REASONING AND DECISIONS IN PARTIALLY OBSERVABLE GAMES

BY

MARK D. RICHARDS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Associate Professor Eyal Amir, Chair
Professor Laxmikant Kale
Professor Steven LaValle
Dr. Timothy Hinrichs, University of Illinois at Chicago

ABSTRACT

The extensive form game is a formalism used to model environments where agents make sequences of decisions, possibly in the face of uncertainty about the state of the world and the decisions made by other agents. Such games can be expressed in the form of a tree. Game-theoretic solutions to two-player extensive form games are intractable in many cases. This thesis presents a complete system for modeling and reasoning about such games in a general way under practical computational restrictions.

We address three key challenges related to reasoning about extensive form games: (1) compact representation of the game rules; (2) reasoning about an agent's own knowledge and the knowledge of its opponents; and (3) making decisions in games where it is not possible to generate the full game tree.

The utility of any language designed to express games depends on its expressive power, the ability of game designers to naturally and compactly describe game rules, and the reasoning and inferential tools that the language constructs enable. We present a description language that captures the same semantics as extensive form games but allows their expression in more natural and compact constructs. This logical language is an extension to the Planning Domain Description Language (PDDL), which is commonly used for AI planning problems. We demonstrate the utility of our language by showing that it can be used to express the rules of several popular games and that the language constructs can be used by our game-playing system to make effective decisions.

A second key challenge for our game-playing system is to model the uncertainty related to opponents' decisions and chance events. We outline a decision-making framework based on the notion of *information set generation*. Information sets are similar to *belief states*. Whereas a belief state encodes a probability distribution over possible current worlds, an information set

represents the set of all possible game histories that are consistent with a player's observations. By simulating these possible game histories, an agent can reproduce possible past and future decision points for its opponents, including a re-creation of the knowledge state that the opponent would have in making those decisions. We frame information set generation and sampling as a constraint satisfaction problem and show significant gains in efficiency over the existing method on several games. We also show that this operation scales efficiently to massively parallel machines. We achieve speedups of over 500 times in some cases, surpassing current limits of parallelism for fully observable games.

The third challenge that we address is making decisions in an environment where observation, deliberation, and action are interleaved. Nash equilibrium solutions are a function of the entire game tree and are predicated on the assumption that agents are computationally unbounded. In practice, agents must make decisions about what to do at the present moment, given a sequence of observations received so far and only limited exploration of the game tree. We present three different decision-making algorithms and show that their optimality depends on whether the game has a dominant, pure, or mixed strategy equilibrium. We describe the circumstances under which an agent can be exploited by an omniscient opponent when using one of these strategies. As a case study, we implement our opponent modeling strategy for the popular board game Scrabble. Our reasoning agent outperforms the *de facto* world champion system.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

One of the key goals of artificial intelligence research is to develop human-level (or better) decision-making agents. Real-world decision problems are often characterized by and complicated by several or all of the following: (1) a large number of alternatives to consider; (2) uncertainty due to noisy sensors and actuators; (3) uncertainty due to random acts of nature; (4) incomplete knowledge about cause-effect relationships in the environment, especially when those relationships change over time; (5) uncertainty about the knowledge, motives, capabilities, and actions of other agents, whose decisions affect the utility of outcomes; (6) the need to make a *sequence* of decisions, interleaved with acquisition of additional information (e.g. sensor data).

Researchers in mathematics, computer science, economics, and other fields have developed formalisms for dealing with subsets of these challenges. Examples include Markov Decision Processes and their partially observable counterparts, decision trees, and game trees. Each formalism addresses some subset of the challenges just mentioned. They are useful both because of their direct utility when applied to decision environments that exhibit only those features which they support, and also because they provide valuable insight into more complicated problems to which they cannot be directly applied.

1.1 Extensive Form Games

The focus of this thesis is the *extensive form game* (EFG) [Kuhn, 1953]. Many real-world decision problems involving multiple competing agents can be framed as games, including securities trading, business acquisitions and mergers, and military operations [Papayoanou, 2010]. Typically depicted in the form of a tree, EFGs can be used to reason about environments which involve

multiple agents who make finite sequences of decisions in the presence of imperfect information—uncertainty about the state of the world and the decisions made by other agents.

In this thesis, we assume that all agents know the rules of the game as well as the utility to each agent of each possible outcome. We assume that the game tree is finite. We further assume *common knowledge* about these things: agents know these attributes, they know that the other agents know them; they know that the other agents know that the other agents know, and so forth [Fagin *et al.*, 1995]. Non-deterministic events are modeled as actions taken by a special player, CHANCE, whose decisions are characterized by a known (discrete) probability distribution.

One of the attractive features of the EFG formalism is that it elegantly captures the way agents can be exploited because of their own uncertainty and the way they can exploit the uncertainties that the other agents are known to have. Another attractive feature is its intrinsic epistemic system. The structure of the tree implicitly but precisely encodes what agents know about the current world state, what agents know about what other agents know, and so forth.

We focus here on recreational games that are accessible and well-defined, allowing thorough study and evaluation. Examples include card or tile games (e.g., Poker, Scrabble), grid-and-piece games (Stratego, Kriegspiel), and guessing games (Mastermind, Battleship). Rather than focusing on strategy for any one particular game, our goal is to identify more general decision principles that are common to the broader class of partially observable games.

We do not address games involving continuous decision spaces. Environments in which there is uncertainty about the rules of engagement or about the motives or capabilities of other agents are also beyond the scope of this thesis.

We address three key challenges to leveraging the extensive form game formalism: (1) representation, (2) reasoning about imperfect information, and (3) making decisions under practical computational constraints.

1.2 Compact Representation

Popular games like Scrabble and Texas Hold'em Poker can be theoretically formalized as game trees. Unfortunately, these trees are too large to be explicitly generated. Game description languages for fully observable games, such as GDL and GDDL, allow game designers to express the rules of their games compactly using logical constructs [Love *et al.*, 2006; Edelkamp and Kissmann, 2007]. Such languages typically share some syntactic and semantic constructs with first-order logic. Systems built on these languages depend on the full observability of the game in order to determine how the world state should be updated based on agent decisions. Partial observability complicates matters because agents can no longer assume that every proposition of interest can be definitively proven true or false, even in theory. For example, in a poker game, a player might benefit significantly from knowing whether `(holds opponent ace)` is true. Irrespective of the complexity of any theorem-proving mechanism, the rules of the game may prohibit the player from definitively knowing this fact one way or the other, unless a showdown forces the opponent to reveal its hand.

In Chapter 3, we present the Partially Observable Game Domain Definition Language (POGDDL) as a straightforward extension to PDDL, a language commonly used in the AI-planning community. POGDDL supports partial observability by including observation specifications for each agent in the effects declarations of each action. We show that the language is equivalent in expressivity to the EFG formalism. Our language is similar to Thielscher's GDL-II but has faster state transition updates and provides support for numerical fluents [Thielscher, 2010].

POGDDL allows a game designer to express game rules compactly in terms of a collection of logical predicates and numerical fluents (a.k.a. functions). Action schemata describe chance events and moves that players may choose to make. Each action schema defines its precondition, a logical formula that must be satisfied in order for the action to be legal on a given turn. Actions also define their consequences in terms of Add Effects (predicates that become true when the action is executed), Delete Effects (predicates that become false), updates to the values of numerical fluents, and observations provided to each player. Additional constructs define the payoffs to each player when the game ends, depending on the final values of the domain predicates and fluents.

Besides defining the rules of the game, a POGDDL description provides a reasoning agent with the ability to perform operations needed to reason about games in a general way through game tree search. Each game's initial state is explicitly defined. The complete state of the game associated with each node in the game tree comprises the actual truth values of the predicates, the numerical values of the fluents, and the observation list for each player. The list of observations is the only explicit representation of an agent's knowledge. Given a game state, the rules allow an agent to determine what moves are legal from that state. Given a state and an action, an agent can determine the next state (the descendant in the game tree), including the observations that each player would receive[1]. Finally, given a terminal state, the game description allows an agent to detect that the game has ended and determine the payoffs for each player. With these operations, an agent can perform forward search in the game tree. The amount of search that is actually possible depends on an agent's computational budget.

1.3 Information Sets

The second challenge comes from the fact that agents must reason effectively about the possible world and about what other agents know. As noted, the full game state comprises the true state of the world *and* the knowledge states (observation lists) of each agent. A natural approach that has worked well in other domains involving agent uncertainty (e.g., POMDPs) is to estimate a probability distribution over possible worlds–a belief state–and update this distribution as actions are taken and observations received. However, no one has yet demonstrated how to apply this approach to games expressed in a game description language such as GDL-II or GALA [Koller and Pfeffer, 1997]. One key challenge is that it is not possible, in general, to maintain a compact representation of the distribution of possible worlds. Another challenge is that a belief state would only quantify the distribution over values of predicates and fluents; it would not capture or adequately account for the knowledge states of other agents.

[1]Note that for a particular action, the effects are deterministic, including the observations received by each player. To describe probabilistic effects or observations, a game designer must specify a deterministic action followed by a chance event.

In game theory nomenclature, an *information set* is the set of all nodes in an EFG that are indistinguishable to an agent at one of its decision points. For a game described in POGDDL, an information set corresponds to all nodes in the game tree that share the same list of observations for the agent whose turn it is to act. We view information sets not only as nodes in the game tree but as the sequences of actions that lead to those nodes.

Other researchers have noted the need to identify game tree nodes that are consistent with a player's observations [Parker *et al.*, 2005; Russell and Wolfe, 2005]. We refer to this operation as *Information Set Generation*, or ISG. The existing method involves simulating play from the root of the game tree and performing depth-first search, comparing observations in the simulation against observations actually received, and pruning (backtracking) whenever those observations do not match. Unfortunately, this naïve approach does not scale well; the simulation may need to explore a large portion of the game tree.

In Chapter 4, we frame information set generation as a constraint satisfaction problem. We show how our algorithm more effectively prunes large portions of the search space to improve the likelihood of finding nodes that are consistent with a player's observations. We give results for multiple non-trivial games that demonstrate that our algorithm works effectively in many cases where the existing approach does not.

1.4 Point Decisions

The third challenge stems from the fact that game-theoretic concepts such as Nash equilibrium are predicated on an assumption that agents are computationally unbounded. In particular, it is assumed that agents can perform operations on the full game tree in order to find the equilibrium solution. In reality, agents have only limited capabilities to analyze the potential consequences of their decisions and to model how their opponents might respond to their actions. A practical treatment of general extensive form games requires a method for reasoning about game decisions based on only a limited subset of the nodes in the tree. Rather than relying solely on a theoretical strategy that specifies decisions for every contingency in the game, it is important for an agent to

be able to reason about a single decision point at a time, based on whatever observations it has received up to the present time.

In Chapter 5, we present three algorithms that can be used to reason about games one decision point at a time, in environments where deliberation, sensing, and action are interleaved. We classify EFGs according to the complexity of their equilibrium solutions and explain the circumstances under which each of our algorithms is optimal.

We also present an implementation of our opponent modeling strategy for the game of Scrabble. We describe our Scrabble-playing system and its winning performance against the *de facto* world-champion system QUACKLE. We also present the first general game player for partially observable games and give examples of the kind of decisions the system is capable of making in non-trivial games. We discuss the capabilities and practical limitations of our system in terms of reasoning about imperfect information, opponent modeling, and decision making.

1.5 Context and Scope

In Chapter 6, we discuss related work and describe how the present work extends or complements the state of the art in decision making in partially observable games. Finally, in Chapter 7, we summarize our work and discuss promising directions for future research.

In this thesis, we primarily focus on two-player, zero-sum games with elements of chance. We note specifically those cases where the work generalizes beyond those settings. We also assume that there is common knowledge about the rules of the game. We do not address environments where there is uncertainty about the dynamics of the world itself, about opponents' preferences over outcomes, or about the legality of actions from a particular (knowledge) state. We also do not address situations where the decision space is continuous or infinite.

CHAPTER 2

BACKGROUND

In this chapter, we review the foundational concepts in game theory and game tree search that are utilized in this work. We first provide a formal definition of extensive form games. We then review the concept of Nash equilibrium and explain it in the context of game trees with imperfect information. To make these concepts concrete, we provide a detailed example of a simple poker game and describe some of the strategic insights that are needed in order to maximize payoffs. A key focus of this thesis is reasoning about information sets, which stands in contrast to the more commonly used approach of reasoning about belief states. The two terms are sometimes used interchangeably, so we include in this chapter a brief explanation of the distinction between the two. We then review the concept of heuristic evaluation functions which have historically been at the heart of game tree search programs. The recent development of the Monte Carlo Tree Search method, which we describe next, has led to systems that do not require a heuristic evaluation function [Kocsis and Szepesvári, 2006]. We conclude with a discussion of the applicability of game theory to real-world decision-making problems.

2.1 Formal Definition of the Extensive Form Game

An extensive form game is a tree that describes the outcomes that result from sequences of decisions made by one or more agents (players). Each interior node in the tree represents a decision point for one of the players. The outgoing edges from each node represent a finite set of alternatives available to the player at that point, and the children of a node represent the consequences of those choices. Terminal (leaf) nodes correspond to possible outcomes of the interaction. A tuple associated with each terminal node specifies the numeric utility or payoff for each player if that

outcome is achieved.

The random acts of nature, such as card shuffling or dice rolls in a recreational game, can be modeled by a player, CHANCE, whose decisions are made according to a known probability distribution.

Each node for a particular agent encodes that agent's knowledge state implicitly. Imperfect information (partial observability) can be encoded by requiring that the knowledge state at two or more nodes be the same. An *information set* comprises all nodes for a particular player where the knowledge state is the same. A *strategy* is the (possibly probabilistic) decision of each player at each of its decision points.

The following is a notational adaptation of Kuhn's original definition of extensive games [Kuhn, 1953]. An extensive form game Γ is a tuple: $\langle K, (V, E), H, A, U, \sigma_0 \rangle$, where

1. K is an integer, where $K + 1$ is the number of players in the game. We refer to the players as $P_0, P_1, \ldots, P_K$. By convention, P_0 denotes CHANCE or nature.

2. The pair (V, E) is a finite directed tree with nodes V and edges E. The set $T \subset V$ denotes the terminal (leaf) nodes of the tree. $D = V \setminus T$ is the set of decision nodes. For a node x, $N(x)$ denotes its children.

3. $H = \{H_{kj}\}_{1 \leq k \leq K,\ 1 \leq j \leq J_k}$ is a partition of V into the *information sets* of the game. The value J_k is the number of information sets for P_k. For $1 \leq i \leq K$, $H_k = \bigcup_{j \in J_k} H_{kj}$ are the decision points for P_k. For each $h \in H_{kj}$ and for each pair of nodes $x, x' \in h$, $|N(x)| = |N(x')|$.

4. A is the set of actions. Let $E_{kj} = \{(u, v) \in E | u \in H_{kj}\}$, and $E_k = \bigcup_j E_{kj}$. Note that $\bigcup_{k,j} E_{kj} = E$. Partition each E_{kj} into equivalence classes A_{kji} of size $|H_{kj}|$ for $1 \leq k \leq |E_{kj}|/|H_{kj}|$, where $(u, v) \in A_{kji} \wedge (u', v') \in A_{kji} \implies u \neq u'$. That is, each set A_{kji} contains exactly one outgoing edge from each node in H_{kj}. Each A_{kji} is an action, and $A = \bigcup_{k,j,i} A_{kji}$.

5. $U : T \mapsto \mathbb{R}^K$ is the utility function for the outcomes. For $t \in T$, $U_k(t)$ is the payoff to P_k if the game ends at t.

6. $\sigma_0 : H_0 \times E_0 \mapsto [0,1]$ where $\forall j \ \sum_{A_{0ji} \in E_{0j}} \sigma_0(H_{0j}, A_{0ji}) = 1$. This is the probability distribution for decisions at CHANCE nodes.

The definition of actions as *sets* of edges in the game tree is non-intuitive and merits further discussion. An information set H_{kj} is a collection of nodes in the game tree that are indistinguishable to P_k. That is, if the true state of the game is a particular node in H_{kj}, P_k knows only that the true state of the game is one of those nodes in H_{kj} but does not know exactly which one. The outgoing edges from nodes in H_{kj} represent the alternatives available to P_k in that information set.

The actions available at each node in H_{kj} must be identical; otherwise a player would be able to distinguish between the nodes based on the different available alternatives. Thus, an action is not just an edge, but a set of edges, with one edge in the set at each of the nodes in the corresponding information set. The descendants of nodes in H_{kj} represent consequences of those decisions. The consequence of selecting an action obviously depends on which node in H_{kj} is the true state of the game, but the decision itself must be dependent only on H_{kj}. We present a concrete example in Section 2.3 to further clarify this key concept.

2.2 The Nash Equilibrium Solution Concept

A *strategy* for P_k, denoted σ_k, specifies a probability distribution over possible actions for each of P_k's decision points in the game. A *strategy profile* $\sigma = \langle \sigma_0, \sigma_1, \ldots, \sigma_K \rangle$ specifies a strategy for each player and therefore a choice for each decision point in the game:

$$\sigma_k : H_k \times A_k \mapsto [0,1] \ \text{ where } \ \forall j \sum_{A_{kji} \in E_{kj}} \sigma_i(H_{kj}, A_{kji}) = 1$$

In essence, a strategy profile assigns a probability to every edge in the tree and satisfies constraints that require each player to act according to its knowledge (characterized by the information set) rather than the full state of the world (the actual node in the information set). Since each edge in the tree is assigned a probability, a strategy profile induces a probability distribution over the outcomes of the game. The probability of an outcome is the product of the probabilities of each

edge along the path from the root to the terminal. Let $U_k(\Gamma, \sigma)$ be the expected utility for P_k in game Γ under strategy profile σ. We write $U_k(\sigma)$ when the specific game is implied.

For a particular player P_k, we denote by $\sigma_{\text{-}k}$ the strategies of all other players, so that $\sigma = \langle \sigma_k, \sigma_{\text{-}k} \rangle$. We say that σ_k^* is a *best response* to $\sigma_{\text{-}k}$ for P_k iff

$$\forall \sigma_k \;\; U_k(\langle \sigma_k, \sigma_{\text{-}k} \rangle) \leq U_k(\langle \sigma_k^*, \sigma_{\text{-}k} \rangle)$$

A strategy profile σ^* is a *Nash equilibrium* iff σ_k^* is a best response to $\sigma_{\text{-}k}^*$ for $1 \leq k \leq K$. Under a Nash equilibrium, no player can increase its utility by unilaterally changing its own strategy. In a two-player game, if P_k plays σ_k^* that is part of a Nash equilibrium, then P_k is guaranteed a payoff at least equal to $U_k(\sigma^*)$. If there is correct, common knowledge about all players being rational utility maximizers and players are computationally unbounded, then it is arguably reasonable to assume that all players will in fact play a Nash equilibrium strategy.

In this thesis, we address the problem from the perspective of players who are computationally bounded. For large games, it would not be feasible to compute—and therefore not reasonable to play—the Nash equilibrium, nor would it be reasonable to assume that opponents would do so. Nevertheless, these game-theoretic concepts provide an important frame of reference.

2.3 A Simple Poker Example

To make the above definitions more concrete, we present a small example. Figure 2.1 shows a simple poker game for two players. The decision point for CHANCE (or P_0, the dealer) is depicted as a circle. Decisions for P_1 and P_2 are shown as triangles and diamonds, respectively. Both P_1 and P_2 initially hold two dollars and are required to put one dollar in the pot as an ante.

The dealer, holding a King and a Jack ($K > J$), randomly deals one card to P_1 and the other to P_2, with P_1 holding the King in the left subtree and the Jack in the right subtree. P_1 is allowed to look at its card; P_2 is not allowed to look at anyone's card. P_1 and P_2, in turn, then choose to fold (left branch) or to bet their other dollar (right branch).

The payoffs shown at the terminals are for P_1; P_2's payoffs are the negation of these values.

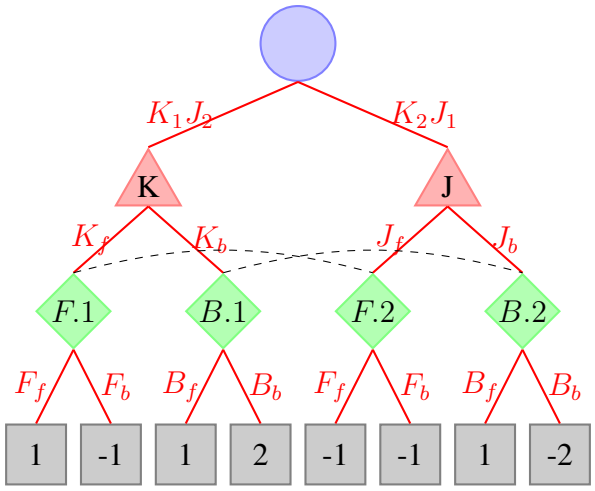

Figure 2.1: A simple poker game. Nodes connected by curved lines are in the same information set. Nodes without curved lines are in singleton information sets.

If one player bets and the other passes, the bettor wins a net payoff of 1 (the other player's ante). Otherwise, in the case of a showdown, the player with the stronger hand wins a net payoff of two dollars if both players bet or one dollar if both folded. The payoffs shown are for P_1. The game is zero-sum, so that $U_2(t) = -U_1(t)$ for all $t \in T$.

If the game had perfect information, the equilibrium strategy for both players would be to bet with the King and fold with the Jack; the expected value of the game would be 0 for both players. But because P_2 is not allowed to look at its card, its information is the same at $F.1$ and $F.2$. Let $H_{21} = \{F.1, F.2\}$. In both cases, P_2 "knows" only that P_1 folded but does not know who has the stronger hand. These two nodes are therefore in the same *information set* (shown with the dotted lines connecting the nodes), and this set captures the fact that P_2 does not know whether CHANCE selected branch K_1J_2 or K_2J_1. Similarly, the information set $H_{22} = \{B.1, B.2\}$ encodes the knowledge that P_1 has bet.

A player must make its decisions based only on the knowledge that it is has at the time the decision is made. Thus, each player's strategy must specify a decision for each of its information sets. Action set A_{21} is the partition of the edges from nodes in H_{21}. So $A_{21} = \{F_f, F_b\}$, where F_f is the set of all edges labeled F_f and F_b is the set of all edges labeled F_b. For example, if P_1 decides to fold with probability p when holding the Jack, then $\sigma_1(J, J_f) = p$ and $\sigma_1(J, J_b) = 1-p$. P_1 must

also choose probabilities for the singleton information set K such that $\sigma_1(K, K_f)+\sigma_1(K, K_b) = 1$. By contrast, the information sets for P_2 are both non-singletons. The information set relationships constrain the strategy probability for the edges labeled B_b to be the same at both $B.1$ and $B.2$ even though the consequences are different (a payoff of -2 vs. 2).

For imperfect information games such as this one, the Nash equilibrium can be found by converting the tree to a linear program, as outlined in [Koller *et al.*, 1994]. In this case, P_1 can exploit the fact that when it bets from J, P_2 will not know whether the true state is $B.1$ or $B.2$. The Nash equilibrium is $\sigma_1(K, K_b) = 1; \sigma_1(J, J_b) = 1/3; \sigma_2(F, F_b) = 1; \sigma_2(B, B_b) = 2/3$. That is, P_1 bluffs $1/3$ of the time on the Jack, which deters P_2 from folding every time P_1 bets. P_1 receives a payoff of 2 with probability $1/3$ *in exchange* for incurring a payoff of -2 with probability $1/9$. Under the equilibrium strategy, the value of the game is a net gain of $1/6$ dollars for P_1. Thus, P_1 is able to exploit P_2's uncertainty to *guarantee* an advantage of at least $1/6$ over the expected value of the perfect information game under optimal play.

In a perfect information game, the equilibrium value of an interior node (the expected value of a game that reaches that node) can be computed based only on its descendants. This is true even in perfect information games that include an element of chance, such as backgammon. But in an imperfect information game, the node values are complicated by the entangling relationships induced by the information sets. Once the strategy profile is fixed, this information can be used much like in the perfect information case to compute the expected value of a game that reaches any particular node. For example, under the equilibrium strategy $V(B.2) = (1/3)(1) + (2/3)(-2) = -1$. Note that this is different than the minimax value (-2) at that node.

2.4 Finding a Nash Equilibrium Through Linear Programming

In this section, we give an overview of how linear programming is used to solve for equilibrium strategies in a key class of two-player, zero-sum, imperfect information games. This formulation is key to understanding the nature of equilibrium strategies in imperfect information games, the limitations of previous work in general game playing, and our results in Chapter 5.

	(F_f, B_f)	(F_f, B_b)	(F_b, B_f)	(F_b, B_b)
(K_f, J_f)	0	0	-1	-1
(K_f, J_b)	1	$-\frac{1}{5}$	0	$-\frac{3}{2}$
(K_b, J_f)	0	$\frac{1}{2}$	0	$\frac{1}{2}$
(K_b, J_b)	1	0	1	0

Figure 2.2: Simple poker game in normal form. Rows (resp. columns) correspond to pure strategies for P_1 (resp. P_2). Matrix entries specify the expected payoff for P_1 under the corresponding strategy profile, assuming both deals are equally likely.

We previously defined EFGs in terms of nodes, edges, information sets, and payoffs. Such games may be expressed equivalently in *matrix form*, also known as *normal form*. In the two-player case, each row in the game matrix A corresponds to a pure strategy for P_1 and each column corresponds to a pure strategy for P_2. Matrix entry $A[i, j]$ gives the payoff for P_1 in expectation (over the chance nodes) when P_1 and P_2 employ strategies i and j respectively. The payoff for P_2 is understood to be the negation of the corresponding entry[1]

The payoff matrix for our simple poker example is shown in Figure 2.2. The pure strategy for P_1 shown in the first row is to always fold with the King and always fold with the Jack. The pure strategy for P_2 shown in the first column is to always fold when P_1 folds and to always fold when P_1 bets. Under these strategies, the result will be a payoff of 1 (for P_1I) when P_1 is dealt the King and -1 when P_1 is dealt the Jack, for an expected value of 0, which is the matrix value shown in the upper-left corner.

A mixed strategy is a linear combination of pure strategies. In order to use the familiar notation of linear programming, we will say that P_1's strategy is a vector x, where element x_i is the probability assigned to pure strategy in the ith row. Likewise, P_2's strategy is a vector y, where element y_j is the probability assigned to the pure strategy in the jth column. Note that $\sum_i x_i = \sum_j y_j = 1$. The expected value of the game under strategy profile $\langle x, y \rangle$ is $x^T A y$. Note that matrix A is dense: all of the entries may be nonzero.

For a fixed strategy x for P_1, the problem of finding a best response y to x can be expressed as

[1]For games that are not zero-sum, there would be a payoff matrix for each player, and the equilibrium would be found using techniques for solving linear complementarity problems [Koller *et al.*, 1994]. For games with more than two players, multi-dimensional grids can be used to define the game, but there is no polynomial-time algorithm to solve for the equilibrium unless $P = NP$ [Blair *et al.*, 1996].

a linear program:

$$\begin{aligned} \underset{y}{\text{maximize}} \;\; & (x^T B)y \\ \text{subjectto} \;\; Fy &= f \\ y &\geq 0 \end{aligned} \tag{2.1}$$

where $B = -A$. Here, F is just a $1 \times n$ matrix of ones and f is the scalar 1, and this single constraint enforces the requirement that a mixed strategy must be a probability distribution over pure strategies. The dual of this linear program has one unconstrained variable for every constraint in the primal. (In this case, there is only one.) The roles of the objective function and the right-hand side of the constraint are reversed:

$$\begin{aligned} \underset{q}{\text{minimize}} \;\; & q^T f \\ \text{subjectto} \;\; q^T F &\geq x^T B \end{aligned} \tag{2.2}$$

By the weak duality theorem, a pair of feasible solutions y, q satisfy

$$q^T f = q^T F y \geq (x^T B)y$$

And by the strong duality theorem,

$$q^T f = q^T F y = (x^T B)y$$

at optimality. A similar primal-dual pair can be defined for the goal of finding a best response x to a given strategy y:

$$\begin{aligned} \underset{x}{\text{maximize}} \;\; & (x^T A)y \\ \text{subjectto} \;\; x^T E^T &= e^T \\ x &\geq 0 \end{aligned} \tag{2.3}$$

$$\begin{aligned} \underset{p}{\text{minimize}} \;\; & e^T p \\ \text{subjectto} \;\; E^T p &\geq Ay \end{aligned} \tag{2.4}$$

To find an equilibrium, it is necessary to find x and y that are best responses to each other. This can be done by solving

$$\begin{aligned} \underset{y,p}{\text{minimize}} \quad & e^T p \\ \text{subjectto} \quad -Ay + E^T p &\geq 0 \\ -Fy &= -f \\ y &\geq 0. \end{aligned} \tag{2.5}$$

If the simplex method (see [Cormen *et al.*, 2009] is used to solve Equation 2.5, the strategies for both P_1 and P_2 can be extracted from the tableau and the value of the objective function gives the equilibrium value of the game: the expected payoff when both players play the equilibrium strategy.

As noted above, matrix A is dense. The number of pure strategies for each player is exponential in the number of nodes of the game tree. The number and size of the constraints in the LP are determined by the number of pure strategies for the players. Therefore, the size of the LP is exponential in the size of the tree.

Koller *et al.*'s key insight is that the important information to capture for a strategy pair is the distribution induced over the terminal nodes [Koller *et al.*, 1994]. This can be expressed more compactly than in the dense normal form matrix explained above by reducing the amount of redundant information that is encoded.

The *support* of a mixed strategy σ_k is the set of pure strategies with non-zero probability in σ_k. Koller *et al.* show that for any mixed strategy, there is an equivalent mixed strategy such that the number of pure strategies in its support is no more than the number of the terminal nodes in the tree. Using this insight, they are able to reformulate the Nash equilibrium problem as a linear program that is linear in the size of the game tree. We refer to this algorithm hereafter as ESLP (Equilibrium through Sparse Linear Programming). We will elaborate on this approach in Chapter 5. To achieve this efficiency, a game must have *perfect recall*, which we define below.

While the ESLP algorithm from Koller *et al.* reduces the time need to find an equilibrium from exponential in the number of nodes in the tree (using the matrix form) to polynomial (using

the sequence form) the algorithm still faces significant practical limitations. As noted, for large games, the number of nodes may be prohibitively large (e.g., more than 10^{100}). And while Koller and Pfeffer have noted that, empirically, solving the LP takes less time than converting the tree to an LP, in practice, it is not possible to traverse the whole tree in order to create the LP. And unfortunately, the algorithm does not have the "anytime" property. It is not clear how to make use of a partially generated LP, since existing entries in the LP may be updated over time as new nodes in the corresponding information sets are discovered. As a result, GALA, the game-playing system built on top of ESLP, has not achieved widespread use [Koller and Pfeffer, 1997].

2.4.1 Perfect Recall

Definition 1. *An n-person game has* perfect recall *iff the following conditions are met:*

1. *For all information sets h, every path in (V, E) includes at most one member of h.*

2. *If v and w are nodes in the same information set and there is a node u that precedes v and $D(u) = D(v)$ (i.e., the same player makes the decision at both nodes) then there must be some node x that is in the same information set as u and precedes w and the decisions for $D(u)$ along the path taken from u to v are the same as from x to w.*

Intuitively, the definition requires that for any two nodes in the same information set, the sequence of choices made by each player in order to arrive at those nodes are the same. This means that at any particular decision point, a player always remembers its own decisions at all previous decision points.

One example of a game that can be modeled as *not* having perfect recall is Bridge. Normally, the game is played between two pairs of two players who each hold their own set of cards. A player does not (at first) know the cards held by her teammate. If this game is modeled as a two-player game, where each player in the model represents a two-handed team, then at each decision point, the model must enforce that the acting player "remembers" the contents of only one of the hands of cards. Many games naturally satisfy the property of perfect recall. Indeed, games that do not satisfy perfect recall seem to be more the exception than the rule.

2.5 Information Sets vs. Belief States

In large games, it would be unreasonable to compute or even express a full strategy for a player. (The total size of a strategy profile is proportional to the number of edges in the game tree.) Instead, decision-making agents are often required to make one decision at a time, with opportunities between "turns" to perform additional limited deliberations based on new observations. In partially observable environments, agents' decisions should depend on the probabilities that they have assigned to possible world states.

The term *belief state* is sometimes used interchangeably with *information set* to refer to a probability distribution over possible worlds. We make a key distinction here. A node in a game tree denotes not only the current state of the game, but also uniquely defines a path—a sequence of actions—from the initial state or root node. Thus, a game tree node implicitly encodes not only the current state of the game but also the complete history of all decisions made by all players (including nature) up to that point in the game. Generating an information set means identifying all game histories that are consistent with a player's observations up to a certain point in the game.

In the example above, it would be dangerous for P_2 to assume when making a decision from information set $\{B.1, B.2\}$ that $B.1$ and $B.2$ are equally likely to be the "real world" just because both of the initial deals are equally probable. In fact, P_1's strategies at both K and J impact the relative likelihood of $B.1$ and $B.2$.

By evaluating possible complete histories, an agent can recreate decision points faced by other players—including the knowledge that other players had when they made their decisions. The dual view of a game tree node as corresponding both to the current state of the world *and* the complete history of the game will be a key theme in this thesis.

2.6 Heuristic Evaluation Functions

We assume that an agent can determine the payoffs for each player when it encounters a terminal node in the game tree. These values can be propagated backwards in the tree to determine the expected value of each interior node. However, since many trees are too large to fully explore,

game-playing agents often seek to approximate the values of non-terminals.

A Heuristic Evaluation Function (HEF) returns an estimate of the minimax value of a node based on extracted features of the game state at that node[2]. For example, the HEF for the DEEP-BLUE chess engine included over 8,000 features that were selected by human grandmasters and tuned through computer analysis of millions of specific positions [Campbell *et al.*, 2002]. These features included material comparisons (e.g., a queen is worth more than a rook) and positional assessments, such as pawn structure, king safety, and control of the center.

Tesauro's temporal difference learning-based backgammon agent employed an evaluation function with over 4,000 features [Tesauro, 1995]. This function was tuned through more than a million games of self-play.

Obviously, if the HEF were completely accurate, there would be no need for search. A program could simply apply the function to the current node's children and choose the move that leads to the best value. In reality, HEFs give only an approximation to the true value of a node. Ostensibly, the quality of these estimates improves as the agent gets closer to the end of the game. Nau has shown that certain properties of the evaluation function can cause MINIMAX to actually perform *worse* as search proceeds deeper into the tree [Nau, 1983]; such situations are beyond the scope of this thesis.

In some domains, many endgame positions are completely solved and therefore do not require an HEF. For example, databases have been created that return the exact equilibrium values for all checkers positions involving fewer than 10 pieces [Schaeffer *et al.*, 2007]. The exact value can also be computed for many endgame chess positions that involve only a few pieces.

It is not always clear what features should be used for an HEF. For example, despite concerted effort by the research community, a good HEF has proven elusive for the game of Go. HEF-based computer Go systems can still be beaten by relatively weak human players.

[2]Some authors use the term *static evaluation function* to mean the same thing.

2.7 Monte Carlo Tree Search

Historically, game-playing programs have used HEFs in conjunction with *alpha-beta* pruning [Knuth and Moore, 1975; Huntbach and Burton, 1988]. The goal of a pruning algorithm is to avoid exploring parts of the game tree that are provably not on the optimal path for either player. In the best case, alpha-beta pruning can compute the true minimax value of the root of an n-node game tree by examining only $\Theta(\sqrt{n})$ nodes.

A new family of algorithms, called *Monte Carlo Tree Search* (MCTS) obviates the need for an HEF by using what is called *simulation* or *rollout*-based techniques. These methods, described in this section, have revolutionized the field of game tree search. New MCTS-based Go agents are now competitive with some of the lower-level professional players [Gelly and Wang, 2006; Lee *et al.*, 2009]. MCTS also plays a central role in the CADIAPLAYER system for general game playing, which won the AAAI General Game Playing competition in 2007 and 2008 [Finnsson, 2007].

A Monte Carlo Tree Search algorithm consists of four main subroutines: selection, expansion, simulation, backpropagation. The individual components may vary somewhat based on the implementation and application. The algorithm generates only a (connected) subset of the nodes in the full game tree, and these nodes compose the *search tree*.

Algorithm 2.1 Generic Monte Carlo Tree Search Algorithm

```
1: function MONTECARLOPLANNING(node,timeLimit)
2:     while currentTime() ≤ timeLimit do
3:         currentNode = node
4:         while currentNode.hasChildren() do
5:             currentNode = SELECTONE(currentNode.children())
6:         simulationNode = EXPAND(currentNode)
7:         simulationResult = SIMULATE(simulationNode)
8:         BACKPROPAGATE(simulationNode,simulationResult)
9:     return bestAction(state,0)
```

Pseudocode for Monte Carlo Tree Search is shown in Algorithm 2.1. The algorithm generates as many game simulations (samples) as time will allow. For each sample, the algorithm traverses the search tree from the root using its selection subroutine. When the selection routine requests an

ungenerated child of an existing node, the expansion routine is called to generate it. Now at the fringe of the search tree, the algorithm performs a *rollout* (described below) to rapidly arrive at a terminal node. The payoff value at that terminal is then propagated back up the search tree from the node on the fringe of the search tree to the root. We next describe each subroutine in more detail.

Selection The (recursive) decision of which node to explore/expand in the search tree can be compared to the widely studied k-armed bandit problem. In the classic formulation, an agent is faced with k slot machines (or k different arms on the same machine) denoted $m_1, \ldots, m_k$, each with its own unknown reward distribution. The agent is allotted n trials and wishes to maximize the expected value of its sampling strategy, $E[V(A)]$, where

$$V(A) = \sum_{i=1}^{n} X_{A(i)}.$$

The $A(i) \in \{1, \ldots, k\}$ subscript is the index of the machine sampled by algorithm A on the ith trial. For $1 \leq j \leq k$, the random variable X_j captures the reward distribution for m_j.

Let μ_j be the expected value of the reward distribution for m_j, μ^* the expected reward from the optimal machine, and $A_j(n)$ be the number of times that m_j is sampled in the first n trials under allocation strategy A. Then the *regret* of strategy A is defined as

$$\mu^* n - \sum_{j=1}^{k} E[A_j(n)].$$

In other words, it is the expected loss incurred by A as a result of the fact that the optimal arm is not selected on every trial. It has been shown that the regret of the optimal allocation strategy is logarithmic in the number of trials [Lai and Robbins, 1985]. Pseudocode for UCB1[3], a simple algorithm that achieve these bounds, is shown in Algorithm 2.2[Auer *et al.*, 2002].

Initially, each arm/machine is sampled once. Thereafter, the decision about which arm to sample is based on an index, computed for each arm, which consists of the sum of two terms. The first term is the mean reward achieved over all prior samples for that particular arm. The second

[3]UCB stands for Upper Confidence Bounds

Algorithm 2.2 Solution to k-armed bandit problem

1: **function** UCB1 (k,N)
2: Sample each arm once
3: **for** $n = k + 1$ to N **do**
4: choose arm j that maximizes $\bar{x}_j + \sqrt{\frac{2\ ln(n)}{n_j}}$

term depends on the number of times the arm has been sampled compared to the total number of samples that have been taken so far. This bonus decreases each time the arm in question is sampled and increases whenever any other arm is sampled. Thus, even when an arm with a high mean reward initially appears poor simply because of "unlucky" early sampling, the second term will eventually become high enough that the arm will be sampled again. The form of this second term is not arbitrary; it is directly related to the Chernoff-Hoeffding bounds, which are used to prove the logarithmic bounds on the regret.

Intuitively, this puts a concrete upper bound on the probability that the sum (mean) of several samples (the rewards from the arms of the bandit) will deviate from the expected sum (mean) by more than a certain amount. Using the bonus term just described, Auer *et al.* showed that the number of times that non-optimal arms will be sampled results in logarithmic regret in the worst case, provided only that the distributions of the rewards for each arm have their support on $[0, 1]$.

In MCTS, this selection rule is applied recursively for each sample. Note that the values used to compute the first term in the indices for each arm (i.e., the $\bar{x}_j$ values) depend on the outcomes of the sampling that takes place lower in the search tree. As a result of this, the distribution of the reward functions for nodes closer to the root is likely to vary significantly between early trials and later trials.

In their seminal work on MCTS, Kocsis *et al.* demonstrate that the variation exhibited satisfies some concretely defined "drift conditions." Using this property, they are able to show that the overall procedure is consistent. In other words, if the search tree is sampled indefinitely in this manner, eventually the estimates of the true expected values of each node will converge to the proper values.

Expansion When the selection algorithm reaches the fringe of the search space, new nodes are added to the search tree to extend the frontier. Coulom proposed an expansion algorithm

whereby only a single new node is added to the search tree on each iteration [Coulom, 2006]. The primary motivation for this conservative approach is memory usage. By adding only a single node each time, the result is that the nodes stored in the tree are primarily those nodes that are visited numerous times throughout the exploration. More aggressive algorithms are certainly possible, but we will assume hereafter that this expansion algorithm is used. Note that if the selection algorithm is able to traverse the search tree all the way to a terminal node, no expansion takes place on that iteration.

Simulation Starting from the node just added to the search frontier, if there is one, the MCTS algorithm then performs a single *rollout*. In its simplest (and computationally least expensive) form, a rollout descends through the tree by choosing actions uniformly at random until it reaches a terminal node. The payoff of that terminal node is returned as the result of the simulation. Note that from any node u in the tree, the expected value of a simulation is the average over all terminals of the subtree rooted at u. The nodes generated during the rollout *are not* stored in memory for future use.

Backpropagation Given a newly expanded node and the result of a single rollout from that node, the backpropagation subroutine of the MCTS algorithm determines how that information will change the estimates of the values of the nodes further up in the search tree. This information, in turn, affects which paths in the tree will be selected for exploration on future iterations. A common method is to estimate the value of a node as the average reward achieved by all trials that have passed through the node.

This seems counterintuitive at first since it does not appear to account for the preferences of the MAX and MIN players. However, when combined with the aforementioned selection criterion, this method does lead to convergence and seems to work well in practice. Over time, the UCB1 selection criterion will cause the best child c of a node u to be selected significantly more often than the other children of u. Thus, the ratio of total rewards to total trials at u will come to be dominated by c, which is the desired outcome.

Again, a key reason for the preference of this backpropagation method is its simplicity. The search tree is completely updated by performing two additions and one division operation (to

update the value estimate) at each node along the path from the simulated node to the root at each iteration. No other nodes in the search tree are affected.

An alternative backpropagation function can be used if the value estimates at each node are defined as the best reward-per-trial value for any of the node's *children*. This method will also converge and is perhaps more intuitive, since this corresponds exactly to the MINIMAX definition of the value of a node.

Before convergence, this second method will tend to overestimate the true value (at MAX nodes) because the early overperformance of any of a node's children due to sampling error will be magnified. The first backpropagation method, on the other hand, will tend to *underestimate* the value (at MAX nodes) before convergence, because the samples of suboptimal children dilute the strength of the parent [Van den Broeck *et al.*, 2009].

The theoretical bounds, simplicity of implementation, and effectiveness in practice have led to widespread usage of the MCTS approach. Although MCTS does not require an HEF, it can take advantage of one. In the same way that an HEF can be used to limit search depth in an alpha-beta or minimax search, an HEF could be used to limit the depth of a rollout simulation in MCTS.

2.8 The Applicability of Game Theory

It is reasonable to challenge the applicability of game-theoretic concepts to real-world decision problems. Blindly playing a Nash equilibrium may be too conservative when there is reason to believe that opponents are weak, as it fails to exploit their faults. Furthermore, the concept of Nash equilibrium is predicated on several key assumptions that may not hold in practice: (1) all agents know the strategies available to each player; (2) all agents know the utility that each agent assigns to each outcome; (3) all agents are rational (expected) utility maximizers; (4) agents are computationally unbounded; (5) there is common knowledge of all of the above.

When navigating traffic, a driver is unlikely to know *a priori* whether her fellow travelers are trying to minimize their travel time, maximize their fuel economy, or simply avoid a collision while talking on a cell phone. There is certainly no common knowledge about strategies or utilities.

When shopping for a home, a buyer may struggle to pin down *his own* valuation of a property, let alone know exact valuations of the seller or competing prospective buyers. These are questions of psychology as much as economics. And such complexities are beyond the scope of this thesis.

Here, we focus on a paradigm where assumptions $1, 2, 3$ and 5 *are* reasonable. Such applications might include participation in a computer network file-sharing scheme or online marketplace for securities, where adherence to published protocols is a prerequisite for participation. The emerging field of *mechanism design* [Nisan *et al.*, 2007] is concerned with the *engineering* of multi-agent interaction environments that have desirable properties: fairness for all players, maximum overall benefit to society achieved when agents play selfishly, etc. An understanding and respect for game theoretic principles are essential in such applications.

Even in less formal interactions where there is unlikely to be common knowledge, game theory can be helpful. Consider a security application. The adversary's capabilities may be unknown, but the defender might model the situation as a game according to its own best understanding of the mechanics. If it can prove that given those rules—and common knowledge among all agents about those rules—that it has an equilibrium strategy with a utility value above some threshold, this can be a useful starting point for charting a course of action. The value of the equilibrium strategy may not be optimal if the opponent turns out to be weaker than expected, but it provides a guaranteed minimum payoff. Having a lower-bound is often of critical importance in security applications. And if there is concern about the accuracy of one's own model of the situation, one might then proceed to test the sensitivity of that same strategy to variations of the rules or opponents, again applying game theory analysis.

Of key concern in this thesis is the fourth assumption about unlimited computational resources. Even when all other assumptions are satisfied, it is still unreasonable to expect that a Nash equilibrium would be played if one or more of the agents does not have the power to compute it. The games we consider in this thesis are sufficiently large that even the best known algorithms are not fast enough to compute the N.E. What then? One of the goals of this thesis is to show how an understanding of the game theoretic principles and the structure of game trees can inform decision-making, in cases where agents have the computational resources to truly consider and analyze only

a subset of the nodes in the tree. Where concrete answers may be elusive, we believe that these theoretical principles at least shed important insights into the nature of the problem.

CHAPTER 3

EXPRESSING GAMES IN POGDDL

In this chapter, we outline the Partially Observable Game Domain Definition Language (POGDDL)[1] The motivation for creating this language is the desire to facilitate the development of general decision-making systems that are able to reason effectively about a game given only a description of the rules. While world-class performance in any particular domain will likely require domain-specific knowledge, there are fundamental principles and building blocks that many decision systems will have in common. Furthermore, having a common language for expressing games allows the comparison of different systems.

The key metrics for evaluating a description language are (1) expressive power, (2) verbosity, (3) ease of reading and writing, and (4) how well it facilitates reasoning algorithms. As we have seen, one way to describe a game tree is by explicitly enumerating the edges and nodes, together with the information set partitioning of the nodes and the action set partitioning of the edges. This is tedious and infeasible for large games. For example, the game tree for Scrabble is estimated to have over 10^{100} nodes. Our language uses more intuitive logic constructs and builds on description languages for other game and planning domains.

Specifically, we have extended the Planning Domain Description Language (PDDL) which has been used extensively in the AI planning community[McDermott *et al.*, 1998; Fox and Long, 2003]. Notably, it has been used for the biennial international planning competitions affiliated with ICAPS. PDDL extends the core syntax and constructs of the STRIPS language: actions are defined using predicate logic constructs that express their preconditions and effects [Lifschitz, 1986]. In particular, the effects descriptions include lists of Add Effects and Delete Effects; these specify

[1] A brief overview of POGDDL has been accepted for publication in AAAI-12. This is joint work with Eyal Amir [Richards and Amir, 2012].

which propositions become true and false, respectively, when the action is executed.

3.0.1 Example

Figure 3.0.1 shows a PDDL encoding of an airport and cargo domain (modified from [Russell and Norvig, 2010]). The `domain` specification (top) defines object types, predicates (with their associated arities and argument types) and action schemata for the general domain. A `problem` (or instance) specification(bottom) defines objects, initial conditions, and goals for a specific task.

The objects and predicates together implicitly define a fixed, finite set of grounded terms or *facts*. In this example, the instance specification defines one plane, two airports, and one cargo object. This implies that there are two grounded instances of the `onground` predicate: `(onground p1 a1)` and `(onground p1 a2)`. Similarly, there is one grounded `in` fact `(in c1 p1)` and two `at` facts: `(at c1 a1)` and `(at c1 a2)`. If the number of grounded facts for a particular problem instance is n, then the size of the state space is 2^n, since each fact can be either true or false.

The `init` declaration specifies the subset of the grounded facts that are true in the initial state. Thereafter, the state can be updated for each action by adding and deleting facts as specified by the action schema. Note that although the syntax of the language requires action effects to be specified as a conjunction of literals, the semantics merely requires that the negative literals in the effects be removed from the state and that the positive literals be added.

The `type`, `object`, and `action` declarations implicitly define a fixed, finite set of grounded actions. In this example, there would be four grounded `fly` actions, two `load` actions, and two `unload` actions. Note that some actions, for example, `(fly p1 a1 a1)`, may never be executable in practice because either the precondition cannot be satisfied or, as is this case here, the Add and Delete Effects conflict. (We cannot both add and delete `(onground p1 a1)`.

As defined, PDDL is appropriate for planning domains where the goal of an automated planner is to find a sequence of actions that leads to a state where the goal conditions are satisfied. In this example, a solution would be:

```
1. (load c1 p1 a1) 2. (fly p1 a1 a2) 3. (unload c1 p1 a2)
```

```
(define (domain airport)
        (:requirements :strips :typing)
        (:types airport cargo plane)
        (:predicates
                (onground ?p - plane ?a - airport)
                (in ?c - cargo ?p - plane)
                (at ?c - cargo ?a - airport))
        (:action fly
                (:parameters ?p - plane ?f ?t - airport)
                (:precondition        (onground ?p ?f))
                (:effect (and
                                (not (onground ?p ?f))
                                (onground ?p ?t))))
        (:action load
                (:parameters ?c - cargo ?p - plane ?a - airport)
                (:precondition (and
                                        (onground ?p ?a)
                                        (at ?c ?a)))
                (:effect        (and
                                        (not (at ?c ?a))
                                        (in ?c ?p))))
        (:action unload
                (:parameters ?c - cargo ?p - plane ?a - airport)
                (:precondition (and
                                        (onground ?p ?a)
                                        (in ?c ?p)))
                (:effect        (and
                                        (not (in ?c ?p))
                                        (at ?c ?a)))))
(define (problem airport1)
        (:domain airport)
        (:objects
                p1 - plane
                a1 a2 - airport
                c1 - cargo)
        (:init
                (onground p1 a1)
                (at c1 a1))
        (:goal (at c1 a2)))
```

Figure 3.1: A PDDL description of an airport domain. Cargo can be moved between airports by loading it onto a plane, flying the plane to another airport, and unloading the cargo. A domain file defines the predicates and the actions of the environment. A problem (a.k.a. instance) file defines specific objects, initial conditions, and goal conditions for a specific task. Applying an action, when its preconditions are satisfied, changes the world by removing some facts from the state and adding others.

Later versions of PDDL support additional features that capture more realistic constraints: plans may be required to satisfy some optimality criterion (e.g., shortest, cheapest); actions may be of non-instantaneous duration; multiple actions may be executed in parallel; action effects may be conditioned on state variables; etc.

Edelkamp and Kissmann have shown how to extend PDDL for games; they call their language the Game Domain Definition Language (GDDL) [Edelkamp and Kissmann, 2007]. GDDL requires the declaration of special objects to define the players and one additional syntactic construct to define the payoffs that players realize at the end of the game:

```
(:gain <typed list(variable)> <f-exp> <GD>)
```

The first parameter specifies a list of players (usually one). The second parameter specifies a numerical reward which those players attain when the goal condition, a logical formula given by the third parameter, is achieved in a terminal state. For example,

```
(:gain p1 100
        (and
                (sunk p2 battleship)
                (sunk p2 carrier)))
```

specifies that player `p1` achieves a payoff of 100 if `p2`'s ships are sunk.

A PDDL planner can be used to identify a viable plan before any actions are executed. The output of the system is a complete sequence of actions which can then be executed to achieve the goal. Alternatively, for large planning problems where it is not reasonable to solve the whole problem in advance, a planning system can be used to identify a single "next action" to take, based on the current state and some limited analysis of which actions are likely to lead toward the goal. In such cases, we say that deliberation and action are interleaved.

In GDDL, a decision-making system would be given the game description and would be assigned one of the pre-defined roles. Other decision-making agents would assume the other roles. Random events can be modeled by an additional system whose actions are taken according to some probability distribution.

The situation is now complicated by the fact that the best actions to take may depend on the choices made by other agents. With this additional complexity, it becomes even less plausible, in

practice, to identify in advance an optimal sequence of decisions. The determination of the optimal probability to assign to every possible action in every possible situation that an agent may find itself in, given some assumptions about the other players' behaviors, is referred to as a *strategy* in the game theory literature. Since many games are too large to find a complete strategy *a priori*, we focus our efforts on decision-making systems that do some limited analysis to choose one action at a time. Given the current state of the environment (or, in the case of partial observability, our sequence of percepts), we seek to identify a single next action that will maximize our eventual payoff. This goal informs the decisions we make about the design and implementation of our game description language.

3.1 Syntax

We now present our Partially Observable Game Domain Definition Language (POGDDL) for describing and reasoning about general games of imperfect information. In order to support partial observability, we extend level 2 of PDDL 2.2 [Edelkamp and Hoffmann, 2004]. Notably, this subset of PDDL offers support for derived predicates and numerical fluents. Again, due to PDDL's modularity, advanced features provided by later versions (timed initial literals, durative actions, spontaneous events, etc.) may potentially be used later as general game playing technology matures. We start with the GDDL `gain` extension just mentioned.

The key syntactic extension that we add to support imperfect information is an `observe` construct that specifies what observations are provided to which players as a result of the execution of an action. These observations may vary by player. For example, the result of an action by P_0 to deal a King to P_1 could be that P_1 receives the observation `(dealt p1 king)` and all other players receive `(dealt p1 _)`.

The changes/additions needed to the formal grammar given in the appendix of [Edelkamp and Hoffmann, 2004] are as follows.

```
<p-effect> ::= <observation>
<observation>  ::=
  (observe (<term> <atomic formula(term)>)*)
```

The first item in each pair of the `observe` list specifies a recipient—the player who will receive the message—and the second item an atomic logical formula that contains the actual observation to be given to that player.

Adding `<observation>` to the list of possible `p-effect` productions allows a game designer to specify observations wherever Add Effects, Delete Effects, and/or updates to numerical fluents are allowed. In particular, observations may be included as conditional effects.

The full grammar for POGDDL is given in Appendix A. In this section we give only a brief summary of the main constructs.

The game description includes lists of

1. object types (e.g., card, tile, square)
2. player (role) names (e.g., p1, p2, chance)
3. predicate names, together with the number and types of their arguments
4. observation names, together with the number and types of their arguments
5. typed (constant) objects (e.g., ace king jack - card)
6. action schemata (parameters, preconditions, effects, observations)
7. gain descriptions (payoffs for each player in terminal states)

As in PDDL, ground predicates are defined over all combinations of declared objects of the appropriate types. Therefore, the `objects` and `predicates` sections together define a fixed, finite set of propositional variables that can have intuitive names, e.g., `(holds p1 queen_hearts)` or `(whose_turn p1)`. We will refer to this ground set of predicates as $\mathcal{P}$. The sets of actions and observations are similarly finite and fixed. A special symbol, '`_`', provides a placeholder in observations that facilitates an intuitive withholding of information, e.g., `dealt(p1 _)`.

An action is defined by (1) a precondition, expressed as a logical formula over $\mathcal{P}$; (2) a list of Add Effects and Delete Effects (disjoint subsets of $\mathcal{P}$); and (3) a list of observations together with the names of the players who should receive them. Observations have the same form as atomic predicates: a name followed by a list of arguments.

3.2 Semantics

A POGDDL definition describes typed objects and the relationships those objects may have with each other. The grounded actions defined by the action schemata describe how those relationships may change and the observations that are given to each player when specific actions are applied. This section explains how these constructs, taken together, can be interpreted as an extensive form game.

A POGDDL game Π defines a tuple $\langle K, \mathcal{P}, \mathcal{O}, \mathcal{A}, S, T, G, H, s_0 \rangle$, where

1. K is an integer, where $K+1$ is the number of players in the game. We refer to the players as $P_0, P_1, \ldots, P_K$. By convention, P_0 denotes nature.

2. $\mathcal{P}$ is a set of propositions. Let $2^{\mathcal{P}}$ denote the power set of $\mathcal{P}$. For $0 \leq k \leq K$, the special proposition w_k indicates whose turn it is to move; w_k is TRUE iff it is P_k's turn to move.

3. $\mathcal{O}$ is a set of observations. Let $o_{1:t}$ denote a sequence of t observations $o_1, o_2, \ldots, o_t$, where $o_i \in \mathcal{O}$ for $1 \leq i \leq t$. Let $\mathcal{O}_+$ denote the set of all observation sequences $o_{1:t}$, $1 \leq t \leq T_{max}$.

4. $\mathcal{A}$ is a set of actions. For each $a = \langle a^*, a^-, a^+ \rangle \in \mathcal{A}$, let precondition a^* be a logical formula over $\mathcal{P}$; $a^+ \subseteq \mathcal{P}$ is the set of Add Effects; and $a^- \subseteq \mathcal{P}$ is the set of Delete Effects.

5. $S \subseteq 2^{\mathcal{P}} \times \mathcal{O}_+^K$ is a set of states, where for $s \in S$, the state $s = \langle \mathsf{P}, o_{1:t}^{(1)}, \ldots, o_{1:t}^{(K)} \rangle$. $\mathsf{P} \in \mathcal{P}$ is the set of propositions that are TRUE in s, and $o_{1:t}^{(i)}$ represents the sequence of observations that P_i has at state s. In practice, t represents the distance of s from the root.

6. $T \subseteq S \times \mathcal{A} \times S$ is the transition relation. For all s^-, a, s^+, $(s^-, a, s^+) \in T$ iff,

   - $s^- = \langle \mathsf{P}^-, o_{1:t}^{(1)}, \ldots, o_{1:t}^{(K)} \rangle \in S$;
   - $a = \langle a^*, a^-, a^+ \rangle \in \mathcal{A}$;
   - $s^+ = \langle \mathsf{P}^+, o_{1:t+1}^{(1)}, \ldots, o_{1:t+1}^{(K)} \rangle \in S$;
   - $s^- \models a^*$;
   - $\mathsf{P}^+ = \mathsf{P}^- \setminus a^- \cup a^+$; and

- For $1 \leq k \leq K$, $o_{1:t+1}^{(k)} = o_{1:t}^{(k)} o_{t+1}^{(k)}$ for some $o_{t+1}^{(k)} \in \mathcal{O}$.

The following conditions must also hold:

- **Unique achievability.** For all s, a, s', a', s'', if $(s, a, s'') \in T$ and (s', a', s'') then $s' = s$ and $a' = a$.
- **Reachability.** If $(s, a, s^+) \in T$ then $s = s_0$ or there exists s' and a' such that $(s', a', s) \in T$.

7. G is a set of *gain structures*. For each state $s \in S$ and each $g = \langle \phi, x, P_k \rangle \in G$, player $g.P_k$ realizes a gain of $g.x$ (a real value) iff $s \models g.\phi$.

8. H is a partition of S into information sets for each player:

   - For all states $s \in S$, there exist i and j such that $s \in H_{ij}$
   - For all i, j and for all $s, s' \in H_{ij}$ such that $s = \langle \mathsf{P}, o_{1:t}^{(1)}, \ldots, o_{1:t}^{(K)} \rangle \in S$ and $s' = \langle \mathsf{P}', o_{1:t}^{(1)'}, \ldots, o_{1:t}^{(K)'} \rangle \in S$, the following must hold:
     - $w_i \in \mathsf{P}$ and $w_i \in \mathsf{P}'$;
     - For $i \neq m$: $w_m \notin P, w_m \notin P'$;
     - $o_{1:t}^{(i)} = o_{1:t}^{(i)'}$.
     - For all $a \in \mathcal{A}$, $s \models a^* \iff s' \models a^*$

9. $s_0 \in S$ is the initial state, which is known.

The semantics of the game description is then as follows:

- The $K+1$ players are named in the constants sections as having the type `role`. The keyword `chance` is a reserved name for P_0.
- The sets of predicates $\mathcal{P}$ and observations $\mathcal{O}$ are defined by the list of constants, predicates, and observation declarations. These in turn define the space for the set of states S.

- The action schemata define the set of actions. The actions may be parameterized to allow for a more compact representation. Each action schema includes a list of parameters, together with a precondition (a Boolean expression over ground terms), Add Effects, Delete Effects, and Observation Effects. These action schemata define the set of actions $\mathcal{A}$.

- The actions and S together define the relation T.

- An information set H_{ij} consists of all the states s where $s \models w_i$ and $o_{1:t}^{(i)}$ match. The number of distinct values of $o_{1:t}^{(i)}$ over all the states in S determines the maximum value of j for each i (i.e., the number of information sets per player).

- The gain statements in the description define G.

3.3 Expressive Power

We now show that POGDDL is equivalent in expressive power to the extensive form game (EFG) formalism.

Theorem 1. *For every valid POGDDL instance, there is a valid extensive form game.*

Proof. Recall the earlier definitions:

- For extensive form games: $G = \langle K, (V, E), H, A, U, \sigma_0 \rangle$

- For POGDDL descriptions: $\Gamma = \langle K, \mathcal{P}, \mathcal{O}, \mathcal{A}, S, T, G, H, s_0 \rangle$

We will use dot notation to refer to constituent parts. So, for example, $\Gamma.S$ refers to the set of states in POGDDL game Γ.

We prove, by construction, that for every POGDDL instance Γ, there is a corresponding extensive form game G_Γ:

- $G_\Gamma.K = \Gamma.K$

- For every $s \in \Gamma.S$, there is a corresponding node (call it v_s) in $G_\Gamma.V$.

- There is an edge from v_s to $v_{s'}$ in $G_\Gamma.E$ iff there exists an $a \in \Gamma.A$ such that $(s, a, s') \in \Gamma.T$.
- For every $s \in \Gamma.s$ where $s \models w_0$, $v_s \in G_\Gamma.H_0$.
- The information sets in $G_\Gamma.H$ are of the same number and size as those in $\Gamma.H$. For all information sets H_{ij} $(i > 0)$ and for every pair $v_s, v_{s'} \in G_\Gamma.V$, let v_s, v'_s be in $G_\Gamma.H_{ij}$ iff $s, s' \in \Gamma.H_{ij}$.
- Let $G_\Gamma.E_{ij}$ denote the set of edges in $G_\Gamma.E$ that originate in $G_\Gamma.H_{ij}$. These edges are partitioned into action sets $G_\Gamma.A_{ijk}$, $(1 \leq k \leq |E_{ij}|/|H_{ij}|)$. Two edges $(v_s, v_{s'}), (v_{s''}, v_{s'''}) \in A_{ijk}$ iff there exists an $a \in \Gamma.A$ such that $(s, a, s') \in \Gamma.T$ and $(s'', a, s''') \in \Gamma.T$.
- For every $s \in \Gamma.S$ where there *does not* exist a, s' with $(s, a, s') \in \Gamma.T$, set $G_\Gamma.U_i(v_s) = \sum_{g \in G^*} g.x$, where $G^* = \{g | g \in \Gamma.G, s \models g.\phi, g.P_k = i\}$.

□

Theorem 2. *For every EFG, there is a corresponding POGDDL description.*

Proof. This proof is also by construction. We can construct a POGDDL description Γ that corresponds to an EFG G as follows:

- $G.K = \Gamma.K$. For every player in G there is a corresponding player in Γ.
- For $v_1, \ldots, v_n \in G.V$, let there be $v_1, \ldots, v_n \in \Gamma.\mathcal{P}$. For each information set $h_{ij} \in G.H$, let there be $h_{ij} \in \Gamma.\mathcal{P}$.
- For each information set $h_{ij} \in G.H$, let there be $h'_{ij} \in \Gamma.\mathcal{O}$.
- Let $h \in \Gamma.\mathcal{P}$ denote the predicate corresponding to the information set that contains the vertex $v \in V$ at the root of the game tree. Let $\Gamma.s_0 = h$.
- For every edge (v_s, v_d) in $G.E$ such that $v_s \in G.H_{ij}$ and $v_d \in G.H_{i'j'}$, define an action $a = \langle a^*, a^-, a^+ \rangle \in \Gamma.\mathcal{A}$ in the following way:
  1. $a^* = h_{ij} \wedge w_i$

2. $a^- = \{h_{ij}, w_i\}$

3. $a^+ = \{h_{i'j'}, w_{i'}, v_d\}$

- For each $v_s \in G.T$, define $\Gamma.g = \langle \phi, x, P_k \rangle$ so that $\phi = v_s$ and $G.U_k(v_s) = x$.
- For $\Gamma.s \in \Gamma.S$, $\Gamma.s.o_{1:t}^{(k)} = \{o_\epsilon, \ldots, o_\epsilon, h'_{ij}, o_\epsilon, \ldots, o_\epsilon, h'_{i'j'}\}$ is an observation sequence of length t such that the h'_{ij} correspond to the time steps where P_k is the active agent, and the o_ϵ null observations occur at all other time steps.
- In the set P at $\Gamma.s$, v_i is true iff the state corresponding to v_i is passed through on the way to $\Gamma.s$.
- $\Gamma.T$ is defined so that $(\Gamma(v_s), a, \Gamma(v_d)) \in \Gamma.T$ iff the corresponding edge $(v_s, v_d) \in G.V$.

□

We note that in our construction of Γ from G, the number of propositions in $\Gamma.\mathcal{P}$ is equal to $|G.V\ G.T| + |G.H|$ and is therefore verbose. In practice, POGDDL descriptions can be useful only if they are sufficiently compact.

3.4 A Simple Poker Example

Figure 3.2 shows the key constructs of a POGDDL encoding of the poker game shown in Figure 3.3. For P_1, the information set (labeled H_{11} in the tree) that corresponds to observation sequence `[(stronger max)]` is `[deal-stronger(max)]`, and for observation sequence `[(stronger min)]`, it is `[deal-stronger(min)]` (which corresponds to H_{12} in the diagram).

For P_2, the information set H_{21} corresponding to sequence `[(stronger unknown), (maxplayed pass)]` is `[ deal-stronger(max), max-choice(pass)]`, `[ deal-stronger(min), max-choice(pass)]` and for observation sequence `[(stronger unknown), (maxplayed bet)]`, the information set (H_{22} in the tree) is `[ deal-stronger(max), max-choice(bet)]`, `[deal-stronger(min), max-choice(bet)]`.

```
(:constants
  max min chance - role
  bet pass - choice
  all unknown - other)

(:init
   (= (potsize) 1)
   (whoseturn chance)
   (gambler max) (gambler min)
   (potadding bet))

(:action deal-stronger
 :parameters (?p - role)
 :precondition (and
  (whoseturn chance)
  (gambler ?p))
 :effect (and
  (not (whoseturn chance))
  (whoseturn max)
  (stronghand ?p)
  (observe (max (stronger ?p))
           (min (stronger unknown)))))

(:action max-choice
 :parameters (?c - choice)
 :precondition (whoseturn max)
 :effect (and
   (not (whoseturn max))
   (whoseturn min)
   (decision max ?c)
   (observe (all (maxplayed ?c)))))
```

```
(:action min-choice
 :parameters (?c - choice)
 :precondition (whoseturn min)
 :effect (and
   (not (whoseturn min))
   (decision min ?c)
   (when (and
     (potadding ?c)
     (decision max bet))
     (assign (potsize) 2))
   (observe (all (minplayed ?c)))))

(:gain ?p (potsize)
  (or
    (and
      (stronghand ?p)
      (or
        (decision ?p bet)
        (exists (?o - role)
          (and
            (oppof ?p ?o)
            (decision ?o pass)))))
    (and
      (decision ?p bet)
      (exists (?o - role)
        (and
          (oppof ?p ?o)
          (decision ?o pass))))))
```

- Information Set H_{11}: O_1 = [(stronger max)]
  - a_1 = [deal-stronger(max)]
- Information Set H_{12}: O_1 = [(stronger min)]
  - a_1 = [deal-stronger(min)]
- Information Set H_{21}: $O_{1:2}$ = [(stronger unknown),(maxplayed pass)]
  - $a_{1:2}$ = [deal-stronger(max), max-choice(pass)]
  - $a_{1:2}$ = [deal-stronger(min), max-choice(pass)]
- Information Set H_{22}: $O_{1:2}$ = [stronger(unknown),(maxplayed bet)]
  - $a_{1:2}$ = [deal-stronger(max), max-choice(bet)]
  - $a_{1:2}$ = [deal-stronger(min), max-choice(bet)]

Figure 3.2: Example actions in a POGDDL description of a simple poker game.

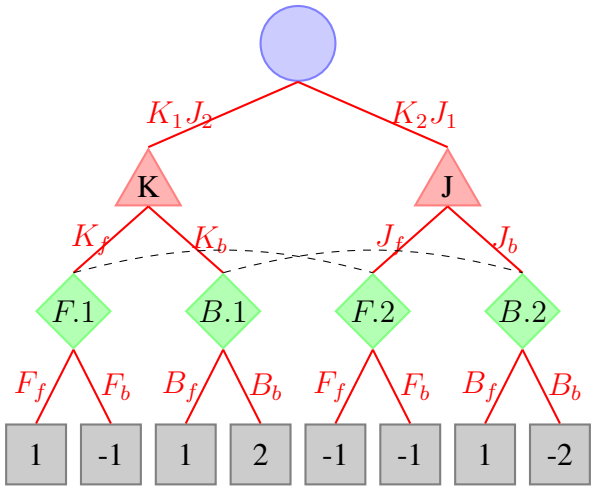

Figure 3.3: A simple poker game. Nodes connected by curved lines are in the same information set. Nodes without curved lines are in singleton information sets.

Given an observation sequence, a player can simulate any action sequence in the corresponding information set—as though he were the moderator—to identify a possible world state. Using this full world state, he can identify what his legal actions are. In order for the game to be well-defined, the same subset of actions must be legal from every node in an information set. This follows directly from the definition of imperfect information games [Kuhn, 1997]. So, for example, if P_2 receives observation sequence `[(stronger unknown), (maxplayed pass)]`, he can simulate the sequence `[deal-stronger(min),max-choice(pass)]` and see that the world state in that case would be s ={`(whoseturn min), (decision max pass), ..., (potsize)=0`}, from which he could determine that his legal actions are `min-choice(pass)` and `min-choice(bet)`. Because of conditional effects, the actual effects of either of those actions may be different in the different possible world states, but the legality of the actions must be the same.

POGDDL descriptions for several non-trivial games are provided in Appendix B.

3.5 Epistemic Reasoning System

We mentioned at the beginning of the chapter that a key aspect in evaluating the utility of a game description language is how effectively it facilitates reasoning algorithms for decision-making systems. If the goal is to find the equilibrium strategy of the game, then any description language that allows an efficient traversal of the game tree will suffice. The GALA system, for example, is embedded in the PROLOG language and converts a game description to a linear program which is then solved using the ESLP algorithm [Koller *et al.*, 1994; Koller and Pfeffer, 1997]. If the game is too large to fully enumerate the tree, however, GALA is unable to do any reasoning whatsoever.

A key design consideration for POGDDL has been to facilitate single-point decision algorithms that do not assume that the full tree can be generated. When reasoning about partially observable games in this manner, it is crucial for agents to be able to reason effectively not only about which predicates may be true in the current state but also about what other agents know. None of the constructs in the language explicitly define what an agent knows about the current state or what an

agent knows about what other agent knows. Instead, the epistemic system of POGDDL is implicit.

The information that an agent has to work with when deliberating are the POGDDL description of the game rules and the sequence of observations that it has received up to the current point in the game, as provided by those rules. The POGDDL description gives the agent the ability to perform the following operations: (1) given a game state, determine which actions are legal; (2) apply an action to a state to produce a new state and determine which observations each player should receive; (3) determine payoffs for each player given a terminal state. By simulating action sequences according the rules, the agent can identify possible sequences of moves that are consistent with its observations.

For example, suppose that P_k has received observation sequence $o_{1:t}$. Suppose the agent simulates a random sequence of t actions $a_{1:t}$ that are legal according to the rules. The result of the simulation is a full game state $s = \langle P, o^{(1)}_{1:t}, o^{(2)}_{1:t}, \ldots, o^{(K)}_{1:t} \rangle$. This simulated state defines truth values for all predicates and observation lists for P_k and all its opponents. Is this the true state of the game? The agent does not know. If $s.o^{(k)}_{1:t} \neq o_{1:t}$, then it definitely is not, because the observations actually received are inconsistent with the observations that would have been received under $a_{1:t}$.

On the other hand, if $s.o^{(k)}_{1:t}$ does match $o_{1:t}$, then $a_{1:t}$ is a plausible game history and s is a plausible current state. Suppose that, through simulation, P_k is able to identify all the plausible action sequences and, therefore, all the possible current states. The player's information set comprises these states. Now, how can P_k use this information to reason about what its opponent, $P_{\text{-}k}$, knows? For each possible state s, P_k can identify the observation list $s.o^{(\text{-}k)}_{1:t}$ that $P_{\text{-}k}$*would have received if that were the true state*. Using that list of observations and the ability to simulate action sequences from the game rules, P_k can identify the action sequences that $P_{\text{-}k}$ would consider possible if in fact s were the current state.

The ability to reason about information sets in this manner is the key feature of POGDDL. It may not be reasonable to assume that a player can generate all of the states in its information set. For some games, the information sets are simply too large. However, if an agent can at least effectively sample from the information set, then it can get an idea of what might be the current state, and what its opponents might consider possible. The ability to perform this operation

effectively is the subject of the next chapter.

3.6 Extensions

Our implementation of POGDDL includes additional features for convenience, which complicate the presentation of the formal semantics but which do not fundamentally change the intuition behind it. POGDDL supports universally and existentially quantified variables in preconditions and universally quantified variables in effects. Still, the set of ground terms is fixed and finite, so these features amount to syntactic sugar to make the expression of action schemata more compact in practice.

Conditional effects are allowed in POGDDL and have the same syntax as they do in PDDL. But note that they are not syntactic sugar as they are in PDDL, where an action a with a conditional effect premised on ϕ can be replaced with a pair of actions a', a'' where preconditions $a'^* = a^* \wedge \phi$ and $a''^* = a^* \wedge \neg\phi$. This transformation cannot be done in POGDDL. Because of partial observability in POGDDL, the effects of an action—including the observations provided to each player—may depend on values of propositions that are not known to the player executing the action. A well-defined game requires that the same actions be legal from all nodes (states) in the same information set, so this kind of flattening is not possible.

POGDDL also supports derived predicates, as in PDDL 2.2, which facilitates the expression of transitive closures, such as the reachable spaces for pieces in a kriegspiel game.

POGDDL has limited support for numerical fluents. These can be useful for keeping track of a running score in a game like Scrabble or a pile of chips in poker, where the use of a successor relation (common in GDL games) would be cumbersome. In practice, our reasoning system cannot be effectively applied to games where action preconditions include tests on fluents whose values are not known to all players. Nevertheless, we believe that their availability provides convenience often enough to justify their inclusion in the language.

Stochastic Effects Although we treat problems with uncertainty related to chance events, we view all actions as having deterministic effects. This is admittedly an unnatural worldview for

those accustomed to stochastic action languages. Such languages allow the specification of a probability distribution over effects, given a chosen action. Consider the specification of a robot navigating the physical world. A move action may be modeled as having a certain probability of success, of moving the requested distance in the specified direction. With some probability, the action results in a movement that is more or less than what the robot intended, due to wheel slippage or other circumstances.

We cannot model such an action directly in POGDDL because all effects are deterministic. However, we can model stochastic action effects by using a sequence of two actions. The first action is viewed as the decision to act and leads to deterministic observations for each player and a transfer of control to the *Chance* player. The moves that are then available to *Chance* correspond to the stochastic effects in the standard formulation, e.g., `move-exact`, `slip-high`, `slip-low`. POGDDL treats all alternatives for the *Chance* player as having equal probability, but the language could be modified to allow the specification of specific probabilities

```
(choose
        :parameters (?x ?y)
        :condition (is-wet)
        :select (05         (slip-low(?x ?y))
                 .95        (slip-high(?x ?y))))
```

CHAPTER 4

REASONING ABOUT INFORMATION SETS

The relationship between a description language for games and a general game playing system is similar to the relationship between a general purpose programming language and a compiler. In order for the compiler to produce optimized code, the language constructs must force the programmer to expose and concretely define key data structures and relationships that the compiler can reason about and efficiently manipulate. The language designer's dilemma is to create a syntax and semantics that are both easy to understand and use but that also provide the compiler with the information it needs in order to generate efficient code. Similarly, in order for a game-playing system to act effectively in partially observable domains, the description language constructs must be sufficiently intuitive to write and read but must also enable the system to perform key operations.

Existing game description languages for perfect information games allow a designer to describe the initial state of the game, the preconditions for each action, and the way the state of the world changes based on agent decisions. Using these operators, agents can theoretically explore the whole tree. In imperfect information games, a subset of the nodes of the tree may be indistinguishable to the agent because of its lack of full information. The current set of possible nodes is called the agent's information set. The key motivating factor behind our development of POGDDL is to facilitate the implementation of an operation that we call *information set generation*[1] This is the ability to identify plausible game histories that are consistent with an agent's observations.

In this chapter, we describe the prevailing approach for information set generation and explain its limitations. We then present our novel approach, which is based on a constraint-satisfaction

[1]The work in this chapter is based on joint work with Eyal Amir, presented at the GIGA-09 workshop [Richards and Amir, 2009]. The algorithms and results presented in Sections 4.3 and 4.4 have been accepted for publication in AAAI-12 [Richards and Amir, 2012]. The work in Section 4.6 is a collaboration with Abhishek Gupta, Osman Sarood, and Laxmikant Kale. Figure 4.4 was created by Osman Sarood.

formulation of the problem. We give information set generation results for well-known games, described in POGDDL. Our method performs effectively in many cases where the existing method fails to find even a single node in the information set.

Although our approach is effective, it is still computationally intensive. This can be problematic, since information set generation is only one of many subroutines that is repeatedly called by our game-playing system. We show that, in contrast to some operations needed for reasoning about perfection information games, the information set generation operation can be effectively parallelized. We give results for the game of kriegspiel (partially observable chess) that show efficient parallelization on up to 1,024 processors.

4.1 Definitions

A path through a game tree, determined by the choices of agents and the random acts of nature, can be described as a sequence of actions $a_1, a_2, \ldots, a_t$, where each a_i labels an edge of the tree and can be named by a grounded action in a POGDDL description or by a member of some A_{kji} action set in the corresponding EFG definition. An action sequence is abbreviated by $a_{1:t}$. We use $a_{1:t-1}a_t$ to denote the extension of $a_{1:t-1}$ by a_t to produce action sequence $a_{1:t}$. We use the following definitions in our subsequent discussion of information set generation.

Definition 2. *Let $S(a_{1:t})$ denote the state that is reached when action sequence $a_{1:t}$ is executed from the start state:*

$$S(a_{1:t}) = \begin{cases} s' & \text{when} \quad (S(a_{1:t-1}), a_t, s') \in T \\ s_0 & \text{when} \quad t = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 3. *Given a sequence of observations $o_{1:t}^{(k)}$ for P_k, the information set $I(o_{1:t}^{(k)})$ is the set of action sequences that lead to a state where P_k would have those observations. That is, a sequence $a_{1:t} \in I(o_{1:t}^{(k)})$ iff $S(a_{1:t})$ is defined and for the state $s' \stackrel{def}{=} \langle \mathsf{P}', o_{1:t}^{(1)'}, \ldots, o_{1:t}^{(K)'} \rangle$ such that $s' = S(a_{1:t})$, the observations $s'.o_{1:t}^{(k)} = o_{1:t}^{(k)}$.*

For every node v in a game tree, there is a unique path from the root of the tree to v. That path may be encoded as a sequence of actions. We can therefore view each member of an information set as a node in the tree or as the sequence of actions that leads to that node. We will use both perspectives. Viewing a member of an information set as a sequence of actions is useful, in combination with opponent modeling techniques, to assess the probability that that member represents the true state. On the other hand, viewing members of information sets as nodes is useful when estimating the value of that member, e.g., through a heuristic evaluation function or through the use of limited lookahead from that node.

The information set generation problem is to identify nodes in the game tree where the world state is consistent with a player's observations, or in other words, to identify which game histories are *possible*. Note that information set generation deals only with the question of what is *possible*, not what is *probable*. We will show that separating these two questions leads to more scalable solutions for practical, online decision-making system for general games.

4.2 Information Set Generation as Depth-First Search

Algorithm `DFS-Infoset`, shown in Figure 4.1, outlines a naïve but generally applicable way for identifying sequences of actions that are consistent with a player's observations. This straightforward approach is alluded to in [Parker *et al.*, 2005] and [Russell and Wolfe, 2005].

Given a list of observations $o_{1:t}^{(k)}$, the algorithm traverses the game tree in a depth-first manner, returning all sequences of actions for which the observations for P_k would be exactly those specified by $o_{1:t}^{(k)}$.

At the ith level of recursion, the function checks that the simulated observation for P_k at a node at the ith level of the tree is the same as the actual observation o_i given in the input, and only makes a recursive call if they match.

While straightforward and generally applicable in theory, the algorithm is often prohibitively expensive.

Suppose the first move of the game is a random face-down deal of one of n house cards, such

```
1: function DFS-INFOSET(a_{1:t}, o^{(k)}_{1:T}, I, s, t)
2:     if t > T then
3:         I ← I ∪ a_{1:t}
4:     else
5:         for each legal action a at s do
6:             s' = GETNEXTSTATE(s,a)
7:             if s'.o^{(k)}_{t+1} = o^{(k)}_{t+1} then
8:                 DFS-INFOSET(a_{1:t}a, o^{(k)}_{1:T}, I, s', t + 1)
```

Figure 4.1: Information set generation as depth-first search. The function should be called with an empty action sequence, P_k's current observation list, $I = \{\}$, $s = s_0$, and $t = 0$. When the top-level call returns, I holds the full information set.

that P_k's observation is `(dealt ?)`, regardless of which card is dealt. Then, m moves later, that card is revealed to be `c1`, so that $o^{(k)}_{m+1} =$ `(house c1)`. Let the branching factor of the game tree be b for levels 2 to $m + 1$. Now suppose that the first branch searched at level 1 corresponds to move `(deal c0)`. The `DFS-Infoset` function may potentially search a subtree of size b^m exhaustively, pruning only at level $m + 1$ each time when it is discovered that the simulated observation `(house c0)` does not match the actual observation `(house c1)`.

Such effort is both inefficient and unnecessary. P_k should be able to infer from $o^{(k)}_{m+1} =$ `(house c1)` that $a_1 =$`(deal c1)`, without having to search a large portion of the game tree up to depth $m + 1$. Systems targeted to one game tend to hard-code such domain-specific understanding.

4.3 Information Set Generation as Constraint Satisfaction

A better approach for the general case is to frame the information set problem as a more general constraint satisfaction problem that makes it possible to use *all* relevant observations to appropriately prune the search space. Algorithm `CSP-InfosetSample` (Figure 4.2) follows this approach.

This algorithm solves a general T-variable constraint satisfaction problem, where T is the total number of actions that have been executed by all players up to the current point in the game. For $1 \leq t \leq T$, the tth variable corresponds to the tth action taken, and the possible values are the

```
 1: function CSP-InfosetSample(SG, o^(k)_{1:T}, A[])
 2:     if Inconsistent(SG) then
 3:         return ∅
 4:     else if Complete(A) then
 5:         return A
 6:     else
 7:         t ← RandomUnsetTimestep(SG)
 8:         L ← PlausiblyLegalMoves(SG)
 9:         a = RandomElement(L)
10:         A[t] ← a
11:         SG' = Propagate(SG, t, a, o^(k)_{1:T})
12:         CSP-InfosetSample(SG', o^(k)_{1:T}, A)
```

Figure 4.2: Information Set Generation as a Constraint Satisfaction Problem.

possible actions in the game. A player's information set is the set of all feasible solutions to the CSP. As written, the algorithm returns a single feasible action sequence (or $\emptyset$, if the algorithm's non-deterministic choices lead to a conflict.) A sample of possible nodes from the information set can be obtained by running the algorithm repeatedly.[2]

The basics of the backtracking search are implemented in CSP-InfosetSample 4.3. The inputs are an initialized special data structure which we call a *stage graph* (described in detail below), the observation sequence for the active player up to the current point in the game, and an empty fixed-length vector A representing an action sequence.

At the initial invocation, each action in A is undefined. The call to CSP-InfosetSample results in either a conflict, an action sequence representing a node from the information set, or a recursive call with an additional slot in A fixed to a single concrete action. In the latter case, the time step selected is chosen at random (line 7) from among all the remaining time steps for which actions have not already been fixed by previous calls to the function. [3]

Having selected a particular time step t, the algorithm uses the game rules and information in the stage graph to identify all possible actions whose preconditions and effects are not inconsistent

[2] Alternatively, the algorithm can be modified to produce the full information set by changing the random selection at line 9 to iterate over all legal actions and then to collect the sequences found at line 5 into a set. This may be prohibitively expensive in games with large information sets.

[3] Rather than choosing at random, a commonly applied heuristic for such problems is to select the most constrained remaining variable, i.e., the decision point for which there are the fewest possible actions, given the constraints imposed by the other existing variable assignments.

```
1: function PROPAGATE(SG[], t, a, o^(k)_{1:T})
2:     Redo[i] ← FALSE for 1 ≤ i ≤ T
3:     Update SG[t-1] with precondition of a
4:     Update SG[t] with effects of a
5:     Redo[t-1], Redo[t + 1] ← TRUE
6:     return REEVALUATE(SG, Redo, o^(k)_{1:T})
7:
8: function REEVALUATE(SG[], Redo[], o^(k)_{1:T} )
9:     while ∃i Redo[i] = TRUE do
10:        for each t s.t. Redo[t] = TRUE do
11:            Redo[t] ←FALSE
12:            [DefPre,DefEff,PossEff] ←
13:             PLAUSIBLEACTIONINFO(SG[t-1],SG[t],o^(k)_t)
14:            for each p ∈ DefPre s.t. SG[t-1,p]=UNKNOWN do
15:                SG[t-1, p] ← TRUTHVAL(DefPre, p)
16:                Redo[t-1] ← TRUE
17:            for each p ∈ DefEff s.t. SG[t,p]=UNKNOWN do
18:                SG[t,p] ← TRUTHVAL(DefEff, p)
19:                Redo[t + 1] ← TRUE
20:    return SG
21:
22: function INITGRAPH(o^(k)_{1:T}, s_0 )
23:    SG[0] = s_0
24:    for 1 ≤ t ≤ T do
25:        [DefPre,DefEff,PossEff] ←
26:         PLAUSIBLEACTIONINFO(SG[t-1],SG[t],o^(k)_t)
27:        for each p ∈ DefPre s.t. SG[t-1,p]=UNKNOWN do
28:            SG[t-1, p] ← TRUTHVAL(DefPre,p)
29:            Redo[t-1] ← TRUE
30:        for each p ∈ P do
31:            if p ∈ DefEff then
32:                SG[t, p] ← TRUTHVAL(DefEff,p)
33:            else if p ∈ PossEff then
34:                SG[t, p]←UNKNOWN
35:            else
36:                SG[t, p] ← SG[t-1, p]
37:    return REEVALUATE(SG, Redo, o^(k)_{1:T})
```

Figure 4.3: Helper functions for CSP-INFOSETSAMPLE. TRUTHVAL(S,p) returns TRUE when p appears as a positive literal in S and FALSE when p is a negative literal.

with other actions in A that have been fixed by previous calls and whose resulting observation matches o_t. From this set of possible actions, one is randomly selected (line 9). From the perspective of solving a CSP, this corresponds to assigning a value to a variable. Fixing a particular action a for $A[t]$ (line 10) constrains the truth value of some predicates at time t-1 because of the preconditions of a and constrains some predicates at time t because of the effects of a.[4]

The PROPAGATE function returns an updated stage graph that encodes these ramifications (line 11). Then CSP-INFOSETSAMPLE is recursively called with the updated stage graph and action sequence. If a subset of actions selected in A lead to a situation where there are no legal actions possible for one or more other time steps, then a conflict is generated and the function returns (line 3). Otherwise, if an action is successfully assigned at every time step in A, then the action sequence A corresponds to a node in the information set and is returned. The CSP-INFOSETSAMPLE function can be viewed as the high-level management of the backtracking search for feasible solutions.

The PROPAGATE, REEVALUATE, and INITGRAPH functions manage the stage graph data structure that serves to reduce the branching factor of the search for solutions in the CSP by eliminating many actions from consideration. A stage graph is similar to the planning graph structures used in AI planning [Blum and Furst, 1995; Blum and Furst, 1997]. For each time step, there is a stage that stores the set of potential actions that are plausible at that timestep and a vector with one of three values for each proposition in the resulting state: TRUE, FALSE, or UNKNOWN.

In function INITGRAPH, The 0th stage of the graph is initialized with the (known) start state; there is no action associated with the 0th stage (line 23). For timesteps t from 1 to T, stage t of the graph is initialized (lines 24–36) by (1) computing the set of actions for which the resulting observation would match the observation $o_t^{(k)}$ actually received and for which the preconditions do not conflict with the propositional values at stage t-1 (line 26); (2) setting known values for the propositions at t that are common effects of all the potential actions (line 32); (3) propagating the values from stage t for propositions that are unaffected by any of the potential actions (line 36); and (4) marking as UNKNOWN those propositions that appear in the effects of at least one plausible action but not in all (line 34). Additionally, if there are any propositions entailed by the disjunction

[4]Our convention is that s_0 is the initial state and that action a_i is applied to state s_{i-1} to produce s_i, for $i \geq 1$.

of the preconditions of potential actions at t that were UNKNOWN at stage t-1, those values are updated and stage t-1 is marked to be re-evaluated. Specifically, the plausible action set at that stage must be recomputed. (lines 27–29).

The PLAUSIBLEACTIONINFO function identifies actions for a particular time step that are not inconsistent with information already stored in the stage graph. This is different than simply identifying provably legal actions because of the presence of UNKNOWN values in the stage graph.

Suppose we are considering plausible actions at time t. If action a has precondition $p_1 \wedge p_2$ and stage t holds that $p_1 =$ TRUE and $p_2 =$ FALSE, then the action is definitely not plausible. However, if $p_1 =$ TRUE and $p_2 =$ UNKNOWN, then a *might* be legal. We cannot prove that it *is* legal, but neither can we prove that it is not. We therefore say that it is "plausible" and that it does not conflict with the information in the stage graph.

Propositions (or their negations) that are entailed by the disjunction of the preconditions of all plausible actions *must* hold at the previous time step. These "definite preconditions" are returned from PLAUSIBLEACTIONINFO as `DefPre`. Similarly, the set of propositions that are entailed by the disjunction of the effects of possible actions *must* hold at t. These "definite effects" are returned as `DefEff`. Both `DefPre` and `DefEff` include one set of known positive literals and one set of known negative literals.

Propositions that are entailed by the effects of one or more of the possible actions but not all of them are "possible effects" and are returned as `PossEff`. The PLAUSIBLEACTIONINFO function requires as parameters the observation at time t and the graph stages for time steps t-1 and t. The function ensures that an action is excluded from the plausible set for t iff its precondition conflicts with known predicate truth values at t-1 or its effects conflict with known predicates at t.

During the initialization of stage t, it is possible to identify propositions whose truth values at t-1 are determined by `DefPre` but whose values were UNKNOWN during the initialization of stage t-1 (lines 27–29). Therefore, a vector of Boolean flags `Redo` is maintained that keep track of stages whose set of plausible actions may need to be recomputed.

The REEVALUATE function (lines 8–20) runs a fixed point calculation that repeatedly computes plausible action sets until there are no additional propositions at any stage whose values can be

changed from UNKNOWN to either TRUE or FALSE.

When a particular action a is fixed for time step t in CSP-INFOSETSAMPLE, the PROPAGATE function (lines 1–6) updates the stage graph to reflect the fact that the preconditions of a hold at t-1 and the effects of a hold at t. As in stage graph initialization, the ramifications of these updates are computed by invoking REEVALUATE. [5]

Time and Space Complexity The space taken by the stage graph is $O(|\mathcal{P}|T)$, where $|\mathcal{P}|$ is the number of ground propositions in the game description.The overall running time depends primarily on the REEVALUATE function, and in particular on the number of iterations of the loop at lines 9–19. The number of iterations is bounded by the number of propositions marked as UNKNOWN during the stage graph initialization phase. Each iteration of the REEVALUATE function (other than the last) reduces the number of UNKNOWNs by at least 1, so there are at most $|\mathcal{P}|T$ iterations.

The overall expense of information set generation depends on the expected number of times that CSP-INFOSETSAMPLE must be called in order to obtain a viable action sequence. This efficiency varies greatly from game to game and will require additional analysis.

4.4 Empirical Results

We have encoded three non-trivial games in POGDDL. Racko and Battleship are popular table games; the Game of Pure Strategy comes from the game theory literature [Luce and Raiffa, 1957]. For each game type, we varied complexity parameters (number of cards, size of game grid, etc.) to measure the performance and scalability of both `DFS-Infoset` and `CSP-InfosetSample`. For each game instance, we generated ten random positions and tested each algorithm's ability to sample from the current player's information set. For each position studied, each algorithm was allotted up to 30 seconds to search for nodes in the information set.

[5] In fact, the stage graph need only allocate space for the mutable subset of $\mathcal{P}$, *i.e.*, those propositions that appear in at least one action's effects.

4.4.1 Racko

Racko consists of a deck cards numbered 1 to n and a rack for each player with k ordered slots, with $n > 2k$. The cards are shuffled and then the first $2k$ cards are dealt into the players' rack slots, such that players see only their own cards. The next card is placed face-up as the initial discard pile, and the remainder of the cards are placed face-down as the draw pile. Each player's goal is to get the cards in his rack in ascending order. On each turn, a player may exchange a card on his rack with the top card from the draw pile or the most recently discarded card. The card that is swapped out is placed face-up on the discard pile. The game ends when one player's rack is totally ordered or when a set number of turns have been played.

Results for information set generation in Racko are shown in Table 4.1 (left). The table shows experimental data for instances of the game ranging in complexity from six slots per player and 30 total cards to the standard configuration of 10 slots per player and 60 total cards. Each entry in the table shows the number of information set nodes identified within the time limit. `CSP-InfosetSample` successfully samples information set nodes in each case. By contrast, `DFS-Infoset` is not able to find a single information set node in 19 of 20 games.

In the standard game, a player may have over ten billion nodes in its information set, so the ability to sample from the information set (rather than enumerate it in full) is critical. As play proceeds and a player has opportunities to see what cards the opponent swaps out of its initial rack, the size of the information sets decreases. The propagation of the ramifications of these observations through the stage graph allows the `CSP-InfosetSample` algorithm to concretely identify the only plausible actions by the dealer during the rack initialization phase, which greatly improves the efficiency of search. This kind of inference would be hard-coded in a system designed specifically to play Racko. `CSP-InfosetSample` implements this reasoning in a generalized form.

	crds slts	30	40	50	60	cells ships	4	6	8	10	crds bids	10	20	30	40
CSP	6	131	106	109	149	1	86	287	177	555	2	300	300	300	300
DFS		0	30	0	0		86	232	114	166		300	300	60	30
CSP	7	175	158	126	158	2	70	121	2161	792	4	300	300	300	300
DFS		0	0	0	0		70	117	141	36		0	30	0	30
CSP	8	278	146	180	228	3	39	656	318	792	6	300	300	300	300
DFS		0	0	0	0		39	65	0	0		0	0	0	0
CSP	9	260	158	118	87	4	1598	90	345	865	8	300	300	300	300
DFS		0	0	0	0		194	1	0	0		0	0	0	0
CSP	10	308	273	106	179	5	NA	173	1087	1074	10	300	300	300	300
DFS		0	0	0	0			0	0	0		0	0	0	0

Table 4.1: **(left)** Racko. Variables: slots per player (row), total cards (column). **(middle)** Battleship. Variables: ships per player (row), grid cells per side (column). **(right)** Game of Pure Strategy (GOPS). Variables: bidding rounds (row), extra cards (column).

4.4.2 Battleship

In Battleship, players secretly deploy five ships on a 10x10 grid. Each ship occupies 2–5 consecutive horizontal or vertical grid spaces. Players then take turns calling out grid cells where they believe the opponent's ships are. When a player calls out a cell, the opponent responds with 'hit' if the cell is occupied by a ship and 'miss' if it is not. The opponent is required to declare which ship was hit and must also declare 'sunk' if all other cells occupied by that ship have already been hit. The first player to sink all of her opponent's ships wins.

We varied the number of ships in the game from one to five and the size of the grid (per side) from four cells to ten. In our experiments (Table 4.1, middle), `CSP-InfosetSample` is always successful in sampling information set nodes and in many cases produces the full information set. By contrast, the DFS algorithm finds fewer nodes in the information set and cannot find any information set nodes for many of the larger instances (more than three ships, or grids larger than 6x6).

4.4.3 Game of Pure Strategy

The Game of Pure Strategy (GOPS) is described in [Luce and Raiffa, 1957]. A subset of a deck of cards is dealt such that each player has n cards and n additional cards lie in the middle. One by one, the cards in the middle are exposed. Players use the cards in their own hands to bid on the card shown. Players simultaneously declare their own bids, with the high bidder winning the card and scoring the number of points shown on the card. The player who has accumulated the most points at the end of the n bidding rounds wins.

Our results for GOPS (Table 4.1, right) show that, once again, `CSP-InfosetSample` succeeds in every case, while the `DFS-Infoset` is unable to find any information set nodes in over half of the game instances.

4.5 Information Set Sampling as Particle Filtering

An alternative approach to information set sampling would be particle filtering. Algorithm 4.1 shows a straightforward adaptation of the basic particle filtering algorithm described in [Russell and Norvig, 2010]. Particle filtering algorithms are designed to efficiently approximate probability distributions in dynamic Bayesian networks (DBNs) by iteratively updating a population of samples using the conditional probability tables of the DBNs and reweighting the samples based on how consistent they are with a given observation sequence.

Particle filtering could be adapted for use in general imperfect information games as follows. Each particle in the population of samples represents a possible node in the game tree, encoded implicitly as a sequence of actions. At iteration t, the ith particle in the population of samples, $S[i]$, encodes some sequence of actions $a_{1:t}$. Recall that actions are edges in the game tree, so that a sequence of actions from the root of the game tree maps to a unique node in the tree.

The proportion of samples at iteration T that represent a particular action sequence $a_{1:T}$ gives the approximate probability of the corresponding node in the information set at time T. At each iteration, each sample $S[i]$ is updated (line 11) by (1) using knowledge of the rules Γ to identify legal moves from the current game represented by $S[i]$, and (2) depending on whose turn it is at

Algorithm 4.1 Information Set Generation using Particle Filtering

1: **function** PARTICLE-INFOSET($o_{1:T}$, t, N, Γ, σ_k, $\hat{\sigma}_{-k}$)
2: **inputs**: $o_{1:T}$, the observation sequence for sequence up to current depth T
3: t, depth of current samples
4: N, the number of samples to be maintained
5: Γ, description of game rules, including the transition model $\mathbf{P}(s_{t+1}|s_0, a_{1:t})$,
6: the observation model $\mathbf{P}(o_t|s_t, a_t)$, and σ_0, the true strategy for CHANCE
7: σ_k, the current player's true strategy, and $\hat{\sigma}_{-k}$, a model for opponents
8: **persistent**: S, a vector of size N, initialized with s_0 (maintains full action sequences)
9: **local variables**: W, a vector of weights of size N
10: **for** $i = 1$ to N **do**
11: $S[i] \leftarrow$ sample based on $S[i]$, Γ, σ_k, $\hat{\sigma}_{-k}$.
12: $W[i] \leftarrow P(o_{1:t+1}|S[i])$
13: $S \leftarrow$ WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S, W)
14: **return** S

t, using σ_0, σ_k, or $\hat{\sigma}_{-k}$ to provide the distribution that determines the probability that $S[i]$ will be extended with any particular action a. Then, at (line 12), the weight for each sample $W[i]$ is determined based on the observation rules given in Γ. The purpose of the resampling phase (line 13) is to ensure that the proportion of samples in the population with sequence $a_{1:T}$ approximates the probability that the true state of the game is $S(a_{1:T})$.

Although game trees can in fact be viewed as a special case of DBNs, this particle filtering algorithm is not well suited to the information set generation problem. It is already well known that particle filtering algorithms can suffer from inefficiency when the evidence variables depend on many variables that come earlier in the topological ordering of the DBN. In POGDDL, the probabilistic uncertainty of the environment (such as dice rolls or card shuffling or unknown moves of the opponent) are captured by the transition function defined in Γ and the strategies of the players σ. Given a known sequence of actions, the observation sequence is deterministic. Therefore, the weights produced at line 12 are always 1 or 0. In games with large branching factors, it may not be uncommon for all particles to acquire 0 weight at the same time.

Suppose that early actions made by CHANCE correspond to random deals of cards whose identities are only revealed by actions that come late in the sequence. A player's observations are the same for all such unknown deals (e.g., $o_i = dealt(opp, ?)$, so that samples corresponding to all

deals are equally likely to be maintained in the early iterations of the filtering. When the card is actually revealed later on and the observation uniquely corresponds to the card's identity, the probability that any of the samples in the population will match (i.e., be assigned a weight of 1 at line 12) can be vanishingly small.

4.6 Parallel Implementation

As we have noted, theoretical solutions for both fully and partially observable games can be found in polynomial time in the number of nodes in the tree. Since the size of the tree is often too large to make even polynomial algorithms feasible, approximation algorithms often rely on tree search, and efficient implementation becomes critical.

With the current silicon-based uniprocessors approaching their theoretical limitations, additional speedup for software programs must be achieved through efficient parallelization. High-end commodity computing systems may have thousands of processing cores, while the number of cores in state-of-the-art supercomputers is approaching one million. Leveraging this additional processing power to improve decision-making in game tree search programs has proved challenging.

Brute-force tree search capabilities played a key role in the success of IBM's DEEPBLUE in its 1996 and 1997 matches against World Chess Champion Garry Kasparov. The move expansion algorithm and heuristic evaluation function were implemented directly in hardware, enabling DEEPBLUE to evaluate an average of 100 million moves per second [Campbell *et al.*, 2002].

By contrast, GGP systems built around GDL that participate in the annual AAAI competitions may be running symbolic theorem-proving operations in software to generate successor states and may be able to generate only tens or hundreds of nodes per second [Genesereth *et al.*, 2005]. Since tree search is an integral part of successful general game playing algorithms and decision quality is, to some degree, a function of search depth, some attention must be paid to implementation details.

Early research in parallel game tree search focused on the alpha-beta pruning algorithm, with applications to chess [Felten and Otto, 1988; Huntbach and Burton, 1988; Marsland and Campbell, 1982; Marsland, 1986; Schaeffer, 1989] and othello [Ferguson and Korf, 1988]. In this section,

we explore the implementation issues related to the efficient parallelization of our information set generation routine. We show how some aspects of partially observable games make them more amenable to parallelization than perfect information games.

Let $\tau(g, 1)$ be the running time of a search of tree g on a single processor and let $\tau(g, p)$ denote the running time when search is parallelized on p processors.

The *speedup* achieved by parallelizing the search in tree g on p processors is

$$s(g, p) = \frac{\tau(g, 1)}{\tau(g, p)}.$$

Efficiency is defined as

$$\varepsilon(g, p) = \frac{s(g, p)}{p}.$$

The efficiency is much less than 1 when the parallelization is hampered by (1) communication between processors, (2) processors idling between computations due to load imbalance or communication barriers, (3) redundant computations performed to avoid communication between processors, and/or (4) speculative loss, which is computation that is performed in parallel that would have been preempted in a sequential execution. Efficiency can be greater than 1 (superlinear speedups) if the parallel overhead is low and if the parallelization causes significant improvements to the performance of the memory/storage hierarchy (e.g., better cache performance).

The most straightforward way to parallelize search is to assign the various processors a specific part of the tree to explore. But it could be the case that all or part of the subtree assigned to, say, processor p would have been completely pruned based on the results from the search performed by processors 1 to $p - 1$. Executing the work in parallel is speculative because some of the work may turn out to be superfluous. This loss has an adverse effect on the overall speedup achieved by the parallelization.

DEEPBLUE, in spite of its success, likely suffered from a high level of inefficiency because of speculative loss. Many of the hundreds of millions of nodes analyzed by the 192 special-purpose move generator chips may not have been explored had the computation been performed in serial. Modern chess-playing programs, which are far superior to the top echelon of human players, run on commodity hardware on a small number of cores (often fewer than 10) and analyze only a few

million moves per second.

Systems for imperfect information games have much higher potential for high efficiency on modern machine because there tends to be more work that can be done in parallel. In particular, for games with large information sets, the information set generation routine shows great promise for efficient parallelization.

4.6.1 Kriegspiel

We have implemented a search engine for information set generation in kriegspiel (partially observable chess). Kriegspiel is played on a chessboard with a regular set of chess men. Players see their own pieces but not their opponent's. When a player attempts a move that is legal from the perspective of his own board but is illegal because of the configuration of the opponent's pieces, a referee announces that the move is disallowed and the player must make another move.[6]

4.6.2 Charm++

We built our search engine on top of the CHARM++ run-time system for parallel processing [Kalé and Krishnan, 1993; Kale and Zheng, 2009]. CHARM++ is an extension to the C++ language that provides machine-independent infrastructure for parallel computation. The language and its accompanying runtime system have been ported to many shared-memory and distributed-memory platforms.

In contrast to other popular parallel programming frameworks such as MPI and OPENMP, the CHARM++ language is based on object-oriented programming principles. The programmer is responsible to decompose its problem into a collection of objects that represent elements of work to be done. Typically the number of work objects far exceeds the number of processing elements (cores) available. These objects are defined so as to be able to be migrated between processors *during execution*, in order to improve the overall load balancing.

[6]Historically, kriegspiel required a third-party moderator who kept track of the full game state by managing a board with both sets of pieces. Players submitted their move requests in secret, and he notified the players when it was their turn to move. It is now common to play kriegspiel over a computer network, with a software system acting as moderator.

Work objects, known as *chares*, communicate with each other via asynchronous message-passing. Objects can invoke *entry methods* on other objects, which are executed according to a schedule set by the CHARM++ runtime system. The runtime is responsible to interleave computation and communication tasks in order to maximize efficiency. To facilitate this, each processor maintains an *incoming queue* of messages targeted to chares that it currently holds. A corresponding *outgoing queue* collects messages generated by objects on the processor that need to be sent to objects assigned to other processors.

Some chares may inherently require more computation than others, but the amount of computation associated with each piece of work is not always known before execution. In tree search, for example, a chare may be defined as the computation of a minimax value on a node in the tree using alpha-beta pruning. A natural decomposition is to assign different nodes to different processors for evaluation. Because some nodes will be pruned at shallower levels of the tree than others, the amount of work performed at each subtree can vary considerably. In a traditional parallel framework, this would result in many processors idling while a small number of processors execute the larger workloads. By allowing the CHARM++ runtime system to schedule a large number of small pieces of work, better adaptive load balancing is achieved.

4.6.3 The CHARM++ Parallel State Space Search Engine

Sun *et al.* have developed the Parallel State Space Search Engine (PARSSSE) within the CHARM++ framework, which exposes an API that is useful for a variety of graph-based and tree-based search problems [Sun *et al.*, 2011]. In PARSSSE the fundamental unit of work is the processing of a node. Generally, this means identifying whether the node corresponds to a solution and/or generating descendant nodes and spawning work tasks related to them. A task is *terminal* if it spawns no children.

A *grainsize control* parameter specifies a bound on the size of a piece of work that can be passed along to the runtime system for scheduling. Once a task has been subdivided into an appropriate number of subtasks, those tasks run to completion on a single processor without preemption and without generating any additional parallel tasks. If the grainsize is too large, then the number

of pieces of work to be done may be lower than the number of processors available, forcing some processors to idle. If the grainsize is too small, then the runtime system is flooded with a large number of trivial tasks, and the system gets bogged down by the overhead of creating and scheduling tasks instead of actually doing them. PARSSSE supports an adaptive grainsize management, so that the threshold for sequential processing can be changed during the execution of the program, depending on the properties of the problem and/or the realtime performance analysis of the runtime system.

PARSSSE currently supports two different load balancing strategies. The first is the randomized strategy. In this approach, each newly created object is assigned to a random processor. Generally, if a large enough number of chares is created, the system will achieve an approximately uniform distribution of work across processors. The approach is also appealing because of its simplicity. However, randomization can sometimes lead to inefficiencies from parallel overhead. For a program running on p processors, the probability that a piece of work is assigned to a processor other than the one that created it is $1-1/p$. As the number of processors gets large, this probability approaches 1, which means that the parallel overhead related to the creation and scheduling of a new task is incurred for almost every piece of new work.

A second load balancing strategy is known as *work stealing*. In this approach, all new tasks are inserted in the message queue of the processor that spawns them. Whenever a processor is idle it becomes a *thief*. A thief randomly selects another processor as its *victim* and sends a message to that processor requesting work. When the victim receives the message, it removes the work with the highest priority from its own queue and sends it to the thief. If it has no work available, it sends a negative acknowledgment message to the thief, which causes the thief to look for another victim.

The advantage of the work stealing strategy is that the message-passing overhead is only incurred if at least one processor is idling. Furthermore, because victims respond by sending the highest priority work, the most important tasks are more likely to get done sooner.

In addition to these strategies, a user may define its own load balancing strategies by overriding CHARM++ task dispersal functions.

Procs	Time (s)	Speedup
1	1524	1
2	812	1.88
4	407	3.74
8	208	7.32
16	102	14.8
32	53.2	28.7
64	28.0	54.5
128	14.9	102
256	7.70	197
512	4.59	331
1024	2.64	576

Table 4.2: Speedups for a randomly selected problem instance on 1–1,024 processors. The input observations were from the 10th ply of the game tree and the size of the information set in this instance is 4,495,121.

4.6.4 Empirical Results

Table 4.2 shows the performance results of DFS-INFOSET on a randomly selected problem instance in which the size of the information set was about 4.5 million.[7] The table shows that the problem is indeed highly parallelizable. The speedups are nearly linear for up to 32 processors, and continue to increase up to 1,024 processors.[8]

In this particular instance, the sequential algorithm takes almost 25 minutes to run. But when parallelized on 1,024 processors, the search takes less than three seconds. This represents a speedup of 576 times. It might not be reasonable to use the algorithm at all if it takes 25 minutes or more to run for some turns. But if the cost is only a few seconds, it may become feasible.

We also analyzed the *isoefficiency* of the problem, which is a measurement of how the efficiency of the parallelization is affected by the *size* of the problem. To test this we varied the number of plies in the test instance from 8 to 12. We used the work stealing load balancing strategy. The results are shown in Table 4.4. In the easiest case (depth 8), there isn't enough additional parallel work to justify the jump from 256 to 1,024 processors, and performance actually degrades.

[7] This instance is not necessarily representative of all games or states of the game, but it was not cherry-picked in any way.

[8] The performance results reported in this section were obtained by running our program on the Blueprint machine with up to 1,024 processors.

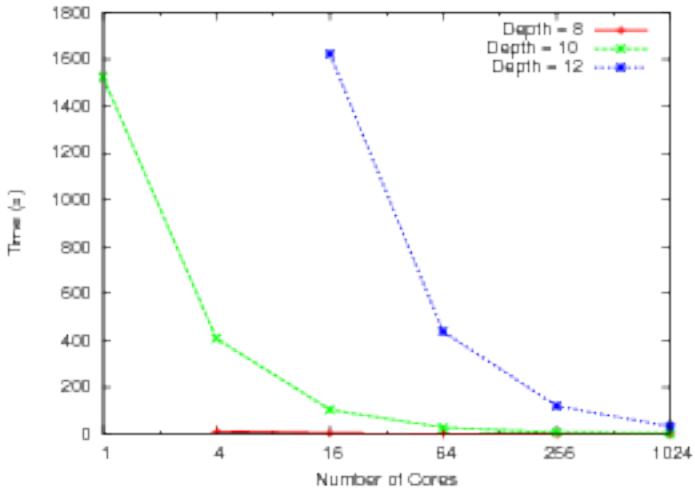

Figure 4.4: Running times for three different sizes of problem as the number of cores used increases, with a linear y-axis for the running time and a log-scale x-axis for the number of cores.

Aside from this anomaly, however, we see consistent speedups as we increase the number of processors, an indication that the search algorithm is able to take advantage of more processors when the problems become more difficult.

In addition to the random instances, we also evaluated our information set generation routine on 40 positions from actual kriespiel games, including human vs. human and computer vs. computer instances. Many of these were trivial to solve in a few seconds or less on a desktop computer. The running times and speedups for two of the more challenging instances are shown in Tables 4.5–4.6.

These tables show the results for runs on one to $1,024$ cores. Each column corresponds to a different grainsize parameter. In this case, the grainsize is the number of levels of the search tree that are searched in parallel. Up to that threshold, each node that is generated spawns a new piece of work, which the CHARM++ runtime schedules to execute on one of the processing elements according to its load balancing strategy. Above the threshold the entire subtree is searched to completion on a single processor, with no additional parallel overhead.

Note that the optimal grainsize varies for the different numbers of cores and the pattern is relatively consistent across problem instances. The optimal grainsize increases as the number of cores increases. This makes sense, because as this parameter grows, the number of distinct pieces of work increases. When the number of pieces of work per processor is high, much of

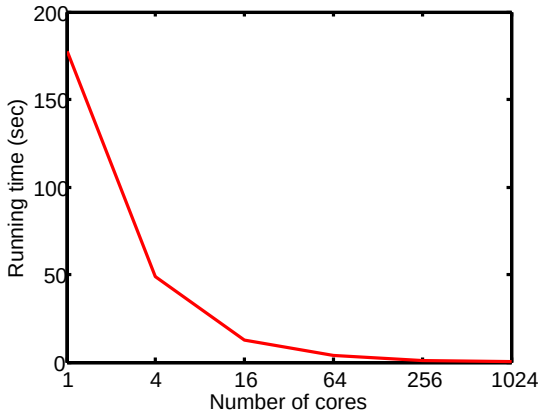

Figure 4.5: Kriegspiel position from Virgil-David 1969 with information set size 764,209.

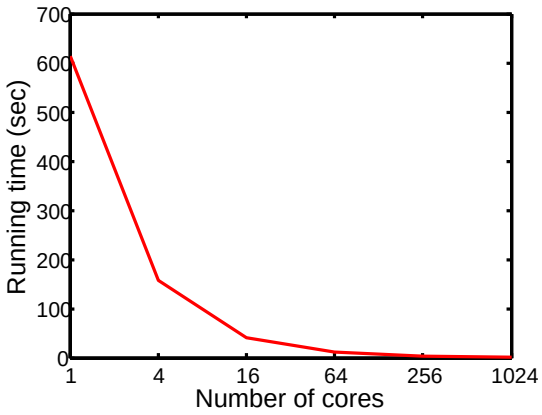

Figure 4.6: Kriegspiel position from Game 6 of Kbot-Darkboard at the 2006 Computer Chess championship. Information set size is 5,099,257.

the overhead of parallelization is wasted. But as the number of cores increases, the work can be efficiently broken up into smaller pieces and doled out to all the cores. With more pieces of work, it becomes easier to balance the load.

PE/G	2	4	6	8
1	**177**	178	184	493
4	71	53	**48**	75
16	34	19	**12**	16
64	24	7	**3.7**	4.1
256	24	4.4	1.2	**1.0**
1024	26	3.8	0.64	**0.40**

Table 4.3: Running times, in seconds, to find full information set of size 764,209 at position 10 of Virgil-David 1969. Optimal grainsize (G) for each number of cores (PE) is shown in bold

PE/G	2	4	6	8
1	617	**612**	650	3312
4	213	170	**158**	367
16	83	49	**40**	64
64	54	19	**11**	14.9
256	54	9.8	3.5	**3.5**
1024	58	5.5	1.3	**1.2**

Table 4.4: Running times, in seconds, to find full information set of size 5,099,257 at position 10 of Game 6 Kbot-Darkboard 2006. Optimal grainsize for each number of cores is shown in bold.

PE/G	2	4	6	8
1	1	0.99	0.96	0.36
4	2.48	3.34	3.63	2.35
16	5.19	9.27	13.8	10.7
64	7.25	23.6	47.39	43.1
256	7.1	40.1	145	172
1024	6.61	46.5	276	**443**

Table 4.5: Speedups for Virgil-David 1969.

Note that both problems scale well even up to 1,024 processors. We achieve a speedup of 443 in the game between humans and up to 536 in the game between computers. Figures 4.5 and 4.6 show the plots of the running times for the best grainsize for each core setting. These plots show that the running time continues to decrease as the number of cores increases.

PE/G	2	4	6	8
1	1	1	0.95	0.19
4	2.89	3.63	3.89	1.68
16	7.41	12.4	15.1	9.57
64	11.3	32.3	52.9	41.4
256	11.3	62.9	172	174
1024	10.6	111	464	**536**

Table 4.6: Speedups for Kbot-Darkboard 2006.

Our results compare favorably to previous work in parallel game tree search, shown in Table 4.7. The previous best parallelization results were for parallelizing alpha-beta pruning. Ferguson and Korf achieved a speedup of 12 on 32 cores in the game of Othello [Ferguson and Korf, 1988]. In the game of chess, Felton and Otto showed a speedup of 100 on 256 cores [Felten and Otto, 1988]. This limited scalability is due to speculative loss. Parts of the game tree that are searched during the parallel execution would have been pruned had the algorithm run sequentially. This is a fundamental limitation in alpha-beta search.

For partially observable games, information set generation represents a different variety of tree search that does not suffer from speculative loss. All of the nodes that are visited in parallel would have also been visited in a sequential run. The key challenge for parallelizing information set generation is load balancing: some parts of the tree will be pruned at a higher level than others, resulting in a wide variation in cost for each subtree. The CHARM++/PARSSSE framework is perfectly suited to handle the load balancing issues that this problem naturally presents.

Note also that information set generation is just one subroutine needed to reason about partially observable games. Once plausible game histories have been found, an agent may simulate some sequences of moves up to previous moves made by opponents and then recursively call the information set generation procedure from the perspective of the opponent. This results in many parallel applications of information set generation. Furthermore, given a node in an information set, it is common for an agent to perform forward search on that node (e.g., using MCTS). With these two additional levels of potential parallelism on top of the information set generation problem, it is conceivable that the problem of playing partially observable games could scale efficiently to millions of processors or more.

Authors	**Game**	**Year**	**Best Speedup (cores)**
Ferguson & Korf	Othello	1988	12(32)
Felton & Otto	Chess	1988	100(256)
Richards *et al.*	Kriegspiel	2012	178(256)
Richards *et al.*	Kriegspiel	2012	**536(1024)**

Table 4.7: Best Speedups reported for game tree applications.

CHAPTER 5

POINT DECISIONS

In this chapter, we develop and analyze algorithms for decision making in partially observable games, assuming an environment where the agent receives a description of the game rules in a logical language and then interleaves sensing, deliberation, and action. Recall that in a game tree, each node represents a decision point or a chance event. From a game-theoretic view of optimal decision making, a player's strategy for the game comprises the probability distributions over alternatives at each of its decision points.

We noted earlier that the ESLP algorithm finds the Nash equilibrium of a game by converting the game tree to a linear program [Koller *et al.*, 1994]. ESLP is the back end for GALA, a general game playing description language and reasoning engine [Koller and Pfeffer, 1997].

In practice, however, game trees are often so large that not all decision points and their associated alternatives can be enumerated, rendering ESLP intractable. A more practical approach, which we adopt here, is to search for the optimal probability distribution only for the current decision point, based on observations received during the game and limited computation. We refer to this as a *point decision.*

Theoretically, we could apply the point-decision algorithm at *every* decision point in the game and then combine the probability distributions for each individual decision into a mixed strategy for the whole game. Ideally, the global strategy induced by repeated application of the local decision algorithm should be a Nash equilibrium solution, and the performance of the induced global strategy should approach the performance of the Nash equilibrium solution as the number of nodes explored by the search agent approaches the number of nodes in the tree.

We have noted that the performance of a tree search-based decision agent depends, to a large degree, on its heuristic evaluation function. Whether the HEF is computed through MCTS or

through traditional means, the goal of applying an HEF to a node is generally the same: to estimate the expected value of a game that passes through that node.

In this chapter, we explore the idea of an *ideal HEF*. We noted earlier that if a heuristic evaluation function for perfect information games is perfect (i.e., returns the true minimax value for every node), then there is no need for advanced search procedures. A player could act optimally by applying the HEF to descendants of the current node and returning the action that leads to the node with the best value. This would result in optimal (equilibrium) behavior. However, this approach does not translate exactly to the imperfect information case.

In Section 5.1 we review the concept of point decisions and the practical use of HEFs in the context of perfect information games. In Section 5.2, we present additional definitions needed to extend these concepts to partially observable games.

In Section 5.3, we reframe the general HEF-based point-decision algorithm of [Parker *et al.*, 2005] in terms of information sets. This algorithm requires no explicit opponent modeling. We show that if the game has a dominant strategy, then there is an HEF that can be used with this algorithm to induce an optimal global strategy.

In Section 5.4, we expand this algorithm to use explicit opponent modeling, and we derive a general method for doing that modeling. We show that if the game has a pure strategy equilibrium, then there is still an ideal HEF that induces a globally optimal strategy. We also show, however, that if there is no pure strategy, then there is *no* HEF that induces a global strategy.

This suggests the need for an alternate approach for dealing with mixed-strategy games. In Section 5.5, we present a linear programming approach that is based on ESLP. We show that under certain restrictions, there is an HEF for this algorithm that induces the global mixed equilibrium strategy.

Since a perfect HEF function for any non-trivial game is unlikely to be found, even if it exists in theory, the key benefit of these results is in providing general frameworks in which different kinds of partially observable games can be approached. Some games may not have dominant strategies but may have pure strategies which are close to optimal. And since optimal opponents also generally do not exist in practice, strategies which are close to optimal may often be good

enough. As a case study, we consider the game of Scrabble in Section 5.6[1]. We present an agent based on our opponent modeling techniques which outperformed the *de facto* world champion computer program.

Throughout this chapter, we assume there is common knowledge about the mechanics of the game. In particular, we assume that an agent can perform the following operations: (1) given a logical representation of a complete world state, determine the set of legal actions; (2) given a state and a legal action, determine the next state and the observations that will be provided to each agent; (3) given a sequence of observations, identify the set of sequences of actions that characterize the active player's information set; (4) given a terminal state, recognize that the state is terminal and determine the payoff for each player; (5) given a non-terminal state, provide a heuristic estimate of the expected value of the game if that node is reached. We assume that there is common knowledge that all agents have these capabilities.

Each of these operations requires a non-trivial amount of computation, particularly the information set generation operation, as we saw in Chapter 4. The decision-making algorithms described here will be of practical use only in situations where these operations can be performed efficiently enough to be repeatedly applied. From a POGDDL game description, identifying the legal moves from a fixed position requires checking the satisfiability of an arbitrary logical formula. It is possible to design games for which this operation alone is exponential in the size of the input. On the other hand, for many games of interest, each of these operations can in fact be performed efficiently in practice, and the computational challenge stems from the size of the game tree rather than any single tree search operation. For the purposes of this chapter, we will therefore treat all of these queries as primitive operations. We model the computational boundedness of the agents by assuming that they are capable of performing only a fixed number of these primitive operations, rather than exploring the game tree in its entirety.

[1]This joint work with Eyal Amir was originally published in IJCAI-07 [Richards and Amir, 2007]

```
1: function CHOOSEMOVE(Node S(a_{1:t}))
2:     L = LEGALMOVES(S(a_{1:t}))
3:     for each a ∈ L do
4:         V(S(a_{1:t}), a) ← V(S(a_{1:t}a))
5:     return argmax_{a∈L} V(S(a_{1:t}), a)          ▷ argmin for min player
```

Figure 5.1: Decision procedure for a computationally unbounded agent in fully observable games.

5.1 Point Decisions in Fully Observable Games

A basic method for decision making in perfect information games is backwards induction. For computationally unbounded agents, the value of a node $V(S(a_{1:t}))$ can be computed as follows:

$$V(S(a_{1:t})) = \begin{cases} U(S(a_{1:t})) \text{ if } S(a_{1:t}) \in T \\ \max_{a \in L(S(a_{1:t}))} V(S(a_{1:t})) \text{ if } D(S(a_{1:t})) = P_1 \\ \min_{a \in L(S(a_{1:t}))} V(S(a_{1:t})) \text{ if } D(S(a_{1:t})) = P_2 \\ \sum_{a \in L(S(a_{1:t}))} \sigma_0(S(a_{1:t}), a) V(S(a_{1:t})) \text{ if } D(S(a_{1:t})) = P_0 \end{cases}$$

For terminal nodes, the value is the utility (payoff) to P_1, where it is assumed that the payoff to P_2 is the negation of that value. For non-terminal nodes that are decision points for P_1, the value is the maximum value of any of its descendants, since P_1 will seek to maximize the game's value. The value for P_2's decision points will be the minimum over the descendants, as P_2 seeks to minimize the game's value. The value of a chance node is its expected value: the sum of the values of the descendants, weighted by the probability $\sigma_0(S(a_{1:t}), a)$ that the descendant will be reached.

Figure 5.1 outlines a simple decision-making method based on this node evaluation scheme. Note that invoking CHOOSEMOVE from the root node would require the exploration of the entire game tree because of the recursive computations required to evaluate the node values at line 4.

In practice, for games described in logic, the costs of the node value computation are dominated by the determination of the legal moves at each non-terminal, and the identification of a next state given a state-action pair. A computationally bounded agent may perform only a limited number of such operations, making the full exploration of the tree infeasible.

Note also that in Figure 5.1 the modeling of the opponent is implicit: it is assumed that the

opponent employs the Nash equilibrium/expectiminimax solution). This may be a reasonable assumption when the game is small enough that both an agent and its opponent can run the minimax (or alpha-beta pruning) algorithm to completion. It is less clear how to proceed when the agents are computationally bounded.

Because real-world agents are bounded in the number of tree exploration operations they can perform at each decision point, it is generally necessary to approximate the node values. This can be done by (1) performing a limited lookahead search that evaluates possible future sequences of moves up to a certain depth and then (2) estimating the value of nodes beyond the search frontier. These estimates are made by way of a heuristic evaluation function An HEF uses locally available information to estimate a node's value. (See Section 2.6.)

We noted that Monte Carlo Tree Search algorithms do not require an HEF (See Section 2.7). An alternative view is that the simulated rollouts in MCTS act as a special kind of heuristic evaluation function. They involve some additional tree search operations, but this computational expense is bounded by a small constant (related to the maximum depth of the tree).

From the perspective of P_k using opponent model $\hat{\sigma}_{\text{-}k}$, d-level lookahead in the tree, and HEF ϕ, the estimated expected value of a game that reaches u is :

$$\hat{V}_d(u) = \begin{cases} \phi(u) \text{ if } d = 0 \\ U(u) \text{ if } u \in T \\ \max_{v \in N(u)} \hat{V}_{d\text{-}1}(v) \text{ if } P_k = P_1, D(u) = 1 \\ \min_{v \in N(u)} \hat{V}_{d\text{-}1}(v) \text{ if } P_k = P_2, D(u) = 2 \\ \sum_{a \in L(u)} \hat{\sigma}_{\text{-}k}(u,a) \hat{V}_{d\text{-}1}(N(u,a)) \text{ if } D(u) = \text{-}k \\ \sum_{a \in L(u)} \sigma_0(u,a) \hat{V}_{d\text{-}1}(N(u,a)) \text{ if } D(u) = 0 \end{cases}$$

As a general rule, it is expected that the deeper an agent searches and the closer the nodes on the search frontier are to the terminal nodes, the better those estimates will be. That is, for $d' > d$, we expect that $|\hat{V}_{d'}(u) - V(u)| \leq |\hat{V}_d(u) - V(u)|$, and in particular, that $|\hat{V}_{d'}(u) - V(u)| \leq |\phi(u) - V(u)|$. Otherwise, we would make decisions by simply choosing the descendant of the current state with the best ϕ value.

We noted that in MCTS, the implementation of ϕ may involve a small, bounded number of additional tree search operations. The computation of $\hat{\sigma}_{\text{-}k}$, the agent's opponent model, might similarly involve some additional search operations, this time from the perspective of the opponent.

The decision CHOOSEMOVE procedure in Figure 5.1 is easily adapted to use $\hat{V}_d(u)$ instead of $V(u)$ on line 7. The value of d is tuned based on the agent's computational budget. At each decision point u, an agent computes the values of $\hat{V}_d$ for each descendant of u and chooses the action that leads to the child with the best expected value.

Note the similarity between how opponent nodes and chance nodes are treated. Since an opponent model is simply an estimate of the probability distribution that $P_{\text{-}k}$ uses at its decision points, it is possible to compute a best response to the model by taking the weighted average of possible future states, just as with chance nodes.

5.2 Additional Prerequisites for Partially Observable Games

In a partially observable game, the true probability that a particular node is reached can be computed as follows:

$$P(a_{1:t}) = \prod_{t'=1}^{t} \sigma_{D(S(a_{1:t'}))}(S(a_{1:t'\text{-}1}), a_{t'}). \tag{5.1}$$

Recall that $\sigma_k(x, a)$ is the probability that P_k will choose a at x under strategy σ_k and that $D(x) = k$ if it is P_k's turn to move at x. Let $D_k(a_{1:t}) = \{t' | 1 \leq t' \leq t, D(S(a_{1:t})) = k\}$ and let $\Pi_k = \prod_{t' \in D_k(a_{1:t})} \sigma_k(S(a_{1:t'\text{-}1}), a_{t'})$. Then an equivalent expression is

$$P(a_{1:t}) = \Pi_0 \Pi_k \Pi_{\text{-}k} \tag{5.2}$$

Intuitively, the probability of reaching a node $S(a_{1:t})$ is the product of all the probabilities along the path $a_{1:t}$ that leads to it. Assuming perfect recall, the Π_k term will be equivalent for all nodes

in the active player's current information set. Thus, $P(a_{1:t})$ will be proportional to the products of the chance moves and the opponent's moves:

$$P(a_{1:t}) \propto \Pi_0 \Pi_{\text{-}k} \tag{5.3}$$

Definition 4. *For algorithm X, $\sigma_k[X]$ is the complete strategy induced if P_k invokes algorithm X at each of its decision points.*

Definition 5. *For action sequence $a_{1:t}$, let $E(a_{1:t})$ be the set of all extensions of $a_{1:t}$. A sequence $a_{1:T}$ is in $E(a_{1:t})$ iff $a_{1:t}$ is executable and $a_{1:T} = a_{1:t}a_{t+1:T}$ for some $a_{t+1:T}$. We denote by $E^+(a_{1:t})$ the set of terminal extensions: $a_{1:T} \in E^+(a_{1:t})$ if $a_{1:T} \in E(a_{1:t})$ and $S(a_{1:T})$ is terminal.*

Definition 6. *The* value *of the node $S(a_{1:t})$ under strategy profile σ reached by action sequence $a_{1:t}$ is*

$$V_\sigma(S(a_{1:t})) = \frac{\sum_{a_{1:T} \in E^+(a_{1:t})} P_\sigma(a_{1:T}) U(S(a_{1:T}))}{P_\sigma(a_{1:t})}$$

Definition 7. *The* value *of the information set $I(o_{1:t})$ under strategy profile σ characterized by observation sequence $o_{1:t}$ is*

$$V_\sigma(I(o_{1:t})) = \frac{\sum_{a_{1:t} \in I(o_{1:t})} \sum_{a_{1:T} \in E^+(a_{1:t})} P_\sigma(a_{1:T}) U(S(a_{1:T}))}{\sum_{a_{1:t} \in I(o_{1:t})} P_\sigma(a_{1:t})}$$

Definition 8. *A strategy σ_k^+ is* dominant *iff $\forall \sigma_{\text{-}k}\ \forall \sigma_k\ U_k(\langle \sigma_k^+, \sigma_{\text{-}k} \rangle) \geq U_k(\langle \sigma_k, \sigma_{\text{-}k} \rangle)$.*

Definition 9. *A strategy σ_k is* pure *iff for all $S(a_{1:t})$ such that $D(S(a_{1:t})) = k$ and for all $a \in \mathcal{L}(S(a_{1:t}))$, $\sigma_k(S(a_{1:t}), a) \in \{0, 1\}$. That is, all point decisions in σ_k are deterministic.*

Definition 10. *A strategy σ_k is* mixed *iff it is not pure.*

Definition 11. *A strategy σ_k is* consistent *with an opponent model $\hat{\sigma}_{\text{-}k}$ iff $\forall \sigma_{\text{-}k}\ U_{\text{-}k}(\sigma_k, \hat{\sigma}_{\text{-}k}) \geq U_{\text{-}k}(\sigma_k, \sigma_{\text{-}k})$.*

5.3 Games with Dominant Strategies

Algorithm 5.1 Simple Decisions for Partially Observable Games (SDPOG)

1: **function** CHOOSEMOVE(Obs $o_{1:t}$, $\hat{\phi}$, $\hat{\sigma}_{\text{-}k}$)
2: $\mathcal{I}$ = INFORMATIONSET($o_{1:t}$)
3: $\mathcal{L}$ = LEGALMOVES($\mathcal{I}$)
4: **for each** $a_{1:t} \in \mathcal{I}$ **do**
5: **for each** $a \in \mathcal{L}$ **do**
6: $\hat{P}(a_{1:t}) \leftarrow$ ESTPROBHISTORY($a_{1:t}, \hat{\sigma}_{\text{-}k}$)
7: $\hat{V}(a_{1:t}, a) \leftarrow \hat{\phi}(S(a_{1:t}a))\hat{P}(a_{1:t})$
8: **return** $argmax_{a \in \mathcal{L}} \sum_{a_{1:t} \in \mathcal{I}} \hat{V}(a_{1:t}, a)$

Algorithm 5.1 is a first-cut approach to decision making in partially observable games. This is the general game-playing algorithm from [Parker *et al.*, 2005], recast in terms of information sets. We refer to this as SDPOG (Simple Decisions for Partially Observable Games). Instead of passing in a set of possible game states as an argument, we pass in a player's observations and use an information set generation routine to identify the possible game histories. This emphasizes the fact that identifying possible states at each decision point requires computation. Parker and Nau describe a number of heuristics for the probability estimates (line 7) in a specific game, kriegspiel. They assume the use of a perfect-information search routine to estimate the state-action values at line 8. Here, we use an HEF $\hat{\phi}$ that produces the expected payoff under equilibrium if this node is reached.

Theorem 3. *If P_k has a dominant strategy σ_k^+ for P_k, then there is a perfect HEF $\hat{\phi}$ such that σ_k[SDPOG($\hat{\phi}$)] is σ_k^+.*

Proof. Let $V_{\sigma^*}(u)$ be the expected value of u under equilibrium σ^*. If there exists σ_k^+, then for all $o_{1:t}^{(k)}$, $\exists a \forall a_{1:t} \in I(o_{1:t}^{(k)}) \forall a' \neq a . V_{\sigma^*}(S(a_{1:t}a)) \geq V_{\sigma^*}(S(a_{1:t}a'))$. We need to show that a will be selected with probability 1. Now let $\hat{P}(\cdot)$ be *any* probability estimates. Choose $\hat{\phi} = V_{\sigma^*}$. Since

$$\hat{V}(a_{1:t}, a) = V_{\sigma^*}(S(a_{1:t}a))\hat{P}(a_{1:t}) \geq V_{\sigma^*}(S(a_{1:t}a'))\hat{P}(a_{1:t}) = \hat{V}(a_{1:t}, a')$$

for all a', it must be that

$$\sum_{a_{1:t} \in I(o_{1:t}^{(k)})} V_{\sigma^*}(S(a_{1:t}a))\hat{P}(a_{1:t}) \geq \sum_{a_{1:t} \in I(o_{1:t}^{(k)})} V_{\sigma^*}(S(a_{1:t}a'))\hat{P}(a_{1:t})$$

for all a'. Therefore a will be the action selected. □

While SDPOG is provably optimal for games with a dominant equilibrium strategy, it can be applied to any game, as long as the information set generation and HEF functions can be efficiently implemented. Given its simplicity, it is also a good candidate to try first when an off-the-shelf algorithm is needed.

5.3.1 Empirical Results

We now present some empirical results for SDPOG. The implementation of this algorithm for POGDDL descriptions represents the first general purpose point-decision system for partially observable games. We first present a simple game, which we call Difference. The game is played with cards numbered 1 to n, with one card dealt initially to the center and one card face-down to each player. Each player, in turn, draws up his hand to two cards and then plays one of his cards on the center card, scoring the absolute value of the difference between the card played and the previous center card (i.e., playing 7 on 3 or 3 on 7 scores four points). The game ends when the deck is exhausted; high score wins. The full POGDDL description is provided in Appendix B.4.

A simple HEF is to play whichever card scores the most points on the current turn. We refer to this as GREEDY. A more sophisticated strategy would consider probable opponent responses, based on the cards already played and those yet unseen. MCTS can be used as an alternative HEF. We created MCTS players that were allowed 100 and 1000 samples. The RANDOM strategy chooses uniformly at random over all legal moves. The results are shown in Table 5.1. For each entry, the first (resp. second) number gives the number of wins for the row (resp. column) player. For each pairing, we simulated 200 games, 100 games for each player acting as P_1.

As expected the GREEDY heuristic far outperforms RANDOM, winning more than 75% of

Difference	Greedy	MCTS-100	MCTS-1000
Random	47-153	28-172	25-175
Greedy		79-121	67-133
MCTS-100			91-109

Table 5.1: Results for SDPOG in the game of Difference.

the time. Against RANDOM, the MCTS-100 and MCTS-1000 players are even stronger than GREEDY is. The MCTS algorithms also win handily against GREEDY, and, as expected the additional samples for MCTS improve its relative performance. In the head-to-head matchup between MCTS players, the extra computation gives the stronger player a 55% to 45% edge.

These results are significant because the MCTS algorithm requires no game-specific knowledge. All the agent needs is the encoding of the rules. There is no explicit understanding that scoring points is good or that an agent should consider possible counter moves of its opponent when deciding what to play. But the emergent behavior is consistent with those elements of intelligence.

The results for Battleship, Racko, and GOPS are shown in Table 5.2. We have not yet implemented greedy HEF for these games. The high variation in the outcome of GOPS (due to luck) reduces the variability in the results for the different agents. But overall, the results are generally consistent with the results for DIFFERENCE. The MCTS algorithm produces results that are significantly better than RANDOM. And allowing MCTS to take more samples leads to better outcomes, albeit with an expected diminishing return (i.e., 10 times the number of samples does not mean 10 times the performance).

These results are encouraging because they represent the first online general game player for partially observable games. It is difficult to assess the true strength of the agents, because we are only comparing other agents of our own creation. Still, we can see from the game logs that the behavior of the MCTS agents is consistent with what we might consider intelligence. For example, when the Battleship agent registers a hit on its opponent, it fires in adjacent squares, as it should.

The MCTS results will serve as a baseline for future researchers to compare against.

Battleship	MCTS-100	MCTS-1000
Random	16-184	12-188
MCTS-100		63-137

Racko	MCTS-100	MCTS-1000
Random	1-199	3-197
MCTS-100		82-118

GOPS	MCTS-100	MCTS-1000
Random	74-121	80-119
MCTS-100		99-99

Table 5.2: Results for SDPOG in Battleship, Racko, and GOPS.

5.4 Games with Pure Strategy Equilibria

Theorem 3 shows that for games with a dominant solution, the estimates of the probability that a particular member of the information set is the true current state are immaterial. However, in other circumstances, these probability estimates are crucial to competent decision making. As a special case, we next consider the class of games for which there is a pure strategy equilibrium.

Since these values are not known, we use an opponent model $\hat{\sigma}_{\text{-}k}$ to estimate them. Because the tree is large, we assume that $\hat{\sigma}_{\text{-}k}$ is not given in closed form. Rather, we assume that these probabilities are computed "online" as they are required. Because one of our ways of viewing information sets is as a set of possible game histories, we can simulate each possible game history up to each decision point of our opponent. By simulating the moves in a prefix $a_{1:t'}$ of a possible game history $a_{1:t}$, we are not only able to recreate what the true state of the world $S(a_{1:t'})$ would have been, we are also able to determine what $P_{\text{-}k}$'s observations $S(a_{1:t'}).o_{1:t'}^{(\text{-}k)}$ would have been at that point. Using the information set generation subroutine, we can then determine $I(o_{1:t'}^{(\text{-}k)})$: the set of all worlds (or game histories) that $P_{\text{-}k}$ would have believed were possible at $S(a_{1:t'})$.

The nature of $P_{\text{-}k}$'s decision at the simulated point along a possible game history is the same as the decision that P_k faces in the present: to make a decision about the future based on a set of observations. Given that the agents are computationally bounded, it is not feasible for P_k to use tree search to simulate $P_{\text{-}k}$ exactly as though P_k itself were making the decision at that point (i.e., by determining what it (P_k) what would do if it were in $P_{\text{-}k}$'s situation and then assuming that

that is what $P_{\text{-}k}$ would do), because that would lead to an infinite amount of nesting of opponent modeling. Instead, we assume that P_k models $P_{\text{-}k}$'s decisions as being based on fundamentally less computation than P_k's.

As a first attempt at opponent modeling, suppose that $\hat{\sigma}_{\text{-}k}$ is evaluated by assuming that $P_{\text{-}k}$ applies SDPOG with $d = 1$ and an HEF $\hat{\phi}$. P_k uses $\hat{\sigma}_{\text{-}k}$ to estimate the probability $P(a_{1:t})$ for each member of the information set. We call this DOMIS (Decision-making through Opponent Modeling and Information Sets). This algorithm applies limited local search and optimization to produce a deterministic move at each decision point. If the algorithm is applied at each of P_k's decision points in the game, the global strategy induced is $\sigma_k[DOMIS(\hat{\phi}, \hat{\sigma}_{\text{-}k})]$.

Theorem 4. *Under opponent model $\sigma_{\text{-}k}[SDPOG(\hat{\phi}, d = 1)]$, there exists $\hat{\phi} = \phi_{\sigma^*}$ for which $\hat{\sigma}_k[DOMIS(\hat{\phi}, \hat{\sigma}_{\text{-}k})]$ is consistent with $\hat{\sigma}_{\text{-}k}$ iff the game has a pure strategy equilibrium solution.*

Proof. ($\Leftarrow$) First, we show that if the game has a pure strategy equilibrium $\sigma^* = \langle \sigma_k^*, \sigma_{\text{-}k}^* \rangle$, then there is an HEF that can be used with DOMIS to produce a consistent strategy. Let $\phi_{\sigma^*}(u)$ be the expected value of the game if u is reached and if both players play according to σ^* thereafter. Let $\hat{\phi} = \phi_{\sigma^*}$. The deterministic decisions returned by $\hat{\sigma}_{\text{-}k}$ are those that optimize with respect to the true equilibrium values and must therefore match those in $\sigma_{\text{-}k}$. Otherwise, $\sigma_{\text{-}k}$ could be improved upon. The probabilities computed at line 6 therefore reflect the true probabilities under σ^*, and the deterministic best response computed by P_k matches σ_k.

($\Rightarrow$) We must show that if there is a ϕ that makes the strategy induced by DOMIS consistent with the opponent model it uses, then there must be a pure strategy equilibrium. Equivalently, if there is no pure strategy equilibrium, then there is no ϕ that makes the strategy induced by DOMIS consistent with the opponent model it uses. Assume initially that $P_k = P_1$. Let $\sigma^{=} \langle \sigma_1^*, \sigma_2^* \rangle$ denote the mixed equilibrium solution. By definition of Nash equilibrium, $\forall \sigma_1 \forall \sigma_2 \; U(\sigma_1, \sigma_2^*) \leq U(\sigma_1^*, \sigma_2^*) \leq U(\sigma_1^*, \sigma_2)$. Because we are requiring that σ^* be a mixed strategy, then any pure strategy, σ_2^P must *not* have the same safety guarantee as σ_2^*. That is, there is a best response strategy $B(\sigma_2^P)$ for P_1 that will strictly improve P_1's payoff:

$$U(\sigma_1, \sigma_2^*) \leq U(\sigma_1^*, \sigma_2^*) \leq U(\sigma_1^*, \sigma_2) < U(B(\sigma_2^P), \sigma_2^P)$$

DOMIS models the opponent as making deterministic decisions and computes a deterministic best response. Let $\hat{\sigma}_{\text{-}k} = \sigma_2^P$ be strategy that P_1 assumes for P_2, and $\sigma_k[DOMIS(\hat{\sigma}_{\text{-}k})] = B(\sigma_2^P)$ be the best response strategy that P_1 computes to counter σ_2^P. But note the leftmost inequality in the expression: if P_2 switches to σ_2^*, then the payoff will be strictly lower for P_1 (and thus higher for P_2) because the left-hand-side inequality holds for all σ_1, including $B(\sigma_2^P)$. Therefore $\hat{\sigma}_{\text{-}k}$ must not be consistent. Analogous reasoning applies if P_2 is the active player and P_1 is the opponent. □

In short, DOMIS cannot produce a consistent strategy to its deterministic opponent model for games that have only mixed equilibria because it is not capable of producing randomized strategies. Consistent opponent modeling techniques for games with mixed strategies require a way for agents to model probabilistic decisions of their opponent and to respond probabilistically. This unpredictability is the key to not being exploited by a clever opponent.

5.5 Games with Mixed Strategy Equilibria

In this section, we take key steps towards the development of a consistent, general algorithm for generating probabilistic point decisions in games with mixed strategy equilibria. Our method, Bounded Equilibrium through Linear Programming (BELP), shown as Algorithm 5.1 avoids the philosophical and computational complications of opponents modeling the opponent models of their opponents. The key to this algorithm is the reasoning agent's ability to generate information sets.

We begin by generating a *search tree* (line 2), which is a fragment of the full game tree. The search tree is initialized by all the nodes along the paths from the root to each member of the information set characterized by input parameter $o_{1:t}$, the current player's observations. Nodes are added to the search tree by repeatedly applying the operations outlined in the introduction to this chapter (e.g., finding successor states and generating information sets). The set of generated nodes always maintains the tree property because identifying a node $S(a_{1:t})$ in the information set $I(u.o_{1:t}^{(k)})$ of an existing node u in the tree implies that all nodes $S(a_{1:t'})$ ($t' < t$) along the path to

that node are also identified (and included in the search tree).

Once the computational budget for tree exploration has been expended, we process the generated search tree $\mathcal{T}$, according to $ESLP$, with some modification to allow for the approximation of information set values for nodes that are terminal in $\mathcal{T}$ but not in Γ. The resulting linear program includes variables for each of P_k's actions and for each of $P_{\text{-}k}$'s information sets in $\mathcal{T}$. In the linear expression to be optimized, the coefficient for the root information set is 1; all other coefficients are zero. The probability distribution for each point decision for P_k and the utilities of the information sets for $P_{\text{-}k}$ can be extracted from the solution to the primal linear program. A strategy for $P_{\text{-}k}$ and information set values for P_k can be extracted from the solution to the dual.

The constraints in the LP are of two kinds: inequality constraints corresponding to each of $P_{\text{-}k}$'s actions and equality constraints corresponding to each of P_k's information sets, both with respect to $\mathcal{T}$. The inequality constraints are used to ensure that the value of an information set does not exceed the worst-case value of an information set for any of its children. The equality constraints are used to ensure that the total probabilities assigned by P_k at any of its decision points is exactly equal to the sum of the probabilities assigned to its descendants.

For all the nodes in an information set, there is an outgoing edge corresponding to each possible choice that a player has in that set. This set of edges is collectively referred to as an action in Kuhn's definition of extensive form games. When a logical description language is used to express game rules, the same syntactic action $e.g., \texttt{bet}$ may be possible in different information sets. However, the information sets can be uniquely named with a list of observations. Thus, we can use an observation-list + action-name pair to uniquely name the coefficients in the linear program.

Each node τ in $\mathcal{T}$ is processed in a loop (lines 3-10). We use $P_{k'}$ to denote the active player at τ. The PREVACTION subroutine invoked at line 5 finds the previous action taken by P'_k in the game history specified by τ. This is used to augment a tree structure $I_{k'}$ that tracks the parent-child relationships among information sets for $P_{k'}$.

A value v is computed for each terminal in $\mathcal{T}$. If τ is a leaf node in the full game tree, this v is simply the payoff of the game at that point (the cost of the game, since the LP minimizes). Otherwise, τ represents a node on the frontier of the search tree, whose value must be approximated

with the expected value of the information set of which τ is a member, not the expected value of τ itself.

Lines 11-14 complete the construction of inequality constraints by adding information about all possible descendant information sets. The result is that the inequality constraint for (o, a) says that the cost associated with information set o is not greater than the total costs for all P_k responses that lead to terminals (in $\mathcal{T}$) plus the value of any information set descendants of o.

Lines 15-19 construct the equality constraints using the structure for P_k built at line 6.

Line 20 produces an equilibrium strategy profile σ^* by solving the linear program for the game $\mathcal{T}$, where terminals in $\mathcal{T}$ that are not terminal in Γ have been approximated by HEF ψ. σ_k^* encodes the strategy for all of P_k decision points that were discovered in the generation of $\mathcal{T}$. From this, the algorithm extracts the decision probabilities specified for the information set specified by input parameter $o_{1:t}$ and returns a choice for this current decision point according to that distribution.

As noted earlier, the LP for game trees can be both constructed and solved in polynomial time in the number of nodes in the tree. Our algorithm presents the user with a tradeoff by reducing the number of tree nodes to $|\mathcal{T}|$, which depends on the computational budget allotted to the GENERATESEARCHTREE routine, in exchange for the increased computational expense needed to approximate the information set values of nodes that are terminal in $\mathcal{T}$ but not in Γ.

Definition 12. *A node $S(a_{1:t})$ in Γ is* information set independent *if for all $a_{1:t'} \in E(a_{1:t})$ and for all $k \in \{1, \ldots, \Gamma.K\}$, the following condition holds: $a_{1:t'} \in I(S(a_{1:t'}).o_{1:t'}^{(k)}) \Rightarrow a_{1:t'} \in E(a_{1:t})$.*

Definition 13. *A node $S(a_{1:t})$ in Γ has a* sub-game dominant solution *if for all $a_{1:t'} \in E(a_{1:t})$, the following condition holds: there exists a and σ_k^* with $\sigma_k^*(S(a_{1:t'}), a) = 1$ such that for all σ_k' with $\sigma_k'(S(a_{1:t'}), a) \neq 1$ and for all $\sigma_{\text{-}k}$, $U_k(\sigma_k^*, \sigma_{\text{-}k}) \geq U_k(\sigma_k', \sigma_{\text{-}k})$.*

Definition 14. *A node $S(a_{1:t})$ in Γ is* sub-game independent *if it is information set independent or has a sub-game dominant solution.*

Theorem 5. *If* GENERATESEARCHTREE$(\cdot)$ *returns a tree $\mathcal{T}$ such that every terminal in $\mathcal{T}$ is terminal or sub-game independent in Γ, then there exists HEF ϕ^* for which* BELP*[ϕ^*] induces the Nash equilibrium strategy, σ_k^*, even if σ_k^* is mixed.*

Algorithm 5.1 Bounded Equilibrium Linear Programming

function CHOOSEMOVE(Obs $o_{1:t}$, $\hat{\psi}$)
  $\mathcal{T} \leftarrow$ GENERATESEARCHTREE($o_{1:t}$, $\hat{\psi}$)
  for $\tau \in \mathcal{T}$ **do**
    $P_{k'} \leftarrow D(\tau)$
    $(o,a)^{(k')} \leftarrow$ PREVACTION(τ, k')
    Insert $((o,a)^{(k')}, \tau.o^{(k')})$ into $\mathcal{I}_{k'}$
    if $N_{\mathcal{T}}(\tau) = \emptyset$ **then**
      $v \leftarrow U(\tau)\Pi_0(\tau)$ **if** $\tau \in T_\Gamma$ **else** $\hat{\psi}(\tau)$
      In $Ineq[(o,a)^{(\text{-}k')}]$
        Increase coeff. of $(o,a)^{(k')}$ by v
  for each ineq. const. (key $((o,a)^{(\text{-}k)})$) **do**
    Coeff. $(o,a)^{(\text{-}k)} \leftarrow -1$
    for each $x_{\text{-}k}$ s.t. $((o,a)^{(\text{-}k)}, x^{\text{-}k}) \in \mathcal{I}_{\text{-}k}$ **do**
      Coeff. $x_{\text{-}k} \leftarrow 1$
  for each eq. const. (key $o^{(k)}$) **do**
    Find $(o,a)^{(k)}$ s.t. $((o,a)^{(k)}, o^{(k)}) \in \mathcal{I}_k$
    Coeff $(o,a)^{(k)} \leftarrow -1$
    for each $a, o^{(k)'}$ s.t. $((o^{(k)}, a), o^{(k)'}) \in \mathcal{I}_k$ **do**
      Coeff of $(o^{(k)}, a) \leftarrow 1$
  Solve for σ^* according to generated constraints.
  return move acc. to $\sigma_i^*(o_{1:t})$.

Proof. The full proof requires detailed understanding of the composition of the constraints in the ESLP algorithm. Here, we only sketch the details. Let LP be the linear program that is constructed and solved by ESLP for Γ and LP' be the linear program for BELP. For every constraint in LP', there is a corresponding constraint in LP. Because the search tree $\mathcal{T}$ in BELP is only a subtree of Γ, some of the constraints in LP do not appear in LP'. And some of the variables in the constraints of LP do not appear in the corresponding constraints of LP'. Because of the sub-game independent requirement on the terminals in $\mathcal{T}$, there is a mechanical transformation from LP to LP' by iteratively removing the extra constraints and variables from LP while ensuring that the optimum solution is unchanged and that the variable assignments for the solution to LP' are the same as the variable assignments to the corresponding variable assignments in the solution for LP. □

BELP is a first step towards general reasoning for games in which bluffing and deception are key components of a winning strategy. We have not yet implemented BELP in our game-playing system.

5.6 Case Study: Point Decisions in Scrabble

We now describe the application of our opponent modeling techniques to Scrabble. In addition to a quality decision-making procedure, high-level play in Scrabble also requires an efficient move generation function. We therefore implemented our opponent modeling scheme within QUACKLE, an existing open-source Scrabble framework (and *de facto* world champion player), in order to take advantage of that heavily optimized functionality [O'Laughlin *et al.*, 2006].

Scrabble is a popular crosswords game played by millions of people worldwide. Competitors make plays by forming words on a 15 x 15 grid (see Figure 5.2), abiding by constraints similar to those found in crossword puzzles. The individual tiles are assigned a point value that is loosely related to frequency of usage in real words: common letters, including all the vowels, are worth one point each, while less frequently used letters Q and Z are worth ten points each. Each player has a rack of letter tiles that are randomly drawn from a bag that initially contains 100 tiles. The

distribution of tiles is also somewhat correlated to frequency of usage in real words: there are 12 E tiles and only one Z. Two blank tiles, denoted '?', are wild; they may be used to represent any letter.

On each turn, a player replenishes her rack to seven tiles and then places one or more tiles from her rack onto the same row or column of the board, combining with at least one tile already on the board[2] to form one or more words. All words formed must be listed in the Official Scrabble Player's Dictionary. The score for the play is the sum of the point value for each new word formed. The score for a word is the sum of the scores for the individual tiles. Premium squares on the board double or triple the letter or word on which they are played. These squares only score a bonus on the turn in which they are played upon.

Achieving a high score requires a delicate balance between maximizing one's score on the present turn and managing one's rack in order to achieve high-scoring plays in the future. Defensive considerations are also important; skilled players avoid opening positions on the board where the opponent is likely to be able to make a high-scoring play.

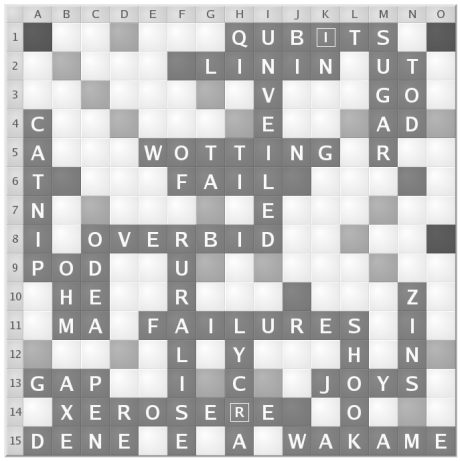

Figure 5.2: A sample Scrabble game. The shaded premium squares on the board double or triple the value of single letter or a whole word. Note the frequent use of obscure words.

Because opponent's tiles are hidden and because tiles are drawn randomly from the bag on each turn, Scrabble is a game of imperfect information. We have shown empirically that for state-of-the-art computer players, being allowed to know the opponent's letters leads to 40% more victories,

[2]The requirement for the first word played is only that it cover the center square of the board.

all other player parameters being equal [Richards and Amir, 2007].

In a real game, we would not know exactly what tiles our opponent has, but we can make some inferences based on his most recent move. Consider the game situation shown in Figure 5.3. Our opponent, playing first, held ?IIMNOO and played <8E IMINO (?O) 16>on row 8, column E for 16 points, leaving letters ?O on his rack.We can observe only the letters he played—IMINO—and the letters on our own rack GLORRTU, leaving 88 letters which we have not seen: 86 in the bag and two on his rack. When the opponent draws five letters to replenish his rack, each of the tiles in the bag is equally likely to be drawn. But it would be an oversimplification to assume that the two letters left on his rack can also be viewed as being randomly and uniformly drawn from the 88 letters that we have not seen.

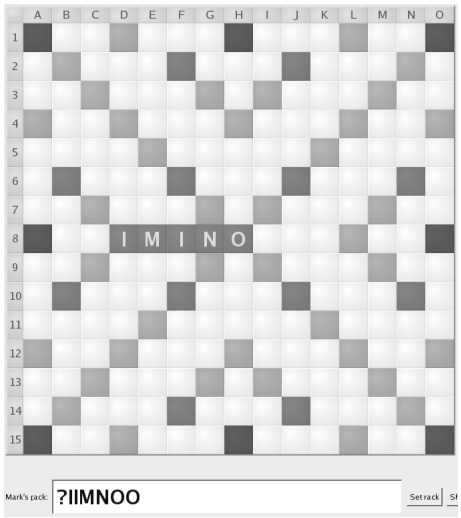

Figure 5.3: The state of the board after observing the opponent play IMINO with a leave of ?O. The active player holds GLORRTU. Between the two letters left on the opponent's rack and the letters left in the bag, there are 88 unseen tiles.

Of the 372 possible two-letter pairs for the opponent's leave (??, ?A, . . . , ?Z, AA, AB, . . . , AZ, . . . , YZ), some are considerably more probable, given the most recent play. Suppose our opponent's leave is ?H. That would mean that he held ?HIIMNO before he played. If he had held that rack, he could have played <8D HOMINId 80>, a bingo which would have earned him 64 more points than what he actually played. Were our opponent a human, we would have to account for the possibility that he does not know this word or that he simply failed to recognize the

opportunity to play it. But since our opponent is a computer, we feel confident that he would not have made this oversight and conclude that his leave after IMINO was not ?H. Likewise, we can assume that his leave was not ?L (<8D MILlION 72>).

Suppose his leave was NV. With IIMNNOV, he could have played <8G VINO (IMN) 14>. While this would have scored two fewer points than IMINO, it has a much better leave (IMN instead of NV) and would likely have been preferred. In general, we can consider each of the possible leaves that an opponent may have had (based on the set of tiles we have not seen), reconstruct what his full rack would have been in each case, generate the legal moves he could have made with that rack, and then use that information to estimate the likelihood of that leave. Using Bayes' theorem

$$P(leave \mid play) = \frac{P(play \mid leave)P(leave)}{P(play)}$$

The term $P(leave)$ is the prior probability of a particular leave. It is the probability of a particular combination of letters being randomly drawn from the set of all unseen (by us) letters. This is the implicit assumption that QUACKLE makes about the opponent's leave. The prior probability for a particular draw D from a bag B is

$$P(leave) = \frac{\prod_{\alpha \in D} \binom{B_\alpha}{D_\alpha}}{\binom{|B|}{|D|}}$$

where α is a distinct letter, B_α is the number of α-tiles in B, D_α is the number of α-tiles in D, and $|B|$ and $|D|$ are respectively the size of the bag and size of the draw.

We will be interested in computing probabilities for all of the possible leaves; we can therefore take advantage of the fact that

$$P(play) = \sum_{leave} P(play \mid leave)P(leave)$$

The $P(play \mid leave)$ term is our model of the opponent's decision-making process. If we are given to know the letters that comprise the $leave$, then we can combine those letters with the tiles that we observed our opponent play to reconstruct the full rack that our opponent had when he played that move. After generating all possible legal plays for that rack on the actual

board position, we must estimate the probability that our opponent would have chosen to make that particular play. We are assuming that our opponent is a computer, so it might be reasonable to believe that our opponent also makes his decisions based on the results of some simulations.

Unfortunately, simulating our opponent's simulations would not be practical from a computational standpoint. Instead, we naively assume that the opponent chooses the highest-ranked play according to the same static move evaluation function that we use. In other words, we assume that our opponent would make the same move that we would make if we were in his position and did not do any simulations.

This model of our opponent's decision process is admittedly overly simplistic. However, it is likely to capture the opponent's behavior in many important situations. For example, one of the things we are most interested in is whether our opponent can play a bingo (or would be able to play a bingo if we made a particular move). If a bingo move is possible, it is very likely to be the highest-ranking move according to our static move evaluator anyway.

A key advantage of modeling the opponent's decision-making process in this way is that the calculation of $P(play \mid leave)$ is straightforward. If the highest-ranked word for the corresponding whole rack matches the word that was actually played, we assign a probability of 1; otherwise, we assign a probability of 0. Let M be the set of all leaves for which $P(play \mid leave) = 1$. Then the computation simplifies to

$$P(leave \mid play) = \frac{P(leave)}{\sum_{leave \in M} P(leave)}$$

Returning to our earlier example, if the opponent plays IMINO, there are only 27 of 372 possibilities to which we assign non-zero probability. Using only the prior values for $P(leave)$, the actual leave ?O is assigned a probability of 0.003. After conditioning on $play$, that leave is assigned a 0.02 probability. In this example, there were only 372 possible leaves to consider, but in general, there could be hundreds of thousands.

It may be too expensive to run simulations for every possible leave, but we would like to consider as much of the probability mass as possible. Using the posterior probabilities, 60% of the probability mass is assigned to about 10 possible leaves. Using only the priors, the 10 most

probable leaves do not even account for 10% of the probability mass. However many samples we can afford computationally, we expect to get a much better feel for what our opponent's response to our next move might be if we bias our sampling of leaves for him to those that are most likely to occur.

During simulation, after sampling a leave according to the distribution discussed above, we randomly draw tiles from the remaining unseen letters to create a full rack. We have created an Inference Player in the QUACKLE framework that is very similar to QUACKLE's Strong Player[3]. It runs simulations to the same depth and for the same number of iterations as the Quackle player. The only difference is in how the opponent's rack is composed during simulation.

Experimental Results Table 5.3 shows the results from 630 games in which our Inference agent competed against QUACKLE's Strong Player. While there is still a great deal of variance in the results, including big wins for both players, the Inference Player scores, on average, 5.2 points per game more than the QUACKLE Strong Player and wins 18 more games. The difference is statistically significant with $p < 0.045$.

	With Inferences	**Quackle**
Wins	324	306
Mean Score	427	422
Biggest Win	279	262

Table 5.3: Summary of results for 630 games between our Inference Agent and QUACKLE's Strong Player.

The five-points-per-game advantage against a non-inferencing agent is also significant from a practical standpoint. To give the difference some context, we performed comparisons between a few pairs of strategies. The baseline strategy is the greedy algorithm: always choose the move that scores the most points on the present turn. An agent that incorporates a static leave evaluation into the ranking of each move defeats a greedy player by an average of 47 points per game.

When the same Static Player competes against QUACKLE's Strong Player, the simulating agent wins by an average of about 30 points per game. To be able to average five points more per

[3]QUACKLE is an open-source software package for word-playing games and includes multiple agents for playing Scrabble

game against such an elite player is quite a substantial improvement. In a tournament setting, where standings depend not only on wins and losses but also on point spread, the additional five points per game could make a significant difference. The improvement gained by adding opponent modeling to the simulations would seem to justify the additional computational cost. The expense of inference calculations has not been measured exactly, but it is not excessive considering the costs of simulation in general.

In addition to publication at IJCAI-07 [Richards and Amir, 2007], our work on Scrabble was also featured in New Scientist and IEEE Computer magazines and on the front page of the Daily Illini, the University of Illinois at Urbana-Champaign's daily newspaper.

CHAPTER 6

RELATED WORK

In this chapter, we provide additional details about related work that this thesis extends, complements, utilizes, or supercedes. We first give a broad overview of how this thesis work fits into the overall landscape of the state of the art in reasoning under uncertainty. In the remaining subsections, we review related work in the areas of economics and game theory, game tree search, planning under uncertainty, and representation languages for sequential reasoning problems (both planning and game playing).

6.1 State of the Art in Reasoning Under Uncertainty

This thesis touches on a variety of topics and concepts that are broadly explored in many disparate fields of research, including economics and game theory, game tree search, planning, and knowledge representation. Here, we outline some key similarities and differences of our work in comparison to these fields.

Game theory focuses on the formalization of concepts such as rationality and optimality in multi-agent decision-making environments. We focus primarily on Nash equilibrium solutions for two-player, zero-sum games, which guarantee that neither agent can improve its utility by changing its strategy, as long as the other agent does not change its strategy. A key distinction in our treatment of games compared to the game theory literature is that we do not assume that agents are computationally unbounded. Instead, we present a novel model where we assume that agents have common knowledge about the transition function (generating a successor state from a current state and action) and can determine the payoffs to all agents in a terminal state. However, agents must model their opponents and reason about their own decisions using only a bounded

number of such operations. Instead of aiming to identify a Nash equilibrium solution for the whole game before actually playing, we assume that agents have an opportunity to perform limited deliberations between each received observation. The theoretical goal in our framework is for the global strategy that would result by calling our decision procedure at every decision point to be a Nash equilibrium, were the number of operations allowed to tend to infinity.

As such, our work relies heavily on the game tree search literature. We utilize the increasingly popular Monte Carlo Tree Search method to determine which nodes to expand when performing forward search in the tree from a particular node. The key distinction in our work is our use of the information set generation operation to look *backward* in the game tree, i.e., to identify possible game histories. Relative to other systems designed to play one game, our system is notable for its ability to reason about games in a general way, without any game-specific expert domain knowledge.

Our work also has much in common with the AI planning literature. We explore the differences between AI-planning and game-playing below. Two differentiating factors of our work stand out. First, success in our framework depends on an agent's ability to identify many terminal states with varying payoffs to the players. Second, in the general case, agents need to reason about the knowledge and likely actions of their opponent. However, we do not model the knowledge of agents explicitly in terms of those predicates that can be proved true or false at a particular point in the game or in terms of a more general formula that can be proved to be satisfied. Instead, we reason in terms of information sets. We use our information set generation procedure to enable an agent to identify possible game histories that are consistent with its observations. The agent's knowledge comprises this set of sequences. Another key distinction in the way that we reason about knowledge is in our clear separation between determining what is *possible* and estimating what is *probable*.

Our game description language (POGDDL) has much in common with other languages for describing planning and game-playing environments. A key contrast with planning languages is the ability to specify observations and payoffs for each participating agent. Compared to GDL-II, another previously published language for imperfect information games, we demonstrate equivalent

theoretical expressive power but provide more computationally efficient state transition operations, as well as limited support for numerical fluents.

There are some key aspects of decision-making under uncertainty that we do not address at all. We do not consider continuous or infinite horizon decision problems. We also do not consider situations where the dynamics of the world or the agents' preferences over outcomes are not common knowledge. The Harsanyi transformation allows games with these aspects of uncertainty to be converted into extensive form games [Harsanyi, 2004]. However, an assessment of the practicality of this theoretical transformation is beyond the scope of this work.

6.2 Economics and Game Theory

A key goal of game theory, which is often studied in connection with economic theory, is to predict how rational agents will behave when they interact. We focus here on two-agent adversarial environments, where outcomes that are favorable for one agent are necessarily unfavorable for the other. Other works treat collaborative and multiplayer environments [Shoham, 2009; Rasmussen, 2006].

Predicting how rational agents will behave requires a definition of "rational." We adopt the common view that a rational agent is one that seeks to selfishly maximize its own utility (in expectation). In theory, any altruistic utility–value achieved when others experience high payoffs–can be incorporated into an agent's own utility function.

Game theory reduces multi-agent interactions to sets of *strategies* and *payoffs*. In sequential games, a strategy specifies a plan for every contingency in the game. Each player chooses a strategy and the joint selection—the strategy profile—determines the payoffs for each agent. In two-player, zero-sum games, the payoff for one player is the negation of the payoff for the other. The basics of game theory can be found in several introductory text books [Kuhn, 1997; Kuhn, 2003; Osborne and Rubinstein, 1994; Rasmussen, 2006]. Real-world applications of game theory are discussed in [Papayoanou, 2010; Rasmussen, 2006; Nisan *et al.*, 2007].

In order to address decision making mathematically, game theorists typically make concrete

assumptions about the possible strategies, outcomes, and knowledge of each agent involved in an interaction. A *solution concept* specifies a subset of the combinations of strategies of the players that satisfy some properties of interest, typically with a goal of predicting agent behavior in the game.

We focus primarily on Nash equilibrium solutions, which guarantee that no agent can improve its utility by changing its strategy, as long as all other agents do not change their strategies. Some works focus on special cases of Nash equilibria or on altogether different solution concepts. For example, a strategy profile is *Pareto optimal* if there is no other solution in which no player has a lower utility and at least one player has a strictly higher utility. In other words, Pareto optimality implies that no player's payoff can be increased without reducing another player's payoff. Pareto optimality is generally of interest when games are not zero-sum.

Koller *et al.* show how a Nash equilibrium for a non-zero sum game can be computed in polynomial time by converting the game tree to a linear complementarity problem [Koller *et al.*, 1994]. We do not treat such games in this thesis, but the information set generation and Monte Carlo Tree Search algorithms that we describe would be equally applicable to them. The decision-making algorithms in Chapter 5 would need to be adapted.

As mentioned, we address the computational limitations of real-world agents by assuming limited tree search capabilities. Gilpin and Sandholm provide an alternative approach to handling large games for a subset of EFG known as signaling games [Gilpin, 2009; Gilpin and Sandholm, 2005]. A *signaling game* is a two-player game in which CHANCE selects a game to be played according to a known distribution. P_1 is informed of the choice and makes a move. P_2 then moves, knowing P_1's choice but not CHANCE's [Shoham, 2009]. Gilpin and Sandholm show how to automatically transform such game trees into smaller games, while preserving their equilibrium properties. This approach enabled them to find exact solutions to poker games that are four times larger than games that had been solved previously. However, many games are still too large to solve exactly, and not all extensive form games satisfy the signaling game conditions.

While the Nash equilibrium is perhaps the most widely studied solution concept in the game theory literature, it is predicated on several assumptions that may not hold in practice (see Sec-

tion 2.2). A key distinction in our work is that we do not assume that agents are computationally unbounded. Even elite human players are not able to solve for a Nash equilibrium in complicated games such as chess or Poker. Their competent levels of play stem from an ability to think about the game at a high level, to abstract key features and patterns of a game state to estimate its overall utility [Li, 1994; Hsu, 2002; Chen, 2006]. They may also benefit from the suboptimal, technically irrational behavior of their opponents, who also are unable to compute the optimal solutions. Instead, humans perform some limited analysis of potential move and counter-move sequences and then consider the quality of the resulting state according to those features and patterns. Their move choices are informed by their assessments of their opponents' knowledge and tendencies. Our model of rationality (that agents are able to perform bounded tree search capabilities and seek to maximize their expected worst-case payoffs against similarly endowed opponents) is motivated by the fact that human reasoning can be approximated this way.

6.3 Game Tree Search

We use the term *game tree search* to refer to algorithms that explore a subset of the nodes in a game tree in order to determine which move to actually make at a particular decision point. Such strategies have been popular since Shannon proposed his chess-playing system in 1950 [Shannon, 1950]. The key components of a tree search algorithm are (1) given the nodes in the tree that have been visited so far, determine which node to visit (generate) next; and (2) given the nodes that have been generated, determine which move to make in the game at the current decision point. Under assumptions of rationality, common knowledge about the world dynamics, and computational unboundedness, the optimal decision can be found by backward induction in the perfect information case and by linear programming in the imperfect information case. (See Chapter 2.) Both algorithms require full traversal of the tree. We assume, by contrast, that agents can perform only a bounded number of tree expansion operations.

Our general decision-making algorithms for game trees rely extensively on Monte Carlo Tree Search Techniques [Kocsis and Szepesvári, 2006; Coulom, 2006], which we review in section 2.7.

The basic MCTS search method requires no heuristic evaluation function. Non-terminal node values are estimated by randomly sampling their terminal descendants. The number of samples allocated to each non-terminal is based on recursive k-armed bandit strategies, which balance exploration and exploitation in the tree [Audibert *et al.*, 2009; Berry and Fristedt, 1985; Hardwick and Stout, 1991; Gittins, 1989; Holland, 1992]. Node value estimates converge to their true minimax values in perfect information game trees, as the number of samples approaches infinity. MCTS strategies have been applied to many games, including backgammon [Lishout *et al.*, 2009], Go [Gelly and Wang, 2006], and poker [Van den Broeck *et al.*, 2009].

In this section, we discuss related search techniques for both perfect and imperfect information games.

6.3.1 Alpha-Beta Pruning

Computing the MINIMAX value of the root node of a game tree using backward induction requires the traversal of the whole tree. The alpha-beta pruning algorithm reduces the number of nodes that must be visited to determine the MINIMAX value (and the corresponding strategy profile that achieves that outcome) by pruning (ignoring) those parts of the tree which are provably suboptimal for either of the players [Knuth and Moore, 1975; Huntbach and Burton, 1988]. The number of nodes traversed is minimized when the best child of each node is examined first. In the best case, the number of nodes searched is approximately twice the square root of the number of nodes in the tree [Slagle and Dixon, 1969]. Pearl showed that the number of terminal nodes reached by alpha-beta pruning is asymptotically optimal [Pearl, 1982]. Alpha-beta pruning is typically used in conjunction with heuristic evaluation functions, to limit the depth of the search.

Numerous extensions to the standard alpha-beta pruning algorithm have been proposed. Many of these have specific utility in the game of chess, which has been widely studied in the AI community. One such adaptation is *singular extensions*, which increase the search depth at positions deemed to be volatile [Anantharaman *et al.*, 1990; Beal, 1990]. For example, in a chess position with many pieces attacking the same square, the search engine would extend the search to see how capture-and-recapture sequences would likely play out before applying the heuristic evaluation.

The SCOUT algorithm [Pearl *et al.*, 1980; Reinefeld, 1983], invented by Judea Pearl, consists of two procedures, EVAL and TEST. The EVAL method returns the minimax value of a node and is called along the principal variation (first child) of each node. Then the TEST procedure speculatively searches for a better value among the siblings, searching only as long as it is *possible* for the optimal value to be along that branch. If the TEST procedure indicates that it is possible, then the EVAL procedure is called on that node to determine the exact value. The result is that while some nodes are visited more than once, the total amount of search tends to be reduced.

A search agent using the *null move heuristic* attempts to deepen its search by allowing one player to take two consecutive turns [Adelson-Velskiy *et al.*, 1988; David-Tabibi and Netanyahu, 2002]. The idea is that a player's move is likely to be less desirable if allowing an additional free move does not leave the player in a significantly stronger position. Assessing the applicability of the null move heuristic requires game-specific knowledge and is not considered here.

Many of these search control heuristics could be adapted for use in our general game-playing system, provided that suitable heuristic evaluation functions can be automatically learned. Efforts to perform such learning have been moderately successful in perfect information games [Kuhlmann *et al.*, 2006; Schiffel and Thielscher, 2007; Boyan and Moore, 1996]. Their approaches could be adapted to the imperfect information case, but it is not clear whether heuristic functions should be applied to individual nodes, information sets, or both.

6.3.2 Systems Targeted to One Perfect Information Game

Numerous tree search-based AI agents have been developed that demonstrate mastery of a single specific game environment [Schaeffer, 2000; Schaeffer and van den Herik, 2002]. The game of checkers has been completely solved; Schaeffer *et al.* showed that perfect play leads to a draw [Schaeffer *et al.*, 2007]. Adaptations of MCTS have been effective in producing agents for the game of Go that are competitive with mid-level professional human players [Gelly and Wang, 2006; Lee *et al.*, 2009]. IBM's DEEPBLUE defeated human champion Garry Kasparov in chess in 1997 [Campbell *et al.*, 2002; Hsu, 2002]. Tesauro's TDGAMMON won a match against the human backgammon champion [Tesauro, 1995]. All of these systems required and exploited extensive,

game-specific human expertise. For example, while the heuristic evaluation functions used for chess and backgammon are carefully tuned through extensive computer experiments, the features employed in those functions are suggested by expert human players. Our system is unique in that the same data structures and search algorithms can be used for any game, given only a description of the rules.

6.3.3 General Tree Search in Perfect Information Games

Finnsson and Bjornsson won the AAAI General Game Playing competition in 2007 and 2008 by applying MCTS to games described in GDL [Finnsson, 2007; Finnsson and Bjornsson, 2008]. Much of their work focuses on parameter tuning; they use the basic MCTS algorithm described in [Kocsis and Szepesvári, 2006]. Their work does highlight the effectiveness of MCTS– which requires no game-specific knowledge– in a variety of games.

6.3.4 Tree Search in Imperfect Information Games

A key difference when searching in imperfect information game trees is that the node corresponding to the current state could be any one of the nodes in the active player's current information set. In addition to the forward search that an agent performs from a particular node in order to estimate its value, agents must also be able to perform search in the tree in order to identify nodes in the information set.

Parker *et al.* [Parker *et al.*, 2005] informally describe the challenge of generating (or sampling from) information sets by randomly simulating paths in a game tree and comparing the simulated observations to the observations actually received. They propose alternative sampling methods that, for example, match only the most recent observation received. This heuristic is fundamentally error prone, as nodes that match only on the last observation may be fundamentally different from the nodes in the information set (which, by definition, match on all observations). Their goal is to address the problem of kriegspiel, rather than game playing in general. Their inability to identify nodes in the information set by ordinary tree search from the root of the game tree is a key

motivation for our current work. Our constraint-based approach to information set search allows us to essentially reshape the search space and to use all of the observations together to improve pruning behavior.

6.3.5 Systems Targeted to One Game

AI systems that are competitive with or superior to the top human players in a single game of imperfect information exist for bridge [Ginsberg, 1996; Ginsberg, 1999], some variants of poker [Billings *et al.*, 2002; Bowling *et al.*, 2008; Davidson, 2002; Gilpin, 2009; Oliehoek, 2005; Van den Broeck *et al.*, 2009; Waugh *et al.*, 2009], and Scrabble [Richards and Amir, 2007; Sheppard, 2002]. DARKBOARD, the state-of-the-art system for kriegspiel, is an above-average player with a record of 6170-5633 on the Internet Chess Club [Paoloc, 2012]. Kriegspiel engines based on MCTS are now emerging as well [Ciancarini and Favini, 2009; Favini, 2010].

As is the case with the aforementioned single-game engines targeted to perfect information games, these systems rely heavily on extensive, game-specific human knowledge of the targeted game environment. For example, success in poker requires an understanding of the nature of bluffing and the necessity of disguising one's strategy through unpredictable play. Success in bridge requires the ability to estimate the value of one's own hand (including the partner's cards) by considering the possibilities and probabilities for how the remaining cards are distributed among the two opponents and how the sequence of tricks would play out.

We note also that the success of QUACKLE, the top Scrabble-playing program, relies on a heavily optimized move generation algorithm [Gordon, 1994; O'Laughlin *et al.*, 2006]. While the rules (and dictionary) of Scrabble could theoretically be encoded in a description language like GDL-II or POGDDL, the performance of the move generation algorithm would likely be several orders of magnitude slower than the state-of-the-art. Similarly, our parallelized information set generation system for kriegspiel is carefully optimized for that game. For example, we take advantage of the fact that a chessboard has 64 squares and modern computer architectures are optimized for some 64-bit data structures. These implementation advantages are necessarily lost in our general POGDDL-based system. The advantage of our POGDDL system is its generality.

6.4 Planning with Partial Observability

6.4.1 Planning vs. Game-Playing

Many aspects of reasoning under uncertainty are addressed in both the game-playing literature and the planning literature. Broadly speaking, the inputs to both planning and game-playing systems may include (1) a set of states; (2) a set of actions; (3) a transition function that maps state-action pairs to states; (4) an observation model or description of available sensors; (5) a specification of the initial state (or initial belief state if there is uncertainty about the initial state); and (6) a description of the goal state(s). The output is a sequence of actions to be executed to achieve the goals. In both cases, agents are required to make sequences of decisions, potentially in the presence of uncertainty about how the world is affected by the agent's actions or chance events. Our description of extensive form games includes all of these elements in some form.

Problems such as Free-cell, the 15-puzzle, and peg solitaire[1] are colloquially referred to as games and can be made to fit into our definition of extensive form games with a single player, plus CHANCE. But such problems can also be naturally expressed in a planning language such as PDDL and solved using common planning techniques. Thus, the choice to label a problem as a game or a planning problem or a control problem may depend more on the background and understanding of the researchers than on the nature of the problem itself.

One important difference to consider when framing a problem as planning or game-playing is the presence of other decision-making agents. Our view of information sets as sequences of actions is critical to the success of our general game-playing agent because it allows us to simulate possible game histories from the perspective of an opponent and then reason about how the opponent might act. This kind of opponent modeling, in general, is a key element of game-playing that receives less attention in the planning community.

The classification of a problem as planning or game playing may depend not so much on the presence of specific features but on which features of the problem present the greatest challenge.

[1] In this problem, pegs are placed in holes on a board. A single agent moves by "jumping" one peg over an adjacent peg into an empty slot, removing the peg that was jumped over from play. The goal is to end with a single peg in the middle of the board. The agent fails if multiple non-adjacent pegs remain on the board

For many AI planning problems, it can be computationally difficult to identify *any* sequence of actions that leads to a goal state (e.g., peg solitaire or Rubik's cube). In other planning problems, goal-satisfying paths may be plentiful, but the challenge is to find a sequence of actions that is *optimal* under some criterion. For example, in a logistics planning problem, an agent may be charged to minimize the amount of time or fuel needed to move resources to their desired locations. Planning problems may be complicated by the sheer number of actions required to achieve the goal. The number of possible action sequences may be infinite. Actions may often be executed in parallel and may be continuous, rather than discrete events in time. Some actions may require a longer duration than others.

By contrast, in the game-playing problems that we consider in this thesis, the number of possible sequences of actions by all agents–including CHANCE–is finite. The number of actions from the root to a terminal state is at most a few hundred. And the number of goal (i.e., "winning") states tends not to be vanishingly small in relation to the total number of terminal states. All actions are modeled as discrete, instantaneous events in time.

All of these features are critical to the applicability of Monte Carlo Tree Search, upon which our most general decision-making algorithms rely. In Poker or Scrabble, an agent might expect to identify some winning positions simply by simulating random sequences of moves. This is not always the case. In chess or kriegspiel, for example, an agent simulating random moves has an extremely low probability of producing a checkmate position for either player. The vast majority of random sequences lead to draws and therefore do not provide adequate feedback to the MCTS selection algorithm. The MCTS technique described in this thesis is likely to be effective only in those games with a high variance in the utility of terminal nodes reached by random search.

The challenge in game-playing tends not to be in identifying *possible* sequences of actions that lead to a desired outcome but in identifying a course of action that is (1) optimal given the expected behavior of other agents or (2) guarantees the best worst-case performance. (Nash equilibria satisfy both criteria under standard rationality assumptions.) Considering the motives and capabilities of other agents is therefore paramount. In particular, an agent may need to account for how its own actions are likely to be modeled and countered by its opponents.

As our focus is on sequential decision processes that can be naturally framed as extensive form games, we do not treat topics involving continuous variables or navigation of physical spaces that are common in robot motion planning or control theory. A good overview of all of these types of planning, including their relationships to game-playing, can be found in [LaValle, 2006].

Whether deemed to fall under the umbrella of planning or game playing, the type of problems targeted in this work have the following properties:

1. The sequence of decisions to be made is finite. Every decision therefore includes an element of progression towards the ultimate goal. In some cases, this feature is added artificially. For example, a kriegspiel game can be defined to end in a draw if a fixed number of consecutive moves is made without a pawn advancement or capture, as in chess. We assume that this number is small enough that it is reasonable for a decision maker to cache the sequence of decisions made.

2. The number of states and actions is large enough that it is infeasible to specify a probability distribution for actions over every contingency.

3. Sensing and acting are interleaved with deliberation. Rather than devising a fixed sequence of actions up front, the decision-making process emphasizes the selection of the next action, based on the entire sequence of observations received.

4. The initial state is known. Again, this condition can be artificially added. For example, one view of a poker game is that the start state is unknown because of the unseen cards held by opponents and possibly unseen community cards. Our view of a poker game is that the start state is fixed and known as the state where the dealer holds the whole deck of cards. The first several actions are made by CHANCE (or "the world" or "nature"); these actions randomly distribute the cards. When a player first acts, its information set includes all possible initial deals, and they are all equally probable.

5. Consequences of actions are deterministic. The modeling of stochastic effects must be framed as an agent choice followed by a chance event. For simplicity, we assume in POGDDL that at decision points for CHANCE, all legal actions are selected with equal probability. It

would be a straightforward syntactic extension to allow specification of arbitrary probability distributions over CHANCE actions.

6. Some case-by-case analysis is deemed to be necessary. Our approach is unlikely to be suitable for environments in which the solution depends on a high-level "trick". It would not be difficult to design environments that would be trivial for humans to solve and would be difficult or impossible for our system.

7. Opponents are to be modeled based on an evaluation of their possible past decisions and potential future prospects. We therefore emphasize the concept of information sets as corresponding to plausible game histories rather than emphasizing belief states as a single logical formula or probability distribution over primitive features at the current time step.

8. Identifying *possible* paths to a utility-maximizing goal state is not the key challenge. Agents can easily simulate some sequences of actions that lead to desirable outcomes. The challenge instead lies in making decisions in the presence of other decision-making agents with competing motivations.

9. Performance is measured by the expected payoff when playing against another agent or perhaps a set of other agents in a tournament setting.

6.4.2 Conformant Planning

When there is uncertainty about the effects of an agent's actions, opponents' actions, or chance probabilistic events, a planning agent should ideally identify a sequence of actions that leads to the goal in every contingency. A search for such plans that guarantee success is called *conformant planning* [Goldman and Boddy, 1996]. Conformant planning addresses the problem of acting under uncertainty where the decision maker has no sensors. From the perspective of game playing, this corresponds to a situation where an agent is seeking a "forced win," a strategy that is guaranteed to succeed regardless of what the other player does. Russell and Wolfe have explored such problems in the context of kriegspiel endgames [Russell and Wolfe, 2005;

Wolfe and Russell, 2007]. The connection to this thesis is with the games described in Section 5.3 that have dominant solutions. Games (or subgames) with only pure or mixed strategy equilibria (but no dominant strategy) cannot be solved by conformant planning techniques.

6.4.3 Planning with Knowledge and Sensing

The Planning with Knowledge and Sensing (PKS) planner uses principles of modal logic to frame planning problems explicitly in terms of an agent's knowledge [Bacchus and Petrick, 1998; Petrick and Bacchus, 2002; Petrick and Bacchus, 2004]. Traditional planning approaches have framed actions in terms of their preconditions and effects on the state of the world. Petrick and Bacchus note that in uncertain environments, a planner must design a sequence of actions based only on its knowledge (observations) rather than on the (unknown) true state of the world. In particular, the plan must be constructed based on the planner's understanding of how actions will change its knowledge and on what knowledge the agent will actually have during the plan's *execution*.

Instead of using a single database (i.e., of propositions which may become true or false, depending on the actions executed), PKS maintains a collection of databases, each maintaining a different type of knowledge. Actions are defined in terms of how they modify each of these databases.

The first database K_f contains positive and negative literals. For each literal $l \in K_f$, the agent knows the truth value: $K(l)$. The second database K_w contains a collection of formulas ϕ. For each ϕ, the agent knows that it is true or knows that it is false: $K(\phi) \vee K(\neg\phi)$. The third database K_v stores information about function values that the agent will *come to know* at execution time. K_w and K_v are used in the modeling of sensing actions that an agent may choose to perform. K_w is used in the case of truth values and K_v in the case of function values. K_x contains information about knowledge of "exclusive or" relationships among literals, such as the knowledge that exactly one of l_1 and l_2 are true.

The agent's knowledge base KB comprises the information in these four databases. Certain types of knowledge cannot be modeled in this paradigm. For example, general disjunctions such as $K(P(a) \vee Q(b))$ are not allowed. Our game-playing system targets domains where an agent's knowledge might naturally include such disjunctions (e.g., that a player holds either or both of a

pair of cards).

An example action for the BOMB IN THE TOILET domain would be

Action	Precondition	Effects
$dunk(x)$	$K(\neg clogged)$	$add(K_f, disarmed(x))$
		$add(K_f, clogged)$
$flush$		$add(K_f, \neg clogged)$

In this domain, one of p packages contains a bomb. Dunking a package in the toilet has the effect of disarming the bomb and clogging the toilet. Flushing the toilet unclogs it. The goal is to ensure that the bomb is defused. A traditional encoding of this domain would have a proposition x_i for each package that is true iff the ith package contains an armed bomb. The FLUSH action would encode the deleting of x_i as its effect, which would do nothing if x_i were already FALSE.

In devising a plan, traditional algorithms would end up keeping track of the different possibilities with respect to which of the packages contains the bomb. PKS is able to reason that the only way to know that the bomb is disarmed is to dunk all the packages, flushing in between each dunk. Thus, it is able to avoid the paralysis that other planners encounter when the number of possible worlds is large.

The authors of PKS acknowledge that "plans that could be discovered by reasoning at the level of possible worlds might not be found" this approach. Domains that do require such reasoning are the target of our approach in this work.

Another key distinction of our work is the ability of our POGDDL agent to implicitly reason about knowledge that would be cumbersome to express explicitly in the language of a PKS knowledge base. Our agent's "knowledge" is simply a set of possible sequences of actions. If that set is too large to enumerate, we have a straightforward method for sampling from it.

A related effort has focused on converting domain specifications in which actions are defined in terms of the effects that they have on the world to specifications in which actions are defined based on their effects on an agent's knowledge [Petrick and Levesque, 2002].

6.4.4 Logical Filtering

Somewhat related to planning under uncertainty is logical filtering. Amir and Russell [Amir and Russell, 2005] show how an agent can update a belief state based on observations expressed in logical form. They encode belief states and observations as propositional formulas. They prove that, in general, it is not possible to maintain a compact belief state as the number of observations increases, but they identify some special cases where the problem is tractable. Nance and Vogel apply logical filtering to state estimation in kriegspiel [Nance *et al.*, 2006]. However, to maintain compactness they must assume that the type of piece moved on each turn is known, which is a major simplification of the problem.

Approximate algorithms for logical filtering may drop some terms from the full belief state formula in order to maintain compactness. However, it is possible that inferences for future observations that could have naturally reduced the complexity of the complete belief state formula may no longer be applicable after the simplifying approximation. In our information set generation approach, the system is required to permanently maintain only the sequence of observations, which is linear in the number of time steps. It is possible (and often the case) that observations at the end of the sequence can uniquely determine the actions that were taken at earlier time steps. In many such cases, we are able to make useful inferences that would not have been possible with logical filtering.

6.4.5 Planning as Satisfiability

Kautz and Selman showed how AI planning problems can be converted to instances of SAT [Kautz and Selman, 1996]. Because of the availability of optimized general-purpose SAT solvers [Eén and Sörensson, 2006], they were able to solve classical planning problems faster than existing solvers. This SATPLAN approach works by assigning a propositional variable to each state variable at each time step, and to each grounded action at each time step. The preconditions and effects of actions are encoded as Boolean constraints that must hold, depending on what actions are executed at each time step. The number of time steps is fixed at n, and the SAT problem then has a satisfying

assignment if and only if there is a plan with length (makespan) n that achieves the goal. If the SAT solver declares that the problem is unsatisfiable, then the value of n must be increased and the process repeats. Unfortunately, this approach will search in vain indefinitely if in fact there is no plan of any length that achieves the goal.

In our initial work on information set generation [Richards and Amir, 2009], we applied the SAT approach. The number of actions is fixed and known in each instance of the information set generation problem (i.e., equal to the number of observations), so there is no need to guess the value of n. The SAT formulation assumes that each game is framed in terms of pieces being moved to various locations. We showed that every game can be framed this way (though not necessarily naturally or efficiently). Generic constraints, independent of any particular game, encode the fact that a piece may be in only one location at any given time. While this approach works effectively for some games, one key challenge is that each satisfying assignment corresponds to only one node in the information set. In order to generate all of the nodes in the information set, it is necessary to iteratively add the negation of the formula corresponding to each discovered node back into to the SAT problem. Rerunning the SAT-solver on the new problem forces the solver to find another information set node if there is one.

6.5 Representation Languages

Since explicitly enumerating all of the nodes and branches in a game tree is not practical, a key element to a general game-playing solution is to identify a description language that is sufficiently expressive, compact, and intuitive for the game designer. Moreover, the language chosen must facilitate the computations that are needed to reason effectively about the game. In this section, we review several description languages for planning and game playing and discuss their advantages and disadvantages.

6.5.1 Multi-Agent Influence Diagrams

Koller and Milch have proposed a formalism called Multi-Agent Influence Diagrams (MAIDs) for expressing games of imperfect information [Koller and Milch, 2001; Koller and Milch, 2003]. They note that the matrix and extensive forms of games can, in many cases, obscure important game structures. MAIDs exploit structure that is common in many decision environments to improve the compactness of the game representation and increase the efficiency of finding Nash equilibrium solutions for the game. MAIDs combine some of the features of Bayesian Networks [Pearl, 1988] and Influence Diagrams [Howard and Matheson, 1984] .

A MAID is defined on a set of agents $\mathcal{A}$ and variables $\mathcal{V} = \mathcal{X} \cup \mathcal{D} \cup \mathcal{U}$ where $\mathcal{X}$ is a set of *chance variables*, $D = \bigcup_{a \in \mathcal{A}} D_a$ is a set of *decision variables* for each agent and $\mathcal{U} = \bigcup_{a \in \mathcal{A}} \mathcal{U}_a$ are a set of real-valued *utility variables* for each agent. These variables are arranged in a directed acyclic graph, as in a Bayesian Network. Utility variables are always leaf nodes. Conditional probability tables are specified for each interior node. The parents of a decision node are the variables whose values are known when the agent must decide on a value for that decision variable.

The probability distribution over an agent's decision nodes define that agent's *strategy* for the game, and the sum of the values of its utility variables determine its payoff. The value of each utility variable must be a deterministic function of its parents (i.e., for each combination of values of parent variables, exactly one value of each utility variable has probability 1).

The authors show how this structure can be exponentially more compact than the equivalent game in extensive or matrix form and how the Nash equilibrium solution can be computed more efficiently in many cases. The MAID representation of the game is not always more compact and in some cases is exponentially larger. The main advantage of this representation is that there is an easily derived graphical rule for identifying which of a node's parents are relevant to the decision at that point. Identifying variables that are irrelevant to a particular decision point is key to reducing computational requirements for finding game-theoretic solutions.

MAIDs are an appealing option when (1) the game can naturally be framed as a sequence of simple variables whose values are systematically disclosed over time, (2) decisions points are naturally viewed as choosing the value of a single variable, and (3) the optimal choice for a variable

does not depend on previous decisions that can have a large combination of values. The games that we have presented as examples for POGDDL do not satisfy these criteria. For example, in a poker game, the values of a chance variables could correspond to dealing a card to a hand. The probability distribution for the result of each card dealt would depend on all of the previous cards dealt. This would lead to some very large conditional probability tables. Furthermore, it is not clear how numerical values for tracking dynamically changing values like a player's running score or pile of poker chips could be handled.

MAIDs are designed to facilitate the computation of Nash equilibria. We focus in this thesis on games for which it would not be feasible to even express an explicit strategy for the full game because of the overwhelming number of contingencies. Regardless of how it is represented, a full strategy for a poker game would require a betting decision for every player for every possible deal and every possible combination of previous bets. A full strategy for Racko would require a decision about whether to draw or swap for every possible initial deal and sequence of preceding swaps made by both players. The sheer number of possibilities for which a decision must be made suggests that it is unlikely that a MAID would be able to provide a compact representation of the game.

Another constraint that MAIDs have is that the information that a player will have at decision time needs to be explicitly encoded through the *structure* of the MAID. Consider a situation in a Racko game where P_1 moves after the initial deal. The parents of that first decision node would be the ten variables corresponding to the player's ten slots. Suppose that P_1 swaps a card from slot s_{11}. Then P_2 will know the value of s_{11}. The node corresponding to P_2's first decision should have as parents the variables whose values are known. In this case, it was s_{11}. But it could have been any of the other nine slots. Therefore, we cannot simply define the parents of P_2's first decision variable as the variables representing P_1's ten slots. Conceivably, an auxiliary variable that takes on values covering all possible combinations of slots and cards could be added. This node would be a descendant of P_1's first decision node and a parent of P_2's first decision node. Such an auxiliary variable would need to be added between each pair of regular decision variables, and would depend on all variables up to that point in the topological order. The constructs provided

in POGDDL provide significantly more flexibility and expressivity in terms of what knowledge an agent has when a decision needs to be made.

6.5.2 GDL

The Game Description Language (GDL) has been used to encode the games for the annual AAAI General Game Playing Competition since 2005 [Genesereth *et al.*, 2005; Love *et al.*, 2006]. However, it can only describe games of perfect information. GDL models games with successor-state axioms in a first-order-like syntax, with a semantics that is similar to Herbrand logic [Hinrichs and Genesereth, 2006].

The theorem-proving mechanisms that are used to generate legal moves and next states rely on *negation as failure*: everything that cannot be proved true is known to be false. This proof system does not extend well to the case of imperfect information, where the rules of the game prohibit an agent from being able to prove some things true or false with certainty.

6.5.3 GALA

Koller and Pfeffer embedded the GALA system [Koller and Pfeffer, 1995; Koller and Pfeffer, 1997], which does support imperfect information games, in the PROLOG programming language. The system is built on the LP algorithms of [Koller *et al.*, 1994], which find exact Nash equilibrium solutions for games in time that is polynomial in the number of nodes in the game tree. These algorithms rely on GALA's ability to generate the full game tree and convert it to a sparse linear program. Koller and Pfeffer note that in their experiments, generating the full tree took longer than solving the resulting linear program. Unfortunately, generating the full game tree is tractable for only very small games. Because GALA generates the full game tree, identifying individual information sets is easy. The system only needs to be able to *detect* that two given nodes are in the same information set. This can be done simply by checking the corresponding information states for equality (e.g., through a lexicographical comparison of the `reveal` statement strings). This is the same way potential paths are checked in the baseline information set algorithm described in

Chapter 4. Because GALA assumes the full game tree can and will be generated, the language constructs used in the `reveal` statements are not critically important.

GALA cannot perform any reasoning about the game until after the tree has been completely traversed. Thus, for large games where complete traversal is infeasible, GALA has little utility. It has not achieved widespread use.

6.5.4 RGL

The Regular Game Language [Kaiser, 2007] is another description language for imperfect information games. Rules are described in PROLOG with a few additional special predicates. All game rules are described in terms of the manipulation of pieces and locations. Each piece is characterized by one or more specific attributes (e.g., rank, color, suit). The `reveal` statements allow a player to know a specific attribute for a particular piece at a known location. The ability to make observations about movement are conspicuously absent. This formulation was a key inspiration for our initial work on information set generation [Richards and Amir, 2009]. To our knowledge, no GGP system has been built based on RGL, so it is difficult to assess its utility.

6.5.5 GDL-II

Thielscher has proposed an extension to GDL to support games of chance and imperfect information [Thielscher, 2010]. In addition to the special predicates already included in GDL (`role, init, true, legal, does, next, terminal, goal`), GDL-II supports predicates of the form `sees(R,P),` which signifies that `R` perceives `P` in the `next` position. For example, the following expressions specify part of the observation model for kriegspiel players:

`sees(R,badMoveTryAgain)` $\Leftarrow$ `does(R,M)` $\wedge$ $\neg$`validMove(M)`

`sees(black,yourMoveNow)` $\Leftarrow$ `does(white,M)` $\wedge$ $\neg$`validMove(M)`

`sees(white,yourMoveNow)` $\Leftarrow$ `does(black,M)` $\wedge$ $\neg$`validMove(M)`

Here, when an invalid move is made, both players get the same message that indicates that an invalid move was attempted. Alternatively, when `white` makes a legal move, `black`'s only

observation is `yourMoveNow`, with the intended meaning that it is now `black`'s turn to move. Player `white` receives similar observations when `black` moves.

Another special predicate `random` denotes the role of chance. Players have common knowledge that the choices made by `random` in position `M` will be selected uniformly at random from among all `R` such that `legal(R,M)` holds.

Schiffel and Thielscher have shown how GDL-II can be fully embedded in the situation calculus to provide a sound and complete reasoning system [Schiffel and Thielscher, 2011]. This is important from a theoretical standpoint but does not really address the question of how reasoning in large games should be done in practice, particularly when information sets are large. To date, there is no implementation of a general game playing system based GDL-II.

CHAPTER 7

CONCLUSION

In this work, we have addressed several key aspects of the problem of acting in partially observable games in a general way, including representation, reasoning about player knowledge, and decision making. We formally defined the Partially Observable Game Domain Definition Language, a simple extension of PDDL, to encode partially observable games. The language allows designers to define game rules using logical constructs, including actions that specify (1) how the state of the world changes based on agent decisions and (2) observations to be given to each player. State updates in POGDDL are, in general, more computationally efficient than in other existing languages. POGDDL also provides more convenient support for numerical quantities.

We presented a novel framework for reasoning about games in terms of information sets rather than belief states. We provided a method of information set generation and sampling based on constraint satisfaction, which scales much better than previous methods. We also showed that even a naïve method of information set generation can be efficiently scaled up to thousands of processors on massively parallel machines. Because information set generation is only a subroutine in a game-playing engine and because many of the other components are also efficiently parallelizable, this suggests that game playing in partially observable games is a problem that can effectively scale to hundreds of thousands of processors or more.

We have shown how our view of information sets as sequences of actions can be used to define general opponent modeling techniques. These techniques are based on the idea that simulating possible game histories up to an opponent's previous decision point allows an agent to reconstruct the game from that opponent's perspective and reason about what that opponent would have known and likely would have chosen. We have shown how to combine these opponent modeling techniques with Monte Carlo Tree Search methods for forward search in order to produce three

algorithms for single-point decision making in partially observable games. We have provided a taxonomy of games that determines which of these algorithms is optimal, based on whether the game has a dominant, pure, or mixed equilibrium solution. As a case study, we implemented our opponent modeling techniques to the game of Scrabble. Our reasoning agent outperformed the *de facto* world champion system.

Our combination of these components is the first implementation of a general player for imperfect information games that does not require complete enumeration of the game tree. We have encoded several popular games, such as Racko and Battleship, that will provide future researchers with benchmarks for comparison, as the state of the art in general game playing continues to advance.

Important areas for future research include efficient automatic translation of games rules between POGDDL and GDL-II, automatic generation of heuristic functions for imperfect information games, and automated analysis to determine the appropriateness of the various decision-making procedures defined here.

A key motivation for our work was to relax the standard game-theoretic assumption that agents are computationally unbounded. Other standard requirements that need to be re-assessed in order to broaden the applicability of game-playing systems to more realistic environments are the assumption that there is common knowledge about the rules. Our working implementation of a general system for reasoning about knowledge and making decisions in partially observable games is a foundation upon which future work along these lines can be built.

APPENDIX A

FORMAL GRAMMAR FOR POGDDL

Additions to PDDL 2.2 that are needed for POGDDL are shown in bold.

<domain> ::= (define (domain <name>)
[<require-def>]
[<types-def>]:typing
[<constants-def>]
[<predicate-def>]
[<functions-def>]:fluents
<structure-def>*)

<require-def> ::= (:requirements <require-key>+

<types-def> ::= (:types <typed list (name)>)

<constants-def> ::= (:constants <typed list (name)>)

<predicateds-def> ::= (:predicates <formula skeleton>+)

<formula skeleton> ::= (<function-symbol> <typed list (variable)>)

<formula skeleton> ::= (<predicate> <typed list (variable)>)

<predicate> ::= <name>

<variable> ::= ?<name>

<function-symbol> ::= <name>

<functions-def> ::= :fluents(:functions <function typed list (function skeleton)>)

<structure-def> ::= <action-def>

<structure-def> ::= :durative-actions <dudrative-action-def>

<structure-def> ::= :derived-predicates <derived-def>

<**structure-def**> **::=** **:pogdl** <**gain-def**>

<typed list(x)> ::= x^*

<typed list(x)> ::= :typing x^+- <type> <typed list(x)>

<primitive-type> ::= <name>

<type> ::= (either <primitive-type$^+$)

<type> ::= <primitive-type>

<function typed list (x)> ::= x^*

<function typed list (x)> ::= :typing x^+ - <function type> <function typed list(x)>

<function type> ::= number

<action-def> ::= (:action <action-symbol>
:parameters(<typed list (variable)>)
<action-def body>)

<action-symbol> ::= <name>

<action-def body> ::= [:precondition <GD>]
[:effect <effect>]

<GD> ::= ()

<GD> ::= <atomic formula(term)>

<GD> ::= :negative-preconditions <literal(term)>

<GD> ::= (and <GD>*)

<GD> ::= :disjunctive-preconditions (or <GD>*)

<GD> ::= :disjunctive-preconditions (not <GD>)

<GD> ::= :disjunctive-preconditions (imply <GD> <GD>)

<GD> ::= :existential-preconditions (exists (<typed list(variable)>*) <GD>)

<GD> ::= :universal-preconditions (forall (<typed list(variable)>*) <GD>)

<GD> ::= :fluents <f-comp>

<f-comp> ::= (<binary-comp> <f-exp> <f-exp>)

<literal(t)> ::= <atomic formula(t)>

<literal(t)> ::= (not <atomic formula(t)>

<atomic formula(t)> ::= (<predicate> t^*)

<term> ::= <name>

<term> ::= <variable>

<f-exp> ::= <number>

<f-exp> ::= (<binary-op> <f-exp> <f-exp>)

<f-exp> ::= (- <f-exp>)

<f-exp> ::= <f-head>

<f-head> ::= (<function-symbol> <term>*)

<f-head> ::= <function-symbol>

<binary-op> ::= + | - | * | /

<binary-comp> ::= > | < | = | <= | >=

<number> ::= *integer or floats of form n.n*

<effect> ::= ()

<effect> ::= (and <c-effect>*)

<effect> ::= <c-effect>

<c-effect> ::= :conditional-effects (forall (<variable>* <efffect>)

<c-effect> ::= :conditional-effects (when <GD> <cond-effect>)

<c-effect> ::= <p-effect>

<p-effect> ::= (<assign-op> <f-head> <f-exp>)

<p-effect> ::= (not <atomic formula(term)>)

<p-effect> ::= <atomic formula(term)>)

<p-effect> ::= :fluents (<assign-op> <f-head> <f-exp>)

<p-effect> ::= :pogdl<observation>

<observation> ::= :pogdl(observe (<typed list(variable)>
(<atomic formula(term)>)*)

<gain-def> ::= :pogdl (:gain <typed list(variable)> <f-exp> <GD>)

<cond-effect> ::= (and <p-effect>*)

<cond-effect> ::= <p-effect>

<assign-op> ::= assign

<assign-op> ::= scale-up

<assign-op> ::= scale-down

<assign-op> ::= increase

<assign-op> ::= decrease

<derived-def> ::= (:derived <typed list (variable)> <GD>)

APPENDIX B

SELECTED POGDDL GAME DESCRIPTIONS

B.1 Battleship

```
(define (domain battleship)
(:requirements :strips :typing :pogddl :derived-predicates :fluents)
(:types col phase position role row ship)
(:constants
        chance p1 p2 - role
        placing shooting - phase
        all others u - other)
(:predicates
        ; encodes ship location
        (at ?p - role ?s - ship ?c - col ?r - row)
        ; true if ?p has already shot at ?c, ?r
        (guessed ?p - role ?c - col ?r - row)
        (lastshot ?p - role ?c - col ?r - row)
        (occupied ?p - role ?c - col ?r - row)
        ; used sequence placement of ships from smallest to largest
        (current ?p - role ?s - ship)
        (inphase ?p - phase)
        (last ?p - role ?s - ship)
        (next ?p1 ?p2 - role ?s1 ?s2 - ship)
        ; restricts placement of horizontal ships to adjacent cells
        (adjacentc ?c1 ?c2 - col)
        ; restricts placement of horizontal ships to vertical cells
        (adjacentr ?r1 ?r2 - row)
        (whoseturn ?p - role)
        (opponent ?r ?o - role)
        (pending ?p - role)
        (size2 ?s - ship)
        (size3 ?s - ship)
        (size4 ?s - ship)
        (size5 ?s - ship)
)
(:functions
        ; # of opponent's ships sunk
        (shipssunk ?p - role)
        ; number of hits per ship
        (nhits  ?p - role ?s - ship)
        ; # hits to sink each ship
        (maxhits ?s - ship)
        ; # of ships needed to sink to win
        (needtosink)
        ; # of hits on opponent ships
        (totalhits ?p - role)
)
```

```
; Action for placing a ship.  Players alternate placing
; ships (in a fixed order) until all 10 ships are placed
; Ships are placed in a straight line and may not overlap
; ?pc - role to place current ship
; ?pn - role to place next ship
; ?s1 - type of ship to place now
; ?s2 - type of ship to place next
; ?d - across or down
; ?c1-?c5 names of cells that ship will occupy
(:action place-across-two
        :parameters (?pc ?pn - role ?s1 ?s2 - ship
          ?r - row ?c1 ?c2 - col)
        :precondition (and
                        (whoseturn ?pc)
                        (inphase placing)
                        (current ?pc ?s1)
                        ; for determining next ships to be placed
                        (next ?pc ?pn ?s1 ?s2)
                        ; make sure ship is right length
                        (size2 ?s1)
                        ; no overlapping ships
                        (not (occupied ?pc ?c1 ?r))
                        (not (occupied ?pc ?c2 ?r))
                        ; cells must be in same row or column
                        (adjacentc ?c1 ?c2))
        :effect         (and
                        (not (whoseturn ?pc))
                        (whoseturn ?pn)
                        (not (current ?pc ?s1))
                        ; set next ship to be placed
                        (current ?pn ?s2)
                        ; prevent future ships here
                        (occupied ?pc ?c1 ?r)
                        (occupied ?pc ?c2 ?r)
                        ; records location of ship for shooting phase
                        (at ?pc ?s1 ?c1 ?r)
                        (at ?pc ?s1 ?c2 ?r)
                        (when (last ?pc ?s1) ;
                                (and
                                        (not (inphase placing))
                                        (inphase shooting)))
                        ; Opponent does NOT observe location
                        (observe (?pc (placed2a ?pc ?pn ?s1 ?s2
                                        ?r ?c1 ?c2 ))
                                 (?pn (placed2s ?pc ?pn ?s1 ?s2))))) ;
```

```
(:action place-across-three
        :parameters (?pc ?pn - role ?s1 ?s2 - ship
          ?r - row ?c1 ?c2 ?c3 - col)
        :precondition (and
                        (whoseturn ?pc)
                        (inphase placing)
                        (current ?pc ?s1)
                        ; for determining next ships to be placed
                        (next ?pc ?pn ?s1 ?s2)
                        ; make sure ship is right length
                        (size3 ?s1)
                        ; no overlapping ships
                        (not (occupied ?pc ?c1 ?r))
                        (not (occupied ?pc ?c2 ?r))
                        (not (occupied ?pc ?c3 ?r))
                        ; cells must be in same row or column
                        (adjacentc ?c1 ?c2)
                        (adjacentc ?c2 ?c3))
        :effect         (and
                        (not (whoseturn ?pc))
                        (whoseturn ?pn)
                        (not (current ?pc ?s1))
                        ; set next ship to be placed
                        (current ?pn ?s2)
                        ; prevent future ships here
                        (occupied ?pc ?c1 ?r)
                        (occupied ?pc ?c2 ?r)
                        (occupied ?pc ?c3 ?r)
                        ; records location of ship for shooting phase
                        (at ?pc ?s1 ?c1 ?r)
                        (at ?pc ?s1 ?c2 ?r)
                        (at ?pc ?s1 ?c3 ?r)
                        (when (last ?pc ?s1) ;
                                (and
                                        (not (inphase placing))
                                        (inphase shooting)))
                        ; Opponent does NOT observe location
                        (observe (?pc (placed3a ?pc ?pn ?s1 ?s2
                                        ?r ?c1 ?c2 ?c3 ))
                                 (?pn (placed3s ?pc ?pn ?s1 ?s2))))) ;
```

```
(:action place-down-two
        :parameters (?pc ?pn - role ?s1 ?s2 - ship
          ?c - col ?r1 ?r2 - row)
        :precondition (and
                        (whoseturn ?pc)
                        (inphase placing)
                        (current ?pc ?s1)
                        ; for determining next ships to be placed
                        (next ?pc ?pn ?s1 ?s2)
                        (size2 ?s1)
                        ; no overlapping ships
                        (not (occupied ?pc ?c ?r1))
                        (not (occupied ?pc ?c ?r2))
                        ; cells must be in same row or column
                        (adjacentr ?r1 ?r2))
        :effect         (and
                        (not (whoseturn ?pc))
                        (whoseturn ?pn)
                        (not (current ?pc ?s1))
                        ; set next ship to be placed
                        (current ?pn ?s2)
                        ; prevent future ships here
                        (occupied ?pc ?c ?r1)
                        (occupied ?pc ?c ?r2)
                        ; records location of ship for shooting phase
                        (at ?pc ?s1 ?c ?r1)
                        (at ?pc ?s1 ?c ?r2)
                        (when (last ?pc ?s1) ;
                                (and
                                        (not (inphase placing))
                                        (inphase shooting)))
                        ; Opponent does NOT observe location
                        (observe (?pc (placed2d ?pc ?pn ?s1 ?s2
                                        ?c ?r1 ?r2))
                                 (?pn (placed2s ?pc ?pn ?s1 ?s2))))) ;
```

```
(:action place-down-three
        :parameters (?pc ?pn - role ?s1 ?s2 - ship
          ?c - col ?r1 ?r2 ?r3 - row)
        :precondition (and
                        (whoseturn ?pc)
                        (inphase placing)
                        (current ?pc ?s1)
                        ; for determining next ships to be placed
                        (next ?pc ?pn ?s1 ?s2)
                        (size3 ?s1)
                        ; no overlapping ships
                        (not (occupied ?pc ?c ?r1))
                        (not (occupied ?pc ?c ?r2))
                        (not (occupied ?pc ?c ?r3))
                        ; cells must be in same row or column
                        (adjacentr ?r1 ?r2)
                        (adjacentr ?r2 ?r3))
        :effect         (and
                        (not (whoseturn ?pc))
                        (whoseturn ?pn)
                        (not (current ?pc ?s1))
                        ; set next ship to be placed
                        (current ?pn ?s2)
                        ; prevent future ships here
                        (occupied ?pc ?c ?r1)
                        (occupied ?pc ?c ?r2)
                        (occupied ?pc ?c ?r3)
                        ; records location of ship for shooting phase
                        (at ?pc ?s1 ?c ?r1)
                        (at ?pc ?s1 ?c ?r2)
                        (at ?pc ?s1 ?c ?r3)
                        (when (last ?pc ?s1) ;
                                (and
                                        (not (inphase placing))
                                        (inphase shooting)))
                        ; Opponent does NOT observe location
                        (observe (?pc (placed3d ?pc ?pn ?s1 ?s2
                                        ?c ?r1 ?r2 ?r3))
                                 (?pn (placed3s ?pc ?pn ?s1 ?s2))))) ;
```

```
; Players alternate 'firing' at a single cell
; Different observations for misses and hits
; Both players 'receive' the same message
(:action shoot
        :parameters (?p ?o - role ?c - col ?r - row)
        :precondition (and
                        (inphase shooting)
                        (whoseturn ?p)
                        (opponent ?p ?o)
                        (not (guessed ?p ?c ?r)))
        :effect (and
                        (not (whoseturn ?p))
                        (whoseturn chance)
                        (pending ?o)
                        (observe (all (shotat ?p ?c ?r)))
                        (guessed ?p ?c ?r);
                        (lastshot ?p ?c ?r))) ; Can't guess same cell twice
(:action declare-miss
        :parameters (?p ?o - role ?c - col ?r - row)
        :precondition (and
                        (whoseturn chance)
                        (opponent ?p ?o)
                        (pending ?o)
                        (lastshot ?p ?c ?r)
                        (not (occupied ?o ?c ?r)))
        :effect (and
                        (not (whoseturn chance))
                        (not (pending ?o))
                        (not (lastshot ?p ?c ?r))
                        (whoseturn ?o)
                        (observe (all (missed ?p ?c ?r)))))
(:action declare-hit
        :parameters (?p ?o - role ?c - col ?r - row ?s - ship)
        :precondition (and
                        (whoseturn chance)
                        (opponent ?p ?o)
                        (pending ?o)
                        (lastshot ?p ?c ?r)
                        (at ?o ?s ?c ?r)
                        (occupied ?o ?c ?r)
                        (< (nhits ?o ?s) (maxhits ?s)))
        :effect (and
                        (not (whoseturn chance))
                        (not (pending ?o))
                        (not (lastshot ?p ?c ?r))
                        (whoseturn ?o)
                        (increase (nhits ?o ?s) 1)
                        (increase (totalhits ?p) 1)
                        (observe (all (hit ?p ?s)))))
```

```
(:action declare-sunk
        :parameters (?p ?o - role ?c - col ?r - row ?s - ship)
        :precondition (and
                        (whoseturn chance)
                        (opponent ?p ?o)
                        (pending ?o)
                        (lastshot ?p ?c ?r)
                        (at ?o ?s ?c ?r)
                        (occupied ?o ?c ?r)
                        (= (nhits ?o ?s) (maxhits ?s)))
        :effect (and
                        (not (whoseturn chance))
                        (not (pending ?o))
                        (not (lastshot ?p ?c ?r))
                        (whoseturn ?o)
                        (increase (shipssunk ?p) 1)
                        (increase (totalhits ?p) 1)
                        (observe (all (sunk ?p ?s)))))
; A player wins when all five of the opponent's ships are sunk
(:gain ?p - role 1
        (= (shipssunk ?p) (needtosink))
)
)
(define (problem b1)
(:domain battleship)
(:objects
        col01 col02 col03 col04 - col
        row01 row02 row03 row04 - row
        destroyer cruiser submarine battleship carrier - ship
)
(:init
        (size2 destroyer)
        (size3 cruiser)
        (size3 submarine)
        (size4 battleship)
        (size5 carrier)
        (whoseturn p1)
        (inphase placing)
        (= (shipssunk p1) 0)
        (= (shipssunk p2) 0)
        (= (totalhits p1) 0)
        (= (totalhits p2) 0)
        (= (needtosink) 2)
        (= (nhits p1 destroyer) 0)
        (= (nhits p2 destroyer) 0)
        (= (nhits p1 cruiser) 0)
        (= (nhits p2 cruiser) 0)
        (= (nhits p1 submarine) 0)
        (= (nhits p2 submarine) 0)
        (= (nhits p1 battleship) 0)
        (= (nhits p2 battleship) 0)
        (= (nhits p1 carrier) 0)
        (= (nhits p2 carrier) 0)
        (= (maxhits destroyer) 1)
        (= (maxhits cruiser) 2)
        (= (maxhits submarine) 2)
        (= (maxhits battleship) 3)
        (= (maxhits carrier) 4)
        (current p1 destroyer)
        (next p1 p2 destroyer destroyer)
        (next p2 p1 destroyer cruiser)
        (next p1 p2 cruiser cruiser)
        (next p2 p1 cruiser submarine)
        (next p1 p2 submarine submarine)
        (next p2 p1 submarine battleship)
        (next p1 p2 battleship battleship)
        (next p2 p1 battleship carrier)
        (next p1 p2 carrier carrier)
```

```
        (next p2 p1 carrier destroyer)
        (last p2 cruiser)
        (opponent p1 p2)
        (opponent p2 p1)
        (adjacentc col01 col02)
        (adjacentc col02 col03)
        (adjacentc col03 col04)
        (adjacentr row01 row02)
        (adjacentr row02 row03)
        (adjacentr row03 row04)
))
```

B.2 Racko

```
(define (domain racko)
(:requirements :strips :typing pogddl :derived-predicates)
(:types card slot role phase)
(:constants
        dealer - slot
        chance p1 p2 - role
        dealing dealtop randomdraw choosesrc drew - phase
        all others u - other)
(:predicates
        (at ?c - card ?s - slot)
        (drawn ?d ?cd - card)
        (drawing ?p - role)
        (nextdeal ?s1 ?s2 - slot)
        (lastdeal ?s - slot)
        (current ?s - slot)
        (currentdrawn ?c - card)
        (inphase ?p - phase)
        (owns ?p - role ?s - slot)
        (same ?c1 ?c2 - card)
        (successor-c ?c2 ?c1 - card)
        (successor-s ?s2 ?s1 - slot)
        (succeeds-s ?s3 ?s1 - slot)
        (succeeds-c ?c3 ?c1 - card)
        (top ?c - card)
        (whoseturn ?r - role)
        (oppof ?r ?other - role)
)
(:derived (succeeds-s ?s3 ?s1 - slot)
        (or (successor-s ?s3 ?s1)
            (exists (?s2 - slot)
                    (and
                        (successor-s ?s3 ?s2)
                        (succeeds-s ?s2 ?s1)))))
(:derived (succeeds-c ?c3 ?c1 - card)
        (or (successor-c ?c3 ?c1)
            (exists (?c2 - card)
                    (and
                        (successor-c ?c3 ?c2)
                        (succeeds-c ?c2 ?c1)))))
(:action deal
        :parameters (?dest ?next - slot ?p - role ?c - card)
        :precondition (and
                        (inphase dealing)
                        (whoseturn chance)
                        (nextdeal ?next ?dest)
                        (owns ?p ?dest)
                        (current ?dest)
                        (at ?c dealer))
        :effect        (and
                        (when (lastdeal ?dest)
                                (and
                                        (not (inphase dealing))
                                        (inphase dealtop)))
                        (not (at ?c dealer))
                        (at ?c ?dest)
                        (not (current ?dest))
                        (current ?next)
                        (observe (?p (dealt ?dest ?c))
                                (others (dealt ?dest u)))))
```

```
(:action init-top
        :parameters (?c - card)
        :precondition (and
                        (inphase dealtop)
                        (whoseturn chance)
                        (at ?c dealer))
        :effect (and
                        (not (at ?c dealer))
                        (top ?c)
                        (not (whoseturn chance))
                        (not (inphase dealtop))
                        (inphase choosesrc)
                        (whoseturn p1)
                        (observe (all (firsttop ?c)))
        ))
(:action choose-draw
        :parameters (?p - role)
        :precondition (and
                        (inphase choosesrc)
                        (whoseturn ?p))
        :effect                 (and
                                 (not (inphase choosesrc))
                                 (not (whoseturn ?p))
                                 (whoseturn chance)
                                 (drawing ?p)
                                 (inphase randomdraw)
                                 (observe (all (randomlydrawing)))
                        ))
(:action random-draw
        :parameters (?p - role ?t ?c ?cd - card)
        :precondition (and
                        (inphase randomdraw)
                        (whoseturn chance)
                        (currentdrawn ?cd)
                        (drawing ?p)
                        (top ?t)
                        (at ?c dealer))
        :effect                         (and
                                 (not (inphase randomdraw))
                                 (inphase drew)
                                 (not (drawing ?p))
                                 (not (whoseturn chance))
                                 (whoseturn ?p)
                                 (drawn ?c ?cd)
                                 (not (at ?c dealer))
                                 (not (top ?t))
                                 (observe (?p (drawncard ?c))
                                         (others (drawncard u)))
))
(:action pass
        :parameters (?p ?o - role ?d ?cd ?cdn - card)
        :precondition (and
                        (whoseturn ?p)
                        (oppof ?o ?p)
                        (currentdrawn ?cd)
                        (successor-c ?cdn ?cd)
                        (drawn ?d ?cd)
                        (inphase drew))
        :effect                 (and
                                 (not (whoseturn ?p))
                                 (whoseturn ?o)
                                 (not (currentdrawn ?cd))
                                 (currentdrawn ?cdn)
                                 (not (inphase drew))
                                 (inphase choosesrc)
                                 (not (drawn ?d ?cd))
                                 (top ?d)
                                 (observe (all (newtop ?d)))))
```

```
(:action swap-top
        :parameters (?p ?o - role ?t - card ?s - slot ?c - card)
        :precondition (and
                        (inphase choosesrc)
                        (whoseturn ?p)
                        (oppof ?o ?p)
                        (not (same ?t ?c))
                        (top ?t)
                        (owns ?p ?s)
                        (at ?c ?s))
        :effect                 (and
                                 (not (whoseturn ?p))
                                 (whoseturn ?o)
                                 (not (top ?t))
                                 (not (at ?c ?s))
                                 (top ?c)
                                 (at ?t ?s)
                                 (observe (all (swapt ?c ?t ?s)))
                                 ))
(:action swap-drawn
        :parameters (?p ?o - role ?s - slot ?c ?d ?cd ?cdn - card)
        :precondition (and
                        (inphase drew)
                        (whoseturn ?p)
                        (successor-c ?cdn ?cd)
                        (currentdrawn ?cd)
                        (drawn ?d ?cd)
                        (oppof ?o ?p)
                        (not (same ?c ?d))
                        (owns ?p ?s)
                        (at ?c ?s))
        :effect                 (and
                                 (not (whoseturn ?p))
                                 (whoseturn ?o)
                                 (not (currentdrawn ?cd))
                                 (currentdrawn ?cdn)
                                 (not (inphase drew))
                                 (inphase choosesrc)
                                 (not (drawn ?d ?cd))
                                 (not (at ?c ?s))
                                 (top ?c)
                                 (at ?d ?s)
                                 (observe (?p (swapd ?d ?c ?s))
                                         (others (swapd u ?c ?s)))
                                 ))
(:gain ?p - role 1
        (and
                (not (inphase dealing))
                (not
                        (exists (?s1 ?s2 - slot ?c1 ?c2 - card)
                                (and
                                        (owns ?p ?s1)
                                        (successor-s ?s2 ?s1)
                                        (at ?c1 ?s1)
                                        (at ?c2 ?s2)
                                        (not (succeeds-c ?c2 ?c1)))))))
)
```

```
(define (problem racko10-60)
(:domain racko)
(:objects
     u01 d01 u02 d02 u03 d03 u04 d04 u05 d05
     u06 d06 u07 d07 u08 d08 u09 d09 u10 d10  - slot
     c01 c02 c03 c04 c05 c06 c07 c08 c09 c10
     c11 c12 c13 c14 c15 c16 c17 c18 c19 c20
     c21 c22 c23 c24 c25 c26 c27 c28 c29 c30
     c31 c32 c33 c34 c35 c36 c37 c38 c39 c40
     c41 c42 c43 c44 c45 c46 c47 c48 c49 c50
     c51 c52 c53 c54 c55 c56 c57 c58 c59 c60  - card)
(:init
     (inphase dealing)
     (current u01)
     (currentdrawn c01)
     (lastdeal d10)
     (whoseturn chance)
     (oppof p1 p2)
     (oppof p2 p1)
     (at c01 dealer)
     (at c02 dealer)
     (at c03 dealer)
     (at c04 dealer)
     (at c05 dealer)
     (at c06 dealer)
     (at c07 dealer)
     (at c08 dealer)
     (at c09 dealer)
     (at c10 dealer)
     (at c11 dealer)
     (at c12 dealer)
     (at c13 dealer)
     (at c14 dealer)
     (at c15 dealer)
     (at c16 dealer)
     (at c17 dealer)
     (at c18 dealer)
     (at c19 dealer)
     (at c20 dealer)
     (at c21 dealer)
     (at c22 dealer)
     (at c23 dealer)
     (at c24 dealer)
     (at c25 dealer)
     (at c26 dealer)
     (at c27 dealer)
     (at c28 dealer)
     (at c29 dealer)
     (at c30 dealer)
     (at c31 dealer)
     (at c32 dealer)
     (at c33 dealer)
     (at c34 dealer)
     (at c35 dealer)
     (at c36 dealer)
     (at c37 dealer)
     (at c38 dealer)
     (at c39 dealer)
     (at c40 dealer)
     (at c41 dealer)
     (at c42 dealer)
     (at c43 dealer)
     (at c44 dealer)
     (at c45 dealer)
     (at c46 dealer)
     (at c47 dealer)
     (at c48 dealer)
     (at c49 dealer)
     (at c50 dealer)
     (at c51 dealer)
     (at c52 dealer)
     (at c53 dealer)
```

```
(at c54 dealer)
(at c55 dealer)
(at c56 dealer)
(at c57 dealer)
(at c58 dealer)
(at c59 dealer)
(at c60 dealer)
(nextdeal d01 u01)
(nextdeal u02 d01)
(nextdeal d02 u02)
(nextdeal u03 d02)
(nextdeal d03 u03)
(nextdeal u04 d03)
(nextdeal d04 u04)
(nextdeal u05 d04)
(nextdeal d05 u05)
(nextdeal u06 d05)
(nextdeal d06 u06)
(nextdeal u07 d06)
(nextdeal d07 u07)
(nextdeal u08 d07)
(nextdeal d08 u08)
(nextdeal u09 d08)
(nextdeal d09 u09)
(nextdeal u10 d09)
(nextdeal d10 u10)
(nextdeal dealer d10)
(owns p1 u01)
(owns p2 d01)
(owns p1 u02)
(owns p2 d02)
(owns p1 u03)
(owns p2 d03)
(owns p1 u04)
(owns p2 d04)
(owns p1 u05)
(owns p2 d05)
(owns p1 u06)
(owns p2 d06)
(owns p1 u07)
(owns p2 d07)
(owns p1 u08)
(owns p2 d08)
(owns p1 u09)
(owns p2 d09)
(owns p1 u10)
(owns p2 d10)
(same c01 c01)
(same c02 c02)
(same c03 c03)
(same c04 c04)
(same c05 c05)
(same c06 c06)
(same c07 c07)
(same c08 c08)
(same c09 c09)
(same c10 c10)
(same c11 c11)
(same c12 c12)
(same c13 c13)
(same c14 c14)
(same c15 c15)
(same c16 c16)
(same c17 c17)
(same c18 c18)
(same c19 c19)
(same c20 c20)
(same c21 c21)
(same c22 c22)
(same c23 c23)
(same c24 c24)
(same c25 c25)
```

```
(same c26 c26)
(same c27 c27)
(same c28 c28)
(same c29 c29)
(same c30 c30)
(same c31 c31)
(same c32 c32)
(same c33 c33)
(same c34 c34)
(same c35 c35)
(same c36 c36)
(same c37 c37)
(same c38 c38)
(same c39 c39)
(same c40 c40)
(same c41 c41)
(same c42 c42)
(same c43 c43)
(same c44 c44)
(same c45 c45)
(same c46 c46)
(same c47 c47)
(same c48 c48)
(same c49 c49)
(same c50 c50)
(same c51 c51)
(same c52 c52)
(same c53 c53)
(same c54 c54)
(same c55 c55)
(same c56 c56)
(same c57 c57)
(same c58 c58)
(same c59 c59)
(same c60 c60)
(successor-s u02 u01)
(successor-s d02 d01)
(successor-s u03 u02)
(successor-s d03 d02)
(successor-s u04 u03)
(successor-s d04 d03)
(successor-s u05 u04)
(successor-s d05 d04)
(successor-s u06 u05)
(successor-s d06 d05)
(successor-s u07 u06)
(successor-s d07 d06)
(successor-s u08 u07)
(successor-s d08 d07)
(successor-s u09 u08)
(successor-s d09 d08)
(successor-s u10 u09)
(successor-s d10 d09)
(successor-c c02 c01)
(successor-c c03 c02)
(successor-c c04 c03)
(successor-c c05 c04)
(successor-c c06 c05)
(successor-c c07 c06)
(successor-c c08 c07)
(successor-c c09 c08)
(successor-c c10 c09)
(successor-c c11 c10)
(successor-c c12 c11)
(successor-c c13 c12)
(successor-c c14 c13)
(successor-c c15 c14)
(successor-c c16 c15)
(successor-c c17 c16)
(successor-c c18 c17)
(successor-c c19 c18)
(successor-c c20 c19)
(successor-c c21 c20)
(successor-c c22 c21)
```

```
        (successor-c c23 c22)
        (successor-c c24 c23)
        (successor-c c25 c24)
        (successor-c c26 c25)
        (successor-c c27 c26)
        (successor-c c28 c27)
        (successor-c c29 c28)
        (successor-c c30 c29)
        (successor-c c31 c30)
        (successor-c c32 c31)
        (successor-c c33 c32)
        (successor-c c34 c33)
        (successor-c c35 c34)
        (successor-c c36 c35)
        (successor-c c37 c36)
        (successor-c c38 c37)
        (successor-c c39 c38)
        (successor-c c40 c39)
        (successor-c c41 c40)
        (successor-c c42 c41)
        (successor-c c43 c42)
        (successor-c c44 c43)
        (successor-c c45 c44)
        (successor-c c46 c45)
        (successor-c c47 c46)
        (successor-c c48 c47)
        (successor-c c49 c48)
        (successor-c c50 c49)
        (successor-c c51 c50)
        (successor-c c52 c51)
        (successor-c c53 c52)
        (successor-c c54 c53)
        (successor-c c55 c54)
        (successor-c c56 c55)
        (successor-c c57 c56)
        (successor-c c58 c57)
        (successor-c c59 c58)
        (successor-c c60 c59)
))
```

B.3 Game of Pure Strategy (GOPS)

```
(define (domain gops)
(:requirements :strips :typing :pogddl :fluents)
(:types card slot role phase other)
(:constants
        dealing announcing bidding determining - phase
        chance p1 p2 - role
        b01 dealer - slot
        all others u - other)
(:predicates
        (at ?c - card ?s - slot)
        (first ?p - role)
        (owns ?p - role ?s - slot)
        (lastdeal ?s - slot)
        (lastbid ?s - slot)
        (whoseturn ?p - role)
        (inphase ?p - phase)
        (bidby1 ?c - card)
        (bidby2 ?c - card)
        (oppof ?r ?o - role)
        (same ?c1 ?c2 - card)
          (current ?s - slot)
        (next ?s2 ?s1 - slot)
)
(:functions
        (score ?p - role)
        (value ?c - card)
)
(:action deal
        :parameters (?c - card ?d ?n - slot ?p - role)
        :precondition (and
                        (whoseturn chance)
                        (inphase dealing)
                        (at ?c dealer)
                        (owns ?p ?d)
                        (current ?d)
                        (next ?n ?d))
        :effect (and
                        (observe (?p (deal ?c ?d ?n ?p))
                                (others (deal u ?d ?n ?p)))
                        (not (at ?c dealer))
                        (at ?c ?d)
                        (not (current ?d))
                        (current ?n)
                        (when (lastdeal ?d)
                                (and
                                        (not (inphase dealing))
                                        (inphase announcing)
                                        (current b01)))))
(:action announce
        :parameters (?c - card ?s - slot)
        :precondition (and
                        (inphase announcing)
                        (whoseturn chance)
                        (at ?c ?s)
                        (current ?s))
        :effect (and
                        (not (inphase announcing))
                        (inphase bidding)
                        (not (whoseturn chance))
                        (whoseturn p1)
                        (observe (all (announce ?c ?s)))))
```

```
(:action bid1
        :parameters (?c - card ?s - slot)
        :precondition (and
                        (inphase bidding)
                        (whoseturn p1)
                        (owns p1 ?s)
                        (at ?c ?s))
        :effect (and
                        (not (whoseturn p1))
                        (observe (p1 (bid1 ?c ?s))
                                 (p2 (bid1 u ?s)))
                        (whoseturn p2)
                        (not (at ?c ?s))
                        (bidby1 ?c)))
(:action bid2
        :parameters (?c - card ?s - slot)
        :precondition (and
                        (inphase bidding)
                        (whoseturn p2)
                        (owns p2 ?s)
                        (at ?c ?s))
        :effect (and
                        (not (whoseturn p2))
                        (whoseturn chance)
                        (not (inphase bidding))
                        (inphase determining)
                        (observe (p2 (bid2 ?c ?s))
                                 (p1 (bid2 u ?s)))
                        (not (at ?c ?s))
                        (bidby2 ?c)))
(:action determine
        :parameters (?c1 ?c2 ?b - card ?s1 ?s2 - slot)
        :precondition (and
                        (whoseturn chance)
                        (inphase determining)
                        (current ?s1)
                        (next ?s2 ?s1)
                        (bidby1 ?c1)
                        (bidby2 ?c2)
                        (at ?b ?s1))
        :effect (and
                (not (inphase determining))
                (inphase announcing)
                (not (current ?s1))
                (not (bidby1 ?c1))
                (not (bidby2 ?c2))
                (when (not (lastbid ?s1))
                        (current ?s2))
                (when (> (value ?c1) (value ?c2))
                        (and
                          (increase (score p1) (value ?b))
                          (observe (all (player1wins ?c1 ?c2 ?b ?s1 ?s2)))))
                (when (< (value ?c1) (value ?c2))
                        (and
                          (increase (score p2) (value ?b))
                          (observe (all (player2wins ?c1 ?c2 ?b ?s1 ?s2)))))))
(:gain ?p - role 1
        (or
                (and
                        (first ?p)
                        (> (score p1) (score p2)))
                (and
                        (not (first ?p))
                        (> (score p2) (score p1))))
)
)
```

```
(define (problem gops10-0)
(:domain gops)
(:objects
     s11 s21 b02 s12 s22 b03 s13 s23 b04 s14
     s24 b05 s15 s25 b06 s16 s26 b07 s17 s27
     b08 s18 s28 b09 s19 s29 b10 s110 s210 - slot
     c01 c02 c03 c04 c05 c06 c07 c08 c09 c10
     c11 c12 c13 c14 c15 c16 c17 c18 c19 c20
     c21 c22 c23 c24 c25 c26 c27 c28 c29 c30 - card)
(:init
     (first p1)
     (lastdeal s210)
     (lastbid b10)
     (whoseturn chance)
     (inphase dealing)
     (current b01)
     (next b02 b01)
     (next b03 b02)
     (next b04 b03)
     (next b05 b04)
     (next b06 b05)
     (next b07 b06)
     (next b08 b07)
     (next b09 b08)
     (next b10 b09)
     (next s11 b10)
     (next s21 s11)
     (next s12 s21)
     (next s22 s12)
     (next s13 s22)
     (next s23 s13)
     (next s14 s23)
     (next s24 s14)
     (next s15 s24)
     (next s25 s15)
     (next s16 s25)
     (next s26 s16)
     (next s17 s26)
     (next s27 s17)
     (next s18 s27)
     (next s28 s18)
     (next s19 s28)
     (next s29 s19)
     (next s110 s29)
     (next s210 s110)
     (next b01 s210)
     (owns chance b01)
     (owns p1 s11)
     (owns p2 s21)
     (owns chance b02)
     (owns p1 s12)
     (owns p2 s22)
     (owns chance b03)
     (owns p1 s13)
     (owns p2 s23)
     (owns chance b04)
     (owns p1 s14)
     (owns p2 s24)
     (owns chance b05)
     (owns p1 s15)
     (owns p2 s25)
     (owns chance b06)
     (owns p1 s16)
     (owns p2 s26)
     (owns chance b07)
     (owns p1 s17)
     (owns p2 s27)
     (owns chance b08)
     (owns p1 s18)
     (owns p2 s28)
     (owns chance b09)
     (owns p1 s19)
     (owns p2 s29)
```

```
      (owns chance b10)
      (owns p1 s110)
      (owns p2 s210)
      (at c01 dealer)
      (at c02 dealer)
      (at c03 dealer)
      (at c04 dealer)
      (at c05 dealer)
      (at c06 dealer)
      (at c07 dealer)
      (at c08 dealer)
      (at c09 dealer)
      (at c10 dealer)
      (at c11 dealer)
      (at c12 dealer)
      (at c13 dealer)
      (at c14 dealer)
      (at c15 dealer)
      (at c16 dealer)
      (at c17 dealer)
      (at c18 dealer)
      (at c19 dealer)
      (at c20 dealer)
      (at c21 dealer)
      (at c22 dealer)
      (at c23 dealer)
      (at c24 dealer)
      (at c25 dealer)
      (at c26 dealer)
      (at c27 dealer)
      (at c28 dealer)
      (at c29 dealer)
      (at c30 dealer)
      (= (value c01) 1)
      (= (value c02) 2)
      (= (value c03) 3)
      (= (value c04) 4)
      (= (value c05) 5)
      (= (value c06) 6)
      (= (value c07) 7)
      (= (value c08) 8)
      (= (value c09) 9)
      (= (value c10) 10)
      (= (value c11) 11)
      (= (value c12) 12)
      (= (value c13) 13)
      (= (value c14) 14)
      (= (value c15) 15)
      (= (value c16) 16)
      (= (value c17) 17)
      (= (value c18) 18)
      (= (value c19) 19)
      (= (value c20) 20)
      (= (value c21) 21)
      (= (value c22) 22)
      (= (value c23) 23)
      (= (value c24) 24)
      (= (value c25) 25)
      (= (value c26) 26)
      (= (value c27) 27)
      (= (value c28) 28)
      (= (value c29) 29)
      (= (value c30) 30)
      (= (score p1) 0)
      (= (score p2) 0)
))
```

B.4 Difference Game

```
(define (domain difference)
(:requirements :strips :typing :pogddl :fluents)
(:types card slot role phase)
(:constants
        chance p1 p2 - role
        drawing playing - phase
        all others u - other)
(:predicates
        (at ?c - card ?s - slot)
        (center ?c - card)
        (dealer ?c - card)
        (owns ?p - role ?s - slot)
        (inphase ?p - phase)
        (whoseturn ?r - role)
        (oppof ?r ?other - role)
        (pending ?p - role)
        (empty ?s - slot)
        (first ?r - role)
)
(:functions
        (score ?p - role)
        (value ?c - card)
        (diff ?c1 ?c2 - card)
)
(:action draw
:parameters (?p ?o - role ?s - slot ?c - card)
:precondition (and
                (inphase drawing)
                (whoseturn chance)
                (pending ?p)
                (oppof ?p ?o)
                (owns ?p ?s)
                (dealer ?c)
                (empty ?s))
:effect (and
                (not (inphase drawing))
                (not (whoseturn chance))
                (not (pending ?p))
                (not (dealer ?c))
                (not (empty ?s))
                (inphase playing)
                (whoseturn ?p)
                (at ?c ?s)
                (observe (?p (drew ?p ?o ?s ?c))
                          (?o (drew ?p ?o ?s u)))))
(:action play
:parameters (?p ?o - role ?s - slot ?c ?q - card)
:precondition (and
                (whoseturn ?p)
                (oppof ?p ?o)
                (inphase playing)
                (owns ?p ?s)
                (at ?c ?s)
                (center ?q))
:effect (and
                (not (whoseturn ?p))
                (not (inphase playing))
                (not (center ?q))
                (not (at ?c ?s))
                (whoseturn chance)
                (inphase drawing)
                (pending ?o)
                (empty ?s)
                (center ?c)
                (observe (all (played ?p ?o ?s ?c ?q)))
                (increase (score ?p) (diff ?c ?q))))
```

```
(:gain ?p - role 1
        (and
                (or
                        (and
                                (first ?p)
                                (> (score p1) (score p2)))
                        (and
                                (not (first ?p))
                                (>=  (score p2) (score p1)))))
))
(define (problem difference-9)
(:domain difference)
(:objects
        s11 s12 s21 s22 - slot
        c01 c02 c03 c04 c05 c06 c07 c08 c09 - card
)
(:init
        (oppof p1 p2)
        (oppof p2 p1)
        (whoseturn chance)
        (inphase drawing)
        (pending p1)
        (first p1)
        (empty s12)
        (empty s22)
        (owns p1 s11)
        (owns p1 s12)
        (owns p2 s21)
        (owns p2 s22)
        (dealer c01)
        (dealer c02)
        (dealer c03)
        (dealer c04)
        (dealer c05)
        (dealer c06)
        (at c07 s11)
        (at c08 s21)
        (center c09)
        (= (score p1) 0)
        (= (score p2) 0)
        (= (diff c01 c02) 1)
        (= (diff c02 c01) 1)
        (= (diff c01 c03) 2)
        (= (diff c03 c01) 2)
        (= (diff c01 c04) 3)
        (= (diff c04 c01) 3)
        (= (diff c01 c05) 4)
        (= (diff c05 c01) 4)
        (= (diff c01 c06) 5)
        (= (diff c06 c01) 5)
        (= (diff c01 c07) 6)
        (= (diff c07 c01) 6)
        (= (diff c01 c08) 7)
        (= (diff c08 c01) 7)
        (= (diff c01 c09) 8)
        (= (diff c09 c01) 8)
        (= (diff c02 c03) 1)
        (= (diff c03 c02) 1)
        (= (diff c02 c04) 2)
        (= (diff c04 c02) 2)
        (= (diff c02 c05) 3)
        (= (diff c05 c02) 3)
        (= (diff c02 c06) 4)
        (= (diff c06 c02) 4)
        (= (diff c02 c07) 5)
        (= (diff c07 c02) 5)
        (= (diff c02 c08) 6)
        (= (diff c08 c02) 6)
        (= (diff c02 c09) 7)
        (= (diff c09 c02) 7)
        (= (diff c03 c04) 1)
```

```
        (= (diff c04 c03) 1)
        (= (diff c03 c05) 2)
        (= (diff c05 c03) 2)
        (= (diff c03 c06) 3)
        (= (diff c06 c03) 3)
        (= (diff c03 c07) 4)
        (= (diff c07 c03) 4)
        (= (diff c03 c08) 5)
        (= (diff c08 c03) 5)
        (= (diff c03 c09) 6)
        (= (diff c09 c03) 6)
        (= (diff c04 c05) 1)
        (= (diff c05 c04) 1)
        (= (diff c04 c06) 2)
        (= (diff c06 c04) 2)
        (= (diff c04 c07) 3)
        (= (diff c07 c04) 3)
        (= (diff c04 c08) 4)
        (= (diff c08 c04) 4)
        (= (diff c04 c09) 5)
        (= (diff c09 c04) 5)
        (= (diff c05 c06) 1)
        (= (diff c06 c05) 1)
        (= (diff c05 c07) 2)
        (= (diff c07 c05) 2)
        (= (diff c05 c08) 3)
        (= (diff c08 c05) 3)
        (= (diff c05 c09) 4)
        (= (diff c09 c05) 4)
        (= (diff c06 c07) 1)
        (= (diff c07 c06) 1)
        (= (diff c06 c08) 2)
        (= (diff c08 c06) 2)
        (= (diff c06 c09) 3)
        (= (diff c09 c06) 3)
        (= (diff c07 c08) 1)
        (= (diff c08 c07) 1)
        (= (diff c07 c09) 2)
        (= (diff c09 c07) 2)
        (= (diff c08 c09) 1)
        (= (diff c09 c08) 1)
))
```

REFERENCES

[Adelson-Velskiy *et al.*, 1988] G. M. Adelson-Velskiy, V. L. Arlazarov, and M. V. Donskoy. Some methods of controlling the tree search in chess programs. In David Levy, editor, *Computer chess compendium*, pages 129–135. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[Amir and Russell, 2005] Eyal Amir and Stuart Russell. Logical filtering. Technical Report UIUCDCS-R-2005-2500, University of Illinois at Urbana-Champaign, 2005.

[Anantharaman *et al.*, 1990] Thomas Anantharaman, Murray S. Campbell, and Feng-hsiung Hsu. Singular extensions: adding selectivity to brute-force searching. *Artif. Intell.*, 43(1):99–109, 1990.

[Audibert *et al.*, 2009] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. Exploration-exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19):1876–1902, 2009.

[Auer *et al.*, 2002] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[Bacchus and Petrick, 1998] Fahiem Bacchus and Ron Petrick. Modeling an agent's incomplete knowledge during planning and execution. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pages 432–443, San Francisco, CA, June 1998. Morgan Kaufmann Publishers.

[Beal, 1990] Don F. Beal. A generalised quiescence search algorithm. *Artif. Intell.*, 43(1):85–98, 1990.

[Berry and Fristedt, 1985] Donald A. Berry and Bert Fristedt. *Bandit Problems*. Chapman and Hall, London, England, 1985.

[Billings *et al.*, 2002] Darse Billings, Aaron Davidson, Jonathan Schaeffer, and Duane Szafron. The challenge of poker. *Artificial Intelligence.*, 134:201–240, 2002.

[Blair *et al.*, 1996] Jean R. S. Blair, David Mutchler, and Michael van Lent. Perfect recall and pruning in games with imperfect information. *Computational Intelligence*, 12:131–154, 1996.

[Blum and Furst, 1995] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *IJCAI-95: International Joint Conference on Artificial Intelligence*, pages 1636–1642, 1995.

[Blum and Furst, 1997] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence.*, 90:281–300, 1997.

[Bowling *et al.*, 2008] Michael Bowling, Michael Johanson, Neil Burch, and Duane Szafron. Strategy evaluation in extensive games with importance sampling. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 72–79, New York, NY, USA, 2008. ACM.

[Boyan and Moore, 1996] Justin A. Boyan and Andrew W. Moore. Learning evaluation functions for large acyclic domains. In *ICML*, pages 63–70, 1996.

[Campbell *et al.*, 2002] Murray Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu. Deep Blue. *Artificial Intelligence.*, 134:57–83, 2002.

[Chen, 2006] Bill Chen. *The Mathematics of Poker*. Conjelco, Bethesda, MD, 2006.

[Ciancarini and Favini, 2009] Paolo Ciancarini and Gian Piero Favini. Monte carlo tree search techniques in the game of kriegspiel. In *IJCAI-09: International Joint Conference on Artificial Intelligence*, pages 474–479, Pasadena, CA, July 2009.

[Cormen *et al.*, 2009] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, third edition, 2009.

[Coulom, 2006] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *In: Proceedings Computers and Games 2006*, pages 72–83. Springer-Verlag, 2006.

[David-Tabibi and Netanyahu, 2002] Omid David-Tabibi and Nathan S. Netanyahu. Verified null-move pruning. *ICGA Journal*, 25(3):153–161, 2002.

[Davidson, 2002] A. Davidson. Opponent modeling in poker: Learning and acting in a hostile environment, 2002.

[Edelkamp and Hoffmann, 2004] Stefan Edelkamp and Jorg Hoffmann. PDDL – the Planning Domain Definition Language – version 1.2. Technical Report TR-195, Albert-Ludwigs-Universitat Freiburg, Institut fur Informatik, 2004.

[Edelkamp and Kissmann, 2007] Stefan Edelkamp and Peter Kissmann. Symbolic exploration for general game playing in pddl. In *Workshop on Planning and Games at the International Conference on Automated Planning and Scheduling*, 2007.

[Eén and Sörensson, 2006] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.

[Fagin *et al.*, 1995] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, MA, USA, 1995.

[Favini, 2010] Gian-Piero Favini. *The dark side of the board: advances in chess kriegspiel*. PhD thesis, University of Bologna, 2010.

[Felten and Otto, 1988] Edward W. Felten and S. W. Otto. A highly parallel chess program. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1001–1009, 1988.

[Ferguson and Korf, 1988] C. Ferguson and R. E. Korf. Distributed tree search and its application to alpha-beta pruning. In *AAAI-88*, pages 128–132, 1988.

[Finnsson and Bjornsson, 2008] Hilmar Finnsson and Yngvi Bjornsson. Simulation-based approach to general game playing. In *Twenty-third Conference on Artificial Intelligence (AAAI-08)*, July 2008.

[Finnsson, 2007] Hilmar Finnsson. CADIA-Player: A General Game Playing Agent. Master's thesis, Reykjavik University, 2007.

[Fox and Long, 2003] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence*, 20:61–124, 2003.

[Gelly and Wang, 2006] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: UCT for Monte-Carlo Go. in: Nips-2006: On-line trading of exploration and exploitation workshop. In *In Twentieth Annual Conference on Neural Information Processing Systems (NIPS)*, 2006.

[Genesereth *et al.*, 2005] Michael Genesereth, Nathaniel Love, and BarneyPell. General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62–72, 2005.

[Gilpin and Sandholm, 2005] Andrew Gilpin and Tuomas Sandholm. Finding equilibria in large sequential games of imperfect information. Technical Report CMU-CS-05-158, Carnegie Mellon University, 2005.

[Gilpin, 2009] Andrew Gilpin. *Algorithms for abstracting and solving imperfect information games (proposal)*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2009.

[Ginsberg, 1996] Matthew L. Ginsberg. Partition search. In *In Proceedings of AAAI-96*, pages 228–233. AAAI Press, 1996.

[Ginsberg, 1999] M. L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 584–589, 1999.

[Gittins, 1989] John C. Gittins. *Multi-armed bandit allocation indices*. Wiley-Interscience series in Systems and Optimization. John Wiley and Sons, New York, USA, 1989.

[Goldman and Boddy, 1996] R. Goldman and M. Boddy. Expressive planning and explicit knowledge. In *AIPS*, pages 110–117, 1996.

[Gordon, 1994] Steven A. Gordon. A faster scrabble move generation algorithm. *Softw. Pract. Exper.*, 24(2):219–232, February 1994.

[Hardwick and Stout, 1991] Janis Hardwick and Quentin F. Stout. Bandit strategies for ethical sequential allocation. *Computing Science and Statistics*, 23:421–424, 1991.

[Harsanyi, 2004] John C. Harsanyi. Games with incomplete information played by "bayesian" players, i-iii. *Manage. Sci.*, 50(12 Supplement):1804–1817, December 2004.

[Hinrichs and Genesereth, 2006] Timothy Hinrichs and Michael Genesereth. Herbrand logic. Technical Report LG-2006-02, Stanford University, 2006.

[Holland, 1992] John H. Holland. *Adaptation in natural and artificial systems*. MIT Press/Bradford Books, Cambridge, MA, USA, 1992.

[Howard and Matheson, 1984] Ronald A. Howard and James E. Matheson. Influence diagrams. In *Readings on the Principles and Applications of Decision Analysis*, Menlo Park, CA, 1984. Strategic Decisions Group.

[Hsu, 2002] Feng-Hsiung Hsu. *Behind Deep Blue*. Princeton University Press, 2002.

[Huntbach and Burton, 1988] Matthew M Huntbach and F Warren Burton. Alpha-beta search on virtual tree machines. *Information Sciences*, 44:3–17, 1988.

[Kaiser, 2007] David Michael Kaiser. *The Structure of Games*. PhD thesis, Florida International University, Miami, FL, 2007.

[Kalé and Krishnan, 1993] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[Kale and Zheng, 2009] Laxmikant V. Kale and Gengbin Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[Kautz and Selman, 1996] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional search, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1194–1201, Portland, OR, USA, 1996. AAAI Press/MIT Press.

[Knuth and Moore, 1975] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[Koller and Milch, 2001] Daphne Koller and Brian Milch. Multi-agent influence diagrams for representing and solving games. In *International Joint Conference on Artificial Intelligence*, pages 1027–1034, 2001.

[Koller and Milch, 2003] Daphne Koller and Brian Milch. Multi-agent influence diagrams for representing and solving games. *Games and Economic Behavior*, 45(1):181–221, 2003. Full version of paper in IJCAI '03.

[Koller and Pfeffer, 1995] Daphne Koller and Avi Pfeffer. Generating and solving imperfect information games. In *International Joint Conference on Artificial Intelligence*, pages 1185–1193, 1995.

[Koller and Pfeffer, 1997] Daphne Koller and Avi Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, 94(1-2):167–215, 1997.

[Koller *et al.*, 1994] Daphne Koller, Nimrod Megiddo, and Bernhard von Stengel. Fast algorithms for finding randomized strategies in game trees. In *Symposium on the Theory of Computation*, pages 750–759, 1994.

[Kuhlmann *et al.*, 2006] Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic heuristic contruction in a complete general game player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, July 2006.

[Kuhn, 1953] Harold W Kuhn. Extensive games and the problem of information. *Contributions to the Theory of Games*, 2:193–216, 1953.

[Kuhn, 1997] Harold W Kuhn. *Classics in Game Theory*. Princeton University Press, Princeton, NJ, 1997.

[Kuhn, 2003] Harold W Kuhn. *Lectures on the Theory of Games*. Princeton University Press, Princeton, NJ, 2003.

[Lai and Robbins, 1985] T. L. Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985.

[LaValle, 2006] Steven M. LaValle. *Planning Algorithms*. Cambridge, 2006.

[Lee *et al.*, 2009] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 2009.

[Li, 1994] David H. Li. *Chess Under Uncertainty*. Premier Publishing, Bethesda, MD, 1994.

[Lifschitz, 1986] Vladimir Lifschitz. On the semantics of STRIPS. In *Proceedings of the Workshop on Reasoning about Actions and Plans*, 1986.

[Lishout *et al.*, 2009] Francois Van Lishout, Guillaume Chaslot, and Jos W.H.M. Uiterwijk. Monte-carlo tree search in backgammon. In *In: Computer Games Workshop*, pages 175–184, 2009.

[Love *et al.*, 2006] Nathaniel Love, Timothy Hinrichs, and Michael Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford University, 2006.

[Luce and Raiffa, 1957] R Duncan Luce and Howard Raiffa. *Games and Decisions*. John Wiley & Sons, Inc., New York, USA, 1957.

[Marsland and Campbell, 1982] T Anthony Marsland and Murray Campbell. Parallel search of strongly ordered game trees. *Computing Surveys*, 14(4):533–551, 1982.

[Marsland, 1986] T Anthony Marsland. A review of game-tree pruning. *ICCA Journal*, 9(1):3–9, 1986.

[McDermott *et al.*, 1998] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl – the planning domain definition language – version 1.2. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, 1998.

[Nance *et al.*, 2006] Megan Nance, Adam Vogel, and Eyal Amir. Reasoning about partially observable actions. In *Proceedings of the Twenty-first Conference on Artificial Intelligence*, 2006.

[Nau, 1983] Dana S. Nau. Decision quality as a function of search depth on game trees. *J. ACM*, 30(4):687–708, 1983.

[Nisan *et al.*, 2007] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA, 2007.

[O'Laughlin *et al.*, 2006] John O'Laughlin, Jason Katz-Brown, and John Fultz. Quackle, 2006. [Online; accessed 5-May-2006].

[Oliehoek, 2005] F. Oliehoek. Game theory and AI: a unified approach to poker games, 2005.

[Osborne and Rubinstein, 1994] Martin J Osborne and Ariel Rubinstein. *A Course in Game Theory*. MIT Press, Cambridge, MA, 1994.

[Paoloc, 2012] Paoloc. Darkboard on the internet chess club, 2012. [Online; accessed 22-May-2012.

[Papayoanou, 2010] Paul Papayoanou. *Game Theory for Business*. Probabilistic Publishing, 2010.

[Parker *et al.*, 2005] Austin Parker, Dana S. Nau, and V. S. Subrahmanian. Game-tree search with combinatorially large belief states. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 254–259. Professional Book Center, 2005.

[Pearl *et al.*, 1980] J. Pearl, Los Angeles. School of Engineering University of California, and Cognitive Systems Laboratory Applied Science. *SCOUT: A Simple Game-searching Algorithm with Proven Optimal Properties*. UCLA-ENG-CSL-. University of California, School of Engineering and Applied Science, Cognitive Systems Laboratory, 1980.

[Pearl, 1982] Judea Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of the ACM*, 25(8):559–564, 1982.

[Pearl, 1988] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, San Francisco, CA, 1988.

[Petrick and Bacchus, 2002] Ronald P. A. Petrick and Fahiem Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, pages 212–221, Menlo Park, CA, April 2002. AAAI Press.

[Petrick and Bacchus, 2004] Ronald P. A. Petrick and Fahiem Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 2–11, Menlo Park, CA, June 2004. AAAI Press.

[Petrick and Levesque, 2002] Ronald P. A. Petrick and H. J. Levesque. Knowledge equivalence in combined action theories. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 2002.

[Rasmussen, 2006] Eric Rasmussen. *Games and Information: An Introduction to Game Theory*. Blackwell Publishers, Cambridge, MA, 4th edition, 2006.

[Reinefeld, 1983] Alexander Reinefeld. An improvement to the scout tree search algorithm. *International Chess Association Journal*, 6(4):4–14, 1983.

[Richards and Amir, 2007] Mark Richards and Eyal Amir. Opponent modeling in scrabble. In *IJCAI-07: International Joint Conference on Artificial Intelligence*, pages 1482–1487, Hyderabad, India, January 2007.

[Richards and Amir, 2009] Mark Richards and Eyal Amir. Information set sampling in general imperfect information positional games. In *IJCAI Workshop on General Intelligence in Game Playing Agents*, Pasadena, CA, 2009.

[Richards and Amir, 2012] Mark Richards and Eyal Amir. Information set generation in partially observable games. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Toronto, July 2012. AAAI Press.

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

[Russell and Wolfe, 2005] Stuart Russell and Jason Wolfe. Efficient belief-state and-or search, with application to kriegspiel. In *Proceedings of the 19th international joint conference on Artificial intelligence*, pages 278–285, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[Schaeffer and van den Herik, 2002] Jonathan Schaeffer and H. Jaap van den Herik. Games, computers, and artificial intelligence. *Artificial Intelligence.*, 134:1–7, 2002.

[Schaeffer *et al.*, 2007] Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 2007.

[Schaeffer, 1989] Jonathan Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6:90–114, 1989.

[Schaeffer, 2000] Jonathan Schaeffer. The games computers (and people) play. In *AAAI/IAAI*, 2000.

[Schiffel and Thielscher, 2007] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful game playing robot. In *National Conference on Artificial Intelligence (AAAI-07)*, July 2007.

[Schiffel and Thielscher, 2011] Stephan Schiffel and Michael Thielscher. Reasoning about general games described in GDL-II. In *Proceedings of the AAAI Conference on Artificial Intelligence*, San Francisco, August 2011. AAAI Press.

[Shannon, 1950] Claude Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314):256–275, 1950.

[Sheppard, 2002] Brian Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence.*, 134:241–245, 2002.

[Shoham, 2009] Yoav Shoham. *Multi-agent Systems: Algorithmic, Game-theoretic, and Logical Foundations*. Cambridge, 2009.

[Slagle and Dixon, 1969] J. R. Slagle and J.K. Dixon. Experiments with some programs that search game trees. *Journal of the ACM*, 16(2):189–207, 1969.

[Sun *et al.*, 2011] Yanhua Sun, Gengbin Zheng, Pritish Jetley, and Laxmikant V. Kale. An Adaptive Framework for Large-scale State Space Search. In *Proceedings of Workshop on Large-Scale Parallel Processing (LSPP) in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2011*, Anchorage, Alaska, May 2011.

[Tesauro, 1995] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, 1995.

[Thielscher, 2010] Michael Thielscher. A general game description language for incomplete information games. In *Twenty-third Conference on Artificial Intelligence (AAAI-10)*, pages 994–999, July 2010.

[Van den Broeck *et al.*, 2009] Guy Van den Broeck, Kurt Driessens, and Jan Ramon. Monte-carlo tree search in poker using expected reward distributions. In *ACML*, pages 367–381, 2009.

[Waugh *et al.*, 2009] Kevin Waugh, Nolan Bard, and Michael Bowling. Strategy grafting in extensive games. In *Advances in Neural Information Processing Systems 22 (NIPS)*, pages 2026–2034, 2009.

[Wolfe and Russell, 2007] Jason Wolfe and Stuart Russell. Exploiting belief state structure in graph search. In *ICAPS Workshop on Planning in Games*, 2007.