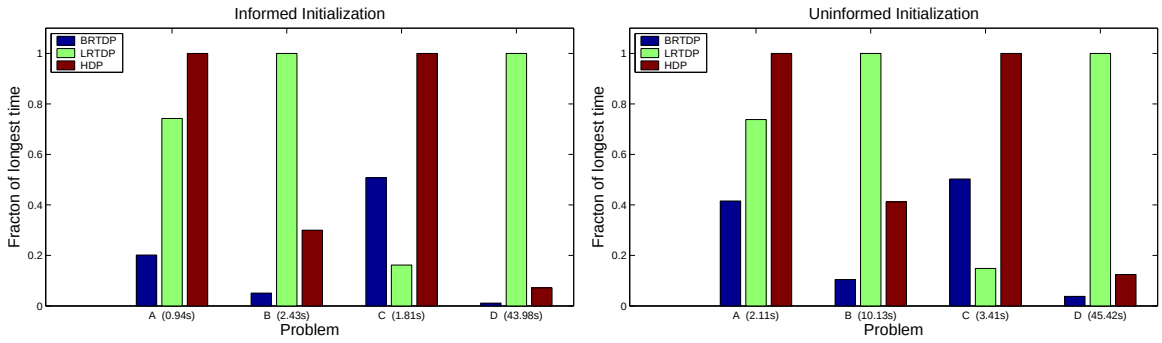


Figure 2.17: CPU time required for convergence with *informed* (left) and *uninformed* (right) initialization of the algorithms.

for stationary policies, make BRTDP a very attractive option for anytime applications.

Off-line Convergence

We compare off-line convergence times for BRTDP to those of LRTDP and HDP.[15] Again, we consider both informed and uninformed initialization. Informed LRTDP and HDP have their value functions initialized to v_d, while uninformed initialization sets them to zero. Time spent computing informed initialization values is not charged to the algorithms; this time will be somewhat longer for BRTDP as it also uses an upper bound heuristic, however, this time is typically dominated by the algorithm runtime.

We evaluate the algorithms by measuring the time it takes to find an α-optimal partial policy. For BRTDP, since we maintain upper and lower bounds, we can simply terminate when $(v_u(\mathsf{s}) - v_\ell(\mathsf{s})) \leq \alpha$; we used $\alpha = 0.1$ in our experiments. As discussed in Section 2.4.2 we initialized v_u to a monotone upper bound, so the greedy policy with respect to the final v_u will be within α of optimal. The other tested algorithms measure convergence by stopping when the max-norm Bellman error drops below some tolerance ϵ. Without further information there is no way to translate ϵ into a bound on policy quality: we can incur an extra cost of ϵ at each step of our trajectory, but since our trajectory could have arbitrarily many steps we could be arbitrarily suboptimal by the end. To provide an approximately equivalent stopping criterion, we used the following heuristic: pick an optimal policy π^* and let $\ell^\star(x)$ be the expected number of steps to reach g from x by following π^*. Then take $\epsilon = \alpha/\ell^\star(\mathsf{s})$. This heuristic yielded $\epsilon = 0.004$, 0.005, 0.001, and 0.002 for

[15] Improved LAO* is very similar to HDP without labeling solved states, and [Bonet and Geffner, 2003a] shows HDP has generally better performance, so LAO* was not considered in our experiments.

problems (A) through (D).

As expected, on (B) and (D), the problems with dense noise, BRTDP significantly outperformed the other algorithms. On (D), uninformed BRTDP is 3.2 times faster than uninformed HDP, and informed BRTDP is 6.4 times faster than informed HDP. Uninformed BRTDP outperforms informed HDP on (D) by a factor of 1.8. More importantly, on (B) and (D) HDP and LRTDP visit all of S before convergence, while (for example) on (B), informed BRTDP visits 28% of S and only brings $|v_u(x) - v_\ell(x)| \leq \alpha$ for 10% of S. If we, for example, add additional states beyond those BRTDP does not visit, the performance gap will become arbitrarily large.

Chapter 3

Bilinear-payoff Convex Games

Convex games generalize zero-sum matrix games by allowing arbitrary convex strategy sets in the place of explicitly enumerated finite strategy sets. This very general framework can compactly represent large games with sequential decisions. For example, extensive-form games can be represented compactly in this way; but, so can games with other kinds of structure, including path-planning games with uncertain outcomes and adversary controlled costs, and routing problems with adversary-controlled demands. In this chapter, we define the convex game model, introduce notation, and describe previous theoretical results on convex games. To demonstrate the utility of the framework, we then discuss three games that can be modeled as convex games, and also show how we can generalize stochastic games using convex games.

The representational power of convex games makes algorithms for their solution particularly important. It was shown by Koller et al. [1994] that polyhedral convex games can be solved via linear programming (that work focuses on the application to extensive-form games, but the formulation in fact holds for general convex games). Since that seminal result, the reduction to linear programming has been the state of the art for solving this class of problems. For example, sophisticated game-abstraction techniques combined with linear programming only recently allowed for the exact solution of Rhode Island Hold'em poker, a simplified version of the standard game of heads up, limit Texas Hold'em. Even after the application of the equilibria-preserving abstraction, solving the corresponding linear program exactly took over 7 days of CPU time and 25 GB of memory [Gilpin and Sandholm, 2005].

In fact, convex games often have significant structure that is not exploited by general-purpose linear programming algorithms. One way such structure can be exploited is through fast algorithms for calculating a best response strategy to a fixed strategy of

the opponent. The classic fictitious play algorithm takes advantage of such oracles, and demonstrates remarkably good performance on Rhode Island Hold'em. In Chapter 5, we develop a new algorithm for solving convex games that also uses best-response oracles; it outperforms fictitious play and dramatically outperforms the direct application of linear programming techniques.

While convex games are a straight-forward generalization of matrix games, the ability to represent arbitrary convex strategy sets lets us take advantage of structure in many types of games, often yielding exponentially smaller representations. In the coming sections, we will consider three examples that illustrate this point:

- As previously mentioned, **extensive-form games** (EFGs) can be transformed to convex games. While there are typically exponentially many (in the size of the game tree) pure strategies for an EFG, the set of behavioral strategies can be represented concisely as a convex set of achievable sequence weight vectors.

- The well-studied problem of computing an **optimal oblivious routing** can in fact be expressed as convex game. In this game, one player picks a routing in a network and the other picks traffic demands on source-sink pairs, subject to some constraints. There are exponentially many deterministic routings (pure strategies in the matrix game), but again there is a concise[1] representation of the set of strategies as a polyhedron. The details of expressing this problem as a convex game follow from work by Azar et al. [2003], though they did not connect their work to the convex game model and in fact a polynomial-sized LP formulation did not appear until [Applegate and Cohen, 2003]. The observation that optimal oblivious routing is a convex game is new, and the algorithms presented in Chapter 5 may be of practical interest for this problem.

- In **cost-paired MDP games**, each player selects a stochastic policy in an MDP, and their choice determines the costs in the opponent's MDP. The set of strategies (stochastic policies in the MDPs) for each player has a polynomial-sized representation as a polyhedron, but there are exponentially many deterministic policies and so the corresponding matrix game is exponential in both rows and columns.

We also show how convex games can be used to generalize stochastic games. Stochastic games extend MDPs to multiple players by embedding a matrix game at each state in an MDP; the next state distribution and cost to each player depends on the joint action

[1]The size of the representation of the constraints is polynomial in the size of the representation of the problem.

selected. In Section 3.5 we show how these matrix games can be replaced by convex games; this allows the embedding of extensive-form games at the nodes of stochastic games, creating a tractable class of partially-observable stochastic games (POSGs).

As these examples demonstrate, fast algorithms for convex games have far-reaching applicability. Before considering these examples in greater depth, we first establish some technical details about convex games.

3.1 From Matrix Games to Convex Games

A zero-sum matrix (normal-form) game is played by two players, player row with strategies $R = \{1, \ldots, m\}$ and player column with strategies $C = \{1, \ldots, n\}$. A $m \times n$ matrix M specifies the payoffs, so that if row plays strategy $i \in R$ and column plays $j \in C$, the payment from row to column is the $(i, j)th$ entry of M, denoted $M(i, j)$. The players select their strategies simultaneously, without knowledge of the other player's choice. We restrict our attention to two-player, zero-sum games as that is the simplest case and the one most relevant to our solution techniques; however, n-player general-sum matrix games (and convex games) can be defined analogously.

We use $\Delta(\cdot)$ to denote the probability simplex over a finite set, so for example

$$\Delta(R) = \left\{ p \in \mathbb{R}^m \;\middle|\; \sum_{i=1}^{m} p(i) = 1 \text{ and } p(i) \geq 0 \right\}.$$

A mixed strategy is an element $p \in \Delta(R)$ for the row player or $q \in \Delta(C)$ for the column player, corresponding to a distribution over the rows or columns, respectively. If the players select mixed strategies p and q, the expected payoff $V(p, q)$ from row to column is

$$V(p, q) = E_{i \sim p, j \sim q}[M(i, j)] = p^T M q.$$

A solution to the game is given by a minimax equilibrium (p^*, q^*), a pair of mixed strategies such that neither player has an incentive to play differently given that the other player plays their strategy from the pair. The minimax theorem says that if the players are allowed to select mixed strategies, there is no advantage to playing second. That is,

$$\min_{x \in \Delta(R)} \max_{y \in \Delta(C)} x^T M y = \max_{y \in \Delta(C)} \min_{x \in \Delta(R)} x^T M y. \tag{3.1}$$

Thus, solving either the min max or max min optimization problem from (3.1) finds a minimax equilibrium for the matrix game. This problem can easily be converted to a linear program and solved via standard techniques.

An ϵ-approximate minimax equilibrium for a matrix game is a pair of strategies (p', q') where neither player can gain more than ϵ value by switching to some other strategy. That is,

$$V(p', q') \leq \min_{p \in \Delta(R)} V(p, q') + \epsilon \tag{3.2}$$

$$V(p', q') \geq \max_{q \in \Delta(C)} V(p', q) - \epsilon. \tag{3.3}$$

Note that if $\epsilon = 0$ we have an exact minimax equilibrium.

Two-player zero-sum bilinear-payoff convex games are a natural generalization of matrix games; we will simply refer to this class as "convex games" for the sequel. This formulation was first introduced by Dresher and Karlin [1953], but convex games have received remarkably little attention in the literature considering the generality and usefulness of the framework. One of the goals of this chapter is to highlight several interesting special cases of convex games, and suggest that the class deserves much greater attention from an algorithmic perspective.

Convex games allow arbitrary convex sets X and Y in place of the probability simplices $\Delta(R)$ and $\Delta(C)$ for matrix games. A convex game is specified by a tuple (X, Y, M) where $X \subseteq \mathbb{R}^m$ and $Y \subseteq \mathbb{R}^n$ are the strategy sets for the two players, and M is a $m \times n$ payoff matrix. The first player (who we will call x) selects a action $x \in X$, the second player (called y) simultaneously chooses $y \in Y$, and the payoff from player x to player y is given by

$$V(x, y) = x^T M y.$$

The concepts of equilibria and ϵ-approximate equilibria naturally generalize to convex games. Throughout the thesis, we assume all convex action sets (X and Y in this case) are nonempty; for simplicity, we generally also assume X and Y are bounded, though this restriction can often be removed or easily enforced.[2] This insures that the games we consider have finite value (that is, $\inf_{x,y} V(x, y)$ and $\sup_{x,y} V(x, y)$ are finite).

A polyhedron is a convex set defined by a finite number of linear equality and inequality constraints. Generally, we will represent these constraints in matrix notation, for example,

$$X = \{x \in \mathbb{R}^m \mid Ax = b, \quad x \geq 0\}$$

[2]For example, if X is the set of stochastic policies for an MDP represented as state-action visitation frequencies, we can add an additional constraint to prohibit policies that loop indefinitely (making X bounded). However, often it is sufficient to simply show that these policies are never optimal (e.g., in the case of positive costs) and so have no impact on the optimization.

Matrix Games	**Convex Games**
Pure (single row)	*Pure* (an extreme point of X)
Mixed (distribution over rows)	*Implicit Mixed* (any point in X)
	Explicit Mixed (distribution over X)

Table 3.1: Strategy classes for matrix and convex games.

for a suitable matrix A and vector v. We say a convex game is polyhedral if X and Y are polyhedra. Polyhedral convex games can be solved in polynomial time via linear programming: though the term "convex game" was not used, the efficient solution technique for zero-sum EFGs presented in [Koller et al., 1994] essentially depends on a transformation from an EFG to an equivalent convex game. Koller et al. then show how to solve the resulting convex game efficiently. We review their solution method here, as in the next chapter we present a non-trivial extension of their technique which solves a generalization of EFGs.

Consider a matrix game defined by payoff matrix M, and the corresponding polyhedral convex game (X, Y, M) where $X = \Delta(R)$ and $Y = \Delta(C)$. We write $\mathsf{Cn}(Z)$ to denote the extreme points (corners) of an arbitrary polyhedron Z. Note that $\mathsf{Cn}(Z)$ is a finite set, however, in general its size may be exponential in the size of the representation of Z. However, since X is a probability simplex, we have $|\mathsf{Cn}(X)| = m$, and there is a natural mapping between $\mathsf{Cn}(X)$ and R (and similarly between $\mathsf{Cn}(Y)$ and C): each corner of X corresponds to the probability distribution that picks a particular row of M deterministically. Interior points of X correspond to mixed strategies in the matrix game.

Table (3.1) shows the different classes of strategies that we consider for matrix and convex games. Based on the analogy to matrix games, we call the strategies in $\mathsf{Cn}(X)$ the *pure strategies*, while we call strategies from the full set X (including interior points) *implicit mixed* strategies. This is natural given that if $X = \Delta(R)$, the interior points of X correspond to mixed strategies. An *explicit mixed* strategy is given by a distribution over some subset of X (possibly given as a probability density on all of X, or perhaps simply a distribution on the extreme points of X).

It would be equally reasonable to term an interior point of X a pure strategy, as it is a single strategy drawn from the set X of possible strategies. In this case, a mixed strategy (distribution over pure strategies) would be what we call an explicit mixed strategy. We use the more precise terms pure, implicit mixed, and explicit mixed to avoid these possible ambiguities.

The interpretation of these types of strategies depends on the nature of the convex game. In particular, in some games the interior points of X can be considered primitive actions (actions that can be implemented in the world directly), but in others the corners are the only primitive actions, and interior points must be interpreted as probability distributions over these. We will have more to say about the interpretation of point in X later in the section; it will turn out these distinctions do not matter from an optimization viewpoint.

Next, we will show how to solve polyhedral convex games via linear programming, and simultaneously prove the minimax theorem for polyhedral convex games. This minimax result is in fact achieved by implicit mixed strategies; we will go on to show that neither player has any incentive to play explicit mixed strategies.

3.1.1 Solution via Convex Optimization, and the Minimax Theorem

The method for solving convex games described in this section is due to Koller et al. [1994]; Von Stengel [2002] gives a more detailed presentation with examples. We consider the case where where X and Y are polyhedra, $X = \{x \mid Ax = b, x \geq 0\}$ and $Y = \{y \mid Cy = d, y \geq 0\}$, but the result can be extended to general convex sets. Suppose player x announces he will play a fixed strategy $x \in X$. Then, we can find a best response for player y by solving:

$$\begin{aligned} \max_{y} \quad & (x^T M)y \\ \text{subject to} \quad & Cy = d \\ & y \geq 0 \end{aligned}$$

The dual of this LP is

$$\begin{aligned} \min_{q} \quad & q^T d \\ \text{subject to} \quad & q^T C \geq x^T M. \end{aligned}$$

Strong duality holds,[3] so the values of the two programs are equal for all x. Thus, we can solve the game where first x picks a strategy x and then y observes this and picks a

[3]This is ensured because X and Y are bounded, nonempty polyhedra, but a direct proof (perhaps using Slater's constraint qualifications [see Boyd and Vandenberghe, 2004, Section 5.2.3]) may be necessary in the case of general convex X and Y.

response y via the following program:

$$\min_{x \in X} \max_{y \in Y} x^T M y \quad = \quad \min_{x \in X} \begin{bmatrix} \max\limits_{y} & (x^T M) y \\ \text{subject to} & Cy = d \\ & y \geq 0 \end{bmatrix}$$

and substituting the dual for the primal,

$$= \quad \min_{x \in X} \begin{bmatrix} \min\limits_{q} & q^T d \\ \text{subject to} & q^T C \geq x^T M \end{bmatrix}.$$

This then simplifies to the LP

$$\begin{aligned} \min_{x,q} \quad & q^T d \\ \text{subject to} \quad & q^T C \geq x^T M. \\ & Ax = b \\ & x \geq 0. \end{aligned} \tag{3.4}$$

By an analogous argument we can solve the game where y plays first by

$$\begin{aligned} \max_{y \in Y} \min_{x \in X} x^T M y \quad = \quad \max_{y,p} \quad & p^T b \\ \text{subject to} \quad & A^T p \leq My \\ & Cy = d \\ & y \geq 0. \end{aligned} \tag{3.5}$$

It is straightforward to verify that LP (3.5) is in fact the dual of LP (3.4), and strong duality then gives the minimax theorem for convex games:

$$\min_{x \in X} \max_{y \in Y} x^T M y = \max_{y \in Y} \min_{x \in X} x^T M y. \tag{3.6}$$

Thus, we can solve the convex game by constructing the linear program from either Equation (3.4) or Equation (3.5) and applying any standard linear programming solver. It is worth noting that the LP (3.4) can also be expressed as

$$\begin{aligned} \min_{x,\lambda} \quad & \lambda \\ \text{subject to} \quad & x \in X \\ & \lambda \geq (x^T M) y \quad \text{for all } y \in Y. \end{aligned} \tag{3.7}$$

For general Y we can solve this problem via the Ellipsoid algorithm by using an optimization over Y with the fixed linear objective function $x^T M$ to detect violations of constraints (3.7) at the current (x, λ). This is important because the optimization for fixed $x^T M$ may be efficiently solvable using a domain-specific algorithm; we will show in Chapter 5 that such best-response oracles play a central role in designing fast algorithms for convex games.

Mixed strategies are equivalent to interior points We have shown that a minimax equilibria exists when both players choose from the sets X and Y directly, without randomization. But perhaps one of the players could do better by playing an explicit mixed strategy? In fact, the answer is no. It is sufficient to show that even if a player (say, x) goes first and announces his strategy, he has no reason to announce a distribution over X (explicit mixed strategy) rather than a single $x \in X$ (implicit mixed strategy). This result is not an immediate consequence of the minimax theorem (Equation (3.6)), because that statement *assumes* both players are limited to playing only implicit mixed strategies.

Suppose x plays first and selects an explicit mixed strategy given by p and $\bar{X}$, where $\bar{X} = \{x_1, \ldots x_k\}$ is a finite subset of X and p_i is the probability she selects $x_i \in \bar{X}$ (the case where x selects a probability density over all points in X is similar). Let $\bar{x} = \sum_i p_i x_i$; this point is in X by the definition of convexity. Player y is told that player x's choice is $(p, \bar{X})$ and then he selects a best response. Thus, the expected payoff from x to y is:

$$\begin{aligned} E[V] &= \max_{y \in Y} \sum_i \Pr(\text{x plays } x_i) V(x_i, y) \\ &= \max_{y \in Y} \sum_i p_i \left(x_i^T M y \right) \\ &= \max_{y \in Y} \left(\sum_i p_i x_i \right)^T M y \\ &= \max_{y \in Y} \bar{x}^T M y. \end{aligned} \tag{3.8}$$

Note that (3.8) is exactly the payoff if x had announced $\bar{x} \in X$ instead of the explicit mixed strategy given by p. This implies neither player can get a better payoff by choosing an explicit mixed strategy. There may be many different weights p that represent the point $\bar{x}$. Thus, $\bar{x}$ may be viewed as defining an equivalence class of payoff-equivalent explicit mixed strategies. See Figure (3.1) for an example; this figure is discussed in detail in the next section.

Interpretations of the action sets How should we interpret the convex set of actions X? We consider two possibilities. First, we may view every element $x \in X$ as a primitive (playable) action that can be selected and then directly implemented in the world. For example, x might correspond to an allocation of money among m different investments, subject to some constraints.

On the other hand, consider the case of a matrix game represented as a convex game. We can solve the convex game via the linear program (3.4). Then, the optimal feasible point x will (in general) correspond to an interior point of the set X. Since X is a probability simplex, $|\mathsf{Cn}(X)| = \dim(X) = m$ and we can naturally interpret x as a probability distribution over $\mathsf{Cn}(X)$ and hence R. To "play" the matrix game, we can sample from this distribution and play that corner (row).

There are other cases where the set X is a polyhedron and only the corners $\mathsf{Cn}(X)$ are primitive actions. For example, the set of stochastic policies for an MDP can be represented as a convex set, and the corners $\mathsf{Cn}(X)$ correspond to the deterministic policies; we will discuss this example in detail in Section 3.4.3. In this case, $|\mathsf{Cn}(X)|$ can be exponentially larger[4] than $\dim(X)$, and so even explicitly representing an arbitrary distribution over $\mathsf{Cn}(X)$ may be infeasible. But, if the definition of the game requires we select an extreme point as an action, how do we interpret the interior point x? Fortunately, we have the following representation theorem (this version is from Bazaraa et al. [1990]):

Theorem 3.1.1. *Any bounded polyhedron $X \subseteq \mathbb{R}^m$ has a finite set of extreme points (corners), say $\mathsf{Cn}(X) = \{x_1, \ldots, x_k\}$. Any $x \in \mathbb{R}^m$ is a member of X if and only if there exists $p \in \mathbb{R}^k$ with $p_i \geq 0$ and $\sum_i p_i = 1$ (that is, $p \in \Delta(\mathsf{Cn}(X))$) such that $x = \sum_{i=1}^k p_i x_i$. Further, there always exists a representation such that no more than $m+1$ of the p_i coefficients are non-zero and such a representation can be found in polynomial time.*

Thus, to play a "corners only" convex game we can solve the LP formulation to find an interior point solution, generate a small-support distribution over the corners by the method of Theorem (3.1.1), and then sample from this to determine the actual primitive action to take; each of these steps takes only polynomial time. We are effectively solving an exponentially-sized matrix game (the game with rows $\mathsf{Cn}(X)$ and columns $\mathsf{Cn}(Y)$), albeit one with a very special structure: the exponential set of actions has a low-dimensional representation.

As an example, consider Figure (3.1). The convex strategy set X has corners $\mathsf{Cn}(X) = \{c_1, c_2, c_3, c_4, c_5\}$. The labeled interior point x falls inside the convex hull of $\{c_2, c_4, c_5\}$

[4]It is quite common to solve linear programs where enumerating the full set of extreme points would be impossible

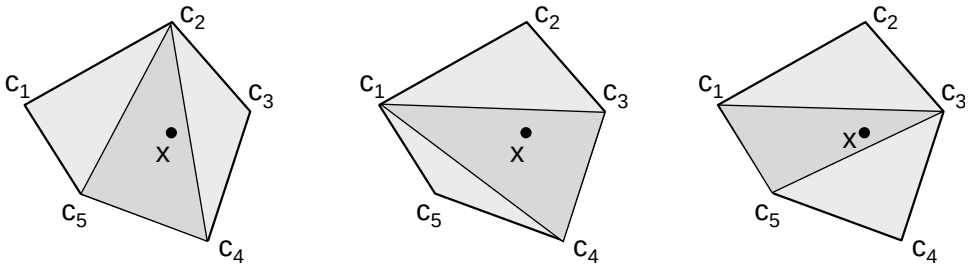


Figure 3.1: Three different explicit representations of an implicit mixed strategy x.

(left figure), which implies there exists a mixture $(p_2, p_4, p_5) \in \Delta(\{c_2, c_4, c_5\})$ such that $x = p_2c_2 + p_4c_4 + p_5c_5$. Similarly, the middle figure implies there exist $(q_1, q_3, q_4) \in \Delta(\{c_1, c_3, c_4\})$ such that $x = q_1c_1 + q_3c_3 + q_4c_4$ and the right figure shows that such a distribution r can be found on $\{c_1, c_3, c_5\}$ as well. Thus p, q, and r are different explicit mixed strategies corresponding to the single implicit mixed strategy x. Equation (3.8) shows that these explicit mixed strategies all get payoff equal to the payoff x achieves against any opponent strategy. In fact, there is an infinite equivalence class of such distributions and it is a convex set: if we view p and q as distributions over the full $\mathsf{Cn}(X)$, then for any $\theta \in [0, 1]$ the distribution $\theta p + (1 - \theta)q$ will also be an explicit mixed strategy equivalent to x. Also, observe that $m = 2$ (that is, $X \subseteq \mathbb{R}^2$) and it was possible to represent $x \in X$ with a distribution supported by only $m + 1 = 3$ points. The representation theorem says that this holds for all m.

In general, we can apply this approach for any game with a restricted set of primitive actions, $X' \subseteq X$, as long as every point $x \in X$ can be represented as a convex combination of points in X', and we have an efficient algorithm to find such a representation. This argument will extend to our generalized versions of stochastic and extensive form games as well, and so we do not worry about these two different interpretations of the action sets, as from an optimization point of view they make no difference: in both cases, finding an optimal implicit mixed strategy is sufficient.

3.1.2 Repeated Convex Games

A convex game (X, Y, M) can be played in a repeated-game setting (say by x) even if x does not know M or Y. On each round, she selects some $x \in X$, and simultaneously her adversary y selects $y \in Y$ (but x does not know Y). Then x pays y $x^T My$, where M is also unknown to x. We can play this game using online convex programming techniques and

achieve strong performance guarantees: player x will achieve at least the minimax value of the game, and can potentially do much better against a sub-optimal adversary.

If after each round x observes the effective cost vector My, then the online linear programming algorithm of Kalai and Vempala [2003] can be applied in the case of a polyhedron X, or the online convex programming algorithms of Gordon [1999] or Zinkevich [2003] can be used for general convex X. If player x only observes the amount of the payoff, $x^T My$, then a bandit-style algorithm must be used: for example the algorithm of McMahan and Blum [2004] for polyhedra (this algorithm is described in detail in Chapter 6), or the algorithm of Flaxman et al. [2005] for general convex problems. Note that for the performance guarantees of these algorithms to hold, some bounds on Y and M will be needed: a bound on the one-round maximum payoff, $\sup_{x \in X, y \in Y} |x^T My|$, is usually sufficient.

In fact, these algorithms can be used in the general sum case, as there is no dependence on player y's incentives or payoffs. In this case x is not guaranteed to approximately achieve the value of any particular equilibrium, but will at least achieve her safety value,[5] and can potentially do much better (say, if playing against a cooperative player y). No-regret algorithms can also be used in an offline fashion to approximately solve for the minimax equilibria of convex games, using techniques from Freund and Schapire [1996]. The idea is to run two no-regret algorithms against each other in the game, which often yields algorithms similar to fictitious play. We will consider this relationship in more detail in Section 5.1.

The Importance of Convex Games Before proceeding to our examples of convex games, it is worthwhile to review the importance of the class. Why is it worthwhile to show that a game falls into this class? There are both theoretical and computational advantages. By showing that a game has a convex game representation, we have also shown:

- A minimax solution exists, and it can be found by convex programming in polynomial time.
- There are a collection of fast algorithms that can be applied to the problem: for example, fictitious play and the bundle-based oracle algorithms of Chapter 5.
- There exist computationally efficient no-regret algorithms for the game.

[5]The safety value of the game for x is the value of the game when player y ignores his own payoffs and instead tries maximize player x's loss.

We now proceed to the examples.

3.2 Extensive-form Games

In this section we review extensive-form games with the aim of connecting known results to our notation and perspective. The formulation of EFGs commonly used today was original conceived by Kuhn [1953], who generalized an earlier formulation of von Neumann.

Two-player, zero-sum extensive-form games can model competitive strategic interactions that involve a sequence of decisions and random events. The game is specified via a game tree, where at each node either one of the players selects an action (corresponding to a successor of the current node) or nature picks a random successor according to a fixed probability distribution. Partial observability in the game is modeled via information sets: an information set is a subset of a player's nodes that are indistinguishable to the player. That is, a player's policy is only allowed to be a function of his observed information set, not the exact node in the game tree; necessarily, all nodes in an information set must have an equal number of successors.

We only consider games with *perfect recall*, which ensures each player's information sets form a tree. This implies that all of a player's past actions and observations can be inferred given the current information set. With perfect recall, it is sufficient to consider only behavior policies, that is, policies which specify a probability distribution over actions at each information set. For the sequel, when we write extensive-form game (or EFG) we mean a two-player, zero-sum, perfect-recall extensive-form game unless otherwise specified.

As an example, we consider the simple two-player two-card poker game shown in Figure (3.2). In this game a dealer (the initial random node) gives each player a single face-down card, either the ace or the king. Then, the players proceed to bet: first, player x can either fold (losing her ante of $\$1$), or bet an additional dollar. If she bets, player y can either fold (losing his ante of $\$1$), call (matching x's bet), or bet (raise by matching x's dollar and adding another). In the case of a call, the game ends, and if y has the best hand (a deal of (K,A)), then he wins $\$2$ from x, otherwise he loses $\$2$ to x. If he bets, then x can either fold (losing $\$2$), or call (adding another dollar to the pot to match y's raise). If x calls, then the player with the winning hands gets $\$3$ from the other.

Figure (3.2) is a representation of this game as an extensive-form game. This representation is not unique, though it is perhaps the most natural. The game tree has nodes $V = \{r_1, x_1, x_2, y_1, y_2, x_3, x_4\}$ and three information sets. Player x's set of information

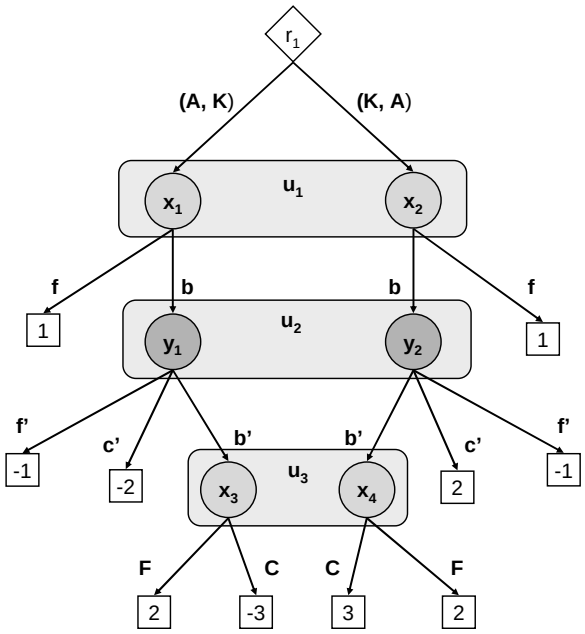
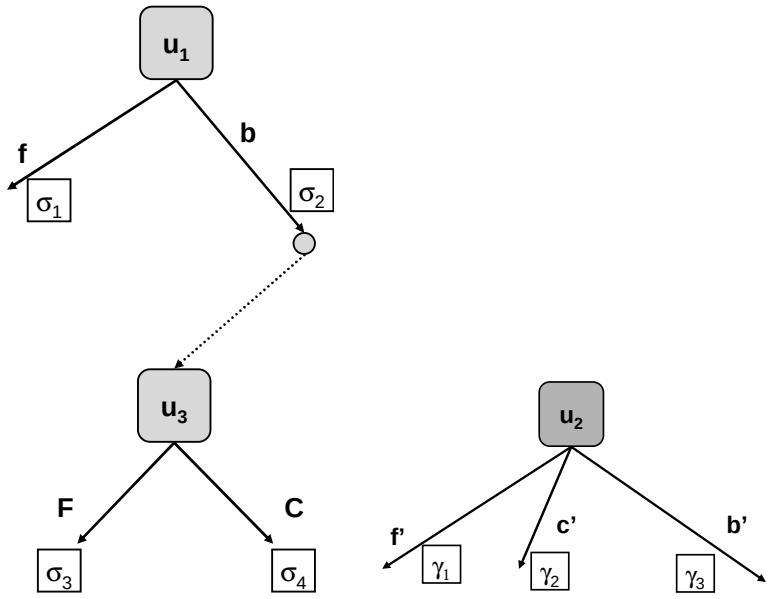


Figure 3.2: A simple poker game with a two card deck (Ace and King), represented as an extensive-form game.

sets is $U_{\mathsf{x}} = \{u_1, u_3\}$ where $u_1 = \{x_1, x_2\}$ and $u_3 = \{x_3, x_4\}$; player y's set of information sets is $U_{\mathsf{y}} = \{u_2\}$, where $u_2 = \{y_1, y_2\}$. Information sets are shown in the figure by including the constituent nodes in a rounded rectangle. The set of actions (labels on outgoing edges, also called choices) available to x at u_1 is $A(u_1) = \{b, f\}$, and similarly $A(u_2) = \{f', c', b'\}$ and $A(u_3) = \{F, C\}$. The first node r_1 is a random node, with a fixed probability distribution $(0.5, 0.5)$ over the two possible deals. The leaves indicate the payoffs from x to y.

We can, in general, model two-player poker games in this way. Each node in the game tree encodes a fixed setting of all the cards dealt so far as well as the betting history. But, in general there will be some cards that a player cannot see. At a point in the game where a player must select an action (usually bet, fold or call), the nodes corresponding to the different possible settings for the unobserved cards are grouped into an information set. For example, in our very simple game the players do not observe either of the cards, and so they cannot tell which of the two possible deals occurred.

The key results for extensive-form games that pertain to our work are the fact that extensive-form games can be transformed to convex games, and that computing best re-



(a) Player x's Sequence Tree (b) Player y's Sequence Tree

Figure 3.3: Sequence trees for the example poker game.

sponses for an extensive-form game is very fast.

First, we consider the representation of an EFG as a convex game. Since a behavior strategy can be represented as the Cartesian product of probability simplices, the set of such strategies is convex. Unfortunately, the payoff for a pair of strategies represented this way is not bilinear (it can not be written as $x^T My$), and so the the EFG cannot be written as a convex game using these strategy sets. Instead, we turn to the sequence weight representation of strategies.

For a given EFG, we construct a convex game (X, Y, M) where the strategy set X has one dimension for each possible *sequence* of (information set, action) pairs for player x (and analogously for Y). For the example in Figure (3.2), the possible sequences σ_i for player x and γ_i for player y are:

$$\begin{aligned}
\sigma_0 &= \emptyset & \gamma_0 &= \emptyset \\
\sigma_1 &= (f) & \gamma_1 &= (f') \\
\sigma_2 &= (b) & \gamma_2 &= (b') \\
\sigma_3 &= (b, F) & \gamma_3 &= (c') \\
\sigma_4 &= (b, C). &
\end{aligned}$$

We assume each possible action label is associated with a single information set, so we can omit the information sets from the sequences: for example (b, C) uniquely corresponds to (u_1, b, u_3, C). Perfect recall ensures sequences are in one-to-one correspondence with the action labels, so in fact each sequence is uniquely identified by its last action. Thus, we can use action labels and sequences interchangeably. The perfect recall assumption also means that the information sets of each player (and hence, the sequences) form a tree. That is, any non-empty sequence has a unique predecessor sequence.

The sequence trees for the example EFG of Figure (3.2) are given in Figure (3.3). The information sets are shown as large rounded rectangles, and each edge out of an information set corresponds to a particular action label and sequence. The small round node indicates a junction where the next information set reached is determined by choices of the adversary and nature (shown as a dotted line); this node is trivial in the example tree as it has a single successor. For an arbitrary EFG, junction nodes and information-set nodes alternate and junction nodes may have many successors.

We now define a strategy representation that admits a bilinear objective function. The strategy polyhedron X has one dimension for each sequence for x; elements of X are called *sequence weight vectors*. The *sequence weight* x_i associated with the sequence σ_i can be interpreted as the probability that the sequence σ_i occurs under the policy x represents, conditioned on the other player and nature deterministically taking actions compatible with this sequence.

There is a natural mapping between behavior policies β for player x and sequence-weight vectors $x \in X$. If β is behavior policy, the corresponding x has weight x_i on sequence σ_i equal to product of the probabilities that β places on each action in σ_i. As an example, suppose β is the policy for x that folds 1/3 of the time at u_1, and always calls at u_2. The corresponding sequence-weight vector is

$$x = \left[1, \frac{1}{3}, \frac{2}{3}, 0, \frac{2}{3}\right].$$

For any sequence-weight vector $x \in X$, we define an associated behavior policy β_x. For example, given the vector x above, the probability with which β_x bets at u_1 is

$$\beta_x(u_1, \mathsf{b}) = \frac{x_2}{x_1 + x_2} = \frac{2/3}{1/3 + 2/3} = 2/3$$

(note the zero indexing of x). Similarly, the probability β_x calls at u_3 is $x_4/(x_3 + x_4)$, and so on. In general $\beta_x(u, a)$ is defined by

$$\beta_x(u, a) = \frac{x_a}{\sum_{a' \in A(u)} x_{a'}}.$$

The policy β_x will be undefined at information sets it never reaches, but this is fine. For formal proofs that these properties hold consult Koller and Megiddo [1992].

The set X of all sequence weight vectors that correspond to valid behavior policies is a polyhedron. For the example game, the constraints are:

$$\begin{aligned}
\sigma_0 &= 1 \\
\sigma_0 &= \sigma_1 + \sigma_2 \\
\sigma_2 &= \sigma_3 + \sigma_4 \\
\sigma_i &\geq 0 \quad \forall i \in \{0, \ldots, 4\}.
\end{aligned}$$

This technique generalizes to all extensive-form games, and X can always be represented as a polyhedron with one equality constraint for each information set along with non-negativity constraints [Koller and Megiddo, 1992].

The payoff matrix M corresponding to the sequence-weight strategy polyhedra X and Y has one entry for each pair of sequences (σ, γ). Let L be the set of leaves for the extensive-form game, and let $m : L \to \mathbb{R}$ give the payoff at each leaf. For $\ell \in L$, define $I(\ell, \sigma, \gamma)$ to be 1 when the set of action labels on the path to ℓ exactly equals the set of action labels in $\sigma \cup \gamma$, and 0 otherwise.[6] Let $\text{rand}(\ell)$ be the product of the probabilities on all random edges on the path to ℓ. Then, the value $M(\sigma_{\mathsf{x}}, \sigma_{\mathsf{y}})$ is

$$\sum_{\ell \in L} I(\ell, \sigma, \gamma)\text{rand}(\ell)m(\ell).$$

Two sequences σ and γ are inconsistent if no path in the game tree has exactly those actions: for example, σ_1 and γ_1 are inconsistent because it is impossible for both players to fold. For such a sequence pair, $M(\sigma, \gamma) = 0$, and so in general M may have considerable sparsity. In order to show that the convex game (X, Y, M) is equivalent to the EFG, one must show for all $x \in X$ and $y \in Y$ that

$$V(\beta_x, \beta_y) = x^T M y.$$

We do not give a formal proof here; however, in the next chapter we give the corresponding proof for convex extensive-form games.

Fast best-response algorithms for extensive-form games use dynamic programming on the response player's sequence tree (Figure (3.3)). If we fix an opponent strategy y, the vector $c = My$ assigns a cost $c(\sigma)$ to each player x sequence or equivalently, action.

[6]The assumption that the action labels from different information sets are distinct is critical to this definition.

A value function on information sets can then be computed via dynamic programming, working from the leaves backward to the root. For player x (the min player), at each leaf information set a greedy action with respect to the costs $c(\sigma)$ on the leaf sequences is selected, and the corresponding value is assigned to the information set. Internal junction nodes are given value equal to the sum of the values of their predecessor information set nodes. Internal information-set nodes have value equal to the minimum over action labels a of the immediate cost $c(\sigma_a)$ plus the value of the successor junction node reached after a. Any behavior policy that is greedy with respect to these values is a best response to y. Computing these values and reading off a best response takes time $\mathcal{O}(m)$, since m equals the number of edges in the tree. Note that computing $c = My$ is much more expensive, in general $\mathcal{O}(mn)$. For a more detailed treatment of the best-response problem in extensive-form games, see [Koller and Megiddo, 1992]; for the transformation of extensive-form games to convex games see [Koller et al., 1994].

3.3 Optimal Oblivious Routing

A significant amount of work has been done on the problem of computing an oblivious routing for a graph, in both the exact and approximate cases [see Azar et al., 2004, Bienkowski et al., 2003, and references therein]. However, the observation that optimal oblivious routing can be expressed as a convex game is new to this thesis.

Expressing the problem as a convex game provides access to a large array of theoretical results and efficient algorithms. For example, this representation immediately shows that there is a polynomial-sized linear program for optimal oblivious routing. This fact did not appear in the literature until [Applegate and Cohen, 2003]. Further, as Chapter 5 of this thesis will demonstrate, efficient algorithms exist for convex games that can be much faster than standard LP codes. While the algorithm of Azar et al. [2003] is impractical for large real-world problems, the application of our fast convex game solvers to this problem might make generating optimal solutions to very large oblivious routing problems possible.

The polyhedral convex game representation opens up new possibilities for the online (repeated-play) version of the problem as well. For example, the algorithm of Bansal et al. [2003] requires a call to a projection oracle on each iteration in order to perform gradient ascent. Finding such a projection requires solving a semi-definite program. However, the polyhedral convex game representation implies that the online algorithm of Kalai and Vempala [2002] can be used to get similar bounds, while requiring only the solution of a linear program[7] on each iteration. The convex game representation also shows that many

[7] In fact, the LP represents a standard multi-commodity flow problem.

variations on the standard problem can be solved in polynomial time, and automates the process of producing compact LP representations for these variants.

We state the problem using the terminology of Azar et al. [2003]. The optimal oblivious routing game is specified by a directed graph $G = (V, E)$ with edge capacities $c : E \to \mathbb{R}_+$. The routing player selects a *routing* for traffic in G, represented by a one-unit flow f_{ij} from each vertex $i \in V$ to each vertex $j \neq i$. The term oblivious refers to the fact that the demands are not known at the time this routing is computed. The variable $f_{ij}(e)$ specifies the volume of flow on edge e associated with routing 1 unit of volume from i to j using f_{ij}. Thus, routing demand d_{ij} from i to j using f_{ij} results in volume $d_{ij} f_{ij}(e)$ on each edge e. The set F of all valid routings is convex and specified by a number of constraints polynomial in the size of the input graph.

Given a set of demands $d = \{d_{ij} \in \mathbb{R} \mid i, j \in V, i \neq j, d_{ij} \geq 0\}$, the congestion on an edge e under routing f is given by

$$\mathsf{econg}(e, f, d) = \frac{\sum_{i,j} d_{ij} f_{ij}(e)}{c(e)}.$$

The congestion of the routing is then

$$\mathsf{cong}(f, d) = \max_{e \in E} \ \mathsf{econg}(e, f, d).$$

For a fixed set of demands d, there exists a minimal congestion routing, with congestion

$$\mathsf{opt}(d) = \min_{f \in F} \ \mathsf{cong}(f, d)$$

which can be found via linear programming.

If $|E| = n$, then there are $n(n-1)$ flows that define f, and each one has an associated demand. Let $D = \{d \in \mathbb{R}^{n(n-1)} \mid d_{ij} \geq 0\}$ be the set of all possible demands. The adversary in the optimal oblivious routing game selects a demand $d \in D$, and the overall game is then

$$\min_{f \in F} \max_{d \in D} \ \frac{\mathsf{cong}(f, d)}{\mathsf{opt}(d)}.$$

While F and D are convex sets, the objective function is not bilinear. Instead, we reformulate the objective as follows. Scaling d by a positive constant does not change the objective, as it scales both the numerator and denominator equally. Thus, it is equivalent to optimize over the set $D_1 = \{d \in D \mid \mathsf{opt}(d) \leq 1\}$. Using this observation and the definition of cong, we can rewrite the game as

$$\min_{f \in F} \max_{d \in D_1} \max_{e \in E} \ \mathsf{econg}(e, f, d).$$

Now we have a game with a multi-linear objective function (because the capacities $c(e)$ are a constant given by the problem specification). The set D_1 is in fact a polyhedron, as shown by Equations (8) and (9) of [Azar et al., 2003]. The key idea is to think of the adversary as picking an arbitrary multi-commodity flow with congestion at most 1; the corresponding demands d are a linear function of the multi-commodity flow. But, the objective is not bilinear if we think of the same player (the adversary) choosing both d and e. Since we can imagine the $f \in F$ as fixed when the adversary chooses his action, we can allow the adversary to pick an arbitrary distribution $\mu \in \Delta(E)$, giving the game

$$\min_{f \in F} \max_{\mu \in \Delta(E)} \max_{d \in D_1,} \sum_{e \in E} \mu(e) \mathsf{econg}(e, f, d). \tag{3.9}$$

If $\mu \in \Delta(E)$ was a fixed parameter of the problem, for example, if we wished to minimize the average edge congestion rather than the maximum, then we would be done. However, the objective as given above is non-linear for the max player: to form a convex game, we need the value to be a linear function of a fixed adversary strategy f.

We sketch a way that this can be accomplished; the details of the technique are actually closely related to the convex extensive-form game formulation introduced in the next section. Let $D_1^c = \{(\alpha d, \alpha) \mid d \in D_1, \alpha \geq 0\}$. We call this set the cone extension of D_1; since D_1 is a polyhedron, so is D_1^c (see Appendix B). We define the adversary's strategy polyhedron $\tilde{D}_1^c$ by the variables $d_{ij}^c(e)$ and $\mu(e)$ via the following constraints:

$$\begin{aligned} \mu &\in \Delta(E) \\ (d^c(e), \alpha(e)) &\in D_1^c \qquad \forall e \in E \\ \mu(e) &= \alpha(e) \qquad \forall e \in E \end{aligned}$$

It can then be shown that the optimal oblivious routing game (Equation 3.9) is equivalent to the convex game:

$$\min_{f \in F} \max_{d^c \in \tilde{D}_1^c} \sum_{e \in E} \frac{1}{c(e)} \sum_{i,j} d_{ij}^c(e) f_{ij}(e). \tag{3.10}$$

The equivalence follows from the fact that for all e, $d_{ij}^c(e) = \mu(e) d_{ij}(e)$ for some demands $d(e) \in D_1$. We are thus allowing the max player to pick a different set of demands for each edge, but since the max is achieved at a single edge this does not change the value of the game. For a fixed f, the best response problem is exactly the problem solved by the separation oracle of Azar et al. [2003], which solves an independent problem for each edge $e \in E$. It is worth noting that the linear program due to Applegate and Cohen [2003] is different than the one obtained by applying Equation (3.4) to the convex game of Equation (3.10). The relative merits of the two different LP formulations have not been investigated as of yet.

The convex game representation of Equation (3.10) implies that many variations on the basic problem are also solvable in polynomial time. For example, we can introduce additional constraints on μ, replacing $\Delta(E)$ with an arbitrary convex subset of $\Delta(E)$. Similarly, we could further constrain D_1 to only allow the adversary to pick demands that are convex combinations of demands that have been observed in the past. Since these transformations preserve the convexity of the adversary's strategy set, the game remains convex.

3.4 MDPs with Adversary-controlled Costs

We investigate methods for planning in a Markov Decision Process where the cost function is chosen by an adversary after a policy for the MDP has been chosen by the planning player. First we consider the case where the opponent is restricted to a finite set of cost functions, and then we consider the case of an arbitrary convex set of cost vector.[8] The later situation includes games where the cost function in player one's MDP is a linear function of the state-action frequency representation of the policy chosen by player two in another MDP. This work originally appeared in [McMahan et al., 2003, McMahan and Gordon, 2003]. This section provides all the necessary background to read Section (5.2); the reader with immediate interest in algorithmic approaches should feel free to consult that section after completing the present one.

As a running example, we consider a robot path planning problem where costs are influenced by sensors that an adversary places in the environment. We formulate the problem as a zero-sum matrix game where rows correspond to deterministic policies for the planning player and columns correspond to cost vectors the adversary can select. This exponentially large matrix game has a concise representation as a convex game; we explore that representation and other details of the problem formulation in this section. For a fixed cost vector, fast algorithms (such as value iteration) are available for solving MDPs. In Chapter 5, we develop algorithms that use these fast best response oracles, and show that for our path planning problem they can be several orders of magnitude faster than direct solution of the linear programming formulation.

[8] Since we consider only finite state and action spaces, we use the terms *cost function* and *cost vector* interchangeably.

3.4.1 Introduction and Motivation

Imagine a robot in a known (previously mapped) environment which must navigate to a goal location. We wish to choose a path for the robot that will avoid detection by an adversary. This adversary has some number of fixed sensors (perhaps surveillance cameras or stationary robots) that he will position in order to detect our robot. These sensors are undetectable by our robot, so it cannot discover their locations and change its behavior accordingly. What path should the robot follow to minimize the time it is visible to the sensors? Or, from the opponent's point of view, what are the optimal locations for the sensors?

We assume that we know the sensors' capabilities. That is, given a sensor position we can calculate what portion of the world it can observe. So, if we know where the opponent has placed sensors, we can compute a cost vector for the MDP: each entry assigns a constant observation cost to each world state observed by a sensor. We also add a small movement cost, so that the robot prefers shorter paths to the goal. Given this fixed cost vector, we can apply efficient planning algorithms (value iteration in stochastic environments, A* search in deterministic environments) to find a path for the robot that minimizes the total observation time. Of course, in the full game we don't know the sensor locations; instead we have a set of possible cost vectors, one for each allowable sensor configuration, and we must minimize the expected cost under the worst-case distribution over cost vectors. In this section, we discuss different ways to model the general problem we have described, and discuss the variation we solve. In particular, we show that the problem can be formulated as a convex game.

Our algorithms are practical for problems of realistic size, and we have used our implementation to find plans for robots playing laser tag as part of a larger project [Rosencrantz et al., 2003]. Figure (3.4) shows the optimal solutions for both players for a particular instance of the problem. The map is of Rangos Hall at Carnegie Mellon University, with obstacles corresponding to overturned tables and boxes placed to create an interesting environment for laser tag experiments. The optimal strategy for the planner is a distribution over paths from the start (s) to one of the goals (g), shown in (3.4)A; this corresponds to a mixed strategy in the matrix game, that is, a distribution over the rows of the game matrix. The optimal strategy for the opponent is a distribution over sensor placements, or equivalently a distribution over the columns of the game matrix. This figure is discussed in detail in Section (3.4.4).

3.4.2 Model Formulation

There are a number of ways we could model our planning problem. The model we choose, which we call the *no observation, single play formulation*, corresponds to the assumptions outlined above. Initially, we restrict the opponent to choosing a cost vector from a finite though possibly large set, but later we relax this to allow arbitrary convex sets of possible costs. The planning agent knows this set as well as the dynamics of the MDP, and so constructs a policy that optimizes worst-case expected cost given these allowed cost vectors. Let Π_D be the set of proper deterministic policies available to the planning agent, let $K = \{c_1, \ldots, c_k\}$ be the set of cost vectors available to the adversary, and let $V(\pi, c)$ be the value[9] of policy $\pi \in \Pi_D$ under cost vector $c \in K$. The goal is to solve the matrix game with one row for each $\pi \in \Pi_D$ and one column for each $c \in K$; the entry in the payoff matrix for row π and column c is then $V(\pi, c)$. Equivalently, we wish to solve the optimization

$$\min_{p \in \Delta(\Pi_D)} \max_{q \in \Delta(K)} E_{\pi \sim p, c \sim q}[V(\pi, c)], \tag{3.11}$$

along with the distributions p and q that achieve this value. We now discuss the assumptions behind this formulation of the problem in more detail, and examine several other possible formulations.

Our most limiting assumption is that our planning agent cannot observe the adversary's effect on the cost vector. In our example domain, the robot incurs fixed, observable costs for moving, running into objects, etc.; however, it cannot determine when it is being watched and so it cannot determine the cost vector selected by the adversary. This is a reasonable assumption for some domains, but not others. If the assumption does not hold, our algorithms will produce suboptimal policies: for example, we would not be able to plan to check whether a path was being watched before following it.

The no-observation assumption, while sometimes unrealistic, is what allows us to develop efficient algorithms. Without this assumption, the planning problem in general becomes a partially-observable Markov decision process even when we know the distribution over cost vectors the opponent has chosen: the unknown cost vector is the hidden state and the costs incurred are observations. POMDPs are known to be difficult to even approximately solve [Kaelbling et al., 1996]; on the other hand, the planning problem without observations admits polynomial-time algorithms, as we will show. Later, we will use a generalization of extensive-form games (introduced in Chapter 4) to relax the no-observation assumption somewhat. We will be able to model problems where the planning player can

[9] That is, the expected value of the start state under policy π. This can be found by solving a set of linear equations; see Section 2.2

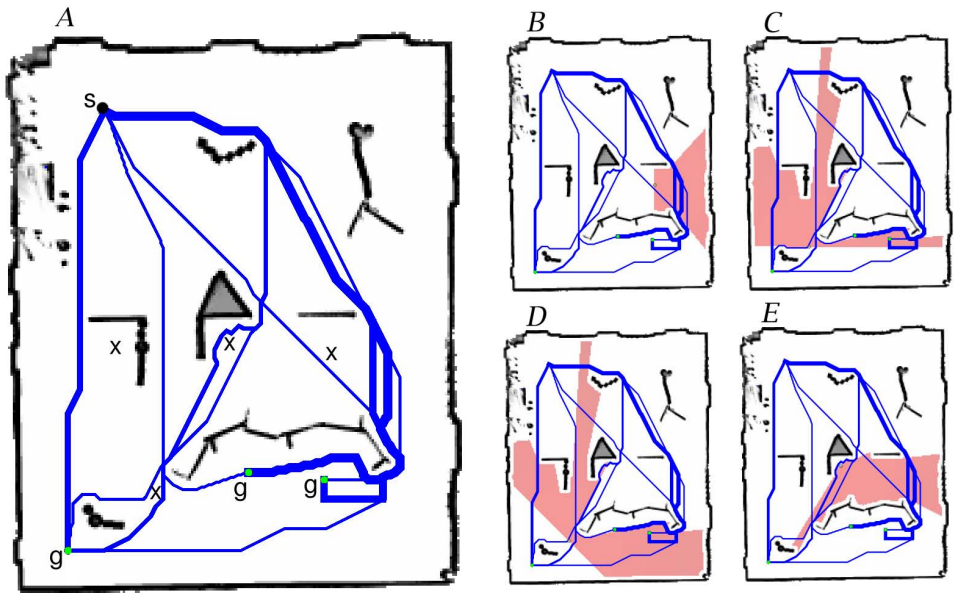


Figure 3.4: Planning in a robot laser tag environment. Part A: A mixture of optimal trajectories for a robot traveling from start location (s) to one of three goals (g). The opponent can put a sensor in one of 4 locations (x), facing one of 8 directions. The widths of the trajectories correspond to the probability that the robot takes the given path. Parts B,C,D,E: The optimal opponent strategy randomizes among the sensor placements that produce these four fields of view.

make some limited observations (perhaps only from a constant number of states, or on only at a constant number of fixed times) and still maintain computational tractability.

In addition to the POMDP formulation, our problem can also be framed in an online setting where the MDP must be solved multiple times for different cost vectors. The planning agent must pick a policy for the nth game based on the cost vectors it has seen in the first $n-1$ games. The goal is to do well in total cost, compared to the best fixed policy against the opponent's sequence of selected cost vectors. To obtain tractable algorithms we still make the no-observation assumption, but it is not necessary to assume the opponent chooses cost vectors from a finite or convex set. When this formulation is applied to shortest path problems on graphs, it is the online shortest path problem for which some efficient algorithms are already known [Takimoto and Warmuth, 2002]. Once we have shown the transformation to an equivalent convex game, the algorithms for the repeated setting discussed at the end of Section 3.1 immediately apply.

It is worth noting the relationship between our problem and stochastic games. Our setting is more general in some ways and less general in others: we allow hidden state (the cost function), but stochastic games allow players to make a sequence of interdependent moves while we require both players to select their policies simultaneously at the outset. Our work also differs from that of Bagnell et al. [2001], in that they consider uncertainty about the dynamics model, while we consider uncertainty about the cost function.

In general, the no-observation assumption is applicable in two cases: when observations are actually impossible, and when observations are possible, but once they have been made there is nothing to be done. The way we initially phrased our robot path-planning problem, it falls in the first case: the sensors cannot be detected. On the other hand, if we can detect a sensor but have already lost the game once we detect it, the problem falls in the second case.

So far we have imagined an adversary selecting one cost vector from a set of cost vectors; however, our formulation applies to the case where the actual cost is given by the highest cost of the chosen policy with respect to any of the cost vectors. For example, suppose there is a competition to control a robot performing an industrial welding task. In the first round the robots will be evaluated by three human judges, each of which has the ability to remove a robot from consideration. It is known that one judge will prefer faster robots, another will be more concerned with the robots' power consumption, and another with the precision with which the task is performed. If the task is formulated as an MDP, then each judge's preference can be turned into a cost vector, and our algorithm will find the policy that maximizes the lowest score given by any of the three judges. The policy calculated will be optimal if all three judges evaluate the policy and then assign the lowest of their scores, or if an adversary picks a distribution from which only one judge will be chosen.

We now proceed with some background on MDPs and linear programming, and then present the transformation to a convex game.

3.4.3 Solving MDPs with Linear Programming

In this section, we review techniques for solving MDPs via linear programming, as this background will lead directly to a linear programming formulation of the adversarial MDP model as well as provide the tools for transforming the problem into an equivalent convex game.

Consider an MDP M with a state set S and action set A. The dynamics for the MDP are specified for all $s, s' \in S$ and $a \in A$ by $P^a_{ss'}$, the probability of moving to state s'

if action a is taken from state s. In order to express problems regarding MDPs as linear programs, it is useful to define a matrix E as follows: E has one row for every state-action pair and one column per state. The entry for row (s, a) and column s' contains $P^a_{ss'}$ for all $s \neq s'$, and $P^a_{ss} - 1$ for $s = s'$. A cost function for the MDP can be represented as a vector c that contains one entry for each state-action pair (s, a) indicating the cost of taking action a in state s. A stochastic policy for an MDP is a mapping $\pi : S \times A \to [0, 1]$, so that $\pi(s, a)$ gives the probability an agent will take action a in state s. Thus, for all s we must have $\sum_{a \in A} \pi(s, a) = 1$. A deterministic policy is one that puts all its probability on a single action for each state, so that it can be represented by $\pi : S \to A$. The Markov assumption implies that we do not need to consider history[10] dependent policies; the policies we consider are stationary, in that they depend only on the current state. For an MDP with a fixed cost function c there is always an optimal deterministic policy, and so stochastic policies play a lesser role. In our adversarial formulation, however, optimal policies are typically stochastic.

We are primarily concerned with undiscounted shortest path optimality: that is, all states have at least one finite-length path to a zero-cost absorbing state, and so undiscounted costs can be used. Our results can be adapted to discounted infinite horizon problems by multiplying all the probabilities $P^a_{ss'}$ by a discount factor γ when the matrix E is formed. The results can also be extended to an average reward model, but this requires slightly more complicated changes to the linear programs introduced below.

There are two natural representations of a policy for a MDP, one in terms of frequencies and another in terms of total costs or values. Each arises naturally from a different linear programming formulation of the MDP problem. For any policy π we can compute a value function, $v_\pi : S \to \mathbb{R}$, that associates with each state the total cost $v_\pi(s)$ that an agent will incur if it follows π from s for the rest of time. If π is optimal then the policy achieved by acting greedily with respect to v_π is optimal. Thus, value functions can represent deterministic greedy policies, but not arbitrary stochastic policies; hence, to find an optimal policy for our adversarial problem we will need a different policy representation.

The optimal value function for an MDP with cost vector c and fixed start state distribution $\mu_s \in \Delta(S)$ can be found by solving the following linear program:

$$\begin{aligned} \max_{v} \quad & \mu_s \cdot v \\ \text{subject to} \quad & Ev + c \geq 0. \end{aligned} \tag{3.12}$$

[10]We assume the standard definition of the history, where it contains only states and actions. If costs incurred appear in the history then our formulation does not apply, as we are in the POMDP case.

The set of constraints $Ev + c \geq 0$ is equivalent to the statement that

$$v(s) \leq c(s,a) + \sum_{s' \in S} P^a_{ss'} v(s')$$

for all $s \in S$ and $a \in A$, corresponding to the Bellman equations (see Section 2.2).

Fixing an arbitrary stochastic policy π and start state distribution μ_s uniquely determines a set of state-action visitation frequencies $f \in \mathbb{R}^{|S||A|}$, where $f(s,a)$ gives the expected number of times action a is taken from state s before the goal is reached, given the the initial state is drawn from μ_s and the agent follows π. We write f_π when we wish to show the dependence on π; the dependence on μ_s is implicit. The dual of (3.12) is the linear program whose feasible region is the set of state-action visitation frequency vectors (that correspond to some stochastic policy), and is given by

$$\begin{aligned} \min_f \quad & f \cdot c \\ \text{subject to} \quad & E^T f + \mu_s = 0 \\ & f \geq 0. \end{aligned} \tag{3.13}$$

The constraints $E^T f + \mu_s = 0$ require that the sum of all the frequencies into a state x equal the sum of all the frequencies out of x. The objective $f \cdot c$ represents the expected value of the start state drawn from μ_s under the policy π which corresponds to f. For any cost vector c we can compute the value of π as

$$V(\pi, c) = f_\pi \cdot c. \tag{3.14}$$

3.4.4 Representation as a Convex Game

Our game will have the convex strategy set

$$F = \{f \in \mathbb{R}^{|S||A|} \mid E^T f + \mu_s = 0,\ f \geq 0\} \tag{3.15}$$

for the planning player. As mentioned in the previous section, there is a correspondence between the set F and the set of stochastic policies. In fact, this correspondence is very similar to the correspondence between behavioral policies and sequence-weight vectors in extensive-form games. As in that case, a state-action visitation vector f will not specify the distribution over actions to be taken at states never reached by the corresponding policy.

It can be shown that $\mathsf{Cn}(F) = \{f_\pi \mid \pi \in \Pi_D\}$, that is, corners of F are deterministic policies. Thus, a mixed strategy $p \in \Delta(\Pi_D)$ for the row player in the matrix game

of Equation (3.11) corresponds exactly to an explicit mixed strategy over F, and hence is equivalent to the implicit mixed strategy $f = \sum_{\pi \in \Pi_D} f_\pi p(\pi)$. In other words, every stochastic policy (when represented as a state-action visitation frequency vector) can be represented as a convex combination of deterministic policies, and every convex combination of deterministic policies corresponds to some stochastic policy. Puterman [1994, Sec. 6.9] gives a detailed proof of this fact; it can also be proved as a consequence of Theorem (3.1.1).

We denote the convex hull of a finite set such as K by

$$\mathsf{H}(K) = \left\{ \sum_{c \in K} q(c)\, c \mid q \in \Delta(K) \right\}.$$

A mixed strategy $q \in \Delta(K)$ for the column player is equivalent in expectation to the cost vector $\sum_{c \in K} q(c)c$ in the convex set $\mathsf{H}(K)$. Given a state-action frequency vector $f \in F$ and an implicit mixed cost vector $c \in \mathsf{H}(K)$, the value of the game is given by $c \cdot f$ by Equation (3.14). Thus, we can reduce our exponentially large matrix game (given by Equation (3.11) to the convex game $(F, \mathsf{H}(K), I)$ where I is the identity matrix. We can also define the equivalent convex game $(F, \Delta(K), M_K)$, where M_K is the matrix with columns $c_1, \ldots, c_k$. The two representations of the convex game correspond to writing the objective function as $f^T I(M_K q)$ versus $f^T M_K q$.

While we could conclude this section here, it is instructive to directly construct the linear programs for the convex game $(F, \Delta(K), M_K)$. We do so by extending (3.13) with another variable z which represents the maximum cost of the policy f over all possible opponent cost vectors:

$$\begin{aligned} \min_{z,f} \quad & z \\ \text{subject to} \quad & E^T f + \mu_s = 0 \\ & \mathbf{1} \cdot z + M_K^T f \leq 0 \\ & f \geq 0. \end{aligned} \tag{3.16}$$

The primal variables f of (3.16) give an optimal implicit mixed strategy for the planning player. Taking the dual of (3.16), we have

$$\begin{aligned} \max_{v,q} \quad & v \cdot \mu_s \\ \text{subject to} \quad & Ev + M_K q \geq 0 \\ & \mathbf{1} \cdot q = 1 \\ & q \geq 0, \end{aligned} \tag{3.17}$$

where q gives the optimal mixture of costs for the adversary, and v is the value function when playing against this distribution. The value function v induces a deterministic policy that gives a best response if the opponent chooses the distribution q over cost vectors, but in general this pair of strategies will not be a minimax equilibrium. However, the pair (f, q) pair will be. It is straightforward to verify that Equation (3.16) is a special case of Equation (3.4) and that Equation (3.17) is a special case of Equation (3.5).

Figure (3.4) shows a solution to the robot path planning problem formulated in this way. The left portion, A, shows the minimax optimal strategy for the planner. The sample problem has deterministic dynamics, so a deterministic policy from Π_D is simply a start to goal path (note there are 3 goal states in the example domain). The optimal stochastic policy is shown as a distribution over deterministic paths; the width of the path line indicates the relative probability with which it is selected. Each of the four right-hand panels (B, C, D, and E) corresponds to a cost vector from K, shown as the field of view of the sensor placement. These four are the most likely cost vectors selected by the opponent's minimax optimal policy; they are chosen with probability 0.18, 0.42, 0.11, and 0.28 respectively. The remainder of the probability mass is on other sensor placements.

Unfortunately, many interesting MDPs are too large to allow efficient solution via linear programming, and so neither of the above linear programs may be practical; however, for a fixed cost function value iteration or other MDP algorithms can solve such large problems. In Chapter 5 we develop techniques that allows us to use an arbitrary MDP solution technique as a best-response oracle in an iterative algorithm for solving (3.16).

3.4.5 Cost-paired MDP Games

In this section we described a generalization to the adversarial-cost MDPs of the previous section. In cost-paired MDP games, *both* players select a policy in a separate MDP, but the costs associated with a policy in one of the MDPs depend (via a linear function) on the policy selected by the other player. These cost-paired MDPs games represent an interesting and computationally tractable class of adversarial planning problems; they can be formulated as polynomial-sized convex games.

Revisiting the Sensor-placement Game

In the previous section we considered a fixed, finite set of possible sensor configurations that determined costs; the techniques introduced in this section let us consider a mobile sensor platform that must decide on an observation strategy represented as a policy in an

MDP. The observer's rewards depend on its own policy as well as on the motion of the entity which it is trying to observe. Suppose that the output from the sensor cannot be processed in real time due to latency, insufficient on-board computation, or the need for human expert analysis; suppose also that the entity being observed is aware that it may be observed, but cannot detect when observations happen.

One natural instance of this problem is scientific data collection from a satellite or planetary rover. We want to maximize the amount of time that the sensor spends observing a particular natural phenomenon. Communication delays prevent the sensor from altering its actions based on the data collected so far. Nature is oblivious to the sensor's actions, but we treat her as an adversary in order to compute a robust plan. We need not model nature as *purely* adversarial: to the extent we have good estimates of the probabilities that govern the behavior of nature, we can embed these into nature's MDP. In this way the only degrees of freedom we leave the adversary correspond to uncertainty for which we have no good statistical model. Note that if we play this game multiple times, then we can use online learning (see Chapter 6) to capitalize on the fact that nature may not be purely adversarial even if we lack any probabilistic model.

Problem Model

We have a two-player, zero-sum game, with players x and y as usual. Let $\mathcal{M}_\mathsf{x} = (S^\mathsf{x}, A^\mathsf{x}, P^\mathsf{x}, \mu_s^\mathsf{x})$ and $\mathcal{M}_\mathsf{y} = (S^\mathsf{y}, A^\mathsf{y}, P^\mathsf{y}, \mu_s^\mathsf{y})$ be MDPs, one for each player. For each MDP, S is a finite set of states, A is a finite set of actions, $P : (S \times A) \to \Delta(S)$ is a transition function, and μ_s is a distribution over start states. Each MDP would normally have a vector of state-action costs, but we leave the costs unspecified for now; costs in $\mathcal{M}_\mathsf{x}$ will depend on the policy in y chooses for $\mathcal{M}_\mathsf{y}$, and vice versa. Let $m = |S^\mathsf{x}||A^\mathsf{x}|$ and $n = |S^\mathsf{y}||A^\mathsf{y}|$. Let Π_D^x (Π_{ND}^x) be the set of deterministic (stochastic) policies for $\mathcal{M}_\mathsf{x}$, and define Π_D^y and Π_{ND}^y analogously for $\mathcal{M}_\mathsf{y}$. We rule out policies with infinite visitation frequencies; we can do so either by introducing a discount factor (in which case all discounted frequencies will be finite) or by assuming positive edge costs for X, negative costs for Y, and no "orphan" states (in which case the agents will never choose nonterminating policies). As we observed in the previous section, the set Π_{ND}^x can be represented via a convex set X of state-action visitation frequencies using Equation (3.15), and similarly Π_{ND}^y can be represented by Y. These will be our strategy sets for the convex game; it remains to define the payoffs.

The cost vector for $\mathcal{M}_\mathsf{x}$ will be a linear function of y's state-action visitation frequencies $y \in Y$, and vice versa. In particular, we define the value of a pair of policies $x \in X$

and $y \in Y$ as

$$V(x, y) = c^{\mathsf{x}} \cdot x + x^T G y - c^{\mathsf{y}} \cdot y.$$

Here c^{x} and c^{y} are fixed cost vectors for X and Y, while the matrix G governs the interaction between the two players; the fixed costs may account for movement costs or other costs in the game that are independent of the other player's policy. Since we interpret x as the min player and y as the max player, y's fixed cost $c^{\mathsf{y}} \cdot y$ gets a negative sign. To represent V as a bilinear form $x^T M y$ we can add a new dimension with a fixed value of 1 to X and Y, and then add c^{x} (c^{y}) as a new column (row) of G.

The matrix G has a row for each state-action pair (s, a) in $\mathcal{M}_{\mathsf{x}}$ and similarly a column for each state-action pair (t, b) in $\mathcal{M}_{\mathsf{y}}$. We can interpret this entry $G((s, a), (t, b))$ as the cost associated with the product of the number of times that x took action a from s and y took action b from state t. In this way we have a convex game equivalent to

$$\min_{x \in \Pi^{\mathsf{x}}_{ND}} \max_{y \in \Pi^{\mathsf{y}}_{ND}} V(x, y), \tag{3.18}$$

the cost-paired MDP game.

Modeling costs in the mobile-sensor game We might model a mobile sensor placement/avoidance problem in the following way: both MDPs $\mathcal{M}_{\mathsf{x}}$ and $\mathcal{M}_{\mathsf{y}}$ have the same state-space S; time is explicitly encoded in the state space, so each $s \in S$ corresponds to being at a particular location $\mathsf{loc}(s)$ at a particular time $\mathsf{time}(s)$. Player y is the sensing player; when he is in state t he can observe all of the states in $\mathsf{obs}(t) \subseteq S$. In particular, $s \in \mathsf{obs}(t)$ if and only if $\mathsf{time}(s) = \mathsf{time}(t)$ and $\mathsf{loc}(s)$ is visible from $\mathsf{loc}(t)$. Then for each state t, we define

$$\forall s \in \mathsf{obs}(t), \quad G((s, a), (t, b)) = z \tag{3.19}$$

for all $a \in A^{\mathsf{x}}$ and all $b \in A^{\mathsf{y}}$, where $z \in \mathbb{R}$ is the cost associated with y observing x for a single timestep.

Consider state-action visitation frequencies $x \in X$ and $y \in Y$. Then let $x_s = \sum_{a \in A^{\mathsf{x}}} x_{(s,a)}$ and $y_t = \sum_{b \in A^{\mathsf{y}}} y_{(t,b)}$. Time is explicitly encoded in the state space and increases after each action, so no state can be reached more than once. Thus, we can interpret x_s as the probability that x is in at location $\mathsf{loc}(s)$ at time $\mathsf{time}(s)$, and similarly for y_t. If $s \in \mathsf{obs}(t)$, then $x_s y_t$ is the probability that y observes x at location $\mathsf{loc}(s)$ at time $\mathsf{time}(s) = \mathsf{time}(t)$ from location $\mathsf{loc}(t)$; our definition of G in Equation (3.19) ensures that $x^T M y$ (when multiplied out) contains the term $x_s y_t z$ to account for this expected cost.

3.5 Convex Stochastic Games

Stochastic games (SGs, also called Markov games) generalize MDPs to multiple players by putting a matrix game, called a stage game, at each state. The game is fully observable, and on each round the players play the stage game by simultaneously selecting a row or column. The immediate payoff from one player to the other and the distribution over the next state in the MDP are both functions of the pair of actions chosen in this way. There is a large body of literature on stochastic games: Neyman and Sorin [2003] and Owen [1995] both offer a good general starting point, while Bowling and Veloso [2000] provide an introduction from a reinforcement learning point of view. Littman [1994] also shows the usefulness of stochastic games as a model for multi-agent learning. Partially observable stochastic games (POSGs) are much more expressive but less tractable than stochastic games. Recent research has shown the usefulness of POSGs, see Emery-Montemerlo et al. [2004] and Hansen et al. [2004] for a variety of applications.

In this section we introduce Convex Stochastic Games (CSGs), stochastic games with convex games in place of the usual matrix stage games. This allows us to embed extensive-form games (transformed to convex games) as the stage games, yielding a tractable class of partially observable stochastic games.

We consider the zero-sum case played by x and y. The convex stochastic game is played on a set S of states; each $s \in S$ is associated with a convex stage-game. The convex game at s has actions sets $X_s \subseteq \mathbb{R}^{m_s}$ and $Y_s \subseteq \mathbb{R}^{n_s}$ for x and y. Next state transition probabilities are defined via non-negative matrices $F^{ss'} \in \mathbb{R}^{m_s \times n_s}$ for every pair of states $s, s' \in S$. The probability that the game transitions from s to s' given that x played $x \in X_s$ and y played $y \in Y_s$ is then defined to be $x^T F^{ss'} y$, so we require

$$\forall s \in S,\ \forall x \in X_s,\ \forall y \in Y_s, \qquad \sum_{s' \in S} x^T F^{ss'} y = 1$$

and

$$\forall s, s' \in S, \forall x \in X_s,\ \forall y \in Y_s, \qquad x^T F^{ss'} y \geq 0.$$

Payoffs are specified via a matrix M^s, so when x plays x and y plays y, the payoff from x to y is $x^T M^s y$. It is straightforward to verify that CSGs generalize stochastic games. The mapping from matrix stage-games to convex game stage-games is given exactly by the transformation described in the Section 3.1. Suppose R_s and C_s are the row and column strategies from the original matrix game at s. Then $X_s = \Delta(R_s)$ and $Y_s = \Delta(C_s)$, the probability simplices explicitly representing the sets of possible mixed strategies for each stage game. The stochastic game's transitions are specified by probabilities $\Pr(s' \mid s, i, j)$

for each $s, s' \in S$ and $i \in R_s$, $j \in C_s$. We construct the transition matrix $F^{ss'}$ by letting $F^{ss'}_{i,j} = \Pr(s' \mid s, i, j)$. Then for mixed strategies $x \in X_s$ and $y \in Y_s$,

$$\Pr(s' \mid s, x, y) = x^T F^{ss'} y.$$

Given a discount factor $\gamma \in (0, 1)$ on future payoffs, the convex stochastic game can be solved via minimax value iteration (this is a straightforward extension to Littman [1994]). In minimax value iteration for SGs, game values are calculated by solving a stage-game modified to take into account estimated future payoffs. The convergence of the algorithm follows because backups are a contraction given $\gamma < 1$ [see Owen, 1995, for a proof]. These same results carry over to CSGs. Let $v \in R^{|S|}$ be the current value function estimate. To backup at s, we solve the convex game at s with the modified objective function

$$x^T M^s y + \gamma \sum_{s' \in S} (x^T F^{ss'} y)\ v(s') \tag{3.20}$$

This objective includes the immediate payoff $x^T M^s y$ and the discounted term $\gamma E[v(s')|x, y]$, which estimates the expected cost of the rest of the game given that x and y are played. Note that Equation (3.20) can be rewritten as

$$x^T \left(M^s + \gamma \sum_{s' \in S} F^{ss'} v(s') \right) y$$

and so it is a bilinear function of x and y. Thus, we can implement the backup operator for minimax value iteration by solving the modified stage games via convex programming, and so solve discounted CSGs.

Solving a class of POSGs Using the convex game transformation reviewed in this chapter, we can embed extensive-form games as the stage games in convex stochastic games. We can view this overall structure as an EFG with loops, and can "unroll" this embedding with the following interpretation: each EFG stage-game corresponds to a subgame (both players know which subgame is being played). These subgames have partial observability, but after a subgame completes a fully-observable transition is made to another subgame. However, all "back edges" must be to nodes that begin subgames. This game is a fairly general POSG: it has partial observability and states can repeat. It can be solved in polynomial time using the techniques introduced in this section. The key is that the periods of partial observability are of bounded duration (equal to the height of one of the embedded EFGs); the solution time is polynomial in the representation of the game, but possibly exponential in this horizon time.

For some interesting applications, assuming that only short periods of partial observability occur between periods of full observability is reasonable — for example, we could plan how to handle a temporary failure of a lighting system, GPS localization, or other sensors. We could also model a two-player poker tournament where each stage game corresponds to playing a game of poker with a fixed (fully observable) initial number of chips for each player. The result of each poker game produces a fully observable transition to another game (where the number of starting chips for each player depends on the outcome of the last game), or the end of the tournament (say, if one player runs out of chips).

However, if the partial observability in some domain is due to an adversary that may go unobserved for long periods of time, this approach will not produce tractable games. After we introduce CEFGs in the next chapter, we will argue that more realistic problems can be modeled by embedding CEFGs as stage games in a convex game, because they provide a powerful method for treating a complex sequence of decisions as a single decision, thus decreasing the effective horizon of the embedded games.

Chapter 4

Generalizing Extensive-form Games with Convex Action Sets

This chapter develops the class of *convex extensive form games* (CEFGs). These games are a powerful generalizations of extensive-form games that can still be solved in polynomial time in the size of the game representation, under reasonable assumptions. Like an EFG, a CEFG is a game with partial information played on a game tree, however, in CEFGs:

1. An arbitrary subset of players simultaneously select actions at each node, much like in a normal form game.
2. The sets of actions available to each player at a given information set is a convex subset of $\mathbb{R}^n$, rather than a discrete set.
3. Payoffs are made at internal nodes as well as at leaves, and are given by a multi-linear function of the players' actions.
4. A linear function associates a product distribution over successor nodes with each possible joint action.
5. Two nodes that are both in the same information set may have different numbers of successor nodes.

These generalizations allow us to embed arbitrary convex games at the nodes of a CEFG. This effectively unifies the problems of planning in a MDP and solving for the minimax solution of an EFG: an MDP is a single-player, single-node CEFG, and an EFG can be transformed to a CEFG on the same game tree. The problem of solving an MDP

where one player selects a policy and another player chooses the cost function was addressed in Section 3.4. This problem can be modeled as a two-player, single-node CEFG. More general versions of this problem, where the players have some limited opportunities to observe their opponent's past actions, can also be solved as CEFGs.

This unification has practical applications to problems typically modeled as EFGs as well. In particular, our results make it possible to efficiently model games with outcome uncertainty. Modeling outcome uncertainty (where a single action can result in a distribution over outcomes rather than a single deterministic outcome) in standard EFGs causes an exponential blowup in the representation size, but with CEFGs we avoid this blowup. This has ramifications both to computing sequentially-rational equilibria and opponent modeling. We discuss these and other applications of CEFGs in Section 4.4.

Efficient computation of equilibria in zero-sum EFGs depends on the property of *perfect recall*; the problem is NP-hard without this assumption [Koller and Megiddo, 1992]. "Perfect recall" CEFGs would not be tractable due to the exponential or infinite number of possible actions at each node. We develop a generalization of perfect recall for CEFGs, *sufficient recall*, that allows some "forgetting" of past actions. A principal contribution of this work is showing that sufficient-recall zero-sum CEFGs can be transformed to convex games and hence solved efficiently.

We define sufficient recall as the combination of observation memory and sufficient action memory (we define all these terms in Section 4.2), and then show that this formulation is equivalent to a notion of *sequence recall* that is more akin to the definition of perfect recall in EFGs. Thus, this result serves as a new characterization of perfect recall for EFGs. Other characterizations of perfect recall have recently appeared in the literature (see Bonanno [2004] and the references therein). Some of these characterizations may be related to our characterization when it is applied to EFGs represented as CEFGs, but as of yet, we have not investigated this relationship.

We are not aware of any similar generalizations of extensive-form games currently in the literature. Selten [1999] does mention a multistage game model where multiple agents select actions at each stage. He only considers the perfect information case, but mentions that "the framework could be made as general as that of an extensive game by the additional introduction of information partitions." However, he provides no additional discussion of the methods for doing this, or of their ramifications; he also does not consider convex action sets.

Components of a CEFG	
$N = \{0, 1, 2, \ldots, n\}$	the n players of the game; 0 a is a chance player.
$T = \langle V, E \rangle$	the extensive form game tree on nodes V and edges E
$V_p \subseteq V$	player p's decision nodes
U_p	set of player p's information sets
$X_u \subseteq \mathbb{R}^{n_u}$	convex sets defining the action space at u (u defines player)
$f_p^{ss'}$	linear transition function, $f_p^{ss'} : X_u \to [0, 1]$
M_p^s	payoff function at node s for player p
Additional Notation	
p	an arbitrary player, $p \in N$
s, t	nodes in V
$\mathcal{A}(s) \subseteq N$	set of players active at $s \in V$
$u \in U_p$	$u \subseteq V_p$, an information set for player p
$\phi_p(s) \in U_p$	p's information set containing s if $s \in V_p$; $\Diamond_p$ otherwise
$\bar{X}_s$	set of joint actions possible at node s
$x_u \in X_u$	an action taken at u
X_u^c	cone extension of set X_u
$w(s \mid \pi_p)$	p's sequence weight on state s under policy π_p
$w(u \mid \pi_p)$	p's sequence weight on info set u under policy π_p

Table 4.1: Summary of notation for convex extensive-form games.

4.1 CEFGs: Defining the Model

In this section, we introduce Convex Extensive Form Games (CEFGs) with n players and general payoffs, and then provide brief commentary on the interpretation of the model and its connection to EFGs.

As one would expect, CEFGs generalize EFGs. The principal differences between the two representations were outlined in the introduction. In this section, we formally define the model and associated notation. While we mention some differences to EFGs and sketch a transformation from a EFG to an equivalent CEFG, our definition of the model has no direct dependence on EFGs. Table (4.1) summarizes the components that define a CEFG, as well as some associated notation introduced here and in subsequent sections. After formally introducing the model, we present several examples and interpretations in order to make the definition more concrete, show the connection to standard EFGs, and demonstrate the expressive power of CEFGs.

The game tree and information sets A CEFG is played on a directed, finite game tree $T = \langle V, E \rangle$ rooted at s^*. For any $s \in V$, there is a unique $s^* \to s$ path, with nodes denoted by $\mathsf{path}(s) = (s^1, s^2, \ldots, s^k)$ where $s^1 = s^*$ and $s^k = s$, and edges $\mathcal{E}(s) = ((s^1, s^2), (s^2, s^3), \ldots, (s^{k-1}, s^k))$. The game is played by a set $N = \{0, 1, 2, \ldots, n\}$ of players, where the 0 player is an optional chance (or "nature") player. We generally state results for an arbitrary player p; when it is clear to which player we are referring, we omit the subscript p to simplify notation.

Each player is active (selects an action) on an arbitrary subset of the internal (non-leaf) nodes,[1] $V_p \subseteq V$; these are player p's decision nodes. Let $\mathcal{A}(s) = \{p \mid s \in V_p\}$, the set of active players at s. We require $|\mathcal{A}(s)| \geq 1$ for all internal nodes s and $|\mathcal{A}(s)| = 0$ for leaves.

As in EFGs, the decision nodes V_p for each player are partitioned into information sets U_p. Formally, $\bigcup_{u \in U_p} u = V_p$, and for all $u, u' \in U_p$, we have $u \cap u' = \emptyset$ whenever $u \neq u'$. When play reaches a node s with $s \in u$ for an information set u of player p, then player p observes (is told) that the game has reached u, but the specific $s \in u$ is not revealed; that is, all $s, s' \in u$ are indistinguishable to p.

For nodes s where p is not active ($s \notin V_p$), we define (for notational convenience) a special "non-information set" $\diamondsuit_p$. In particular, $\diamondsuit_p \notin U_p$ and player p never receives $\diamondsuit_p$ as an observation. For any node $s \in V_p$, there exists exactly one information set $u \in U_p$ such that $s \in u$; let $\phi_p(\cdot)$ be the function that identifies this u, that is for any $s \in V_p$, $\phi_p(s) \in U_p$ and $s \in \phi_p(s)$; when $s \notin V_p$, let $\phi_p(s) = \diamondsuit_p$, so that the domain of ϕ_p is all of V. To simplify notation, when u is not otherwise specified it can be read as $\phi_p(s)$.

We explicitly exclude the property of absent-mindedness (see Piccione and Rubinstein [1997]) by requiring that if $s, s' \in V_p$ are on some path to a leaf, then $\phi_p(s) \neq \phi_p(s')$ (note that if $s, s' \notin V_p$ then $\phi_p(s) = \phi_p(s') = \diamondsuit_p$, but this doesn't matter). For any state s and player p, let $\mathsf{obs}_p(s)$ be the sequence of information sets that occurs on the path to s: $\mathsf{obs}_p(s)$ has an entry u for each $\phi_p(s^i) \neq \diamondsuit_p$.

A few more notes on notation: to indicate that a particular entity belongs to a particular player, we subscript with either p or u, for example x_u for an action or π_p for a policy.[2]

[1] One can imagine a matrix game is played at each node by some subset of the players, though as we will see our model is much more general than this. The fact that we may have *strict* subsets of players active at each node will require some notational gymnastics, however, this is *necessary* to maintain a direct transformation from EFGs to CEFGs; we will return to this point once we have fully described our model.

[2] This is not technically precise in the case of x_u, as formally u is simply a subset of V, and hence two different players, say p and p', could "share" an information set, that is, $u \in U_p$ and $u \in U_{p'}$. However, when we write u it will be clear from context that u is associated with a particular player, almost always player p.

We indicate an entity is a tuple over players with a bar: for example, $\bar{x}$ is a joint action. Entries in a tuple over players are indexed with a subscript $p \in N$, and shown without the bar. That is, $\bar{x} = \{x_0, x_1, x_2, \ldots, x_n\}$. We use a bar over capital symbols to denote sets of such tuples, for example, $\bar{X}_s$ is a set of possible joint actions.

Actions and costs Consider a play of the game that reaches node s, and suppose $u = \phi_p(s)$ for player $p \in \mathcal{A}(s)$. At s player p only observes u (that is, p cannot differentiate between the nodes in u), so we require that all nodes in u share the same set X_u of actions available to p. However, it is possible for two nodes $s, s' \in u$ to have different numbers of successors in the game tree, unlike in EFGs.

In an EFG, X_u would typically be a small finite set; a principal difference in CEFGs is that X_u is a convex subset of $\mathbb{R}^{n_u}$. At s, each player $p \in \mathcal{A}(s)$ selects an action $x_p \in X_u$. Note that when a player p selects an action at u, she may well not even know how many players are simultaneously selecting an action, as this is a function of the (unobserved) state $s \in V$. For notational simplicity, we define the joint action $\bar{x} = (x_0, x_1, x_2, \ldots, x_n)$ as a tuple over all the players, where we have $x_p \in X_u$ for $p \in \mathcal{A}(s)$, and arbitrarily fix $x_p = 1$ for $p \notin \mathcal{A}(s)$; in fact, we simply define $X_{\diamondsuit_p} = \{1\}$, and so the set of all possible joint actions at s is $\bar{X}_s = \bigotimes_{p \in N} X_{\phi_p(s)}$ (we use $\bigotimes$ to denote the Cartesian set product). This allows us to view joint actions as a tuple of actions over all the players, even though some of the players do not actually make a decision and in fact have no knowledge (other than what is conveyed later via their information sets) of the fact that s was reached.

Costs are incurred at internal nodes, not just at leaves as for EFGs. The payoff to each player p (for all $p \in N$, not just those $p \in \mathcal{A}(s)$) is given by a function $M_p^s : \bar{X}_s \to \mathbb{R}$. We require that M_p^s be a multi-linear function of $x_0, x_1, \ldots, x_n$, that is, it is a linear function of x_p when the actions of all other players are held constant.[3] Note that this definition of the action sets and payoffs implies that a convex game is being played at each internal node. However, the players involved may have uncertainty about the game: they know their own X_u, but may not know the payoff matrix (bilinear payoff function for $n > 2$), which other players are playing, and what actions those players have available.

Leaf nodes have $\mathcal{A}(s) = \emptyset$, and so the payoff to each player is a constant, written as $M_p^s(\vec{1})$ where $\vec{1}$ is the vector of $(n+1)$ 1s.

Successors and transitions While we may have an infinite set of possible actions X_u, we wish to avoid infinite branching in the game tree T. Thus, we assume each internal

[3]The assumption is necessary for efficient computation on G, and is also necessary to show the equivalence of explicit behavior and implicit behavior policies (introduced later).

node $s \in V$ has a finite set of successors, denoted $\mathsf{succ}(s)$. Given a joint action $\bar{x}$ at s, we need to specify how the successor state $s' \in \mathsf{succ}(s)$ is chosen. We do this via a product distribution over $\mathsf{succ}(s)$ that is a linear function of each player's individual action. In particular, for each $p \in \mathcal{A}(s)$ and $s' \in \mathsf{succ}(s)$, we define a linear function $f_p^{ss'} : X_u \to \mathbb{R}$. The probability that s' is the next state after s is given by:

$$\Pr(s' \mid s, \bar{x}) = \prod_{p \in \mathcal{A}(s)} f_p^{ss'}(x_p). \tag{4.1}$$

Thus, we require that these functions satisfy the following constraints for all $s \in V$ and $\bar{x} \in \bar{X}_s$:

$$\prod_{p \in \mathcal{A}(s)} f_p^{ss'}(x_p) \geq 0 \qquad \text{and} \qquad \sum_{s' \in \mathsf{succ}(s)} \prod_{p \in \mathcal{A}(s)} f_p^{ss'}(x_p) = 1. \tag{4.2}$$

Again, to avoid special cases we define $f_p^{ss'}$ for all $p \notin \mathcal{A}(s)$ as $f_p^{ss'}(x_p) = 1$, that is, the identity function, since $X_{\diamondsuit_p} = \{1\}$. Thus, we can replace the products over $p \in \mathcal{A}(s)$ in Equation (4.2) with products over $p \in N$. Note that the $f_p^{ss'}$ functions can, for example, be constant functions specifying a fixed probability distribution, and so functions satisfying the constraints (4.2) always exist. For leaf nodes s, we assume $|\mathcal{A}(s)| = 0$, and no transition functions are defined.

The assumptions in Equation (4.2) are sufficient for the CEFG to be well defined, that is, they specify a valid probability distribution over successors that is a product distribution. However, the model will be difficult to interpret if some $f_p^{ss'}(x) > 1$. Thus, in this paper we make the following assumption:

Assumption 4.1.1. *For all $(s, s') \in E$, all $p \in \mathcal{A}(s)$, and all $x_p \in X_u$, $f_p^{ss'}(x_p) \in [0, 1]$.*

This assumption allows us to view $f_p^{ss'}(x)$ as the probability of some event depending only on player p's action x (we explore this idea in more detail in the next section). This assumption is in fact made without loss of generality; see Appendix A for the proof.

Let G and G' be two CEFGs. We say that G and G' are *f-equivalent* if they are identical except for their f functions, and for all (s, s'), for all $\bar{x} \in \bar{X}_s$, $\Pr_G(s' \mid s, \bar{x}) = \Pr_{G'}(s' \mid s, \bar{x})$.

We model the random player as having a separate information set for each of her decision nodes, and fix $X_u = \{1\}$ for all $u \in U_0$. Thus, the random player makes no decisions, and is defined by her (effectively) constant f-functions, $f_0^{ss'}$.

Gameplay and payoffs A CEFG is played in a similar fashion to an EFG. We can imagine a referee who starts the game at $s^* = s^1$. All players in $p \in \mathcal{A}(s^*)$ simultaneously

and independently select an action from $X_{\phi_p(s^*)}$, and the referee assembles these choices into a joint action vector $\bar{x}^1$, using 1 as the action for all players not active at s^*. The referee then computes the payoff $M_p^s(\bar{x}^1)$ to each player p and selects the successor state s^2 based on $\bar{x}^1$ according to Equation (4.1). Each player $p \in \mathcal{A}(s^2)$ receives $\phi_p(s^2)$ as an observation and is asked for an action, and the game continues in this fashion until a leaf is reached.

The *partial history* h of the gameplay so far can be written as

$$h = ((s^1, \bar{x}^1), (s^2, \bar{x}^2), \ldots, (s^k, \bar{x}^k), s^{k+1})$$

where s^1 is always the root node of the game. A *complete history* or *play* is a partial history where s^{k+1} is a leaf node. Such a history can be interpreted as saying for each tuple i: node s^i was reached, each player $p \in \mathcal{A}(s^i)$ observed their information set $\phi_p(s^i)$, and then played $x_p \in X_{\phi_p(s^i)}$. The game transitioned to s^{i+1}, some successor of s^i with $\Pr(s^{i+1} \mid s^i, \bar{x}^i) > 0$. The value of a history h to player p (that is, the total payoff to p) is given by

$$V_p(h) = \sum_{(s,\bar{x}) \in h_p} M_p^s(\bar{x}).$$

To avoid notational hassles, if the last state in the partial history h is a leaf, we assume it is associated with a vacuous joint action so it is included in this sum and so the final (constant) payoff is counted. This value can be thought of as the sum of the payoffs of the individual convex games played along the path to the leaf. The goal of each player in the game is to maximize their own total payoff, $V_p(h)$.

It is also useful to define the *partial player history*, h_p, the portion of the history h observable by player p,

$$h_p = ((u^1, x_p^1), (u^2, x_p^2), \ldots (u^k, x_p^k), u^{k+1})$$

ending in a player p information set. The partial player history only contains tuples corresponding to observations p received, that is, it has no tuples where u^i is $\diamondsuit_p$. Each tuple i can be read as: player p observed $u^i \in U_p$, selected action $x^i \in X_{u^i}$, and then at some later point was "woken up" with the observation of u^{i+1}. If h is a history of k, the length of the partial player history for any particular player p may have length much less than k, possibly even zero.[4] Let $\mathcal{H}$ be the set of all possible complete plays of the CEFG, and let H_p be the set of all partial player histories for player p.

[4]In an EFG, length$(h) = \sum_p$ length(h_p), as each internal node is in exactly one decision set.

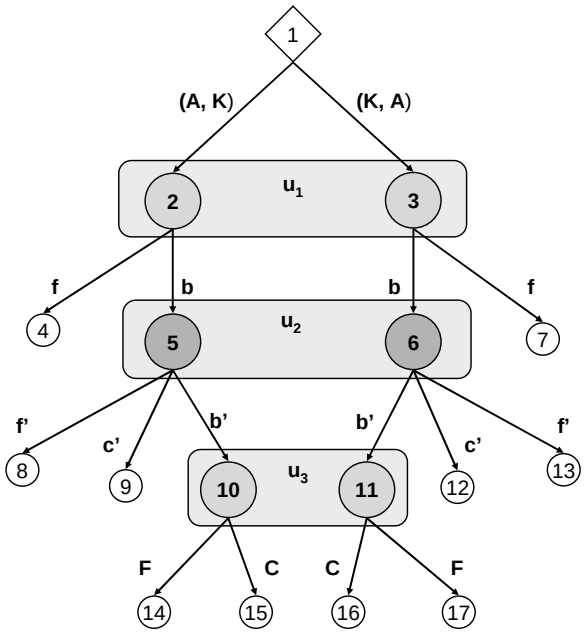


Figure 4.1: A simple poker game with a two card deck (Ace and King), represented as a convex extensive-form game.

Interpretations and Examples

In this section we present several small examples and demonstrate the connection to standard EFGs.

A simple poker game, revisited We show how to represent the poker game from Section 3.2 as a convex extensive-form game. The CEFG representation is given in Figure 4.1; this figure is essentially the same as Figure 3.2, but we have renumbered the states (including the leaves), and do not show payoffs for clarity (they are identical). We have $V = \{1, \ldots, 17\}$, with $V_{\mathsf{x}} = \{2, 3, 10, 11\}$ and $V_{\mathsf{y}} = \{5, 6\}$. The information sets are again $U_{\mathsf{x}} = \{u_1, u_3\}$ and $U_{\mathsf{y}} = \{u_2\}$, where $u_1 = \{2, 3\}$, $u_2 = \{5, 6\}$, and $u_3 = \{10, 11\}$. The labels on edges are for reference only, and cannot in general be viewed as actions in a convex extensive form game. However, in Section 4.2, we will introduce the concept of outcomes which partition the edges out of information sets. The labels shown in the figure can be interpreted as outcome labels. The action sets are $X_{u_1} = \Delta(\{f, b\})$, $X_{u_2} = \Delta(\{f', c', b'\})$, and $X_{u_3} = \Delta(\{F, C\})$. The transition functions associated with

state 2 are

$$f_{\mathsf{x}}^{2,4}(x) = x_1 \qquad f_{\mathsf{y}}^{2,4}(y) = 1$$
$$f_{\mathsf{x}}^{2,5}(x) = x_2 \qquad f_{\mathsf{y}}^{2,5}(y) = 1,$$

and it is easy to verify that in fact $f_{\mathsf{x}}^{2,4}(x) f_{\mathsf{y}}^{2,4}(y) \geq 0,\ f_{\mathsf{x}}^{2,5}(x) f_{\mathsf{y}}^{2,5}(y) \geq 0$, and

$$f_{\mathsf{x}}^{2,4}(x) f_{\mathsf{y}}^{2,4}(y) + f_{\mathsf{x}}^{2,5}(x) f_{\mathsf{y}}^{2,5}(y) = 1$$

for all $x \in \Delta(\{f, b\})$ and $y \in X_{\diamondsuit_{\mathsf{y}}} = \{1\}$, as required by Equation (4.2).

For a general convex extensive-form game the transition functions from states 2 and 3 could be different, and in fact each state could have a different number of outcomes; but, in this case we are simply transforming an EFG and so the functions will be identical. More precisely, we have $f_{\mathsf{x}}^{2,4} = f_{\mathsf{x}}^{3,7}$ (corresponding to the fold outcome), and $f_{\mathsf{x}}^{2,5} = f_{\mathsf{x}}^{3,6}$ (corresponding to the bet outcome). Equalities of this type will play a central roll in our definition of sufficient recall.

We have omitted the constant $f = 1$ functions of the random player at node 2; at node 1, only the random player is active, with $f_0^{1,2} = 0.5$ and $f_0^{1,3} = 0.5$. The payoff functions M at internal nodes are the constant zero function, and the payoffs at the leaves are constant functions corresponding to the payoffs shown in Figure (3.2).

A useful interpretation of the f-functions Imagine that at a node s, player p selects an action $x_p \in X_u$, and that action the action x_p determines a probability distribution over some disjoint events[5] $A_p = \{a_1, a_2, \ldots, a_{k_p}\}$. This distribution is independent of the other player's action if the other player happens to be active at s. Suppose that this probability distribution is, in fact, a linear function of x, and, WLOG, $x_1, \ldots x_{k_p}$ are the probabilities[6] of $a_1, \ldots, a_{k_p}$, that is $x_i = \Pr(a_i \mid x)$. The f-functions are linear, so suppose

$$f_p^{ss'}(x) = \sum_{i=1}^{k_p} c_i x_i.$$

[5] Any resemblance to the notation used for actions in an EFG is purely unintentional.

[6] This is without loss of generality because if it is not the case we can embed the linear functions that define $\Pr(a_i \mid x)$ into a new dimension of X_u via an equality constraint.

If we choose coefficients $c_i \in \{0, 1\}$ then each $f_p^{ss'}$ function simply adds up the probabilities on some union of events in A. Letting $A_p(s') = \{a_i \in A_p \mid c_i = 1\}$, we have

$$f_p^{ss'}(x) = \sum_{i=1}^{k_p} c_i x_i = \sum_{i=1}^{k_p} c_i \Pr(a_i \mid x) = \Pr(A_p(s') \mid x).$$

Thus, holding other player's action constant, the probability with which the game transitions to s' is proportional to the probability of the event $A_p(s')$.

We can independently interpret each player's f-functions in this way. The total set of f-functions (one for each player/successor pair) determining the transition probabilities at s must be chosen to ensure that each joint event (some $a \in A_p$ for each player p) is associated with exactly one successor state s'. Since the distribution on each A_p is independent, the resulting distribution is a product distribution.

As a concrete example, consider two players, x and y, at node s, where player x's action determines a distribution on events $A = \{a_1, a_2\}$, and player y's action determines a distribution on $B = \{b_1, b_2\}$. Suppose x has action set

$$X = \{(x_1, x_2) \mid x_i \geq 0, x_1 + x_2 = 1\} = \Delta(A),$$

that is, she can choose any distribution she wants on her events, and similarly $Y = \Delta(B)$. Suppose from s there are 3 possible successors: s^1, s^2, and s^3. Then, we might have $\Pr(s^1) = \Pr(a_1)$ and $\Pr(s^2) = \Pr(a_2 \wedge b_1)$ and $\Pr(s^3) = \Pr(a_2 \wedge b_2)$. The corresponding f functions would be:

$$\begin{array}{lll} f_{\mathsf{x}}^{s,s^1}(x) = x_1 & f_{\mathsf{x}}^{s,s^2}(x) = x_2 & f_{\mathsf{x}}^{s,s^3}(x) = x_2 \\ f_{\mathsf{y}}^{s,s^1}(y) = y_1 + y_2 & f_{\mathsf{y}}^{s,s^2}(y) = y_1 & f_{\mathsf{x}}^{s,s^3}(y) = y_2 \end{array}$$

and so

$$\begin{array}{lll} \Pr(s^1) = x_1(y_1 + y_2) = \Pr(a_1)\Pr(b_1 \vee b_2) & = \Pr(a_1) \\ \Pr(s^2) = x_2 y_1 \quad = \Pr(a_2)\Pr(b_1) & = \Pr(a_2 \wedge b_1) \\ \Pr(s^3) = x_2 y_2 \quad = \Pr(a_2)\Pr(b_2) & = \Pr(a_2 \wedge b_2). \end{array}$$

The relationship between the successors $\{s^1, s^2, s^3\}$ and the joint probability space $A \bigotimes B$ is shown pictorially in the following table:

	b_1	b_2
a_1	s^1	s^3
a_2	s^2	s^3

The f-functions can always be represented as a partition of the product space of the individual event sets A_p. However, arbitrary partitions are not possible: each successor state must be associated with a "rectangle" of the product event space, that is, each successor s' corresponds to some Cartesian product of subsets of each A_p, namely $\bigotimes_{p \in N} A_p(s')$

In all of our applications, the f-functions can be interpreted in this way. For example, consider an MDP embedded at node s of a CEFG, so X_u for player p is the set of state-action visitation frequencies corresponding to valid stochastic policies (this set is defined in Section 3.4.3). Further, suppose only p is active at s. If the MDP has k terminal states (absorbing goal states), then we can define events $a_1, \ldots, a_k$ where a_i is the event that the policy selected takes the agent to terminal state i. These probabilities are a linear function of $x \in X_u$, and so we might have k successors s', where each $f^{ss'}$ function computes the probability that the agent reaches a particular terminal state under the chosen policy.

Interpretations of the action sets As with with convex games, we have two possible interpretations of the set X_u:

1. We interpret the set X_u as a continuous set of primitive actions.
2. We treat only the extreme points $\mathsf{Cn}(X_u)$ as primitive actions, with interior points defining an equivalence class of mixed strategies.[7]

In Section (3.1) we showed that all distributions over $\mathsf{Cn}(X)$ (and hence, X_u for CEFGs) correspond to (immediate) payoff equivalent actions (strategies for the convex game). However, in CEFGs we must also worry about the rest of the game: two actions that lead to the same immediate payoff but produce potentially different distributions over successor nodes in the game tree are clearly not equivalent. However, because the transition functions f are linear, it is easy to modify the arguments for immediate payoff equivalence to show that any two probability distributions over corners that produce the same interior point x also produce the same distribution over successor nodes for any (fixed) joint policy for the other players. Hence, even if we consider the set $\mathsf{Cn}(X_u)$ to be the actual set of primitive actions, we can optimize over the set X_u and sample from a small support representation of an interior point if needed.

Connection to EFGs In EFGs, no generality is gained by assigning costs at internal nodes s, as we can always simply add any immediate costs at s to the cost at every leaf

[7]As with convex games, we could use any subset $X' \subseteq X$ instead of $\mathsf{Cn}(X_u)$, as long as we have an efficient algorithm to express any point $x \in X$ as a distribution over points in X'.

reachable from s. However, in CEFGs the costs we incur may depend on the exact action x_u, not just the successor state, and so deferring costs to the leaf would require "remembering" the action x_u in the tree, giving rise to an infinite branching factor. By assigning some cost based on $\bar{x}_s$ immediately at the internal node s, our model can handle costs that depend on continuous, multi-dimensional actions. The key is that the full action impacts only the immediate cost incurred, and that once that cost is accounted for, only a finite number of successor states are possible.

The fact that we have continuous, multi-dimensional actions has significant ramifications for tractable algorithms. The efficient solution of two-player, zero-sum EFGs relies on the assumption of *perfect recall*: informally, each information set u for player p uniquely determines the sequence of actions p has taken up to that point. Clearly, we will not be able to have this property given continuous actions and a finite tree—some forgetting must be allowed. Thus, in the next section we introduce the concept of *sufficient recall*, which provides enough "memory" in the tree to allow optimal play given only the current information set without fully encoding all past actions.

Finally, we note that CEFGs generalize EFGs. In particular, for any EFG G there is a mapping to an equivalent CEFG G', such that an equilibrium solution in G' can be mapped back to an equilibrium solution of G. The mapping from an EFG G to a CEFG G' is the natural one. Each node in G corresponds to a node in G'; information sets are also mapped directly, so that in G' we will have $|\mathcal{A}(s)| = 1$ for all internal nodes, and $|\mathcal{A}(s)| = 0$ at the leaves. Each internal node has a payoff function $M_p^s(\bar{x}) = 0$, and each leaf has a constant payoff $M_p^s(\bar{1}) \in \mathbb{R}$ equal to the payoff at the corresponding leaf in G.

For each internal node s, if the active player p's choices in G for $u = \phi_p(s)$ were $C_u = \{c_1, \ldots, c_k\}$, we define $X_u \subseteq \mathbb{R}^k$ to be the k-dimensional probability simplex $\Delta(C_u)$. The corners $\mathsf{Cn}(X_u)$ correspond to the choices/outcomes in G, while an interior point x corresponds to a behavior (a distribution over choices). The successors of s are (say) $s^1, \ldots, s^k$, and the f-functions are simply $f_p^{ss^i}(x) = x_i$.

It is easy to show that this mapping produces a valid CEFG, and that policies can be transferred back and forth between G and G', with payoffs being identical in G or G'.

Compact representations with CEFGs We provide a quick illustration of the power of the CEFG model by showing there are games with a CEFG representation that is exponentially smaller than the EFG representation. Consider the following zero-sum game played by x and y in k rounds. On each round, each player chooses a number from $\{1, \ldots, 10\}$. At the end of k rounds, player x pays y an amount that depends arbitrarily on the sequence of k numbers x picked, plus \$1 for each time y guessed x's action. Neither player observes

the other player's past actions.

This game can be represented as an EFG, played on a height $2k$ tree with branching factor 10: there is one leaf for each possible pair of length k sequences, corresponding to the numbers picked by x and y. Thus, this tree has 10^{2k} leaves. This game has an exponentially smaller representation as a CEFG: the game tree is now of height k, and each leaf corresponds to the sequence of x's choices; the choices of y are "forgotten" by the tree. Each internal node is a matrix game where x pays \$1 if y guesses his actions (that is, M^s is the 10×10 identity matrix). Thus, the possible immediate payoff of \$1 depends on both actions, but the successor state depends only on x's action (all $f_{\mathsf{y}}^{ss'} = 1$). An additional constant payoff is given at each leaf based on the sequence of x's choices. This tree has only 10^k leaves, and so the representation is exponentially smaller than the EFG (encoding the matrix games only increases the size by a constant factor).

This game might model a situation where y is placing bets on player x's location, while player x is trying to accomplish some task. More realistic games can certainly be constructed; the goal here is to illustrate the representative power of CEFGs. Note that we have not fully exploited the power of this model: our action sets were still only probability simplexes, and all nodes in an information set had the same number of successors.

4.2 Sufficient Recall and Implicit Behavior Reactive Policies

In this section, we develop some important theoretical results concerning CEFGs. Our principal result will be developing a notion of sufficient recall (analogous to perfect recall in EFGs) and showing that for CEFGs with sufficient recall, a class of behavior strategies always contains an optimal policy. These results allow us to construct a polynomial-time algorithm for solving sufficient-recall CEFGs in the next section.

Policies and Probability

In this section, we formally define payoffs for a CEFG, define some policy classes, and define payoff equivalence.

Policy classes A policy (or strategy) is a complete description of how to play a game; in the case of CEFGs, it is a means of selecting an action $x \in X_u$ at each information set u that occurs. We can think of policies as functions or programs depending on our point of

view. Generally speaking, however, any reasonable policy must select its action at u based only on its past observations and actions, and possibly some source of randomness.

We formalize policies as functions and define terminology by specifying different possibilities for the domain and range. A policy function is *history dependent* if it takes as input the partial player history so far, that is, its domain is H_p. A policy is *reactive* (or memoryless) if it only depends on the current information set, but not on its past actions or observations. For the range: a *pure* policy chooses a single action from the set of primitive actions[8] $\mathsf{Cn}(X_u)$. An *implicit behavior* policy picks an interior point of X_u, and interprets this point as a distribution over the primitive actions $\mathsf{Cn}(X_u)$ (that is, it samples a corner from an arbitrary probability distribution from the equivalence class of such distributions defined by the interior point). Finally, an *explicit behavior* policy specifies a particular distribution over $\mathsf{Cn}(X_u)$. An explicit behavior policy might put positive probability on an exponential number of corners of X_u, but an implicit behavior at u can always be represented concisely.

These choices give us 6 classes of policy functions, for each combination of domain and range. When naming policies we specify the range first, then the domain, and so refer to: pure history policies, implicit behavior history policies, implicit behavior reactive policies, and so on. A *mixed* policy is defined by a probability distribution over one of the above classes. Considering the mixed versions of the above classes gives a total of 12 policy classes. Fortunately, we will only need to focus on a few of these classes.

The literature on EFGs generally considers mixed, pure, and behavior strategies. An EFG pure policy is a pure reactive policy in our terms, an EFG mixed policy is a mixed pure reactive policy, and an EFG behavior policy is an (implicit or explicit) behavior reactive policy.[9] The set of *general* policies is the union of the policy classes just mentioned: it can be thought of as the set of all possible strategies a player could use. We will be particular concerned with the class of *implicit behavior reactive policies* (IBRPs), which are policies specified by a function from the current information set u to the set X_u. We often denote such policies by β, and write $\beta(u) \in X_u$ for the action selected at u.

We use κ_p to denote a general player p policy. We write $\bar{\kappa}_{-p}$ for a joint policy for all players except p, that is,

$$\bar{\kappa}_{-p} = (\kappa_1, \kappa_2, \ldots, \kappa_{p-1}, \kappa_{p+1}, \ldots, \kappa_n)$$

[8]For simplicity we assume the set of primitive actions is $\mathsf{Cn}(X_u)$, as this is the most common case. But for some games it might be all of X_u or some other subset of X_u.

[9]In an EFG, the set X_u is the probability simplex over choices, and so there is a one-to-one correspondence between interior points and distributions over corners. Hence, the class of implicit behavior and explicit behavior policies are identical in EFGs (even if represented as CEFGs)

and let $(\kappa_p, \bar{\kappa}_{-p})$ be the joint policy where players other than p play according to $\bar{\kappa}_{-p}$ and player p follows κ_p.

Probability We introduce the basic probability measure used for probability statements about CEFGs, and also establish our notation for various events. For simplicity, we assume each policy only ever plays actions from a countable subset of X_u. This simplifies the notation and proofs in the next section by allowing us to always work directly with probabilities rather than probability densities; it also makes the connection to results for EFGs more clear. In particular, this assumption allows us to work with the probability that a policy picks a certain action given that u is reached, $\Pr(x \mid u)$. Based on this assumption, we abuse notation slightly by writing sums like $\sum_{x \in X_u} \Pr(x \mid u)$, when we implicitly mean only summing over only those $x \in X_u$ that the policy might actually play.

With suitable attention to technical detail, these results should go through for policies that select actions from all of X_u by working with the appropriate probability densities. In fact, for the case of polytopes, the restriction to policies that play from a countable subset of X_u is without loss of generality: any policy that sometimes plays interior points can be interpreted as a policy that plays a distribution over extreme points.[10]

Any joint policy $\bar{\kappa} = \{\kappa_1, \ldots, \kappa_n\}$ where each player fixes some general policy κ_p induces a probability distribution on $\mathcal{H}$. When we want to make it clear which joint policy is associated with a given probability or expectation, we include the policy as a condition, for example, $\Pr(s \mid \bar{\kappa})$; subscripting $\Pr$ would be more precise, but is typographically cumbersome.

For a fixed $\bar{\kappa}$, V_p is a random variable, and the expected payoff $\mathcal{V}_p$ to player p under joint policy $\bar{\kappa}$ is

$$\mathcal{V}_p(\bar{\kappa}) = E[V_p].$$

When a state s appears where an event (a subset of $\mathcal{H}$) is appropriate, we treat s as the subset of the complete histories in $\mathcal{H}$ in which s occurs (s is reached at some point in the play). We denote by $\neg s$ the complement of this set, the set of plays in which s does not occur. Similarly, we view an information set u for player p as the subset of plays in $\mathcal{H}$ where some $s \in u$ is reached; in this context $u = \cup_{s \in u} s$. We write (u, x_p) for the event that player p plays $x_p \in X_u$ from information set u. When u and p are clear from context, we write simply x for this event.

A policy κ_p for player p is *payoff equivalent* to another policy κ'_p, if for all $\bar{\kappa}_{-p}$ for the other players, for all players $q \in N$, $\mathcal{V}_q(\kappa_p, \bar{\kappa}_{-p}) = \mathcal{V}_q(\kappa'_p, \bar{\kappa}_{-p})$. This definition of

[10]Some care must still be taken when dealing with an unbounded polyhedron X.

equivalence is standard [see Dalkey, 1953, for example]. Finally, we state one reasonable additional assumption:

Assumption 4.2.1. *For every $s \in V$, there exists at least one joint policy $\bar{\kappa}$ such that* $\Pr(s \mid \bar{\kappa}) > 0$.

If a CEFG of interest does not satisfy this assumption, unreachable states can be removed in a pre-processing step.

Sequence Weights

In this section, we prove some basic results that apply to all CEFGs, even those without sufficient recall. In particular, we introduce a generalized notion of sequence weights,[11] which we then use to show that the probability that a given state is reached, $\Pr(s \mid \bar{\kappa})$, is given by a product distribution.

First, we prove an important structural lemma that shows that we can calculate the value of the game by summing the expected payoff at each state weighted by the probability that the state is reached. This combined with representing the probabilities of each state as a product distribution identifies the problem structure which we later exploit to obtain an efficient algorithm.

Define $\bar{x}_s$ to be a random vector giving the joint action taken at s when $s \in h$, that is, $\bar{x}_s(h) = \bar{x}'$ when the tuple $(s, \bar{x}')$ appears in h. When $s \notin h$, $\bar{x}_s(h)$ is undefined. Thus, when we use $\bar{x}_s$ in expectations or probabilities, we will always condition on the fact that $s \in h$.

Lemma 4.2.2. *For any joint policy $\bar{\kappa}$ and any player p, let*

$$R = \{s \mid s \in V,\ \Pr(s \mid \bar{\kappa}) > 0\}.$$

Then,

$$\mathcal{V}_p(\bar{\kappa}) = \sum_{s \in R} \Pr(s \mid \bar{\kappa}) E[M_p^s(\bar{x}_s) \mid s, \bar{\kappa}].$$

Proof. Define random variables $v_p^s : \mathcal{H} \to \mathbb{R}$ for $s \in V$, $p \in N$, by

$$v_p^s(h) = \begin{cases} M_p^s(\bar{x}_s(h)) & \text{if } s \in h \\ 0 & \text{otherwise} \end{cases}$$

[11] The name is by analogy to sequence weights in EFGs, see Koller and Megiddo [1992] and Koller et al. [1994] in particular.

Then,

$$E[v_p^s] = \Pr(s)E[v_p^s \mid s] + \Pr(\neg s)E[v_p^s \mid \neg s] = \Pr(s)E[v_p^s \mid s] = \Pr(s)E[M_p^s(\bar{x}_s) \mid s] \tag{4.3}$$

because $E[v_p^s(h) \mid \neg s] = 0$. Also, $V_p(h) = \sum_{s \in T} v_p^s(h)$ as each state occurs at most once in a given history (since T is a tree). Then ,

$$\mathcal{V}_p(\bar{\kappa}) = E[V_p] = E\left[\sum_{s \in V} v_p^s\right] = \sum_{s \in V} E\left[v_p^s\right] = \sum_{s \in V} \Pr(s \mid \bar{\kappa})\, E\left[M_p^s(\bar{x}_s) \mid s\right], \tag{4.4}$$

where we have used linearity of expectation and Equation (4.3). □

Now, we define the *sequence weight* $w_p(s \mid \kappa_p)$ of $s \in V$ for an arbitrary arbitrary policy κ_p for player p. Intuitively, the sequence weight $w_p(s \mid \kappa_p)$ is the probability we reach s given that all other players (and their randomness) "conspire" to force us to s. We formalize this notion in the proof of the next lemma, which then allows us to formally define sequence weights for a CEFG.

Lemma 4.2.3. *For any player p using policy κ_p and any two joint policies for the other players $\bar{\kappa}_{-p}$ and $\bar{\kappa}'_{-p}$, for any $s \in V_p$ where $\Pr(s \mid (\kappa_p, \bar{\kappa}_{-p})) > 0$ and $\Pr(s \mid (\kappa_p, \bar{\kappa}'_{-p})) > 0$, we have*

$$\Pr((s, x_p) \mid s, (\kappa_p, \bar{\kappa}_{-p})) = \Pr((s, x_p) \mid s, (\kappa_p, \bar{\kappa}'_{-p})).$$

Proof. Fix a CEFG G, a particular state $s \in V$, and a player p. We construct a single-player game $\mathcal{G}(p, s)$ as follows: The game includes the states $\mathsf{path}(s) = s^1, \ldots s^k$ (where s^1 is the root, and $s^k = s$). Let $S = \{s^1, \ldots, s^{k-1}\}$. The game starts at s^1. At each $s^i \in S$, if $p \notin \mathcal{A}(s^i)$, the game always continues to s^{i+1}. Thus, G is really only defined by the states $s^i \in V_p \cap S$. If $p \in \mathcal{A}(s^i)$, the player chooses an action $x \in X_{\phi_p(s)}$; with probability[12] $f_p^{s^i, s^{i+1}}(x)$ the game continues to s^{i+1}, and otherwise it stops. The game always ends if play reaches $s (= s^k)$.

Observe that we can use any policy κ_p for G to play $\mathcal{G}(p, s)$: at each s^i where $p \in \mathcal{A}(s^i)$, we tell the policy it is in information set $u = \phi_p(s^i)$, and it returns an action $x \in X_u$. In fact, there is no way for the policy κ_p to realize it is not being used to play G. Thus, each κ_p induces a probability distribution on histories of $\mathcal{G}(p, s)$ (a sequence of actions taken up until the end of the game). In particular, $\Pr_{\mathcal{G}(p,s)}(s^i \mid \kappa_p)$ is the probability that the game reaches s^i under κ^p.

[12] By Assumption (4.1.1), $f_p^{s^i, s^{i+1}}(x) \in [0, 1]$

We can also consider κ's action selection in $\mathcal{G}(p,s)$. In particular, if $\Pr_{\mathcal{G}(p,s)}(s^i \mid \kappa_p) > 0$, then $\Pr_{\mathcal{G}(p,s)}(x_p \mid s^i, \kappa_p)$ is also well defined. Whenever κ_p selects an action in $\mathcal{G}(p,s)$, it behaves exactly as if it were selecting an action in G; the lemma follows immediately. □

As a consequence of this lemma, we write $\Pr(x_p \mid s, \kappa_p)$ for this probability; it is defined for any state where $\Pr_{\mathcal{G}(p,s)}(s \mid \kappa_p) > 0$. We then define the sequence weight of s given κ_p by $w(s \mid \kappa_p) = \Pr_{\mathcal{G}(p,s)}(s \mid \kappa_p)$. Note that if the path to s contains no player p information sets then $w(s \mid \kappa_p) = 1$. When $w(s \mid \kappa_p) > 0$, we can then calculate it as

$$w(s \mid \kappa_p) = \Pr_{\mathcal{G}(p,s)}(s \mid \kappa_p) = \prod_{(t,t') \in \mathcal{E}(s)} \sum_{x \in X_u} \Pr(x \mid t, \kappa_p) f_p^{tt'}(x), \tag{4.5}$$

where we have $\Pr(1 \mid s) = 1$ and $f_p^{ss'}(1) = 1$ when $p \notin \mathcal{A}(s)$ (equivalently, we take the product to only be over edges (s, s') where $s \in V_p$). It is also useful to define

$$E[x \mid s, \kappa_p] = \sum_{x \in X_u} \Pr(x \mid s, \kappa_p)\, x$$

(when $w(s \mid \kappa_p) > 0$) and then using the linearity of f, we have for nonzero $w(s \mid \kappa_p)$,

$$w(s \mid \kappa_p) = \prod_{(t,t') \in \mathcal{E}(s)} f_p^{tt'}(E[x \mid t, \kappa_p]). \tag{4.6}$$

Define $\mathsf{REL}(\kappa_p) = \{s \mid w(s \mid \kappa_p) > 0\}$. This is exactly the set of states such that there exists a $\bar{\kappa}_{-p}$ such that $\Pr(s \mid (\kappa_p, \bar{\kappa}_{-p})) > 0$ (this can be proved based on the definition of the $\mathcal{G}(p,s)$ game and Assumption (4.2.1)). Any state $s \notin \mathsf{REL}(\kappa_p)$ is *ruled out* by κ_p: it is never reached when player p uses κ_p. We extend the REL notation to joint policies, by defining $\mathsf{REL}(\bar{\kappa}) = \cap_p \mathsf{REL}(\kappa_p) = \{s \mid \Pr(s \mid \bar{\kappa}) > 0\}$, and $\mathsf{REL}(\bar{\kappa}_{-p}) = \cap_{p' \neq p} \mathsf{REL}(\kappa_{p'})$.

Lemma 4.2.4. *For any $s \in V$ and any joint policy $\bar{\kappa}$,*

$$\Pr(s \mid \bar{\kappa}) = \prod_p w_p(s \mid \kappa_p).$$

Proof. First, observe that if for any p we have $w_p(s \mid \kappa_p) = 0$ then $s \notin \mathsf{REL}(\kappa_p)$ and $\Pr(s \mid \bar{\kappa}) = 0$, and so the equality holds. Now, suppose $\Pr(s \mid \bar{\kappa}) > 0$. If s is ever reached under $\bar{\kappa}$,

$$\Pr((s, \bar{x}) \mid s, \bar{\kappa}) = \prod_p \Pr(x_p \mid s, \kappa_p)$$

as a consequence of Lemma (4.2.3), and so for any reachable $s \in \mathsf{REL}(\bar{\kappa})$ with successor s',

$$\begin{aligned}
\Pr(s' \mid s, \bar{\kappa}) &= \sum_{\bar{x} \in \bar{X}_s} \Pr(\bar{x} \mid s, \bar{\kappa}) \Pr(s' \mid s, \bar{x}) \\
&= \sum_{\bar{x} \in \bar{X}_s} \left(\prod_p \Pr(x_p \mid s, \kappa_p) \right) \left(\prod_p f_p^{ss'}(x_p) \right) \\
&= \sum_{\bar{x} \in \bar{X}_s} \prod_p \Pr(x_p \mid s, \kappa_p)\, f_p^{ss'}(x_p). \\
&= \prod_p \sum_{x_p \in X_u} \Pr(x_p \mid s, \kappa_p)\, f_p^{ss'}(x_p). \qquad (4.7)
\end{aligned}$$

Thus,

$$\begin{aligned}
\Pr(s \mid \bar{\kappa}) &= \prod_{t,t' \in \mathcal{E}(s)} \Pr(t' \mid t, \bar{\kappa}) \\
&= \prod_{t,t' \in \mathcal{E}(s)} \prod_p \sum_{x_p \in X_{\phi_p(t)}} \Pr(x_p \mid t, \kappa_p) f_p^{tt'}(x_p). && \text{By Eq. (4.7)} \\
&= \prod_p \prod_{t,t' \in \mathcal{E}(s)} \sum_{x_p \in X_{\phi_p(t)}} \Pr(x_p \mid t, \kappa_p) f_p^{tt'}(x_p). \\
&= \prod_p w_p(s \mid \kappa_p) && \text{By Eq. (4.5)}
\end{aligned}$$

□

For convenience, for any joint policy $\bar{\kappa}$, we define $w(s \mid \bar{\kappa}) = \prod_p w_p(s \mid \kappa_p)$, and similarly for a policy $\bar{\kappa}_{-p}$ for all players other than p, we define $w(s \mid \bar{\kappa}_{-p}) = \prod_{p' \neq p} w_{p'}(s \mid \kappa_{p'})$. We now prove a lemma that is very useful in proving two policy classes are equivalent:

Lemma 4.2.5. *If κ_p and κ'_p are two policies for player p such that*

$$E[x_p \mid s, \kappa_p] = E[x_p \mid s, \kappa'_p]$$

for all $s \in \mathsf{REL}(\kappa_p) \cap \mathsf{REL}(\kappa'_p)$, then $\mathsf{REL}(\kappa_p) = \mathsf{REL}(\kappa'_p)$, and further κ_p and κ'_p are payoff equivalent.

Proof. Observe that $\mathsf{REL}(\kappa_p)$ and $\mathsf{REL}(\kappa'_p)$ are trees rooted at s^*: if $s \in \mathsf{REL}(\kappa_p)$, then all $s' \in \mathsf{path}(s)$ are also in $\mathsf{REL}(\kappa_p)$. Thus, $\mathsf{REL}(\kappa_p) \cap \mathsf{REL}(\kappa'_p)$ is also a tree, and so for $s \in \mathsf{REL}(\kappa_p) \cap \mathsf{REL}(\kappa'_p)$ we have $w_p(s \mid \kappa'_p) = w_p(s \mid \kappa_p)$ by Equation (4.6). Suppose $\mathsf{REL}(\kappa_p) \neq \mathsf{REL}(\kappa'_p)$. Then, WLOG, there exists $(s, s') \in E$ such that $s \in \mathsf{REL}(\kappa_p) \cap \mathsf{REL}(\kappa'_p)$, $s' \notin \mathsf{REL}(\kappa_p)$, and $s' \in \mathsf{REL}(\kappa'_p)$. Then, $w_p(s' \mid \kappa_p) = 0$ and $w_p(s' \mid \kappa'_p) > 0$. Equation (4.6) implies

$$w_p(s' \mid \kappa'_p) = w_p(s \mid \kappa'_p) \cdot f_p^{ss'}(E[x_p \mid s, \kappa'_p]).$$

Further, $w_p(s \mid \kappa'_p) = w_p(s \mid \kappa_p)$ because $s \in \mathsf{REL}(\kappa_p) \cap \mathsf{REL}(\kappa'_p)$ and $E[x_p \mid s, \kappa'_p] = E[x_p \mid s, \kappa_p]$, and we must have $w_p(s' \mid \kappa'_p) = w_p(s' \mid \kappa_p)$, a contradiction. Thus, we conclude $\mathsf{REL}(\kappa_p) = \mathsf{REL}(\kappa'_p)$.

Now, we proceed to show payoff equivalence. Fix any $\bar{\kappa}_{-p}$ for the other players. Equation (4.6) shows κ_p and κ'_p have equal sequence weights, and so by Lemma (4.2.4), $\Pr(s \mid (\kappa_p, \bar{\kappa}_{-p})) = \Pr(s \mid (\kappa'_p, \bar{\kappa}_{-p}))$ for all s.

Using Lemma (4.2.2), it is now sufficient to show that the expected payoff for an arbitrary player $q \in N$ at each state reached with positive probability is equal. This is clearly true at states where $p \notin \mathcal{A}(s)$. Consider some $s \in V_p$ where $s \in u$. The key is that the payoff function M_q^s is multi-linear. Let $\bar{x}_{-p}$ be a joint action for all players other than p, so that for any $x_p \in X_u$, $(x_p, \bar{x}_{-p}) \in \bar{X}_s$ is a joint action at s. Then, multi-linearity implies there exists a vector $\vec{m}_s(\bar{x}_{-p}) \in \mathbb{R}^{n_u}$ such that for any $x_p \in X_u$,

$$M_q^s((x_p, \bar{x}_{-p})) = \vec{m}_s(\bar{x}_{-p}) \cdot x_p.$$

Now, for κ_p and any s with $\Pr(s \mid (\kappa_p, \bar{\kappa}_{-p})) > 0$,

$$\begin{aligned}
E\left[M_q^s(\bar{x}_s) \mid s, (\kappa_p, \bar{\kappa}_{-p})\right] &= \sum_{\bar{x} \in \bar{X}_s} \Pr(\bar{x} \mid s) M_q^s(\bar{x}) \\
&= \sum_{\bar{x}_{-p}} \sum_{x_p \in X_u} \Pr(\bar{x}_{-p} \mid s, \bar{\kappa}_{-p}) \Pr(x_p \mid s, \kappa_p) M_q^s(\bar{x}) \\
&= \sum_{\bar{x}_{-p}} \Pr(\bar{x}_{-p} \mid s, \bar{\kappa}_{-p}) \sum_{x_p \in X_u} \Pr(x_p \mid s, \kappa_p)(\vec{m}_s(\bar{x}_{-p}) \cdot x_p) \\
&= \sum_{\bar{x}_{-p}} \Pr(\bar{x}_{-p} \mid s, \bar{\kappa}_{-p}) \left(\vec{m}_s(\bar{x}_{-p}) \cdot \sum_{x_p \in X_u} \Pr(x_p \mid s, \kappa_p)\, x_p \right) \\
&= \sum_{\bar{x}_{-p}} \Pr(\bar{x}_{-p} \mid s, \bar{\kappa}_{-p}) \left(\vec{m}_s(\bar{x}_{-p}) \cdot E[x_p \mid s, \kappa_p] \right).
\end{aligned}$$

Since $E[x_p \mid s, \kappa_p] = E[x_p \mid s, \kappa'_p]$ at the relevant states, it follows that

$$E\left[M_q^s(\bar{x}_s) \mid s, (\kappa_p, \bar{\kappa}_{-p})\right] = E\left[M_q^s(\bar{x}_s) \mid s, (\kappa'_p, \bar{\kappa}_{-p})\right],$$

and so by Lemma (4.2.2), we conclude κ_p and κ'_p are payoff equivalent. □

Sufficient Recall

We say a CEFG has *sufficient recall* if for all players p it has both:

- *observation memory*: For all $u \in U_p$, and all $s, s' \in u$, $\mathsf{obs}_p(s) = \mathsf{obs}_p(s')$. That is, the information sets for p form a forest.
- *action memory*: For any two policies κ_p and κ'_p for p, and any policy $\bar{\kappa}_{-p}$ for the other players, and for any $u \in U_p$ with $\Pr(u \mid (\kappa_p, \bar{\kappa}_{-p})) > 0$ and $\Pr(u \mid (\kappa'_p, \bar{\kappa}_{-p})) > 0$, and any $s \in u$, we have

$$\Pr(s \mid u, (\kappa_p, \bar{\kappa}_{-p})) = \Pr(s \mid u, (\kappa'_p, \bar{\kappa}_{-p})).$$

It is worth emphasizing that both observation memory and action memory are properties of the game itself, not of players or policies.

Observation memory says that the current information set uniquely specifies the sequence of information sets (which we can view as the history of observations) that have previously occurred; hence player p has no incentive to remember the information sets visited. Action memory implies that if we know the current information set is u, then remembering the policy we followed up until we reached u provides no information about the actual $s \in u$. Thus, the player need not remember the policy followed so far. The name sufficient action memory might be more appropriate, as the exact exact actions taken at past information sets are not remembered.

Informally, then, if the game has sufficient recall for player p, then player p should be able to play optimally by selecting an action purely as a function of the current information set, as from this all relevant past actions and observations can be derived. We use this dual characterization of sufficient recall because this intuition seems so clear.

However, to formally prove that implicit behavior reactive policies are "strong enough" to play sufficient-recall CEFGs optimally, we will introduce *sequence recall*, an alternative characterization of sufficient recall that makes establishing certain structural lemmas more natural. Further, sequence recall can be viewed as a generalization of perfect recall as it is usually defined for EFGs. Before introducing sequence recall, we need to define the notion of the "outcome" for a player at a state.

Generalized outcomes In an EFG, all states in an information set u have the same out degree d, and each outgoing edge from some $s \in u$ is labeled with one of d outcome or choice labels. Thus, the action set in an EFG is the set of outcome labels. We can view the choice labels in an EFG as partitioning all of the edges out of u into d different equivalence classes based on the labels.

In a CEFG, nodes in u may have different out-degrees, and the successor of s is chosen from a product distribution that is a function of each players' action, using the functions $f_p^{ss'}$. Thus, we will need a more complex partition. We define a partition on the edges out of $u \in U_p$ via an equivalence relation $\sim_p$ on pairs of edges. For any two edges (s, s') and (t, t') out of u (e.g., $s, t \in u$), we have $(s, s') \sim_p (t, t')$ if and only if there exists a constant $\alpha \geq 0$ such that for all $x \in X_u$

$$f_p^{ss'}(x) = \alpha f_p^{tt'}(x). \tag{4.8}$$

Let O_u be the set of such equivalence classes at u defined by $\sim_p$, so $o \in O_u$ is a maximal set of edges such that any pair of edges in o satisfies Equation (4.8), and $\bigcup_{o \in O_u} o$ is the set of all edges out of u. In fact, if we "normalize" the CEFG in the manner suggested by the next lemma, we can assume that $\alpha = 1$ in Equation (4.8) without loss of generality.

Lemma 4.2.6. *For any CEFG G, there exists an f-equivalent CEFG G' such that if $(s, s') \sim_p (t, t')$ in G, then for all $x \in X_u$, in G'*

$$g_p^{ss'}(x) = g_p^{tt'}(x),$$

where we use g to denote the f-functions in G'.

Proof. It is sufficient to show the transformation on pairs of edges. Suppose G has edges (s, s') and (t, t') out of u that fall into the same partition, but the corresponding f-functions are not equal. Then there must exist some α such that

$$f_p^{ss'}(x) = \alpha f_p^{tt'}(x).$$

WLOG, $\alpha <= 1$ (if not, divide both sides by α and take $\alpha' = 1/\alpha$).

Even if G has an "inactive" random player (that is, $f_0^{ss'} = 1$ for all states s and s' in G), G' will have an active one. We write $g_0^{ss'}$ for the constant f-functions of the random player in G'. The f-functions in G' are the same as in G (in particular, $g_p^{tt'} = f_p^{tt'}$), except we set $g_p^{ss'}(x) \leftarrow f_p^{tt'}(x)$, and $g_0^{ss'} = \alpha f_0^{ss'}$. Now, in G' the f-functions on (t, t') and (s, s') are identical (satisfy Equation (4.8) with $\alpha = 1$), as we have "moved" the constant difference

α into the randomness on the (s, s') edge. That is,

$$\begin{aligned}\Pr_G(s' \mid \bar{x}, s) &= f_0^{ss'} f_1^{ss'}(x_1) \cdots f_p^{ss'}(x_p) \cdots \\ &= f_0^{ss'} f_1^{ss'}(x_1) \cdots \alpha f_p^{tt'}(x_p) \cdots \\ &= (\alpha f_0^{ss'}) f_1^{ss'}(x_1) \cdots f_p^{tt'}(x_p) \cdots \\ &= g_0^{ss'} g_1^{ss'}(x_1) \cdots g_p^{ss'}(x_p) \cdots \\ &= \Pr_{G'}(s' \mid \bar{x}, s),\end{aligned}$$

and so transition probabilities in the two games are identical. □

We call CEFGs that have been maximally transformed using Lemma (4.2.6) *f-normalized* ; such CEFGs satisfy

$$(s, s') \sim_p (t, t') \quad \Rightarrow \quad f_p^{ss'} = f_p^{tt'}. \tag{4.9}$$

For the remainder of this paper, we assume all CEFGs are f-normalized. Under this assumption, we write $f_p^{u,o}$ for the f function shared by all edges out of u in outcome partition $o \in O_u$.

Sequence recall Using this notion of outcome, we can now define the player p *sequence* $\sigma_p(s)$ associated with a state s. The sequence $\sigma_p(s)$ is the list of player p's information sets and outcomes on the unique path in T to s. Edges from states s where $p \notin \mathcal{A}(s)$ do not appear in the sequence $\sigma_p(s)$. We write:

$$\sigma_p(s) = ((u^1, o^1), (u^2, o^2), \ldots, (u^k, o^k)).$$

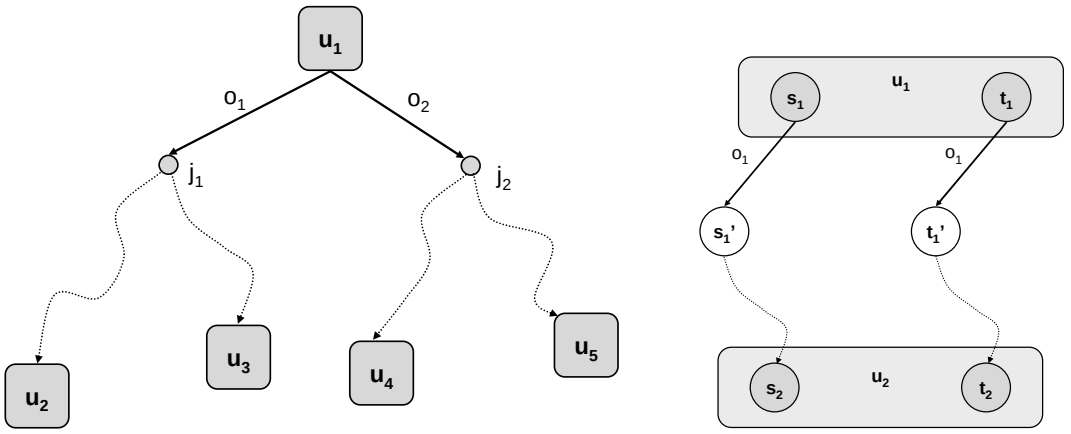
In general, we can view $\sigma_p(s)$ as a refinement of $\mathsf{obs}_p(s)$: two states $s, s' \in u$ might have the same observation history, but different sequences. We say a CEFG has sequence recall for player p, if for all $u \in V_p$ and all $s, s' \in u$, $\sigma_p(s) = \sigma_p(s')$. Note that sequence recall immediately implies observation memory. In fact, sequence recall and sufficient recall are equivalent. Before proving this result, we establish that action-selection probabilities in sequence-recall CEFGs satisfy the following structural property:

Lemma 4.2.7. *Suppose G is an f-normalized CEFG where $\sigma_p(s) = \sigma_p(s')$ for some $s, s' \in u, u \in U_p$. For any policy κ_p for player p,*

$$w(s \mid \kappa_p) = w(s' \mid \kappa_p),$$

and when $w(s \mid \kappa_p) > 0$, for any$x \in X_u$,

$$\Pr(x \mid s, \kappa_p) = \Pr(x \mid s', \kappa_p).$$



(a) An information set forest.

(b) The part of the game tree corresponding to u_1 and u_2.

Figure 4.2: An example CEFG.

Proof. The proof follows immediately from the definition of $w(s \mid \kappa_p)$ and $\Pr(x \mid s, \kappa_p)$ solely in terms of the f-functions on $\mathcal{E}(s)$ (Lemma (4.2.3) and Equation (4.5)).

In particular, recall that the sequence weights and action probabilities are defined in terms of the one-player games $\mathcal{G}(p, s)$ and $\mathcal{G}(p, s')$. Since $\sigma_p(s) = \sigma_p(s')$, by the definition of sequence both games pass through the same information sets in the same order, and (because the of sequence recall and the f-normalized assumption), the $f^{s^i, s^{i+1}}$ functions that determine if the game continues are also identical. Hence, the games $\mathcal{G}(p, s)$ and $\mathcal{G}(p, s')$ are equivalent; since we define $\Pr(x \mid s', \kappa_p)$ and $w(s \mid \kappa)$ in terms of these games, the lemma follows. □

Corollary 4.2.8. *In an f-normalized CEFG with sequence recall, for any policy κ_p for player p, any $u \in U_p$, and any $s, s' \in u$ and $x \in X_u$, then $w(s \mid \kappa_p) = w(s' \mid \kappa_p)$, and when $w(s \mid \kappa_p) > 0$, $\Pr(x \mid s, \kappa_p) = \Pr(x \mid s', \kappa_p)$.*

Corollary (4.2.8) reveals the significant structure of sequence weights in CEFGs with sequence recall. This structure is basically identical to the structure of the sequence weights in perfect-recall EFGs, hence justifying our adoption of the term "sequence weights" for the $w_p(s \mid \kappa_p)$ values. The principal results and associated notation are given below; they are stated using the relationships of the states and information sets of Figure 4.2, but hold in general. For CEFGs with sequence recall we extend our notation for sequence weights and write $w(u \mid \kappa_p) = w(s \mid \kappa_p)$ for any $s \in u$.

- Each non-root information set u_2 for player p has a unique (information set, out-

come) "parent": if u_1 is the unique predecessor of u_2 in the information set forest, then any path from u_1 to u_2 must begin with an edge in some fixed outcome, say o_1. Let $\mathsf{upred}_p(u_2) = (u_1, o_1)$ identify this parent; if u is a root information set, we write $\mathsf{upred}_p(u) = \emptyset$. This situation is shown in the information set forest of Figure (4.2)(a).

- Any state s occurring after some player p information set (that is, with a non-empty $\sigma_p(s)$) has a unique (information set, outcome) predecessor, namely the last tuple in $\sigma_p(s)$. We extend the upred notation to this case, so for example $\mathsf{upred}(s_2) = (u_1, o_1)$ (see Figure (4.2)(b)). It follows that in fact

$$w(s_2 \mid \kappa_p) = w(\mathsf{upred}(s_2) \mid \kappa_p) = w(u_1, o_1 \mid \kappa_p).$$

- Any state s occurring before any player p information set has $w(s \mid \kappa_p) = 1$, for example $w(s_1 \mid \kappa_p) = 1$.

- Consider the (partial) game tree shown in Figure 4.2(b). We have $u_1 = \{s_1, t_1\}$, and $u_2 = \{s_2, t_2\}$. State s_1 has successor s_1', and t_1 has successor t_1'. These edges are in the same outcome partition o_1, so $f_p^{s_1,s_1'} = f_p^{t_1,t_1'}$. Thus Corollary (4.2.8) implies that $w(s_1' \mid \kappa_p) = w(t_1' \mid \kappa_p)$. In general, any immediate successor state of u reached via an edge in a fixed outcome partition o must have the same sequence weight; we write $w(u, o \mid \kappa_p)$ for this value.

In summary, for any node $s \in u_2$ where $(u_1, o_1) = \mathsf{upred}_p(u_2)$, we write any of the following equivalently:

$$w_p(\sigma_p(s) \mid \kappa_p) = w_p(s \mid \kappa_p) = w_p(u_2 \mid \kappa_p) = w_p(u_1, o_1 \mid \kappa_p). \tag{4.10}$$

Sequence recall equals sufficient recall We now turn to proving that sequence recall and sufficient recall are equivalent. We will need the following two Lemmas:

Lemma 4.2.9. *Suppose that $e_1 = (s, s')$ and $e_2 = (t, t')$ with $s, t \in u$ are in different outcomes for player p, that is, $(s, s') \not\sim_p (t, t')$. Then, there exist $x_1, x_2 \in X_u$ such that*

$$\frac{f_p^{ss'}(x_1)}{f_p^{tt'}(x_1)} \neq \frac{f_p^{ss'}(x_2)}{f_p^{tt'}(x_2)}.$$

Proof. If e_1 and e_2 are in different outcomes, then for any constants $\alpha > 0$, there exists an $x \in X_u$ such that

$$f_p^{ss'}(x) \neq \alpha f_p^{tt'}(x).$$

Fix $\alpha = 1$, and let $x_1 \in X_u$ such that $f_p^{ss'}(x_1) \neq \alpha f_p^{tt'}(x_1)$. Now, use $\beta = f_p^{ss'}(x_1)/f_p^{tt'}(x_1) \neq 1$ as the constant, and let x_2 such that

$$f_p^{ss'}(x_2) \neq \beta f_p^{tt'}(x_2). \tag{4.11}$$

Dividing both sides of Equation (4.11) by $f_p^{tt'}(x_2)$ and using the definition of β yields the Lemma. □

The next lemma shows that the ratio $\Pr(s)/\Pr(s')$ for $s, s' \in u$ does not depend on player p's policy.

Lemma 4.2.10. *A CEFG has action memory for player p if and only if for all $u \in U_p$, any two policies κ_p and κ'_p for p, and any joint policy $\bar{\kappa}_{-p}$ for the other players, for any $s, s' \in u$ with $s' \in \mathsf{REL}(\kappa_p, \bar{\kappa}_{-p})$,*

$$\frac{\Pr(s \mid (\kappa_p, \bar{\kappa}_{-p}))}{\Pr(s' \mid (\kappa_p, \bar{\kappa}_{-p}))} = \frac{\Pr(s \mid (\kappa'_p, \bar{\kappa}_{-p}))}{\Pr(s' \mid (\kappa'_p, \bar{\kappa}_{-p}))}. \tag{4.12}$$

Proof. Let $a_i = \Pr(i \mid (\kappa_p, \bar{\kappa}_{-p}))$ for $i \in u$, and let $b_i = \Pr(i \mid (\kappa'_p, \bar{\kappa}_{-p}))$ for $i \in u$. Observe that $\Pr(u \mid (\kappa_p, \bar{\kappa}_{-p})) = \sum_i a_i$, and similarly for b_i.

Fix $s, s' \in u$ with $s' \in \mathsf{REL}(\kappa_p, \bar{\kappa}_{-p})$. If $b_{s'} = \Pr(s' \mid (\kappa'_p, \bar{\kappa}_{-p})) = 0$, then Equation (4.12) fails to hold because $s' \in \mathsf{REL}(\kappa_p, \bar{\kappa}_{-p})$ implies $\Pr(s' \mid (\kappa_p, \bar{\kappa}_{-p}) > 0$; it is also easy to show that action memory does not hold in this case, and so the lemma holds. We now consider the case where $b_{s'} > 0$. Action memory is exactly the condition that

$$\frac{a_s}{\sum_i a_i} = \frac{b_s}{\sum_i b_i} \tag{4.13}$$

for all $s \in u$. Inverting both sides of Equation (4.13) for s' we have $(\sum_i a_i)/a_{s'} = (\sum_i b_i)/b_{s'}$. Multiplying the left-hand-side of this equality with the left-hand-side Equation (4.13), and similarly the right with the right, gives

$$\frac{a_s}{a_{s'}} = \frac{b_s}{b_{s'}},$$

which is exactly the claimed equality. □

Now, we can prove the main theorem:

Theorem 4.2.11. *A CEFG has sufficient recall for player p if an only if it has sequence recall for player p.*

Proof. For the first direction, assume the game has sequence recall. Then, for any $s, s' \in u$ with $s' \in \mathsf{REL}(\kappa_p, \bar{\kappa}_{-p})$, and any policy κ_p for player p, and any joint history policy $\bar{\kappa}_{-p}$ for the other players, we have

$$\frac{\Pr(s \mid (\kappa_p, \bar{\kappa}_{-p}))}{\Pr(s' \mid (\kappa_p, \bar{\kappa}_{-p}))} = \frac{w_p(s \mid \kappa_p) w_p(s \mid \bar{\kappa}_{-p})}{w_p(s' \mid \kappa_p) w_p(s' \mid \bar{\kappa}_{-p}))} = \frac{w_p(s \mid \bar{\kappa}_{-p})}{w_p(s' \mid \bar{\kappa}_{-p}))}$$

since $w_p(s \mid \kappa_p) = w_p(s' \mid \kappa_p)$ by Corollary (4.2.8). Since the right hand side of the equality does not depend on κ_p, we conclude Equation (4.12) holds, and so by Lemma (4.2.10), we have action memory. Observation memory is immediate from sequence recall.

For the other direction, assume the game has sufficient recall for player p. Assume for contradiction there exists an information set u_2 where sequence recall does not hold: there exist $s_2, t_2 \in u_2$ such that $\sigma_p(s_2) \neq \sigma_p(t_2)$. Both s_2 and t_2 share a predecessor information set u_1 (observation recall holds, and the observation history cannot be empty or their sequences would agree). Let s_1 be the state in u_1 on the path to s_2, and let s_1' be s_1's successor on the path (possibly $s_1' = s_2$), so (s_1, s_1') is an edge. Similarly identify an edge (t_1, t_1') out of u_1 on the path to t_2. Without loss of generality, assume $\sigma_p(s_1) = \sigma_p(t_1)$ (if this doesn't hold immediately, let $s_2 \leftarrow s_1$ and $t_2 \leftarrow t_1$, and continue until this process as needed). This situation is shown in Figure (4.2b).

Since $\sigma_p(s_1) = \sigma_p(t_1)$, but $\sigma_p(s_2) \neq \sigma_p(t_2)$, then these two sequences must differ on the last outcome (the outcome from u_1), that is, $(s, s') \not\sim_p (t, t')$. By Lemma (4.2.9) there exist an $x_1, x_2 \in X_u$ such that

$$\frac{f_p^{s_1 s_1'}(x_1)}{f_p^{t_1 t_1'}(x_1)} \neq \frac{f_p^{s_1 s_1'}(x_2)}{f_p^{t_1 t_1'}(x_2)}. \tag{4.14}$$

Let π_1 be any pure reactive policy with $w_p(s_1 \mid \pi_1) > 0$ and $\pi(u_1) = x_1$, and let π_2 be the same as π_1, except that $\pi_2(u_1) = x_2$. Then, fix any $\bar{\kappa}_{-p}$ with $t_2 \in \mathsf{REL}(\bar{\kappa}_{-p})$, and let $B = w(s_2 \mid \bar{\kappa}_{-p}) / w(t_2 \mid \bar{\kappa}_{-p})$, so

$$\frac{\Pr(s_2 \mid (\pi_1, \bar{\kappa}_{-p}))}{\Pr(t_2 \mid (\pi_1, \bar{\kappa}_{-p}))} = \frac{w_p(s_2 \mid \pi_1) w(s_2 \mid \bar{\kappa}_{-p})}{w_p(t_2 \mid \pi_1) w(t_2 \mid \bar{\kappa}_{-p})} = B \frac{w_p(s_1 \mid \pi_1) f_p^{s_1 s_1'}(x_1)}{w_p(t_1 \mid \pi_1) f_p^{t_1 t_1'}(x_1)} = B \frac{f_p^{s_1 s_1'}(x_1)}{f_p^{t_1 t_1'}(x_1)}$$

since $\sigma_p(s_1) = \sigma_p(t_1)$ and so by Lemma (4.2.7) $w_p(s_1 \mid \pi) = w_p(t_1 \mid \pi)$. By an analogous argument,

$$\frac{\Pr(s_2 \mid (\pi_2, \bar{\kappa}_{-p}))}{\Pr(t_2 \mid (\pi_2, \bar{\kappa}_{-p}))} = B \frac{f_p^{s_1 s_1'}(x_2)}{f_p^{t_1 t_1'}(x_2)}, \tag{4.15}$$

and so Equation (4.14) implies.

$$\frac{\Pr(s_2 \mid (\pi_1, \bar{\kappa}_{-p}))}{\Pr(t_2 \mid (\pi_1, \bar{\kappa}_{-p}))} \neq \frac{\Pr(s_2 \mid (\pi_2, \bar{\kappa}_{-p}))}{\Pr(t_2 \mid (\pi_2, \bar{\kappa}_{-p})).}$$

Thus, by Lemma (4.2.10) action memory does not hold at u_2, contradicting the assumption that the game has sufficient recall for p, and so we conclude sequence recall holds for all player p information sets. □

Based on Theorem (4.2.11), we can apply the notation from Equation (4.10) to sufficient recall games.

A payoff equivalence theorem Now we can give this sections principal result: implicit behavior reactive policies are payoff equivalent to general policies in sufficient-recall CEFGs. This is critical, as our optimization technique will let us find the best implicit behavior reactive policy.

Theorem 4.2.12. *For sufficient-recall CEFGs, for any policy κ_p for player p, there exists a payoff equivalent implicit behavior reactive policy.*

Proof. Let κ_p be an arbitrary policy for p. A consequence of Lemma (4.2.7) is that for all $s, s' \in u$, when $s, s' \in \mathsf{REL}(\kappa_p)$,

$$E[x_p \mid s, \kappa_p] = E[x_p \mid s', \kappa_p].$$

Call this value x_u for each u where it is defined (e.g., where $\exists s \in u$ such that $s \in \mathsf{REL}(\kappa)$), and pick x_u arbitrarily in X_u for the remaining $u \in U_p$. Then, we define an implicit behavior policy β_κ by $\beta_\kappa(u) = x_u$. These two policies must play the same action in expectation at any state s where $w(s \mid \kappa_p) > 0$ and $w(s \mid \beta) > 0$. Thus, by Lemma (4.2.5) they are payoff equivalent.

□

Theorem (4.2.12) shows that when playing sufficient recall CEFGs, it suffices to consider only implicit behavior reactive policies. In the next section we show that for two-player zero-sum sufficient recall CEFGs, the set of IBRPs for each player can be represented as a convex set $\mathcal{W}$ in such a way that the value of the game is multi-linear in $\mathcal{W}$. Thus, we can solve zero-sum sufficient-recall CEFGs using linear programming on the convex game defined by the sets $\mathcal{W}$ and corresponding multi-linear objective function.

4.3 Solving a CEFG by Transformation to a Convex Game

We consider a zero-sum CEFG with two players, x and y, and possibly a random player 0. To differentiate the two players, we use u and X_u to denote player x's information and action sets, and similarly v and Y_v for y. In the two-player, zero-sum case the costs at a node s where both players play is specified via a payoff matrix M^s of dimension $n_u \times n_v$; the payoff from x to y is then is then $x^T M^s y$ when x plays x and y plays y. Since $X_{\diamondsuit_p} = \{1\}$, we use this same notation to indicate payoffs where only one player selects an action; in this case the payoff is the dot product of a cost vector with the action of the active player. The random player, if present, does not affect payoffs directly, and so this notation still applies in the presence of a random player. In fact, the random player only affects the game through her sequence weights, which we write as $w_0(s)$, because the random player has no policy.

An IBRP for player x can be viewed as a vector from the convex set

$$\tilde{X} = \bigotimes_{u \in U_{\mathsf{x}}} X_u.$$

The set $\tilde{X}$ is a Cartesian product of convex sets, and so it is also a convex set. Define $\tilde{Y}$ analogously for y, and let $\beta_{\mathsf{x}} \in \tilde{X}$ and $\beta_{\mathsf{y}} \in \tilde{Y}$ be two IBRPs. Let $u = \phi_{\mathsf{x}}(s)$ and $v = \phi_{\mathsf{y}}(s)$, and define

$$\begin{aligned} \mathcal{V}(s) &= \Pr(s \mid (\beta_{\mathsf{x}}, \beta_{\mathsf{y}}))\ E[M^s(\bar{x}_s) \mid s, (\beta_{\mathsf{x}}, \beta_{\mathsf{y}})] \\ &= w_0(s) w(s \mid \beta_{\mathsf{x}})\, w(s \mid \beta_{\mathsf{y}})\ \left[\beta_{\mathsf{x}}(u)^T\, M^s\, \beta_{\mathsf{y}}(v)\right] \end{aligned} \tag{4.16}$$

using Lemma (4.2.4). The expected payoff from x to y is

$$\mathcal{V} = \sum_{s \in \mathsf{REL}(\beta_{\mathsf{x}}, \beta_{\mathsf{y}})} \mathcal{V}(s)$$

by Lemma (4.2.2). Unfortunately, $\mathcal{V}(s)$ is not bilinear in β_{x} and β_{y}, as $w(s \mid \beta_{\mathsf{x}})$ is a product of $f^{tt'}(\beta_{\mathsf{x}}(\phi_{\mathsf{x}}(s))$ terms along the path to s, and each of these terms is a linear function of β_{x}. Further, each term contains both the $w(s \mid \beta_{\mathsf{x}})$ term and the $\beta_{\mathsf{x}}(\phi_{\mathsf{x}}(s))$, so even if $w(s \mid \beta_{\mathsf{x}})$ wasn't nonlinear, $\beta_{\mathsf{x}}(\phi_{\mathsf{x}}(s)) w(s \mid \beta_{\mathsf{x}})$ would be.

We now develop an alternative convex representation for IBRPs in which $\mathcal{V}(s)$ is bilinear. Our use of sequence weights as variables is analogous to the technique in Koller et al. [1994], but our approach must also represent the implicit behavior taken at each X_u, as this is not defined by the sequence weights alone. More precisely, we construct a set $\mathcal{W}_{\mathsf{x}}$ such that there is a (nonlinear) bijection between $\mathcal{W}_{\mathsf{x}}$ and $\tilde{X}$, so each vector in $\omega_{\mathsf{x}} \in \mathcal{W}_{\mathsf{x}}$ has a natural interpretation as an IBRP. Further, the value $\mathcal{V}$ of the game is linear in ω_{x} for a fixed policy for y.

The sequence form of CEFGs We describe the policy representation set $\mathcal{W}_{\mathsf{x}}$ for player x, it is analogous for y. Our construction of the set $\mathcal{W}_{\mathsf{x}}$ relies on the sets

$$X_u^c = \{(\alpha x, \alpha) \mid x \in X_u, \alpha \geq 0\} \subseteq \mathbb{R}^{n_u+1}$$

for each $u \in \phi_{\mathsf{x}}$. The set X_u^c is the *cone extension* of X_u, and it is also convex [see Boyd and Vandenberghe, 2004, Sec. 2.1.5]; in fact, if X_u is a polyhedron (defined by a finite number of linear equalities and inequalities), then so is X_u^c; see Appendix (B). We will treat elements of X_u^c as tuples, writing $(x_u^c, \alpha) \in X_u^c$ where $x_u^c \in \mathbb{R}^{n_u}$ and $\alpha \in \mathbb{R}$.

Define

$$\tilde{X}^c = \bigotimes_{u \in U_{\mathsf{x}}} X_u^c.$$

We will have $\mathcal{W}_{\mathsf{x}} \subseteq \tilde{X}^c$. We work with a vector $\omega_{\mathsf{x}} \in \tilde{X}^c$ by writing $\omega_{\mathsf{x}} = \langle (x_u^c, w_u) \mid u \in U_{\mathsf{x}} \rangle$, where the $x_u^c \in \mathbb{R}^{n_u}$ and $w_u \in \mathbb{R}$ variables are defined for all $u \in U_{\mathsf{x}}$ by ω_{x}.

The set $\mathcal{W}_{\mathsf{x}}$ is defined by the following constraints:

$$(x_u^c, w_u) \in X_u^c \tag{4.17}$$

$$w_u = 1 \qquad \forall u \in U_{\mathsf{x}} \text{ with } \mathsf{upred}_p(u) = \emptyset \tag{4.18}$$

$$w_u = f_{\mathsf{x}}^{u',o'} \cdot x_{u'}^c \qquad \forall u \in U_{\mathsf{x}} \text{ with } \mathsf{upred}_p(u) = (u', o'). \tag{4.19}$$

We write $f_{\mathsf{x}}^{u',o'}(x_{u'}^c)$ as $f_{\mathsf{x}}^{u',o'} \cdot x_{u'}^c$ to emphasize the linearity of the f functions. The set $\mathcal{W}_{\mathsf{x}}$ is convex as $\tilde{X}^c$ is convex and the constraints are linear.

First, we show $\mathcal{W}_{\mathsf{x}}$ is in 1-1 correspondence with a set of IBRP policies (represented as elements in $\tilde{X}$). We do not consider the full set $\tilde{X}$ for technical reasons: A behavior policy $\beta \in \tilde{X}$ can be "over-specified," in that β defines an action $\beta(u) \in X_u$ even when $w(u \mid \beta) = 0$ (and hence u cannot possibly be reached when playing β). For each $u \in U_{\mathsf{x}}$, pick an arbitrary action $\acute{x}_u \in X_u$. We define the function $J : \tilde{X} \to \tilde{X}$ which we use to specify a canonical representation of behavior policies. Define

$$\acute{J}(\beta)(u) = \begin{cases} \beta(u) & \text{when } w(u \mid \beta) > 0 \\ \acute{x}_u & \text{otherwise} \end{cases}$$

so that $\acute{J}(\tilde{X}) = \{\acute{J}(\beta) \mid \beta \in \tilde{X}\}$. For any $\beta \in \tilde{X}$, the policies β and $\acute{J}(\beta)$ play the same action at all information states possibly reached, and so must be payoff equivalent. Hence, optimizing over $\acute{J}(\tilde{X})$ is equivalent to optimizing over $\tilde{X}$.

We now show a bijection g between $\mathcal{W}_{\mathsf{x}}$ and $\acute{J}(\tilde{X})$, defined by $g(\omega_{\mathsf{x}}) = \beta_{\omega_{\mathsf{x}}}$ where $\beta_{\omega_{\mathsf{x}}} \in \tilde{X}$ is the IBRP defined by

$$\beta_{\omega_{\mathsf{x}}}(u) = \begin{cases} (1/w_u) x_u^c & \text{when } w_u > 0 \\ \acute{x}_u & \text{otherwise.} \end{cases}$$

Sense $(x_u^c, w_u) \in X_u^c$, it follows from the definition of cone extension that $(1/w_u)x_u^c \in X_u$, and so β is a valid IBRP and so g is well-defined. Next, we prove g is a bijection:

Theorem 4.3.1. *The function g is a bijection between $\mathcal{W}_{\mathsf{x}}$ and $\acute{J}(\tilde{X})$.*

Proof. We need to show g is 1-1 and onto.

To show g is onto $\acute{J}(\tilde{X})$, consider an arbitrary $\beta \in \acute{J}(\tilde{X})$. Define a ω_{x} by $w_u = w(u \mid \beta)$ and $x_u^c = w(u \mid \beta)\beta(u)$. It follows from the definition of g and $\acute{J}$ that $g(\omega_{\mathsf{x}}) = \beta$. It remains to $\omega_{\mathsf{x}} \in \mathcal{W}_{\mathsf{x}}$. First, it is straightforward to verify that Constraints (4.17) and (4.18) are satisfied. For Constraint (4.19), let $(u', o') = \mathsf{upred}_p(u)$, and observe that

$$w_u = w(u \mid \beta) = w(u', o' \mid \beta) = w(u' \mid \beta) f^{u',o'}(\beta(u)) = f^{u',o'} \cdot w(u' \mid \beta)\beta(u) = f^{u',o'} \cdot x_{u'}^c.$$

For 1-1, suppose $\omega_{\mathsf{x}} = \langle (x_u^c, w_u) \rangle$ and $\omega'_{\mathsf{x}} = \langle (y_u^c, v_u) \rangle$ in $\mathcal{W}_{\mathsf{x}}$, such that $\omega_{\mathsf{x}} \neq \omega'_{\mathsf{x}}$, but $g(\omega_{\mathsf{x}}) = g(\omega'_{\mathsf{x}})$. Let $\beta = g(\omega_{\mathsf{x}})$ and $\beta' = g(\omega'_{\mathsf{x}})$. WLOG, let u be an information set where $(x_u^c, w_u) \neq (y_u^c, v_u)$, but for all earlier information sets ω_{x} and ω'_{x} agree. Then, w_u and v_u must be equal by Constraint (4.19), and so x_u^c and y_u^c must differ. However, this implies β and β' must play differently at u, a contradiction. □

When we refer to elements of $\mathcal{W}_{\mathsf{x}}$ as policies, we mean the corresponding IBRP given by the bijection g. Now, we show that payoffs are bilinear in the $\mathcal{W}_{\mathsf{x}}$ representation.

Theorem 4.3.2. *In a two-player, zero-sum, sufficient recall CEFGs, represent x's IBRPs as $\mathcal{W}_{\mathsf{x}}$, and player y's IBRPs as $\mathcal{W}_{\mathsf{y}}$. Then, for any $\omega_{\mathsf{x}} \in \mathcal{W}_{\mathsf{x}}$ and $\omega_{\mathsf{y}} \in \mathcal{W}_{\mathsf{y}}$, the payoff $\mathcal{V}(g(\omega_{\mathsf{x}}), g(\omega_{\mathsf{y}}))$ is a bilinear function of ω_{x} and ω_{y}.*

Proof. Equation (4.16) shows the payoff is a sum over states that are reached with positive probability under ω_{x} and ω_{y}. It is sufficient to show that the payoff term for each state is bilinear.

Let $\omega_{\mathsf{x}} = \langle x_u^c, w_u \rangle$ and $\omega_{\mathsf{y}} = \langle y_u^c, q_u \rangle$, and let β_{x} and β_{y} be the corresponding IBRPs. The exact representation depends on which players are active. First, consider the case where both x and y are active at s, say $u = \phi_{\mathsf{x}}(s)$ and $v = \phi_{\mathsf{y}}(s)$. Then, we have

$$\begin{aligned}
\mathcal{V}(s) &= w_0(s)\, w(s \mid \beta_{\mathsf{x}})\, w(s \mid \beta_{\mathsf{y}})\, \beta_{\mathsf{x}}(u)^T\, M_{\mathsf{x}}^s\, \beta_{\mathsf{y}}(v) \\
&= w_0(s)\, (w(u \mid \beta_{\mathsf{x}})\, \beta_{\mathsf{x}}(u))^T\, M_{\mathsf{x}}^s\, w(v \mid \beta_{\mathsf{y}})\, \beta_{\mathsf{y}}(v) \\
&= w_0(s)\, x_u^c M_{\mathsf{x}}^s\, y_v^c,
\end{aligned}$$

and so the payoff is bi-linear. The case where only one player, say x, is active at s is similar. Let $u = \phi_{\mathsf{x}}(s)$, and let $(v, o) = \mathsf{upred}_{\mathsf{y}}(s)$. For this case, it is useful to define $w_{v,o} = f_{\mathsf{y}}^{v,o} \cdot y_u^c$. Note that hen $w_{v,o} = w(v, o \mid \beta_{\mathsf{y}})$. Then, we have,

$$\begin{aligned}
\mathcal{V}(s) &= w_0(s)\, w(s \mid \beta_{\mathsf{x}})\, w(s \mid \beta_{\mathsf{y}})\, \beta_{\mathsf{x}}(u)^T\, M_{\mathsf{x}}^s\, \beta_{\mathsf{y}}(\diamondsuit_p) \\
&= w_0(s)\, w(u \mid \beta_{\mathsf{x}})\, w(v, o \mid \beta_{\mathsf{y}})\, \beta_{\mathsf{x}}(u)^T\, M_{\mathsf{x}}^s\, 1 \\
&= w_0(s)\, x_u^c\, M_{\mathsf{x}}^s\, w_{v,o},
\end{aligned}$$

and again the payoff is bi-linear. The case where only y is active is analogous, and the case where neither player is active (e.g., leaf nodes) is a simple extension. □

4.4 Applications of CEFGs

In this section, we give high-level descriptions of how a variety of problems can be modeled as CEFGs, and note that modeling these problems as standard stochastic games or EFGs would require at least an exponential blow-up in representation size.

4.4.1 Stochastic Games and POSGs

We have demonstrated that a CEFG can be represented as a bilinear-payoff convex game, and so we can use such games as the stage games of a convex stochastic game. In Section (3.5) we discussed using EFGs in this manner. This approach is quite powerful, but the time to compute a minimax equilibria will still in general be exponential in the number of actions taken between periods of full observability.

In planning applications it is quite common that each player fully observes their own position, but only has partial observability of the adversary. Further, observations of the adversary may occur relatively rarely compared with the selection of primitive actions. In this case, it may be possible to represent the sequence of actions selected between observations as a single choice from a convex action set, for example the selection of a (partial) policy in an MDP. To represent such a scenario in a standard EFG, each action choice in the MDP would be an action in the EFG as well, requiring us to roll out the MDP until an observation occurs. This EFG would likely be prohibitively large. Using CEFGs, however, we can use a single node to model the selection of a partial policy that determines actions up to the next point where an observation might occur by representing the set of such policies as a convex set. Thus, the depth of the CEFG embedded in the convex game only depends on the number of potential observations involving the adversary between

periods of full observability, rather than the number of primitive actions (which could be much larger), producing an exponentially smaller representation.

If an observation can happen at any time, this approach will not work: the observation model needs to limit observations to occur only at certain times (say, every 15 seconds) or at certain designated states. This restricted observation model could be the true observation model, or it could be an approximate model designed to yield a tractable planning problem. Using an approximate observation model for planning does *not* limit what observations are actually *used*, it only limits what observations for which we can *plan*. That is, during the actual execution of a policy, if we get an observation while in the middle of executing some partial policy we can always re-plan from that point based on the new observation. However, this approach can make no guarantees about the quality of solution executed.

4.4.2 Extending Cost-paired MDP Games with Observations

In Section 3.4, we introduced the notion of a game where one player selects a policy in an MDP, and the other player selects a cost vector for that MDP. This allowed the modeling of an interesting sensor-placement problem. We also showed how the model can be generalized to the case where both players select policies in an MDP, and the total cost of a policy is expressed via a bi-linear function of the two players state-action visitation frequencies. Representing this interesting convex game as an EFG, however, requires using the standard transformation to the normal form representation, which entails an exponential blowup in the size of the representation. This problem can, however, be modeled as a single-node CEFG by simply embedding the convex game representation. This is one demonstration of the representational power of CEFGs.

Further, the CEFG representation makes it possible to represent interesting variations on this problem that cannot be represented as cost-paired MDP games. In particular, we can model some observations of the other player's actions using a deeper game tree. The details of the observation formulation are important: generally, the size of the CEFG will be exponential in the number of states in the underlying MDP where observations can be made; however, the CEFG formulation lets us solve approximations where only the most important observations are considered. We can trade off computation time and approximation accuracy by considering more or fewer observation points.

For example, suppose the robot *can* detect the adversary's sensors in the observation avoidance game. Modeling the possibility of making these observations at all states gives rise to the full (intractable) POSG model (see Section 3.4.2). However, suppose we only designate a few states where observations are considered—perhaps those states that corre-

spond to the robot peeking around a corner where a sensor is particularly likely. Then we can use the CEFG representation to construct a game tree that is exponential in the size of this small set of "observation states," but with only polynomial dependence on the size of the full state space.

This approach can also be applied to approximately solving a generalization of the adversarial Canadian traveler's problem.

The adversarial Canadian traveler's problem The Canadian traveler's problem (CTP) is the problem of computing a shortest path on a graph that is known, except that certain edges may be impassable; whether an edge is passable or not is only revealed when the agent reaches an adjacent node. There has been work on both the *stochastic* version, where there is a known probability distribution that determines whether an edge is passable or not, and the *adversarial* version, where an adversary picks which edges are impassable (with some restrictions). We generalize the adversarial version by allowing the adversary to pick an assignment of costs to the edges; an extremely high cost can be used to model an impassable edge.[13] The stochastic version of the problem can be formulated as a POMDP, while the adversarial version is a POSG; even the stochastic version is #P-hard [Bar-Noy and Schieber, 1991, Papadimitriou and Yannakakis, 1991].

This problem arises naturally in mobile robot path planning, where the uncertainty over edges in the graph might corresponds to uncertainty about whether a door will be open or closed or a bridge will be up or down. The robot-helicopter coordination problem of Likhachev et al. [2005] can be formulated as a CTP; the belief space is finite, and Likhachev et al. solve large instances of this problem by ignoring the POMDP structure and instead using a clever application of heuristic search to the mostly-deterministic belief-space MDP. The resulting algorithm is called MCP. Ferguson et al. [2004] give the PAO* algorithm for "deterministic decision problems with hidden state," which can easily be transformed to instances of the Canadian traveler's problem on a particular graph. Both Likhachev et al. [2005] and Ferguson et al. [2004] construct a compressed representation comprised only of the states adjacent to edges which may be impassable (and hence observations may occur); in PAO* this compressed representation is always fully constructed, while MCP only constructs the portion relevant to the search from a fixed start state to a fixed goal. Blei and Kaelbling [1999] also consider the CTP and discuss its representation as an MDP; they call the problem the "bridge problem." Lita et al. [2001] consider a multi-agent version of the CTP.

[13] While this approach works well in the offline case, for the repeated game (online learning) case bounds typically depend on the maximum edge cost, and so this approach may force online learning algorithms to have poor bounds.

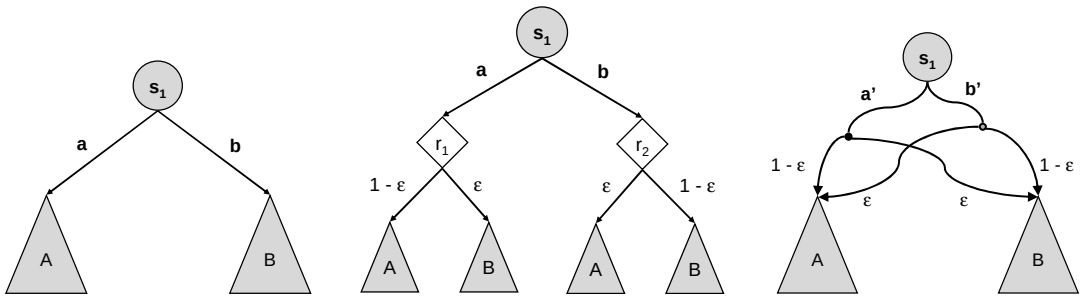
Our adversarial-cost generalization of this problem is formulated as follows: player one needs to get from a fixed start state to a known goal state in a graph. Player two selects a cost vector (assigning a fixed cost to each edge) from some finite set.[14] This is just an MDP with adversary-controlled costs *under the assumption that player one doesn't observe the costs she incurs*. In some domains this may be reasonable, but in others it might not be—for example, if the opponent-chosen costs correspond to the placement of obstacles or active interference. For these domains, we can adopt the CTP observation model, namely that player one can observe the cost of an edge from an adjacent state. If we allow plans that take all of these possible observations into account, we have the full CTP. But, suppose that there are only a few edges that can be made arbitrarily expensive: in a navigation example these might correspond to doors that can be shut, bridges that can be destroyed, or narrow passes that can be blocked. We can efficiently approximate this problem by optimizing over the set of policies that only take into account observed edge costs from states adjacent to potentially expensive edges. This class of plans still has great power to reason about the fact that the adversary has some control over the costs of *all* edges: we simply restrict ourselves from selecting policies that are contingent upon observing these costs. As with the application of CEFGs to convex stochastic games, if we use such a limited observation model we can re-plan upon receiving an observation the original plan did not take into account.

4.4.3 Perturbed Games and Games with Outcome Uncertainty

Selten [1975] originally introduced perturbed EFGs in his investigation of models of sequential rationality. He describes how a perturbed EFG is formed from a standard EFG by introducing a model of "trembles" at each information set: each time a player selects an action, there is a small probability that a different action is taken instead. It is assumed that these probabilities are common knowledge. For a modern introduction to different equilibria refinements, consult Perea [2002].

We show that the class of perturbed extensive-form games can be compactly represented as CEFGs, while their EFG representations are exponentially larger. We begin with the model of Selten, but then extend his model to general outcome uncertainty. This lets us generalize extensive-form games in much the same way that Markov decision processes generalize deterministic path planning. The analogy is not perfect, because perturbed EFGs are still representable as EFGs (but at the cost of an exponential blowup in size), while a general MDP cannot be modeled by any deterministic planning problem. The advantage of using CEFGs to represent perturbed EFGs is that we can avoid the exponential

[14] We can relax this if we further restrict the kinds of observations player one makes.



(a) The Original EFG (b) The Perturbed EFG (c) The Perturbed CEFG

Figure 4.3: Representing a perturbed EFG as an EFG and as a CEFG.

blowup in the size of the representation.

Fix a standard EFG, and let $A(u)$ be the set of actions at an information set u for player one. In the unperturbed EFG, the player chooses some action $a \in A(u)$, and the dynamics of the game ensure that this choice is actually "executed" in the world (whatever that means for the particular game). A perturbed EFG introduces a level of indirection between a player's selection of an action and the execution of that action in the world. A perturbation function $t_u : A(u) \to \Delta(A(u))$ maps the choice a made by the player to a distribution $t_u(a)$ over $A(u)$ from which the action that actually occurs is taken. We write $t_u(a)(a')$ for the probability of action a' actually being executed given that the player selected action a. For example, in constructing trembling-hand equilibria, it is common to consider perturbation functions

$$t_{u,\epsilon}(a)(a') = \begin{cases} 1 - \epsilon(|A_u| - 1) & \text{if } a' = a \\ \epsilon & \text{otherwise} \end{cases}$$

where $|A_u|\epsilon$ is some small probability of a getting a uniform random action rather than the chosen action [Selten, 1975]. It is standard to assume that the player at u observes which a' actually occurred; other players in the game only observe this if they observed the player's action at u in the unperturbed game.

A perturbed EFG is in fact still an extensive form game. The player selects an action $a \in A(u)$ as in the original game, but after this choice a new random node is inserted. The game transitions to this random node, which has successors in 1-1 correspondence with $A(u)$. The action/successor a' at this node is then chosen according to the distribution $t_u(a)(a')$, and the game continues as if the player had selected a' at u in the original game. While we can represent the perturbed game in this way, we have in general doubled the number of nodes on any root to leaf path; of course, doubling the depth of the tree in gen-

eral causes an exponential blowup in its size, and hence the size of the EFG representation. Since algorithms for EFGs are polynomial in this size, we have likely just taken a tractable problem and made it intractable.

This transformation is shown in parts (a) and (b) of Figure (4.3). Part (a) shows the original node s_1 in the EFG game tree where we will introduce perturbations; there are two actions from node s_1, a and b. If action a is taken, the game continues to the subgame A, while if b is taken the game continues to subgame B. Note that A and B can be arbitrarily large trees. For simplicity we do not consider information sets. Part (b) of the figure then shows the introduction of random nodes r_1 and r_2 that implement the ϵ perturbations. The game tree of (b) thus remembers both which action the player wanted to happen as well as which action actually happened: hence there are two copies of the subtrees A and B, doubling the size of the EFG. Applying this transformation at every information set leads to an exponential increase in representation size. Part (c) shows the efficient CEFG representation, which we discuss below.

We now show that a perturbed EFG has a representation as a CEFG of size polynomial in the size of the original game and the size of the representation of the functions t_u. Before introducing this representation, we first generalize our notion of perturbed EFGs to include a complete model of outcome uncertainty.

We generalize Selten's model by decoupling the set of actions available to the player from the set of "actions" (perhaps better called outcomes) that may actually be executed in the world. Formally, we no longer assume t_u maps from the original set of actions to distributions on this same set. Instead, let O_u represent the outcomes that may occur in the world, and define some new set $A'(u) = \{p_1, \ldots, p_k\}$ of probabilistic (meta-)actions for the player. Each action p_i specifies a distribution over possible outcomes, that is, $p_i \in \Delta(O_u)$. Hence the analogy to MDPs, where an action at a state is defined by the distribution over successor states it induces. The perturbed game is played as follows: when $p \in A'(u)$ is selected, the actual outcome that occurs is sampled from O_u according to the distribution p. That is, in this model we have $t_u : A'(u) \rightarrow \Delta(O_u)$. But since we defined each $p \in A'(u)$ as a distribution over O_u, the perturbation function for a game defined in this way is simply the identity function, $t_u(p) = p$ for $p \in A'(u)$.

We now show how to transform an EFG with a perturbation model into a compact CEFG. To represent an *unperturbed* EFG as a CEFG, we kept the same game tree, and replace the finite action set $A(u)$ with the convex action set $\Delta(A(u))$ at each information set u. To represent the *perturbed* EFG we again keep the same tree structure, but the set of available actions at u will be the convex set $X_u \subseteq \Delta(O_u)$ corresponding to those distributions that are realizable given the choices in $A'(u)$. We treat each $p \in A'(u)$ as a

vector in $\Delta(O_u) \subseteq \mathbb{R}^{n_u}$, and so the set of achievable distributions is

$$X_u = \mathsf{H}(A'(u)),$$

the convex hull of the set of explicitly allowed distributions. An example of this representation is shown in part (c) of Figure (4.3). We have $A'(u) = \{a', b'\}$, where a' gives the distribution $(1-\epsilon, \epsilon)$ on the outcomes (A, B), while b' gives the distribution $(\epsilon, 1-\epsilon)$. In the CEFG representation, there is no need to "remember" in the game tree whether A occurred because the player chose it explicitly, or because randomness picked it. In fact, this distinction is not even well-defined in the representation: how would the action $c' = (0.5, 0.5)$ be interpreted?

Of course, in general there is no need to require that X_u is represented as the convex hull of some finite set of actions/distributions $A'(u)$. We can have $X_u \subseteq \Delta(O_u)$ be any complex structured convex set, in particular, X_u can have exponentially many corners while still having a concise representation. Even if the only representation we have for X_u is the explicit one, $X_u = \mathsf{H}(A'(u))$, the CEFG still gives an exponentially smaller representation than an EFG. In a CEFG, increasing the size of the set $A'(u)$ does not change the game tree, and so the corresponding increase in representation size is linear in the size of the new entries added to $A'(u)$. In an EFG, however, increasing the number of actions $A(u)$ increases the branching factor of the tree, producing an exponential blowup in size.[15]

The fact that CEFGs concisely represent perturbed EFGs immediately gives a simple polynomial-time algorithm for finding approximate trembling-hand equilibria (also called perfect equilibria) for extensive-form games: namely, one simply solves the CEFG version of the original EFG perturbed by $t_{u,\epsilon}$. Solving for perfect equilibria (or some other form of sequential equilibria) can be very important in practice, but only very recently have algorithms for finding such equilibria been investigated [Miltersen and Sorensen, 2006].

We have modeled outcome uncertainty efficiently using CEFGs, but have not fully tapped the class's representational power. In particular, we have not used the ability to model both players simultaneously playing at a single node, and we have not used the ability to model different numbers of outcomes at different states in the same information set. Both of these abilities can potentially enable exponentially smaller representations. In the next section we discuss a multi-stage path planning problem where the ability to have both agents selecting actions simultaneously is critical.

[15]The blowup is exponential if we increase $|A(u)|$ at all u; if we increase $|A(u)|$ at only a single u, the size of the game tree increases multiplicatively.

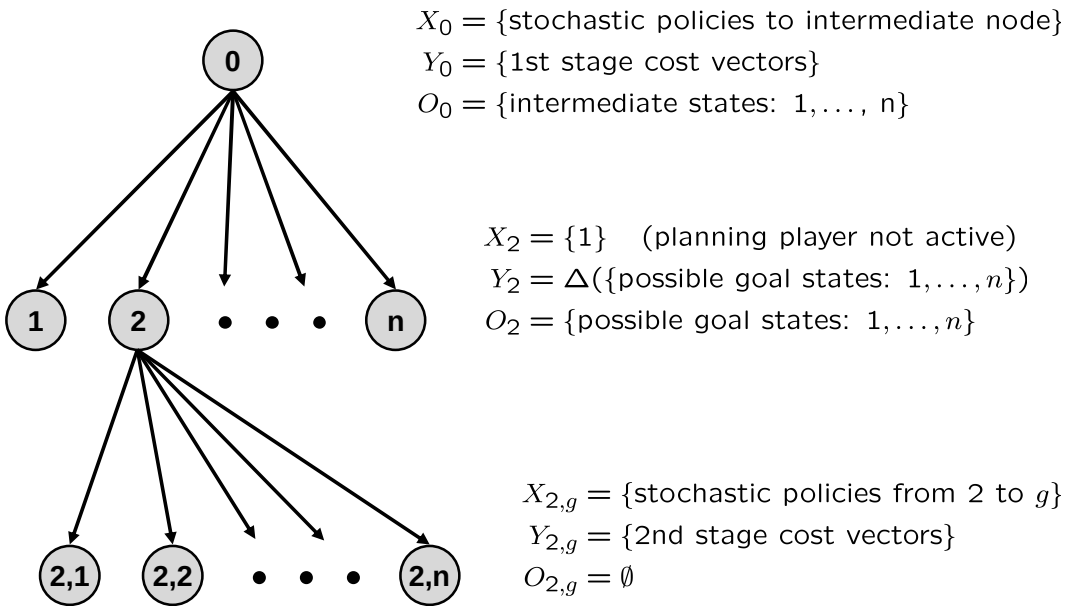


Figure 4.4: The CEFG game tree for a two-stage path planning game.

4.4.4 Uncertain Multi-stage Path Planning

We described a two-stage path planning problem on a graph with with adversarial and stochastic uncertainty about costs and goals. There are two players: the planner, who starts at some designated node in the graph, and the adversary. In the first stage, the planner chooses an initial policy to follow (taking her to some intermediate node); the adversary has some control over costs on the edges in the graph, in the manner of an MDP with adversary-controlled costs. In the second stage, the actual destination node is revealed, and the planning player then selects a policy to go from her intermediate state to the revealed goal; the adversary again has some control over costs.

A portion of a CEFG game tree for this model is given in Figure (4.4). Information sets are not shown for simplicity. The initial node in the game tree corresponds to policy selection and cost selection for the first round. There is one node in the 2nd level for each possible intermediate state. Only the adversary is active at the 2nd level, where he selects the goal state. The final level of the game tree has one node for each (intermediate state, goal state) pair; the figure only shows the states corresponding to intermediate state 2. At this level both players are again active: the planner chooses a policy to follow from the intermediate state to the goal state, and the adversary chooses a cost vector.

Using this CEFG representation, we can model the following types of uncertainty:

- **Outcome uncertainty:** We view the planner as operating in an MDP rather than a deterministic path planning problem. Thus, the intermediate node is not deterministically chosen, but rather will occur according to some distribution induced by the policy of the planner. The MDP here is the appropriate path-planning MDP augmented with a "stop-and-wait" action (always available) that indicates that the planner wants to stop at the current state and wait for the next stage.[16] By introducing time as a state variable in the MDP we can force the planner to use the stop action by a certain deadline, or model costs that increase as a function of time so that an optimal policy will always execute the stop action after some finite amount of time.

- **Stochastic and adversarial control of costs**: In each round, some combination of randomness and adversarial activity determines the cost associated with each edge (state, action) pair in the MDP). For example, it is possible that first the adversary selects a probability distribution from some convex set of probability distributions on costs, and then nature picks the realized costs from that distribution.

- **Stochastic and adversarial control of the goal state**: After the first round, the actual destination may be selected by the adversary, or, as with the costs, some combination of randomness and adversarial choice may select the actual destination.

- **Partial observability:** Information sets can be used to control what the adversary knows. For example, the adversary may be given complete knowledge of the planner's intermediate position (each 2nd level node in the game tree is in its own information set), partial knowledge (the 2nd level nodes are partitioned into some number of information sets), or no knowledge whatsoever (all 2nd level states are in the same information set). Similarly, we can model the planner having only incomplete knowledge of the goal state: she would then have to guess the goal state, and upon arriving at that state execute a "stop-here-because-I-think-it-is-the-goal" action; the reward received would depend on whether or not the state chosen was actually the goal.

[16]We model this by adding a terminal (absorbing) goal state to the MDP that can only be reached by taking the stop-and-wait action. Any proper policy for this MDP induces a probability distribution on the state where the stop-and-wait action is taken; since the stop-and-wait action must be taken exactly once, this distribution can directly be read from the (state,action)-visitation frequency vector for the policy. Thus, we can use this distribution for the transition probabilities in the overall CEFG.

Generalizations to multiple rounds where information about the final destination is revealed incrementally (say, by refining some subset in which the destination actually lies) can easily be constructed. The initial node could also be chosen by the adversary or randomly. The adversary might or might not have full knowledge of the planner's initial location. Even if the destination is fixed in advance, a multi-stage version of this problem could be interesting in that it allows the planner to decide to stop and wait, hoping for a more favorable cost function on the next round. For example, if the planner reaches a door that is closed (modeled as a very high cost), she might decide to stop-and-wait until the next round to see if the door opens, rather than taking a long way around. It is also possible that the adversary's choice at the 2nd level of the game tree determines not only the goal state, but also the dynamics of the MDP for the 3rd level; this can be modeled as long as the planning player observes which dynamics model is active, as the dynamics model determines the action set X_u available to her.

There is a rich tradition in operations research of using both stochastic and adversarial models to handle uncertainty. Purely stochastic models of uncertainty in two-stage and multi-stage problems have received the most attention [Ravi and Sinha, 2004, Gupta et al., 2004, Immorlica et al., 2004], but purely adversarial models have also been considered [Dhamdhere et al., 2005, Bailey et al., 2006]. The CEFG framework can bridge the gap between purely adversarial and purely stochastic formulations, as this example path-planning domain demonstrates. However, the problems considered in operations research and stochastic optimization are typically NP-hard, and hence cannot have polynomial representations as CEFGs. Extending the CEFG framework to include mixed-integer programming models is a exciting avenue for future work. It should also possible to extended our framework to NP-hard problems by modifying the algorithms of the next chapter to use approximation algorithms for the best response oracles.

4.5 Conclusions

In this chapter we introduced convex extensive-form games, showed how to transform CEFGs into convex games, and presented several examples demonstrating the modeling power offered by the CEFG class. Chapter 3 discussed several other interesting problems that can be modeled as convex games. While the results of that chapter showed that convex games can be solved in polynomial time, in the next chapter we turn our attention to constructing algorithms that are much faster in practice than the direct linear programming approach.

Chapter 5

Fast Algorithms for Convex Games

In this chapter we introduce a family of practical algorithms for solving convex games, with a particular focus on the application of the algorithms to MDPs with adversarial costs and extensive-form games. For a review of convex games, refer back to Chapter 3. We begin with a discussion of best response algorithms for particular convex games, and present the well-known fictitious play algorithm that can exploit such oracles. Section 5.2 introduces a special-purpose algorithm for the problem of planning in a MDP where an adversary selects the cost vector from a small finite set of possibilities. Section 5.3 then presents our general convex game algorithm, beginning with an intuitively straightforward version and then proceeding to our full algorithm which addresses some deficiencies of the simplified version. Finally, in Section 5.5.2 we present experiments on both adversarial-cost MDPs and on EFG representations of Rhode Island Hold'em poker. Our results demonstrate dramatic improvements over commercial linear programming software.

5.1 Best Responses and Fictitious Play

A central feature of the algorithms we present in this chapter is that they leverage fast best-response oracles. Consider the convex game $G = (X, Y, M)$, and suppose one player (say, player y) fixes a strategy $y \in Y$. Then, letting $c = My$ (think of c as a cost vector), the best-response problem is to compute:

$$\min_{x \in X} \; c \cdot x. \tag{5.1}$$

If X is a polyhedron, then this is just a standard linear program. But, in many cases, much faster algorithms are available for solving Equation (5.1). In the case of cost-paired

MDP games, solving Equation (5.1) is exactly the problem of planning in an MDP with known costs; this problem can be solved efficiently by any number of algorithms, for example value iteration or even A^* in the special (but practically very useful) case of positive costs and deterministic transitions. For optimal oblivious routing, Equation (5.1) corresponds to solving a multi-commodity flow problem for one of the players. And in the case of extensive-form games, finding a best-response policy is accomplished efficiently via a special dynamic program, as discussed in Section 3.2. In all of these cases, the special-purpose algorithms are likely to perform much better than applying generic linear programming techniques.

Recall that $X \subseteq \mathbb{R}^m$ and $Y \subseteq \mathbb{R}^n$. For the remainder of this chapter, we assume we have efficient algorithms (best-response oracles) $\text{BR}_\mathsf{x} : \mathbb{R}^m \to X$ and $\text{BR}_\mathsf{y} : \mathbb{R}^n \to Y$ for solving Equation (5.1) We view these oracles as functions from cost vectors (rather than opponent strategies) to strategies, so $x = \text{BR}_\mathsf{x}(My)$ is a best response for x to the strategy y, and similarly $y = \text{BR}_\mathsf{y}(x^T M)$ gives a best response for y to x. The matrix-vector multiplications with M are often a dominating computational cost, and so explicitly tracking such multiplications is important; however, to avoid clutter in our pseudo-code we hide the multiplications with M, for example writing $x = \text{BR}_\mathsf{x}(y)$. Further, in the cases just described, the best-response algorithms are better thought of as functions of some suitably chosen cost or reward vector and do not depend in any way on the properties of M.

It is natural to look for algorithms for solving the overall game that can exploit these special purpose best-response oracles. One simple, well-studied algorithm that accomplishes this is fictitious play: the algorithm simulates two players repeatedly playing the convex game G. Each time G is played, each player chooses to play a best response to the average of all her opponent's previous plays.[1] While no guarantees can be made about the performance of each of these players in the simulation, the average over their past plays eventually converges to a minimax equilibrium. For a recent treatment of fictitious play, see [Leslie and Collins, 2006]. Pseudo-code for this simple algorithm is given in Figure (5.1). The average of x's plays is x^{cntr}, and on each iteration this average is updated by taking a step towards $x^{\mathsf{srch}} = \text{BR}_\mathsf{x}(My^{\mathsf{cntr}})$. Each call to a best-response oracles generates an upper or lower bound for the minimax value v^* of G: if x (the min player) plays x^{cntr}, then the max player y can do no better than playing $y = \text{BR}_\mathsf{y}((x^{\mathsf{cntr}})^T M)$, and so we conclude $v^* \leq V(x^{\mathsf{cntr}}, y)$. A similar argument holds for calls to BR_x. The sequence of bounds corresponding to $(x_t^{\mathsf{cntr}}, y_t^{\mathsf{cntr}})$ need not improve monotonically, so in line (1) we use $\max$ and $\min$ to guarantee a monotonic sequence. An implementation can then

[1] Because the sets X and Y are convex, this average is also a valid strategy, and hence we can compute a best response to it.

$x_0^{\mathsf{cntr}} \leftarrow$ any strategy in X

$y_0^{\mathsf{cntr}} \leftarrow$ any strategy in Y

$\mathsf{lb} \leftarrow -\infty$ $\qquad$ $\mathsf{ub} \leftarrow \infty$

$t \leftarrow 0$

while $((\mathsf{ub} - \mathsf{lb}) > \epsilon)$

$t \leftarrow t + 1$

$x_t^{\mathsf{srch}} \leftarrow \mathrm{BR}_{\mathsf{x}}(y_{t-1}^{\mathsf{cntr}})$ $\qquad$ $y_t^{\mathsf{srch}} \leftarrow \mathrm{BR}_{\mathsf{y}}(x_{t-1}^{\mathsf{cntr}})$ $\qquad$ (1)

$v_x = V(x_{t-1}^{\mathsf{cntr}}, y_t^{\mathsf{srch}})$ $\qquad$ $v_y = V(x_t^{\mathsf{srch}}, y_{t-1}^{\mathsf{cntr}})$

$\mathsf{lb} \leftarrow \max(\mathsf{lb}, v_y)$ $\qquad$ $\mathsf{ub} \leftarrow \min(\mathsf{ub}, v_x)$

$x_t^{\mathsf{cntr}} \leftarrow \left(\frac{t}{t+1}\right) x_{t-1}^{\mathsf{cntr}} + \left(\frac{1}{t+1}\right) x_t^{\mathsf{srch}}$ $\qquad$ $y_t^{\mathsf{cntr}} \leftarrow \left(\frac{t}{t+1}\right) y_{t-1}^{\mathsf{cntr}} + \left(\frac{1}{t+1}\right) y_t^{\mathsf{srch}}$

end

return $(x^{\mathsf{cntr}}, y^{\mathsf{cntr}})$ corresponding to ub and lb, respectively

Figure 5.1: The fictitious play algorithm.

track the corresponding argmax and argmin strategies, and return these if the algorithm is interrupted and asked to produce a solution in an anytime fashion; this pair of strategies forms a $(\mathsf{ub} - \mathsf{lb})$-approximate minimax equilibrium. This anytime ability to produce a pair of strategies that form an ϵ-approximate minimax equilibrium is very attractive, as it allows us to trade solution quality against computation time.

This anytime performance can be particularly important when considering very large games where abstractions (approximations) must be introduced to make any solution possible. For example, there has been much recent work on abstraction for extensive-form games, and poker in particular [Billings et al., 2003, Gilpin and Sandholm, 2006a]. In such applications, approximately solving a larger (less abstracted) game may be preferable to exactly solving a more heavily abstracted version.

There is a close connection between fictitious play (especially smooth versions of fictitious play) and running a pair of no-regret algorithms in self-play, one for each player. For example, the algorithms of Kalai and Vempala [2003] and Gordon [2005] can be used in self-play in the same general form as Algorithm (5.1); the best-response oracle is replaced with a special-purpose oracle that, intuitively, introduces additional smoothing such that the agent randomizes among strategies that have similarly good performance. The regret

bounds for such algorithms immediately give both convergence-rate guarantees as well as performance guarantees for the agents in the simulation.

We now investigate another method that utilizes a best-response oracle; however, in this case the oracle is used to generate separating hyperplanes (cutting planes) in a linear programming approach. In the next section we concentrate on the particular case of an MDP with adversarial costs; we introduce more general algorithms for convex games in the following section.

5.2 The Single-Oracle Algorithm for MDPs with Adversarial Costs

In this section we develop an efficient algorithm to solve MDP planning problems where an adversary selects the cost vector from a finite set K of possible costs. This problem was introduced in Section 3.4. In particular, we use Benders' decomposition [Benders, 1962] to capitalize on the existence of best-response oracles like A^*-search and value iteration. The double oracle algorithms introduced later generalize this technique to the case where a best-response oracle is also available for the adversary.

Recall the problem formulation from Section 3.4.2: We have an MDP $\mathcal{M}$ with known dynamics and a fixed start-state distribution μ_s, and a set $K = \{c_1, \ldots, c_k\}$ of cost vectors. Simultaneously, player x selects a policy π for $\mathcal{M}$ and player y selects a cost vector $c \in K$. Then, player x pays y the amount $V(\pi, c)$, the expected cost of following policy π from a state sampled from μ_s under cost vector $c \in K$. The dynamics of the MDP are captured by the matrix E. For a fixed start-state distribution μ_s, the set of stochastic policies for player x can be represented as the set of valid state-action visitation frequencies,

$$F = \{f \in \mathbb{R}^{|S||A|} \mid E^T f + \mu_s = 0,\ f \geq 0\}.$$

Thus, the game can be formulated as the convex game $(F, \Delta(K), M)$, where M is the matrix with columns $c_1, \ldots, c_k$.

Our iterative algorithm is an application of Benders' decomposition, a general method for decomposing certain linear programs first studied by Benders [1962]. We focus on the application of this technique to the problem at hand, and refer the reader to Bazaraa et al. [1990] for a more general introduction. Benders' decomposition is dual to the Dantzig-Wolfe decomposition, and can also be viewed as a specialization of the Kelley cutting plane method to linear programs [Hiriart-Urruty and Lemaréchal, 1993].

The set

$$\mathsf{H}(K) = \{c \mid c = Mq, q \in \Delta(K)\}$$

is the set of all (expected) cost vectors the adversary can potentially achieve by playing implicit mixed strategies from $\Delta(K)$. Our algorithm is applicable when we have an oracle $\mathsf{BR}_{\mathsf{x}} : \mathsf{H}(K) \to F$ that for any cost vector $c \in \mathsf{H}(K)$ provides a best-response policy π. Our algorithm requires that π be represented by its state-action frequency vector f_π. If the oracle algorithm actually used provides a policy represented as a value function (for example, if we implement the oracle using value iteration) we can calculate f_π with a matrix inversion or by iterative methods.

We quickly restate the relevant linear programming results from Section 3.4.3. The value of the MDP for for a fixed cost c (typically, $c = Mq$ for the game) can be found by solving the linear program

$$\begin{aligned} \max_{v} \quad & v \cdot \mu_s \\ \text{subject to} \quad & Ev + c \geq 0, \end{aligned} \tag{5.2}$$

or via the dual,

$$\begin{aligned} \min_{f} \quad & f \cdot c \\ \text{subject to} \quad & E^T f + \mu_s = 0 \\ & f \geq 0. \end{aligned} \tag{5.3}$$

We can solve the adversarial MDP convex game via the linear program

$$\begin{aligned} \max_{v,q} \quad & v \cdot \mu_s \\ \text{subject to} \quad & Ev + Mq \geq 0 \\ & \mathbf{1} \cdot q = 1 \\ & q \geq 0, \end{aligned} \tag{5.4}$$

or via its dual,

$$\begin{aligned} \min_{z,f} \quad & z \\ \text{subject to} \quad & E^T f + \mu_s = 0 \\ & \mathbf{1} \cdot z + M^T f \leq 0 \\ & f \geq 0. \end{aligned} \tag{5.5}$$

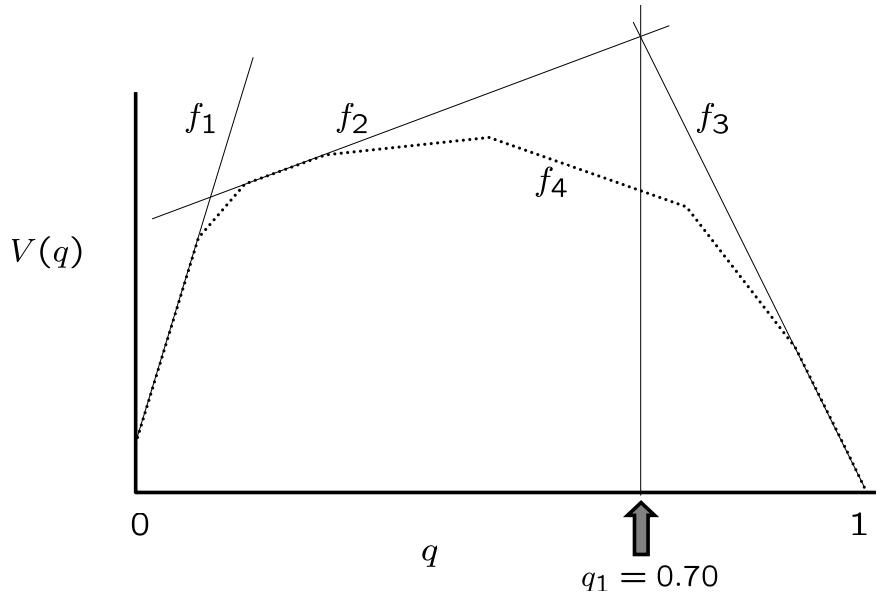


Figure 5.2: The piece-wise linear concave function V (dotted line) and an approximation $V_{\mathcal{B}}$ based on the bundle $\mathcal{B}_{\mathsf{x}} = \{f_1, f_2, f_3\}$ (the minimum of the three thin black lines). The maximum with respect to the approximation $V_{\mathcal{B}}$ is at $q_1 = 0.7$.

Let $V(q)$ be the optimal value of (5.2) for a fixed cost vector $c = Mq$ for $q \in \Delta(K)$; we can evaluate V using the best-response oracle BR_{x}. Then, we can rewrite (5.4) as the program

$$\max_{q \in \Delta(K)} \begin{bmatrix} \max\limits_{v} & v \cdot \mu_s \\ \text{subject to} & Ev + Mq \geq 0, \end{bmatrix} = \max_{q \in \Delta(K)} V(q) \tag{5.6}$$

We will work with the right-hand-side of this representation. Unfortunately, $V(q)$ is not linear so we cannot solve the program directly as a linear program over $\Delta(K)$. However, it can be solved via a convergent sequence of approximations that capitalize on the availability of our oracle BR_{x}. Using strong duality for linear programming and Equation (5.3), we can rewrite V as

$$V(q) = \min_{f \in F} \quad (Mq) \cdot f.$$

Since V is the minimum over a polyhedral set of linear functions, it is piecewise linear and concave [see Boyd and Vandenberghe, 2004, for example]. Let $\mathcal{B}_{\mathsf{x}} \subseteq F$ be a finite subset of F. Then,

$$V_{\mathcal{B}}(q) = \min_{f \in \mathcal{B}_{\mathsf{x}}} \quad (Mq) \cdot f,$$

is a piecewise linear and concave upper bound on V. If $\mathsf{Cn}(F) \subseteq \mathcal{B}_{\mathsf{x}}$, then it can be shown that $V_{\mathcal{B}} = V$. Figure (5.2) shows V and $V_{\mathcal{B}}$ for an example with $|K| = 2$, and so $\Delta(K)$ effectively has a single dimension. The dotted line indicates V; it is defined in the case by 7 linear segments. The approximation $V_{\mathcal{B}}$, based on the set of strategies $\mathcal{B}_{\mathsf{x}} = \{f_1, f_2, f_3\}$, gives an upper bound; the rest of the figure will be explained shortly.

Our algorithm performs two steps for each iteration. On iteration i, first we solve for an optimal mixture of costs q_i under the assumption that the planner is only allowed to select a policy[2] from a restricted set $\mathcal{B}_{\mathsf{x}} = \{\pi_1, \pi_2, \ldots, \pi_i\} \subseteq F$. Then, we use the oracle to compute $\text{BR}_{\mathsf{x}}(Mq_i) = \pi_{i+1}$, an optimal deterministic policy with respect to the fixed cost vector $c = Mq_i$. The policy π_{i+1} is added to $\mathcal{B}_{\mathsf{x}}$, and these steps are iterated until an iteration where π_{i+1} is already in $\mathcal{B}_{\mathsf{x}}$.

All that remains is to show how to find the optimal cost mixture q_i given that the planner will select a policy from the set $\mathcal{B}_{\mathsf{x}} = \{f_1, f_2, \ldots, f_i\}$ of feasible solutions to (5.3). That is, we wish to solve Equation (5.6) with V replaced by $V_{\mathcal{B}}$. This problem can be solved with the linear program

$$\max_{q,v} v \text{ subject to} \qquad (5.7)$$
$$v \leq f_j^T M q \quad \text{for } 1 \leq j \leq i,$$

which is essentially the same program as (5.5), where f is restricted to be a member of $\mathcal{B}_{\mathsf{x}}$ rather than an arbitrary stochastic policy. The key difference is that $f_j^T M$ is a constant vector for each $f_j \in \mathcal{B}_{\mathsf{x}}$, and so the size of the linear program is independent of the size of the MDP $\mathcal{M}$ (there is no dependence on $|S|$ or $|A|$, as there is in Equations (5.5) and (5.4). We interpret this program as solving the matrix game with one column for each $c \in K$, and one row for each $f \in \mathcal{B}_{\mathsf{x}}$. This program is known as the *master* program of the Benders' decomposition. Equation (5.3) is the *slave* program, which in our case is solved not as a linear program, but using the fast best-response oracle. In Figure (5.2), solving the master program (5.7) for the set $\mathcal{B}_{\mathsf{x}} = \{f_1, f_2, f_3\}$ produces the minimax solution $q_1 = 0.7$ for the cost-selecting player (and so $q_2 = 0.3$). The algorithm then computes the corresponding cost vector, $c = 0.7c_1 + 0.3c_2$, and then calls $\text{BR}_{\mathsf{x}}(c)$, which returns the best-response strategy f_4. This strategy is then added to $\mathcal{B}_{\mathsf{x}}$ and the process continues. Pseudo-code for the complete algorithm is given in Figure (5.3).

Since $V_{\mathcal{B}}$ gives an upper bound on V, the solution to the master program (5.7) gives an upper bound on the value of the game. Since we only ever tighten the approximation of V, the sequence of upper bounds generated by the algorithm is non-decreasing. As mentioned

[2]We write π_i and f_i equivalently, depending on which interpretation we wish to emphasize: π_i is a stochastic policy, and f_i is the corresponding state-action visitation frequency vector

$\mathcal{B}_{\mathsf{x}}^0 \leftarrow \{\}$
$q_0 \leftarrow$ arbitrary $q \in \Delta(K)$
$\mathsf{lb} \leftarrow -\infty \qquad \mathsf{ub} \leftarrow \infty$
$t \leftarrow 1$
while $((\mathsf{ub} - \mathsf{lb}) > \epsilon)$
  $f_t = \text{BR}_{\mathsf{x}}(Mq_{t-1})$
  $\mathsf{lb} \leftarrow \max(\mathsf{lb}, V(f_t, q_{t-1}))$
  $(q_t, v) \leftarrow$ solution to the LP of Equation (5.7)
  $\mathsf{ub} \leftarrow v$ // v improving monotonically
  $t \leftarrow t + 1$
end
return best $(\bar{f}, q)$

Figure 5.3: The single oracle algorithm.

before, each call to BR_{x} generates a lower bound on the value of the game, but these need not be monotonically increasing, and so we use the max operator.

The ϵ-approximate minimax cost mixture is given by the q corresponding to the best lower bound. The approximately optimal policy for the planning player can be expressed as a distribution over the policies in $\mathcal{B}_{\mathsf{x}}$. These values are given by the dual variables (say, p) of (5.7), and can thus be found via matrix inversion or may be immediately available depending on the linear programming technique used to solve the program. Given the dual variables p, we compute the best f as the stochastic policy

$$\bar{f} = \sum_{i=1}^{t} p_i f_i.$$

The convergence and correctness of this algorithm are immediate from the corresponding proofs for Benders' decomposition; in the worst case all of the strategies in $\mathsf{Cn}(F)$ will be added to $\mathcal{B}_{\mathsf{x}}$, ensuring finite though possibly exponential runtime. In Section 5.5.1, we demonstrate experimentally that the sequence of bounds converges quickly in practice. We refer to this algorithm as the single oracle algorithm because it relies only on a best-response oracle for the row player. While we have stated the algorithm of this section in

terms of the particular convex game $(F, \Delta(K), M)$ for MDPs with adversarial costs, this technique can be applied any time one of the players in the convex game has strategies given via a relatively small explicitly enumerated set let K. The constraint-generation approach used here can also be applied to solving the linear programs of Equation (5.5) or Equation (5.4) via the ellipsoid or analytic centering algorithms [Bazaraa et al., 1990, Hiriart-Urruty and Lemaréchal, 1993].

Motivation for the Double Oracle Algorithm

The single oracle algorithm is sufficient for problems when the set K is reasonably small; in these cases solving the master problem, Equation (5.7), is fast. For example, we use this approach in our path planning problem if the opponent is confined to a small number of possible sensor locations and we know that he will place only a single sensor. However, suppose there are a relatively large number of possible sensor locations (say 50 or 100), and that the adversary will actually place 2 sensors. If the induced cost function assigns an added cost to all locations visible by one or more of the sensors, then we cannot decouple the choice of locations, and so there will be $\binom{100}{2}$ possible cost vectors in K. The single oracle algorithm is not practical for a problem with this many cost vectors; simply representing them all in memory would be prohibitive.

We now derive a generalization of the single oracle algorithm that can take advantage of best-response oracles for both players. We present this algorithm as it applies to arbitrary convex games.

5.3 A Bundle-based Double Oracle Algorithm

In this section, we introduce two algorithms that take advantage of a best-response oracle for both players. The basic double oracle bundle algorithm (first introduced in [McMahan et al., 2003]) is described first; we then extend basic DOBA with line search, aggregation, and a method for interpolating with fictitious play. We call this extended algorithm DOBA+.

The basic DOBA builds up a collection of strategies (called a bundle) for each player. On each iteration it solves an approximate game where each player is only allowed to randomize among the strategies contained in his or her bundle. Given the optimal strategies in this restricted game, it calls the oracles to find best responses for each player in the full game, and then adds these responses to the bundles to improve the approximation. The double oracle bundle method is related to the family of cutting plane and bun-

$\mathcal{B}_{\mathsf{x}}^0 \leftarrow \{x_0\}$ $\quad$ $\mathcal{B}_{\mathsf{y}}^0 \leftarrow \{y_0\}$

$\tilde{M}_{x_0,y_0} \leftarrow V(x_0, y_0)$

$\mathsf{lb} \leftarrow -\infty$ $\quad$ $\mathsf{ub} \leftarrow \infty$

$t \leftarrow 0$

while $((\mathsf{ub} - \mathsf{lb}) > \epsilon)$

$(p, q) \leftarrow$ **solveMatrixGame**$(\tilde{M})$

$x_t^{\mathsf{mix}} \leftarrow \sum_{x \in \mathcal{B}_{\mathsf{x}}} p_x x$ $\quad$ $y_t^{\mathsf{mix}} \leftarrow \sum_{y \in \mathcal{B}_{\mathsf{y}}} q_y y$

$x_{t+1} \leftarrow \mathrm{BR}_{\mathsf{x}}(y_t^{\mathsf{mix}})$ $\quad$ $y_{t+1} \leftarrow \mathrm{BR}_{\mathsf{y}}(x_t^{\mathsf{mix}})$

$v_x = V(x_t^{\mathsf{mix}}, y_{t+1})$ $\quad$ $v_y = V(x_{t+1}, y_t^{\mathsf{mix}})$

$\mathsf{lb} \leftarrow \max(\mathsf{lb}, v_y)$ $\quad$ $\mathsf{ub} \leftarrow \min(\mathsf{ub}, v_x)$

$\mathcal{B}_{\mathsf{x}}^{t+1} \leftarrow \mathcal{B}_{\mathsf{x}}^t \cup \{x_{t+1}\}$ $\quad$ $\mathcal{B}_{\mathsf{y}}^{t+1} \leftarrow \mathcal{B}_{\mathsf{y}}^t \cup \{y_{t+1}\}$

$(\forall y' \in \mathcal{B}_{\mathsf{y}})\ \tilde{M}_{x_{t+1},y'} \leftarrow V(x_{t+1}, y')$

$(\forall x' \in \mathcal{B}_{\mathsf{x}})\ \tilde{M}_{x',y_{t+1}} \leftarrow V(x', y_{t+1})$

$t \leftarrow t + 1$

end

return best $(x^{\mathsf{mix}}, y^{\mathsf{mix}})$

Figure 5.4: The basic double oracle bundle algorithm.

dle algorithms for non-smooth optimization, and to Benders' decomposition in the case of polyhedra [Hiriart-Urruty and Lemaréchal, 1993]. However, the direct application of those techniques to convex games yields algorithms that only take advantage of the best-response oracle for one of the players, not both. There is great potential for future work in adapting the rich set of techniques from that literature to the particular problem of solving convex games; the work presented here is but a first step in that direction.

5.3.1 The Basic Algorithm

In this section we describe the basic double oracle algorithm. This algorithm can require an amount of memory exponential in the problem size. While it still manages very good performance on adversarial cost MDP problems, it is of only theoretical interest for solv-

ing large extensive-form games like Rhode Island Hold'em; the full DOBA+ algorithm, introduced next, builds on the basic double oracle algorithm by providing a way to limit memory consumption. Pseudo-code for the basic algorithm is given in Figure (5.4).

The principal intuition of the double bundle oracle algorithms is to use our best-response oracles to build up an approximate version of the full convex game. Fix $G = (X, Y, M)$ as the convex game we wish to solve. Our approximate game $\tilde{G}$ will also be convex, given by $(\bar{X}, \bar{Y}, M)$, where $\bar{X} \subseteq X$ and $\bar{Y} \subseteq Y$. The set $\bar{X}$ will be constructed from a set of best responses for x to various player y strategies, and similarly for $\bar{Y}$. It should be clear that, if $\bar{X}$ approaches X and $\bar{Y}$ approaches Y, the approximate game becomes more and more similar to the exact one. Of course, we hope that the approximation becomes good before $\bar{X}$ and $\bar{Y}$ become intractably large. The key difference between the basic algorithm and DOBA+ is that the latter explicitly manages the complexity of $\bar{X}$ and $\bar{Y}$ while still guaranteeing convergence to a solution to the overall game G.

In both the basic algorithm and DOBA+, we maintain a finite bundle of strategies for each player, $\mathcal{B}_\mathsf{x} \subseteq X$ and $\mathcal{B}_\mathsf{y} \subseteq Y$ respectively. The convex hull of $\mathcal{B}_\mathsf{x}$, written $\mathsf{H}(\mathcal{B}_\mathsf{x})$, is a convex subset of X, and similarly $\mathsf{H}(\mathcal{B}_\mathsf{y})$ is a convex subset of Y. Thus, we can define the convex game $\tilde{G} = (\mathsf{H}(\mathcal{B}_\mathsf{x}), \mathsf{H}(\mathcal{B}_\mathsf{y}), M)$ which we will use as a model of G.

We can define an equivalent matrix game $\tilde{M}$ which has strategy sets $\mathcal{B}_\mathsf{x}$ for player x and $\mathcal{B}_\mathsf{y}$ for player y, with payoffs $\tilde{M}(x, y) = V(x, y)$. The convex game $\tilde{G}$ is equivalent to $\tilde{M}$, in the sense that strategies in $\tilde{G}$ and $\tilde{M}$ can be translated back and forth without altering expected payoffs. More precisely, the bi-linearity of V means that if (p, q) is a solution to $\tilde{M}$, then $x^{\mathsf{mix}} = \sum_{x \in \mathcal{B}_\mathsf{x}} p(x)x$ and $y^{\mathsf{mix}} = \sum_{y \in \mathcal{B}_\mathsf{y}} q(y)y$ form a solution to $\tilde{G}$.

We will move back and forth between the two equivalent representations $\tilde{G}$ and $\tilde{M}$. For interpretation we will use $\tilde{G}$, since its relationship to G is more clear. But for computation we will work with $\tilde{M}$, since its size is independent of m and n (it is $|\mathcal{B}_\mathsf{x}| \times |\mathcal{B}_\mathsf{y}|$). This last fact is critical for large games: for example, in Rhode Island Hold'em, m and n are both approximately 1×10^6, while we fix $|\mathcal{B}_\mathsf{x}| = |\mathcal{B}_\mathsf{y}| = 55$.

Both the basic algorithm and DOBA+ build up the model game $\tilde{M}$ in an intuitive way using our best-response oracles. We initialize the bundles with one or more arbitrarily chosen strategies for each player. These could be the an arbitrary or randomly chosen strategy, but there is the opportunity to increase performance by seeding the algorithm with a collection of expert-generated strategies.

Given the current bundles $\mathcal{B}_\mathsf{x}$ and $\mathcal{B}_\mathsf{y}$, we solve the corresponding matrix game $\tilde{M}$, producing a mixed strategy (p, q). We then compute the corresponding strategies $x^{\mathsf{mix}} \in X$ and $y^{\mathsf{mix}} \in Y$; $(x^{\mathsf{mix}}, y^{\mathsf{mix}})$ is a valid strategy pair in either $\tilde{G}$ or G. Because $(x^{\mathsf{mix}}, y^{\mathsf{mix}})$ is a strategy pair for G, we can use our oracles to generate best responses $\mathrm{BR}_\mathsf{x}((x^{\mathsf{mix}})^T M)$

and $\text{BR}_{\mathsf{y}}(My^{\mathsf{mix}})$. We add these new strategies to the bundle, and also use the fact that they are best responses to update upper and lower bounds on the value of the game G. The algorithm continues in this fashion until the gap $\epsilon = (\mathsf{ub} - \mathsf{lb})$ reaches an acceptably small level.

Discussion On each iteration, the size of each bundle increases by one, as does each dimension of the matrix game $\tilde{M}$. This is, in fact, the principal weakness of the basic algorithm: the cost of each iteration and the size of the bundles grow, making it infeasible to run an arbitrary number of iterations. In particular, for Rhode Island Hold'em, storing each strategy in the bundle requires about 7MB of memory, and so physical memory rapidly limits the size of the bundles and hence the number of iterations we can run. To address this issue, DOBA+ uses an aggregation and pruning scheme that allows it to maintain a constant bundle size.

A second deficiency of the basic double oracle algorithm is that inaccuracies in the model $\tilde{G}$ can lead to solutions $(x^{\mathsf{mix}}, y^{\mathsf{mix}})$ that in fact perform poorly in the true game G. However, the direction from the current best pair of strategies (x^*, y^*) towards $(x^{\mathsf{mix}}, y^{\mathsf{mix}})$ usually provides a good direction of improvement. To exploit this fact, we introduce a fast line search procedure that efficiently solves this 1-dimensional optimization problem. Pseudo-code for the complete DOBA+ algorithm is given in Figure (5.5); in the following sections we discuss the principal improvements over the basic version.

5.3.2 Aggregation

Our aggregation and pruning scheme has two components. First, we insert the minimax strategies x^{mix} and y^{mix} into the bundles. This has no effect on the convex hulls of the bundles if we never remove strategies, but since we will be discarding strategies, adding the mixtures is useful: in this way even if we throw out some strategies that support x^{mix}, we may still keep $x^{\mathsf{mix}} \in \mathsf{H}(\mathcal{B}_{\mathsf{x}})$ by explicitly placing $x^{\mathsf{mix}} \in \mathcal{B}_{\mathsf{x}}$.

In order to determine which strategies to discard, each time we solve $\tilde{M}$ we use the mixed strategies to update a weight $w(x)$ (or $w(y)$) associated with each strategy in the bundle: this weight is a discounted average of the probabilities placed on the strategy by past solutions to $\tilde{M}$. Each iteration, we choose to remove the strategies with the smallest weights; we then add to the bundle an aggregate of the removed strategies, with each removed strategy weighted proportionally to its weight. That is, if we remove $x_1, \ldots, x_k$

while $((\mathsf{ub} - \mathsf{lb}) > \epsilon)$
  $(p, q) \leftarrow$ **solveMatrixGame**$(\tilde{M})$
  update strategy weights
  $x^{\mathsf{mix}} \leftarrow \sum_{x \in \mathcal{B}_\mathsf{x}} p(x)x$
  $y^{\mathsf{mix}} \leftarrow \sum_{y \in \mathcal{B}_\mathsf{y}} q(y)y$
  updateCenter(x)
  $\mathsf{ub} \leftarrow \min\left(\mathsf{ub}, V(x^{\mathsf{cntr}}, \mathrm{BR}_\mathsf{y}(x^{\mathsf{cntr}}))\right)$
  updateCenter(y)
  $\mathsf{lb} \leftarrow \max\left(\mathsf{lb}, V(\mathrm{BR}_\mathsf{x}(y^{\mathsf{cntr}}), y^{\mathsf{cntr}})\right)$
  if ($\mathcal{B}_\mathsf{x}$ or $\mathcal{B}_\mathsf{y}$ are too big)
    do aggregation on $\mathcal{B}_\mathsf{x}$ or $\mathcal{B}_\mathsf{y}$
  update ϕ
  $t \leftarrow t + 1$
end

updateCenter(x):
  $x^{\mathsf{srch}} \leftarrow$ **search**$(\mathrm{BR}_\mathsf{x}(y^{\mathsf{cntr}}), x^{\mathsf{mix}}, [0, 1 - \phi])$
  $\mathsf{fpstep} \leftarrow 1/(t + 1)$
  $\alpha \leftarrow \phi \cdot \mathsf{fpstep}$
  $\beta \leftarrow \mathsf{fpstep} + (1 - \phi)(1 - \mathsf{fpstep})$
  $x^{\mathsf{cntr}} \leftarrow$ **search**$(x^{\mathsf{cntr}}, x^{\mathsf{srch}}, [\alpha, \beta])$
  add $\{x^{\mathsf{mix}}, \mathrm{BR}_\mathsf{x}(y^{\mathsf{mix}})\}$ to $\mathcal{B}_\mathsf{x}$ (*)
  add $\{x^{\mathsf{cntr}}, \mathrm{BR}_\mathsf{x}(y^{\mathsf{cntr}})\}$ to $\mathcal{B}_\mathsf{x}$ (*)
end

Figure 5.5: DOBA+: the double oracle bundle algorithm with line search, aggregation, and convergence guarantees. Initialization and updates to $\tilde{M}$ are similar to those in the basic algorithm.

and let $W = \sum_{i=1}^{k} w(x_i)$, then the aggregate strategy is given by

$$x_{\text{aggr}} = \sum_{i=1}^{k} \frac{w(x_i)}{W} x_i.$$

To keep the bundle size constant, we remove five strategies on each step: one each to to make room for the four strategies added to the bundle in the lines marked (*) in Figure (5.5), and one to make room for the aggregated strategy x_{aggr}. Of course, when we remove strategies from the bundles we must remove the corresponding rows and columns from $\tilde{M}$ as well.

5.3.3 Line Search

For extensive-form games, it takes time $\mathcal{O}(m)$ to run the oracle $\text{BR}_{\mathsf{x}}(c_y)$ for a fixed cost vector $c_y = My$, but the cost of the multiplication to compute c_y is $\mathcal{O}(nm)$. While the matrix M may be sparse, multiplications with M will still typically be slower than best-response calls by a considerable constant; for example, this constant is around 20 for Rhode Island Hold'em, and even higher for approximately abstracted versions of Texas Hold'em. However, the multiplication Mq for adversarial MDPs with only a few possible cost vectors (small $|K|$) will be relatively inexpensive, usually much cheaper than solving the MDP with the fixed cost vector $c = Mq$. Thus, the applicability of the techniques of this section will depend on these tradeoffs for the particular convex game at hand.

In this section we show how we can take advantage of the relative speed of computing best responses for fixed cost vectors. Consider a restricted convex game with $\mathcal{B}_{\mathsf{x}} = \{x_1, x_2\}$ and $\bar{X} = \mathsf{H}(\mathcal{B}_{\mathsf{x}})$ but $\bar{Y} = Y$. That is, x has exactly two strategies, while y has full access to his strategy set. We show that we can solve the corresponding restricted game $(\mathsf{H}(\mathcal{B}_{\mathsf{x}}), Y, M)$ efficiently via a line search.

The key is that x's choice of a probability distribution over $\mathcal{B}_{\mathsf{x}}$ only has a single degree of freedom. Using θ to represent this free variable, we can write the problem of solving this game as:

$$\min_{\theta \in [0,1]} \max_{y \in Y} \left((1-\theta)x_1 + \theta x_2\right)^T My \tag{5.8}$$

For simplicity, we write $x(\theta) = ((1-\theta)x_1 + \theta x_2)$. Then, define the function $f : \mathbb{R} \to \mathbb{R}$ by

$$f(\theta) = \max_{y \in Y} x(\theta)^T My \tag{5.9}$$

and so solving Equation (5.8) is equivalent to solving $\min_{\theta \in [0,1]} f(\theta)$. In fact, f is just a piecewise maximum over a set of affine functions, one for each $y \in Y$, and so f is convex. We can minimize such a function via an exact binary line search if we can evaluate f at all $\theta \in [0, 1]$ and also compute a subgradient to f at each θ. The best-response oracle BR_{y} can be used to accomplish both these tasks.

For a fixed θ, we can find a y that achieves the maximum in Equation (5.9) by computing $y = \text{BR}_{\mathsf{y}}(x(\theta)^T M)$, so that $f(\theta) = V(x(\theta), y)$. Further, y corresponds to the linear function $x(\theta)^T My$ which gives a lower bound on f and is tight at θ, so the slope of f_y is a subgradient of f at θ. This can easily be calculated as $(x_2 - x_1)^T My$.

This gives us the necessary components to implement a line search, but each evaluation and subgradient calculation may require a multiplication with M. We avoid these multiplications in the following way. Let $c_1 = x_1^T M$ and $c_2 = x_2^T M$, which we can pre-compute

before the line search[3] and define $c(\theta) = (1-\theta)c_1 + \theta c_2$. For a fixed θ, we can evaluate $c(\theta)$ in $\mathcal{O}(m)$ time. After computing $y = \text{BR}_{\mathsf{y}}(c(\theta))$, we calculate[4] $f(\theta)$ as $c(\theta) \cdot y$. Using the same y, we can then calculate the necessary subgradient as $(c_2 - c_1) \cdot y$. This, each iteration of the binary search can be completed in $\mathcal{O}(m)$ time.

The subroutine **search**$(x_1, x_2, [\alpha, \beta])$ used in DOBA+ solves the problem $\min_{\theta \in [\alpha,\beta]} f(\theta)$. We will see that restricting the allowed interval using to $[\alpha, \beta]$ is useful in interpolating with fictitious play.

5.3.4 Convergence Guarantees and Fictitious Play

Fictitious play (or a no-regret algorithm in self-play) maintains *centers* x^{cntr} and y^{cntr}, estimates of the minimax optimal implicit mixed strategies. On each iteration FP updates these centers in the *search direction*, $x^{\mathsf{srch}} = \text{BR}_{\mathsf{x}}(y^{\mathsf{cntr}})$ and $y^{\mathsf{srch}} = \text{BR}_{\mathsf{y}}(x^{\mathsf{cntr}})$. DOBA+ has a similar structure: it maintains a center for each player, and on each iteration updates these centers towards a search direction. The algorithm maintains a parameter ϕ (the fictitious play fraction), so that when $\phi = 0$ the algorithm runs in an unrestricted fashion, while if $\phi = 1$, the algorithm behaves identically to fictitious play.

The selection of the search direction and update of the center occurs in th **updateCenter**(x) method of Figure (5.5); **updateCenter**(y) is identical, but with the roles of x and y switched. The best response to the opponent's current center is one possible search direction; the solution to the model game $\tilde{M}$ provides another. DOBA+ does a line search between these two possibilities in order to choose its search direction; however, at least ϕ weight is required to be on the best response to the opponent's center, so that when $\phi = 1$ DOBA+ uses the same search direction as FP. This is accomplished via the call to **search**$(\text{BR}_{\mathsf{x}}(y^{\mathsf{cntr}}), x^{\mathsf{mix}}, [0, 1-\phi])$.

Similarly, we update the center by a line search from x^{cntr} towards x^{srch}, but we constrain the interval of the search to linearly interpolate from $[0, 1]$ when $\phi = 0$, to $[1/(t+1), 1/(t+1)]$ when $\phi = 1$. The constants α and β in the call to **search**$(x^{\mathsf{cntr}}, x^{\mathsf{srch}}, [\alpha, \beta])$ accomplish this interpolation; when $\phi = 1$, we have the fixed step-size of $1/(t+1)$ of fictitious play.

We can insure convergence of DOBA+ be updating ϕ based on the rate of change of $(\mathsf{ub} - \mathsf{lb})$ so that if the rate drops lower than that expected of FP, ϕ eventually goes

[3] In fact, for the x_1 and x_2 used in DOBA+, we will need to compute c_1 and c_2 anyway at some point, and so this computation can be made effectively free.

[4] In many cases the oracle also provides the value $V(x, \text{BR}_{\mathsf{y}}(x^T M))$ directly in which case this multiplication is unnecessary.

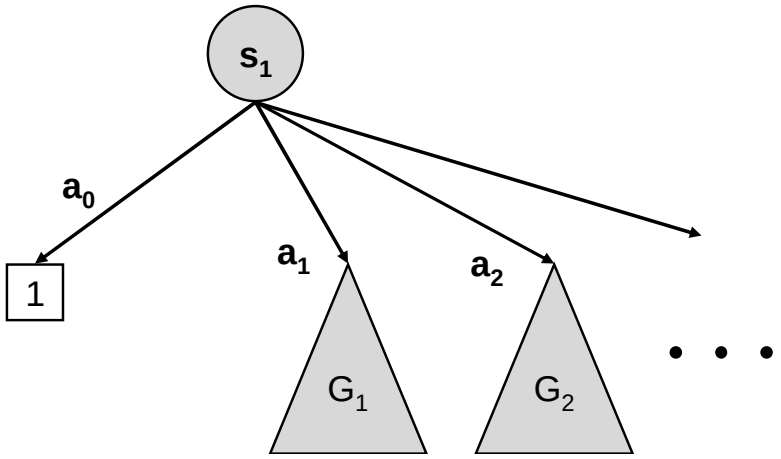


Figure 5.6: An EFG where best responses can be bad.

to 1, and DOBA+ effectively becomes FP. An implementation can be designed so that once $\phi > 0.99$ (say), it switches fully to FP and avoids the overhead of solving $\tilde{M}$ and performing the line searches.

Note that DOBA+ is asynchronous: the update for x is done first, and then the update for y takes into account x's updated x^{cntr}. This is commonly done in FP implementations as well. We ran experiments with several simple methods for updating ϕ; generally these had little impact of the runtime of the algorithm. To avoid conflating the impact of the ϕ updating scheme with the performance of our unconstrained approach, we present experimental results with ϕ fixed at 0.

5.4 Good and Bad Best Responses for Extensive-form Games

To paraphrase George Orwell, "*All best responses are equal but some best responses are more equal than others.*" We now investigate this notion.

Bad best responses Consider the EFG of Figure (5.6). Suppose player x is the active player at state s_1; she can either select a constant payoff of 1 by choosing action a_0, or choose one of the other actions a_i, each of which leads to a different subgame G_i. If x fixes the policy x that that picks a_0 with probability 1, then *any* policy for player y

is a best response to x. Clearly, an arbitrary strategy might do very poorly if player x happens to play some action other than a_0. In general, if a behavior strategy x rules out any information sets for y, then a best-response behavior policy for y can specify *arbitrary* actions at those ruled out information sets.

This could cause problems for the double oracle algorithm. Given x^{mix} on some iteration of the algorithm, we will compute $y = \text{BR}_{\mathsf{y}}((x^{\mathsf{mix}})^T M)$ and add y to $\mathcal{B}_{\mathsf{y}}$. Future strategies for y will then be constructed as distributions over strategies in the bundle. Suppose x^{mix} happens to only ever play action a_1 from s_1 (in Figure (5.6)). Then, the response y will likely be "good" for the subgame G_1, but terrible at all the other subgames . This means y will be of limited use in constructing a good mixed strategy, assuming x starts playing actions other than a_1 from s_1 (for example, x^{cntr} might play many different actions at s_1).

In general, it is possible that x^{mix}, which is produced by solving the matrix game $\tilde{M}$, might be a mixture of relatively few deterministic best responses, and so it might rule out many information sets for y. We call such problematic policies "sparse," because they put zero probability on some actions, hence making large parts of the game tree unreachable. This, in turn, will produce cost vectors $c = x^T M$ that are sparse in the usual since of having many zero entries. We refer to policies that play all actions with some (possibly small) positive probability as dense strategies.

We would like y's best response to "fall back" on some reasonable behavior when selecting a best-response behavior at an unreachable information set. We can achieve this in the following manner: suppose x^{cntr} is a dense strategy for x. Then, instead of computing $\text{BR}_{\mathsf{y}}(x)$ directly, we compute $y = \text{BR}_{\mathsf{y}}((1-\epsilon)x + \epsilon x^{\mathsf{cntr}})$ for some small constant ϵ. The strategy y will still be an arbitrarily good response to x (as ϵ goes to zero), but at any information set that isn't reached by x, it will play some best-response action to x^{cntr}, since x^{cntr} doesn't rule out any information sets.

We address these issues in DOBA+ in the following way:

- Our implementation of the best-response oracle for EFGs picks the uniform distribution over all best-response actions at each information set, rather than just picking one. This assures that the strategies we add to our bundle are as dense as possible, and so hopefully our mixed strategies will not rule out too many information sets. It is also possible to use "soft" best responses that put positive probability on all behaviors. Investigating such approximate best-response oracles is an important avenue for future work.

- When we solve the matrix game $\tilde{M}$, we add an additional constraint that requires the

mixtures p and q to put some small weight (we use $\epsilon = 0.0001$) on the current best strategy for each player (that is, on the x^{cntr} and y^{cntr} corresponding to the current ub and lb). These centers tend to be dense, as they are mixtures of many strategies, and so this ensures that when we compute a best response to x^{mix}, the best response will at worst fall back to playing a best response to y^{cntr}.

We have seen that some best-response strategies are worse than others; we now consider the other side of the spectrum, considering superior best responses. DOBA+ does not currently use the concepts in the next section, but we hope to exploit these ideas algorithmically in future work.

Optimal best responses The problem of solving the convex game (X, Y, M) can be written as

$$\min_{x \in X} V(x)$$

where $V(x)$ is the convex function

$$V(x) = \max_{y \in Y} x^T M y.$$

The best-response problem for a fixed $x \in X$ is simply to compute $V(x)$ (and a y that achieves this value). But, as we saw in the previous section, there may be many possible best responses. Any convex combination of best responses is also a best response, and so the set of all best responses to x is convex: call this set $Y_{\mathsf{br}}(x)$, that is,

$$Y_{\mathsf{br}}(x) = \{y \in Y \mid y \text{ is a best response to } x\}.$$

We define the *optimal best response* as the solution to the game where y is restricted to play from the set $Y_{\mathsf{br}}(x)$, but x can play arbitrarily (in particular, she is under no obligation to actually play x). This is exactly the restriction of the original convex game to $(X, Y_{\mathsf{br}}(x), M)$, which has value

$$\max_{y \in Y_{\mathsf{br}}(x)} \min_{x \in X} x^T M y.$$

Let $\mathrm{BR}^*_{\mathsf{x}}$ compute an optimal best response. Computing an optimal best response is in general as hard as solving an arbitrary convex game. For example, consider the game of Figure (5.6). Computing the optimal best response for player y to the policy for x that always plays a_0 requires finding a minimax optimal policy for G_1, G_2, and all the other possible subgames. However, in some cases computing an optimal best response can be much easier than solving the overall game.

The set $Y_{\mathsf{br}}(x)$ for an extensive-form game has a straight-forward interpretation: it corresponds to the set of behavior strategies that only put positive probability on actions that are local best responses to x (or, more precisely, actions that are local best responses to the value function on player y's information set tree induced by the strategy x). Thus, we can efficiently optimize any linear function over the set $Y_{\mathsf{br}}(x)$ by using the standard linear-time dynamic programming algorithm for computing a best response for an EFG; we simply run the algorithm on the restricted information set tree where we have discarded all actions that are not local best responses.

Optimal best-response strategies have several nice properties. It is easy to verify that an optimal best response $y^* = \text{BR}^*_{\mathsf{y}}(x^*)$ to a minimax strategy x^* is a minimax optimal strategy for y. It can also be shown that for an arbitrary x, computing an optimal best response to x corresponds to finding a direction of steepest feasible decent with respect to $V(x)$ from x: we can think of $y' = \text{BR}^*_{\mathsf{y}}(x)$ as defining the best (that is, minimal slope) mixture of subgradients at x.

5.5 Experimental Results

In this section we test the algorithms introduced on both the sensor-placement / observation-avoidance adversarial MDP problem, and on an extensive-form game representations of Rhode Island Hold'em poker and approximated Texas Hold'em poker.

5.5.1 Adversarial-cost MDPs

We consider the example sensor placement / avoidance game described in Section 3.4.2. We model the robot's path planning problem by discretizing a given map at a resolution of between 10 and 50 cm per cell, producing grids of size 269×226 to 54×45. We do not model the robot's orientation and rely on lower level navigation software to move the robot along planned trajectories. Each cell corresponds to a state in the MDP.

The transition model we use gives the robot 16 actions, corresponding to movement in any of 16 compass directions. Movement in the directions N, S, E, and W corresponds to moving to an adjacent cell, NE, SE, SW, and NE correspond to moving to an adjacent cell diagonally, and the other eight directions (NNE, etc), correspond to moving two cells in one direction, and one cell in an orthogonal direction. Allowing movement in 16 directions means distances in the discretized world approximate a Euclidean distance metric, rather than the Manhattan (L1) metric implied by only allowing movement in the 4 cardinal

	grid size	k	LP	basic DOBA	iter
A	54 x 45	32	56.8 s	1.9 s	15
B	54 x 45	328	104.2 s	8.4 s	47
C	94 x 79	136	2835.4 s	10.5 s	30
D	135 x 113	32	1266.0 s	10.2 s	14
E	135 x 113	92	8713.0 s	18.3 s	30
F	269 x 226	16	-	39.8 s	17
G	269 x 226	32	-	41.1 s	15

Table 5.1: Sample problem discretizations, number of sensor placements available to the opponent, solution time solving Equation (5.4) with CPLEX, and solution time and number of iterations using the basic Double Oracle Algorithm.

directions. Each cell s has a cost weight $m(s)$ for movement through that cell; in our experiments all of these are set to 1.0 for simplicity. The actual movement costs for each action are calculated by considering the distance traveled (either 1, $\sqrt{2}$, or $\sqrt{5}$) weighted by the movement costs assigned to each cell. For example, movement in one of the four cardinal directions from a state u to a state v incurs cost $0.5m(u) + 0.5m(v)$.

Cells observed by a sensor have an additional cost given by a linear function of the distance to the sensor. An additional cost of 20 is incurred if observed by an adjacent sensor, and cost 10 is incurred if the sensor is at the maximum distance. The ratio of movement cost to observation cost determines the planner's relative preference for paths with low expected observation times versus short paths. We assume a fixed start location for our robot in all problems, so pure strategies can be represented as paths.

We present experiments using both the single oracle algorithm and the basic double oracle algorithm on this domain. The basic DOBA was sufficient because at most 30 iterations were needed to solve our test problems. Both algorithms use Dijkstra's algorithm for the planning player's oracle BR_x.

The column (cost player) oracle for the double oracle algorithm is the following naive one: For an arbitrary matrix game M, an oracle may be created (say BR_y for the column player) by finding the minimum entry in the vector $p^T M$ when the row player plays mixed strategy p. Perhaps surprisingly, we show that even using such a naive oracle can yield performance improvements over the single oracle algorithm. If sensor fields of view have limited overlap, then a fast best-response oracle for multiple sensor placement can also be constructed by considering each sensor independently and using the result to bound the cost vector for a pair of sensors

Our implementation is in Java, with an external call to CPLEX 7.1 [ILOG, Inc., 2003] for solving all linear programs. For comparison, we also used CPLEX to solve the linear program (5.4) directly (without any decompositions).

All results given in this paper correspond to the map in Figure (3.4) (found in Chapter 3). We performed experiments on other maps as well, but we do not report them because the results were qualitatively very similar. We solved the problem using various discretizations and different numbers of potential cost vectors to demonstrate the scaling properties of our algorithms. These problem discretizations are shown in Table (5.5.1), along with their double oracle and direct LP solution times. The letters in the table correspond to those in Figure (5.7), which compares the double and single oracle algorithms. All times are wall-clock times on a 1 GHz Pentium III machine with 512M main memory. Results reported are the average over 5 runs. Standard deviations were insignificant, less than 1/10th of total solve time in all cases.

Our results indicate that both the double and single oracle algorithms significantly outperform directly solving the linear program. This improvement in performance is possible because our algorithms take advantage of the fact that the linear program (5.4) is "almost" an MDP, and the planner's row oracle is implemented with Dijkstra's algorithm, which is much faster than general LP solvers. The particularly lopsided times for problems C, D, and E may have been partially caused by CPLEX running low on physical memory; we didn't try solving the LPs for problems F and G because they are even larger. One of the benefits of our decomposition algorithms is their lower memory usage, but even when memory was not an issue our algorithms were significantly faster than CPLEX.

As Figure (5.7) shows, the basic double oracle algorithm outperforms the single oracle version for all problems. The difference is most pronounced on problems with a large number of cost vectors. The time for solving the master LPs and for the column oracle are insignificant, so the performance gained by the double oracle algorithm is explained by its implicit preference for mixed strategies with small support, and the correspondingly smaller $\tilde{M}$.

We ran the oracle algorithms with $\epsilon = 0.005$, which is effectively optimal considering that a single step of movement has cost 1.0, and the minimum cost for being observed is 10.0. Thus, assuming the model expressed by the linear program is accurate, our algorithms produce the best result possible. In practice, it may be that the costs need to be adjusted to obtain the desired result; for example, in our path-planning problem there is a trade-off between shortest paths and being observed. By adjusting the balance between the opponent-controlled and fixed movement costs of the problem, the algorithm can be made to weigh this trade-off differently. Finding the proper balance for a particular problem may require some tweaking of model parameters.

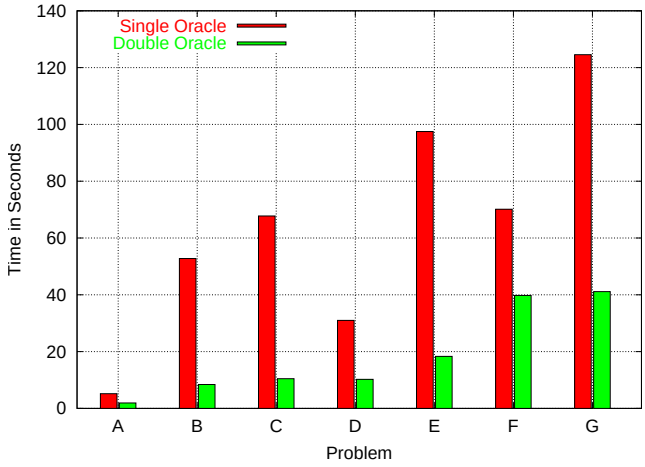


Figure 5.7: Double and single oracle algorithm performance on problems shown in Table (5.5.1).

We also ran some limited experiments in the online setting, using the no-regret algorithm of Kalai and Vempala [2003] to play the repeated game as discussed in Section 3.1.2. Figure (5.8) shows 100 trajectories against an opponent who plays a minimax optimal sensor placement: that is, for each round the adversary samples a sensor placement $c \in K$ according to a minimax optimal distribution q. The Kalai-Vempala algorithm chooses a best response to the average cost vector played by the adversary so far, perturbed with a small amount of randomness. The implementation is straightforward, and the generated paths are reasonable. The random perturbations to the cost vectors introduced by the Kalai-Vempala algorithm tend to produce paths that are somewhat choppy, but these paths can be smoothed by lower-level control routines. As expected, the algorithm converges to a best response to the minimax optimal q played by the adversary.

For these MDP-based problems, there is the possibility for a very large speedup through the use of more sophisticated best-response oracles for the planning player: in our experiments, the planner's oracle calls typically take 90% or more of the total runtime. We used Dijkstra's algorithm to solve our deterministic best-response problem. However, it is straightforward to construct reasonable heuristics for this problem, for example by using the L1 distance for the movement costs plus observation costs. Thus, we could apply the A^*-search algorithm. Further, there is likely to similarity between the optimal solutions from one round to the next. To take advantage of this, we could use an incremental A^* implementation, for example that of Koenig and Likhachev [2001].

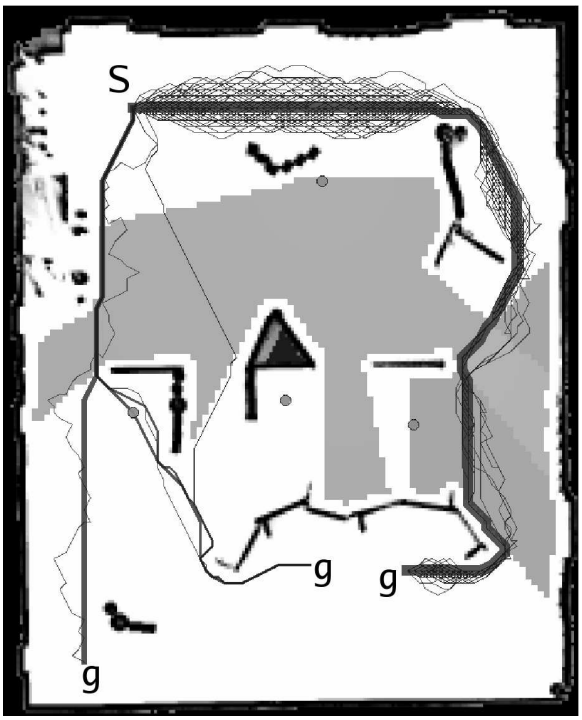


Figure 5.8: Convergence to optimal response in the online setting. The thin lines indicate 100 trajectories produced by the Kalai-Vempala algorithm against a fixed minimax solution of the opponent. The wider lines indicate a minimax solution for the planner. The erratic nature of the Kalai-Vempala paths is caused by the randomness in the cost vectors introduced by that algorithm: the small jogs in the path are caused by the robot driving around small areas where a higher cost has been hallucinated.

5.5.2 Extensive-form Game Experiments

We tested the DOBA+ algorithm and fictitious play on abstracted Rhode Island Hold'em. We chose this as a representative problem both because Rhode Island Hold'em is a well-known AI testbed and because the abstracted version is one of the largest extensive-form games that can be solved in a reasonable amount of time using CPLEX's barrier method implementation on a modern workstation; the game was first solved by Gilpin and Sandholm [2005]. CPLEX's performance provides one benchmark against which to evaluate our results.

Rhode Island Hold'em (RIH) was introduced by Shi and Littman [2001] as a challenge

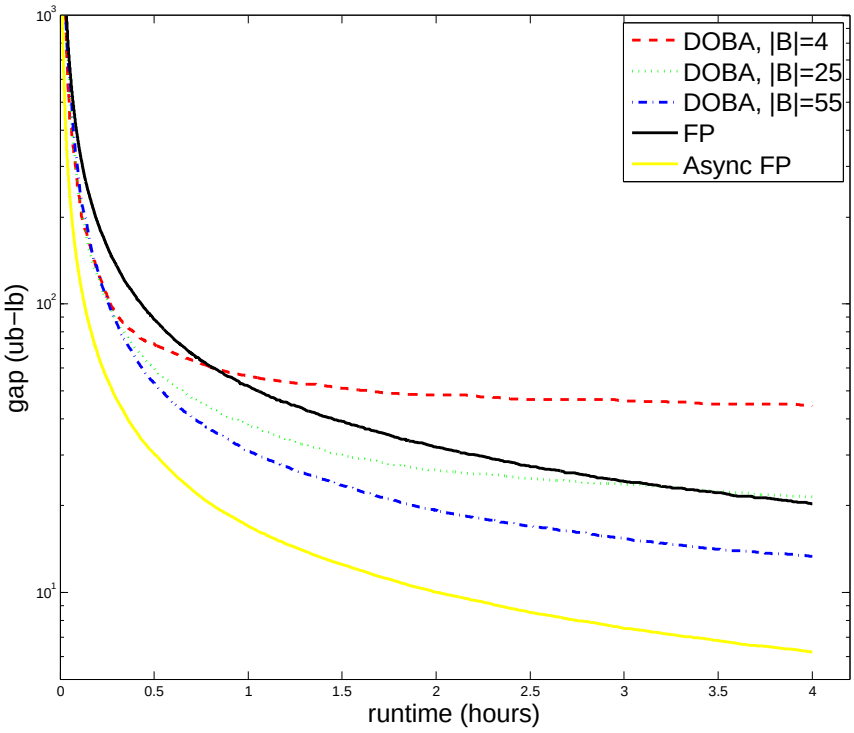


Figure 5.9: Algorithm runtime versus approximation error on Rhode Island hold'em. The Y axis is a log scale plot of ϵ for the best approximate solution the algorithms can return at a given time, in units of $\$0.01$.

problem for AI research. The game is similar to two-player limit Texas Hold'em. It is played with a full deck of 52 cards, but each player receives only a single face-down hole card, and there are only two community cards. There are three rounds of betting, with up to three raises per betting round. Unabstracted Rhode Island Hold'em has a game tree with 3.1 billion nodes, which is still too large to work with conveniently. Instead, Andrew Gilpin was kind enough to provide us with the convex game representation produced by the GameShrink algorithm [Gilpin and Sandholm, 2005]. Sparsely represented, this game has approximately 50×10^6 non-zeros in the payoff and sequence constraint matrices, with dimensions $m = n = 883,741$, taking almost 600MB of memory to store. A solution to this game can be converted to a payoff-equivalent strategy for the unabstracted game. The poker game has $\$5.00$ antes and a maximum pot size of $\$310.00$. The uniform random strategy, from which we started both our algorithm and fictitious play, loses approximately $\$290.00$ per game. The minimax value of the game is $-\$0.64$; the value is negative because

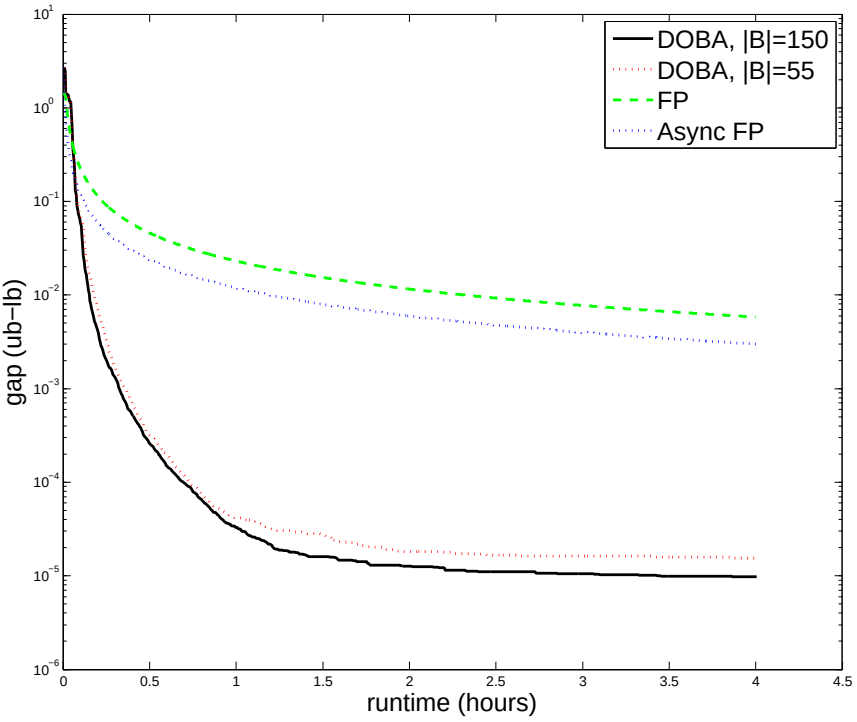


Figure 5.10: Algorithm runtime versus approximation error on approximately abstracted Texas hold'em. Note that with both bundle sizes, DOBA+ significantly outperforms both versions of fictitious play.

player x (the minimizing player) bets second, and thus gains a small advantage based on the information revealed by the first player's initial bet.

The CPLEX commercial linear programming package solved abstracted Rhode Island Hold'em via the barrier method in about 7.5 days, using 25 GB of memory; achieving an $\epsilon = \$0.20$ approximate minimax solution took 110.3 hours, or over 4.5 days. The DOBA+ produced a solution of that quality in 130 minutes; see Figure (5.11).

Figure (5.9) compares the anytime performance of DOBA+ and fictitious play (FP). Both algorithms were initialized using the uniform-random behavior strategy for both players; that is, at each information set the agent selects an action uniformly at random. Especially early on, DOBA+ can produce higher quality solutions for a given amount of time. For example, it takes DOBA+ 11 minutes to bound the value of the game in $[\$0.00, -\$1.35]$, thereby proving that player x has an advantage. It takes synchronous FP

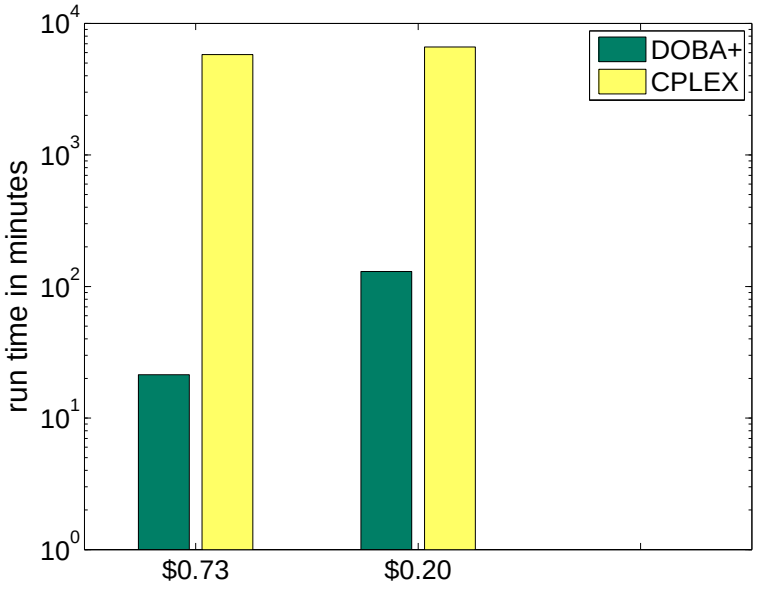


Figure 5.11: Comparison of runtimes for CPLEX and the double oracle Bundle Algorithm (DOBA+) to produce an $\epsilon = \$0.20$ and $\epsilon = \$0.73$ solution to abstracted RIH. Note that the runtimes are on a log scale.

(as in Figure (5.1)) about 20 minutes to get comparable bounds, but asynchronous FP[5] takes only 10 minutes to get these bounds. Experiments on smaller approximately abstracted versions of Rhode Island Hold'em, however, show DOBA+ can outperform FP by an order of magnitude; on the smallest problem we tested (with $m = n = 10^4$), CPLEX outperformed both algorithms.

Figure (5.10) shows results for experiments on an approximately abstracted instance of Texas Hold'em, similar to those discussed in [Gilpin and Sandholm, 2006a]. This instance only models the first three rounds of betting. This problem has significantly more non-zeros (130×10^6) than the Rhode Island hold'em instance, and requires about 1.5GB of memory to represent. However, this problem has lower dimensionality, with $m = n = 236{,}416$. The instance we tested has a small blind (similar to an ante) of

[5]Asynchronous FP does the updates for one player, say ×, before the updates for the other player on each iteration. That is, asynchronous FP is the algorithm of Figure (5.1) where all the statements in the left column in the while loop are executed before the statements in the right column. Note that DOBA+ with ϕ fixed at 1.0 effectively executes asynchronous FP. For many problems, solving the small matrix game takes a negligible amount of time, and so making DOBA+ fully asynchronous by re-solving the matrix game between each call to **updateCenter** may be advantageous [Krause, 2006].

$\$0.50$, and a big blind of $\$1.00$. For this problem, DOBA+ significantly outperformed asynchronous fictitious play: FP bounded the value of the game in $[-0.028, -0.046]$ in a 2 hour run, while DOBA+ achieved better bounds in less than 6 minutes. We used DOBA+ with $\phi = 1$ as our asynchronous FP implementation, and this incurred extra overhead. A direct implementation might allow a 2-3x performance increase, but clearly this would have made little difference for the Texas Hold'em instance.

We conclude that none of these algorithms are a clear winner even when only considering extensive-form games, and so the general problem of designing fast algorithms for all types of convex games is still very much open.

Chapter Acknowledgments Special thanks to Andrew Gilpin who graciously provided us with the sequence-weight representation for the poker problems used in the experiments, and also shared his data from experiments using CPLEX to solve the same instances.

Chapter 6

Online Geometric Optimization in the Bandit Setting

In this chapter we describe a new algorithm for solving online linear programming problems in the bandit setting when facing an adaptive adversary; this work originally appeared in [McMahan and Blum, 2004]. Though it has many potential applications, the algorithm we develop is particularly applicable to the case of a convex game (X, Y, M) that is played repeatedly by player x, in the manner discussed in Section 3.1.2. The approach here is novel in that it does not require observation of the opponent's strategy y on each iteration, or even the cost vector My. Rather, we can run the algorithm of this chapter for player x as long as the total payoff $x^T My$ is observed on each round.

6.1 Introduction and Background

Kalai and Vempala [2003] give an elegant, efficient algorithm for a broad class of online optimization problems. In their setting, we have an arbitrary (bounded) set $S \subseteq \mathbb{R}^n$ of feasible points. At each time step t, an online algorithm $\mathcal{A}$ must select a point $\mathbf{x}^t \in S$ and simultaneously an adversary selects a cost vector $\mathbf{c}^t \in \mathbb{R}^n$ (throughout the chapter we use superscripts to index iterations). The algorithm then observes $\mathbf{c}^t$ and incurs cost $\mathbf{c}^t \cdot \mathbf{x}^t$. Kalai and Vempala show that so long as we have an efficient algorithm for the *offline* problem (given $\mathbf{c} \in \mathbb{R}^n$ find $\mathbf{x} \in S$ to minimize $\mathbf{c} \cdot \mathbf{x}$) and so long as the cost vectors are bounded, we can efficiently solve the *online* problem of performing nearly as well as the best fixed $\mathbf{x} \in S$ in hindsight. This generalizes the classic "expert advice" problem, because we do not require the set S to be represented explicitly: we just need an efficient

oracle for selecting the best $\mathbf{x} \in S$ in hindsight. Further, it decouples the number of experts from the underlying dimensionality n of the decision set, under the assumption the cost of a decision is a linear function of n features of the decision. The standard experts setting can be recovered by letting $S = \{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$, the columns of the $n \times n$ identity matrix.

A problem that fits naturally into this framework is an online shortest path problem where we repeatedly travel between two points a and b in some graph whose edge costs change each day (say, due to traffic). In this case, we can view the set of paths as a set S of points in a space of dimension equal to the number of edges in the graph, and $\mathbf{c}^t$ is simply the vector of edge costs on day t. Even though the number of paths in a graph can be exponential in the number of edges (i.e., the set S is of exponential size), since we can solve the shortest path problem for any *given* set of edge lengths, we can apply the Kalai-Vempala algorithm. (Note that a different algorithm for the special case of the online shortest path problem is given by Takimoto and Warmuth [2002].)

A natural generalization of the above problem, considered by Awerbuch and Kleinberg [2004], is to imagine that rather than being given the entire cost vector $\mathbf{c}^t$, the algorithm is simply told the cost incurred $\mathbf{c}^t \cdot \mathbf{x}^t$. For example, in the case of shortest paths, rather than being told the lengths of all edges at time t, this would correspond to just being told the total time taken to reach the destination. Thus, this is the "bandit version" of the Kalai-Vempala setting. Awerbuch and Kleinberg present two results: an algorithm for the general problem in the presence of an *oblivious* adversary, and an algorithm for the special case of the shortest path problem that works in the presence of an *adaptive* adversary. The difference between the two adversaries is that an oblivious adversary must commit to the entire sequence of cost vectors in advance, whereas an adaptive adversary may determine the next cost vector based on the online algorithm's play (and hence, the information the algorithm received) in the previous time steps. Thus, an adaptive adversary is in essence playing a repeated game. They leave open the question of achieving good regret guarantees for an adaptive adversary in the general setting.

In this chapter, we solve the open question of [Awerbuch and Kleinberg, 2004], giving an algorithm for the general bandit setting in the presence of an adaptive adversary. Moreover, our method is significantly simpler than the special-purpose algorithm of Awerbuch and Kleinberg for shortest paths. Our bounds are somewhat worse: we achieve regret bounds of $\mathcal{O}(T^{3/4}\sqrt{\ln T})$ compared to the $\mathcal{O}(T^{2/3})$ bounds of [Awerbuch and Kleinberg, 2004].

The basic idea of our approach is as follows. We begin by noticing that the only history information used by the Kalai-Vempala algorithm in determining its action at time t is the sum $\mathbf{c}^{1:t-1} = \sum_{\tau=1}^{t-1} \mathbf{c}^\tau$ of all cost vectors received so far (we use this abbreviated notation for sums over iteration indexes throughout the chapter). Furthermore, the way

this is used in the algorithm is by adding random noise $\boldsymbol{\mu}$ to this vector, and then calling the offline oracle to find the $\mathbf{x}^t \in S$ that minimizes $(\mathbf{c}^{1:t-1} + \boldsymbol{\mu}) \cdot \mathbf{x}^t$. So, if we can design a bandit algorithm that produces an estimate $\hat{\mathbf{c}}^{1:t-1}$ of $\mathbf{c}^{1:t-1}$, and show that with high probability even an adaptive adversary will not cause $\hat{\mathbf{c}}^{1:t-1}$ to differ too substantially from $\mathbf{c}^{1:t-1}$, we can then argue that the distribution $\hat{\mathbf{c}}^{1:t-1} + \boldsymbol{\mu}$ is close enough to $\mathbf{c}^{1:t-1} + \boldsymbol{\mu}$ for the Kalai-Vempala analysis to apply. In fact, to make our analysis a bit more general, so that we could potentially use other algorithms as subroutines, we will argue a little differently. Let $\mathrm{OPT}(\mathbf{c}) = \min_{\mathbf{x} \in S}(\mathbf{c} \cdot \mathbf{x})$. We will show that with high probability, $\mathrm{OPT}(\hat{\mathbf{c}}^{1:T})$ is close to $\mathrm{OPT}(\mathbf{c}^{1:T})$ and $\hat{\mathbf{c}}^{1:T}$ satisfies conditions needed for the subroutine to achieve low regret on $\hat{\mathbf{c}}^{1:T}$. This means that our subroutine, which believes it has seen $\hat{\mathbf{c}}^{1:T}$, will achieve performance on $\hat{\mathbf{c}}^{1:T}$ close to $\mathrm{OPT}(\mathbf{c}^{1:T})$. We then finish off by arguing that our performance on $\mathbf{c}^{1:T}$ is close to its performance on $\hat{\mathbf{c}}^{1:T}$.

The behavior of the bandit algorithm will in fact be fairly simple. We begin by choosing a basis B of (at most) n points in S to use for sampling (we address the issue of how B is chosen when we describe our algorithm in detail). Then, at each time step t, with probability γ we *explore* by playing a random basis element, and otherwise (with probability $1 - \gamma$) we *exploit* by playing according to the Kalai-Vempala algorithm. For each basis element $\mathbf{b}_j$, we use our cost incurred while exploring with that basis element, scaled by n/γ, as an estimate of $\mathbf{c}^{1:t-1} \cdot \mathbf{b}_j$. Using martingale tail inequalities, we argue that even an adaptive adversary cannot make our estimate differ too wildly from the true value of $\mathbf{c}^{1:t-1} \cdot \mathbf{b}_j$, and use this to show that after matrix inversion, our estimate $\hat{\mathbf{c}}^{1:t-1}$ is close to its correct value with high probability.

6.2 Problem Formalization

We can now fully formalize the problem. First, however, we establish a few notational conventions. As mentioned previously, we use superscripts to index iterations (or rounds) of our algorithm, and use the abbreviated summation notation $\mathbf{c}^{1:t}$ when summing variables over iterations. Vectors quantities are indicated in bold, and subscripts index into vectors or sets. Hats (such as $\hat{\mathbf{c}}^t$) denote estimates of the corresponding actual quantities. The variables and constants used in this chapter are summarized in Table (6.1).

As mentioned above, we consider the setting of [Kalai and Vempala, 2003] in which we have an arbitrary (bounded) set $S \subseteq \mathbb{R}^n$ of feasible points. At each time step t, the online algorithm $\mathcal{A}$ must select a point $\mathbf{x}^t \in S$ and simultaneously an adversary selects a cost vector $\mathbf{c}^t \in \mathbb{R}^n$. The algorithm then incurs cost $\mathbf{c}^t \cdot \mathbf{x}^t$. Unlike [Kalai and Vempala, 2003], however, rather than being told $\mathbf{c}^t$, the algorithm simply learns its cost $\mathbf{c}^t \cdot \mathbf{x}^t$.

For simplicity, throughout this chapter we assume a fixed adaptive adversary $\mathcal{V}$ and time horizon T. Since our choice of algorithm parameters depends on T, we assume[1] T is known to the algorithm. We refer to the sequence of decisions made by the algorithm so far as a decision history, which can be written $h^t = [\mathbf{x}^1, \ldots, \mathbf{x}^t]$. Let H^* be the set of all possible decision histories of length 0 through $T-1$. Without loss of generality [see Auer et al., 1995, for example], we assume our adaptive adversary is deterministic, as specified by a function $\mathcal{V} : H^* \to \mathbb{R}^n$, a mapping from decision histories to cost vectors. Thus, $\mathcal{V}(h^{t-1}) = \mathbf{c}^t$ is the cost vector for timestep t.

We can view our online decision problem as a game, where on each iteration t the adversary $\mathcal{V}$ selects a new cost vector $\mathbf{c}^t$ based on h^{t-1}, and the online algorithm $\mathcal{A}$ selects a decision $\mathbf{x} \in S$ based on its past plays and observations, and possibly additional hidden state or randomness. Then, $\mathcal{A}$ pays $\mathbf{c}^t \cdot \mathbf{x}^t$ and observes this cost. For our analysis, we assume a L_1 bound on S, namely $\|\mathbf{x}\|_1 \leq D/2$ for all $\mathbf{x} \in S$, so $\|\mathbf{x} - \mathbf{y}\|_1 \leq D$ for all $\mathbf{x}, \mathbf{y} \in S$. We also assume that $|\mathbf{c} \cdot \mathbf{x}| \leq M$ for all $\mathbf{x} \in S$ and all $\mathbf{c}$ played by $\mathcal{V}$. We also assume S is full rank: if it is not we simply project to a lower-dimensional representation. Some of these assumptions can be lifted or modified, but this set of assumptions simplifies the analysis.

For a fixed decision history h^T and cost history $k^T = (\mathbf{c}^1, \ldots, \mathbf{c}^T)$, we define $\mathrm{loss}(h^T, k^T) = \sum_{t=1}^{T} (\mathbf{c}^t \cdot \mathbf{x}^t)$. For a randomized algorithm $\mathcal{A}$ and adversary $\mathcal{V}$, we define the random variable $\mathrm{loss}(\mathcal{A}, \mathcal{V})$ to be $\mathrm{loss}(h^T, k^T)$, where h^T is drawn from the distribution over histories defined by $\mathcal{A}$ and $\mathcal{V}$, and $k^T = (\mathcal{V}(h^0), \ldots, \mathcal{V}(h^{T-1}))$. When it is clear from context, we will omit the dependence on $\mathcal{V}$, writing only $\mathrm{loss}(\mathcal{A})$.

Our goal is to define an online algorithm with low regret. That is, we want a guarantee that the total loss incurred will, in expectation, not be much larger than the optimal strategy in hindsight against the cost sequence we actually faced. To formalize this, first define an oracle $\mathcal{R} : \mathbb{R}^n \to S$ that solves the offline optimization problem, $\mathcal{R}(\mathbf{c}) = \mathrm{argmin}_{\mathbf{x} \in S}(\mathbf{c} \cdot \mathbf{x})$. We then define $\mathrm{OPT}(k^T) = \mathbf{c}^{1:T} \cdot \mathcal{R}(\mathbf{c}^{1:T})$. Similarly, $\mathrm{OPT}(\mathcal{A}, \mathcal{V})$ is the random variable $\mathrm{OPT}(k^T)$ when k^T is generated by playing $\mathcal{V}$ against $\mathcal{A}$. We again drop the dependence on $\mathcal{V}$ and $\mathcal{A}$ when it is clear from context. Formally, we define expected regret as

$$E\left[\mathrm{loss}(\mathcal{A}, \mathcal{V}) - \mathrm{OPT}(\mathcal{A}, \mathcal{V})\right] = E[\mathrm{loss}(\mathcal{A}, \mathcal{V})] - E\left[\min_{\mathbf{x} \in S} \sum_{t=1}^{T} (\mathbf{c}^t \cdot \mathbf{x})\right]. \tag{6.1}$$

Note that the $E[\mathrm{OPT}(\mathcal{A}, \mathcal{V})]$ term corresponds to applying the min operator separately to each possible cost history to find the best fixed decision with respect to that particular

[1] One can remove this requirement by guessing T, and doubling the guess each time we play longer than expected (see, for example, Theorem 6.4 from Auer et al. [2002]).

Choose parameters γ and ϵ, where ϵ is a parameter of GEX
$t = 1$
Fix a basis $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\} \subseteq S$
while playing **do**
  Let $\chi^t = 1$ with probability γ and $\chi^t = 0$ otherwise
  if $\chi^t = 0$ **then**
    Select $\mathbf{x}^t$ from the distribution GEX$(\hat{\mathbf{c}}^1, \ldots, \hat{\mathbf{c}}^{t-1})$
    Incur cost $z^t = \mathbf{c}^t \cdot \mathbf{x}^t$
    $\hat{\mathbf{c}}^t = 0 \in \mathbb{R}^n$
  else
    Draw j uniformly at random from $\{1, \ldots, n\}$
    $\mathbf{x}^t = \mathbf{b}_j$
    Incur cost and observe $z^t = \mathbf{c}^t \cdot \mathbf{x}^t$
    Define $\hat{\boldsymbol{\ell}}^t$ by $\hat{\ell}^t_i = 0$ for $i \neq j$ and $\hat{\ell}^t_j = (n/\gamma)z^t$
    $\hat{\mathbf{c}}^t = (B^\dagger)^{-1}\hat{\boldsymbol{\ell}}^t$
  end if
  $\hat{\mathbf{c}}^{1:t} = \hat{\mathbf{c}}^{1:t-1} + \hat{\mathbf{c}}^t$
  $t = t + 1$
end while

Figure 6.1: The bandit-style geometric decision algorithm against an adaptive adversary (BGA).

cost history, and then taking the expectation with respect to these histories. Auer et al. [1995] give an alternative weaker definition of regret. We discuss relationships between the definitions in Appendix D.

6.3 Algorithm

We introduce an algorithm we call BGA, standing for *Bandit-style Geometric decision algorithm against an Adaptive adversary*. The algorithm alternates between playing decisions from a fixed basis to get unbiased estimates of costs, and playing (hopefully) good decisions based on those estimates. In order to determine the good decisions to play, it uses some online geometric optimization algorithm for the full-observation problem. We denote this algorithm by GEX (*Geometric Experts algorithm*). The implementation of GEX

we analyze is based on the FPL algorithm of Kalai and Vempala [2003]; we detail this implementation and analysis in Appendix C. However, other algorithms could be used, for example the algorithm of Zinkevich [2003] when S is convex. We view GEX as a function from the sequence of previous cost vectors $(\hat{\mathbf{c}}^1, \ldots, \hat{\mathbf{c}}^{t-1})$ to distributions over decisions.

Pseudocode for our algorithm is given in Algorithm (1). On each timestep, we make decision $\mathbf{x}^t$. With probability $(1-\gamma)$, BGA plays a recommendation $\mathbf{x}^t = \tilde{\mathbf{x}}^t \in S$ from GEX. With probability γ, we ignore $\tilde{\mathbf{x}}^t$ and play a basis decision, $\mathbf{x}^t = \mathbf{b}_i$ uniformly at random from a sampling basis $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$. The indicator variable χ^t is 1 on exploration iterations and 0 otherwise.

Our sampling basis B is a $n \times n$ matrix with columns $\mathbf{b}_i \in S$, so we can write $\mathbf{x} = B\mathbf{w}$ for any $\mathbf{x} \in \mathbb{R}^n$ and weights $\mathbf{w} \in \mathbb{R}^n$. For a given cost vector $\mathbf{c}$, let $\boldsymbol{\ell} = B^\dagger \mathbf{c}$ (the superscript $\dagger$ indicates transpose). This is the vector of decision costs for the basis decisions, so $\ell_i^t = \mathbf{c}^t \cdot \mathbf{b}_i$. We define $\hat{\boldsymbol{\ell}}^t$, an estimate of $\boldsymbol{\ell}^t$, as follows: Let $\hat{\boldsymbol{\ell}}^t = 0 \in \mathbb{R}^n$ on exploitation iterations. If on an exploration iteration we play $\mathbf{b}_j$, then $\hat{\boldsymbol{\ell}}^t$ is the vector where $\hat{\ell}_i^t = 0$ for $i \neq j$ and $\hat{\ell}_j^t = \frac{n}{\gamma}(\mathbf{c}^t \cdot \mathbf{b}_j)$. Note that $\mathbf{c}^t \cdot \mathbf{b}_j$ is the observed quantity, the cost of basis decision $\mathbf{b}_j$. On each iteration, we estimate $\mathbf{c}^t$ by $\hat{\mathbf{c}}^t = (B^\dagger)^{-1}\hat{\boldsymbol{\ell}}^t$. It is straightforward to show that $\hat{\boldsymbol{\ell}}^t$ is an unbiased estimate of basis decision costs and that $\hat{\mathbf{c}}^t$ is an unbiased estimate of $\mathbf{c}^t$ on each timestep t.

The choice of the sampling basis plays an important role in the analysis of our algorithm. In particular, we use a barycentric spanner, introduced in [Awerbuch and Kleinberg, 2004]. A barycentric spanner $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ is a basis for S such that $\mathbf{b}_i \in S$ and for all $\mathbf{x} \in S$ we can write $\mathbf{x} = B\mathbf{w}$ with coefficients $w_i \in [-1, 1]$. It may not be easy to find exact barycentric spanners in all cases, but Awerbuch and Kleinberg [2004] prove they always exist and gives an algorithm for finding 2-approximate barycentric spanners (where the weights $w_i \in [-2, 2]$), which is sufficient for our purposes.

6.4 Analysis

6.4.1 Preliminaries

At each time step, BGA either (with probability $1-\gamma$) plays the recommendation $\tilde{\mathbf{x}}^t$ from GEX, or else (with probability γ) plays a random basis vector from B. For purposes of analysis, however, it will be convenient to imagine that we request a recommendation $\tilde{\mathbf{x}}^t$ from GEX on every iteration, and also that we randomly pick a basis to explore, $\mathbf{b}^t \in \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, on each iteration. We then decide to play either $\tilde{\mathbf{x}}^t$ or $\mathbf{b}^t$ based on the

$S \subseteq \mathbb{R}^n$	set of decisions, a compact subset of $\mathbb{R}^n$
$D \in \mathbb{R}$	L_1 bound on diameter of S, $\forall \mathbf{x}, \mathbf{y} \in S,\ \lvert\mathbf{x} - \mathbf{y}\rvert_1 \leq D$
$n \in N$	dimension of decision space
h^t	decision history , $h^t = \mathbf{x}^1, \ldots, \mathbf{x}^t$
H^*	set of possible decision histories
$\mathcal{V} : H^* \to \mathbb{R}^n$	adversary, function from decision histories to cost vectors
$\mathcal{A}$	an online optimization algorithm
G^{t-1}	history of BGA randomness for timesteps 1 through $t-1$
$\mathbf{c}^t \in \mathbb{R}^n$	cost vector on time t
$\hat{\mathbf{c}}^t \in \mathbb{R}^n$	BGA's estimate of the cost vector on time t
$M \in \mathbb{R}^+$	bound on single-iteration cost, $\lvert\mathbf{c}^t \cdot \mathbf{x}^t\rvert \leq M$
$B \subseteq S$	sampling basis $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$
$\beta_\infty \in \mathbb{R}$	matrix max norm on $(B^\dagger)^{-1}$
$\boldsymbol{\ell}^t \in [-M, M]^n$	vector, $\ell_i^t = \mathbf{c}^t \cdot \mathbf{b}_i$ for $\mathbf{b}_i \in B$
$\hat{\boldsymbol{\ell}}^t \in \mathbb{R}^n$	BGA's estimate of $\boldsymbol{\ell}^t$
$T \in \mathbb{N}$	end of time, index of final iteration
$\mathbf{x}^t \in S$	BGA's decision on time t
$\tilde{\mathbf{x}}^t \in S$	decision recommended by GEX on time t
$\chi^t \in \{0, 1\}$	indicator, $\chi^t = 1$ if BGA explores on t, 0 otherwise
$\gamma \in [0, 1]$	the probability BGA explores on each timestep
$z^t \in [-M, M]$	BGA's loss on iteration t, $z^t = \mathbf{c}^t \cdot \mathbf{x}^t$,
$\hat{z}^t \in [-R, R]$	loss of GEX, $\hat{z}^t = \hat{\mathbf{c}}^t \cdot \tilde{\mathbf{x}}^t$

Table 6.1: Summary of notation used in the chapter.

outcome χ^t of a coin of bias γ. Thus, the complete history of the algorithm is specified by the *algorithm history* $G^{t-1} = [\chi^1, \tilde{\mathbf{x}}^1, \mathbf{b}^1, \chi^2, \tilde{\mathbf{x}}^2, \mathbf{b}^2, \ldots, \chi^{t-1}, \tilde{\mathbf{x}}^{t-1}, \mathbf{b}^{t-1}]$, which encodes all previous random choices. The sample space for all probabilities and expectations is the set of all possible algorithm histories of length T. Thus, for a given adversary $\mathcal{V}$, the various random variables and vectors we consider, such as $\mathbf{x}^t, \mathbf{c}^t, \hat{\mathbf{c}}^t, \tilde{\mathbf{x}}^t$, and others, can all be viewed as functions on the set of possible algorithm histories. Unless otherwise stated, our expectations and probabilities are with respect to the distribution over these histories.

A partial history G^{t-1} can be viewed a subset of the sample space (an event) consisting of all complete histories that have G^{t-1} as a prefix. We frequently consider conditional distributions and corresponding expectations with respect to partial algorithm histories. For instance, if we condition on a history G^{t-1}, the random variables $\mathbf{c}^1, \ldots, \mathbf{c}^t, \boldsymbol{\ell}^1, \ldots, \boldsymbol{\ell}^t$, $\hat{\boldsymbol{\ell}}^1, \ldots, \hat{\boldsymbol{\ell}}^{t-1}, \hat{\mathbf{c}}^1, \ldots \hat{\mathbf{c}}^{t-1}, \mathbf{x}^1, \ldots, \mathbf{x}^{t-1}$, and $\chi^1, \ldots, \chi^{t-1}$ are fully determined.

We now outline the general structure of our argument. Let $\hat{z}^t = \hat{\mathbf{c}}^t \cdot \tilde{\mathbf{x}}^t$ be the loss

perceived by the GEX on iteration t. In keeping with earlier definitions, loss(BGA) $= z^{1:T}$ and loss(GEX) $= \hat{z}^{1:T}$. We also let $\mathrm{OPT} = \mathrm{OPT}(\mathbf{BGA}, \mathcal{V}) = \mathbf{c}^{1:T} \cdot \mathcal{R}(\mathbf{c}^{1:T})$, the performance of the best post-hoc decision, and similarly $\widehat{\mathrm{OPT}} = \mathrm{OPT}(\hat{\mathbf{c}}^1, \ldots, \hat{\mathbf{c}}^T) = \hat{\mathbf{c}}^{1:t} \cdot \mathcal{R}(\hat{\mathbf{c}}^{1:t})$.

The base of our analysis is a bound on the loss of GEX with respect to the cost vectors $\hat{\mathbf{c}}^t$ of the form

$$E[\text{loss(GEX)}] \leq E[\widehat{\mathrm{OPT}}] + (\text{terms}). \tag{6.2}$$

Such a result is given in Appendix C, and follows from an adaptation of the analysis of Kalai and Vempala [2003]. We then prove statements having the general form

$$E[\text{loss(BGA)}] \leq E[\text{loss(GEX)}] + (\text{terms}) \tag{6.3}$$

and

$$E[\widehat{\mathrm{OPT}}] \leq E[\mathrm{OPT}] + (\text{terms}). \tag{6.4}$$

These statements connect our real loss to the "imaginary" loss of GEX, and similarly connect the loss of the best decision in GEX's imagined world with the loss of the best decision in the real world. Combining the results corresponding to Equations (6.2), (6.3), and (6.4) leads to an overall bound on the regret of BGA.

6.4.2 High Probability Bounds on Estimates

We prove a bound on the accuracy of BGA's estimates $\hat{\boldsymbol{\ell}}^t$, and use this to show a relationship between OPT and $\widehat{\mathrm{OPT}}$ of the form in Equation 6.4.

Define random variables $\mathbf{e}^0 = \mathbf{0}$ and $\mathbf{e}^t = \boldsymbol{\ell}^t - \hat{\boldsymbol{\ell}}^t$. We are really interested in the corresponding sums $\mathbf{e}^{1:t}$, where $e_i^{1:t}$ is the total error in our estimate of $\mathbf{c}^{1:t} \cdot \mathbf{b}_i$. We now bound $|e_i^{1:t}|$.

Theorem 6.4.1. *For $\lambda > 0$,*

$$\Pr\left[\left|e_i^{1:t}\right| \geq \lambda \frac{nM}{\gamma}\sqrt{t}\right] \leq 2e^{-\lambda^2/2}.$$

Proof. It is sufficient to show the sequence $\mathbf{e}^0, \mathbf{e}^1, \mathbf{e}^{1:2}, \mathbf{e}^{1:3}, \ldots, \mathbf{e}^{1:T}$ of random variables is a bounded martingale sequence with respect to the filter $G^0, G^1, \ldots, G^T$; that is, $E[e_i^{1:t} \mid G^{t-1}] = e_i^{1:t-1}$. The result then follows from Azuma's Inequality [see Motwani and Raghavan, 1995, for example]).

First, observe that $e_i^{1:t} = \ell_i^t - \hat{\ell}_i^t + e_i^{1:t-1}$. Further, the cost vector $\mathbf{c}^t$ is determined if we know G^{t-1}, and so ℓ_i^t is also fixed. Thus, accounting for the $\frac{\gamma}{n}$ probability we explore a particular basis decision $\mathbf{b}_i$, we have

$$E\left[e_i^{1:t} \mid G^{t-1}\right] = \frac{\gamma}{n}\left[\ell_i^t - \frac{n}{\gamma}\ell_i^t + e_i^{1:t-1}\right] + \left(1 - \frac{\gamma}{n}\right)\left[\ell_i^t - 0 + e_i^{1:t-1}\right] = e_i^{1:t-1},$$

and so we conclude that the $e_i^{1:t}$ forms a martingale sequence. Notice that $|e_i^{1:t} - e_i^{1:t-1}| = |\ell_i^t - \hat{\ell}_i^t|$. If we don't sample, $\hat{\ell}_i^t = 0$ and so $|e_i^{1:t} - e_i^{1:t-1}| \leq M$. If we do sample, we have $\hat{\ell}_i^t = \frac{n}{\gamma}\ell_i^t$, and so $|e_i^{1:t} - e_i^{1:t-1}| \leq \frac{nM}{\gamma}$. This bound is worse, so it holds in both cases. The result now follows from Azuma's inequality. □

Let $\beta_\infty = \|(B^\dagger)^{-1}\|_\infty$, a matrix L_∞-norm on $(B^\dagger)^{-1}$, so that for any $\mathbf{w}$, $\|(B^\dagger)^{-1}\mathbf{w}\|_\infty \leq \beta_\infty \|\mathbf{w}\|_\infty$.

Corollary 6.4.2. *For $\delta \in (0, 1]$, and all t from 1 to T,*

$$\Pr\left[\|\hat{\mathbf{c}}^{1:t} - \mathbf{c}^{1:t}\|_\infty \geq \beta_\infty J(\delta, \gamma)\sqrt{t}\right] \leq \delta.$$

where $J(\delta, \gamma) = \frac{1}{\gamma} nM\sqrt{2\ln(2n/\delta)}$.

Proof. Solving $\delta/n = 2e^{-\lambda^2/2}$ yields $\lambda = \sqrt{2\ln(2n/\delta)}$, and then using this value in Theorem (6.4.1) gives

$$\Pr\left[|e_i^{1:t}| \geq J(\delta, \gamma)\sqrt{t}\right] \leq \delta/n.$$

for all $i \in \{1, 2, \ldots, n\}$. Then,

$$\begin{aligned}\Pr\left[\ \|\mathbf{e}^{1:t}\|_\infty \geq J(\delta, \gamma)\sqrt{t}\right] &\leq \sum_{i=1}^{n} \Pr\left[|e_i^{1:t}| \geq J(\delta, \gamma)\sqrt{t}\right] \\ &\leq \delta\end{aligned}$$

by the union bound. Now, notice that we can relate $\hat{\boldsymbol{\ell}}^{1:t}$ and $\hat{\mathbf{c}}^{1:t}$ by

$$(B^\dagger)^{-1}\hat{\boldsymbol{\ell}}^{1:t} = (B^\dagger)^{-1}\sum_{\tau=1}^{t}\boldsymbol{\ell}^\tau = \sum_{\tau=1}^{t}(B^\dagger)^{-1}\boldsymbol{\ell}^\tau = \sum_{\tau=1}^{t}\hat{\mathbf{c}}^\tau = \hat{\mathbf{c}}^{1:t}.$$

and similarly for $\boldsymbol{\ell}^{1:t}$ and $\mathbf{c}^{1:t}$. Then

$$\begin{aligned}
\Pr\left[\|\hat{\mathbf{c}}^{1:t} - \mathbf{c}^{1:t}\|_\infty \geq \beta_\infty J(\delta,\gamma)\sqrt{t}\right] &= \Pr\left[\|(B^\dagger)^{-1}(\hat{\boldsymbol{\ell}}^{1:t} - \boldsymbol{\ell}^{1:t})\|_\infty \geq \beta_\infty J(\delta,\gamma)\sqrt{t}\right] \\
&\leq \Pr\left[\beta_\infty \|\mathbf{e}^{1:t}\|_\infty \geq \beta_\infty J(\delta,\gamma)\sqrt{t}\right] \\
&= \Pr\left[\|\mathbf{e}^{1:t}\|_\infty \geq J(\delta,\gamma)\sqrt{t}\right] \\
&\leq \delta.
\end{aligned}$$

□

We can now prove our main result for the section, a statement of the form of Equation (6.4) relating OPT and $\widehat{\mathrm{OPT}}$:

Theorem 6.4.3. *If we play $\mathcal{V}$ against BGA for T timesteps,*

$$E[\widehat{\mathrm{OPT}}] \leq E[\mathrm{OPT}] + (1-\delta)\left(\frac{3}{2}D\beta_\infty J(\delta,\gamma)\sqrt{T}\right) + \delta MT.$$

Proof. Let $\Phi = \hat{\mathbf{c}}^{1:T} - \mathbf{c}^{1:T}$. By definition of $\mathcal{R}$, $\mathcal{R}(\hat{\mathbf{c}}^{1:T}) \cdot \hat{\mathbf{c}}^{1:T} \leq \mathcal{R}(\mathbf{c}^{1:T}) \cdot \hat{\mathbf{c}}^{1:T}$ or equivalently $\mathcal{R}(\mathbf{c}^{1:T} + \Phi) \cdot (\mathbf{c}^{1:T} + \Phi) \leq \mathcal{R}(\mathbf{c}^{1:T}) \cdot (\mathbf{c}^{1:T} + \Phi)$, and so by expanding and rearranging we have

$$\begin{aligned}
\mathcal{R}(\mathbf{c}^{1:T} + \Phi) \cdot \mathbf{c}^{1:T} - \mathcal{R}(\mathbf{c}^{1:T}) \cdot \mathbf{c}^{1:T} &\leq (\mathcal{R}(\mathbf{c}^{1:T}) - \mathcal{R}(\mathbf{c}^{1:T} + \Phi)) \cdot \Phi \\
&\leq D\|\Phi\|_\infty. \qquad (6.5)
\end{aligned}$$

Then,

$$\begin{aligned}
|\,\mathrm{OPT} - \widehat{\mathrm{OPT}}| &= |\mathcal{R}(\mathbf{c}^{1:T}) \cdot \mathbf{c}^{1:T} - \mathcal{R}(\mathbf{c}^{1:T} + \Phi) \cdot (\mathbf{c}^{1:T} + \Phi)| \\
&\leq |(\mathcal{R}(\mathbf{c}^{1:T}) - \mathcal{R}(\mathbf{c}^{1:T} + \Phi)) \cdot \mathbf{c}^{1:T}| + |\mathcal{R}(\mathbf{c}^{1:T} + \Phi) \cdot \Phi| \\
&\leq (D + D/2)\|\Phi\|_\infty,
\end{aligned}$$

where we have used Equation (6.5). Recall from Section (6.2), we assume $\|\mathbf{x}\|_1 \leq D/2$ for all $\mathbf{x} \in S$, so $\|\mathbf{x} - \mathbf{y}\|_1 \leq D$ for all $\mathbf{x}, \mathbf{y} \in S$. The theorem follows by applying the bound on Φ given by Corollary (6.4.2), and then observing that the above relationship holds for at least a $1 - \delta$ fraction of the possible algorithm histories. For the other δ fraction, the difference might be as much as δMT. Writing the overall expectation as the sum of two expectations conditioned on whether or not the bound holds gives the result. □

6.4.3 Relating the Loss of BGA and its GEX Subroutine

Now we prove a statement like Equation (6.3), relating loss(BGA) to loss(GEX).

Theorem 6.4.4. *If we run BGA with parameter γ against $\mathcal{V}$ for T timesteps,*

$$E[\textit{loss}(\textit{BGA})] \leq (1-\gamma)E[\textit{loss}(\textit{GEX})] + \gamma MT.$$

Proof. For a given adversary $\mathcal{V}$, G^{t-1} fully determines the sequence of cost vectors given to algorithm GEX. So, we can view GEX as a function from G^{t-1} to probability distributions over S. If we present a cost vector $\hat{\mathbf{c}}$ to GEX, then the expected cost to GEX given history G^{t-1} is $\sum_{\tilde{\mathbf{x}}\in S} \Pr(\tilde{\mathbf{x}} \mid G^{t-1})\ (\hat{\mathbf{c}} \cdot \tilde{\mathbf{x}})$. If we define $\bar{\mathbf{x}}^t = \sum_{\tilde{\mathbf{x}}\in S} \Pr(\tilde{\mathbf{x}} \mid G^{t-1})\, \tilde{\mathbf{x}}$, we can re-write the expected loss of GEX against $\hat{\mathbf{c}}$ as $\hat{\mathbf{c}} \cdot \bar{\mathbf{x}}^t$; that is, we can view GEX as incurring the cost of some convex combination of the possible decisions in expectation. Let $\hat{\boldsymbol{\ell}}^{t,j}$ be $\hat{\boldsymbol{\ell}}^t$ given that we explore by playing basis vector $\mathbf{b}_j$ on time t, and similarly let $\hat{\mathbf{c}}^{t,j} = (B^\dagger)^{-1}\hat{\boldsymbol{\ell}}^{t,j}$. Observe that $\hat{\ell}_i^{t,j} = \frac{n}{\gamma}\ell_i^t$ for $j = i$ and 0 otherwise, and so

$$\sum_{j=1}^{n} \hat{\boldsymbol{\ell}}^{t,j} = \frac{n}{\gamma}\boldsymbol{\ell}^t = \frac{n}{\gamma}B^\dagger \mathbf{c}^t. \tag{6.6}$$

Now, we can write

$$\begin{aligned}
E[\hat{z}^t \mid G^{t-1}] &= (1-\gamma)\,0 + \gamma \sum_{j=1}^{n} \frac{1}{n} \sum_{\tilde{\mathbf{x}}^t \in S} \Pr(\tilde{\mathbf{x}}^t \mid G^{t-1})\ (\hat{\mathbf{c}}^{t,j} \cdot \tilde{\mathbf{x}}^t) \\
&= \gamma \left[\sum_{j=1}^{n} \frac{1}{n}\hat{\mathbf{c}}^{t,j}\right] \cdot \bar{\mathbf{x}}^t \\
&= \frac{\gamma}{n}(B^\dagger)^{-1} \left[\sum_{j=1}^{n} \hat{\boldsymbol{\ell}}^{t,j}\right] \cdot \bar{\mathbf{x}}^t, \quad \text{and using Equation (6.6),} \\
&= \mathbf{c}^t \cdot \bar{\mathbf{x}}^t.
\end{aligned}$$

Now, we consider the conditional expectation of z^t and see that

$$\begin{aligned}
E[z^t \mid G^{t-1}] &= (1-\gamma)(\mathbf{c}^t \cdot \bar{\mathbf{x}}^t) + \gamma \sum_{i=1}^{n} \frac{1}{n}(\mathbf{c}^t \cdot \mathbf{b}_i) \\
&\leq (1-\gamma)E[\hat{z}^t \mid G^{t-1}] + \gamma M,
\end{aligned} \tag{6.7}$$

Then we have,

$$\begin{aligned} E[z^t] &= E\left[E[z^t \mid G^{t-1}]\right] \\ &\leq E\left[(1-\gamma)E[\hat{z}^t \mid G^{t-1}] + \gamma M\right] \\ &= (1-\gamma)E\left[E[\hat{z}^t \mid G^{t-1}]\right] + \gamma M \\ &= (1-\gamma)E[\hat{z}^t] + \gamma M, \end{aligned} \tag{6.8}$$

by using the inequality from Equation (6.7). The theorem follows by summing the inequality (6.8) over t from 1 to T and applying linearity of expectation. □

6.4.4 A Bound on the Expected Regret of BGA

Theorem 6.4.5. *If we run BGA with parameter γ using subroutine GEX with parameter ϵ (as defined in Appendix C), then for all $\delta \in (0, 1]$,*

$$E[\textit{loss}(\textit{BGA})] \leq E[\mathrm{OPT}] + \mathcal{O}\left(D\frac{1}{\gamma}nM\sqrt{2\ln(2n/\delta)}\sqrt{T} + \delta MT + \frac{\epsilon}{\gamma^2}n^3M^2T + \frac{n}{\epsilon} + \gamma MT\right)$$

Proof. In Appendix C, we show an algorithm to plug in for GEX, based on the FPL algorithm of Kalai and Vempala [2003] and give bounds on regret against a deterministic adaptive adversary. We first show how to apply that analysis to GEX running as a subroutine to BGA.

First, we need to bound $|\hat{\mathbf{c}}^t \cdot \mathbf{x}|$. By definition, for any $\mathbf{x} \in S$, we can write $\mathbf{x} = B\mathbf{w}$ for weights $\mathbf{w}$ with $\mathbf{w}_i \in [-1, 1]$ (or $[-2, 2]$ if it is an approximate barycentric spanner). Note that $\|\hat{\boldsymbol{\ell}}^t\|_1 \leq (\frac{n}{\gamma})M$, and for any $\mathbf{x} \in S$, we can write $\mathbf{x}$ as $B\mathbf{w}$ where $\mathbf{w}_i \in [-2, 2]$. Thus,

$$|\hat{\mathbf{c}}^t \cdot \mathbf{x}| = |(B^\dagger)^{-1}\hat{\boldsymbol{\ell}}^t \cdot B\mathbf{w}| = |(\hat{\boldsymbol{\ell}}^t)^\dagger B^{-1}B\mathbf{w}| = |\hat{\boldsymbol{\ell}}^t \cdot \mathbf{w}| \leq \|\hat{\boldsymbol{\ell}}^t\|_1 \, \|\mathbf{w}\|_\infty \leq \frac{2nM}{\gamma}.$$

Let $R = 2nM/\gamma$. Suppose at the beginning of time we fix the random decisions of BGA that are not made by GEX, that is, we fix a sequence $X = [\chi^1, \mathbf{b}^1, \ldots, \chi^T, \mathbf{b}^T]$. Fixing this randomness together with $\mathcal{V}$ determines a new deterministic adaptive adversary $\tilde{\mathcal{V}}$ that GEX is effectively playing against. To see this, let $\tilde{h}^{t-1} = [\tilde{\mathbf{x}}^1, \ldots, \tilde{\mathbf{x}}^{t-1}]$. If we combine $\tilde{h}^{t-1}$ with the information in X, it fully determines a partial history G^{t-1}. If we let $h^{t-1} = [\mathbf{x}^1, \ldots, \mathbf{x}^{t-1}]$ be the partial decision history that can be recovered from G^{t-1},

then $\hat{\mathcal{V}}(\tilde{h}^{t-1}) = \chi^t \frac{d}{\gamma} \mathcal{V}(h^{t-1})$. Thus, when GEX is run as a subroutine of BGA, we can apply Lemma (C.0.4) from the Appendix and conclude

$$E[\text{loss}(\text{GEX}) \mid X] \leq E[\widehat{\text{OPT}} \mid X] + \epsilon(4n+2)R^2T + \frac{4n}{\epsilon} \tag{6.9}$$

For the remainder of this proof, we use big-Oh notation to simplify the presentation. Now, taking the expectation of both sides of Equation (6.9),

$$E[\text{loss}(\text{GEX})] \leq E[\widehat{\text{OPT}}] + \mathcal{O}\left(\epsilon n R^2 T + \frac{n}{\epsilon}\right)$$

Applying Theorem (6.4.4),

$$E[\text{loss}(\text{BGA})] \leq (1-\gamma)E[\widehat{\text{OPT}}] + \mathcal{O}\left(\epsilon n R^2 T + \frac{n}{\epsilon} + \gamma MT\right)$$

and then using Theorem (6.4.3) we have

$$\begin{aligned}
&E[\text{loss}(\text{BGA})] \\
&\quad \leq (1-\gamma)E[\text{OPT}] + \mathcal{O}\left(J(\delta,\gamma)D\sqrt{T} + \delta MT + \epsilon n R^2 T + \frac{n}{\epsilon} + \gamma MT\right) \\
&\quad \leq E[\text{OPT}] + \mathcal{O}\left(D\frac{1}{\gamma} nM\sqrt{2\ln(2n/\delta)}\sqrt{T} + \delta MT + \frac{\epsilon}{\gamma^2} n^3 M^2 T + \frac{n}{\epsilon} + \gamma MT\right)
\end{aligned}$$

For the last line, note that while $E[OPT]$ could be negative, it is still bounded by MT, and so this just adds another γMT term, which is captured in the big-Oh term. □

Ignoring the dependence on n, M, and D and simplifying, we see BGA's expected regret is bounded by

$$E[\text{regret}(\text{BGA})] = \mathcal{O}\left(\frac{\sqrt{T}\sqrt{\ln(1/\delta)}}{\gamma} + \delta T + \frac{\epsilon T}{\gamma^2} + \frac{1}{\epsilon} + \gamma T\right).$$

Setting $\gamma = \delta = T^{-1/4}$ and $\epsilon = T^{-3/4}$, we get a bound on our loss of order $\mathcal{O}(T^{3/4}\sqrt{\ln T})$.

6.5 Conclusions and Later Work

We have presented a general algorithm for online optimization over an arbitrary set of decisions $S \subseteq \mathbb{R}^n$, and proved regret bounds for our algorithm that hold against an adaptive

adversary. A number of questions are raised by this work. In the "flat" bandits problem, bounds of the form $\mathcal{O}(\sqrt{T})$ are possible against an adaptive adversary [Auer et al., 2002]. Against a oblivious adversary in the geometric case, a bound of $\mathcal{O}(T^{2/3})$ is achieved by Awerbuch and Kleinberg [2004]. We achieve a bound of $\mathcal{O}(T^{3/4}\sqrt{\ln T})$ for this problem against an adaptive adversary. Auer et al. [2002] give lower bounds showing that the $\mathcal{O}(\sqrt{T})$ result is tight, but no such bounds are known for the geometric decision-space problem. Can our bounds be improved, and what is the corresponding lower bound for the problem?

After the publication of the work described here, these questions were answered by Dani and Hayes [2006]. They show that a tighter analysis of the algorithm presented here in fact has a bound of $\mathcal{O}(T^{2/3})$ on regret, and they show a corresponding lower bound.

A related issue is the use of information received by the algorithm; our algorithm and the algorithm of Awerbuch and Kleinberg [2004] only use a γ fraction of the feedback they receive, which is intuitively unappealing. Further, the lower bound of Dani and Hayes [2006] depends on the assumption that the information from the $1 - \gamma$ fraction exploit rounds is discarded. This leaves open the possibility that an algorithm that uses all of the feedback can possibly achieve lower regret.

Chapter 7

Conclusions

7.1 Summary of Contributions

This thesis makes the following principal contributions; taken together, they provide a powerful set of modeling and algorithmic tools for creating robust plans of action for uncertain environments.

- **Fast algorithms for MDP planning:** The improved prioritized sweeping (IPS) algorithm generalizes Dijkstra's algorithm, and is fast on problems that are "almost" deterministic. The prioritized policy iteration algorithm combines the intuition of IPS with fast policy evaluation using linear solvers, yielding good all-around performance; it is especially effective on problems with a great deal of cycling or on problems that are "almost" policy evaluation problems. For problems with a fixed start state, the bounded real-time dynamic programming (BRTDP) algorithm improves over RTDP by providing stationary policies with provable performance guarantees. BRTDP also offers better convergence properties than many other algorithms for this problem such as HDP and LRTDP.

- **The MDP with adversarial costs formulation:** We introduced a generalization of standard MDP planning that considers a set of potential cost vectors, from which an adversary selects one, rather than a fixed known cost vector. This formulation can be used to model a variety of interesting problems, including a sensor-placement / observation-avoidance game.

- **New uses for the convex game framework:** We show that optimal oblivious routing as well as the above adversarial MDP problem can be modeled as bilinear-payoff

zero-sum convex games, and show how convex games can be used to extend the stochastic game framework to handle periods of partial observability. Combined with the known result that zero-sum extensive-form games can be represented as convex games, these results establish convex games as a useful modeling framework, and highlight the importance of finding fast algorithms for problems in this class.

- **Fast algorithms for convex games:** We introduced the single oracle algorithm and two versions of the double oracle algorithm, and experimentally demonstrated their effectiveness on a variety of convex games. These algorithms dramatically outperform approaches based on directly solving the linear programs for the games. Fictitious play, a very simple oracle-based algorithm, had remarkably good performance on one of the problems, Rhode Island hold'em.

- **A limited-observation geometric no-regret algorithm:** The bandit-style geometric decision algorithm (BGA) provides no-regret guarantees given complex structured action sets and a limited total-cost observation model, even when facing an adaptive adversary. This algorithm can be used to guarantee good performance when playing a repeated convex game.

7.2 Summary of Open Questions and Future Work

In the preceding chapters we have highlight a variety of promising extensions to the work presented here. In this section we summarize some of those possibilities for future work, and also state several open questions and general themes.

- **Convex games:** Chapter 3 presented a variety of examples of convex games, and Chapters 5 and 6 presented practical, efficient algorithms for planning in convex games in the off-line and on-line settings, respectively. We feel that there are potentially many other interesting problems that can be cast as convex games, yielding immediate algorithmic and theoretical results.

- **Extension to NP-hard response problems:** For the convex games we have considered, fast exact best-response oracles were available. However, the double oracle algorithm approach holds great potential for solving problems when even the best-response problem is NP-hard. For example, we might consider delivery problem games where an approximation algorithm for the traveling salesman problem is used as a best-response oracle.

- **Improving the DOBA+ algorithm:** Significant improvements to the DOBA+ algorithm may be possible, while still using the general double oracle game-approximation technique. Future work includes developing efficient techniques for finding "good" best responses (Section 5.4), developing a better understanding of the connections to bundle algorithms for non-smooth optimization (See [Hiriart-Urruty and Lemaréchal, 1993] for example), and proving convergence guarantees (perhaps through better aggregation/discarding strategies) that do not rely on interpolation with fictitious play.

- **Algorithms specialized for EFGs:** In Chapter 5, we focused on developing algorithms applicable to general convex games. However, extensive-form games have significant structure that is perhaps not fully exploited even by considering them as a convex game with very fast best response oracles. For example, rather than a black-box best-response algorithm, the dynamic-programming algorithm for best responses on an EFG allows efficient linear optimization over the full set of best responses, as well as methods for sampling from or even enumerating the set. The tree structure of EFGs also opens up the possibility of decomposition algorithms—we have already done preliminary work on such an algorithm.

- **More efficient EFG representations:** The convex extensive-form game model of Chapter 4 shows that the standard EFG representation can be very inefficient: many EFGs have exponentially more compact representations as CEFGs.

  The GameShrink algorithm of Gilpin and Sandholm [2006b] provides another approach to creating more compact EFG representations: their algorithm can take a special type of EFG game and transform it to a potentially much smaller but strategically equivalent EFG.

  We have already begun a preliminary line of work that shows that the approach of Gilpin and Sandholm [2006b] can be greatly extended. How much more is possible? Clearly arbitrary POSGs cannot be represented as EFGs or CEFGs (barring a complete collapse of the complexity hierarchy), but pushing on the representative power of EFG-like game models seems to be an attractive avenue for scaling up game-theoretic planning approaches.

- **Extensive-form games and cost-paired MDP games:** There appears to be a a close connection between extensive-form games and cost-paired MDP games. An extensive-form game in sequence-weight representation can be specified by two trees (the information set / sequence tree for each player) together with a matrix that provides a linear mapping from strategies in one tree to edge-costs in the other, and vice versa. Each tree has player nodes (corresponding to information sets) and

non-player nodes where the next state is chosen according to the uniform distribution.[1] Under this interpretation, the sequence weights in the extensive-form game become exactly the state-action visitation frequencies in a cost-paired MDP game.

This connections raises several interesting questions. First, the MDPs in cost-paired MDP games can contain cycles, while the information set trees of an EFG are by definition acyclic. In this way, cost-paired MDP games are actually more general than EFGs. What does this generality imply when we interpret a cost-paired MDP game as an EFG?

Second, Even-Dar et al. [2005] present an interesting algorithm for acting in an MDP where costs can change. This is exactly what happens in a cost-paired MDP game (or an EFG under this interpretation) when one player changes their policy. Thus, it should be possible to adapt the theoretical guarantees of Even-Dar et al. [2005] to EFGs where we imagine placing a no-regret (experts) algorithm at each information set for each player. With suitable simulation in this game, it should be possible to prove both players converge to a minimax optimal strategy.

As this summary shows, this thesis raises more questions than it answers. Perhaps the only certainty is that the problem of planning in uncertain environments remains both challenging and important.

[1]This distribution can potentially also directly account for some actions of the random player, making it non-uniform. In the usual sequence-form best response dynamic program, the "junction" nodes are thought of as sum nodes instead of average nodes; however, it is simple to transform the problem between these two interpretations.

Appendix A

The Transition Functions of a CEFG Interpreted as Probabilities

Lemma A.0.1. *For any CEFG G where some $f_p^{ss'}(x_p) \notin [0,1]$, there exists an equivalent CEFG G' where $f_p^{ss'}(x_p) \in [0,1]$.*

Proof. Fix a particular problematic s, s' in G; as these states are fixed, we drop the s, s' superscripts from the f-functions. To prove the theorem, we define an f-equivalent G' with f-functions denote g where $g_p^{ss'} \in [0,1]$ for all p. As G has a finite number of edges, this transformation can be applied repeatedly to prove the lemma. Define

$$f(\bar{x}) = \prod_p f_p(x_p).$$

Each player chooses $x_p \in X_{\phi_p(s)}$ independently. Thus, if for some player p there exist $x, x' \in X_u$ such that $f_p(x) > 0$ and $f_p(x') < 0$, then fixing the other players actions, either x or x' would make $\Pr(s' \mid s, \bar{x}) = \prod_{p \in \mathcal{A}(s)} f_p^{ss'}(x_p)$ negative, violating Equation (4.2).

Thus, $f_p(x_p)$ always has the same sign for all $x_p \in X_u$. Since $\Pr(s' \mid s)$ must be non-negative, there must be an even number (possibly 0) of players where $f_p(x_p) < 0$. We can simply switch the sign on all of these players' f-functions, creating an f-equivalent game where $f_p(x_p) > 0$ for all players.

Suppose for some players and action choices, $f_p(x_p) > 1$. For each player, define $x_p^* = \operatorname{argmax}_{x \in X_u} f_p(x)$, and let $\alpha_p = f_p(x_p^*)$. By assumption, $\beta = \prod_p \alpha_p \leq 1$, as this is the probability of the s to s' transition when each player selects x_p^*. Now, for G' define $g_p = (1/\alpha_p) f_p$. Clearly $g_p(x) \leq 1$, as we are dividing by the maximum value. However,

we are off by the constant β, as for any $\bar{x} \in \bar{X}_s$, $f(\bar{x}) = \beta g(\bar{x})$. We can resolve this easily enough, however, as $\beta \leq 1$. We simply set $g_1 = (\beta/\alpha_1)f_1$ instead of $g_1 = (1/\alpha_1)f_1$ and so $g_1(x_1) \in [0, \beta]$ instead of $g_1(x_1) \in [0, 1]$. We now see that for all $\bar{x}$,

$$\prod_p f_p(x_p) = \prod_p g_p(x_p)$$

and so G' is f-equivalent to G and has f-functions for the $s \to s'$ transition that only take on values in $[0, 1]$. □

Appendix B

The Cone Extension of a Polyhedron

Let $X = \{x \mid Ax = b, x \geq 0\}$ be a polyhedron. Then, the cone extension of X is

$$X^c = \{(\alpha x, \alpha) \mid x \in X, \alpha \geq 0\} \tag{B.1}$$

(B.2)

Thus, $(x^c, \alpha) \in X^c$ if and only if $(1/\alpha)x^c \in X$ and $\alpha \geq 0$. Then, writing (x^c, α) for the (column) vector in $\mathbb{R}^{n+1}$ formed by appending α to the end of x^c, we have

$$\begin{aligned}
(1/\alpha)x^c \in X \text{ and } \alpha \geq 0 \quad &\Leftrightarrow \quad A(1/\alpha)x^c = b \text{ and } (x^c, \alpha) \geq 0 && \text{(B.3)}\\
&\Leftrightarrow \quad Ax^c = \alpha b \text{ and } (x^c, \alpha) \geq 0 && \text{(B.4)}\\
&\Leftrightarrow \quad (A, b)\,(x^c, \alpha) = 0 \text{ and } (x^c, \alpha) \geq 0, && \text{(B.5)}
\end{aligned}$$

where $[A, b]$ is the matrix formed by adding b as a new final column to A. Thus,

$$X^c = \{(x^c, \alpha) \mid (A, b)\,(x^c, \alpha) = 0,\ (x^c, \alpha) \geq 0\}.$$

Appendix C

Specification of a Geometric Experts Algorithm

In this section we point out how the FPL algorithm and analysis of Kalai and Vempala [2003] can be adapted to our setting to use as the GEX subroutine, and prove the corresponding bound needed for Theorem (6.4.5). In particular, we need a bound for an arbitrary $S \subseteq \mathbb{R}^n$ and arbitrary cost vectors, requiring only that on each timestep, $|\mathbf{c} \cdot \mathbf{x}| \leq R$. Further, the bound must hold against an adaptive adversary.

FPL solves the online optimization problem when the entire cost vector $\mathbf{c}^t$ is observed at each timestep. It maintains the sum $\mathbf{c}^{1:t-1}$, and on each timestep plays decision $\mathbf{x}^t = \mathcal{R}(\mathbf{c}^{1:t-1} + \boldsymbol{\mu})$, where $\boldsymbol{\mu}$ is chosen uniformly at random from $[0, 1/\epsilon]^n$, given ϵ, a parameter of the algorithm. The analysis of FPL in Kalai and Vempala [2003] assumes positive cost vectors $\mathbf{c}$ satisfying $\|\mathbf{c}\|_1 \leq A$, and positive decision vectors from $S \subseteq \mathbb{R}^n_+$ with $\|\mathbf{x} - \mathbf{y}\|_1 \leq D$ for all $\mathbf{x}, \mathbf{y} \in S$ and $|\mathbf{c} \cdot \mathbf{x} - \mathbf{c} \cdot \mathbf{y}| \leq R$ for all cost vectors $\mathbf{c}$ and $\mathbf{x}, \mathbf{y} \in S$. Further, the bounds proved are with respect to a fixed series of cost vectors, not an adaptive adversary. We now show how to bridge the gap from these assumptions to our assumptions.

First, we adapt an argument from Awerbuch and Kleinberg [2004], showing that by using our barycentric spanner basis, we can transform our problem into one where the assumptions of FPL are met. We then argue that a corresponding bound holds against an adaptive adversary.

Lemma C.0.2. *Let $S \subseteq \mathbb{R}^n$ be a set of (not necessarily positive) decisions, and $k^t = [\mathbf{c}^1, \ldots, \mathbf{c}^T]$ a set of cost vectors on those decisions, such that $|\mathbf{c}^t \cdot \mathbf{x}| \leq R$ for all $\mathbf{x} \in S$*

and $\mathbf{c}^t \in k^t$. Then, there is an algorithm $\mathcal{A}(\epsilon)$ that achieves

$$E[\text{loss}(\mathcal{A}(\epsilon), k^t)] \leq \text{OPT}(k^t) + \epsilon(4n+2)R^2T + \frac{4n}{\epsilon}$$

Proof. This an adaptation of the arguments of Appendix A of Awerbuch and Kleinberg [2004]. Fix a barycentric spanner $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ for S. Then, for each $\mathbf{x} \in S$, let $\mathbf{x} = B\mathbf{w}$ and define $f(\mathbf{x}) = [-\sum_{i=1}^n \mathbf{w}_i, \mathbf{w}_1, \ldots, \mathbf{w}_n]$. Let $f(S) = S'$. For each cost vector $\mathbf{c}^t$ define $g(\mathbf{c}^t) = [R,\ R + \mathbf{c}^t \cdot bs_1,\ \ldots,\ R + \mathbf{c}^t \cdot \mathbf{b}_n]$. It is straightforward to verify that $\mathbf{c}^t \cdot \mathbf{x} = g(\mathbf{c}^t) \cdot f(\mathbf{x})$, and further $g(\mathbf{c}^t) \geq 0$, $\|g(\mathbf{c}^t)\|_1 \leq (2n+1)R$, and the difference in cost of any two decisions against a fixed $g(\mathbf{c}^t)$ is at most $2R$. By definition of a barycentric spanner, $\mathbf{w}_i \in [-1, 1]$ and so the L_1 diameter of S' is at most $4n$. Note the assumption of positive decision vectors in Theorem 1 of Kalai and Vempala [2003] can easily be lifted by additively shifting the space of decision vectors until it is positive. This changes the loss of the algorithm and of the best decision by the same amount, so additive regret bounds are unchanged. The result of this lemma then follows from the bound of Theorem 1 from Kalai and Vempala [2003]. □

We now need to extend the above bound to adaptive adversaries. The key point here is that the algorithm is *self-oblivious*. A self-oblivious algorithm always plays a decision from some distribution that depends only on the cost history so far and not the outcome of its previous probabilistic choices. Thus, a self-oblivious algorithm can be viewed as a function from cost histories to distributions over decisions. For such algorithms, for any (possibly adaptive) adversary $\mathcal{V}$ there always exists an oblivious adversary that causes at least as much regret. The idea for the proof below is due to Adam Kalai.[1]

Lemma C.0.3. *Fix T, let H^* be the set of decision histories of length 0 to $T-1$, and let K^* be the set of all cost histories of length 0 to $T-1$. Then, fix a decision algorithm $\mathcal{A} : K^* \to \Delta(S)$, where $\Delta(S)$ is the set of probability distributions on the set S of possible decisions. Define*

$$R(\mathcal{A}, \mathcal{V}) = E_{\mathcal{A},\mathcal{V}}\left[\sum_{t=1}^T \mathbf{c}^t\mathbf{x}^t - \min_{\mathbf{x}\in S}\sum_{t=1}^T \mathbf{c}^t\mathbf{x}\right]$$

Let $\mathcal{V}$ be an arbitrary adversary. Then, there exists an oblivious adversary $\mathcal{V}'$ such that

$$R(\mathcal{A}, \mathcal{V}') \geq R(\mathcal{A}, \mathcal{V})$$

[1] We thank Tom Hayes and Varsha Dani for pointing out a bug in the proof we had in the original version of this paper.

Proof. An adversary is t-*oblivious* if its first t costs are chosen obliviously; note all adversaries are 1-oblivious. Let $\mathcal{V}$ be an arbitrary adversary, and suppose it is k-oblivious. If $k = T$, we are done. Otherwise, let $\mathbf{c}_o^1, \ldots, \mathbf{c}_o^k$ be the first k (obliviously chosen) costs selected by $\mathcal{V}$. Expectations are over the random variables $\mathbf{x}^1, \ldots, \mathbf{x}^T$ and $\mathbf{c}^1, \ldots, \mathbf{c}^T$ when $\mathcal{V}$ plays against $\mathcal{A}$, though in this case $\mathbf{c}^1, \ldots, \mathbf{c}^k$ are fully determined. Let $K^T = \mathbf{c}^1, \ldots, \mathbf{c}^T$, the random vector corresponding to the cost history.

Let $g(K^T) = \min_{\mathbf{x} \in S} \sum_{t=1}^{T} \mathbf{c}^t \mathbf{x}$. Using linearity of expectation, we can split the expected regret $R(\mathcal{A}, \mathcal{V})$ into 3 terms:

$$E[\sum_{t=1}^{k} \mathbf{c}_o^t \mathbf{x}^t] + E[\mathbf{c}^{k+1} \mathbf{x}^{k+1}] + E[\sum_{t=k+2}^{T} \mathbf{c}^t \mathbf{x}^t - g(K^T)]$$

Since $\mathcal{A}$ and $\mathbf{c}^1, \ldots, \mathbf{c}^k$ are fixed, $E[\mathbf{x}^{k+1}] = E[\mathcal{A}(\mathbf{c}_o^1, \ldots, \mathbf{c}_o^k)] = \bar{x}$ is also known. Since $\mathcal{V}$ is only k-oblivious, it gets to pick $\mathbf{c}^{k+1}$ with knowledge of $\mathbf{x}^1, \ldots, \mathbf{x}^k$. We have

$$\Pr(\mathbf{c}^{k+1}) = \int_{\mathbf{x}^1, \ldots, \mathbf{x}^k} \Pr(\mathbf{x}^1, \ldots, \mathbf{x}^k) \mathcal{I}[\mathcal{V}(\mathbf{x}^1, \ldots, \mathbf{x}^k) = \mathbf{c}^{k+1}],$$

where $\mathcal{I}$ is an indicator function, returning 1 if $\mathcal{V}(\mathbf{x}^1, \ldots, \mathbf{x}^k) = \mathbf{c}^{k+1}$ and zero otherwise. The probability $\Pr(\mathbf{x}^1, \ldots, \mathbf{x}^k)$ is well defined because $\mathcal{V}$ and $\mathcal{A}$ are fixed. Importantly, note that the distribution over $\mathbf{c}^{k+1}$ is independent of the distribution over $\mathbf{x}^{k+1}$; this follows from the assumption that $\mathcal{A}$ is self-oblivious, that is, it picks its distributions based only on the past cost vectors, not on its own actions. Thus, letting $L^k = E[\sum_{t=1}^{k} \mathbf{c}_o^t \mathbf{x}^t]$ we can write

$$R(\mathcal{A}, \mathcal{V}) = L^k + \bar{x} E[\mathbf{c}^{k+1}] + E[\sum_{t=k+2}^{T} \mathbf{c}^t \mathbf{x}^t - g(K^T)] \tag{C.1}$$

$$= L^k + \int_{\mathbf{c}^{k+1}} \Pr(\mathbf{c}^{k+1}) \left[\mathbf{c}^{k+1} \bar{x} + E[\sum_{t=k+2}^{T} \mathbf{c}^t \mathbf{x}^t - g(K^T) \mid \mathbf{c}^{k+1}] \right] d\mathbf{c}^{k+1} \tag{C.2}$$

$$\leq L^k + \sup_{\mathbf{c}^{k+1}} \left[\mathbf{c}^{k+1} \bar{x} + E[\sum_{t=k+2}^{T} \mathbf{c}^t \mathbf{x}^t - g(K^T) \mid \mathbf{c}^{k+1}], \right] \tag{C.3}$$

where the $\sup$ is over all $\mathbf{c}^{k+1}$ with $\Pr(\mathbf{c}^{k+1}) > 0$. Observe that the quantity inside the supremum is well defined before any costs or decisions are selected, and so $\mathcal{V}$ could do at least as well by selecting $\mathbf{c}^{k+1}$ obliviously to be some c that achieves the supremum. Thus, there is a $(k+1)$-oblivious adversary that causes at least as much regret as $\mathcal{V}$. Extending this result inductively, we conclude there is a fully oblivious (T-oblivious) adversary $\mathcal{V}'$ such that $R(\mathcal{A}, \mathcal{V}') \geq R(\mathcal{A}, \mathcal{V})$. □

Lemma C.0.4. *The regret bound from Lemma C.0.2 applies even if the adversary is adaptive.*

Proof. First, observe that as long as FPL re-randomizes at each timestep, it is self-oblivious, and so Lemma C.0.3 applies. Suppose some adaptive adversary $\mathcal{V}$ causes regret that exceeds the bound in Lemma C.0.2. We can apply Lemma C.0.3 to $\mathcal{V}$ and construct an oblivious $\mathcal{V}'$ that also exceeds the bound, a contradiction. □

Thus, we can use $\mathcal{A}(\epsilon)$ as our GEX subroutine for full-observation online geometric optimization.

Appendix D

Notions of Regret

In Auer et al. [1995], an alternative definition of regret is given, namely,

$$E[\text{loss}_{\mathcal{V},\mathcal{A}}(h^T)] - \min_{x \in S} E\left[\sum_{t=1}^{T} \mathbf{c}^t \cdot \mathbf{x}\right]. \tag{D.1}$$

This definition is equivalent to ours in the case of an *oblivious* adversary, but against an adaptive adversary the "best decision" for this definition is not the best decision for a *particular* decision history, but the best decision if the decision must be chosen before a cost history is selected according to the distribution over such histories. In particular,

$$E\left[\min_{x \in S} \sum_{t=1}^{T} \mathbf{c}^t \cdot \mathbf{x}\right] \leq \min_{x \in S} E\left[\sum_{t=1}^{T} \mathbf{c}^t \cdot \mathbf{x}\right]$$

and so a bound on Equation (6.1) is at least as strong as a bound on Equation (D.1). In fact, bounds on Equation (D.1) can be very poor when the adversary is adaptive. There are natural examples where the stronger definition (6.1) gives regret $\mathcal{O}(T)$ while the weaker definition (D.1) indicates no regret. Adapting an example from Auer et al. [1995], let $S = \{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$ (the "flat" bandit setting) and consider the algorithm $\mathcal{A}$ that plays uniformly at random from S. The adversary $\mathcal{V}$ gives $\mathbf{c}^1 = \mathbf{0}$, and if $\mathcal{A}$ then plays $\mathbf{e}_i$ on the first iteration, thereafter the adversary plays the cost vector $\mathbf{c}^t$ where $c_i^t = 0$ and $c_j^t = 1$ for $j \neq i$. The expected loss of $\mathcal{A}$ is $\frac{n-1}{n}T$. For regret as defined by Equation (D.1), $\min_{\mathbf{x} \in S} E[\mathbf{c}^{1:T} \cdot \mathbf{x}] = \frac{n-1}{n}T$, indicating no regret, while $E[\min_{\mathbf{x} \in S}(\mathbf{c}^{1:T} \cdot \mathbf{x})] = 0$, and so the stronger definition indicates $\mathcal{O}(T)$ regret.

Unfortunately, this implies the proof techniques for bounds on expected weak regret like those in Auer et al. [2002] and Awerbuch and Kleinberg [2004] cannot be used to get

bounds on regret as defined by Equation (6.1). The problem is that even if we have unbiased estimates of the costs, these cannot be used to evaluate the term $E[\min_{x \in S} \sum_{t=1}^{T} (\mathbf{c}^t \cdot \mathbf{x})]$ in (6.1) because $\min$ is a non-linear operator. We surmount this problem by proving high-probability bounds on our estimates of $\mathbf{c}^t$, which allows us to use a union bound to evaluate the expectation over the min operator. Note that the high probability bounds proved in Auer et al. [2002] and Awerbuch and Kleinberg [2004] can be seen as corresponding to our definition of expected regret.

Bibliography

After each entry, a list of section numbers indicates where in this thesis the work is cited.

D. Andre, N. Friedman, and R. Parr. Generalized prioritized sweeping. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. MIT Press, 1998. 2.3, 2.3.1

David Applegate and Edith Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: understanding fundamental tradeoffs. In *SIGCOMM '03*, pages 313–324, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-735-4. 3, 3.3, 3.3

Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. Gambling in a rigged casino: the adversarial multi-armed bandit problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331. IEEE Computer Society Press, Los Alamitos, CA, 1995. 6.2, 6.2, D, D

Peter Auer, Nicol Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002. 1, 6.5, D

B. Awerbuch and R. Kleinberg. Adaptive routing with end-to-end feedback: Distributed learning and geometric approaches. In *Proceedings of the 36th ACM Symposium on Theory of Computing*, 2004. 6.1, 6.3, 6.5, C, C, D

Y. Azar, E. Cohen, A. Fiat, H. Kaplan, and H. Räcke. Optimal oblivious routing in polynomial time. In *Proc. of ACM Symposium on the Theory of Computation*, 2003. 3, 3.3, 3.3

Yossi Azar, Edith Cohen, Amos Fiat, Haim Kaplan, and Harald Räcke. Optimal oblivious routing in polynomial time. *J. Comput. Syst. Sci.*, 69(3):383–394, 2004. ISSN 0022-0000. 3.3

James Bagnell, Andrew Y. Ng, and Jeff Schneider. Solving uncertain markov decision problems. Technical Report CMU-RI-TR-01-25, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 2001. 3.4.2

Matthew D. Bailey, Steven M. Shechter, and Andrew J. Schaefer. Spar: stochastic programming with adversarial recourse. *Oper. Res. Lett.*, 34(3):307–315, 2006. 4.4.4

Nikhil Bansal, Avrim Blum, Shuchi Chawla, and Adam Meyerson. Online oblivious routing. In *SPAA*, pages 44–49, 2003. 3.3

Amotz Bar-Noy and Baruch Schieber. The canadian traveller problem. In *SODA '91: Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 261–270, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics. ISBN 0-89791-376-0. 4.4.2

R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994. 2.3.2

Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artif. Intell.*, 72(1-2):81–138, 1995. ISSN 0004-3702. 2.3, 2.3.5, 2.4, 2.4.5

M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows*. John Wiley & sons, 1990. 3.1.1, 5.2, 5.2

Richard E. Bellman. *Dynamic programming*. Princeton University Press, 1957. 1

J F Benders. Partitioning procedures for solving mixed-variable programming problems. *Numerische Mathematik*, 4:238–252, 1962. 5.2

D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, Massachusetts, 1995. 2.2, 2.3

Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996. 2.2, 2.4.1, 2.4.3

Marcin Bienkowski, Miroslaw Korzeniowski, and Harald Räcke. A practical algorithm for constructing oblivious routing schemes. In *SPAA*, pages 24–33, 2003. 3.3

D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *IJCAI*, 2003. 5.1

David Meir Blei and Leslie Pack Kaelbling. Shortest paths in a dynamic uncertain domain. IJCAI Workshop on Adaptive Spatial Representations of Dynamic Environments, 1999. 4.4.2

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997. ISSN 0004-3702. 1, 2.1

Giacomo Bonanno. Memory and perfect recall in extensive games. *Games and Economic Behavior*, 47(2):237–256, 2004. 4

Blai Bonet and Hector Geffner. Faster heuristic search algorithms for planning with uncertainty and full feedback. In G. Gottlob, editor, *Proc. 18th International Joint Conf. on Artificial Intelligence*, pages 1233–1238, Acapulco, Mexico, 2003a. Morgan Kaufmann. 2.3, 2.3.3, 2.3.5, 2.3.5, 2.4, 2.4.5, 15

Blai Bonet and Hector Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. of ICAPS-03*, pages 12–21, 2003b. 2.3, 2.3.3, 2.3.5, 2.4, 2.4.5

Michael Bowling and Manuela M. Veloso. An analysis of stochastic game theory for multiagent reinforcement learning. Technical report CMU-CS-00-165, Computer Science Department, Carnegie Mellon University, 2000. 3.5

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004. ISBN 0521833787. 3, 4.3, 5.2

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990. 2.1, 2.3

Norman Dalkey. Equivalence of information patterns and essentially determinate games. In H. W. Kuhn and A.W. Tucker, editors, *Contributions To The Theory of Games: Volume 2*, number 28 in Annals of Mathematics Studies, pages 217–243. Princeton University Press, 1953. 4.2

Varsha Dani and Thomas P. Hayes. Robbing the bandit: less regret in online geometric optimization against an adaptive adversary. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 937–943, New York, NY, USA, 2006. ACM Press. ISBN 0-89871-605-5. 6.5

T. Dean and S. Lin. Decomposition techniques for planning in stochastic domains. In *IJCAI*, 1995. 2.3.3

Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artif. Intell.*, 76(1-2):35–74, 1995. ISSN 0004-3702. 2.3.2, 2.4, 2.4.5

Kedar Dhamdhere, Vineet Goyal, R. Ravi, and Mohit Singh. How to pay, come what may: Approximation algorithms for demand-robust covering problems. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 367–378, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2468-0. 4.4.4

T. G. Dietterich and N. S. Flann. Explanation-based learning and reinforcement learning: A unified view. In *12th International Conference on Machine Learning (ICML)*, pages 176–184. Morgan Kaufmann, 1995. 2.3.1

M. Dresher and S. Karlin. Solutions of convex games as fixed-points. In H. W. Kuhn and A.W. Tucker, editors, *Contributions To The Theory of Games: Volume 2*, number 28 in Annals of Mathematics Studies, pages 75–86. Princeton University Press, 1953. 1, 3.1

I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Oxford University Press, Oxford, 1986. 2.3.2, 2.3.3

Rosemary Emery-Montemerlo, Geoffrey J. Gordon, Jeff Schneider, and Sebastian Thrun. Approximate solutions for partially observable stochastic games with common payoffs. In *AAMAS*, pages 136–143, 2004. 3.5

Eyal Even-Dar, Sham M. Kakade, and Yishay Mansour. Experts in a markov decision process. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 401–408. MIT Press, Cambridge, MA, 2005. 7.2

David Ferguson and Anthony (Tony) Stentz. Focussed dynamic programming: Extensive comparative results. Technical Report CMU-RI-TR-04-13, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, March 2004. 2.3, 2.4

David Ferguson, Anthony (Tony) Stentz, and Sebastian Thrun. Pao* for planning with hidden state. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, April 2004. 4.4.2

Abraham Flaxman, Adam Tauman Kalai, and H. Brendan McMahan. Online convex optimization in the bandit setting: gradient descent without a gradient. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005. 3.1.2

Yoav Freund and Robert E. Schapire. Game theory, on-line prediction and boosting. In *Computational Learing Theory*, pages 325–332, 1996. 3.1.2

Andrew Gilpin and Tuomas Sandholm. Optimal rhode island hold'em poker. In *AAAI*, pages 1684–1685, 2005. 3, 5.5.2, 5.5.2

Andrew Gilpin and Tuomas Sandholm. A texas hold'em poker player based on automated abstraction and real-time equilibrium computation. In *AAMAS 06*, 2006a. 5.1, 5.5.2

Andrew Gilpin and Tuomas Sandholm. Finding equilibria in large sequential games of imperfect information. In *EC '06: Proceedings of the 7th ACM conference on Electronic commerce*. ACM Press, 2006b. 7.2

Geoffrey J. Gordon. No-regret algorithms for structured prediction problems. Technical Report CMU-CALD-05-112, Carnegie Mellon University, 2005. 5.1

Geoffrey J. Gordon. *Approximate solutions to markov decision processes*. PhD thesis, Carnegie Mellon University, 1999. Chair - Tom Mitchell. 3.1.2

Anshul Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Trans. Math. Softw.*, 28(3):301–324, 2002. ISSN 0098-3500. 2.3.5

Anupam Gupta, Martin Pál, R. Ravi, and Amitabh Sinha. Boosted sampling: approximation algorithms for stochastic optimization. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 417–426, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-852-0. 4.4.4

Eric A. Hansen and Shlomo Zilberstein. LAO*: a heuristic search algorithm that finds solutions with loops. *Artif. Intell.*, 129(1-2):35–62, 2001. ISSN 0004-3702. 2.3.2, 2.3.5, 2.4, 2.4.5

Eric A. Hansen, Daniel S. Bernstein, and Shlomo Zilberstein. Dynamic programming for partially observable stochastic games. In *AAAI*, 2004. 3.5

Bjrn-Ove Heimsund. Matrix toolkits for java (MTJ). `http://www.math.uib.no/~bjornoh/mtj/`, 2004. 2.3.5

J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms*. Springer Verlag, Heidelberg, 1993. Two volumes. 5.2, 5.2, 5.3, 7.2

R. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960. 1

ILOG, Inc. CPLEX (software). http://www.ilog.com/products/cplex/, 2003. 5.5.1

Nicole Immorlica, David Karger, Maria Minkoff, and Vahab S. Mirrokni. On the costs and benefits of procrastination: approximation algorithms for stochastic combinatorial optimization problems. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 691–700, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. ISBN 0-89871-XXX-X. 4.4.4

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. Technical Report CS-96-08, Brown University, 1996. 3.4.2

Adam Kalai and Santosh Vempala. Geometric algorithms for online optimization. Technical report, MIT, 2002. 3.3

Adam Kalai and Santosh Vempala. Efficient algorithms for online optimization. In *COLT*, 2003. 3.1.2, 5.1, 5.5.1, 6.1, 6.2, 6.3, 6.4.1, 6.4.4, C, C

S. Koenig and M. Likhachev. Incremental A*. In *Advances in Neural Information Processing Systems 14*, 2001. 5.5.1

D. Koller and N. Megiddo. The complexity of two-person zero-sum games in extensive form. *Games and Economic Bahavior*, 4(4):528–552, 1992. 3.2, 4, 11

Daphne Koller, Nimrod Megiddo, and Bernhard von Stengel. Fast algorithms for finding randomized strategies in game trees. In *STOC*, 1994. 3, 3.1, 3.1.1, 3.2, 11, 4.3

Andreas Krause. personal communication, 2006. 5

H. W. Kuhn. Extensive games and the problem of information. In H. W. Kuhn and A.W. Tucker, editors, *Contributions To The Theory of Games: Volume 2*, number 28 in Annals of Mathematics Studies, pages 193–216. Princeton University Press, 1953. 3.2

David S. Leslie and E.J. Collins. Generalised weakened fictitious play. *Games and Economic Behavior*, 56(2):285–298, August 2006. 5.1

Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. Planning for markov decision processes with sparse stochasticity. In Lawrence K. Saul, Yair Weiss, and Léon Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 785–792. MIT Press, Cambridge, MA, 2005. 4.4.2

Lucian Vlad Lita, Jamieson Schulte, and Sebastian Thrun. A system for multi-agent coordination in uncertain environments. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 21–22, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-326-X. 4.4.2

Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *ICML*, New Brunswick, NJ, 1994. Morgan Kaufmann. 3.5

H. Brendan McMahan and Avrim Blum. Online geometric optimization in the bandit setting against an adaptive adversary. In *Proceedings of the Seventeenth Annual Conference on Learning Theory (COLT)*, pages 109–123, 2004. 3.1.2, 6

H. Brendan McMahan and Geoffrey J. Gordon. Generalizing Dijkstra's algorithm and Gaussian elimination for solving MDPs. Technical Report CMU-CS-05-127, Carnegie Mellon University, 2005a. 2.1

H. Brendan McMahan and Geoffrey J. Gordon. Planning in cost-paired markov decision process games. NIPS Workshop: Planning for the Real-World, 2003. 3.4

H. Brendan McMahan and Geoffrey J. Gordon. Fast exact planning in markov decision processes. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2005b. 2.1

H. Brendan McMahan, Geoffrey J. Gordon, and Avrim Blum. Planning in the presence of cost functions controlled by an adversary. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML)*, 2003. 3.4, 5.3

H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, 2005. 2.1

Peter Bro Miltersen and Troels Bjerre Sorensen. Computing sequential equilibria for two-player games. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 107–116, New York, NY, USA, 2006. ACM Press. ISBN 0-89871-605-5. 4.4.3

Andrew Moore and Chris Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993. 2.3, 2.3.1

Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, 1995. ISBN 0-521-47465-5. 6.4.2

Abraham Neyman and Sylvain Sorin, editors. *Stochastic Games and Applications*. Springer, 2003. ISBN 1402014937. 3.5

Andrew Y. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Inverted autonomous helicopter flight via reinforcement learning. In *ISER*. Springer, 2004. 2.4

Guillermo Owen. *Game Theory*. Academic Press, third edition, 1995. 3.5

Christos H. Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. *Theor. Comput. Sci.*, 84(1):127–150, 1991. ISSN 0304-3975. 4.4.2

Andrs Perea. *Rationality in Extensive Form Games*. Springer, 2002. Series: Theory and Decision Library C: , Vol. 29. 4.4.3

Michele Piccione and Ariel Rubinstein. On the interpretation of decision problems with imperfect recall. *Games and Economic Behavior*, 20(1):3–24, 1997. 4.1

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, 2nd edition, 1992. 2.3

Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Interscience, 1994. 2.2, 3.4.4

R. Ravi and Amitabh Sinha. Hedging uncertainty: Approximation algorithms for stochastic optimization problems. In *IPCO*, pages 101–115, 2004. 4.4.4

Matthew Rosencrantz, Geoff Gordon, and Sebastian Thrun. Locating moving entities in dynamic indoor environments with teams of mobile robots. In *AAMAS 2003*, 2003. 3.4.1

N. Roy, G. Gordon, and S. Thrun. Finding approximate POMDP solutions through belief compression. *Journal of Artificial Intelligence Research*, 2004. To appear. 2.4

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. ISBN 0137903952. 1, 2.1

R Selten. Reexamination of the perfectness concept for equilibrium points in extensive games. *Journal International Journal of Game Theory*, 4(1), March 1975. 4.4.3

Reinhard Selten. Multistage game models and delay supergames: Nobel lecture, december 9, 1994. In *Game Theory and Economic Behavior: Selected Essays*, volume 2. 1999. 4

J. Shi and M. Littman. Abstraction methods for game theoretic poker. In *Computers and Games*, pages 333–345, 2001. 5.5.2

Trey Smith and Reid Simmons. Heuristic search value iteration for pomdps. In *Proc. of UAI 2004*, Banff, Alberta, 2004. 11

Eiji Takimoto and Manfred K. Warmuth. Path kernels and multiplicative updates. In *Proceedings of the 15th Annual Conference on Computational Learning Theory*, Lecture Notes in Artificial Intelligence. Springer, 2002. 3.4.2, 6.1

Bernhard Von Stengel. Computing equilibria for two-person games. In R.J. Aumann and S. Hart, editors, *Handbook of Game Theory with Economic Applications*, volume 3, chapter 45, pages 1723–1759. Elsevier, September 2002. 3.1.1

Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999. 1, 2.1

Marco Wiering. *Explorations in Efficient Reinforcement Learning*. PhD thesis, University of Amsterdam, 1999. 2.3.1

Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *ICML*, 2003. 3.1.2, 6.3