Chapter 3.1

The Role of Imperfect Information

An obvious source of approaches to strategy formulation is the field of classical strategic games. Classical game-tree search techniques have been highly successful in classical games of strategy such as chess, checkers, othello, backgammon, and the like. However, all of these games are *perfect-information* games: each player has perfect information about the current state of the game at all points during the game. Unlike classical strategic games, practical adversarial reasoning problems force the decision-maker to solve the problem in the environment of highly imperfect information. Some of the game-tree search techniques used for perfect-information games can also be used in imperfect-information games, but only with substantial modifications.

In this chapter,[1] we classify and describe techniques for game-tree search in imperfect information games. In addition, we offer case studies of how these techniques have been applied to two imperfect-information games: Texas Hold'em, which is a well-known poker variant, and kriegspiel chess, an imperfect-information variant of chess that is the progenitor of modern military wargaming [15].

Classical Game-Tree Search

In order to understand how to use game-tree search in imperfect-information games, it is first necessary to understand how it works in perfect-information games. Most game-tree search algorithms have been designed for use on games that satisfy the following assumptions:

[1]This work was supported by the following grants, contracts, and awards: ARO grant DAAD190310202, ARL grants DAAD190320026 and DAAL0197K0135, the ARL CTAs on Telecommunications and Advanced Decision Architectures, NSF grants IIS0329851, 0205489 and IIS0412812, UC Berkeley contract number SA451832441 (subcontract from DARPA's REAL program). The opinions expressed in this chapter are those of the authors and do not necessarily reflect the opinions of the funders.

- *The game is a sequential-move game*, i.e., it consists of a sequence of actions called *moves*. Examples include moving a piece in a game such as chess, checkers, othello, or backgammon, and playing a card in a game such as poker or bridge.
- *It is a two-player game*, i.e., the moves are made by two independent agents (or, in games such as bridge, two *teams* of agents) called *players*.
- *The game has real-valued payoffs*, i.e., whenever the game ends, each player (or team of players) receives a *payoff* that can be expressed as a real number. For example, a player's payoff in poker is the amount of money that he/she wins or loses. In chess or checkers, the payoffs can be represented numerically as 1 for a win, -1 for a loss, and 0 for a draw.
- *The game is a zero-sum game*, i.e., the two players (whom we will call Max and Min) are adversaries. Technically, a zero-sum game is one in which the sum of the payoffs is always zero, e.g., chess, checkers, and poker. But the term "zero sum" is often used to include many games in which the sum of the payoffs is nonzero, provided that these games can be translated into equivalent games in which the sum is zero (e.g., by subtracting some constant c from the payoffs of both players). Bridge is an example of such a game. Probably the best-known example of a non-zero-sum game is the prisoner's dilemma [3].
- *The game is a perfect-information game*, i.e., the players always have complete information about the game's current state. This includes games such as chess, checkers, othello, and backgammon, but it excludes most card games and some board games (e.g., battleship and kriegspiel chess).

Games satisfying the above requirements can be represented by *game trees* such as the simple one shown in Figure 1. In this figure, the square nodes represent states where it is Max's move, the round nodes represent states where it is Min's move, and the edges represent moves. The terminal nodes represent states in which the game has ended, and the numbers below the terminal nodes are the payoffs. The figure shows the payoffs for both Max and Min; but we will usually show only the payoffs for Max (since Min's payoff is always the negative of Max's payoff).

Suppose that two players are playing a game, the current state is s, and the player to move is p. It has become conventional to say that "perfect" play for p at s consists of choosing the move that produces the highest possible payoff if both players play perfectly from that point onwards. From this, it follows that at s, the

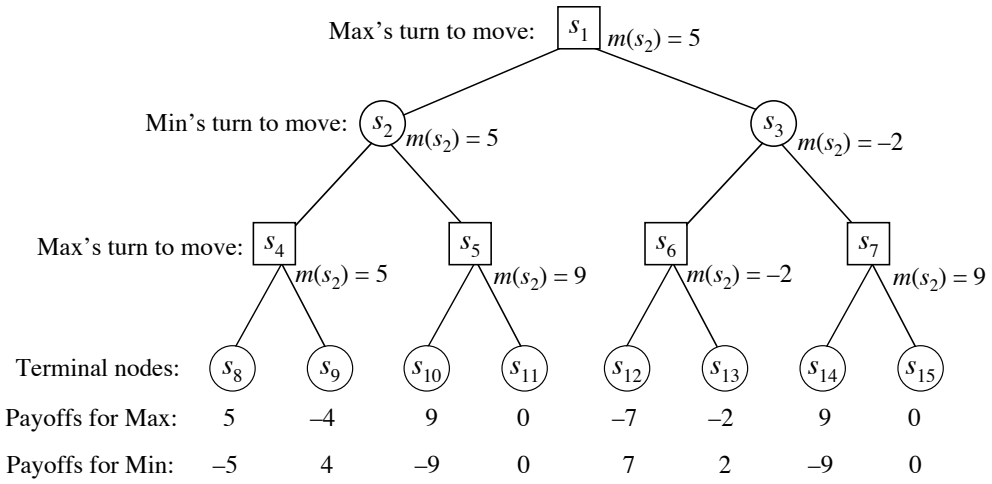

Figure 1: A simple example of a game tree.

effective payoff for Max is

$$m(s) = \begin{cases} \text{Max's payoff at } s & \text{if } s \text{ is a terminal node,} \\ \max\{m(t) : t \text{ is a child of } s\} & \text{if it is Max's move at } s, \\ \min\{m(t) : t \text{ is a child of } s\} & \text{if it is Min's move at } s, \end{cases} \tag{1}$$

where *child* means any immediate successor of s.

For example, in Figure 1,

$$m(s_2) = \min(\max(5, -4), \max(9, 0)) = \min(5, 9) = 5;$$
$$m(s_3) = \min(\max(s_{12}), \max(s_{13}), \max(s_{14}), \max(s_{15}) = \min(7, 0) = 0.$$

Hence the perfect play for Max at s_1 is to move to s_2. The formula for $m(s)$ is a special case of von Neuman and Morgenstern's famous Minimax Theorem, hence $m(s)$ is called Max's *minimax value* at s. The minimax value for Min is, of course, $-m(s)$.

If one chooses a move by applying Eq. (1) directly, this requires searching every state in the game tree, but most nontrivial games have so many states that it is completely infeasible to explore all of them. Hence a number of techniques have been developed to speed up the computation. The best known ones include

- *Alpha-beta pruning*, which is a technique for deducing that the minimax values of certain states cannot have any effect on the minimax value of s, hence those

states and their successors do not need to be searched in order to compute s's minimax value.

- *Limited-depth search*, which uses the following modified version of Eq. 1, where d is an arbitrary number called the *cutoff depth*, and $e(s)$ is a *static evaluation function* that uses *ad hoc* techniques to produce an estimate of $m(s)$:

$$m(s,d) = \begin{cases} p\text{'s payoff at } s & \text{if } s \text{ is a terminal state,} \\ e(s) & \text{if } d = 0, \\ \max\{m(t, d-1) : t \text{ is a child of } s\} & \text{if it is Max's move at } s, \\ \min\{m(t) : t \text{ is a child of } s\} & \text{if it is Min's move at } s. \end{cases} \tag{2}$$

- *Transposition tables.* These are hash tables in which one can store the minimax values of some of the states. This way, if those states occur more than once in the game tree, it will not be necessary to compute their values more than once.
- *Quiescence search.* This is a modified version of Eq. 2 in which if $d = 0$, we will still continue to search below $e(s)$ if s is not *quiescent*, if something is happening at s (e.g., a pending capture in chess) that is likely to make $e(s)$ inaccurate.

Most good game-tree search programs use a combination of these and several other techniques.

Game-Tree Search in Imperfect Information Games

In an imperfect-information game, a player p's *belief state* is the set $b = \{s_1, s_2, \ldots, s_n\}$ of all states that are consistent with the information currently available to p. If the players move in alternation, then in principle it is possible to model the game as an *imperfect-information game tree* G in which each node u represents a belief state b_u, and the set of edges emanating from u represents the $moves(u)$ of all moves that a player *might* be able to make at u (i.e., a move is in $moves(u)$ if it is applicable to at least one state in b_u). In such a game tree, the utility value of u would depend on the probability distribution over b_u; i.e., it depends on the probability, for each state $s \in b_u$, that the current state is actually s. However, in practice this approach presents some immense difficulties:

One difficulty is that the game tree may have a very large branching factor. Generally the set $moves(b_u)$ of all moves that a player might be able to make is much larger than the set of moves applicable to each state $s \in b_u$, hence the

branching factor of the imperfect-information game tree is much larger than what it would be if we had perfect information. Since the size of a game tree generally is exponential in the tree's branching factor, this means that the game tree for an imperfect-information game can dwarf the game tree that we would get if we had perfect information.

As an example, consider a two-player card game in which an ordinary deck of playing cards is divided into its four suits, and each player is dealt one card of each suit. Suppose the players play cards in alternation until neither player has any cards left. If Max can see what cards are in both player's hands, then the branching factor (the number of nodes that are immediate successors) is $5 - n$ at Max's n'th move and $5 - n$ at Min's n'th move, for $n = 1, 2, 3, 4$. Thus the number of leaf nodes in the game tree is

$$4 \times 4 \times 3 \times 3 \times 2 \times 2 \times 1 \times 1 = 48.$$

But suppose that Max can see the cards in his/her hand, but not the cards in Min's hand—just the cards that Min has already played. This changes the branching factor at Min's n'th move to $12 \times (5 - n)$, so the number of leaf nodes in the game tree is

$$(12 \times 4) \times 4 \times (12 \times 3) \times 3 \times (12 \times 2) \times 2 \times (12 \times 1) \times 1 = 11943936.$$

Another difficulty is how to determine, for each possible move that an adversary might be able to make, the probability that the adversary actually *can* make that move. In card games where the cards are dealt to the players, this probability (or a reasonable estimate) can be computed from the probabilities of the possible card distributions. But in a game like kriegspiel chess, in which the uncertainty arises from not knowing what moves the adversary has already made, there is no clear way to compute such a probability.

A third difficulty is that even if the above probabilities were known, it still is not clear how to compute an estimate of a node's utility value. In perfect-information games, the minimax formula has been successful because it provides a pretty good (although not perfect) model of how a strong adversary might behave. But if the adversary's information about the game is imperfect, then it is unlikely that the adversary will make the move predicted by the minimax formula. Some other model of the adversary is needed instead, and often it is unclear what this model should be.

Several ways have been developed to try to circumvent the above difficulties. They can be classified into three main types (which are often used in combination

with each other): aggregation techniques, Monte-Carlo sampling, and opponent modeling. We will now discuss each.

Aggregation Techniques

Similarity-based aggregation. There are cases in which two different states in a game are similar enough to be treated as equivalent. For example, in bridge, cards that have similar rank may often be viewed as equivalent: playing one of them will usually have the same effect as playing any of the others. This effectively decreases the branching factor of the search space by reducing the number of plays that need to be evaluated. This idea was initially developed for use in the game of sprouts [Applegate et al., 1991], and it has been used successfully in bridge in combination with Monte Carlo sampling [Ginsberg, 1996]. More recently, in the game of poker, it has been used in the development of a computer program that plays at the level of an good human player [Billings et al., 2003].

Strategy-based aggregation. The normal approach to game tree generation is action-based: each branch corresponds to an action that one might take. In strategy-based game tree generation, each branch instead corresponds to some strategy that a player might try. For example, bridge play is composed strategies such as ruffing, cross-ruffing, finessing, cashing out, etc. For each strategy, one can define applicability conditions telling whether the strategy is a reasonable thing to try, and partially-instantiated game-tree fragments giving the possible ways that the strategy might be carried out. This reduces the branching factor of the game tree because the number of applicable strategies at each point is usually smaller than the number of possible actions. The primary drawback of this approach is that a substantial amount of human effort needed to make sure that the set of strategies is complete and correct—and for this reason it has not been widely used. However, it has been used successfully in computer bridge [Smith et al., 1998].

Monte-Carlo Sampling

In an imperfect-information game G, suppose we can generate a hypothesis h for what the missing information is. If the hypothesis h is correct, then this will reduce the game to a perfect-information game $G(h)$ that can be searched using ordinary game-tree search techniques. In general, we will not know whether h is correct—

but if we have a probability distribution over the set H of all possible hypotheses, then we can use Monte Carlo techniques as follows. First, use the probability distribution to randomly generate n hypotheses $h_1, \ldots, h_n$ for what the missing information might be. Next, do an ordinary minimax game-tree search on each of the games $G(h_1), G(h_2), \ldots, G(h_n)$, and average the results in order to produce approximate utility values for the nodes of G. The larger the value of n, the better the approximations will be, and even for a large value of n, we can search $G(h_1), G(h_2), \ldots, G(h_n)$ much more quickly than we could search G itself.

Statistical sampling has an important theoretical limitation [13]: it does not produce correct evaluations of information-gathering moves or moves intended to deceive the opponent (see chapters 2.3 and 2.4 of this book), because neither kind of move is even possible in the perfect-information game trees $G(h_1), G(h_2), \ldots, G(h_n)$. Despite this limitation, statistical sampling has worked well in practice in several games. Examples of such games include bridge [30, 14], scrabble [26], and poker [4].

Opponent Modeling

Opponent modeling is an essential part of expert human game-playing. At the professional level, baseball players know which batters are susceptible to fastballs, basketball players know their opponents' better shooting hands, chess players will study the past openings of their opponents before meeting them in tournament play, and poker players are proceeded by their reputations.

In classical game-tree search on perfect-information games, the minimax formula is a simple model for the opponent: it assumes an computationally unbounded opponent capable of making the move that is worst for us at any point in the game. In perfect-information games, very little opponent modeling other than this has been done in any explicit sense,[2] but a certain amount of opponent modeling often appears implicitly. For example, one source of static evaluation functions for chess is automated analysis of databases of the games played by chess grandmasters [16]. A chess program using such a static evaluation function implicitly attempts to push its opponent towards a board position that is would be unfavorable to one of the grandmasters whose games appear in the database.

In partial-information games, opponent modeling has a much more important

[2] A notable exception is [9], an interesting theoretical model for opponent modeling in total information games that provides a syntax and algorithm for dealing with concepts such as "he thinks that I think that he thinks ...."

role to play. Most of the work on opponent modeling in partial-information games has been done in the game of poker, where the need to model an opponent's betting style is quite obvious.

For the game of Texas Hold'em (see the case study later in this chapter), one approach [7] is to keep a record of a player's past moves, and assume that his/her future behavior will be similar to how he/she behaved under similar conditions in the past. A more recent approach [11] uses a neural net to aggregate the available information into a prediction of the opponent's next move. In both of these works, the programs that do opponent modeling far outperform those which do no modeling.

For some additional work on opponent modeling (specifically, a reinforcement algorithm for constructing opponent models in extensive games), see chapter 3.5.[3]

Combining the Techniques

In most practical applications, two or more of the above techniques are used in combination with each other. For example, statistical sampling and aggregation can be combined by using an aggregation technique to construct a simplified version of the game tree, doing the statistical sampling on this simplified game tree, and using the results of the sampling to choose a move in the original (unsimplified) game. This has been used in several bridge programs [30, 14] and at least one poker program [4].

Case Study: Texas Hold'em

Texas Hold'em is a variant of poker that originated sometime during the 1920s. It falls into a class of poker games called "community card" or "shared card" games, which are so called because some of the cards are dealt face-up in the center of the table and are shared by all of the players. The rest of each player's hand consists of "hole cards" which are dealt specifically to that player and are not seen by the other players. Each player's hand is comprised of his/her hole cards together with the community cards.

[3] In addition, we have developed some very successful techniques for building and maintaining opponent models in the Iterated Prisoner's Dilemma with Noise [2], which is a version of the Iterated Prisoner's Dilemma in which accidents and miscommunications can occur. However, the work is beyond the scope of this book.

Texas Hold'em is currently the most popular of the community card poker games, and a number of books have been published on how to play it [17, 27, 22, 28, 10]. It has been called the "Cadillac of Poker,"[4] and is well respected as one of the most strategically complex poker variants. In principle it can be played by up to 22 players, but in practice it is generally played by groups of 2 to 10 people.

Game play in Texas Hold'em works as follows: each player is dealt two cards face down. The cards are the player's *hole cards* and represent their hidden information. There is then a round of betting reffered to as *pre-flop*. Pre-flop betting is begun with a forced bet by the two players left of the dealer. This bet is called the blind. After this betting round, three cards are exposed face-up in the center of the table. These are community cards which all players may use in their hand. These three cards are called the flop and following their exposure there is another round of betting, this time with no forced bets or blinds. After this round, if there are still players left, another community card, called the turn, is exposed. Following another betting round the final community card, called the river, is shown. There is then a final betting round, and if there are players left in the game, there is a showdown in which the players show their highest five card hand (created using the two hole cards and the five community cards). The player with the highest of these hands wins.

In Limit Texas Hold'em, the betting rounds have a specified structure. Players play in clockwise order. On his/her turn, each player has one of three options: bet, call, or fold. The *bet* action raises the amount of money that all players must call by a fixed amount. A *fold* causes a player to leave the hand: the player needs not put any more money into the pot, but also has no ability to win the hand. When a player *calls*, he/she matches the current bet and stays in the current hand. The initial bet is 0 (except in pre-flop where the blind is generally worth one bet). There is generally a cap of 4 on the number of bet actions which may be made by all players in any betting round. Thus, in a two player poker game, there are 17 possible ways which a round of betting may proceed, only 9 of which end in a call.

Play

Poker has been a subject of AI research for yearly 30 years [12], but most work has either consisted primarily of theoretical analysis [34, 35, 18] or has concentrated on specific aspects of poker rather than the entire game [31, 29, 19].

[4]This description was used, for example, in the 1998 movie *Rounders*.

The best-known and most successful work on building complete poker players has been done by a group of researchers at the University of Alberta [7, 6, 4]. Their ambition is to build a poker player for the game of Texas Hold'em that better than the best human players. They are well on their way toward doing so [32].

Their best-known approach, reported in [4], is to use linear programming techniques to compute game-theoretic optimal strategies on a simplified version of 2-player Texas Hold'em, and use those strategies in the real (i.e., unsimplified) game. They have since supplanted this algorithm with one that uses game-tree search techniques along the lines of what we described in the *Game-Tree Search* section earlier in this chapter [5]. But since that work is not yet published, we cannot provide the details here.

Opponent Modeling

In this section we discuss two different opponent-modeling techniques for the game of Texas Hold'em. Both of these techniques were developed by researchers at the University of Alberta [7, 11].

In [7], opponent modeling is accomplished using the probability that the opponent is holding each of the various possible hole cards. These hole cards are initially assigned weights based on the *a priori* random distribution caused by the deal. As the opponent makes actions through the game, the opponent-modeling program recomputes these weights to be consistent with the opponent's actions. For instance, if an opponent calls one bet pre-flop and then raises on the flop, it is likely that the opponent has a hand which was ok pre-flop but likely to win the hand after the flop. That is, the hand may have a been expected to win with probability 0.55 before the flop, and with probability 0.75 after the flop.[5] The player's chances of winning changed, and therefore so did their action. The key to player modeling in [7] is the idea that each player is comfortable with each possible action only when the win probability meets certain criteria, and further, that the criteria is particular to each individual player. That is, a player will call when their hand is likely to win with a probability above 0.5, likely to raise with hands with winning probability more than 0.75, and likely to fold otherwise. On re-weighting, weights of hands consistent with opponent action increase at the expense of those hands which are not consistent.

As an example of this, consider the following situation: we have $9\heartsuit 9\diamondsuit$. This

[5] It should be noted that "win probability" would be more accurately stated as "expected value". For purposes of presentation, this has been left out, but a full explication can be found in [7].

is a good opening hand so we raise when we get the chance. Our model of the opponent at this point gives equal weight to each of the 1225 other starting hands. The opponent then, on her action, re-raises us. This affects our opinion of the opponent's hand – our opponent model tells us that the opponent is likely to re-raise with a hand which will win at least 60% of the time. So we now adjust the weights to take this into account, making only a very few hands possible. In fact, in two player play, only pocket pairs, hands containing an ace, or very few hands containing a king are that likely to win. At this point we call,[6] and we move to the flop. It is our misforturne that $A\clubsuit J\spadesuit 2\clubsuit$ falls on the flop. The opponent bets at this point and we must re-weight the opponent's hands again. Using the same heuristic model, we now increase the weight of all hands which win more than 60% of the time. At this point, there is extremely high probability that the opponent has a pair of aces, which, in poker lingo, dominates our pair of nines. Therefore we fold. Had we not been using any opponent modeling, we may have called, a play which has an expected payoff of about 0.05 on our bet of 1[7], an undesirable situation. Thus we see both that opponent modeling is necessary and how this particular method of opponent modeling fills that need.

In Davidson *et al.* [11], opponent modeling is accomplished via an artificial neural network. The network takes as input all of the various aspects of the game, and returns if it thinks the opponent will bet, check or fold. How this information can be used in deciding our next move is obvious: if we know the opponent wants to fold, we should always bet, forcing the fold. The disadvantage to this technique is that it requires much data and training, and that it is susceptible to subtle, high level sorts of play such as the slowplay. Davidson *et al.* show this technique to work significantly better than the previous weight-based approach in practice against reasonable human players.

Case Study: Kriegspiel Chess

We now describe our own work on the game of kriegspiel chess.[8] Kriegspiel chess [20, 21] is an imperfect-information variant of chess that was developed by a Prussian military officer in 1824 and became popular as a military training exercise. It is the progenitor of modern military wargaming [15].

Kriegspiel chess is like ordinary chess except that neither player can see the

[6]This is likely not the best action at this point, but we assume it for purposes of exposition.

[7]if the opponent actually does have an ace

[8]Most of this section is excerpted from [23], which contains some additional technical details.

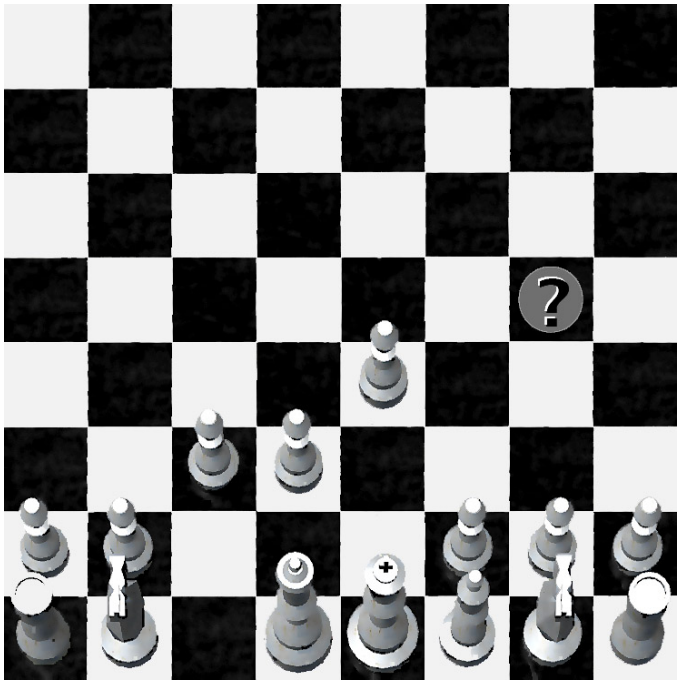

Figure 2: A kriegspiel chess board from the viewpoint of White. Black has just taken White's bishop in the space marked "?". Hence White knows Black has a piece there but not what the piece is.

other's pieces. However, a player can get information about the opponent through interactions such as captures (see Figure 2), checks, and moves that are blocked because an opponent's piece is in the way. When player x captures one of player y's pieces, a referee announces that he/she has made a capture but not what piece was captured, and the referee removes the captured piece from y's board but does not say what piece captured it. When x tries to make a move that is illegal (an attempted pawn take, or moving into check, or attempting to jump an opponent's piece), the referee announces that the move is illegal but not why. When x puts y's king in check, the referee tells both players that y's king is in check, and gives them partial information about how the check occurred (namely by rank, file, long diagonal, short diagonal, knight, or some combination of the above). Both players hear all referee announcements.

The published literature on kriegspiel chess is rather small. Two books have been written on how to play kriegspiel chess [20, 21]. [33] describes a program to act as the referee that advises the kriegspiel players of illegal moves, captures,

and the like. [25] describes a search strategy for some imperfect-information games that are simpler than kriegspiel. [8] describes some search strategies for kriegspiel-chess endgames. [24] describes a program that uses logical reasoning techniques to predict forced wins in kriegspiel chess. But this technique is feasible only at points where the belief states are relatively small, such as the endgame.

How to build a really good program to play kriegspiel chess is largely an unsolved problem. Even among human players, kriegspiel chess is a notoriously difficult game to win [20, 21]; most games end in draws. Kriegspiel chess presents some difficulties that do not occur in other imperfect-information games (e.g., bridge, scrabble, and poker):

1. Twenty moves into a kriegspiel chess game, a conservative estimate is that at each node of the game tree, the belief-state size—i.e., the number of states consistent with the current sequence of observations—can be more than 10^{13}. In comparison, the belief-state size in Bridge is only about 10^7, and in Texas Hold'em it is only about 1000.

2. In bridge and poker, usually the statistical sampling is not done on the game itself, but on a simplified approximation in which the state space and belief states are much smaller. In bridge, the approximation is done by treating various sets of states as if they were equivalent, a technique that was first used in the game of sprouts [1]. In poker, a linear-programming approximation has been used [4]. But in kriegspiel chess, it is unclear how or whether such an approximation could be constructed. For our case study, we did not attempt to construct one.

3. In bridge, poker, and scrabble, the opponent's actions are observable. The uncertainty about the current state arises from external events that can be modeled stochastically: the random deal of the cards in bridge or poker, and the random choice of a tile in Scrabble. This makes it relatively easy to tell whether or not a state s is consistent with a belief state b, and to assign a probability to s given b.

   In kriegspiel chess, the opponent's actions are not observable, and this uncertainty has no simple stochastic model. Telling whether a state s is consistent with the current belief state b means checking whether there is a *history* (i.e., a sequence of moves) that is consistent with b and leads to s, and in the average case this takes exponential time. And there is no clear way to put a probability distribution on the states in the belief state.

   For our case study, we constructed several algorithms for generating the ran-

Table 1: Abstract statistical-sampling algorithm for move evaluation. Game-tree-search is a perfect-information game-tree search algorithm such as the ones described earlier in this chapter.

```
procedure Choose-move(S)
    M ← {moves applicable to states in S}
    for every s ∈ S and every m ∈ M do
        v_{s,m} ← Game-tree-search(γ(s, m))
    return argmax_{m∈M} Σ_{s∈S} v_{s,m} P(s)
```

dom sample of game boards used in the tree search. AOSP generates game boards that are consistent with the entire sequence O of observations that a player has made during the game. LOS only requires consistency with the last observation o_i. HS behaves like AOSP at the beginning of the game, but as the game progresses it gradually switches over to behaving like LOS. The next section describes the algorithms in more detail.

At first glance, one might expect LOS to be the worst of the the algorithms, AOSP to be the best, and HS to be somewhere in between. But our theoretical analysis on some simplified game-tree models suggests that in some cases HS may outperform AOSP; and our experiments show HS outperforming AOSP in kriegspiel chess.

Algorithms

We first introduce some notation. As the game progresses, the players' moves will generate a sequence of states $S_i = \langle s_0, s_1, \ldots \rangle$ called the *game history*. At each state s_i, each player p_j will be able to make an observation o_{ij} of s_i; usually o_{ij} will include complete information about p_j's position and partial information about the other player's position. At s_i, player p_j's *observation history* is $O_{ij} = \langle o_{1j}, o_{2j}, \ldots, o_{ij} \rangle$, and p_j's *belief state* is $b_{ij} = \{$all states that satisfy $O_{ij}\}$.

Our sampling algorithms will be based on the following properties of a state s: s is *last-observation consistent* if it is consistent with o_{ij}, and *all-observation consistent* if it is consistent with O_{ij}.

Table 1 shows an abstract version of statistical game-tree search. S is the sample set of states, $\gamma(s, m)$ is the state produced by performing move m in state s, Game-tree-search is a perfect-information game-tree-search algorithm such as alpha-beta, and P is a probability distribution over the states in S. Some addi-

tional code must be added to handle the case where a move m is applicable to some states but not others; this code is game-specific and we do not discuss it here.

We now can define three different sampling algorithms that provide input for Choose-move. In each case, k is the desired number of states in the statistical sample, and i is how many moves the players have played so far.

- **LOS** (**L**ast **O**bservation **S**ampling) If there are fewer than k last-observation consistent states, then let S contain all of them; otherwise let S contain k such states chosen at random. Return Choose-move(S).
- **AOSP** (**A**ll **O**bservation **S**ampling with **P**ool): AOSP returns a move, and stores a set of states (a *pool*) to use as input to use the next time AOSP is called. Every state in the pool is to be consistent with O_{ij}, though we do not assume that all such states are in the pool. Let S_0 be the pool AOSP returned last time, and $M = \{$all of the other player's possible responses to p_j's last move$\}$. Let $S_1 = \{\gamma(s, m) \mid s \in S_0, m \in M$, m is applicable to s, and $\gamma(s, m)$ satisfies $O_{ij}\}$. If $|S_1| < k$, then let $S_2 = S_1$; otherwise let S_2 contain k states chosen at random from S_1. Let $m =$ Choose-move(S_2). Return $(m, \{\gamma(s, m) \mid s \in S_1\})$.
- **HS** (**H**ybrid **S**ampling): Like AOSP, HS returns a move and a set of states. Compute S_1 and S_2 same as in AOSP. If $|S_2| < k$ then let S_3 be a set of $k - |S_2|$ random last-observation consistent states; otherwise $S_3 = \emptyset$. Let $m =$ Choose-move$(S_2 \cup S_3)$. Return $(m, \{\gamma(s, m) \mid s \in S_1\})$.

Theoretical Analysis

Analyzing the performance of these algorithms is impossible without making simplifying assumptions, but there is more than one set of assumptions one might make. Below we do two analyses, based on two different sets of assumptions. The differing assumptions lead to differing conclusions about which algorithm will do better.

Game-tree analysis. Suppose each state has exactly b children, for some constant b. Suppose that we know all of p_j's moves but not the other player's moves. If the number of states is very large (e.g., 10^{13} as described earlier), then during the early stages of the game, the number of states grows exponentially, with roughly $b^{i/2}$ possible states at the i'th move. Suppose that for each state s where it is the other player's move, the observation history O_{ij} eliminates, on the average, some fraction $1/c$ of that player's possible moves, where $c > 1$. Then the number of

possible states at the i'th move given O_{ij} is $(b/c)^{i/2}$. Thus the probability of any individual state at depth i being consistent with O_{ij} is $(1/c)^{i/2}$, which approaches 0 at an exponential rate as i increases.

Thus, if the game continues to grow as a tree with a branching factor of b, then our analysis suggests the following:

- AOSP's sample set S_2 will decrease in size as the game progresses. The probability of a state s's successors being consistent with b_i is $1/c$, since s is already known to be consistent with b_{i-1}. Hence as the game progresses, S_2 will soon become too small for the results to have much statistical significance and AOSP's play will begin to resemble random play.
- Each board generated by LOS is unlikely to be consistent with the current belief state; thus the values computed by LOS are likely to be close to random.
- At the beginning of the game, HS will behave identically to AOSP. As the game proceeds and the size of S_2 decreases, HS will put more and more randomly generated boards into S_3, thus making the results more noisy. Thus HS's quality of play is likely to be worse than AOSP's.

Game-graph analysis. If there are n possible states in the game, then the number of moves at each level cannot continue to grow exponentially, but will eventually flatten out. The game "tree" will be a graph rather than a tree, with n nodes (at most) at each depth, one for each possible state. There will be b edges from each node at depth i to nodes at depth $i+1$, $1/c$ of which are consistent with any given observation; suppose these edges go to a random set of nodes. Then for each state s, the probability (under certain independence assumptions) that it is reachable in i moves is about $min(1, (n-1)^{i-3}((b/c)/(n-1))^{i-1})$. In other words, the probability that a randomly chosen state s has a history consistent with O_{ij} approaches 1 exponentially. This suggests the following:

- Rather than degrading to random play as in the game-tree analysis of the previous section, AOSP's quality of play will eventually level off at some level above that, depending on the number of states available in the pool.
- As the game proceeds, the probability of a randomly generated board being consistent with the current belief state will increase toward 1; thus LOS will produce increasingly good quality of play. However, its play will be limited by the fact that it has no good way to assign relative probabilities to its randomly generated boards.
- At the beginning of the game, HS will behave identically to AOSP. As the game

proceeds and AOSP's sample size decreases, HS will fill up the rest of the sample with randomly generated boards—but as the game proceeds, it will become increasingly likely that these randomly chosen boards are consistent with the current belief state. Thus HS's quality of play is likely to be better than AOSP's.

Discussion. With the first set of assumptions, AOSP is likely to perform much better than LOS, and somewhat better than HS. With the second set of assumptions, it is unclear which of LOS and AOSP will be better, but HS is likely to perform better than AOSP and LOS.

Since both sets of assumptions represent extremal cases, our analyses suggest that the actual performance is likely to be somewhere in between. In particular, it seems plausible that HS will perform better than AOSP, i.e., that if last-observation consistent boards are included in the statistical sample later in the game, this will help rather than hurt the evaluations. The section after next describes our experimental test of this hypothesis.

Timed Algorithms

In order to make fair comparisons among LOS, AOSP, and HS, they cannot be implemented in the exact way described in the *Algorithms* section. They must be modified to take, as an additional input variable, the amount of time t available to decide on a move. This is necessary so that the algorithm can do as well as it can within that amount of time. We call the modified algorithms Timed LOS, Timed AOSP, and Timed HS.

Timed LOS. Rather than taking the set S as input as shown in Table 1, Timed LOS generates the members of S one at a time and evaluates them as they are generated, so that it can generate and evaluate as many boards as it can during the time available. Once the time is up, it returns the move whose average value is highest, as shown in Table 1.

Timed AOSP. Timed AOSP maintains a pool of states $P = \{s_1, \ldots, s_p\}$ that are known to be consistent with the current belief state b. Using an estimate of how long Game-tree-search will take on each board, it calculates some number of boards k_t that it can evaluate during the available time t. The estimate k_t is deliberately a little low, to try to keep Timed AOSP from running overtime and

to ensure that there will be time left over to attempt to generate more consistent boards. There are three cases:

- If $p \geq k_t$ then Timed AOSP calls Choose-move($\{s_1, \ldots, s_{k_t}\}$), and returns the recommended move.
- If $0 < p < k_t$ then Timed AOSP calls Choose-move(P), and returns the recommended move.
- If $p = 0$ then Timed AOSP returns a random move.

During whatever remains of the available time, AOSP tries to generate more histories that are consistent with b; and for every such history, it adds the resultant board to the pool.

Each time the referee makes an announcement, Timed AOSP must update the pool to be consistent with the announcement. This can cause the pool to either shrink (when Timed AOSP is told a move is illegal) or to grow (when Timed AOSP is told that the opponent has moved). This computation occurs at the beginning of AOSP's turn.

If the pool were allowed to grow unchecked, it could potentially get quite large; hence we limit its size to 20,000 boards. If the number of boards in the pool ever goes higher than this, we remove enough boards to get to the number of boards down to 10,000. Because the 30 second time limit allows only enough time to call Choose-move on a set about 350 boards from the pool, this is believed adequate.

Timed HS. Timed HS works the same as Timed AOSP, with one exception. If $0 \leq p < k_t$, then Timed HS generates a set R that contains $p - k_t$ random boards that are consistent with o_{ij} (we call these *last-observation* consistent boards). Then it calls Choose-move($P \cup R$). This rules out the possibility of ever having to make a random move. It also restricts the amount of time that Timed HS can spend generating additional boards to put into the pool.

We have implemented all three of these algorithms, using a combination of C and C++. For Choose-move's Game-tree-search subroutine, we used the GPL'ed chess program provided by GNU, and modified it to return a minimax value for a particular board.

By the time this chapter appears in print, we intend to have made our implementations publicly available as a kriegspiel-chess game server accessible from the web.

Table 2: Win/loss/draw percentages plus or minus a 95% confidence interval, in games where between Timed LOS, Timed AOSP, and Timed HS were played against a random player.

Algorithms	Win (%)	Loss (%)	Draw (%)	Runs
LOS v rand	39 ± 2	0 ± 0.3	61 ± 2	559
AOSP v rand	63 ± 2	0 ± 0.3	37 ± 2	560
HS v rand	65 ± 2	0.5 ± 0.3	35 ± 2	558

Table 3: Win/loss/draw percentages plus or minus a 95% confidence interval at each move, in games between Timed LOS, Timed AOSP, Timed HS, and a random player.

Algorithms	Win (%)	Loss (%)	Draw (%)	Runs
AOSP v LOS	31 ± 4.8	0 ± 1	69 ± 4.8	190
HS v LOS	38 ± 5	0.5 ± 1	61 ± 5	190
HS v AOSP	13.3 ± 0.4	10.7 ± 0.4	76 ± 0.5	1669

Experiments

Our experimental hypotheses (based on the analyses in the *Theoretical Analysis* section) were that (1) Timed LOS would perform better than random play, (2) Timed AOSP would perform better than Timed LOS, and (3) Timed HS would perform somewhat better than Timed AOSP. We (the authors) disagreed with each other about the third hypothesis because it was based on a notion that not all of us believed: that the computation time spent introducing and evaluating last-observation consistent boards would not be better spent trying to find and evaluate more all-observation consistent boards.

To test our hypotheses, we played all three algorithms against each other and against a player who moved at random. Each player plays approximately half of the games as white and half of the games as black. All experiments were run on Xeon 2.6GHz chips with 500 MB RAM, running Linux. Each player was allowed to spend 30 seconds deciding each move, including moves which are decided after an attempted illegal move.

Table 2 shows the percentage of wins, losses, and draws when each of the three algorithms is played against a player who makes moves at random. Both Timed AOSP and Timed HS do much better against the random player than Timed LOS does. Timed HS does slightly better than Timed AOSP, but the difference

is not statistically significant. The large number of draws is unsurprising, since kriegspiel chess is a notoriously difficult game to win.

Table 3 shows the percentage of wins, losses, and draws when the three players are played head-to-head against each other. Again, both Timed AOSP and Timed HS do much better than Timed LOS. Timed HS does somewhat better than Timed AOSP, and this time the results are statistically significant. Figure **??** shows that Timed HS and Timed AOSP have nearly the same material at each move in the game.

Figure 3 shows the number of last-observation boards used by Timed HS at each move in the games against Timed AOSP. Recall that these are the boards that Timed HS generates using the LOS algorithm when it runs out of time using the AOSP algorithm. Near the start of the game, Timed HS acts like Timed AOSP. In the middle of the game, it acts like a combination of Timed AOSP and Timed LOS. Near the end of the game, whether it acts like Timed AOSP or Timed LOS varies greatly, depending on the particular game.

This behavior is very interesting, because it suggests that our hypothesis about last-observation consistent boards is correct: they become more useful as the game progresses, because they are more likely to be consistent with the current belief state. Even though we do not know what probabilities to assign to them in the last line of Choose-move, they still provide useful information.

Significance of Our Results

One result demonstrated by our work is that statistical sampling approaches can be useful for game-tree search in kriegspiel chess. This was not obvious beforehand because of the immense size of the belief states in this game.

A second and more surprising result is that it is not necessary for the random sample to consist only of game boards that satisfy all of a player's observations. In fact, we were able to win more often by starting out with such boards, but gradually switching over (as the game progressed) to boards that merely are consistent with the latest observation. The reason is that as the game progresses, a board that is consistent with the last move becomes more and more likely to be consistent with the entire set of observations, even if we have no idea what sequence of moves might have actually generated this board. To the best of our knowledge, ours is the first serious attempt at a good player for the entire game of kriegspiel chess.[9]

[9]If you intend to develop your own kriegspiel-chess program, please send email to this

PUBLISHER: PLEASE PLACE FIGURE 3 APPROXIMATELY HERE

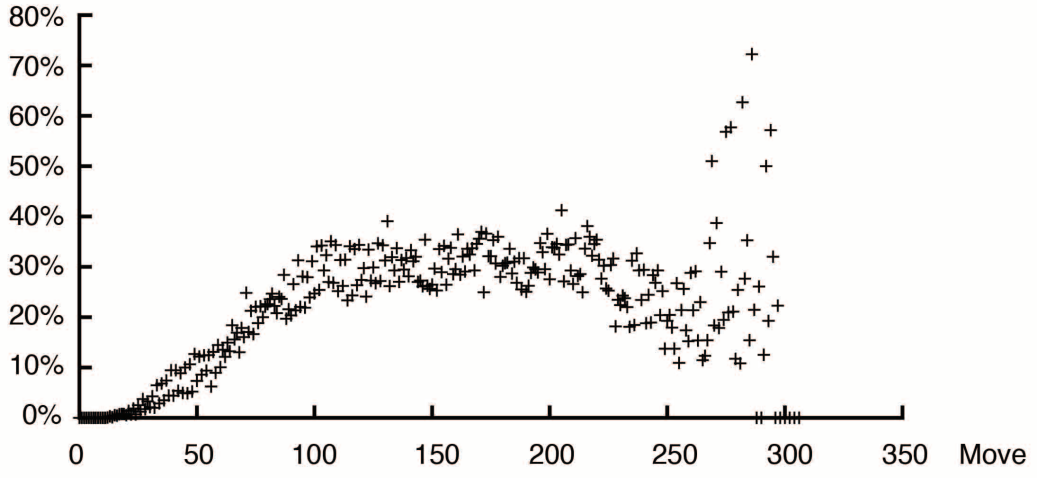

Figure 3: The average percentage of last-observation consistent boards that Timed HS used at each move in its games against Timed AOSP.

Summary

Game-tree search on imperfect-information games is much more difficult than on perfect-information games. The primary problems include (i) exponentially larger game trees than for perfect-information games, (ii) difficulties in calculating the probability that a player can make a given move, and (iii) difficulties in calculating the probability that a player *will* make a given move.

We have discussed the following techniques for addressing those problems: similarity-based aggregation, strategy-based aggregation, monte carlo sampling, and opponent modeling. The University of Alberta's work on Texas Hold'em shows that such techniques can provide a basis for computer programs that perform as well as very good human players. Our work on kriegspiel chess shows that such techniques can also be useful in a game that has many orders of magnitude more uncertainty than poker.

---

chapter's authors at the following addresses: Dana Nau ⟨nau@cs.umd.edu⟩, Austin Parker ⟨austinjp@cs.umd.edu⟩, and V.S. Subrahmanian ⟨vs@umiacs.umd.edu⟩. They would like to inaugurate a kriegspiel chess competition as part of the International Computer Games Association's Computer Olympiad, and need a program to compete against!

References

[1] David Applegate, Guy Jacobson, and Daniel Sleator. Computer analysis of sprouts. Technical report, Carnegie Mellon University, 1991.

[2] Tsz-Chiu Au and Dana Nau. Accident or intention: That is the question (in the iterated prisoner's dilemma). In *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2006.

[3] Robert Axelrod. *The Evolution of Cooperation*. Basic Books, 1985.

[4] D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 661–668, 2003. Won the IJCAI/AAAI 2003 Distinguished Paper Award.

[5] Darse Billings, October 2004. Personal communication.

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

[7] Darse Billings, Denis Papp, Jonathan Schaeffer, and Duane Szafron. Opponent modeling in Poker. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 493–498, 1998.

[8] A. Bolognesi and P. Ciancarini. Searching over metapositions in kriegspiel. In *Computer Games 2004*, 2004.

[9] David Carmel and Shaul Markovitch. Incorporating opponent models into adversary search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1996.

[10] Bob Ciaffone and Jim Brier. *Middle Limit Holdem*. Bob Ciaffone, 2002.

[11] A. Davidson, D. Billings, J. Schaeffer, and D. Szafron. Improved opponent modeling in poker. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1467–1473, 2000.

[12] Nicholas Findler. Studies in machine cognition using the game of poker. *Communications of the ACM*, 20(4):230–245, 1977.

[13] Ian Frank and David A. Basin. Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100(1-2):87–123, 1998.

[14] Matthew L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *IJCAI-99*, pages 584–589, 1999.

[15] COL (R) Bill Gray. History of wargaming. http://www.hmgs.org/history.htm, 2003.

[16] Feng hsiung Hsu, Thomas Anantharaman, Murray Campbell, and Andreas Nowatzyk. A grandmaster chess machine. *Scientific American*, 263(4):44–50, October 1990.

[17] Lee Jones. *Winning Low-Limit Hold-em*. Conjelco, 1994.

[18] Daphne Koller and Avi Pfeffer. Representations and solutions for game-theoretic problems. *Artificial Intelligence*, pages 167–215, 1997.

[19] K. Korb, A. Nicholson, and N. Jitnah. Bayesian poker. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 343–350, 1999.

[20] David Li. *Kriegspiel: Chess Under Uncertainty*. Premier, 1994.

[21] David Li. *Chess Detective: Kriegspiel Strategies, Endgames and Problems*. Premier, 1995.

[22] Ed Miller, David Sklansky, and Mason Malmuth. *Small Stakes Hold'em*. Two Plus Two Publications, 2004.

[23] Austin Parker, Dana Nau, and V.S. Subrahmanian. Game-tree search with combinatorially large belief states. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, August 2005.

[24] Stuart Russell and Jason Wolfe. Efficient belief-state AND-OR search, with application to Kriegspiel. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

[25] M. Sakuta, J. Yoshimura, and H. Iida. A deterministic approach for solving kriegspiel-like problems. In *MSO Computer Olympias Workshop*, 2001.

[26] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

[27] David Sklansky. *Hold 'em Poker*. Two Plus Two Publications, 1996.

[28] David Sklansky and Mason Malmuth. *Hold 'em Poker for Advanced Players*. Two Plus Two Publications, 1999.

[29] S. Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 422–425, 1983.

[30] Stephen J. J. Smith, Dana S. Nau, and Thomas Throop. Computer bridge: A big win for AI planning. *AI Magazine*, 19(2):93–105, 1998.

[31] D. Waterman. A generalization learning technique for automating the learning of heuristics. *Artificial Intelligence*, 1:121–170, 1970.

[32] Peter Wayner. The new card shark. *The New York Times*, pages G1, G7, July 9 2003.

[33] C. S. Wetherell, T. J. Buckholtz, and K. S. Booth. A director for kriegspiel, a variant of chess. *Comput. J.*, 15(1):66–70, 1972.

[34] N. Zadeh. *Winning Poker Systems*. Prentice-Hall, 1974.

[35] N. Zadeh. Computation of optimal poker strategies. *Operations Research*, 25(4):541–562, 1977.