Artificial Intelligence 174 (2010) 670–684

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Monte Carlo tree search in Kriegspiel

Paolo Ciancarini *, Gian Piero Favini

Dipartimento di Scienze dell'Informazione, University of Bologna, Italy

ARTICLE INFO

Article history:
Received 20 September 2009
Received in revised form 4 April 2010
Accepted 4 April 2010
Available online 9 April 2010

Keywords:
Games
Chess
Kriegspiel
Incomplete information
Monte Carlo tree search

ABSTRACT

Partial information games are excellent examples of decision making under uncertainty. In particular, some games have such an immense state space and high degree of uncertainty that traditional algorithms and methods struggle to play them effectively. Monte Carlo tree search (MCTS) has brought significant improvements to the level of computer programs in games such as Go, and it has been used to play partial information games as well. However, there are certain games with particularly large trees and reduced information in which a naive MCTS approach is insufficient: in particular, this is the case of games with long matches, dynamic information, and complex victory conditions. In this paper we explore the application of MCTS to a wargame-like board game, Kriegspiel. We describe and study three MCTS-based methods, starting from a very simple implementation and moving to more refined versions for playing the game with little specific knowledge. We compare these MCTS-based programs to the strongest known minimax-based Kriegspiel program, obtaining significantly better experimental results with less domain-specific knowledge.

1. Introduction

Partial information games provide a good model and testbed for many real-world situations involving decision making under uncertainty. They can be very difficult for a computer program to play well. These games typically require a combination of complex tasks such as heuristic search, belief state reconstruction, and opponent modeling.

Moreover, some games are particularly challenging because at any time the number of possible, indistinguishable states far exceeds the storage and computational abilities of present-day computers. In this paper, the focus is on one such game, Kriegspiel or *invisible chess*. The game is interesting for at least three reasons. Firstly, its rules are identical to those of Chess, a very well-known game; however, the players' perception of the board is different, only being able to see their own pieces. Secondly, it is a game with a huge number of states and limited means of acquiring information. Finally, the nature of uncertainty is entirely dynamic. These issues put Kriegspiel in a category different from other partial information games such as Stratego or Phantom Go (the partial information variant of Go [1]), wherein a newly discovered piece of information remains valid for the rest of the game. Information in Kriegspiel is scarce, precious, and ages fast.

In fact, even if it is an old game, well known to game theorists and even discussed by von Neumann and Morgenstern in [2] under the name of *blind chess*, the first attempt to build an effective Kriegspiel playing program came only in 2005 and was based on Monte Carlo sampling [3]. It was, however, defeated by our first program, described in [4] and based on a form of minimax on a game tree of data structures called *metapositions*. These had been first defined in [5] for a partial information variant of Shogi, that is Japanese Chess. Our program was better than other competing programs, but was not good enough to compete with the best human players.

* Corresponding author.
E-mail address: cianca@cs.unibo.it (P. Ciancarini).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2010.04.017

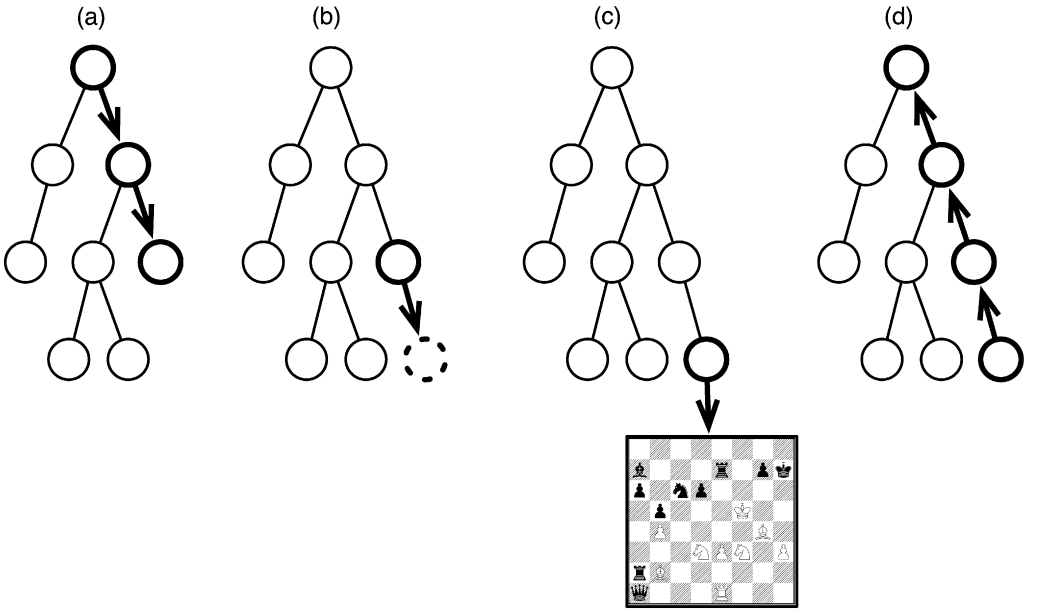

Fig. 1. The four phases of Monte Carlo tree search: selection, expansion, simulation and backpropagation.

In this paper we present and study different ways of applying Monte Carlo tree search to Kriegspiel. Monte Carlo tree search has been imposing itself over the past years as a major tool for games in which traditional minimax techniques do not yield good results due to the size of the state space and the difficulty of crafting an adequate evaluation function. The game of Go is the primary example, albeit not the only one, of a tough environment for minimax where Monte Carlo tree search was able to improve the level of computer programs considerably [6,7]. Since Kriegspiel shares the two traits of being a large game and a difficult one to express with an evaluation function (unlike its complete information counterpart), it is only natural to test a similar approach.

The paper is organized as follows. Section 2 contains a high-level introduction to Monte Carlo tree search (MCTS), with an emphasis on its successful application to Phantom Go. In Section 3, we introduce the game of Kriegspiel, its rules, and what makes it similar, yet very different, to Phantom Go. Section 4 contains the most significant research results on Kriegspiel, especially those related to previous Monte Carlo methods. We give a high-level view of three MCTS approaches in Section 5, showing how they are similar and where they differ; the corresponding programs are then described in greater detail separately. Section 6 contains some experimental tests comparing the strength and the performance of the various programs. Finally, we give our conclusions and some future research directions in Section 7.

2. Monte Carlo tree search

Monte Carlo tree search (MCTS) is an evolution of some simpler and older methods based on Monte Carlo sampling. While the core concept is still the same – a program plays a large number of random simulated games and picks the move that seems to yield the highest victory ratio – the purpose of MCTS is to make the computation converge to the right value much more quickly than pure Monte Carlo. This is accomplished by guiding the simulations with a game tree that grows to accommodate new nodes over time; more promising nodes are, in theory, reached first and visited more often than nodes that are likely to be unattractive.

MCTS is an iterative method that performs the same four steps until its available time runs out. These steps are summarized in Fig. 1.

- **Selection.** The algorithm selects a leaf node from the tree based on the number of visits and their average value.
- **Expansion.** The algorithm optionally adds new nodes to the tree.
- **Simulation.** The algorithm somehow simulates the rest of the game one or more times, and returns the value of the final state (or their average, if simulated multiple times).
- **Backpropagation.** The value is propagated to the node's ancestors up to the root, and new average values are computed for these nodes.

After performing these phases as many times as time allows, the program chooses the root's child that has received the most visits and plays the corresponding move. This may not necessarily coincide with the node with the highest mean value. A discussion about why the mean operator alone does not make a good choice is contained in [8].

MCTS should be thought of as a method rather than a specific algorithm, in that it does not dictate hard policies for any of the four phases. It does not truly specify how a leaf should be selected, when a node should be expanded, how simulations should be conducted or how their values should be propagated upwards. In practice, however, game-playing programs tend to use variations of the same algorithms for several of the above steps.

Selection as a task is similar in spirit to the n-bandit problem since the program needs to strike a balance between exploration (devoting some time to new nodes) and exploitation (directing the simulations towards nodes that have shown promise so far). For example, programs can make use of the UCT algorithm (Upper Confidence bound applied to Trees) first given in [6]. This algorithm chooses at each step the child node maximizing the quantity

$$U_i = v_i + c\sqrt{\frac{\ln N}{n_i}},$$

where v_i is the value of node i, N is the number of times the parent node was visited, n_i is the number of times node i was visited, and c is a constant that favors exploitation if low, and exploration if high.

Expansion varies dramatically depending on the game being considered, its state space and branching factor. In general, most programs will expand a node after it has been visited a sufficient number of times. Simulation also depends wildly on the type of game. There is a large literature dealing with MCTS simulation strategies for the game of Go alone. Backpropagation offers the problem of which backup operator to use when calculating the value of a node.

2.1. MCTS and partial information in Phantom Go

Monte Carlo tree search has been used successfully in large, complex partial information games, most notably Phantom Go. This game is the partial information version of the classic game of Go: the player has no direct knowledge of his opponent's stones, but can infer their existence if he tries to put his own stone on an intersection and discovers he is unable to. In that case, he can try another move instead. [1] describes an MCTS algorithm for playing the game, obtaining a good playing strength on a 9×9 board. A thorough comparison of several Monte Carlo approaches to Phantom Go, with or without tree search, has recently been given in [9]. We are especially interested in Phantom Go because its state space and branching factor are much larger than most other (already complex) partial information games such as poker, for which good Monte Carlo strategies exist; see, for example, [10].

MCTS algorithms for Phantom Go are relatively straightforward in that they mostly reuse knowledge and methods from their Go counterparts: in fact, they mostly differ from Go programs because in the simulation phase the starting board is generated with a new random setup for the opponent's stones every time instead of always being the same. It is legitimate to wonder whether this approach can be easily converted to other games with an equally huge state space, or Phantom Go is a special case, descending from a game that is particularly suited to MCTS. In the next section we discuss Kriegspiel, which is to chess what Phantom Go is to Go, and compare the two games for similarities and differences.

3. Kriegspiel

Kriegspiel, named after the 'war game' used by the Prussian army to train its officers, is a chess variant invented at the end of the XIX century to transform standard chess into a wargame. It has been studied and played by game theorists of the caliber of John von Neumann and Lloyd Shapley. It is played on three different chessboards, one for either player and one for the referee. They are positioned in such a way that the referee sees all the boards while the players can see only their own. From the referee's point of view, a game of Kriegspiel is a game of Chess. The players, however, can only see their own pieces while the opponent's are in the dark, as if hidden by a "fog of war". On his turn, a player selects a move and communicates it to the referee, so that there is no direct communication between the two opponents. If a move is illegal, the referee will reject the move and ask the player to choose a different one. If it is legal, the referee will instead inform both players as to the consequences of that move, if any. This information depends on the Kriegspiel variant being played; see [11] to find out more about the game. On the Internet Chess Club, which hosts the largest community of players of this game, the referee's messages are the following.

- When the move is legal and it is not a check or a capture, the referee will give no information, saying "White moved" or "Black moved". We will call this "silent referee" because no information is given to the players when a move is accepted.
- When a chessman is captured: in this case the referee will say whether the captured chessman is a pawn or a piece and where it was captured, but in the latter case he will not say what kind of piece.
- When the king of the player to move is in check: in this case the referee will disclose the direction (or directions) of check among the following: rank, file, short diagonal, long diagonal, knight.
- In order to speed up the game, when the player to move has one or more capturing moves using his pawns, the referee will announce that ("pawn tries") but will not tell which pawn can perform the capture.
- When the game is over, the referee announces the checkmate or drawn game for any standard condition (e.g. stalemate or not enough material or position repetition for three times, etc.).

These rules are also used for the international Computer Olympiad, where a Kriegspiel tournament has been played in 2006 and 2009.

On a superficial level, Kriegspiel and Phantom Go are quite similar. Both keep the same rules as their complete information versions, only adding a layer of uncertainty in the form of the fog of war managed by a referee. The transcript of a Kriegspiel game is a legal chess game, just like the transcript of a Phantom Go game is a legal Go game. Both involve move attempts as their core mechanics; illegal attempts provide information on the state of the game. In both games, a player can purposely try a move just for the purpose of information gathering. On the other hand, there are differences worth mentioning between the two games. We list some of the most significant ones.

- The nature of Kriegspiel uncertainty is strongly dynamic: while Go stones are, if not immutable, at least largely static and once discovered permanently decrease uncertainty by a large factor, information in Kriegspiel ages and quickly becomes old. One needs to consider whether uncertainty means the same thing in the two games, and whether Kriegspiel is a harsher battlefield in this respect.
- There are several dozen combinations of messages that the Kriegspiel referee can return, compared to just two in Phantom Go. This makes their full representation in the game tree very difficult.
- In Phantom Go there always exists a sequence of illegal moves that will reveal the full state of the game and remove uncertainty altogether; no such thing exists in Kriegspiel, where no sequence of moves can ever reveal the referee's chessboard except near the end of the game.
- Uncertainty grows faster in Phantom Go, but also decreases automatically in the endgame. By contrast, Kriegspiel uncertainty only decreases permanently when a piece is captured, which is rarely guaranteed to happen.
- In Phantom Go, the player's ability to reduce uncertainty increases as the game progresses since there are more enemy stones, but the utility of this additional information often decreases because less and less can be done about it. It is exactly the opposite in Kriegspiel: much like in Battleship, since there are fewer enemies on the board and fewer allies to hit them with, the player has a harder time making progress, but any information can give him a major advantage.
- There are differences carried over from their complete information counterparts, most notably the victory conditions. Kriegspiel is about causing an event that can happen suddenly and at almost any time, whereas Go games are concerned with the accumulation of score. From the point of view of Monte Carlo methods, score-based games tend to be more favorable than condition-based games, if the condition is difficult to observe in a random game. Even with considerable material advantage, it is relatively rare to force a checkmate with random moves.

Hence, there are mixed results from comparing the two games; at the very least, they represent two different kinds of uncertainty, that could be best described as static vs. dynamic uncertainty. We wish to investigate the effectiveness of Monte Carlo methods – and especially MCTS – in the context of dynamic uncertainty.

4. Related works

Research on Kriegspiel can be classified as follows. The earliest papers of the 1970s dealt with protocols for implementing a Kriegspiel referee and are not of interest here. Research in the following decades focused on solving specific subsets of the Kriegspiel game, most notably a few simple endgames. The first serious programs for playing an entire game of Kriegspiel came out around 2005. We discuss four of these, with a special emphasis on two: a Monte Carlo one and a minimax-based one.

4.1. Early results

Because the information shared by the players is very limited, the information set for Kriegspiel is huge. If one considers the number of distinct belief states in a game of Kriegspiel, the KRK (king and rook versus king) ending alone has a number of possible states close to the whole game of checkers. However, many of these states are in practice equivalent since there is no strategy that allows to distinguish them in a normal game. This complexity is the primary reason why, for a long time, research only focused on algorithms for specific endings, such as KRK, KQK or KPK. Ferguson showed game-theoretic algorithms for solving KBNK and KBBK under specific starting conditions; see, for example, [12]. Automatic search through four Kriegspiel endgames was first tackled in [13].

State reconstruction routines are the main object of [14]; [15] focuses on efficient belief state management in order to recognize and find Kriegspiel checkmates. The focus of both papers is on search and problem-solving rather than actually playing the game.

4.2. The first Monte Carlo program

Due to the complexity of the domain, computer programs capable of playing a full Kriegspiel game have only emerged in recent years. The first Monte Carlo (though not MCTS) approach to Kriegspiel is due to Parker, Nau and Subrahmanian [3]. Their program plays by using and maintaining a state pool that is sampled and evaluated with a chess function. In the paper, the authors call the information set associated with a given situation a *belief state*, the set containing all the possible game

states compatible with the information the player has gathered so far. They apply a statistical sampling technique, which has proven successful in several partial information games such as bridge and poker, and adapt it to Kriegspiel. The technique consists of generating a set of sample states (i.e. chessboards, a subset of the information set/belief state), compatible with the referee's messages and analyzing them with well-known complete information algorithms and evaluation functions. This approach feeds the randomly sampled boards to the popular open source GNUChess engine and chooses the move that obtains the highest average score in each sample. The choice of using a chess engine is both the method's greatest strength, as it saves the trouble of defining Kriegspiel domain knowledge, and its most important flaw, as positions are evaluated according to chess standards, with the assumption that each player can see the whole board.

Obviously, in the case of Kriegspiel, generating good samples is far harder than anything in Bridge or Poker. Not only is the state space immensely larger, but also the duration of the game is longer, with many more choices to be taken and branches to be explored. For the same reasons, evaluating a move is computationally more expensive than a position in Bridge, and the program needs to run the evaluation function on each sample; as a consequence, fewer samples can be analyzed even though the size of the state space would command many more.

The authors in [3] describe four sampling algorithms, three of which they have implemented (the fourth, AOS, generating samples compatible with all observations, would equate to generating the whole information set, and is therefore intractable).

- **LOS** (Last Observation Sampling). Generates up to a certain quantity of samples compatible with the last observation only (it has no memory of what happened before the last move).
- **AOSP** (All Observation Sampling with Pool). The algorithm updates and maintains a pool of samples (chessboards), numbering about a few tens of thousands, all of which are guaranteed to be compatible with all the observations so far.
- **HS** (Hybrid Sampling). This works much like AOSP, except that it may also introduce last-observation samples under certain conditions.

They conducted experiments with timed versions of the three algorithms, basically generating samples and evaluating them until a timer runs out, for instance after 30 seconds. They conclude that LOS behaves better than random play, AOSP is better than LOS, and HS is better than AOSP.

It may surprise that HS, introducing a component of the less refined LOS, behaves better than the pure AOSP, but it is in fact to be expected. The size of the AOSP pool is minuscule compared with the information set for the largest part of the game. No matter how smart the generation algorithm may be or how much it strives to maintain diversity, it is impossible to convey the full possibilities of a midgame information set (a fact we also confirm with the present research). The individual samples will begin to acquire too much weight, and the algorithm will begin to evaluate a component of noise. The situation worsens as the pool, which is already biased, is used to evolve the pool itself. Invariably, many possible states will be forgotten. In this context, LOS actually helps because it introduces fresh states, some of which may not in fact be possible, but prevents the pool from stagnating.

4.3. A minimax-based program

The first program based on minimax-like search, as well as the strongest before the present research, is described in our previous paper [4]; here we summarize its workings. The program builds an approximation of the game's information set based on the concept of *metapositions* as a tool for merging an immense amount of game states into a single, small and manageable data structure.

The term metaposition was first introduced by Sakuta [16], where it was applied to some endgame scenarios for the Shogi equivalent of Kriegspiel. The primary goal of representing an extensive form game tree with metapositions is to transform an imperfect information game into one of perfect information, which offers several important advantages and simplifications, including the applicability of the Minimax theorem. A metaposition merges different, but equally likely moves, into one state. A Kriegspiel metaposition can contain a huge number of states, and is obviously represented with an approximated version. We developed a specific function for evaluating a metaposition as a whole, ignoring the individual states that make it up. Then, we built a modified minimax algorithm for searching through metapositions. This leads to a good level of play (which in computer Kriegspiel translates to the level of an average human player), at the price of requiring a custom evaluation function for Kriegspiel and a good deal of domain knowledge. The program was called *Darkboard* and won the first computer Kriegspiel tournament held at the 11th Computer Olympiad in Turin, Italy, in 2006. It also played on the Internet Chess Club where it got a max Elo rating of 1870, but averaging around 1600 points.

An improved version of *Darkboard* with a better evaluation function is used as a benchmark for the Monte Carlo algorithms in this paper.

4.4. Other programs

Recently, there have been two attempts at modeling an opponent in Kriegspiel with Markov decision processes in the limited case of a 4×4 chessboard in [17]. The authors then shifted to a Monte Carlo approach (though not MCTS) with

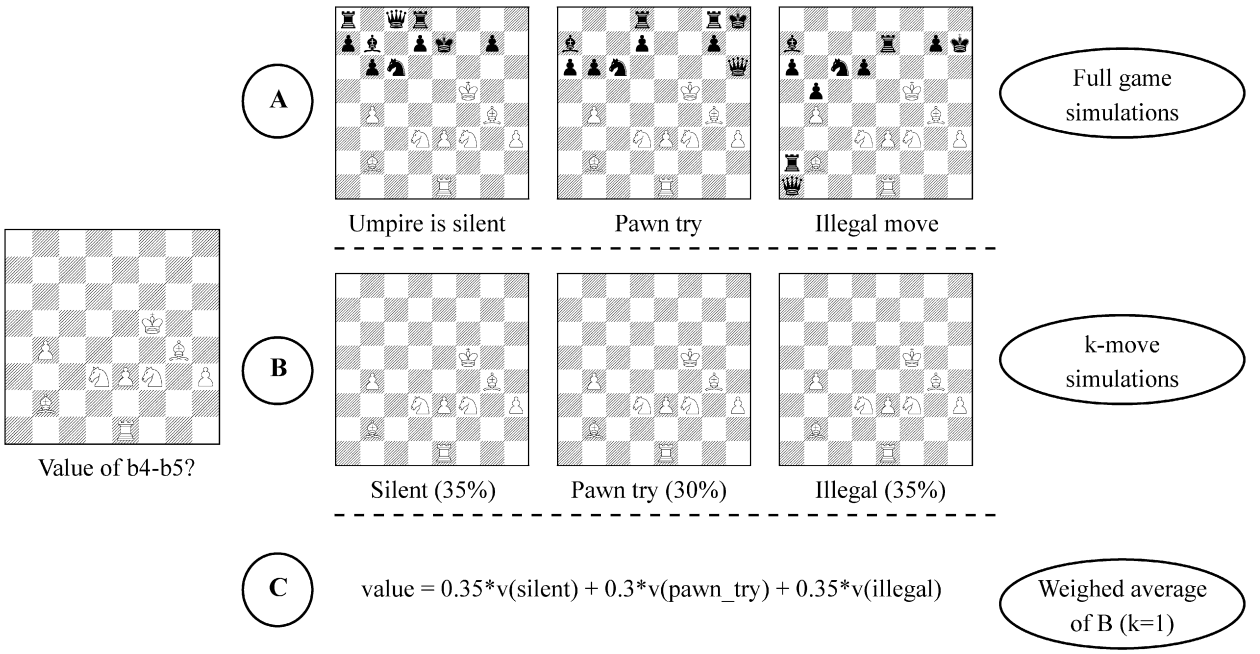

Fig. 2. Comparison of three simulation methods. Approach A is standard Monte Carlo tree search, approach B simulates referee messages only and for k-move runs, approach C immediately computes the value of a node in approach B for $k = 1$.

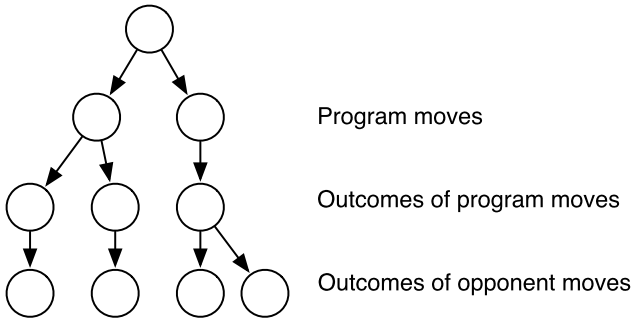

Fig. 3. Three-tiered game tree representation in our algorithms.

particle filtering techniques in [18]; this newer method allows the program to play on a normal 8×8 board. The latter work has some similarities, at least in spirit, with the modeling techniques presented in this paper; however, it is still similar to [3] in that the particle filtering creates plausible chess positions which are evaluated by an engine like GNUChess.

5. Three MCTS approaches

In this section, we provide three Monte Carlo tree search methods for playing Kriegspiel, which we label A, B and C. These approaches are summarized in Fig. 2 and can be briefly described as follows. Approach A is a MCTS algorithm that stays as faithful as possible to previous literature, in particular to existing Phantom Go methods. In this algorithm, a possible game state is generated randomly with each simulation, moves are random as well and games are simulated to their natural end. Approach B is an evolution of MCTS in which the program does not try to generate the opponent's board; instead, only the referee's messages are simulated. In other words, games are simulated from a player's partial point of view instead of the referee's omniscient one. Approach C is a simplification of approach B in which the algorithm can explore more nodes by cutting the simulation after just one move.

These three programs share major portions of code and implementation, in particular making use of the same representation for the game tree, shown in Fig. 3. As there are thousands of possible opponent moves depending on the unknown layout of the board, we resort to a three-level game tree for each two plies of the game, two of which represent referee messages rather than moves. The first two layers could be merged together (program moves and their outcomes), but remain separate for computational ease in move selection.

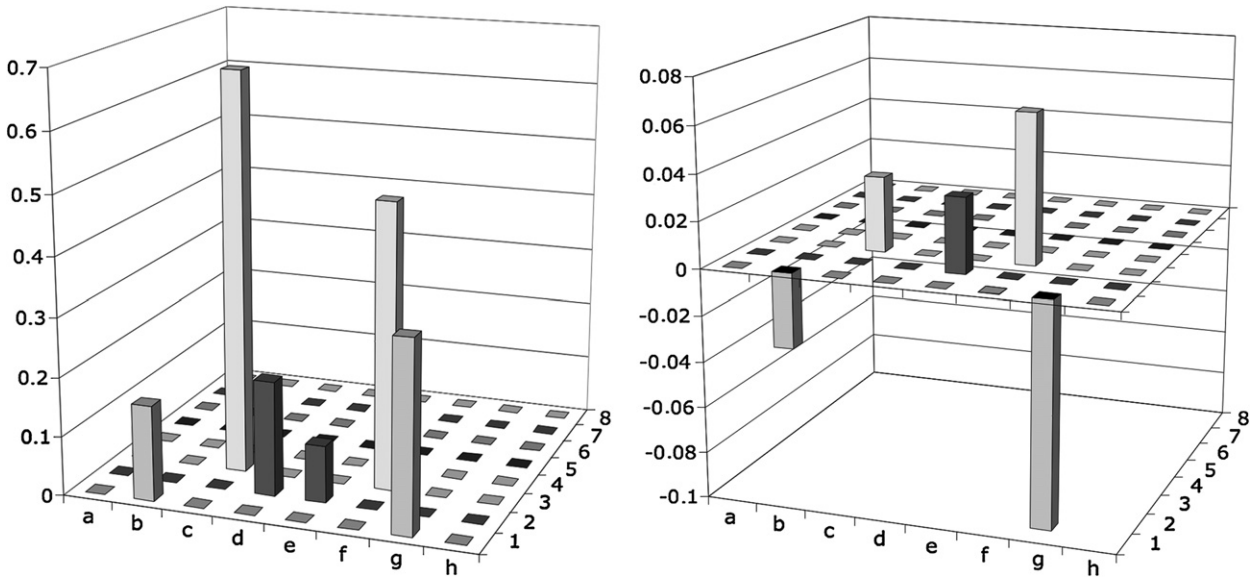

Fig. 4. Database data for handle "paoloc" playing as White, $t = 10$, $p =$ knight, both as absolute probabilities and delta values from move 9.

Initially, we investigated an approach that was as close as possible to the MCTS techniques developed for Go and its imperfect information variant, taking into account the important differences between these games and Kriegspiel. We developed the other two methods after performing several unsuccessful tests in which approach A could not be distinguished from a player moving randomly.

The three approaches all use some profiling data taken from a database of about 12 000 games played by human players on the Internet Chess Club. Because information is scarce, some kind of opponent modeling is an important component of a Kriegspiel program. Our algorithms make use of information from the game database in order to build an opponent's model, either for a specific adversary or for an unknown one: the unknown opponent is considered to be an averaged version of all the players in the database. We will therefore suppose that we have access to two 8×8 matrices $D_w(p,t)$ and $D_b(p,t)$ estimating the probability distribution for piece p at time t when our opponent is playing as White and Black, respectively. These matrices are available for all t up to a certain time when they are deemed too noisy to be of any practical value. Of course, their values can be smoothed by averaging them over several moves or even over neighboring squares, especially later in the game.

These matrices can contain truly vital information, as shown in Fig. 4. Ten moves (twenty plies) into the game, the locations of this player's knights can be inferred with high probability. This is no coincidence: in the almost total absence of information most players will use the same tested strategies over and over again, making them easier to predict. These matrices are used in different ways by our algorithms: approach A uses absolute probabilities (the unmodified values of D_w and D_b) in order to reconstruct realistic boards for MCTS sampling purposes, whereas approaches B and C exploit gradient values, that is, the values of $D(p,t+1) - D(p,t)$ in order to evolve their abstract model from one move to the next.

5.1. Approach A

Pseudocode for approach A is shown in Fig. 5. Our approach A implements the four steps of MCTS as follows.

- Selection is implemented with UCT for the program's own moves, as seen in the pseudocode: the opponent plays the same pseudorandom moves as in the Simulation step. Choosing different values for the exploration constant c did not have any impact on performance. [9] showed that there are two main methods for guessing the opponent's unknown stones in Phantom Go: late random opponent-move guessing and early probabilistic opponent-move guessing. In the former, some stones are added as the opponent plays them and the rest are filled just before the simulation step; in the latter, stones are added after the first move based on their frequency of play during the first move. It is noted that early guessing outperforms late guessing.
  The concept of move is very different in Kriegspiel, so we would not be able to easily build and use frequency statistics in the same way. Nevertheless, recognizing the power of early guessing, we fill the entire board before we even start Selection (note, however, that the tree does not contain boards, but only referee's messages which are used to traverse it; the tree never deals with specific boards). We used the probability distributions D_w and D_b discussed in the previous section. They were collected from a database of online games to estimate density for each piece type at any given point in time. The matrices, including completely *a priori* knowledge, are not the only information used by the algorithm; several heuristics helped to construct the random boards, such as remembering how many pieces are left in play, and

```
function approach_A(Node root) {
    while (availableTime) {
        Board b = generateRandomBoard(root);
        Node n = root;
        Move move;
        while (!isLeaf(n)) {
            if (programTurn(n)) {
                n = uctSelection(n,legalMoves(b),refereeMessage(b,move));
                move = n.move;
            }
            else {
                move = getPseudoRandomMove(b);
                n = getChild(n,refereeMessage(b,move));
            }
            playMove(b,move);
        }
        n = expand(n);
        double outcome = simulation(b);
        backpropagate(outcome,n);
    }
    return mostVisitedChild(root);
}
```

Fig. 5. Pseudocode for approach A.

how many pawns can be on each file. The generator also strived to make positions that did not contradict the last referee's message, as Last Observation Sampling was reported to yield the best results in [3] when applied to the same task.

- We implement Expansion by adding a new random node with each iteration. We considered this random choice to be a reasonable solution; we judged a custom Expansion heuristic to be remarkably difficult to devise in Kriegspiel. Choosing a new node for each simulation also allows to easily compare this approach to an evaluation function-based one exploring the same amount of nodes.
- Simulation raises a number of questions in a game of partial information, such as whether and how to generate the missing information, and how to handle subsequent moves. Existing research is of limited help, since to the best of our knowledge this is the first time MCTS is applied to a game in which knowledge barely survives the next move or two. Go is relatively straightforward in that one can play a random move anywhere except in one's own *eyes*. It is also easier to estimate the length of a simulated Go game, which is generally related to the number of intersections left on the board. Kriegspiel simulations are necessarily heavier to compute due to the rules of the game. Even generating the list of moves is a nontrivial task that requires special care.
  Our simulated players play pseudorandom moves until they draw by the fifty move rule or they reach a standard endgame position with a clear winner (such as king and rook versus king), in which case the game is adjudicated. In order to make the simulation more realistic, both players almost always try to capture back or exploit a pawn try when possible – this is basic and almost universal human behavior when playing the game, and is also shared by all our programs. In this sense the simulated moves are not random, but only pseudorandom.
- We implemented standard Backpropagation, using the average node value as backup operator.

As mentioned, approach A failed, performing little better than a random player and losing badly and on every time setting to a more traditional program based on minimax search. Program A's victory ratio was below 2%, and its victories were essentially random and unintentional mid-game checkmates. Investigating the reasons of the failure showed three main ones, in addition to the obvious slowness of the search.

First, the positions for the opponent's pieces as generated by the program are not realistic. The generation algorithm uses probability distributions for pieces, pawns and king that are updated after each referee message. While the probabilities are quite accurate, this does not account for the high correlation between different pieces, that is, pieces protecting other pieces. Kriegspiel players generally protect their pieces quite heavily, in order to maximize their chances of successfully repelling an attack. As a result, the program tends to underestimate the protection level of the opponent's pieces. Secondly, because moves are chosen randomly, it also underestimates the opponent's ability to coordinate an attack and hardly pays attention to its own defense.

Lastly, but perhaps most importantly, there is the subtler issue of *progress*, defined as decreasing uncertainty. Games where Monte Carlo tree search has been tested most thoroughly have a built-in notion of progress. In Go, adding a stone changes the board permanently, permanently decreasing the possible moves in the future. The same happens in Scrabble when a word is added to the board. Kriegspiel, on the other hand, like most wargames, has no such notion; if the players do nothing significant, nothing happens. In fact, it can be argued that many states have similar values and a program failing to find a good long-term plan will either rush a very risky plan or just choose to minimize the risk by moving the same

```
function approach_B(Node root, int k) {
    while (availableTime) {
        Node n = root;
        Move move;
        while (!isLeaf(n)) {
            if (programTurn(n)) {
                n = uctSelection(n);
                Message msg = probabilisticMessage(n);
                n = getChild(n,msg);
            }
            else {
                Message msg = probabilisticMessage(n);
                n = getChild(n,msg);
            }
        }
        n = expand(n);
        double outcome = simulation(n,k);
        backpropagate(outcome,n);
    }
    return mostVisitedChild(root);
}
```

Fig. 6. Pseudocode for approach B.

piece back and forth. When a Monte Carlo method does not perform enough simulations to find a stable maximum, it can do either.

It is unlikely for a mere implementation of MCTS techniques as seen in Go or Phantom Go to work effectively in Kriegspiel, at least under the resource constraints of current computer systems. In order for it to work, we would have to change the rules of the game or the players would need to receive more information on the state of the board.

5.2. Approach B

An effective Monte Carlo tree search Kriegspiel program needs to converge to a better value, not to mention more quickly, than the implementation presented in approach A. Reducing the major amount of noise in the simulation step is also of paramount importance. As seen, performing Monte Carlo search on individual states, as standard MCTS would dictate, leads to highly unstable results. A possible solution could lay in running simulations but not on individual game states – rather, on their *perception* from the computer player's point of view. This would save us the trouble, both computational and algorithmic, of generating plausible game states that reward intelligent play in simulations.

The core spirit of Monte Carlo methods is preserved by running the simulations as usual, but instead of running them as chess games with complete information, they would be run as Kriegspiel games with partial information. As an aside, simulating an abstract model of the game instead of the game itself has already been done in the context of Monte Carlo programs; for example, [19] does so with a real-time strategy game, for which a detailed simulation over continuous time would be impossible. What the authors do instead is simulating high-level system responses to high-level decisions and strategies, and this is conceptually close to our own goal.

We therefore define our program B, whose pseudocode is listed in Fig. 6. This approach removes the randomness involved in generating single states and instead only simulates the referee messages, without worrying about the enemy layout that generated them. A reduced version of the abstract model used in approach A estimates the likelihood of a given referee message in response to a certain move.

Our model is very utilitarian. For example, there is a chance of the enemy retaliating on a capture one or more times and a chance of a move being illegal. At core, this is based on three 8×8 piece probability matrices *Pk* (king), *Pw* (pawn) and *Pc* (other chessman). P_{ij} contains the probability of a piece of the given type being on square (i, j), with rank 0 being White's first rank and the opponent being Black. We do not distinguish between different pieces such as queens and rooks as most Kriegspiel rulesets do not give a player enough information to do so. In approach A, the same matrices are used to generate random chessboards, but here they serve their purpose directly in `probabilisticMessage`: they determine the probabilities with which referee's messages are picked in response to a move (UCT still selects the move).

We make two sets of assumptions. The first set models the rules of chess to predict the outcomes of the program's own moves from the probability matrices for the opponent's pieces. It also updates the probabilities with the knowledge gained from the referee's responses to the program's moves. The second set provides our opponent model, updating the opponent's probabilities when it is his turn to move and deciding the outcomes of his moves. In other words, the first set of assumptions is nothing more than probability theory applied to chess; the second set is, in fact, an opponent model.

The first set is as follows:

- The probability for the opponent to control a square (i, j) is equal to a sum of components

$$Prob_{control}(i, j) = \sum_{dist(x,y,i,j)=1} Pk_{xy} + Pw_{i-1,j+1} + Pw_{i+1,j+1} + c_1 \sum_{x,y} c_2 Pc_{xy},$$

  meaning the sum of probabilities for the king in the surrounding squares, a pawn in the compatible diagonal squares, and all squares on the same rank, file and diagonals multiplied by suitable coefficients. Here, $c_1 = 3/7$ since at most three out of seven pieces in the starting set other than the king and pawns are able to attack along any given direction: queen and rooks for ranks and files, and queen and bishops for the diagonals. The only exception is the knight check, which only two pieces can perform. c_2 is calculated dynamically so that enemy pieces covering each other are accounted for; basically, c_2 decreases as the distance to (i, j) increases. $Prob_{control}$ can be greater than 1; in fact, it should be read as the expectation for the number of enemy pieces controlling the square.
- The probability for a move to be legal is equal to the probability of all squares on the piece's path Pt from (i_1, j_1) to (i_2, j_2) (except the destination square itself unless it is a straight pawn move) being empty, minus a pin probability. We recall that a piece is pinned if moving it would leave the king in check. That is,

$$Prob_{legal}(Pt) = \prod_{(i,j)\in Pt} (1 - Pk_{ij} - Pw_{ij} - Pc_{ij}) - Prob_{pin},$$

  where

$$Prob_{pin} = \begin{cases} 0 & \text{if the piece is not protecting the king,} \\ Prob_{control}(i_1 j_1) & \text{if the piece is protecting the king,} \\ Prob_{control}(i_2 j_2) & \text{if the piece is the king.} \end{cases}$$

  This, while approximated, accounts for a number of cases, including pieces being pinned by unknown enemy pieces and the king being unable to move to a threatened square.
- The probabilities of capturing a piece or pawn on (i_2, j_2) are equal to $Pc_{i_2 j_2}$ and $Pw_{i_2 j_2}$, respectively.
- The probability of the program causing a check is equal to a sum of Pk_{ij} over the squares threatened by the move, again with a damping coefficient c_2 designed to reduce the impact of far away squares.
- When a square is found to be empty by moving through it or due to a lack of pawn tries, the probabilities for the enemy pieces on that square are set to 0. Conversely, when a square is known to be occupied (usually because of a capture), the sum of the probability matrices for that squares is brought to 1. In both cases, the matrices are normalized afterwards so their total sum over the board does not change.

The second set contains the following assumptions suggested by human play observed on the Internet Chess Club:

- When the program captures something, there is a very high chance of the capturing piece being, in turn, captured by the opponent. This reflects the fact that most pieces are always protected. Long chains of blind sacrifices are common in Kriegspiel: for the second and subsequent captures, the program uses $Prob_{control}$ to determine whether there is retaliation.
- When a check message is heard there is a chance, assumed to be constant in our model, that the checking piece is captured. Human players often try to capture the offending piece as their first reaction to a check. In particular, a player has nothing to lose from probing the check's direction with his king.
- When the opponent moves, there is a fixed chance to suffer a capture. The victim is chosen at random, with the probability of capture being directly proportional to $Prob_{control}$ so that more exposed pieces are captured with higher probability.
- All pieces stand a more or less equal chance of being moved by the opponent; if the program knows that the opponent has k_1 pawns and k_2 pieces left, the probabilities of the king, a pawn, or a piece being moved are, respectively,

$$P_{king} = \frac{1}{k_1 + k_2 + 1}, \qquad P_{pawn} = \frac{k_1}{k_1 + k_2 + 1}, \qquad P_{piece} = \frac{k_2}{k_1 + k_2 + 1}.$$

- The enemy king's movement is modeled as a random walk over a graph corresponding to the set of permissible squares.

$$Pk_{ij}(t+1) = (1 - P_{king}) Pk_{ij}(t) + P_{king} \sum_{i-1\leqslant x\leqslant i+1} \sum_{j-1\leqslant y\leqslant j+1} f_{king}(x, y, t) Pk_{xy}(t),$$

  where f is a suitable function that scales and centers the probability delta values gathered from the game database discussed in Section 5, so that their sum is 1. This function makes use of D_w or D_b, depending on whether the opponent is White or Black. The rationale behind using delta values from the previous move instead of directly comparing the values of D is that delta values represent trends rather than snapshots, and seem to be more likely to carry over even during atypical games.

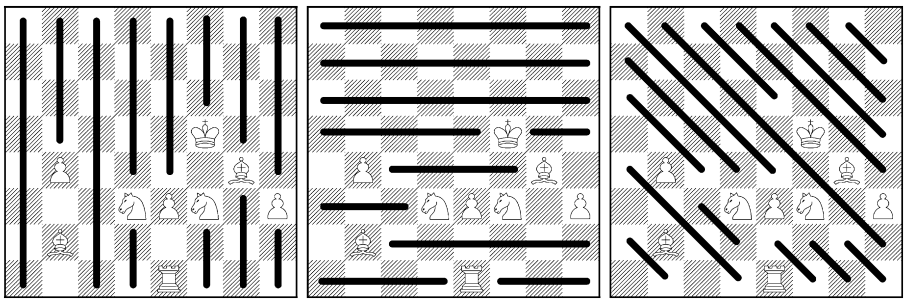

Fig. 7. Density spreading routine in approaches B and C (second diagonal sweep not shown).

- Pawns are modeled separately as one-way Markov chains.
- A generic piece other than a pawn or king is the most complex to model. The computational burden of calculating a custom transition matrix for each chessboard (as its values would change depending on board layout) and discovering which squares can affect which would be too high: MCTS relies on speed and number of simulations. Instead, the board is scanned along several directions, as shown in Fig. 7. Whenever a group of two or more empty squares is found, the program runs a fast random walk update over those squares, still using function f and the database data as long as it is available. If the database is not active or the game has reached a point where it is no longer useful, all squares become equally attractive.

$$Pc_{ij}(t+1) = (1 - P_{piece})Pc_{ij}(t) + c_1 P_{piece} \sum c_2 P_{ij}(t),$$

with c_1 and c_2 indicated as different constants for exposition's sake. c_1 is again a piece probability factor, as not all pieces can move along a given direction. It is also a generic adjustment factor; it indicates the probability of finding a piece that can move as desired and is willing to. In the current implementation, $c_2 = \frac{1}{k-1}$, where k is the number of squares in the sequence.

While this algorithm can help improve performance and run more simulations than approach A can in the same amount of time, the real advantage is that the opponent no longer plays randomly in the simulations. Instead, it plays according to some average, realistic expectations while the actual moves are never disclosed. This is very close to the way a human Kriegspiel player plans his moves.

A second point of interest about method B is that it does not play full games as that proved to be too detrimental to performance in approach A. Instead, it simulates a portion of the game that is at most k moves long (k is passed as a parameter). The algorithm also accounts for quiescence, and allows simulations to run past the limit of k moves after its starting point in the event of a string of captures. The first move is considered to be the one leading to the tree node where simulation begins; as such, when $k = 1$, there is basically no exploration past the current node except for quiescence. Intuitively, a low value of k gives the program less foresight but increases the number of simulations and as such its short term accuracy; a high value of k should do the opposite. At the end of the simulated snippet, the resulting chessboard is evaluated using the only real notion of Kriegspiel theory in this method; that basically reduces to counting how many pieces the player has left, minus the number of enemy pieces left.

5.3. Approach C

The third and final approach, called C and shown in Fig. 8, is approach B taken to the extreme for $k = 1$; it was developed after noticing the success of that value of k in the first tests. There is a tendency already noticed in Kriegspiel literature, first in [3] and then in [18], for myopic searches to outperform their far-sighted counterparts. If anything, using $k = 1$ offers a tremendous performance boost, as each node needs only be sampled once. Since the percentages for each referee message are known in the model, it is easy to calculate the results for each and obtain a weighed average value. As seen in the pseudocode, the function getOutcomeProbabilities interrogates the referee simulator on the probabilities of a given outcome taking place from the penultimate to the latest explored node. Each outcome has a progress value identical to approach B's and equal to the number of allied pieces on the board.

Approach C makes the bold assumption that the value estimated with approach B's abstract model for $k = 1$ is the truth, or at least as close to the truth as one can get. Because simulations are assumed to instantly converge through the weighed average, the backup operator is also changed from the average to the maximum node value. Of course, this is the fastest simulation strategy, blurring the line between simulation and a UCT-driven evaluation function (or, more accurately, a cost function in a pathfinding algorithm), and it can be very discontinuous from one node to the next. If approach C is successful, it means that information in Kriegspiel is so scarce and of such a transient nature that the benefits of global exploration by simulating longer games are quite limited compared to the loss of accuracy in the short run. This emphasizes selection strategies over simulation strategies. Another way to think of approach C is as if simulations happened entirely on the tree

```
function approach_C(Node root) {
    while (availableTime) {
        Node n = root;
        Move move;
        while (!isLeaf(n)) {
            if (programTurn(n)) {
                n = uctSelection(n);
                Message msg = probabilisticMessage(n);
                n = getChild(n,msg);
            }
            else {
                Message msg = probabilisticMessage(n);
                n = getChild(n,msg);
            }
        }
        if (n.explored) n = expand(n);
        double outcomeValues[ ], probabilities[ ], value;
        getOutcomeProbabilities(n,outcomeValues,probabilities);
        for (int a=0; a<outcomeValues.length; a++)
            value += outcomes[a] * probabilities[a];
        n.explored = true;
        backpropagate(outcome,n);
    }
    return mostVisitedChild(root);
}
```

Fig. 8. Pseudocode for approach C.

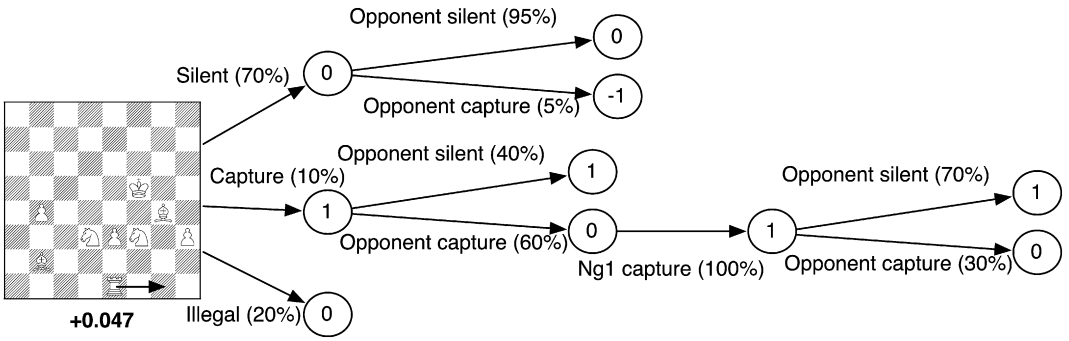

Fig. 9. Example of a C simulation step (simplified).

itself rather than in separate trials, at the rate of one simulation per node. This is based on the assumption that good nodes are more likely to have good children, and the best node usually lies at the end of a series of good or decent nodes.

Fig. 9 shows an example of how approach C might simulate Rg1 on the sample board. The actual program would consider more referee messages than those listed. The nodes contain the material balance at that time: it is positive when the player has captured more pieces than he lost. The weighed average of the leaf nodes amounts to 0.047, which is then backpropagated. The program handles quiescence moves, as seen in the Ng1 retaliatory move if the rook is captured.

6. Tests

6.1. Tests versus a minimax program

We tested our approaches, with the exception of A which is not strong enough to be interesting, against an improved version of our program Darkboard described in [4], which we call "minimax program". Tests versus humans on the Internet Chess Club showed that Darkboard's playing strength is reasonable by human standards, ranking at club level above average (around 1600 Elo points). The program used in our tests is even slightly stronger than the aforementioned one, since it performs a series of hard-coded checks that prevent the program from making obvious blunders. It should be noted that our MCTS programs do not include these checks.

The evaluation function of the minimax program is rather complex and domain-specific, consisting of several components including material, positional and information bonuses. By contrast, our MCTS programs know very little about Kriegspiel: both B and C only know that the more pieces they have, the better. They know nothing about protection, promoting pawns, securing the center or gathering information trying moves which are likely to be illegal.

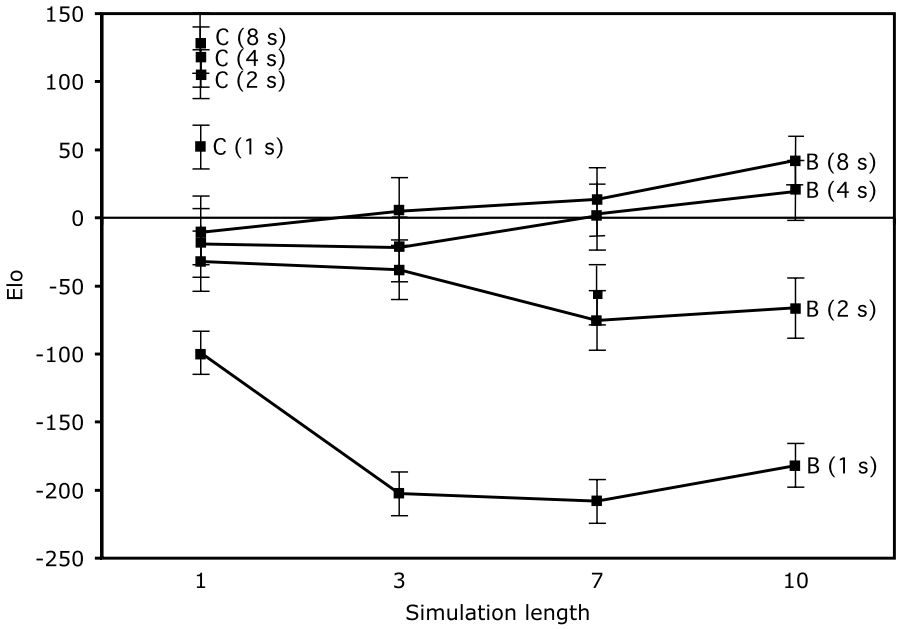

Fig. 10. Comparison of MCTS approaches B and C with a fixed-depth minimax program at different time settings and simulation depths, with error intervals.

The experimental setup is as follows, and the results of the tests are summarized in Fig. 10. We played 16 instances of B and 4 of C against the minimax program (A was judged too weak to test). The instances of B are the combinations of four time settings (1, 2, 4 and 8 seconds per move) with four values of k: 1, 3, 7 and 10. The instances of C correspond to the time settings alone as k is by definition 1 in this approach. The minimax program always plays under the same settings found to be optimal for its algorithm. It explores approximately 10 000 metapositions per move; the presence of a pruning algorithm ensures a search depth of 3 to 5 moves depending on the situation. The minimax program requires between 1 and 2 seconds to run under these settings; the program hits a performance plateau after this point, preventing it from further increases in strength. In other words, this was the strongest version of Darkboard we could ready for use.

The test took place on an eight-core Mac Pro machine, with two sets of tests running simultaneously over three weeks. The MCTS programs can take advantage of multiple processors thanks to a form of root parallelism, whereas the minimax program does not. We ran a variable number of games for each tested instance, ranging from a minimum of 1500 for the 8-second tests to a maximum of 4000. The brackets in the figure represent a 95% confidence interval for the data, expressed as Elo ratings. Programs that perform worse than the minimax player are below 0 in the graph whereas the better programs are above 0. We recall that in the Elo rating system a difference of 200 points corresponds to an expected result of about 0.75 (with 1 being a win and 0.5 being a draw), and a difference of 400 points has an expected result of about 0.9.

The MCTS programs do not have particular optimizations and their parameters have not been thoroughly fine-tuned. The programs are all identical outside the simulation task, with the single exception of the UCT exploration parameter c. Approach C uses a lower value of c leaning towards exploitation more on the basis that each node is only evaluated once. However, this different value of c has only a small beneficial value on the program: most of the advantage of the weighed average method lies in its speed, which allows it to visit many more nodes than the corresponding B program for $k = 1$. The speedup factor ranges from 10 to 20 on average, everything else being equal.

Experimental findings generally confirm our expectation, that is, the lower values of k should be more effective under faster time settings, and the higher values of k should eventually gain the upper hand as the program is given more time to reason. When k is low, the program can execute more simulations which make it more accurate in the short term, thus reducing the number of tactical blunders. On the other hand, given enough time the broader horizon of a higher k finds more strategic possibilities and longer plans through simulation that the lower k cannot see until they are encountered through selection.

At 1 second per move, $k = 1$ has a large advantage over the other B programs. Doubling the time reduces the gap among all programs, and at 4 and 8 seconds per move the longer simulations have a small but significant edge, actually outperforming the minimax program by a slight margin. The only disappointment came from the $k = 3$ programs, which did not really shine under any time setting. It is possible that three moves is just not enough to consistently generate good plans out of random tries. Since Kriegspiel plans can be interleaved with basically useless moves that more or less maintain the status quo on the board, a ten-move sequence can contain a good three-move sequence with higher likelihood.

Given the simplicity of the approach and the lack of specialized knowledge compared to the minimax program's trained parameters and pruning techniques, B programs are quite remarkable, though not as much as the performance of C type programs. These can defeat the benchmark program consistently, ranking over 100 Elo points above it and winning about

three times more games than they lose to it. Since approach C has basically no lookahead past the node being explored, we can infer that UCT selection is the major responsible for its performance, favoring the paths of least danger and highest reward under similar time settings to the minimax program's. The UCT exploration–exploitation method beats the hard pruning algorithms used by the minimax program, showing that in such a game as Kriegspiel totally pruning a node can often remove an interesting, underestimated line of play: there are relatively few bad nodes that can be safely ignored. It appears more profitable to allocate different amounts of time and resources to different moves, like in Monte Carlo tree search and the related n-armed bandit problem.

6.2. Tests versus humans

We collected more experimental evidence that approach C is effective by letting it play against humans over the ICC, where it reached an average of 1750 Elo points after playing 7121 games. The matches are significant as they all took place under the same time settings, 3 minutes without increment. These time settings are considered quite generous for Kriegspiel games on the ICC, meaning that the program does not win by just making its opponents time out. Out of all games, 2777 were played against a set of 20 players recognized as the strongest in the community. Program C's average score against these players is 0.26, whereas the minimax program's average score against the same players was 0.17. This is a further confirmation that there is a gap of about 100 Elo points between the old program and the new, with 200 more points to be gained before the computer can challenge a top human on equal terms.

Last but not least, approach C confirmed its playing strength by winning the gold medal with a perfect score in the Kriegspiel tournament held at the 14th Computer Olympiad in Pamplona, Spain, in May 2009.

7. Conclusions and future work

There are two main conclusions to be drawn from our experiments. First, they show that a Monte Carlo tree search algorithm can converge to good results in a reasonable amount of time even in a very difficult environment like Kriegspiel, whose lengthy simulations might at first appear to be a significant disadvantage of the method. However, the application of a Monte Carlo method must be carefully designed. In fact, we were unable to achieve a good level of play by generating random states and running simulations on them. Instead, we had to abstract the game with a model in which single states are not important, and only their perception matters.

The second conclusion is that approach C, in which simulations only last one move plus quiescence, performed the best, defeating a minimax-based program and faring well against human players. This seemingly counterintuitive result can be explained with the accuracy of algorithm C in simulating short-term tactical opportunities. The difference between B and C decreases as the program is allotted more simulation time, but never totally disappears. This may hint at inaccuracies in the assumptions behind the referee's simulated messages. The simulated games in B would therefore suffer more from these weaknesses.

Our program as a whole can still be improved by a large factor. It is possible to refine and compare the simulation assumptions for the referee starting from the current ones, for example through genetic algorithms or other adaptive methods. It is also possible to consider a hybrid approach between B and C; this method would simulate the first move like C and then launch longer simulations for the next moves. A similar method combining heuristics and simulation for Go is described in [20].

Many more improvements are possible. In the game of Go, Monte Carlo tree search is more and more often combined with game-specific heuristics that help the program in the Selection and Simulation tasks. Since Monte Carlo methods are relatively weaker when they are short on time, these algorithms drive exploration through young nodes when there is little sampling data available on them. Examples of such algorithms are the two progressive strategies described in [21]. Since Kriegspiel is often objective-driven when played by humans, objective-based heuristics are the most likely candidates to make good progressive strategies, and research is already underway in that direction. More sophisticated opponent modeling techniques also seem necessary to conquer the strongest opponents.

Acknowledgements

We thank the anonymous reviewers for their comments and suggestions which improved this paper.

References

[1] T. Cazenave, A Phantom Go program, in: J. van den Herik (Ed.), Proc. Advances in Computer Games, vol. 11, Taipei, Taiwan (revised papers), in: Lecture Notes in Computer Science, vol. 4250, Springer, 2005, pp. 120–126.
[2] J. von Neumann, O. Morgenstern, Theory of Games and Economic Behavior, Princeton University Press, 1944.
[3] A. Parker, D. Nau, V. Subrahmanian, Game-tree search with combinatorially large belief states, in: Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI05), Edinburgh, Scotland, 2005, pp. 254–259.
[4] P. Ciancarini, G. Favini, Representing Kriegspiel states with metapositions, in: Proc. 20th Int. Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India, 2007, pp. 2450–2455.
[5] M. Sakuta, H. Iida, Solving kriegspiel-like problems: Exploiting a transposition table, ICCA Journal 23 (4) (2000) 218–229.

[6] L. Kocsis, C. Szepesvari, Bandit based Monte-Carlo planning, in: Proc. European Conference on Machine Learning (ECML), in: Lecture Notes in Computer Science, vol. 4212, Springer, 2006, pp. 282–293.
[7] S. Gelly, Y. Wang, Exploration exploitation in Go: UCT for Monte-Carlo Go, in: NIPS: Proc. 20th Annual Conference on Neural Information Processing Systems, 2006.
[8] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in: Computers and Games, in: Lecture Notes in Computer Science, vol. 4630, Springer, 2007, pp. 72–83.
[9] J. Borsboom, J. Saito, G. Chaslot, J. Uiterwijk, A comparison of Monte-Carlo methods for Phantom Go, in: Proc. 19th Belgian–Dutch Conference on Artificial Intelligence – BNAIC, Utrecht, The Netherlands, 2007.
[10] D. Billings, A. Davidson, J. Schaeffer, D. Szafron, The challenge of poker, Artificial Intelligence 134 (2002) 201–240.
[11] C. Wetherell, T. Buckholtz, K. Booth, A director for Kriegspiel, a variant of chess, The Computer Journal 15 (1) (1972) 66–70.
[12] T. Ferguson, Mate with bishop and knight in Kriegspiel, Theoretical Computer Science 96 (1992) 389–403.
[13] A. Bolognesi, P. Ciancarini, Computer programming of Kriegspiel endings: The case of KR vs K, in: J. van den Herik, H. Iida, E. Heinz (Eds.), Advances in Computer Games, vol. 10, Kluwer, Graz, Austria, 2003, pp. 325–342.
[14] M. Nance, A. Vogel, E. Amir, Reasoning about partially observed actions, in: Proc. 21st National Conf. on Artificial Intelligence (AAAI '06), Boston, USA, 2006, pp. 888–893.
[15] S. Russell, J. Wolfe, Efficient belief-state AND–OR search, with application to Kriegspiel, in: Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI05), Edinburgh, Scotland, 2005, pp. 278–285.
[16] M. Sakuta, Deterministic solving of problems with uncertainty, PhD thesis, Shizuoka University, Japan, 2001.
[17] A. Del Giudice, P. Gmytrasiewicz, J. Bryan, Towards strategic Kriegspiel play with opponent modeling, in: AAMAS 09: Proc. 8th Int. Conf. on Autonomous Agents and Multiagent Systems, Budapest, Hungary, International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 265–1266.
[18] J. Bryan, P. Gmytrasiewicz, A. Del Giudice, Particle filtering approximation of Kriegspiel play with opponent modeling, in: AAMAS 09 Workshop on Multi-agent Sequential Decision-Making in Uncertain Domains, 2009.
[19] M. Chung, M. Buro, J. Schaeffer, Monte Carlo planning in RTS games, in: Proc. IEEE Symposium on Computational Intelligence and Games, 2005.
[20] S. Gelly, D. Silver, Combining online and offline knowledge in UCT, in: ICML 07: Proc. 24th Int. Conf. on Machine Learning, ACM Press, 2007, pp. 273–280.
[21] G. Chaslot, M. Winands, J. van der Herik, J. Uiterwijk, B. Bouzy, Progressive strategies for Monte-Carlo tree search, New Mathematics and Natural Computation 4 (3) (2008) 343–352.