Alma Mater Studiorum · Università di Bologna

DOTTORATO DI RICERCA IN INFORMATICA
Ciclo XXII

Settore scientifico-disciplinare INF/01

The dark side of the board: advances in chess Kriegspiel

Presentata da: Gian-Piero Favini

Coordinatore:
Prof. Simone Martini

Tutore:
Prof. Paolo Ciancarini

Esame finale anno 2010

To those who have
always
believed in me

Abstract

While imperfect information games are an excellent model of real-world problems and tasks, they are often difficult for computer programs to play at a high level of proficiency, especially if they involve major uncertainty and a very large state space. Kriegspiel, a variant of chess making it similar to a wargame, is a perfect example: while the game was studied for decades from a game-theoretical viewpoint, it was only very recently that the first practical algorithms for playing it began to appear. This thesis presents, documents and tests a multi-sided effort towards making a strong Kriegspiel player, using heuristic searching, retrograde analysis and Monte Carlo tree search algorithms to achieve increasingly higher levels of play. The resulting program is currently the strongest computer player in the world and plays at an above-average human level.

Acknowledgements

First and foremost, I would like to express my gratitude to my tutor, Prof. Paolo Ciancarini, for his support and guidance throughout the years. This thesis is the coronation of a long and fruitful period of collaboration starting in 2003, when I first heard the word Kriegspiel. What begun as a GUI design project for the game developed in directions I could never have foreseen. Two of the most exciting experiences in my life – winning as many gold medals at the Computer Olympiads – would not have been possible without him (among other things).

I would also like to thank all the other people who helped me in the making of this thesis, and especially the external referees, Professors Yngvi Björsson (University of Reykjavyk), Thomas Ferguson (University of California at Los Angeles) and Jos Uiterwijk (University of Maastricht), for their praise and constructive cricitism of my work. My gratitude also goes to the anonymous referees who reviewed the papers related to this thesis.

A special mention goes to the whole Computer Science Department of the University of Maastricht, where I spent four months in 2008. In addition to Prof. Uiterwijk, I would like to thank Prof. Jaap van der Herik and Johanna Hellemons (now of the University of Tilburg), as well as Dr. Mark Winands and the entire Ph.D. student body for their support.

Contents

Chapter 1

Introduction

If you know the enemy and know yourself,
you need not fear the result of a hundred battles.
Sun Tzu

Ever since their inception in the animal kingdom, games have been a metaphor of life, and often one of conflict. Pups engage in playful behavior to learn the tactics on which their survival will depend later in life. They hone the motions and teamwork they will need when they move on to hunting real preys. Over the millennia, humans have invested games with a multitude of meanings, ritualizing their conflicts from military, social and religious standpoints. Without a doubt, games have always been serious business. The game of Senet, depicted in several Egyptian tombs and considered the most ancient example uncovered by archaeologists, probably held deep religious significance [Parlett, 1999]. Its reliance on luck, according to some, would indicate that the winner was believed to be favored by the gods. Pre-Columbian civilizations probably came to a similar conclusion, if it is indeed true that they played ball games to determine who would be sacrificed atop a pyramid. The game of Go may find its roots in divination practices related to flood prediction and control.

Wargames – games which attempt to simulate or capture the essence of war under a strict ruleset – make a very convenient replacement for actual war. People are by their very nature drawn to compete and measure themselves against their peers; it is what [Callois, 1961] would call "agon" or playing out of desire to prevail. Moreover, these games can be used as a training tool for war. Ancient games most likely did not have the presumption to teach much in the way of practical military tactics, though they could certainly train the general's mental acuity and discipline.

The first board game to sport a consistent military background is arguably the Indian game of Chaturanga, even though there is no physical historical evidence about it. Considered to be the ancestor of chess and other chess-like games, including Jangki (Korean chess), Makruk (Thai chess), Shogi (Japanese chess), and Xiangqi (Chinese chess) it was allegedly played in the seventh century AD and its pieces were modeled after the actual Indian military, with the general and his advisor, slowly-advancing infantry, knights for flanking enemy lines, fast but difficult to maneuver chariots (rooks), and devastating war elephants (bishops). Its rules were very similar to those of chess, except that, instead of checkmating the enemy king, one simply had to capture it. The Romans had their own chess-like game, called Ludus Latrunculorum, or simply Latrunculi.

However, it was not until much later that games went full circle, coming back to a functional simulation of what they had come to symbolize. With the invention of Kriegspiel, men re-discovered the hunting games of tiger pups on a much grander scale [Perla, 1990]. It was the highly advanced Prussian military that first understood the potential of a realistic war simulation in the training of their officers and tacticians, but in order to provide such benefits, the game would have to evolve beyond the simplicity of a chess-like game, most importantly abandoning the realm of perfect information.

Kriegspiel was a serious game played on three identical boards representing actual territory. Two generals faced off with an umpire in the middle, the only one knowing the full state of the game. The players would issue orders to their units, and the umpire would carry them out, revealing to each player what their units could see, and no more. He would also resolve combat based on tables, rules and personal experience. Kriegspiel is thought to have been an important instrument for the armies that used it until the XX century. The Japanese navy used Kriegspiel in the Russo-Japanese war (1905), which resulted in the Rising Sun's unexpected major victory.

The modern descendants of Kriegspiel are computer games, especially the real-time and turn-based strategy genres, which owe everything to this original idea. So-called "tabletop wargames" are still widespread, mostly fought with toy soldiers and miniatures, though they eschew imperfect information due to practical difficulties in maintaining three boards. Instead, uncertainty derives from a random factor (dice) and estimating distances between units without using tools.

This thesis is about Kriegspiel, though not the Kriegspiel that the Prussians made. It is about a chess variant of the same name, designed around the same spirit, in hopes of making chess closer to a modern wargame. It is *blind* or *invisible* chess, with players only seeing their own pieces and submitting move attempts to a neutral umpire who can accept or reject them. Kriegspiel is like chess in that it follows the same rules, yet it is very different. For one, computers have a lot of trouble "getting" Kriegspiel compared to regular chess, whereas human players can adapt fairly quickly. Information is scarce, changes all the time and can be misleading, but every little bit of it can decide the outcome of the game. In a way, many Kriegspiel tactics could be likened to the ever elusive common sense that remains one of the most difficult things for computers to grasp.

We study Kriegspiel because it is a complex game that does not seem to fall completely into any one category, which makes it very much like a real-world conflict simulation. Playing a game of Kriegspiel forces you to reason about the past, present and future, to reason about yourself and your opponent, to decide what you know and what you choose to believe. Except in limited endgame scenarios, there is no ultimate perfection that a computer can discover by trying a number of combinations. Poker is a complex game that fits most of these criteria; even so, Texas Hold'em "merely" requires the player to select one of three strategies (check, fold or raise) through a handful of betting rounds. Imagine a game of poker with 40 options to choose from through 50 betting rounds in which your opponent may keep his strategy secret 75% of the time. Yet, maybe surprisingly, the best human players win consistently and computers are starting to make progress, as well. Within the context of this work, much of this progress will be discussed and analyzed.

This thesis is structured as follows. In chapter 2, we give a bird's eye view on the state of the art in game research. As the field is very vast, we will focus primarily on areas that are of particular interest to the present research, either because they introduce concepts and techniques that will be useful to our own Kriegspiel research, or because they offer interesting parallels and contrasts worth discussing.

In chapter 3, we introduce the chess variant of Kriegspiel. We provide the various rulesets adopted at one time or another throughout the 120 years of its history, then we focus on questions such as its complexity, the need for a special format to record Kriegspiel games without loss of information, and finally previous literature on the subject; this includes algorithms for the endgame, methods for solving Kriegspiel problems and game-playing agents. Historically, this is the order in which researchers have tackled the challenge of Kriegspiel.

Chapter 4 is about our first Kriegspiel engine, Darkboard 1.0, an artificial player based on the concept of metapositions. This chapter mostly refers to research contained in [Ciancarini and Favini, 2007a,b]. The main contribution consists of achieving a slightly above average level of play (by human standards) by using a minimax-like method that works despite the lack of perfect information. The method gives the game an illusion of complete knowledge by shifting focus from actual positions to metapositions containing a huge number of possible states, which are evaluated as a whole with a custom Kriegspiel function.

Chapter 5 is concerned with the same problem, but from a radically different viewpoint. Moving away from the limitations of the first approach – namely, the fact that the evaluation function is so inherently specific to Kriegspiel and requires much domain knowledge – we investigate the usage of Monte Carlo Tree Search to create a new Kriegspiel player, Darkboard 2.0. We compare Kriegspiel with other games in which this Monte Carlo method has been used successfully, especially Phantom Go, and we highlight how our algorithms differ from previous Monte Carlo Kriegspiel research. We modify the simulation step of traditional MCTS in order to improve its performance above the level of Darkboard 1.0. This new program works with little domain knowledge, attempts some measure of opponent modeling and could be adapted to any scenario in which one can model future sensory input (the referee, in this case). The chapter is based on research in [Ciancarini and Favini, 2009a,d].

Starting with chapter 6, we specifically deal with the problem of Kriegspiel endgames. These scenarios offer a considerably different challenge, since the amount of possible game states at any given time is small enough for all of them to be considered. As such, we have higher expectations for a computer player to be able to perform well in the endgame, though the task is far from simple. This chapter shows how a specific metaposition-based player can be built to play some Kriegspiel ending effectively. Initial research in this area is due to [Bolognesi and Ciancarini, 2004], and later inspired the minimax-like player described in chapter 4. The chapter is especially interesting as it provides an introduction and a paragon to the next two.

In chapter 7, based on [Ciancarini and Favini, 2009b,c] we describe a new algorithm for playing some Kriegspiel endgames perfectly. Perfection here means that if the starting position and belief set are such that we can win with probability 1, then we will do so in the shortest number of moves in the worst case, and without making any assumptions on the nature of the opponent. He may very well be omniscient, predict our own future moves or even have bribed the referee to let him move his pieces on the fly to other legal states in our belief set; he will still lose. We accomplish this result with a well-known tool in chess literature: retrograde analysis. We build a brute-force algorithm that analyzes Kriegspiel metapositions starting from checkmates and moving back in time, building a tablebase of won metapositions. We show that the tablebases need only be much smaller than the exponential number of theoretical belief sets.

Chapter 8 is the natural follow-up to the previous chapter. We discuss practical findings from the tablebases we have built for some Kriegspiel endgames, namely KRK, KQK, KBBK and KBNK, giving statistics, showing sample positions and finding answers to long-standing questions. Some of these problems, such as whether it is always possible to win the bishop and knight endgame even against the best defense, had been open for almost a century.

Finally, chapter 9 contains our conclusions and future developments in Kriegspiel research as part of the broader field of imperfect information games. Appendix A lists the full Kriegspiel ruleset as enforced on the Internet Chess Club, as this ruleset has been imposing itself as the official one in international competitions. Appendix B contains a Kriegspiel extension of the popular PGN file format for representing chess games.

1.1 Overview of the results

What follows is a short overview of the original research results obtained and documented in this thesis, together with the relevant papers. Our results are given in loosely logical order, starting from more specific achievements and moving to encompass broader problems and in more general terms.

- **Writing search algorithms for the Kriegspiel ending.** We create and define a lightweight, high-performance search algorithm for playing several Kriegspiel endgames convincingly well in most situations. Based on the concept of metapositions, this algorithm is minimax-like, though it does not evaluate single game states but entire information sets. [Bolognesi, Ciancarini, and Favini, 2009]

- **Extending the search algorithm to the entire Kriegspiel game.** We refine and generalize the endgame search algorithm so that a single evaluation function can play a full game of Kriegspiel. This requires a series of approximations to accommodate the much greater uncertainty, but leads to a good level of play. The resulting program will be referred to as Darkboard 1.0. [Ciancarini and Favini, 2007a,b]

- **Adapting Monte Carlo Tree Search to Kriegspiel.** We approach the same problem from a completely different angle, writing a Monte Carlo Tree Search (MCTS) algorithm for Kriegspiel. Unlike the metaposition-based method, this algorithm (called Darkboard 2.0) only requires minimal domain knowledge and it is consistently stronger than Darkboard 1.0; it is also naturally built for opponent modeling. While MCTS has been used in imperfect information games before, this is the first time it proves so successful in games with such highly dynamic and non-monotonic uncertainty. [Ciancarini and Favini, 2009a,d]

- **Creating endgame tablebases for perfect Kriegspiel play.** We apply retrograde analysis to metapositions in order to compute endgame tablebases for several frequent Kriegspiel endgames. The resulting strategies are optimal in the worst case, minimizing the maximum amount of moves it takes to achieve mate against an omniscient opponent. The algorithm can be applied to any game or subgame of imperfect information in which one side can push victory with probability 1; it is only limited by time and resource constraints. [Ciancarini and Favini, 2009b,c]

Chapter 2

State of the art in game research

In this chapter, we discuss the state of the art in game research. As such an analysis would necessarily deserve an entire book of its own, we limit our effort to select topics that will be particularly useful in the context of the next chapters. After an introduction explaining why games are important in computer science, we devote the rest of the chapter to advances in perfect and imperfect information games, respectively.

2.1 The importance of games

Artificial Intelligence and games have always mixed well, even before the birth of modern computer science, and even when it was a scam, such as “the Turk”, a chess-playing automaton (conveniently large enough to hold a person inside) built in Napoleonic times. Games make an excellent simplification of reality, a sandbox in which rules are easily enforced, moves have easily computable consequences, and success or failure are generally unquestionable. Researchers have studied games either for their own sake or in hopes of finding new results to be applied to real-world problems. On this note, articles such as [Bowling et al., 2006], appeared as the introduction to an issue of *Machine Learning*, show that there is an acute interest in game research on the part of the general Artificial Intelligence community. Machine learning is the branch of Artificial Intelligence aimed at making sure that an agent involved in a task of any kind can improve the quality of its decisional outputs by using existing data or previous experience, making them able to adapt to their environment and its dynamic nature. With respect to machine learning alone, the cited article defines a number of areas of interest in view of their applicability to different fields and problems.

- *Learning to play the game.* This is obviously the most explored area; games provide the perfect environment for testing learning procedures, methods and algorithms that will help them to learn more about the world and how to become more competitive players. The environment itself may vary greatly, ranging from classic board games to partial information games and even continuous, real time games.

- *Learning about players.* Opponent modeling (as well as the modeling of non-opponent agents, such as partner and team modeling) is a growing trend in game-related research. This topic is concerned with finding out the thought processes, plans, biases, strengths and weaknesses of other entities involved in the game.

- *Behavior capture of players.* Being able to reproduce the behavior of an existing players realistically and convincingly is becoming a new horizon in game research. The ability to simulate a player's actions is, surprisingly enough, being especially pioneered by commercial real time video games of various genres.

- *Model selection and stability.* This is the area of constructing and selecting learning specific models for a given game, adapting more general models to a particular environment in efficient ways to improve performance without sacrificing accuracy or predictive power.

- *Optimizing for adaptivity.* This task, again stemming from the entertainment needs of commercial video games, is interesting nevertheless since it revolves around the creation of opponents that are interesting to play: that is, able to adjust their level to the player's own and change their style to provide variety to human players.

- *Model interpretation.* An artificial playing agent should not only be able to provide its next move, but also to provide other answers reflecting a higher and more human-like awareness of the game model. In other words, the agent should appear to be using a reasoning process that can be followed and traced from the outside, beyond the simple production of raw numbers listing its reward expectations.

- *Performance.* As many machine learning tasks are extremely resource-intensive, performance is an area in constant need of attention, even in games where the artificial intelligence component does not have to compete with graphics and gameplay for resources.

It seems that every game has an interesting challenge to offer, be it a traditional board game or a modern, commercial console title. In the rest of this work we will mostly focus on the former category. Our main interest lies with well-known board and card games, and especially zero-sum games of imperfect information.

2.2 Perfect information games

In perfect information games, all players have full access to the current state of the game. This definition can accommodate games with a random component, such as Backgammon, as the environment can be thought of as an additional player making independent moves. These games have received the largest amount of attention in research, and in some cases have been solved or can be played by computers at levels that no human can approach. However, the actual level reached by computers and the difficulty in developing better engines depend on many variables, including branching factor, game duration, regularity properties in game states, existence of easily categorizable patterns, convergence to a small number of final states (or, conversely, divergence to a huge number of final states), and more. For all games, the dichotomy is between search-based methods and knowledge-based methods; the former aim at exploring many states, whereas the latter try to find an accurate evaluation of a small number of states. Many programs use a mix of both.

2.2.1 Solving the game

Zermelo's theorem [Zermelo, 1913] proves that any zero-sum game of perfect information can be solved, that is, there exists a perfect strategy for both players yielding a guaranteed minimum result. In the absence of mistakes by either player, the starting position in chess is a win, draw or loss. Simple games such as tic-tac-toe can be strongly solved by brute force methods, which means the perfect strategy is available for each position. [van den Herik et al., 2002] is a general survey on the state of the art in solved games, both at the time of writing and in the near future. Examples of strongly solved games include Awari, a popular African game [Romein and Bal, 2003], Connect Four [Allis, 1988], and Nine Men's Morris [Gasser, 1996].

Solving a game does not necessarily entail finding an optimal strategy for every position – it is possible to discard (sometimes major) portions of the game tree if it can be proved that they are never traversed in an optimal game. For example, if the first player has 1000 moves to chose from but can be shown to force a win with one of them, it is not necessary to explore the remaining 999 when considering his optimal strategies.

An interesting consequence of this fact is that while the size of the game tree is the major factor behind a game's complexity, a game can be smaller than another and still turn out to be more complex to solve. A weak solution to a game leverages this principle, only finding a perfect solution to certain positions that can force a win against any defense. A weak solution may not be able to suggest a strategy for a position outside of such a set, even though forcing a win may still be possible. A solved game may still be interesting to humans, such as in the case of checkers, weakly solved thanks to eighteen years of parallel computation [Schaeffer et al., 2007]. Other weakly solved games include 6x6 Othello [Feinstein, 1993] (a second-player win, although 8x8 Othello is still unsolved and generally believed to be a draw) and Fanorona [Schadd et al., 2008].

An even weaker form of solution is the so-called ultra-weak solution, in which only the game-theoretical value of the starting position is determined – in other words, who would win the game if both players used their best strategies – but no strategy is provided. The value can be derived through general reasoning, most notably the strategy-stealing argument first formulated by Nash [Berlekamp et al., 2003]: if the game rules admit the possibility of skipping a move, or a player can always make a harmless move, the second player does not have a winning strategy. If he had one, the first player could simply skip his first move and effectively become the second player himself. This, for example, allows one to deduce that Hex is a first-player win (the game can be strongly solved, but only on very small boards) and so would Go, if not for *komi*, the bonus points awarded to the second player for fairness. The argument does not apply to chess as player cannot skip a move and a move can be detrimental to the player making it. In fact, there are situations, called zugzwang, in which the player wishes he could just skip the move since every option damages him.

There are other ways to provide this kind of ultra-weak solution, and they generally involve heuristic search through a set of significant states. One such method is proof-number search [Allis et al., 1994]; given a predicate (such as a position being a victory or a draw), this algorithm will try to prove its truth by exploring the game tree based on the number of nodes required to prove the predicate, and choosing the direction that seems to yield the most convenient proof.

It should be noted that, while giving a strong solution is usually not feasible for most games, certain interesting subsets of a game can be strongly solved. The typical example is endgame tablebases in chess; this topic is covered in much greater detail in chapter 7, but here we will just recall that [Bellman, 1965] first interpreted the KPK endgame in chess as a dynamic programming problem, thus laying the foundation for retrograde analysis methods such as [Thompson, 1986]. Endgame tablebases are usually only possible for games that converge to a small number of states near the end: games in which the amount of pieces on the board decreases over time are ideal. Midgame tablebases are also possible in games such as checkers.

2.2.2 Minimax search

The idea of search in perfect information games is a direct consequence of the minimax theorem [von Neumann, 1928], which is in turn a consequence of Zermelo's theorem. Programs that focus on search over a large number of states were called in "type A" in the seminal paper [Shannon, 1950], and they have become the norm in fields such as competitive chess, relegating "type B" knowledge-based programs to academic research, at least as far as chess is concerned.

A full analysis of all minimax-based search techniques and heuristics would be beyond the scope of this work. It suffices to remember that all serious chess programs have sophisticated pruning algorithms, quiescence detection, and a robust move ordering policy, as well as an evaluation function that is as smooth as possible. It can be said this field has seen constant evolution, but not so much revolution. The original alpha-beta pruning [Edwards and Hart, 1963] has since been outperformed and, to a large extent, replaced by newer methods such as principal variation search (or negascout) [Reinefeld, 1989]. See, for example, [Plaat et al., 1996] for a review of several minimax search techniques and [Plaat et al., 1994] for a description of another advanced minimax algorithm, MTD(f).

In actual play, minimax is often applied in tandem with iterative deepening [Korf, 1987], which gradually increases search depth in order to obtain better and better approximations of the best strategy (whereas standard minimax is by its own nature depth-first and might not yield a decent result if the search was aborted before its natural end). Considering the best moves first is of crucial importance in minimax methods; this has led to such improvements as the killer heuristic [Akl and Newborn, 1977], then generalized to the history heuristic [Schaeffer, 1989]. Quiescent moves [Kaindl, 1983] extend search depth when large variations in the evaluation function are likely, for example after a capture. This optimization can be seen as a form of domain knowledge hard-coded into the search algorithm.

2.2.3 Monte Carlo search

The aforementioned methods make the assumption that either games can be explored to the end during the search, so as to discover the game-theoretical value of a given branch, or (much more likely) an evaluation function is available for the given domain. This is a reasonable assumption in chess and other games, but might not be as immediate in other fields. The evaluation may either not exist under realistic constraints or the domain may be obscure enough that humans have not mastered its traits. Therefore, there are search-focused methods that approximate a position's value in different ways from minimax. These methods are typically younger than traditional minimax as they usually require greater computational resources.

Monte Carlo searches, first introduced in [Metropolis and Ulam, 1949], are essentially random walks in the problem space; see, for example, [Kalos and Whitlock, 2008], for a more recent introduction. A very practical method, it was born from the intuition that the probability of winning a round of card solitaire Canfield could be approximated by playing it one hundred times and counting the number of victories. By playing enough rounds, one could reach any level of accuracy while avoiding difficult combinatorial reasoning.

Each sample obtained in this way provides statistical information on the various possible moves and their expected rewards. [Tesauro and Galperin, 1996] shows an early example of Monte Carlo optimization by searching through Backgammon positions. The main problem with the Monte Carlo method lies in the speed (or lack thereof) with which it converges to a reliable solution. Monte Carlo Tree Search (MCTS), an adaptive method that seeks to improve convergence speed of the Monte Carlo method, will be the focus of chapter 5. So far, it has seen the most success in the game of Go, whose strategic nature makes evaluation functions difficult to write, and whose size makes brute force approaches pointless.

A comprehensive definition of MCTS is found in [Chaslot et al., 2008]. Its main peculiarity lies in the fact that, while sampling is indeed random from a certain point in the simulated game, the initial moves are tested according to a deterministic selection algorithm that resembles quite closely reinforcement learning (see the next section); typically, this is the UCT algorithm [Kocsis and Szepesvari, 2006] or some ad hoc version of it tailored to the specific game. This ensures that more promising moves are allotted more simulation time, resulting in faster convergence to a reliable value for the best move.

2.2.4 High-level knowledge and planning

A knowledge-based method can be defined as any approach that attempts to estimate the value of a position without devoting most of its time to visiting other positions. Shannon's original chess player was a simple example of case-based reasoning that could be run manually, even without a computer. These "type B" programs are therefore much closer to the way humans play games. They attempt to understand the game before they act; they try to find patterns in the game structure, using correlations between the structure of a game position and expectation of its game-theoretical value. While a search-based method will almost always be an *online* one (the program determines the best move on the fly), knowledge-based methods often improve their quality *offline*, training themselves and their knowledge base before the game starts.

Planning is the problem of searching through a (smaller) set of states in order to find some state satisfying a given goal. Since planning in general is NP-complete or worse [Pollack, 1992], it has long been known that efficient planning involves constraining one's search to subspaces of the problem that are known, from reasoning or experience, to achieve a given goal. Compared to minimax methods, planning is obviously more dangerous, as there is no such thing as a depth cutoff - if the algorithm stops before a plan is found, nothing is returned. Moreover, planning requires a higher level of expressiveness and strategic awareness.

Planning methods never led to particularly strong chess players, though there are instances of attempts at doing so. We recall, for example, Wilkins [Wilkins, 1980] and his Paradise position solver. This kind of method is remarkable because the agent actually understands the position it is examining, finding high-level patterns influencing the decision-making process, as opposed to simply outputting a number, and provides reasoning and rationale behind its decisions. The weak points are that the knowledge base needs to be inserted manually and there are positions that the player simply cannot play because it lacks deep enough knowledge to do so. Moreover, Paradise was firmly centered on chess and had hard-coded primitives that could not be applied to any other game. Still, it has been shown that chess plans can be learned from databases and even humans can improve their play with these plans [Sadikov and Bratko, 2006].

There are other methods that attempt to provide a computer with intelligence by mimicking the way humans approach a game. Perhaps the most popular among these is chunking, which is based on a known psychological mechanism [Gobet et al., 2001]. Much research has been devoted to the way expert chess players see and reason about the board – see, for example, [Chase and Simon, 1973]. Human players form a repertoire of chunks, or patterns, whose properties they can quick recall and apply. In chess, for example, chunks are specific piece arrangements, whereas in Go they can be certain stone patterns. Programs have been written that could acquire predictive information and usage information from chess chunks [Walczak and II, 1993, Walczak, 2003]. Research in Go patterns has been even more widespread (probably because Go was not killed by minimax), and these patterns are often included in search-based programs. See [Stern et al., 2006] for a recent algorithm, and [Bouzy and Cazenave, 2001] for a more general survey of computer Go techniques.

Finally, we note that research exists in the field of transfer learning, or the ability for an agent to learn across different games, taking features from a better-understood domain and exploiting them in novel games. See, for example, [Banerjee and Stone, 2007] for an example using tic-tac-toe knowledge to improve its play quality in simpler variants of Connect Four and Othello.

2.2.5 Neural networks

No review of computational intelligence methods could do without a mention of TD-Gammon [Tesauro, 1992, 1994], a major success story in game research. This program reached a world-class level of play in Backgammon by using a neural network trained with the method of temporal differences [Sutton, 1988]. This method provides the so-called reinforcement learning, that is, the agent has an expectation concerning the value of an action, tests the action, observes the result, and adjusts the expectation accordingly. Most of the time, the agent will play the move with the highest expected return (exploitation), but occasionally it will chance another move to check for even more profitable options (exploration). In more ways than one, this resembles the Monte Carlo method described above, however this form of reinforcement learning does not play random games; instead, it updates its beliefs on the best policy at the end of each game.

As mentioned, this method proved wildly successful in Backgammon; TD-Gammon acquired grandmaster-class play through self-play alone, and later world champion level with the addition of hard-wired domain knowledge. This is a rarity, as programs that develop their ability by only playing against themselves usually tend to learn quirky tactics that only work against specific players. In fact, even though it was used elsewhere such as in Go [Schraudolf et al., 1994] and chess [Baxter et al., 2000], the success of temporal difference learning was not replicated in most other games; Tesauro attributed it, among other things, to the random factor in Backgammon and certain games being non-Markovian (i.e. the best strategy depending on previous states other than the current one).

Trained neural networks have been a popular alternative to minimax. In addition to Backgammon, they have been applied to chess [Fogel et al., 2004], checkers [Chellapilla and Fogel, 1999] and Othello [Moriarty and Miikkulainen, 1995]. The last contribution was especially successful as the agent learned unexpectedly complex strategies.

2.2.6 Genetic programming

Yet another approach to function optimization – which is what game playing boils down to – is through evolutionary genetic methods. This branch of Artificial Intelligence could be considered an extension of an iterative optimization algorithm called simulated annealing [Kirkpatrick et al., 1983]. In simulated annealing, a function is minimized through iterative adjustment of its parameters. A probabilistic gradient search, simulated annealing would move from a point to one of its better neighbors according to a probability distribution. It was noted that losing track of previous best points could be detrimental to the overall quality of the result, as progress could easily be blocked by a local minimum in the function, and the algorithm would need to backtrack and retry. Genetic programming [Koza, 1992] is an answer to this problem that replaces the single point with a population of candidate best points and applies natural selection dynamics to this population.

Application of genetic methods to games is almost as old as genetic programming itself; [Ferrer and Martin, 1995] is an early example for chess. This is a very powerful method, but one that is generally regarded as rather empirical in many ways, as many results that are observed in practice are extremely difficult to prove in theory. Genetic algorithms typically require fine-tuning and a very careful choice of the selection model to be used, or they can easily yield no result at all.

Genetic algorithms are very general and can be applied to a variety of artificial players; basically, any function that accepts parameters or weights can be evolved with this method. In [Fogel et al., 2004], the authors develop an evaluation function based on classical features as well as neural networks whose inputs are opportunely chosen configurations of chessboard squares. Individual agents differ in the weights assigned to the function as well as those associated with the nodes of the neural network.

The work in [Lassabe et al., 2006] deserves a mention as a genetic method for teaching a computer player how to play, among others, the famous KRK endgame. While the building blocks of the algorithm are elementary patterns and strategies defined by a chess expert, how these blocks are combined and used in response to the situation on the chessboard is decided by a genetically programmed algorithm. This is only one of the most recent results in a series of papers dedicated to the relationship between chess and evolutionary learning.

2.3 Imperfect information games

Imperfect information games are those games in which players are not fully aware of the current state of the game. The term covers a wide range of games that are vastly different from one another: examples include Battleship, Bridge, Kriegspiel, Poker, Risk, Scrabble. Much like the case of perfect information, these games differ in the size of the problem space and the type and number of actions that players can perform. Likewise, we do not expect a single algorithm or method to be effective in solving every imperfect information game.

From a game-theoretical standpoint, imperfect information games are characterized by the fact that players do not know, in general, which state they are in. At any time, there is an *information set* containing all plausible states, but a player cannot distinguish among these individual states. The cardinality of information sets can range from one to infinity. Kuhn trees [Kuhn, 1953] are the imperfect information equivalent of extensive form in perfect information games. Figure 2.1 shows the difference between perfect and imperfect information, with a 1 marking a choice node for the first player, a 2 for the second player, and leaves containing payoffs for both players (the

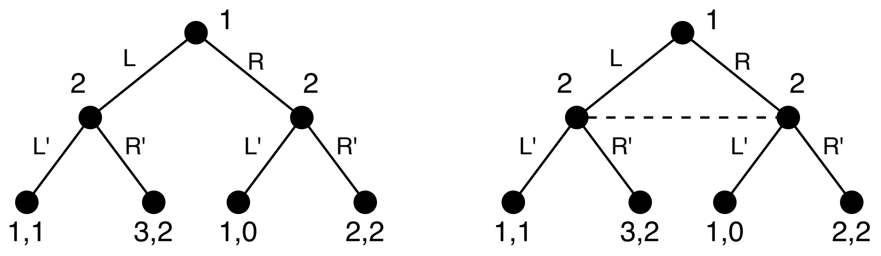


Figure 2.1: Extensive form with perfect or imperfect information.

game in the example is not zero-sum). In the imperfect information case on the right, the second player does not know the first player's move; in other words, his information set is comprised of the two states connected with a dashed line. If he plays L', payoff will be (1,1) or (1,0) depending on whether the first player chose L or R.

Zermelo's theorem does not hold in general for these games. In fact, the optimal strategy in a generic imperfect information game is a mixed strategy that plays different actions according to a probability distribution. In spite of this, the methods used to play such games, while custom, are often variations of perfect information approaches.

2.3.1 Minimax search

Minimax cannot usually be applied without modifications, as there is no single entity to maximize and minimize. Moreover, [Blair et al., 1993] notes that exploring these trees is NP-hard and requires custom algorithms depending on the domain. This is not to say it has not been done; depending on the domain, minimax-like methods can be the most suitable for solving a certain game. If a two-player game of imperfect information can be suitably represented, for example as a one-player game against nature, in which nature plays moves according to some distribution, then there are feasible minimax approaches.

Expectiminimax [Michie, 1966] (often referred to as Expectimax) is minimax with chance nodes. Instead of returning a minimum or maximum value, chance nodes return a weighed average of their children. For example, if a move had three possible outcomes, it could be represented as a chance node; each child would be explored separately and then averaged. While the approach is sound, it is quite inefficient as it implies all children need to be explored: minimax should follow the opposite strategy of pruning as much as possible.

An algorithm known as *-minimax [Ballard, 1983] was invented to serve as the imperfect information equivalent of alpha-beta pruning with chance nodes. While it was not immediately recognized, it was later rediscovered [Hauk et al., 2006] and updated to be the equivalent of newer search algorithms such as negascout. Research on pruning in trees with chance nodes is still ongoing; see, for example, [Schadd et al., 2009] for a recently developed forward pruning algorithm.

2.3.2 Monte Carlo search

Monte Carlo methods are often one of the least expensive choices for games of imperfect information, as they do not necessarily need any domain knowledge. There are, however, some special considerations to be made when using such methods in imperfect information domains, not least of which the choice of which opponent model to use. It is noted in [Frank and Basin, 1998, 2001] that if a Monte Carlo sampling method works by picking possible states at random and reasoning as if each was a game of perfect information (e.g. with a minimax-like algorithm), then there are theoretical limits on the accuracy of the result due to strategy fusion and non-locality. Evidently, a best defense model – assuming the opponent will always act in the best way as if he had perfect information – may not be a realistic assumption or even a useful one [Jamroga, 2001].

This has not prevented some Monte Carlo methods from being popular in Scrabble [Sheppard, 2002], Bridge [Ginsberg, 1999], Poker [Billings et al., 2002], Kriegspiel [Parker et al., 2005, 2006] and more recently Phantom Go [Cazenave, 2005, Borsboom et al., 2007]. It should be noted that while all these programs can be gathered under the generic Monte Carlo umbrella term, they are vastly different from one another and only share the common trait of running many simulations to converge to the result. For example, in Scrabble simulations are concerned with letter assignments, and they are skewed to represent the opponent's tendency to keep good letter sets for later turns, whereas the quoted Phantom Go papers deal with more-or-less textbook Monte Carlo Tree Search, but starting from random plausible layouts for the opponent's stones.

2.3.3 Planning

There is no doubt that planning methods are common in videogame AI. Capturing complex behavior such as troops management in a real-time strategy game or a bot's chasing patterns in a first-person shooter would be difficult without an abstraction layer. Still, in most cases these are hard-wired plans scripted by the developers and usually unchanging [Hoekstra, 2006]. What we are interested in is the dynamic ability to plan in the absence of perfect information.

The first planners were entirely deterministic and assumed perfect knowledge of the domain. This was one of the basic premises of the famous STRIPS language and later methods using it as their foundation [Fikes and Nilsson, 1971]. This is not to say that they could not work in a environment with imperfect information at all. They simply could not understand the changing world; if the agent left a block in a given position and found it in a different one, it would need to recalculate its plan to adapt it to the changing circumstances. It would view the moving block as no more than the effect of some 'poltergeist' that invalidated its reasoning; it would never account for it, or try to predict it.

It was not until much later that languages and agents were expanded to include the handling of imperfect information, or reasoning under uncertainty. Seeing as STRIPS were introduced in the early '70s and these developments did not become popular until the '90s, it took the scientific community two decades to start addressing the problem of an agent that was not omniscient. The theory behind some of this research is that of *partially observable* Markov decision processes, an extension of the classical theory in which not every transition may be known. This purely stochastic approach was followed, for example, in [Kaebling et al., 1998]. Other systems were developed that inherited more from classical planning algorithms, such as the early C-Buridan, as described in [Draper et al., 1994]. Later, Mahinur

[Onder and Pollack, 1999], introduced the first optimization techniques by using heuristics to drive its search. These were both regressive planners, starting from a desired end goal and moving back to the starting state.

An interesting new take on conditional planners is PTLplan [Karlsson, 2001]. PTLplan is a progressive planner, meaning that it starts reasoning from an initial state (which can be probabilistic) and applies rules until it finds a plan satisfying a goal, or realizes that no such plan exists. Aside from the usual fluents and constructs from temporal logic, it uses a series of control formulas modeling strategic knowledge of the domain; these formulas are used as invariants, and dramatically prune the search tree, greatly improving performance. This approach definitely shows potential in a game-based scenario.

Other, newer approaches to planning involve studying the problem as a particular application of a different theory. We recall, for example, the research done in the last decade on planning as model checking (see, for example, [Giunchiglia and Traverso, 1999] for an exhaustive introduction to the subject). Model checking deals with the satisfiability of a set of formulas expressed in a given language; therefore, planning problems can be viewed as a model whose satisfiability equates to the plan being applicable.

As far as games go, planning methods have been successful in bridge, though the general framework is not game-dependent [Smith and Nau, 1993]. The authors build minimax-like trees, but instead of containing states, each node contains a plan: basic formalized strategies that are common in expert human play and bridge manuals. [Smith et al., 1998] is based on a methodology called Hierarchical Task Network planning (see [Sacerdoti, 1977]). Within this paradigm, plans are subdivided into a hierarchy of tasks to be completed, with constraints and conditions depending on the actual situation that the agent encounters.

We also cite [Chung et al., 2005], a cross-over of planning and Monte Carlo in a real-time strategy game. Just like the bridge program replaces states with strategies in a minimax context, it is possible to do the same in a Monte Carlo environment, using a simulator to approximate the outcomes of a given plan. For games set in a continuous domain in both space and time this seems like one of the most promising avenues.

2.3.4 Opponent modeling

The importance of opponent (or adversarial) modeling was understated for a long time, mostly because it is not generally considered a foremost concern in practical chess. While there is research on opponent modeling in chess, it is often the case with chess program that any time spent doing opponent modeling would yield higher returns if spent examining more nodes.

Still, [Jansen et al., 2000] offers a statistical approach to the problem of predicting the opponent's move in chess. The authors make use of probability distributions with data from a database of grandmaster games in order to assign appropriate weights for select 'features' in a given evaluation function, so that the function will try to conform to the playing style of the human being studied. On the other hand, [Willmott et al., 1998] is about opponent modeling in the context of the aforementioned Hierarchical Task Network planning, with applications to chess and Go. The main goal of opponent modeling in [Willmott et al., 2001] is to reduce Go's huge branching factor.

In quite a few imperfect information games, opponent modeling is the only way to achieve expert play. Poker is entirely dependent on the player's ability to classify the opponent as playing tight or loose, and gauge how likely he is to respond to bets by folding, calling or raising. In Poker alone, which has been a true trend-setter in this field, many opponent modeling techniques have been implemented, including neural networks [Billings et al., 2002], Bayesian methods [Southey et al., 2005, Ponsen et al., 2008] and game tree search [Billings et al., 2006].

Bayesian probabilities were also used to model the opponent in a simplified variant of Kriegspiel played on a 4x4 board [Del Giudice et al., 2009], as well as in Stratego [Stankiewicz, 2009]. In other cases, opponent modeling does not target a specific opponent, but captures the features of a generic, average opponent. This is used, for example, to skew the otherwise uniform probabilities in a Monte Carlo approach (selective sampling), as has been done in Scrabble [Sheppard, 2002] and Bridge [Ginsberg, 1999].

Chapter 3

Kriegspiel

In this chapter, we introduce the game of Kriegspiel, which is the focus of this thesis. We first discuss the history, spirit and rules of the game (a slightly less obvious task than one would expect, given that there are several rulesets), then we list the available literature dealing with the game. The topics concerning computer agents for playing Kriegspiel will be touched in much greater depth in the next chapters.

3.1 Overview

Kriegspiel is a chess variant in which the players cannot see their opponent's pieces and moves. The game is played on three chessboards, one for each player and one for the referee (**umpire**), the only one possessing complete information on the state of the game. The players are given a full set of pieces of their opponent's color, and are free to place them anywhere on their chessboards to aid their memory or visualize their guesses on the opponent's deployment, but this has no effect on the game itself.

When a player is requested to move, he or she will announce the move to the umpire (and only the umpire; there should be no direct interaction between the players in Kriegspiel). The umpire will then check on his chessboard whether the attempt is legal.

- If the move is illegal, he will say "illegal" and ask the player to choose another move instead.

- The referee should say “nonsense” if the move was trivially illegal even on the player’s board, for instance if he were to try to move a knight like a rook; this to prevent one player to trick the other with a large number of illegal moves in order to mislead the opponent about his actual resources.
- If the move is legal, the umpire will be silent, or say something along the lines of “Black moved” or “White to move”.

In addition, the umpire will notify both players in the following cases.

- If a piece is captured (specifying where, and possibly some information on the captured piece depending on the rule variant, but never will he say anything on the nature of the offending piece).
- If a player’s King is in check, he will specify the direction (or directions, if it is a double check) from which the check is being given.
    - Rank check.
    - File check.
    - Long diagonal check (from the king’s point of view).
    - Short diagonal check (from the king’s point of view).
    - Knight check.

The umpire’s messages are therefore laconic, and as a rule, everything he says can be heard by both players, even though they will draw different information out of them. In Kriegspiel, you know what you know, but you do not know what your opponent knows.

Unfortunately, Kriegspiel is hardly a standardized game, which is both a cause and a consequence of its scarce popularity throughout the XX century, at least until more recent years. This variant has, itself, several variants that, while keeping the original spirit of the game intact, differ slightly in the way the umpire communicates his messages, and the amount of information contained therein.

3.2 Rule variants

Chess Kriegspiel was born in England, and the oldest ruleset is referred to as 'English rules'. The rules enforced at The Gambit (a famous chess club in London), an example of English rules, are listed in Appendix A. It can be said that the spirit of the English ruleset is the most akin to that of the old Kriegspiel used to simulate war. It makes for slower, but subtler gameplay in which every action is to be carefully considered, and information is expensive to acquire. In fact, the rules are designed to force the player to pay a price for each piece of information he gets.

The most notable rule here is called *'Are there any?'*, a sentence which has become quite famous (Kriegspiel is known as 'Any?' in the Netherlands). It is also the name of a collection of Kriegspiel problems by G.F. Anderson; a problem from that book will be examined in the next section. This rule allows the player to ask the umpire, before his move, whether he has any possible *pawn tries*, that is, legal capturing moves with his pawns. If there is none, the umpire will say 'No'; otherwise he will say 'Try'. In the latter case, the player must try at least one capture with his pawns. If the try is unsuccessful, he is not forced to try another pawn capture. In this way, the player *pays* for the information he has been given, possibly losing his freedom to choose. Also, the English rules do not specify whether a captured piece is a pawn or not.

The second important ruleset is due to J.K. Wilkins, an American mathematician (Kriegspiel has always been most popular in Anglo-Saxon countries). He directed the RAND Institute after the Second World War and introduced Kriegspiel into the Institute as a means of training in the analysis of war scenarios (RAND being a large think tank with the goal of providing advice to the government on many topics, including the new cold war). This ruleset is known as RAND rules and is listed as Appendix B. RAND games are usually faster than games played under the English rules. It was thanks to this connection with the RAND Institute that several world-famous game theorists such as John Nash and Lloyd Shapley became interested in Kriegspiel.

There is an additional American ruleset that lies halfway between the English and RAND rules, and it is called 'Cincinnati style', listed in Appendix

C. This ruleset forms the basis for variant wild 16 (Kriegspiel) on the Internet Chess Club, whose actual rules are described in Appendix D. In these rulesets, pawn tries are automatically announced before every move, with no try being forced upon the player. Since most Kriegspiel games are now played on the ICC, Cincinnati style rules are the obvious candidate for standardization in the event of official competitions. Indeed, the Computer Olympiads have already adopted the ICC rules as the only legal variant, and the programs described in this thesis all support this ruleset.

It should be noted that, in many situations and most scientific literature on Kriegspiel (such as optimal endgame strategies and algorithms), the ruleset of choice is irrelevant. Many Kriegspiel problems can also be solved under more than one ruleset.

There are also a few Kriegspiel-like chess variants, typically with more information disclosed to the players. For example, in *dark chess*, a player can see the squares threatened by his pieces, whereas in *invisible chess* only some pieces are hidden from view. *Stealth chess* (not to be confused with the fictional chess variant from the Discworld novels) is a cross-over of chess and Stratego, in which the nature of a piece is only revealed when attempting a capture.

3.3 Game complexity

The two most important measures of game complexity are state-space size and game-tree size. The former refers to the number of legal distinct states allowed by the game's rules; the latter to the number of distinct games that can be played. For chess, the first estimate was given in the seminal paper [Shannon, 1950], with a lower bound of 10^{43} for state-space size and 10^{120} for game-tree size. This was a conservative estimate and [Allis, 1994] provided a larger one: 10^{50} and 10^{123}, respectively.

If we consider Kriegspiel to be just a game of chess, then we need not go further than this: the estimates also apply to Kriegspiel. However, these numbers refer to the umpire's perspective of what is going on, but they make no sense to the players because they cannot perceive the state of the game to begin with. Rather, the players will define the "state" of the game as either the disposition of their own pieces, or the disposition *with* its associated belief state – that is, the set of all dispositions of enemy pieces compatible with the history of the game so far. The former definition makes the game look simpler, with just about 10^{25} states corresponding to the possible layouts of the player's own pieces, but this is merely an illusion as it completely ignores all the information the player could have collected so far. In other words, the smaller state space is just a reflection of a myopic player's inability to distinguish between states.

Belief states introduce a much higher level of complexity. If we imagine the 10^{50} positions of chess to be 10^{25} dispositions of white pieces, each of which has on average 10^{25} dispositions of black pieces, and each may or may not be included in the current belief state, the number of unique belief states explodes like a power set: $10^{25} \cdot 2^{10^{25}}$. Clearly, this astronomical number is nowhere near the actual complexity of the game, because the umpire is not informative enough to allow a player to distinguish among all possible belief states. Indeed, the number of umpire messages and combinations thereof is the real limit to the combinatorial explosion, and this number is just as important, if not more, than the actual branching factor. Such a consideration is especially interesting in the endgame, as we will see in chapter 8: it is the reason why we can build a database of Kriegspiel belief states (metapositions) with a $10^{10} : 1$ compression factor.

We gathered data from about 12,000 Kriegspiel games played on the Internet Chess Club. According to this collection, the average duration of a Kriegspiel game is 52 moves (104 plies), making it somewhat longer than a chess game, and the perceived branching factor is 40. Of these, about 10 would be illegal if tried. Let us give an estimate on the number of imperfect information states, then. We know that there are about 10^{123} chess games, and if we exclude illegal moves for a moment, we have that each node in each one of those games will correspond to a belief state determined by the previous moves. Make the extreme assumption that all belief states generated by distinct sequences of moves are distinct, and 50 moves on average yield $\sim 10^{125}$ distinct belief states.

Illegal moves complicate matters, but this fact is mitigated by their short horizon: usually, they become uninformative after the opponent's next move, that is, we cannot rule out any states based on the memory of the illegal move. For this reason, we may be able to represent illegal moves as a multiplicative constant for the number of game states instead of a contributor to its combinatorial explosion. If there were 10 illegal moves on average and one tried all their possible subsets, of which there are 2^{10} (the order is irrelevant as one illegal move does not affect the others), the multiplicative constant would not exceed 10^3. Then, we can give an upper bound to the number of belief states at 10^{130}. This is a loose upper bound, and the actual number is probably quite a bit less. It still shows that the number or perceived states is clearly much larger than the number of legal chessboards (though not as large as the number of Go states, which is around 10^{170} [Tromp and Farneback, 2007]). The number of Kriegspiel games is enormous if one takes illegal moves into account, though: between any two chess moves there can be any sequence of illegal moves. Even ignoring the order, that means on average 1000 combinations of illegal moves in between any moves of any chess games.

3.4 Literature

Although it is a fascinating game, played by hundreds of people every day on the Internet Chess Club, only a small number of papers have studied some aspects of Kriegspiel or Kriegspiel-like games. In this section we provide a summary of Kriegspiel literature.

Kriegspiel was often featured in specialized chess variation journals and magazines such as *The chess amateur* as early as the 1920's. For example, [Isham, 1926] contains the earliest claim of the existence of a forced mate for the bishop and knight endgame in Kriegspiel. The game was mentioned in the seminal book *Theory of Games and Economic Behavior* [von Neumann and Morgenstern, 1944] as "blind chess". Actual scientific research on Kriegspiel did not start until much later, though. The first research papers on Kriegspiel tackled the problem of building an automatic referee [Burger, 1967, Wetherell et al., 1972, 1975]. We believe that the best-known automatic referee for Kriegspiel is currently offered by the Internet Chess Club, although it has a few shortcomings due to its chess-like nature. For example, while it allows players to save the transcripts of finished games, it will not record illegal moves, which would be extremely insightful for users to know.

3.4.1 Kriegspiel endings

The next step was the construction of algorithms for playing certain Kriegspiel endings. Players and researchers quickly realized that Kriegspiel endgames were harder than their chess counterparts, but could, in some cases, be won with probability 1 or approaching 1. Thirty years ago, Donald Knuth gave the KRK endgame in Kriegspiel as an assignment to a Stanford class [Van Wyk and Knuth, 1979]. Boyce, a student in Knuth's class, later published a study on KRK, proposing a natural-language procedure to solve it in [Boyce, 1981]. Another algorithm for the same endgame was found independently by [Leoncini and Magari, 1980]. This endgame is well-known in orthodox chess, having been studied since the XIX century; Torres y Quevedo built the first mechanical player for KRK in the last decade of 1800.

Both Boyce and Magari's algorithms are based on a series of informal directives that allow White to achieve checkmate in a bounded number of moves regardless of Black's defense, but it is not proved that this is always the case, or that the strategy leads to mate in the shortest number of moves. In particular, Boyce's algorithm seeks to trap the black king in a single quadrant of the board, pushing it back towards the corner with the white king. Magari's algorithm sweeps the board rank by rank with the rook until a check message is announced, at which point it infers on which side of the board the opponent is and works towards limiting its space not unlike Boyce's algorithm.

Lloyd Shapley found a solution to the KRK endgame even in the case of an infinite chessboard quadrant (see Figure 3.1), showing how checkmate is inevitable in a bounded number of moves. He included this peculiar problem as number 12, "The infinite power of the rook", in his unpublished work *The Invisible Chessboard* [Shapley, 1987]. Later in this thesis we will show another problem from the same book regarding a mate with bishop and knight. The solution to the KRK puzzle has recently been documented in [Ferguson, 2009].

The KPK ending with king and pawn versus king was the first to actually be implemented on a computer system in the Prolog language [Ciancarini et al., 1997]. This paper also provides an example of a Kriegspiel scenario in which the stronger side cannot checkmate with probability 1 but can get

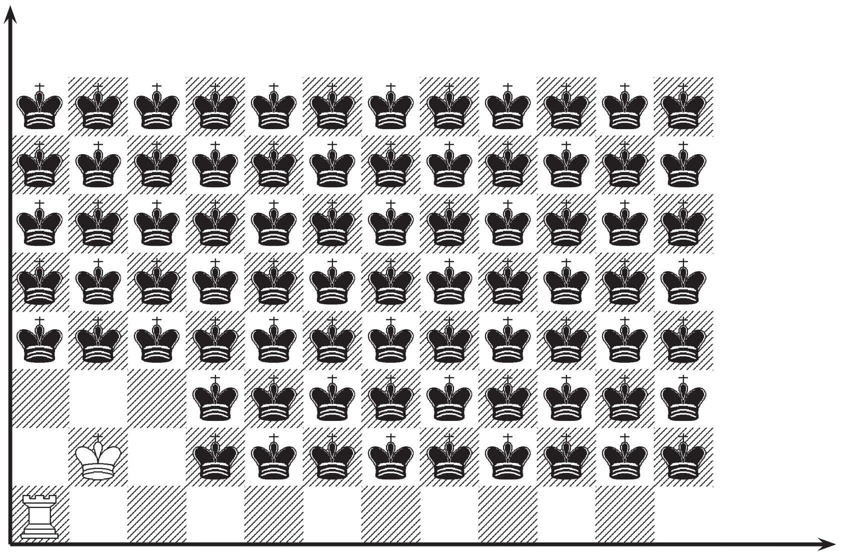


Figure 3.1: KRK on an infinite chessboard.

as close to 1 as desired. This is demonstrated by showing that a particular position is equivalent to a recursive game called Blotto's problem, in which the stronger player needs to take an arbitrarily small risk and is unable to take the full reward of 1. The Prolog player acts under the principle of bound rationality and makes reasonable choices based on the time and resources at its disposal.

The KBNK and KBBK endgames were investigated in [Ferguson, 1992, 1995], respectively. While these problems had been discussed by amateurs for decades, these papers actually show a complete strategy for winning these endgames from the most generic starting positions whenever possible. Again, sometimes the stronger side needs to take a small risk of drawing the game in order to achieve victory; in particular, in KBBK one does not seem to be able to win with certainty if both the king and the bishops start in the 16 central squares.

The first computer player for the KRK, KQK, KBBK and KBNK endgames was described in [Bolognesi and Ciancarini, 2003, 2004]. It is based on the concept of metapositions, a tool for merging game states into a single entity for the purpose of evaluation. See chapter 6 for an in-depth discussion of this method. The Kriegspiel player in [Ciancarini and Favini, 2007a,b], described in chapter 4 extends the method to all lone king endgames by basically extrapolating a generic evaluation function that is helpful in most cases.

3.4.2 Problem solving

Much like chess, Kriegspiel problems can be invented and solved. Usually, these problems require the reader to make good use of whatever information is provided to rule out impossible cases. In some cases, both players know the starting positions of all pieces; in others, only their type and amount are known, and sometimes not even those. Certain problems include a sequence of moves played before reaching the current state of the games. What all problems have in common is that one side needs to win or draw within a certain number of moves, much like any chess problem. [Anderson, 1959], [Li, 1995], as well as the unpublished [Shapley, 1987], [Ciancarini, 2004] contain collections of such problems.

From a scientific point of view, solving Kriegspiel problem is a state reconstruction task. One wants to reconstruct a state, either in the future (checkmate) or in the present and even in the past. [Russell and Wolfe, 2005] is an attempt at finding future checkmates through a search in large AND-OR trees, whereas [Nance et al., 2006] uses Kriegspiel as a test bed to infer data about the opponent's pieces as clauses within the context of a logic framework. There is not much more research specifically devoted to this aspect of Kriegspiel, mostly because practical game-playing programs have usually not focused on the task of reverse engineering the exact state of the game (which is a hopeless task except in the very beginning and the very end).

3.4.3 Player agents

We assume our player does not cheat by accessing the umpire's board. This may seem needless to say, but there used to be a program called Fark on the Internet Chess Club that included, among its modes of play, a perfect information one for unrated games. It is difficult to quantify the value of perfect information, but there exists a very obscure Kriegspiel variant in which one player gains perfect information but must forfeit his queen and rooks to compensate for it. There are also the aforementioned partial-information variants: dark chess, invisible chess and stealth chess. Information has a different value in each of those.

The simplest player agent is, obviously, a random player. The rationale behind the random player is that the opponent cannot see it is moving at random; given that checkmate is harder in Kriegspiel than it is in chess, a random player may have comparatively better luck than in chess. Taking a step forward, most early Kriegspiel bots were semi-random, possessing a set of case-based rules (when captured, capture back; exploit your pawn tries; always promote when possible; etc.) but reverting to random moves when no such condition matched.

For a long time, there were no agents capable of playing a full game of Kriegspiel better than a semi-random player. The first algorithm to do so, aside from the aforementioned [Ciancarini and Favini, 2007a], was [Parker et al., 2005]. This was a Monte Carlo method based on the generation of random boards compatible with the umpire messages so far, evaluated with a chess engine; it would play the move with the best average value. This approach is discussed at length, and compared with our own Monte Carlo algorithm, in chapter 5.

A different method, used in [Del Giudice et al., 2009], consists of representing the Kriegspiel game as a stochastic process, with possible positions being nodes on a random walk. Then, probability theory and past history can model the transitions between one position and the next. While this method was only experimented on a smaller board and with just a subset of the full arsenal of pieces, it was the first serious attempt at modelling the opponent in Kriegspiel.

Very recently, [Bryan et al., 2009] implemented a Kriegspiel player for the full game with a method reminiscent of [Parker et al., 2005], but with a more sophisticated board generation algorithm based on particle filtering techniques.

Chapter 4

Playing Kriegspiel with metapositions

In this chapter, we describe a Kriegspiel-playing program based on the concept of *metaposition*, that is, the merging of a very large set of possible game states into a single entity. This merging operation allows us to exploit traditional perfect information game theory tools such as the Minimax theorem. We provide a general representation of Kriegspiel states through metapositions and describe an algorithm for building and exploring a game tree of metapositions. Our method does not assume that the opponent will react with a best defense model. We evaluated our approach by competing against both human and computer players. We found that this method led to a good quality of play, which outperformed every other available computer agent until we developed the Monte Carlo player described in the next chapter.

The structure of the chapter is as follows: in Section 4.1 we model the notion of Metaposition for Kriegspiel, adapting a concept introduced by Sakuta and Iida for Shogi; in Section 4.2 we describe the basic design of Darkboard 1.0, our program able to play a whole game of Kriegspiel, with a special emphasis on the representation of metapositions, whereas in Section 4.3 we describe how a tree of metapositions is generated and updated; in Section 4.4 we show how a move is selected, exploiting the evaluation function described in Sect. 4.5. Finally, in Section 4.6 we present the results of a number of playing experiments, and draw our conclusions.

4.1 Metapositions

In the context of imperfect information games, if **S** is the set containing every possible game state (for example, every possible chessboard configuration in the game of chess, or all card distributions in a hand of poker), we can define the *information set* $\mathbf{I} \subseteq \mathbf{S}$ as the set of possible game states at any given point during a game, from a player's point of view. The player has no way to distinguish these states from one another, and in the tree representation of imperfect information games (Kuhn trees), these indistinguishable states may be linked with dashed lines meaning that the opponent does not know in what state they are after the move. For example, in Kriegspiel the black player's information set would contain twenty game states after White's opening move, corresponding to the twenty moves a chess player may choose from on their first ply.

The information set for a simple game can be computed and maintained explicitly throughout a game; this is, for example, the case in imperfect information tic-tac-toe (where the opponent's marks are invisible and a referee rejects attempts at placing a mark on an already marked square). However, for complex games like Kriegspiel the storage capacity and processing power required for building and using an information set far exceeds the capabilities of current and foreseeable technology, given that the typical information set for an average middle game position in Kriegspiel may contain about 10^{27} states, and it is certainly possible to envision games with even larger problem spaces.

Clearly, a program that aims at mastering an imperfect information game must capture the nature of the information set and work on it somehow, finding reasonable ways to drastically reduce the size of the problem. For example, the Monte Carlo approach focuses on a small subset of game states on which it then performs its analysis. The approach described in this paper provides a different approximation of the information set based on the concept of *metaposition* as a tool for merging an immense amount of game states into a single, small and manageable data structure.

The word *metaposition* was first introduced by Sakuta [Sakuta, 2001], where it was applied to endgame scenarios for the Shogi equivalent of Kriegspiel. The primary goal of representing an extensive form game through metapo-

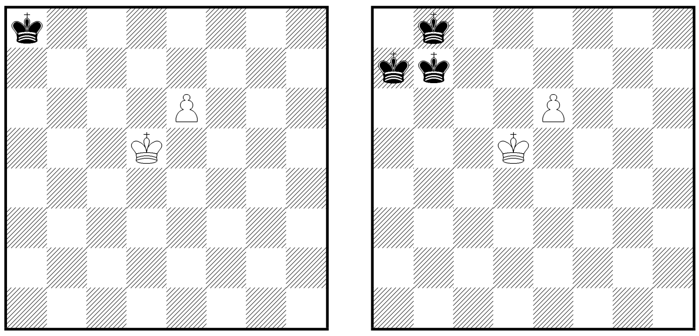


Figure 4.1: Metapositions and uncertainty in Kriegspiel: before and after Black's move.

sitions 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, as described in the quoted work, *merges different, but equally likely moves, into one state* (but it can be extended to treat moves with different priorities).

Let us introduce the concept through a Kriegspiel example based on Figure 4.1. Suppose that Black is now to move. His King has three possible choices: a7, b7 and b8. White's possible moves on the next ply depend on which one Black chooses; in particular, if Black plays Ka7 or Kb8, White has the same 7 king moves plus a pawn move. However, if Black selects b7, White will not be able to play Kc6, and will only have 6 king moves and 1 pawn move to choose from. In other words, a7 and b8, while different moves, do not differ in the strategy space available to White on his next move. They are excellent candidates for merging into a single *metaposition.*

The result of the merging is described by a Kuhn tree [Kuhn, 1953] (a game tree wherein the player with the right to move cannot distinguish between states linked with a dotted line), shown in Figure 4.2. Uncertainty has disappeared, at least officially; White knows where he is from his current strategy space, as no two child nodes can share the same move set (or they would be merged). Also, since the game is now one of perfect information, it makes sense to generate an *evaluation function* and start assigning each node a minimax value. The value of a metaposition node could be, for example, the minimum value across all the positions that make up the metaposition.

However, this definition is ill-suited to a generic game of Kriegspiel, as the player does not know their strategy space beforehand. The very essence of this game is that you do not know whether a move is legal until you try it. Even if this were not the case, the typical Kriegspiel midgame has a branching factor of 60-70 moves *for each player*, and many White moves will have a Black move that makes them impossible (or conversely, make new moves possible), thus generating a large number of strategy spaces and metapositions; hence, relatively few positions could be merged together. Therefore, we move from this definition of metaposition to a more generic one.

Definition. If $\mathbf{S}$ is the set of all possible game states and $\mathbf{I} \subseteq \mathbf{S}$ is the information set comprising all game states compatible with a given sequence of observations (referee's messages), a metaposition $\mathbf{M}$ is any *opportunely coded* subset of $\mathbf{S}$ such that $\mathbf{I} \subseteq \mathbf{M} \subseteq \mathbf{S}$. The strategy space for $\mathbf{M}$ is the set of moves that are legal in at least one of the game states contained in the metaposition. We then speak of *pseudolegal* moves, assumed to be legal from the player's standpoint but not necessarily so from the referee's. A metaposition is endowed with the following functions:

- a *pseudomove* function `pseudo` that updates a metaposition given a move try and an observation of the referee's response to it;
- a *metamove* function `meta` that updates a metaposition after the unknown move of the opponent, given the associated referee's response;
- an *evaluation* function `eval` that outputs the desirability of a given metaposition.

From this definition it follows that a metaposition is any superset of the game's information set (though clearly the performance of any algorithm will improve as $\mathbf{M}$ tends to $\mathbf{I}$). Every plausible game state is contained in it, but a metaposition can contain other states which are not compatible with the history. The reason for this is two-fold: on one hand, being able to insert (opportune) impossible states enables the agent to represent a metaposition in a very compact form, as opposed to the immense amount of memory and computation time required if each state were to be listed explicitly;

on the other hand, a compact notation for a metaposition makes it easy to develop an evaluation function that will evaluate whole metapositions instead of single game states. This is the very crux of the approach: metapositions give the player an illusion of perfect information, but they mainly do so in order to enable the player to use a Minimax-like method where metapositions are evaluated instead of single states. For this reason, it is important that metapositions be described in a concise way so that a suitable evaluation function can be applied.

It is interesting to note that metapositions move in the opposite direction from such approaches as Monte Carlo sampling, which aim to evaluate a situation from a significant subset of plausible game states. This is perhaps one of the more interesting aspects of the present research, which moves from the theoretical limits of several Monte Carlo approaches as stated, for example, in [Frank and Basin, 1998], and tries to overcome them. In fact, *a metaposition-based approach does not assume that the opponent will react with a best defense model*, nor is it subject to strategy fusion because uncertainty is artificially removed. [1]

The opportune coding must be one that will allow the algorithm to examine a metaposition as a single entity, without worrying about the single states contained in it. Any other way would be computationally intractable. In other words, *this coding is a single game state of a different game than Kriegspiel, a game with perfect information.* As shown in Section 4.2, Darkboard's metapositions make use of *pseudopieces* to this purpose, representing a metaposition as a single chessboard where allied pieces coexist with ghostly enemy pieces following their own rules for movement.

The definitions of the functions `pseudo` and *meta* are purposely vague except for their output having to satisfy the basic metaposition constraint $\mathbf{I} \subseteq \mathbf{M}$, that is having to contain every possible state in the updated information set.

We also make use of a simulated referee that generates virtual messages trying to predict the response of the actual referee. Together with metapositions and the functions that operate on them, we are able to construct a game tree and then evaluate it with a weighed Maximax algorithm that

[1]On the other hand, our newer Monte Carlo Tree Search player, described in the next chapter, also avoids the assumption of a best defense model.

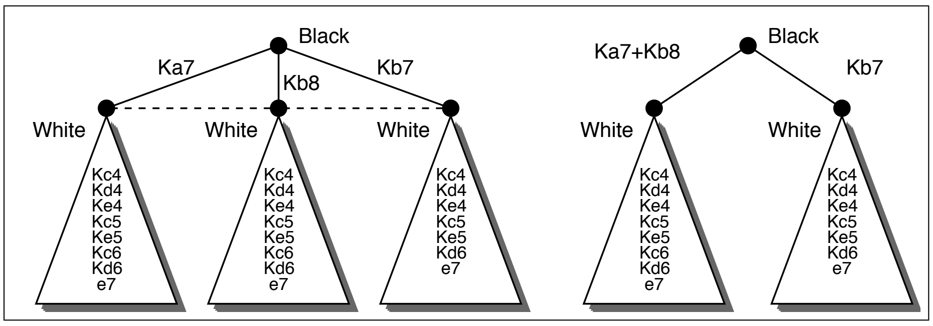

Black
Ka7
Kb8
Kb7
White
White
White
Kc4 Kd4 Ke4 Kc5 Ke5 Kc6 Kd6 e7
Kc4 Kd4 Ke4 Kc5 Ke5 Kc6 Kd6 e7
Kc4 Kd4 Ke4 Kc5 Ke5 Kd6 e7
Ka7+Kb8
Black
Kb7
White
White
Kc4 Kd4 Ke4 Kc5 Ke5 Kc6 Kd6 e7
Kc4 Kd4 Ke4 Kc5 Ke5 Kd6 e7

Figure 4.2: Partial Kuhn tree (left), with state merging and basic metapositions (right).

produces a good level of play.

4.2 Darkboard and metapositions

Darkboard is a game engine for playing Kriegspiel under the ICC ruleset (Cincinnati style). It is written in the Java programming language and runs on any computer with the Java Runtime Environment version 1.3.1 or later. The focus of its design is on the concepts of Player and Umpire as the main actors of a Kriegspiel game. By subclassing the former, one can represent both human and computer players, whereas by subclassing the latter one can add support for additional modes of play, such as LAN or Internet Chess Club matches, or different rulesets.

A simple view on the most important classes is given in Figure 4.3. Three artificial players have been implemented, one trying random pseudolegal moves, a slightly less random one doing the same but always capturing enemy pieces when given the chance (for example, when retaliating on captures or when pawn tries are announced), and finally the Darkboard class implementing the metaposition-based player. The first two players serve as benchmarking tools for gauging the effectiveness of Darkboard, especially its ability to checkmate during the endgame.

Currently, two subclasses of Umpire are available, LocalUmpire which allows for local play against humans or other artificial players, and RemoteUmpire which is used when the other player is not managed by the program

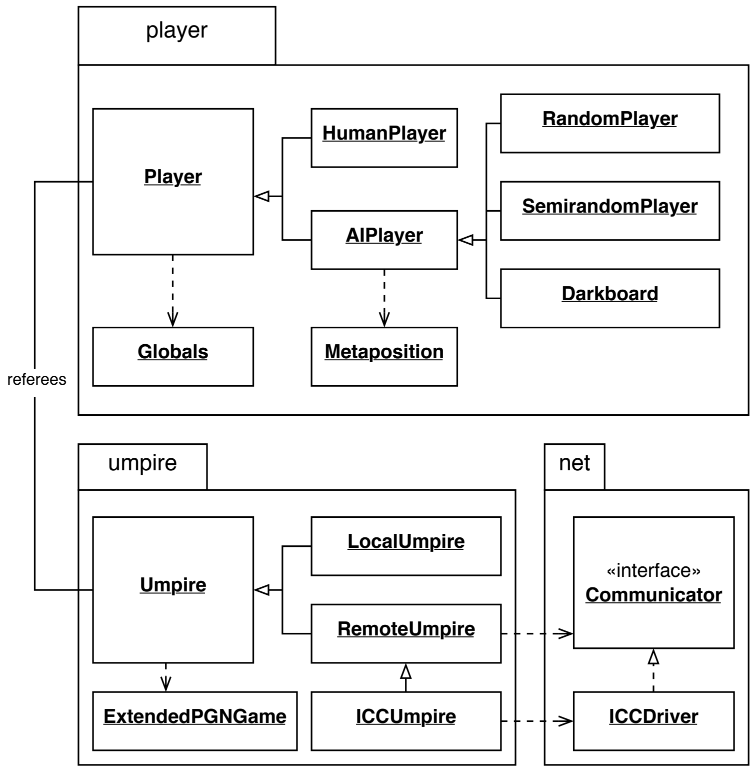


Figure 4.3: Simplified structure of the Darkboard engine.

itself, its only subclass being ICCUmpire for play on the Internet Chess Club. These remote umpires make use of a Communicator interface to separate low-level network management tasks from the higher-level rule enforcement routines. Umpire and its subclasses are endowed with a set of facilities for generating extended PGN games (a backward-compatible derivative of the PGN standard for representing games of Kriegspiel including details on rejected moves and the referee's messages). Local games may be started from any nonstandard chess position by using FEN strings, in which case the initial position of all pieces is assumed to be known by both players.

4.2.1 Representing metapositions

The **Metaposition** class represents a single metaposition, which is then used as building blocks for all of Darkboard's move selection and evaluation routines. Any subclass of Player can make use of metapositions, including HumanPlayer (which is a programming hook for graphical user interfaces accepting human input); for example, the list of current pseudolegal moves can be calculated with a metaposition so that obviously illegal moves are not forwarded to the server during ICC matches.

The program represents metapositions using *pseudopieces*, phantom pieces which act like regular chessmen but can spawn copies of themselves and step over fellow pseudopieces. Pseudopieces have been likened to concepts from quantum mechanics; they can be imagined as an enemy piece being in several locations at the same time. Each pseudopiece moves independently of the others, and all of them move on the opponent's turn. As they move, they spawn new pseudopieces of the same type on their path, and uncertainty increases; as friendly pieces move, they sweep any pseudopieces on their path, and uncertainty decreases. This approach allows to keep the data used to represent a metaposition down to a minimum while at the same time satisfying the definition in Sect. 4.1; in fact, because pseudopieces move just like real pieces, it is possible to obtain any possible state by simply replacing opportune pseudopieces with their real counterparts.

4.2.2 The main array

In Darkboard, a metaposition is represented by a 64-element one-dimensional byte array, where each byte represents one square of the chessboard. It could have been a two-dimensional 8x8 matrix, but performance dictated the use of a single array, especially because evolving a metaposition into another involves duplicating its data structure, and this happens very frequently.

Each byte in the array is actually a bitmask containing information about a single piece. Throughout Darkboard, each piece has a code number associated to it. Our metapositions limit themselves to recording whether at least one of the possible game states has an enemy queen in d4, for example. In fact, the lower 7 bits in each byte of the main array form a bitfield representing the presence of seven different pieces (the six piece types in chess plus the special piece 'empty'). It may appear strange to consider 'empty' as a piece, but metapositions in Darkboard are merely concerned with what is impossible or possible on a given square at a given point; and this includes whether a square being empty is possible or impossible. Bit 0 is set to 1 if there is a possibility for an enemy pawn to be in that square, and so on. The bitmask's eighth and uppermost bit is used to signal the presence of an allied piece on the square. When that bit is set, the other bits no longer represent a possible enemy piece; instead, by performing a simple bitwise AND operation to remove the first bit, we quickly obtain the piece code for the friendly piece; this is simpler than marking the corresponding piece bit and then using a lookup table.

It should be noted that the array bits are set to 1 *if* a piece of that type may occupy a given square, not *if and only if*. Just like metapositions themselves, the process of evolving a metaposition is an approximation of the real process, with trade-offs to allow the program to compute something useful within acceptable time limits. Darkboard's computations maintain the following invariant: *if the piece bit is set to 0, then that piece is guaranteed not to be there.* The reverse is not true, and Darkboard will sometimes mark enemy pieces as possible in places where, strictly speaking, they could not be; this is our implementation of the metaposition constraints in Sect. 4.1, as the resulting metaposition will contain every state compatible with the observations, and also states that are not, but are "close enough" to compatible states.

It is theoretically possible to split a metaposition into individual game states; in fact, it is strongly advised to do so when the information set is small enough to be treated explicitly. However, this is only really feasible in very limited scenarios, typically when the opponent only has one or two pieces left on the chessboard or when solving Kriegspiel problems in which the starting position is known beforehand. Moreover, the algorithm for dividing a metaposition is complicated by a series of technical difficulties such as the lack of information on pawn promotions, making it difficult to establish how many pieces and how many pawns are left.

4.2.3 The age array

The age array is another 64-element array, though its elements are of type *char*. Its main function is to keep information about the metaposition's history because, due to strategy fusion, the best move does not only depend on the current position, but also on what came before it. In practice, Darkboard needs more subtle information than the binary nature of the main array provides. Knowing that a piece may or may not be in a given square is obviously important, but not so much in the middle game as in the endgame. Middlegame metapositions contain so many positions that in many situations pretty much anything is possible, anywhere; the chessboard is a series of small, safe havens surrounded by a sea of uncertainty.

Every square has an associated age value, which generally represents *the number of moves since Darkboard collected information about that square.* This has a broader meaning than just "since the player last visited the square", because physically reaching a square or traveling over it is not always necessary to infer what it contains. For example, the absence of pawn tries (and checks), if the pawn and king are placed in such a way that the former cannot be protecting the latter, may indicate empty squares just as effectively, and their ages would be cleared back to zero.

The age array is involved in several calculations, two of which are especially important to the program: first, squares with high age values are seen as undesirable by the evaluation function, thus encouraging the player to visit them, and secondly, high age is associated with danger when estimating the safety level of a friendly piece. For this reason, when Darkboard finds that a path is obstructed, determining that a piece must be somewhere, it may artificially raise a square's age level to represent increased danger.

4.2.4 Other information

A metaposition also includes several more fields, some of which are typical of a normal chess game (such as castling information), whereas others are unique to Kriegspiel (like the amount of captured material). These data are stored in arrays for faster copying, and include the following:

- Color information (are we White or Black?)
- Castling information (kingside, queenside, both, neither).
- Captured pieces and pawns.

- Minimum and maximum pawn number on each file, inferred through captures and pawn tries. Aside from providing a positive bonus when a file is pawn-free, this is especially important when considering *pawn control* (see next section).
- Last pawn try count for both players.
- Total age count. This is the sum of the age values for each square on the chessboard, stored in a convenient field for performance reasons as it is often needed.
- Depth information. When a metaposition is evolved with a player's pseudomove, the depth is copied over; when it is evolved with the opponent's metamove, depth is increased by one. When building a pseudo-game tree, this field allows the evaluation function to know how deep the search is.

4.3 Working with metapositions

As metapositions are collections of game states, we define several useful operations on metapositions to obtain information on those states, or change them. These operations include:

- Editing the metaposition (i.e. amending the information it contains, thus extending or narrowing the information set).
- Updating the metaposition after a successful player move.
- Updating the metaposition after an unsuccessful player move (illegal move).
- Updating the metaposition after the opponent's metamove and its associated messages.
- Generating the possible pseudomoves for the player to choose from.
- Calculating useful facts about the metaposition, including a protection matrix, and various estimates about the safety of each piece.

- Evaluating a metaposition.

All these operations would be trivially executed if the metaposition were a set of explicitly listed states; for example, listing all pseudolegal moves would translate to checking for moves that are legal for at least one game state in the set. Updating the set with a new message from the referee would be equally simple, merely requiring the algorithm to discard all states that were incompatible with the message and generating every possible move satisfying the condition from the remaining states.

Unfortunately, as a metaposition represents a compact grouping of a very large number of positions which cannot be told apart from one another, it is clear that updating such a data structure is no trivial task; in truth, despite being a simplification of the real information set, this process does account for the better part of Darkboard's computation time, more than the evaluation function itself. Clearly, the specific mechanics of such operations depend on how metapositions are being represented.

4.3.1 Move generation

Generally speaking, the move generation function is of the type

$$(\textbf{Metaposition} \times \mathbb{B}) \to \textbf{Move}[\,],$$

as it accepts a metaposition and a boolean and returns an array of Move objects containing the possible pseudolegal moves for the artificial player. The boolean parameter specifies whether the move is *top-level* or not; that is, whether the input metaposition represents the current state of the chessboard or a possible future evolution of it. When the top-level parameter is false, any matching move is included within the output Vector; but if the move is top-level, *banned moves* (pseudomoves tried and found to be illegal in the current turn) will not be included. Darkboard is actually a little smarter than that, and whenever a move fails, it will not only mark the latest move as banned, but also any moves that are trivially illegal, as well (i.e. if Ra1-a5 fails, there is no point in trying Ra1-a8 except when responding to a check).

The move generation algorithm reasons like a traditional chess algorithm with the same purpose. Each piece travels as far as it can, stopping only when it meets an edge, a friendly piece, or a square whose bitfield has the Empty bit not set. If there are any pawn tries, the algorithm will generate the corresponding moves except for those where the target square does not have any piece bit set.

There are further checks that may be performed to increase the accuracy of a metaposition. For example, Darkboard performs *pawn control*, an operation which makes sure that no pieces of either side can move through a file where an enemy pawn is known to exist.

4.3.2 Updating after a legal move

Darkboard is provided with three different update algorithms, one for manipulating a metaposition after a legal move, one for illegal moves, and one for the opponent's metamoves. All of the above accept a metaposition and the appropriate umpire messages as their inputs, and return a new, updated metaposition. It may appear strange that the heart of the program's reasoning does not lie in the evaluation function but in these algorithms: after all, their equivalent in a chess-playing software would trivially update a position by clearing a bit and setting another. However, the evaluation function's task is to evaluate the current knowledge. The updating algorithms compute the knowledge itself, and given Kriegspiel's strongly imperfect information, it is imperative to infer as much information as possible in the process.

The first algorithm updates a metaposition after a legal move by the player: it accepts a starting metaposition as its input, a move which is assumed to be legal, and the following information: *capture type* (which can take one of the following values: *noCapture*, *capturePawn*, *capturePiece*), *check1/check2* (as there can be up to two simultaneous checks; accepted values are *noCheck*, *knightCheck*, *rankCheck*, *fileCheck*, *shortDiagonalCheck*, *longDiagonalCheck*), *pawn tries* (for the opponent, after this move; the player's pawn tries are handled as part of the opponent's move evolution).

The function performs a few simple operations first, such as setting the visited squares to empty status and clearing their age values. Then, if the player captured something, it updates the count of captured material. If a pawn is captured, and the maximum pawn count for its file was 1, all pawns are removed from that file. In particular, if the amount of captured pawns reaches 8, pawns are removed from the chessboard altogether. Unfortunately, the same cannot be done with pieces when their capture count reaches 7, as pawns may have since attained promotion. However, if the player manages to capture 15 times, everything is cleared off the main array but king bits. In this case, obviously, the metaposition's accuracy increases drastically.

This being said, dealing with checks is the only non-trivial task here. The previous king's locations are scanned one by one, and only those compatible with the latest move and check type are allowed to remain. Actually, reality is a little more complicated than this, and the process does not always prove to be straightforward, due to captures and *discovery checks*, wherein the piece that moves is not the one threatening the king. Therefore, the algorithm proceeds in this way:

- In the event of a double check, there is no ambiguity whatsoever; the piece responsible for the discovery check is also uniquely determined. The intersection of the two sets of squares for the two checks provides the king's exact location.

- If the check type is compatible with the piece being moved, *and that piece is not a pawn*, it cannot be a discovery check. Simply remove the king's current locations that do not match the check type. The king can never be found in the opposite direction of the piece's movement (or it would have been in check even before the move); and it can only be found "at 12 o' clock" if a capture also took place (i.e. the piece that protected it was just captured). This is especially important with diagonal checks, as 'long' or 'short' diagonal refers to the king's perspective, not the attacking piece's.

- If the check type is not compatible with the piece being moved, such as a file check when a knight was moved, look for discovery check candidates. Fire 'beams' from the piece's starting square along every direction that is compatible with the check type. If the beam reaches an allied piece that is compatible with the check type, we have found a candidate. At least one candidate is guaranteed to be found, but in rare cases, depending on the placement of the opponent's pieces, there could be two or more candidates.

For each candidate, there exists a set of target squares for the king to be in. The set is a segment that extends from the moved piece's starting square in the opposite direction than the candidate. We therefore rule out any current location that does not belong to any of the candidates' sets.

- The worst case scenario happens when the player has moved a pawn and the umpire announces a diagonal check. Because pawns do not capture the same way in which they move, it could be either a genuine pawn check, or a discovery check from a bishop or queen behind the pawn. As a consequence, the algorithm will try both schemes and rule out any square that matches neither.

- In order to narrow the choices down even more, pawn control could and should be taken into account with file checks.

4.3.3 Updating after an illegal move

Extracting information from illegal moves is extremely important because, unlike most other umpire messages, such information is asymmetric; there is no way for the opponent to know what move was rejected (and on the ICC, there is no way to know that an opponent's move was rejected to begin with). If we were dealing with a theoretical metaposition, an information set, we would simply drop any position that considered the move as legal. Unfortunately, such a subset consists of highly diverse positions, which are impossible to fully describe with Darkboard's data structures. The following actions can, however, be taken with relative ease.

- If the king is the only enemy piece left, a failed king move will narrow its possible locations to five squares at most. A failed pawn push can pinpoint the king's location.

- If the tentatively moved piece is not protecting its king (that is, the king cannot be found along any of the eight compass directions from its starting square, except the very direction it was trying to take) and the path was two squares long (or one for pawn moves), then the middle square obviously contains something. Its Empty bit is cleared and its age increased.

- If the move spanned across more squares, Darkboard will then create and register a *power move*. A power move is a new pseudomove, whose piece and starting square is the same as the last failed move, but has a shorter scope, usually one square shorter than the original move unless the new destination square is certainly empty, in which case it is further shortened until a possible target square is found. For example, if Ra1-a8 fails and enemy pieces are possible in a7, the power move Ra1-a7 is generated.

  Power moves play an important role. When Darkboard builds a pseudo-game tree, interpreting metapositions as positions of a perfect information game, it will try to evolve the current metapositions by predicting the future umpire's messages. *When evolving a metaposition through a move that is a power move, Darkboard will assume it is a capturing move.* This reflects the common tactic for a player to try long moves, and upon hearing the umpire reject them, shorten them one square at a time until they end up capturing something.

4.3.4 Updating after the opponent's move

To evolve an information set along an opponent's unknown move means to generate every possible evolution (compatible with the umpire's next message) for every position in the set; the union of the resulting sets, barring duplicates, represents the new information set. Again, Darkboard employs an approximation of the real thing that makes the opponent more mobile than it really is. However, it guarantees that every position in the information set is still part of Darkboard's representation.

For each square, we treat each possible piece as a real, existing piece (pseudopiece) and move it according to its rules, just like we generate pseudolegal moves for Darkboard. To this end, a support chessboard is employed, which starts out without any enemy pieces on it. For each possible move, the corresponding piece bit is set on the destination square of the support chessboard. When this phase is over, a bitwise OR operation between the source metaposition and the support chessboard returns the intended evolution.

This is arguably Darkboard's most processor-intensive task, as the number of potential opponent-controlled squares normally outnumbers the number of friendly pieces. In fact, it can be easily seen that on a chessboard of size k and k^2 squares, and assuming that the number of potential opponent squares is, most of the time, $O(k^2)$, with pieces able to move $O(k)$ squares in one or more directions, the resulting complexity is $O(k^3)$.

The algorithm performs a few additional refining steps in the process, among which are the following.

- If the opponent captured a piece, every bit for that square is cleared, including the Empty one, before doing anything else. Also, the pseudo-pieces are only permitted to move to the targeted square, meaning that after the algorithm has run, the square will contain exclusively the piece types that could be responsible for the capture. This can prove useful if the attacking piece is immediately captured back (though, currently, Darkboard does not try to guess which pieces it captures).
- As a corollary of the above point, if a capture takes place but the umpire had mentioned no pawn tries for the opponent, pawns are not considered.
- If a pawn is a potential capturing piece, the minimum pawn count for the adjacent files is decreased by one and the maximum pawn count for the target file is increased by one.
- If a square has the Empty bit not set, meaning that it certainly contains an enemy piece, but at least one move (for any of the possible pseudo-pieces on that square) can move it away from there, the Empty bit is set, otherwise it stays unchanged.
- A pseudo-king will never move to squares that are certainly threatened.
- If the move causes a knight check, only knights are moved.
- If the move does not cause a check, the squares around the friendly king are cleared off the appropriate piece bits (queens and bishops on its diagonals, etc.)

- Pawn control applies normally to pseudo-pieces to at least limit their tendency to behave like "ghosts" that other pseudo-pieces can move through.

- After the algorithm has run, each square is checked. If it is certainly empty, its age is set to 0; else, if it is assuredly non-empty, its age is set to a high, hard-coded constant value; else, its age is increased by one. Also, the total age field is recomputed and updated.

As mentioned, enemy pseudopieces are only blocked by friendly pieces, squares with the Empty bit not set and pawn control. This approximation leads to a weak interpretation of the first two or three moves, wherein pieces are assumed to be able to develop faster than they actually can. However, by the time the first umpire message arrives, the situation will have sufficiently stabilized, and no special treatment seems to be necessary for the very first few moves.

4.4 The move selection routines

Darkboard's core is the move selection algorithm. The main purpose of information sets, and by extension metapositions as well, is to make the construction of a game tree possible even in the context of imperfect information games. In the previous sections several functions have been described that model the possible transitions between metapositions and their evolutions. Such functions can be used to generate child nodes from root nodes, representing both the player and the opponent's moves. The selection algorithm will therefore construct a (pseudo-)game tree and use it to determine its next move.

When dealing with Chess, the selection algorithm is some approximation of a minimax. The reasons why minimax does not apply to Kriegspiel have been discussed, and need not be repeated in full. What matters here is how to proceed in evaluating a metaposition tree to obtain a move that is not necessarily the *best* (which is an empty word in this game as a whole), but a *reasonable* one.

The first fact to consider is that metapositions nodes inside a game tree will be evaluated by an evaluation function. Thus, it appears there will be some function

$$f : (\mathbf{Metaposition} \times \mathbb{B}) \to \mathbb{R}$$

that evaluates metapositions, also accepting a boolean representing whose turn it is to move. But, on closer inspection, the above definition, taken straight from Chess, is not adequate for the task at hand.

Chess evaluation functions are built to judge on a given position, which is a snapshot of the game in progress; past events are meaningless in that context. On the other hand, it has been shown that the optimal strategy for an imperfect information game does not only depend on the current situation, but also on the events that led to it, that is the full history of the game. Currently, Darkboard takes into account the state of the chessboard before and after the move to be evaluated, so that, for example, the piece which was just moved is evaluated as more endangered than the others. Therefore, its evaluation function is of the type

$$f : (\mathbf{Metaposition} \times \mathbf{Move} \times \mathbf{Metaposition}) \to \mathbb{R}.$$

4.4.1 Game tree structure

Since a metaposition's evolution depends exclusively on the umpire's messages, clearly it becomes necessary to anticipate the umpire's next messages if a game tree is to be constructed. Ideally, the game tree would have to include every possible umpire message for every available pseudomove. Unfortunately, a quick estimate of the number of nodes involved rules out such an option. It is readily seen that:

- All pseudomoves may be legal (or they would not have been generated by the previous algorithms).
- All pseudomoves that move to non-empty squares can capture (except for pawn moves), and under ICC rules, we would need to distinguish between pawn and piece captures.
- Most pseudomoves may lead to checks.
- Some pieces may lead to multiple check types.
- The enemy may or may not have pawn tries following this move.

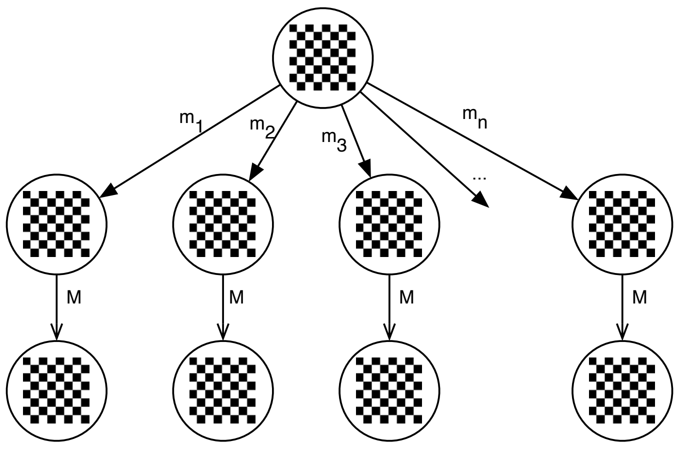


Figure 4.4: Two-ply game tree, $m_1, \ldots, m_n$ are pseudomoves, M's represent metamoves; also denoted with different arrow heads.

A simple multiplication of these factors may yield several dozens potential umpire messages for any single move. But worst of all, such an estimate does not even take into account the possibility of *illegal moves*. An illegal move forces the player to try another move, which can, in turn, yield more umpire messages and illegal moves, so that the number of cases rises exponentially. Furthermore, the opponent's metamoves pose the same problem as they can lead to a large number of different messages.

- On the opponent's turn, most pieces can be captured (all but those marked with a safety rating of 1).
- The king may typically end up threatened from all directions through all of the 5 possible check types.
- Again, pawn tries may or may not occur, and can be one or more.

For these reasons, any metaposition will be only updated in exactly one way, and according to one among many umpire messages. This applies to both the player's pseudomoves and the opponent's hidden metamoves, so that the tree can be summarized as in Figure 4.4.

As a consequence, the tree's branching factor for the player's turns is equal to the number of potential moves, but it is equal to 1 for the opponent's own

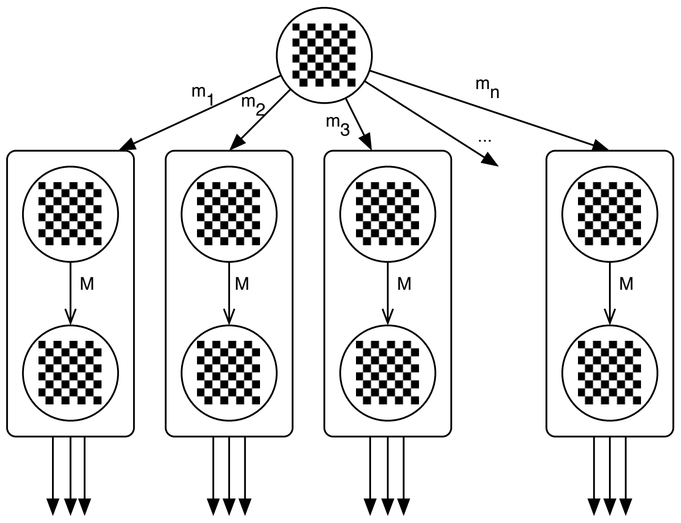


Figure 4.5: Compact form for the game tree; each node but the root contains two metapositions.

moves. This is equivalent to saying that Darkboard does not really see an opponent, but acts like *an agent in a hostile environment.* It also means that the opponent's metamove can be merged with the move that generated it, so that each level in the game tree no longer represents a ply, but a full move (see Figure 4.5).

Interestingly, the branching factor for this Kriegspiel model is significantly smaller than the average branching factor for the typical chess game, seeing as in chess either player has a set of about 30 potential moves at any given time, and Kriegspiel is estimated to stand at approximately twice that value. Therefore, a two-ply game tree of chess will feature about $30^2 = 900$ leaves, whereas Darkboard's tree will only have 60. However, the computational overhead associated with calculating 60 metapositions is far greater than that for simply generating 900 chessboards, and as such some kind of pruning algorithm will be needed.

4.4.2 Umpire prediction heuristics

Darkboard generates the umpire messages that follow its own moves in the following way.

- Every move is always assumed to be legal. Most of the time, an illegal move just provides information for free, so a legal move is usually the less desirable alternative.
- The player's moves do not generally capture anything, with the following exceptions:
  - Pawn tries. These are always capturing moves by their own nature.
  - Non-pawn moves where the destination square's Empty bit is not set, since the place is necessarily non-empty.
  - Power moves obtained from previous illegal moves (see 4.3.3). This applies to the root metaposition only, as hypothetical illegal moves cannot be generated.
- If any of the above apply, the captured entity is always assumed to be a pawn, unless pawns should be impossible on that square, in which case it is a piece.
- Pawn tries for the opponent are generated if the piece that just moved is the potential target of a pawn capture.

On the other hand, the following rules determine the umpire messages that follow a metamove.

- The opponent never captures any pieces, either. The constant risk that allied pieces run is represented by danger ratings instead, which affect the evaluation function by changing the value of a piece.
- The opponent never threatens the allied king. Danger ratings encourage the king's protection.
- Pawn tries for the player are never generated.

The above assumptions are overall 'reasonable', in that they try to avoid sudden or unjustified peaks in the evaluation function. Captures are only considered when they are certain, and no move receives unfair advantages over the others. There is no concept of a 'lucky' move that reveals the opponent's king by pure coincidence, though if that happens, Darkboard will update its knowledge accordingly.

Even so, the accuracy of the prediction drops rather quickly. In the average middle game, the umpire answers with a non-silent message about 20-30% of the time. Clearly, the reliability of this method degrades quickly as the tree gets deeper, and the exploration itself becomes pointless past a certain limit. At the very least, this shows that any selection algorithm based on this method will have to weigh evaluations differently depending on where they are in the tree; with shallow nodes weighing more than deeper ones.

4.4.3 The basic decision algorithm

Now that the primitives have been discussed in detail, it is possible to describe the selection algorithm for the Darkboard player. We shall first discuss the generic version, and then introduce the pruning algorithm that makes the player efficient enough to handle fast online play on the ICC. Such separation is not only for the sake of clarity; in fact, both algorithms have their place in Darkboard, and either one is used depending on the situation. The generic algorithm makes for shallow, but exhaustive searches in the game tree, whereas the pruning-enhanced one allows deeper, but approximated exploration.

The whole stratagem of metapositions was aimed at making traditional minimax techniques work with Kriegspiel. Actually, since MIN's moves do not really exist (MIN always has only one choice) if we use the compact form for the tree, as described in the last section, the algorithm becomes a *weighed maximax.* Maximax is a well-known criterion for decision-making under uncertainty. This variant is weighed, meaning that it accepts an additional parameter $\alpha \in]0, 1[$, called the *risk coefficient.* The algorithm also specifies a maximum depth level k for the search. Furthermore, we define two special values, $\pm\infty$, as possible output to the evaluation function *eval.* They represent situations so desirable or undesirable that they often coincide with victory or defeat, and should not be expanded further.

The selection algorithm makes use of the following functions:

- *pseudo:* (**Metaposition** $\times$ **Move**) $\rightarrow$ **Metaposition**, which gener-

```
function value (metaposition met, move mov, int depth) : real
begin
    metaposition met2 := pseudo(met, mov);
    real staticvalue := eval(met, mov, met2);
    if (depth $\leq$ 0) or (staticvalue = $\pm\infty$)
        return staticvalue
    else
    begin
        //simulate opponent, recursively find MAX.
        metaposition met3 := meta(met2);
        vector movevec := generate(met3);
        real bestchildvalue := $\max_{x \in movevec}$ value(met3, x, depth-1);
        //weighed average with parent's static value.
        return (staticvalue*$\alpha$)+bestchildvalue*$(1-\alpha)$
    end
end.
```

Figure 4.6: Pseudocode listing for value function.

ates a new metaposition from an existing one and a tentative move, simulating the umpire's responses as described in the last section.

- *meta:* **Metaposition** → **Metaposition**, which generates a new metaposition simulating the opponent's move and, again, virtual umpire messages.
- *generate:* **Metaposition** → **Vector**, the move generation function.
- *eval:* (**Metaposition** × **Move** × **Metaposition**) → $\mathbb{R}$, the evaluation function, accepting a source metaposition, an evolved metaposition (obtained by means of *pseudo*), and the move in between.

The algorithm defines a *value* function for a metaposition and a move, whose pseudocode is listed in Figure 4.6. The actual implementation is somewhat more complex due to optimizations that minimize the calls to *pseudo*.

It is easily seen that such a function satisfies the property that a node's weight decreases exponentially with its depth. Given the best maximax sequence of depth d from root to leaf $m_1, \ldots, m_d$, where each node is provided with static value $s_1, \ldots, s_d$, the actual value of m_1 will depend on the static values of each node m_k with relative weight α^k. Thus, as the accuracy of Darkboard's foresight decreases, so do the weights associated with it, and the engine will tend to favor good positions in the short run.

Parameter α is meant to be variable, as it can be used to adjust the algorithm's willingness to take risks. Higher values of α lead to more conservative play, whereas lower values will tend to accept more risk in exchange for possibly higher returns. Generally, the player who is having the upper hand will favor open play whereas the losing player tends to play conservatively to reduce the chance of further increasing the material gap. Material balance and other factors can therefore be used to dynamically adjust the value of α during the game, though this feature is largely untested in Darkboard as of yet.

4.4.4 The enhanced decision algorithm

The previous algorithm suffers from serious performance issues if forced to push its search 3 or more levels down the game tree. For this reason, it is used with a default depth level of 2, and is called upon when any of the following apply:

- The umpire announced captures, checks or pawn tries. Statistics show that all of the above tend to happen in clusters, so that the likelihood of a capture following another capture is much higher than normal. Since our usual assumptions about future umpire messages may not prove reasonable anymore under such circumstances, a deep analysis appears fruitless here, and a shallow, but complete and fast search seems more convenient.

- Under tight time control. Darkboard has a built-in time control manager, and will try to avoid running out of time any way it can. There are several precautions the engine takes under different stress levels, such as reducing the number of metapositions it searches through, but as a last resort, when time is running very short, Darkboard will switch to the shallow but faster search and use it until time climbs back to a safe level.

The enhanced algorithm must necessarily discard some branches of the tree and concentrate on the most promising ones in order to delve deeper into the tree. As a consequence, simple depth-limited recursion does not suffice here, and instead *the number of evaluated metapositions* is used to estimate how far to push the search.

The concept of *killer moves* is well-known in the literature of artificial chess players [Akl and Newborn, 1977]. A move that has been found to be advantageous somewhere in the game tree is likely to be a strong move even in a different context. Chess programs combine killer heuristics with *alpha-beta pruning* to largely reduce the number of positions that need evaluating. Unfortunately, pure alpha-beta pruning is not applicable to a maximax Kriegspiel tree; however, something resembling killer moves seems to be more feasible. The fact itself that Kriegspiel's branching factor is quite large also means that most metapositions belonging to the same tree will share many common moves. The algorithm should evaluate each move the first time it occurs, and remember good moves when they occur again.

Other than the familiar α, we introduce two further integer coefficients: *newMoves* and *oldMoves*. *These coefficients represent the number of branches that will be expanded.* At most *newMoves* branches will be explored whose associated moves do not yet appear in the table; and at most *oldMoves* will be explored among those that already do. The algorithm also accepts a *maxPositions* argument that specifies how many metapositions should be, at most, evaluated, though this is just an estimate and the program's execution will not stop once that number is met.

A simplified listing is given in Figure 4.7; the real version is both longer and more complicated, accepting more parameters to allow major performance gains due to repeated *eval* and *pseudo* calls.

This function can typically reach between four and seven levels deep into the game tree with *maxPositions* set to 5000, *newMoves* set to 5 and *oldMoves* set to 3, bringing good results in practice.

4.5 The evaluation function

Generally speaking, the evaluation function for a chess program includes three main components: material, mobility, and positional issues.

```
function value2 (metaposition met, move mov, int maxPositions) : real
begin
    metaposition met2 := pseudo(met, mov);
    real staticvalue := eval(met, mov, met2);
    if (maxPositions $\leq$ 1) or (staticvalue = $\pm\infty$)
        return staticvalue
    else
    begin
        //simulate opponent, recursively find MAX.
        metaposition met3 := meta(met2);
        vector movevec := generate(met3);
        vector old, new, selected;
        //separate old and new moves.
        foreach x $\in$ movevec do
            if hasEntry(x) then add(x, old) else add(x, new);
        //add entries to the table for the new moves.
        //their scores are the difference with their parent's eval.
        foreach x $\in$ new do
            putEntry(x,eval(met3, x, pseudo(met3, x)) - staticvalue);
        //sort the two move vectors with their values in the table.
        $\textbf{sort}_{x \in \text{new}}$ with getEntry(x);
        $\textbf{sort}_{x \in \text{old}}$ with getEntry(x);
        maxPositions -= vectorSize(new); //update position count.
        //put the best from either vector into selected, and expand.
        putIntoVector(new, selected, newMoves); //up to newMoves elements.
        putIntoVector(old, selected, oldMoves); //up to oldMoves elements.
        maxPositions /= vectorSize(selected); //split maxPositions equally.
        //now proceed just like the simpler algorithm.
        real bestvalue := $\max_{x \in \text{selected}}$ value(met3, x, maxPositions);
        //weighed average with parent's static value.
        return (staticvalue*$\alpha$)+bestvalue*$(1 - \alpha)$
    end
end.
```

Figure 4.7: Pseudocode listing for the pruning-enhanced function.

Darkboard's evaluation function also has three main components that it will try to maximize throughout the game: *material safety*, *position*, and *information.*

4.5.1 Material safety

Material safety is a function of type $(\mathbf{Metaposition} \times \mathbf{Square} \times \mathbb{B}) \rightarrow [0, 1]$. It accepts a metaposition, a square and a boolean and returns a safety coefficient for the friendly piece on the given square. The boolean parameter tells whether the piece has just been moved (as it is clear that a value of *true* decreases the piece's safety). A value of 1 means it is impossible for the piece to be captured on the next move, whereas a value of 0 indicates a very high-risk situation with an unprotected piece.

It should be noted, however, that material safety does not represent a probability of the piece being captured, or even an estimate of such an event; its result simply provides a reasonable measure of the urgency with which the piece should be protected or moved away from danger.

Material safety is obtained by means of a support function, *material danger*. It is a function with the same contract as material safety, but with inverted meaning, wherein 0 means no danger and 1 indicates the highest danger level. Material danger is rather easy to calculate, and is based off of the age matrix values for the squares surrounding a given piece, as well as the protection level of that piece.

4.5.2 Position

Darkboard includes the following factors into its evaluation function, some of which are regularly featured in traditional chess-playing software:

- A pawn advancement bonus. In addition, there is a further bonus for the presence of multiple queens on the chessboard.
- A bonus for files without pawns, and friendly pawns on such files.
- A bonus for the number of controlled squares, as computed with the protection matrix. This factor is akin to mobility in traditional chess-playing software, but its usage in Darkboard is still rather unrefined; in particular, setting this weight too high will cause the pieces to scatter excessively all over the chessboard, weakening the defensive structure. Practical results show that this factor should vary over time and depending on who is winning.

In addition, the current position also affects material rating, as certain situations may change the values of the player's pieces. For example, the value of pawns is increased if the player lacks sufficient mating material.

An additional component is evaluated when Darkboard is considering checkmating the opponent. A special function represents perceived progress towards winning the game, partly borrowed from [Bolognesi and Ciancarini, 2004], together with a matrix associating squares to values encouraging the player to push the king towards locations where mating is easier.

4.5.3 Information

Darkboard will attempt to gather information about the state of the chessboard, as the evaluation function is designed to make information desirable (precisely, it is designed to make the lack of information undesirable). Darkboard's notion of information gathering coincides with reducing a computable function, which the program calls *chessboard entropy* (E). This definition is not directly related to those used in physics or Information Theory, but its behavior resembles that of an entropy function in that:

- The function's value increases after every metamove from the opponent, that is $(m_2 = \text{meta}(m_1)) \Rightarrow E(m_2) \geq E(m_1)$.
- The function's value decreases after each pseudomove from the player, that is $(m_2 = \text{pseudo}(m_1, x \in \mathbf{Move})) \Rightarrow E(m_2) \leq E(m_1)$.

Therefore, the chessboard entropy is constantly affected by two opposing forces, acting on alternate plies. We can define $\Delta E(m,x), m \in \mathbf{Metaposition}, x \in \mathbf{Move}$ as $E(\text{pseudo}(\text{meta}(m,x))) - E(m)$, the net result from two plies. Darkboard will attempt to minimize ΔE in the evaluation function. In the beginning, entropy increases steeply no matter what is done; however, in the endgame, the winner is usually the player whose chessboard has less entropy.

Entropy is computed as follows, using a set of constant values as well as the age matrix. For each piece and each square, a negative constant is given representing how undesirable would be to have that piece on that square. These values are generally small, with the exception of enemy pawns on the player's second or third ranks, close to promoting (a highly undesirable situation). In this way, to each square is associated an *undesirability value*, defined as the sum of the negative constants for any enemy piece whose existence is possible on that square. Actually, Darkboard speeds up the process by precalculating those sums for each and every combination of enemy pieces to be found on a square.

It is then given a function $f_E : \mathbf{N} \rightarrow [0, 1]$, monotone and non-decreasing, with the constraint $f_E(0) = 0$. The parameter in f_E is the age matrix's value for a given square, and the function itself models the increase in uncertainty over time. The entropy for a metaposition is then computed as the sum, for each square, of that square's undesirability value multiplied by $f_E(x)$, where x is the square's age value. It is easily seen that any function matching the mentioned constraints satisfies the two properties given in the beginning. As *pseudo* increases age values, entropy will increase; and as *meta* clears squares, entropy decreases.

4.5.4 Stalemate detection

Stalemate is an additional challenge in Kriegspiel, unlike regular chess where it can be predicted with ease. As it is impossible to generate the opponent's move, it is also difficult to estimate when the opponent has run out of moves; it is even more unfortunate that stalemate occurrences are directly proportional to the amount of friendly material on the board, meaning that it is easy to turn a major victory into a draw (possibly, also a reason why it could be more convenient, at times, to promote pawns to something other than queens in the endgame). Human players face this problem as well, even though the statistics on the ICC do not show it fully because most humans tend to resign when they are left with the king alone. If a Kriegspiel world championship existed, we would probably see much more stubborn defense and many more stalemates.

The program described in [Bolognesi and Ciancarini, 2004] deals with the stalemate issue when using metapositions to solve the KQK endgame (king and queen versus king), as stalemate may occur frequently in this endgame. Darkboard's algorithm is similar in nature; it looks for 'singleton' kings without neighbors. However, the computational cost for this algorithm is not negligible, since it needs to be repeated for every single metaposition. For this reason, the program only performs the test when two or fewer opposing pieces are left, other than the king. It would not make much sense, either, to check for stalemate when the opponent clearly has plenty of movement options left. Mistakes still happen when the king is not alone, though they are not always easy to predict, even for humans.

4.6 Experimental results and conclusions

We remark that the ruleset used for our program is the one enforced on the Internet Chess Club, which currently hosts the largest Kriegspiel community of human players. Our metaposition-based Kriegspiel player was the first artificial player capable of facing human players over the Internet on reasonable time control settings (three-minute games) and achieve above average rankings, with a best Elo rating of 1814 which placed it at the time among the top 20 players on the Internet Chess Club. Darkboard played 5724 games in 2006, winning 2958 (52%), drawing 997 (17%), and losing 1769 (31%) games over a period of four months. We note that Darkboard plays an average of only 1.415 tries per move, and therefore it does not use the advantage of physical speed to try large amounts of moves at the expense of human players.

Darkboard defeats a random-moving opponent approximately 94.8% of the time. The random player maintains and updates a metaposition in order to have access to a list of pseudolegal moves to choose from, but the actual choice is random among the possible moves.

We also define a second benchmark player called the *semi-random* player as a stronger test case for Darkboard. This player employs basic heuristics in order to select a move under certain conditions. Whenever a capture is announced, the player will first try all pseudolegal moves which allow the player to retaliate on the capture (that is, all capturing moves that have the location of the last capture as their destination square). If several moves match this condition, they are attempted in random order. Secondly, if pawn tries are announced, the player will randomly try every capturing move using its pawns instead of considering the other moves. Darkboard defeats the semi-random player approximately 79.3% of the time.

Against both test players, the games which are not won by Darkboard are draws by either stalemate or repetition.

Darkboard won the Gold medal at the Eleventh Computer Olympiad which took place from May 24 to June 1, 2006 in Turin. The player defeated an improved version of the Monte Carlo player described in [Parker et al., 2005] with a score of 6-2.

This Darkboard is referred to as version 1.0; its chief limit is the large amount of domain knowledge required to code the program. The next chapter describes the Monte Carlo Tree Search techniques behind Darkboard 2.0, currently the only program that has proven stronger than this metaposition-based player. While Darkboard 2.0 does not use metapositions in the midgame, the concept will be again essential in chapters 6 and 7, when dealing with the endgame.

Chapter 5

A Monte Carlo Tree Search approach

In this chapter, we describe a different approach, based on Monte Carlo Tree Search (MCTS). This method has brought significant improvements to the level of computer players in games such as Go, and it has been used to play imperfect information games as well, but there are certain games with particularly large trees and reduced information in which this class of algorithms can fail, especially in the presence of long matches, dynamic information and complex victory conditions. In this paper we explore the application of MCTS to Kriegspiel and compare it to the minimax-based player described in the previous chapter. We provide three Monte Carlo methods, starting from a naive textbook transposition and moving to more experimental versions of increasing strength for playing the game with little specific knowledge. We obtain significantly better results with a considerably simpler logic and less domain-specific knowledge.

5.1 Introduction

Imperfect information games provide a good model and test bed for many real-world problems and situations involving decision making under uncertainty. They typically involve a combination of complex tasks such as heuristic search, belief state reconstruction and opponent modeling, and they can be very difficult for a computer agent to play well. 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. Kriegspiel has several features that make it interesting: firstly, its rules are identical to those of a very well-known game and only 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; and finally, the nature of uncertainty is entirely dynamic. This differs from other games such as Phantom Go or Stratego, 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 this chapter we present the first full application of Monte Carlo tree search to the game of 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 players considerably. Since Kriegspiel shares the two traits of being a large game and a difficult one to express with an evaluation function (unlike its perfect information counterpart), it is only natural to test a similar approach. This would also allow to reduce the amount of game-specific knowledge used by current programs by a large amount.

The chapter is organized as follows. Section 5.2 contains a high-level introduction to Monte Carlo Tree Search. We then describe our MCTS approaches in Section 5.5, showing how we built three Monte Carlo Kriegspiel players of increasing strength. These players are then described in greater detail in Sections 5.6, 5.7 and 5.8. Section 5.9 contains experimental tests comparing strength and performance of the various programs. Finally, we give our conclusions and future research directions in Section 5.10.

5.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an evolution of simpler and older Monte Carlo methods. While the core concept is still the same – a program plays a large number of random simulated games and picks the move that seems

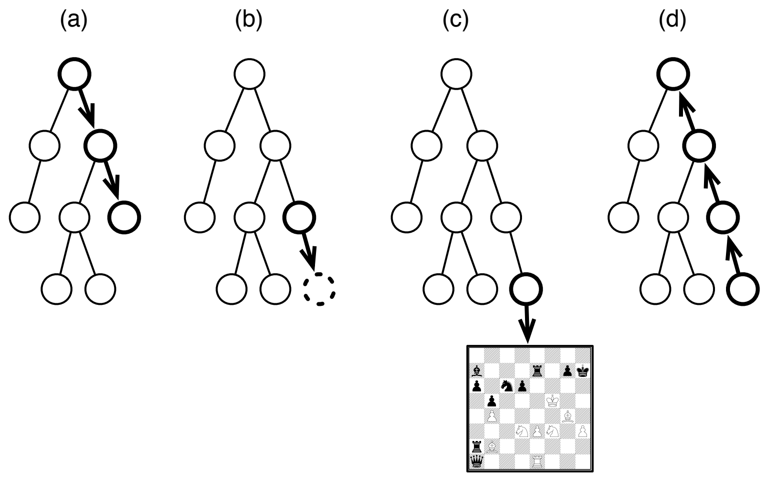


Figure 5.1: The four phases of Monte Carlo Tree Search: selection, expansion, simulation and backpropagation.

to yield the highest victory ratio – the purpose of MCTS is to make the computation converge to stable, reliable values 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 Figure 5.1.

- **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 simply 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 [Coulom, 2007].

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 player needs to strike a balance between exploration (devoting some time to new nodes) and exploitation (directing the simulations towards node that have shown promise so far). Most programs make use of the standard UCT algorithm (Upper Confidence bound applied to Trees) first given in [Kocsis and Szepesvari, 2006]. 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 size and branching factor. In general, most programs will expand a node after it has been visited a certain 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.

5.2.1 MCTS and imperfect information: Phantom Go

Monte Carlo Tree Search has been used successfully in large, complex imperfect information games, most notably Phantom Go. This game is the imperfect 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 which case he can try another move instead. [Cazenave, 2005] describes a MCTS algorithm for playing the game, obtaining a good playing strength on a 9x9 board, and a thorough comparison of several Monte Carlo approaches to Phantom Go, with or without tree search, has recently been given in [Borsboom et al., 2007]. We are especially interested in Phantom Go because its problem space and branching factor are much larger than most other (already complex) imperfect information games such as poker, for which good Monte Carlo strategies exist; see, for example, [Billings et al., 2002].

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 problem 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.

5.3 Kriegspiel vs. Phantom Go

On a superficial level, Kriegspiel and Phantom Go are quite similar. Both maintain the identical rules of their perfect information versions, only adding a layer of uncertainty in the form of 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 several differences worth mentioning between the two games.

- The nature of Kriegspiel uncertainty is completely 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 umpire 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 umpire'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.
- Finally, there are differences carried over from their perfect 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.

5.4 Monte Carlo Kriegspiel

Computer programs capable of playing a full Kriegspiel game have only emerged in recent years due to the complexity of the domain. The first Monte Carlo approach to Kriegspiel is due to [Parker et al., 2005]. This program plays by using and maintaining a state pool that is sampled and evaluated with a chess function. 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 imperfect 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 umpire's messages, analyze them with well-known perfect information algorithms and evaluation functions, such as the popular and open source GNUChess engine, choosing the move that obtains the highest average score in each sample. The choice of using a chess function 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 of poker. Not only is the problem 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 chess move is computationally more expensive than a position in bridge, and a full minimax has to be performed 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 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.

The authors have 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 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). so 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.

More recently, there have been separate attempts at modeling the opponent in Kriegspiel with Markov decision processes in the limited case of a 4x4 chessboard in [Del Giudice et al., 2009], which then evolved into a full Monte Carlo approach with particle filtering techniques in [Bryan et al., 2009]. The

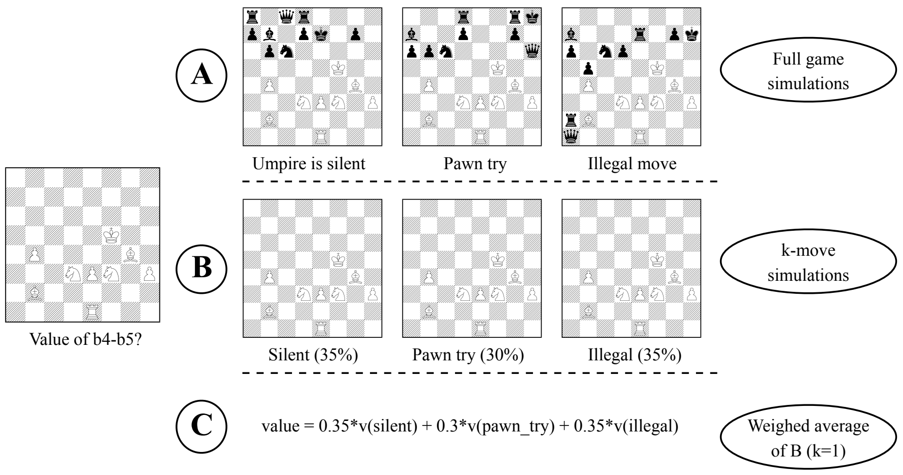


Figure 5.2: Comparison of three simulation methods. Approach A is standard Monte Carlo tree search, approach B simulates umpire messages only and for k-move runs, approach C immediately computes the value of a node in approach B for $k = 1$.

latter work has some similarities, at least in spirit, with the modeling techniques presented in this paper, though it is still similar to [Parker et al., 2005] in that it generates plausible Kriegspiel states which are evaluated by a chess engine.

5.5 Three approaches

In this chapter, we provide three Monte Carlo Tree Search methods for playing Kriegspiel, which we label A, B and C. These approaches are quickly summarized in Figure 5.2 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 further extremization

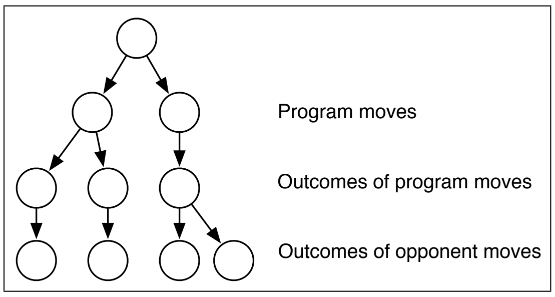


Figure 5.3: Three-tiered game tree representation in our algorithms.

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 Figure 5.3. 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.

Initially, we investigated an approach that was as close as possible to the Monte Carlo techniques developed for Go and its partial information variant, taking into account the important differences between these games and Kriegspiel.; the first version of our program, approach A, was a more or less verbatim translation of established Monte Carlo tree search for Go. We developed the other two methods after performing severely unsuccessful tests – in which approach A could not be distinguished from the random player.

The three approaches all use profiling data taken from a database of about 12,000 human games played on the Internet Chess Club. Because information is scarce, opponent modeling is an important component of a Kriegspiel player. Our programs make use of information from game databases in order to build an opponent's model, either for a specific opponent or for an unknown adversary that is considered to be an averaged version of all the players in the database. We will therefore suppose that we have access to two 8x8 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, respec-

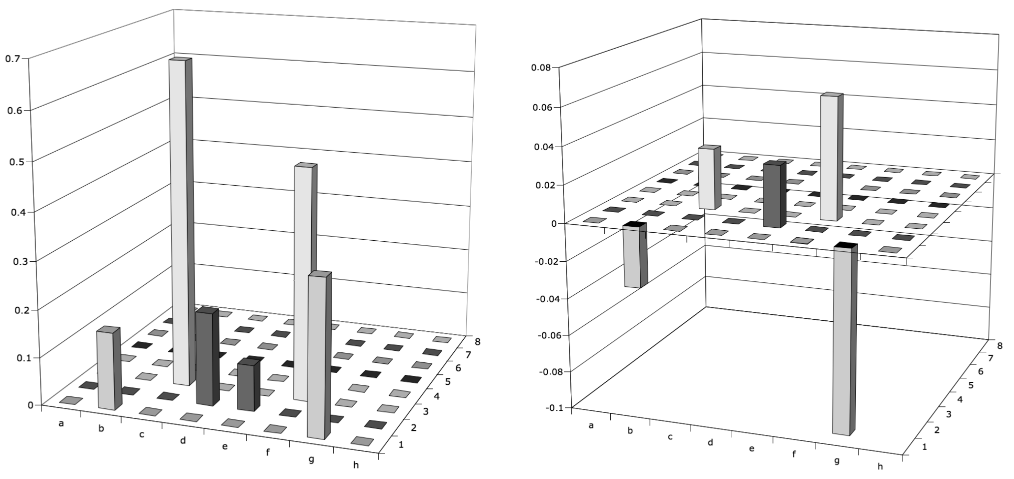


Figure 5.4: Database data for handle "paoloc" playing as White, $t = 10$, $p =$knight, both as absolute probabilities and delta values from move 9.

tively. 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 Figure 5.4. Ten moves (twenty plies) into the game, the locations of this player's knights can be inferred with high probability. This is no coincidence, as 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 Monte Carlo 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.

5.6 Approach A

Pseudocode for approach A is shown in Figure 5.5. Our approach A implements the four steps of Monte Carlo tree search as follows.

Selection is implemented with UCT for the program's own moves, as seen

```
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);
}
```

Figure 5.5: Pseudocode for approach A.

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 seem to have any impact on performance. It is seen in Borsboom et al. [2007] 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 and collected from a database of online games to estimate density for each piece type at any given point in time. The matrices, being 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 how many pawn 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 Parker et al. [2005] when applied to the same task.

For Expansion, our implementation expands a new node chosen randomly with each iteration. We considered this random choice to be a reasonable solution; the alternative would be to have a heuristic, that for a game like Kriegspiel is very difficult to define. 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.

We implemented standard Backpropagation, using the average node value as backup operator.

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 with such high uncertainty - a game in which knowledge barely survives the next move or two. We can, however, compare this task to other games of partial information in which Monte Carlo (not necessarily MCTS) has been successful. In bridge, possibly the most similar domain among the card games, simulation can take on the forms of single- and double-dummy play, such as in GIB Ginsberg [1999]. This kind of simulation is easily carried out, and generating realistic deals, for example with the help of bayesian inferences, is actually the more challenging part. While bridge is half about the cards and half about the betting, poker (in the Texas Hold'em variant that has grown to be the most popular) is more about the betting; opponent modeling is usually tightly integrated with Monte Carlo simulation, as in Billings et al. [2002]. As in bridge, a large part of the task is performing selective sampling, that is, skewing the probabilities for each simulated deal according to the opponent's decisions and known history. These card games have the advantage that the player only has a limited amount of choices (poker moreso than bridge). In Kriegspiel, our ability to generate realistic 'hands' is much more limited than in either game, except near the start and the end of the game;

nor can we acquire much knowledge from playing 'dummy'. Even among board games, 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 high optimization in chess programs.

In approach A, both 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. Trying to achieve direct checkmate with random games immediately appeared to be almost hopeless. In order to make simulation more accurate, 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.

As mentioned, approach A failed, performing little better than the random player and losing badly and on every time setting to a more traditional player 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 were not realistic. The generation algorithm used probability distributions for pieces, pawns and king that were updated after each umpire message. While the probabilities were quite accurate, this did 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 tended to underestimate the protection level of the opponent's pieces. Secondly, because moves were chosen randomly, it also underestimated the opponent's ability to coordinate an attack and hardly paid attention to its own defense.

Lastly, but perhaps most importantly, there is the subtler issue of *progress*. Games where Monte Carlo approaches have been tested most thoroughly have a built-in notion of progress. In Go, adding a stone changes the board per-

```
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);
}
```

Figure 5.6: Pseudocode for approach B.

manently. The same happens in Scrabble. A game of poker consists of just a few betting rounds that require no notion of progress. Kriegspiel, on the other hand, like real-time strategy games 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 dangerous plan or just choose to minimize the risk by moving the same piece back and forth. When a Monte Carlo method does not perform enough simulations to find a stable maximum, it can do either.

In view of these results, we claim that it is unlikely for a mere transposition 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, the game would have to be considerably simplified, or the players would need to receive more information on the state of the board.

5.7 Approach B

A more serious Monte Carlo tree search Kriegspiel program needs to converge much faster than the naive 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 - hence, a possible solution could lay in running simulations but not on individual game states – rather, on their *perception* from the 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 perfect information, they would be run as Kriegspiel games with imperfect 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; for example, [Chung et al., 2005] 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 simulate 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, with pseudocode listed in Figure 5.6. This approach removes the randomness involved in generating single states and instead only simulates referee messages, without worrying about the enemy layout that generated them. A reduced version of the abstract model used in approach A estimates the likelyhood 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 8x8 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 $Pc_{i_2j_2}$ and $Pw_{i_2j_2}$, 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 we deem reasonable for a Kriegspiel opponent, in view of human play observed on the Internet Chess Club:

- When the program captures something, there is a 99% 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.
- There is a chance, assumed to be constant in our model, of the player's piece being captured when a check message is heard. 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 of the player suffering 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,

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

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

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

where f is a suitable function that scales and centers the probability delta values gathered from the game database discussed in Section 5.5, 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.

- 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 for a method relying on speed and number of simulations. Instead, the board is scanned along several directions, as shown in Figure 5.7. 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.

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

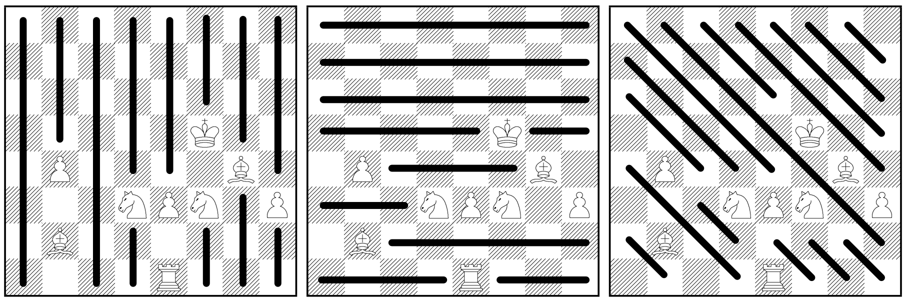


Figure 5.7: Density spreading routine in approaches B and C (second diagonal sweep not shown).

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, as well as 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 to improve the program's 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, the opponent's perceived strategy follows some average, realistic expectations while his actual moves are never disclosed or even generated.

5.7.1 Partial simulations

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.

```
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);
}
```

Figure 5.8: Pseudocode for approach C.

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.

5.8 Approach C

The third and final approach, called C and shown in Figure 5.8, 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. Also, there is a common tendency seen in Kriegspiel literature, first in Parker et al. [2005] and then in Bryan et al. [2009], 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, as outlined in the previous section, that the benefits of global exploration by simulating longer games are quite limited compared to the loss of accuracy in the short run, thus emphasizing selection strategies over simulation strategies. Another way to think of approach C is as if simulations happened entirely on the tree 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.

5.9 Tests

We test our approaches, with the exception of A which is not strong enough to be interesting, against an improved version (about 100 Elo points stronger) of the one described in the previous chapter (Darkboard 1.0). As we have seen, the main feature of that Kriegspiel player is the use of *metapositions* as representations of the game's belief state that can be evaluated in a minimax-like fashion. Tests against humans on the Internet Chess Club showed that the minimax program's playing strength is reasonable by human standards, ranking above average at around 1700 Elo points; specifically, it possesses good defense but is often unable to find a good attack strategy unless the opponent is in a weaker position, which limits its strength as Kriegspiel

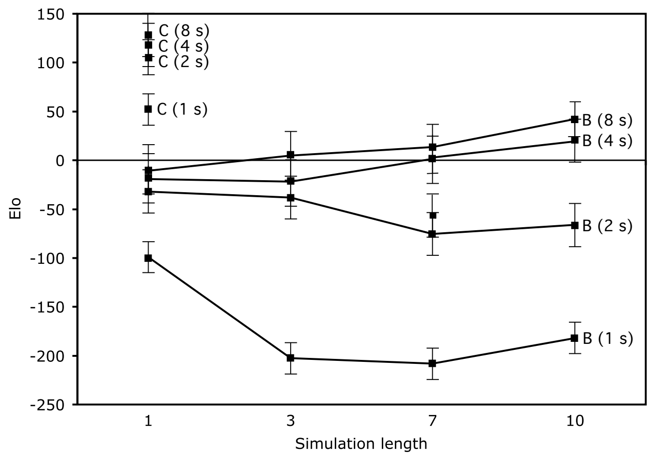


Figure 5.9: Comparison of MCTS approaches B and C with a fixed-depth minimax program at different time settings and simulation depths, with error intervals.

favors a wise attacker. The program used as an opponent for Monte Carlo Tree Search in our tests is probably slightly stronger than the aforementioned one, since it performs a series of hard-coded checks that prevent the agent from making obvious blunders. It should be noted that our Monte Carlo players do not include these checks. The evaluation function of the minimax player is rather complex, consisting of several components including material, positional and information bonuses. By contrast, our Monte Carlo programs know very little about Kriegspiel: approaches 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.

The results of the tests are summarized in Figure 5.9. Programs are evaluated by tournaments against the minimax program, with the value of k on the x axis and the difference in Elo points on the y axis. Thus, programs that perform worse than the minimax program 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 minimax program itself always runs at the same, optimal setting for its evaluation function, requiring between

1 and 2 seconds to run. The program hits a performance plateau after this point, preventing it from further increases in performance.

After witnessing the failure of approach A, we limit our tests to approaches B and C. These 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. In fact, approach C lacks randomness altogether and could be considered a degeneration of a Monte Carlo technique.

Experimental findings more or less confirm our expectation, that is, lower values of k should be more effective under faster time settings, and 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 likelyhood.

Given the simplicity of the approach and the lack of specialized knowledge compared to the minimax program's trained parameters and pruning

	Darkboard 1.0	Darkboard 2.0
Games played	2442	7121
Unique opponents	384 (6.36 games each)	589 (12.09 each)
Avg. Opponent Elo	1534	1646
Avg. Score	0.512	0.470
Avg. Elo	1543	1626
% of games vs higher Elo	47.0%	60.3%
Games vs Top 20 players	792 (32%)	2777 (39%)
Avg. Score vs Top 20 players	0.171	0.26

Table 5.1: Comparison of Darkboard 1.0 and 2.0 (approach C) on the Internet Chess Club.

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.

We collected more experimental evidence that approach C is effective letting it to play against humans over the ICC. The comparison with the heuristic search program, Darkboard 1.0, is shown in Table 5.1. It should be noted that this scoring system is conservative by about 70-90 Elo for all players due to ICC mechanics. Moreover, the top 20 players considered were the same for both programs, so the confrontation is direct.

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 Olympiads in Pamplona, Spain, in May 2009.

5.10 Conclusions and future work

There are several conclusions to be drawn from these 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, precautions need to be taken so that the program does not end up sampling data that is too noisy to be useful; in this case, such a requirement is met by abstracting the game with a model in which single states are not important, and only their perception matters.

Secondly, we can explain the success of approach C, which is basically UCT with a node evaluation function, with its accuracy in simulating the very next move. Also, it can be argued that variable-depth UCT selection can outperform fixed-depth minimax in high-uncertainty situations even under these unorthodox premises. Still, approach B - the more traditional Monte Carlo method - seems to have the largest room for improvement. While experimental data indicates that stable evaluation of the first move is the largest factor towards improving performance and avoiding tactical blunders when time is short, a longer simulation with higher values of k provides better strategic returns under higher time settings. In particular, $k = 10$ shows great improvement at 4 seconds per move. It is possible that, with more effective simulation strategies and more processing power, approach B will be able to outperform approach C. It is too early to reach a conclusion on this point. A hybrid approach, treating the first move like C and the following moves like B, is also worth investigating.

The program as a whole can still be improved by a large factor. 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 Chaslot et al. [2008]. 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.

Chapter 6

The quest for progress in the endgame

This chapter is the first of three devoted to the Kriegspiel endgame, which offers considerably different challenges with respect to the midgame. The results contained in this chapter, largely due to Bolognesi and Ciancarini, serve to illustrate the state of the art in the endgame as far as online search algorithms are concerned; that is, search algorithms that operate at runtime to find the solution to a Kriegspiel endgame problem. While these heuristic-driven algorithms are not optimal, they can find a good solution to most problems in a reasonable time unless the opponent has access to perfect information on the state of the game. By contrast, the new results contained in the next two chapters solve the same Kriegspiel endgames through retrograde analysis and the construction of a tablebase, achieving perfect play wherever victory can be obtained with probability 1. Even so, this method remains useful because of its simplicity and applicability to situations in which White cannot push victory with probability 1.

In the remainder of this chapter we consider an algorithm that builds and searches a Kriegspiel game tree made of nodes called *metapositions*, and uses an evaluation function in order to judge each node and implement a progress heuristic. In order to evaluate this approach, we deal with some basic endgames of Kriegspiel, i.e. those where a player (we assume Black) has only the king left. Thus, in the next sections we will consider White having a king and a rook (in the KRK ending), a king and a queen (KQK), a king and two bishops (KBBK), a king, a bishop, and a knight (KBNK).

These are the same endgames that will be considered in the next chapters and used as a test bed for retrograde analysis.

This chapter has the following structure. In Section 6.1 we describe a way to represent uncertainty using metapositions and show how it is possible to find an optimal strategy that checkmates the opponent in the shortest amount of moves. In Section 6.2 we show how to reduce the game tree in order to simplify the search with the help of an evaluation function and a pruning strategy.

In Section 6.3 we propose such an evaluation function for "progressing in the dark" in the case of the main and most common Kriegspiel endings. Finally, in Section 6.4 we evaluate this approach, especially comparing it with a previous algorithm proposed by [Boyce, 1981].

6.1 Metapositions in the endgame

In this section we re-introduce the concept of *metaposition*, which was first encountered in chapter 4 and will be further expanded and formalized in the next chapter. As previously seen, A *metaposition* is a position able to denote a set of normal Chess positions. For example, in the leftmost diagram depicted in Figure 6.1, White is not sure where the Black King could be: multiple Black Kings represent equally possible positions of the King. Metapositions are useful because a Kriegspiel position can be described by a pair of metapositions, one for each player; each metaposition represents the knowledge inferred by a player.

We will also use the term *reference board*, which is a chessboard diagram annotated with all the possible positions of the opponent's pieces (usually the King alone). Thus White's reference board represents the (uncertain) knowledge about the position of the black king.

A *metamove* is the aggregate of all possible moves Black can make from any state contained in the metaposition. Information about a metamove can be inferred from the referee's messages. In other words, after a Black metamove White should update his reference board taking into account all the new possibilities that are compatible with the latest message.

We will also use the term *pseudomoves* to indicate moves tried by White on a metaposition.

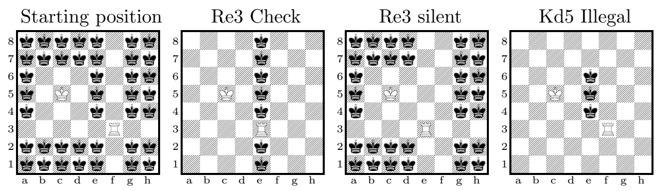


Figure 6.1: Three metapositions resulting after three different referee's answers. to two different White pseudomoves.

We can easily show that White's knowledge increases after a pseudomove and decreases after a metamove. The leftmost diagram in figure 6.1 shows an initial position where White can try, for example, the pseudomoves Re3 and Kd5. The initial metaposition has the greatest uncertainty, as each square not controlled by White can contain a Black King.

If White tries to move his rook to e3 and the referee answers *"Check"*, he will update his reference board and assume that the Black King's location is on the same rank or file occupied by the White Rook, as shown in the second diagram of figure 6.1. If White tries Re3, but the referee remains silent, he will update his reference board clearing the squares around the King and along the Rook's rank and file. This is depicted on the third diagram of figure 6.1. Finally, an attempt may be illegal because White tries to move his King to a square controlled by the Black King. In this case, White will realize that the Black King must be somewhere around e5. The rightmost miniature in figure 6.1 shows White's reference board updated after he tried Kd5 and received the *'illegal'* warning from the referee.

6.1.1 Number of metapositions

In order to highlight the numerical complexity of dealing with uncertainty by means of metapositions, we calculate the number of metapositions for the Rook ending. It takes about 28,000 positions to solve the same ending exhaustively in Chess, as shown in Clarke [1977]. In Kriegspiel, we can simply iterate over the positions of the White pieces and count the number of distinct combinations of Black Kings. If we assume, as a worst case for White, the Rook on a1 and the King on b1, there are 52 possible squares not controlled by White and the total number of metapositions is

$$\sum_{1\leq n\leq 52} \binom{52}{n} = 2^{52} - 1. \tag{6.1}$$

Thus, the number of metapositions is extremely large. Reflecting a metaposition along the axes and main diagonal does not decrease the numerical magnitude of the problem. In the next chapter we show how only a small subset of these metapositions is really significant, however we are clearly unable to evaluate each single metaposition separately.

6.1.2 Optimal search

It follows from the previous section that a Kriegspiel ending played with metapositions can be viewed as a perfect information game. While we have not removed imperfect information, we have changed our representation to account for it, and we can apply the minimax theorem to this scenario. MAX's moves are White's pseudomoves; MIN's moves are the referee's messages. Suppose Black can choose the referee's answer to White's move among those compatible with the current metaposition: the new reference board contains uncertainty, but its computation is entirely deterministic. Even illegal moves fit the model nicely, as an illegal move is still a referee's message and only differs from the others in that the White pieces stand still.

Such a model assumes that Black is not only omniscient, but able to relocate his King at will and as the need arises (Magari was the first to liken the Black King's behavior to quantum mechanics). Under this model, we can build a normal minimax tree. Zermelo's theorem guarantees there are optimal strategies for either player to achieve a minimum payoff.

Clearly, it is pointless to try this opponent model in a full game of Kriegspiel, as White is simply hopeless against a 16-piece board-altering oracle. In the endgame, however, it makes perfect sense to do so, in view of the fact that we already know White can win even against such a gifted opponent. We can therefore use minimax to find the *optimal* mate, that is, the shortest forced mate. All the program needs is the ability to recognize game-ending metapositions and assign their payoffs. The definitions are a simple consequence of Black's powers:

- A metaposition is a checkmate is every Black King on the reference board is checkmated.
- A metaposition is a stalemate if at least one Black King is stalemated.
- A metaposition is an insufficient material draw if the referee can say "capture".

The pseudocode for this algorithm is in Figure 6.2. It is readily seen that this is a very simple minimax.

Figure 6.3 shows an example of optimal search. Without any evaluation besides verifying these final states, searching the game tree leads the program to the best move, Ra1. No other strategy can checkmate Black in two moves.

Just like with normal minimax, problems arise when the final state is distant and deep into the game tree. In this case, the number of positions we need to visit is greater than any feasible search depth. The solution we investigate in this paper is, once again, adapted from Chess minimax: we define evaluation functions for these endgames. The goal is to maximize a function measuring our progress towards checkmate; we are prepared to lose optimality in exchange for being able to play well under any circumstances.

6.2 Game tree reduction

Just like in the perfect information case, it is not feasible to explore a full metaposition tree. It is known from practical findings that a general KRK instance can last 30 or more moves, depending on the opponent's strategy. Seeing as White has, on average, about 15 pseudomoves to choose from and it is not possible to trivially prune the tree except when a move can lead to an instant draw, some level of approximation is necessary even with three-piece endings. We also wish to play positions that are not certain wins, for example because White's Rook is threatened after the first move, with a good strategy maximizing our chance to win.

Our approach is as follows. First, we create an evaluation function (EVAL) that outputs a score for a given metaposition. Then, we also define a pruning strategy for the metaposition tree. The resulting algorithm is still a minimax, with MAX's moves (White) being pseudomoves (moves which are

```
function evalOpt(Metaposition mp, int depth)
{
    if (isCheckmate(mp)) return 1.0;
    if (isStalemate(mp)) return -1.0;
    if (insufficientMaterial(mp)) return -1.0;
    if (fiftyMoveDraw(mp)) return -1.0;
    if (depth==1) return  0.0;
    double max = -INFINITY;
    Move best_move = NULL;
    foreach (mv in getPseudomoves(mp)) do
    {
        double min = INFINITY;
        foreach (msg in getPossibleMessages(mp,mv)) do
        {
            double val = evalOpt(updateMp(mp,mv,msg),depth-1);
            if (val < min)
                min = val;
        }
        if (min > max)
        {
            max = min;
            best_move = mv;
        }
    }
    return max;
}
function optimalIterativeDeepening(Metaposition mp)
{
    for (int k=1; true; k++)
    {
        double val = evalOpt(mp,k);
        if (val==1.0 || val==-1.0) return val;
    }
}
```

Figure 6.2: Pseudocode for full Kriegspiel minimax.

legal for at least one possible configuration of the Black pieces), and MIN's moves being metamoves corresponding to a referee's message, which can be *silent* (S), *check* (C) or *illegal* (I).

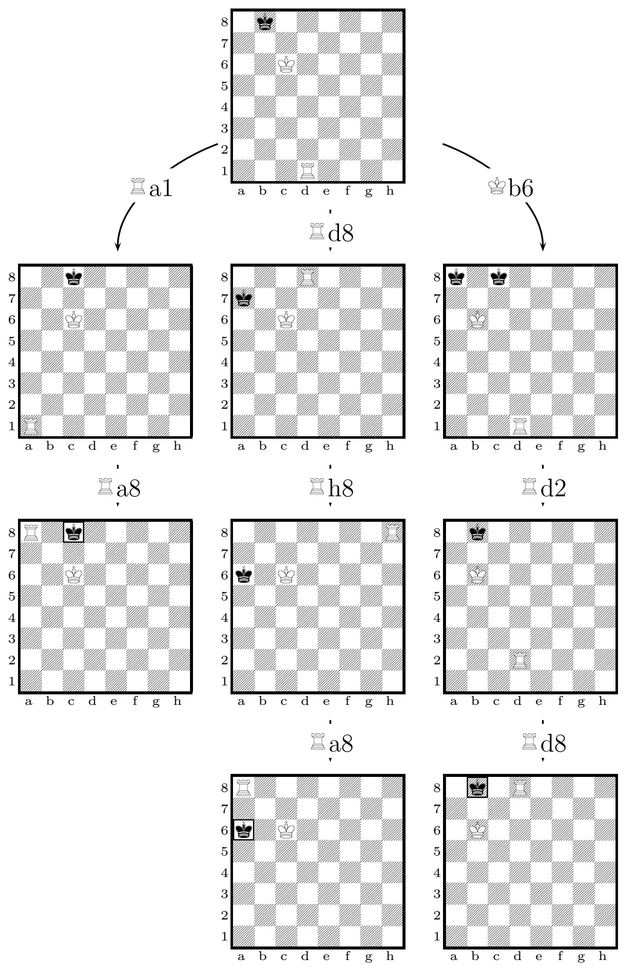


Figure 6.3: Example of optimal search.

The algorithm therefore proceeds as follows. Suppose that White's reference board is the one depicted in Figure 6.5 and that it is White's turn to move. The search algorithm generates all the pseudomoves and creates three new metapositions according to the three (or fewer) possible answers from the referee. Then, it chooses the one with the lowest evaluation. In the example we have 21 pseudomoves. All 21 are compatible with a silent referee, 2 King moves can be illegal and 3 Rook moves can cause a check. In this example we only prune 5 nodes, though we will prune many more when there are many Black Kings on the board.

If the search algorithm has reached the desired search depth it simply re-

```
function evaluate(Metaposition mp, int depth)
{
    if (depth == 0) return EVAL(mp);
    double max = -INFINITY;
    Move best_move = NULL;
    foreach (mv in getPseudomoves(mp)) do
    {
        double min = INFINITY;
        Message worst_message = NULL;
        foreach (message in getPossibleMessages(mp,mv)) do
        {
            Metaposition update = updateMp(mp,mv,message);
            if (EVAL(update) < min)
            {
                min = EVAL(update);
                worst_message = message;
            }
        }
        Metaposition selected = updateMp(mp,mv,worst_message);
        double childEval =evaluate(selected,depth-1);
        if (childEval > max)
        {
            max = childEval;
            best_move = mv;
        }
    }
    return max + EVAL(mp);
}
```

Figure 6.4: Pseudocode for our approximated metaposition search algorithm.

turns the best node, that is the the node with the maximum value, otherwise it applies Black's metamove on each node, decrements the search depth and recursively calls itself on a subtree. Finally, it retracts the pseudomove played and adds to the metaposition's value the score returned by the recursive call, updating the max on that particular search depth.

When the algorithm is done visiting the tree, it returns the best pseudomove to play. As we mention later when dealing with omniscient opponents, we have implemented a loop-breaking mechanism that avoids playing the same move from the same position twice.

Pseudocode for our algorithm is given in Figure 6.4. The pruning heuristic is quite radical, in that we only consider the referee's message with the lowest evaluated score according to EVAL. While this reduces the search space by a large factor, it also implies a high degree of trust in the evaluation function, as we assume that the worst message (basically, the one leading to the longest forced mate) is always the one returned by EVAL.

One reason for choosing this strategy is that there is a very high correlation between the distance to mate and the amount of Black Kings near the middle of the board. It is a notion we can simply derive from endgame databases in Chess: it takes much longer to checkmate the King when it is in the middle. There is a very high probability that the metaposition with the most central Kings will be the longest to solve, and the evaluation function takes that into account. This makes us confident in EVAL whenever this holds true in Chess. The second reason is more practical: metapositions are much slower to work with than Chess positions, and updating Black's metamoves is especially slow. Therefore, we can benefit from aggressive pruning much more than a Chess function would.

6.3 The evaluation function

The evaluation function tries to capture the notion of progress leading a player towards victory. It is a linear, weighted sum of features expressed as

$$\mathrm{EVAL}(m) = w_1 f_1(m) + w_2 f_2(m) + ... + w_n f_n(m) \qquad (6.2)$$

where, for a given metaposition m, w_n indicates the weight assigned to a particular subfunction f_n. For example, a weight might be $w_1 = -1$ and $f_1(s)$ may indicate the number of Black Kings.

The EVAL function is different for each ending, but it has some invariant properties: it avoids playing moves that could lead to a stalemate and it immediately returns a checkmating move, if one exists.

In the following subsections we briefly describe the evaluation functions we use for some Kriegspiel endings.

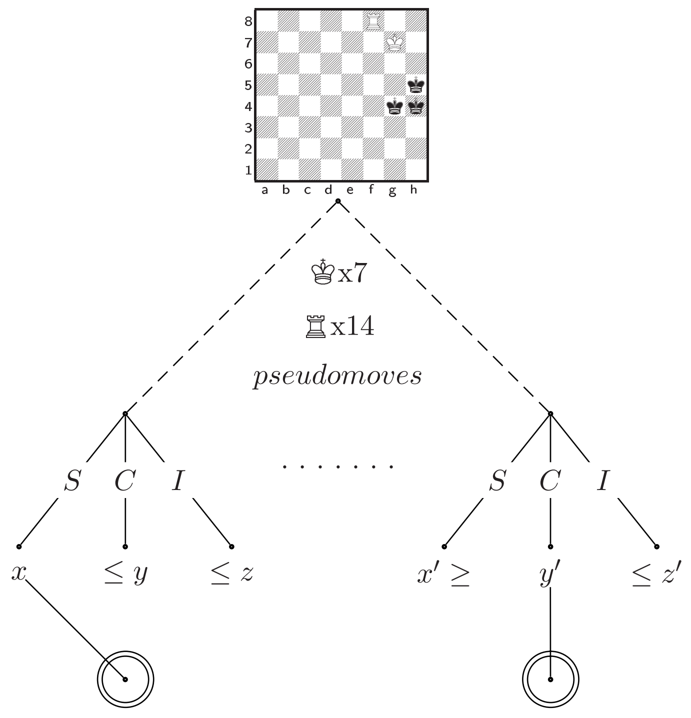


Figure 6.5: Pruning a metaposition tree.

6.3.1 The Rook ending (KRK)

The evaluation function for this ending considers $n = 6$ different features.

1. it keeps the Rook safe: $w_1 = -1000$ and f_1 is a boolean function which is true if the White Rook is under attack;

2. it brings the two Kings closer: $w_2 = -1$ and f_2 returns the distance (number of squares) between the two Kings;

3. it reduces the number of quadrants (as seen from the Rook) with Black Kings in them, as well as the total number of Kings: $w_3 = -1$ and $f_3 = c\sum_{i=1}^{4} q_i$, where $c \in \{1, 2, 3, 4\}$ is the number of quadrants with Black Kings and q_i is the number of Black Kings in i^{th} quadrant;

4. it keeps the Black King from interposing between the White Rook and the White King: $w_5 = -500$ and f_5 returns 1 if the Black King is inside the rectangle formed by the White pieces, or 0 otherwise;

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -2 & -4 & -4 & -2 & 0 & 0 \\ 0 & 0 & -4 & -4 & -4 & -4 & 0 & 0 \\ 0 & 0 & -4 & -4 & -4 & -4 & 0 & 0 \\ 0 & 0 & -2 & -4 & -4 & -2 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Figure 6.6: The numerical matrix $v[]$

5. it keeps the White pieces close to each other: $w_5 = +1$ and f_5 returns 1 if the Rook is adjacent to the King, 0 otherwise;

6. it pushes the Black King towards the corner of the board: $w_6 = +1$ and $f_6 = \sum_{i=0}^{64} K(i) \cdot v[i]$, where v is a numerical 64-element vector, shown in Figure 6.6, that contains higher values for corner and edge squares, and $K(i)$ returns 1 if there is a Black King on square i, 0 otherwise.

6.3.2 The Queen ending (KQK)

The evaluation function is similar to the one described in Section 6.3.1, except we consider the Queen instead of the Rook and we add two features. Thus, $n = 8$ and in the first six are the same as in KRK, with the addition of:

7. it penalizes a large number of Black Kings: $w_7 = -1$ and f_7 is equal to the number of Black Kings on White's reference board;

8. it performs the same checks as feature 3 in KRK, but with the Queen's diagonals used in place of the quadrants: $c \in \{1, 2, 3, 4\}$ is the number of areas with Black Kings in them, and a_i is the number of Kings in the i^{th} area.

6.3.3 The ending with two Bishops (KBBK)

In this ending we have to deal with two White pieces other than the King. The evaluation function uses the same subfunctions considered in the previous endings, but with different weights. In the subfunctions that only mention one Bishop, the values are calculated and summed separately for the two Bishops.

$$\begin{pmatrix}
0 & -10 & -50 & -100 & -100 & -50 & -10 & 0 \\
-10 & -10 & -40 & -40 & -40 & -40 & -10 & -10 \\
-50 & -40 & -40 & -40 & -40 & -40 & -40 & -50 \\
-100 & -40 & -40 & -50 & -50 & -40 & -40 & -100 \\
-100 & -40 & -40 & -50 & -50 & -40 & -40 & -100 \\
-50 & -40 & -40 & -40 & -40 & -40 & -40 & -50 \\
-10 & -10 & -40 & -40 & -40 & -40 & -10 & -10 \\
0 & -10 & -50 & -100 & -100 & -50 & -10 & 0
\end{pmatrix}$$

Figure 6.7: The numerical matrix $b[]$

1. it keeps the Bishops safe: $w_1 = -1000$ and f_1 returns 1 if at least one of the Bishops is under attack;

2. it brings the two Kings closer: $w_2 = -1$ and f_2 returns the distance (number of squares) between the two Kings;

3. it keeps the Black King from interposing between the White rook and the White Bishops: $w_3 = -500$ and f_3 returns 1 if a Black King is in the rectangle formed by the White King with at least one Bishop;

4. it keeps the White Bishops closer: $w_4 = +2$ and f_4 returns 1 if the Bishops are adjacent to each other, 0 otherwise;

5. it pushes the Black King toward a corner of the board: $w_5 = +1$ and $f_5 = \sum_{i=0}^{63} K(i) \cdot b[i]$, where b is a numerical 64-element vector, shown in figure 6.7, and $K(i)$ returns 1 if there is a Black King on square i.

6. it keeps the White King on the Bishop's rank or file: $w_6 = +1$and f_6 returns 1 if the King and the Bishop are on the same rank or file;

7. it reduces the number of Black Kings on the areas traced by the Bishop's diagonals: $f_7 = c\sum_{i=1}^{4} a_i$ where $c \in \{1, 2, 3, 4\}$ is the number of areas that contain at least one Black King, a_i is the number of possible Black kings on i^{th} area, and

   if $f_5(m) \leq -600$ (very low information on the Black King) then $w_7 = -4$;

   else $w_7 = \frac{1}{6}$

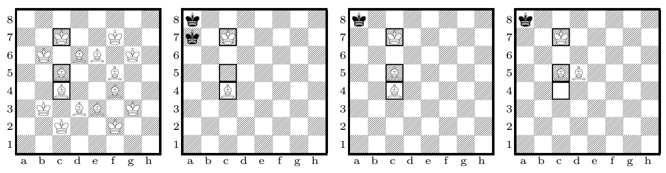


Figure 6.8: Key Bishop positions.

8. it prefers some particular positions (we will refer to them with the term *key Bishop positions*) for the White King and Bishops, highlighted in figure 6.8; for example Kc7, Bc4 and Bc5. Therefore, $w_8 = +30$ and f_8 returns 1 if the Bishops and the King are arranged in one of the key positions.

6.3.4 The ending with Bishop and Knight (KBNK)

The evaluation function for the White Bishop is the same as in Section 6.3.3. For the Knight we cannot consider any division of the board, so the evaluation for this chessman consists of reducing the number of Black Kings on White's reference board and of supporting the Bishop.

We also used a large set of key metapositions similar to those for the Bishops ending shown in figure 6.8. Unfortunately, this was not enough to obtain a good evaluation function for the KBNK endgame, as we will explain in the next section.

6.4 Tests and comparisons

In order to evaluate our algorithm, we have written a rule-based program that implements the procedure proposed in Boyce [1981] for the Rook ending. Boyce showed a way to force checkmate by considering positions where both Kings are in the same quadrant of the board as seen from the Rook, or where the Black King is restricted to one or two quadrants of the board. Thus, our rule-based program uses a search algorithm and a small evaluation function to obtain an initial position (with White's pieces near a corner) from which it can start following the directives in Boyce [1981]. This component brings White to a valid starting position as quickly as possible, and if a position is encountered that Boyce can solve, the algorithm immediately switches to

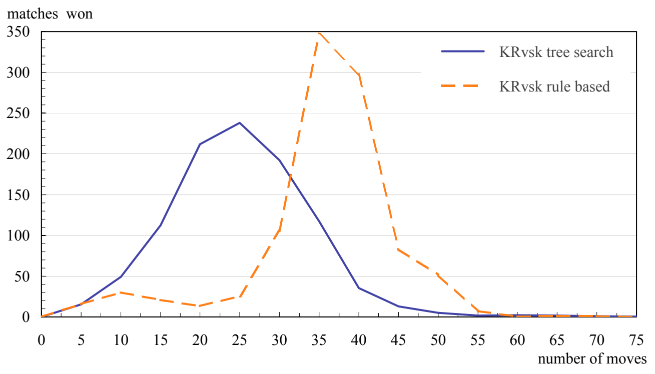


Figure 6.9: Rook endgame: Comparing our function with Boyce's algorithm.

Boyce; this usually happens as a response to an illegal move. The Black King is not omniscient in this test. Instead, it does what any rational Kriegspiel player would do: it tries to move towards the center. The algorithm and will always pick its moves in decreasing order of distance from the nearest corner.

6.4.1 Rook endgame: comparing our function with Boyce's algorithm

We played about 28,000 matches, one for each possible starting position in KRK. White always starts out with the highest amount of uncertainty: every square not controlled by White contains a Black King. The results are summarized in Figure 6.9. The number of matches won (normalized to 1000) is on the ordinate and the number of moves needed to win each match is on the abscissa.

The rule based program spends the first 25 moves looking for one of the initial positions; when it reaches one of these positions the checkmate procedure is used and the program wins very quickly. However, the average of moves needed is around 35. Our tree-searching program is faster, winning after 25 moves on average.

The main advantage is that the program tries to make progress towards checkmate from the very first move. On the other hand, the rule-based program is faster than the tree-searching program in deciding the move to play. The rule-based program has a constant execution time, whereas the second one is exponential in the search depth.

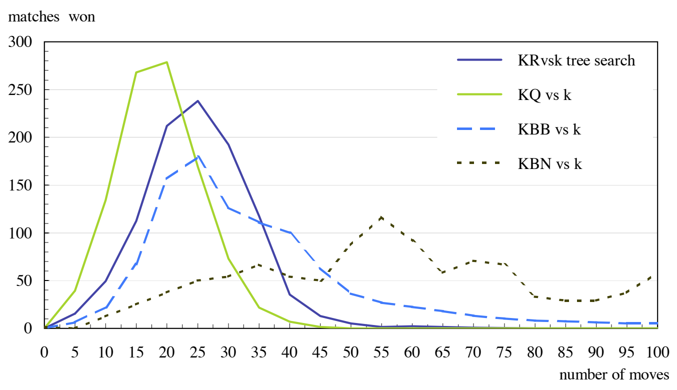


Figure 6.10: Comparison of basic endings.

6.4.2 Evaluating the search algorithm

Figure 6.10 compares game durations for the four tested endings. We simply chose random starting positions for KRK, KQK, KBBK, and KBNK; we gave White the highest uncertainty as in the previous test, and measured the length of each game.

We see that the program wins the KQK ending faster than KRK. This result was expected, because the Queen is more powerful than the Rook: the Queen controls more space so metapositions have a lesser degree of uncertainty.

On the other hand, KBBK is more difficult than KRK. In fact, the Bishop ending almost always requires many more moves: sometimes our program needs more than 100 moves.

Finally, we see that the behavior of our program in the KBNK ending is quite bad. The program often requires more than 100 moves to win and the distribution of victories does not converge: we conclude that our program is not really able to make progress in this ending.

6.4.3 Progress through Uncertainty

An effective way to analyze the progress toward victory consists of considering how the value of White's reference board changes after playing each pseudomove.

Figure 6.11 shows the trend of evaluations assigned to each reference

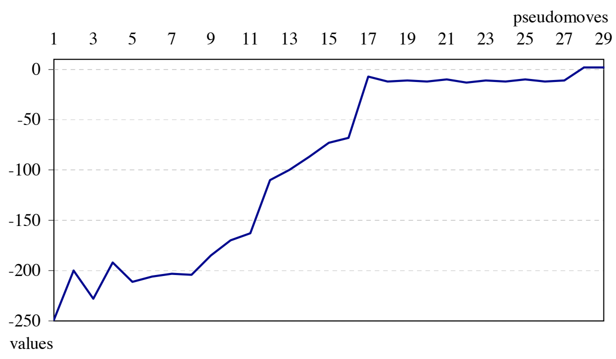


Figure 6.11: Trend of evaluations assigned to metapositions crossed. during the KQK ending.

board reached during a whole match for the KQK ending. The number of moves into the game is shown on the abscissa, while the average scores assigned by the evaluation function are on the ordinate.

We see that, at each step, the value of metapositions increases. From White's point of view, this means that the state of the game is improving and this is actually a good approximation of the real situation.

We have performed the same test for the KBNK ending; the result is shown in Figure 6.12. Here the progress is not controlled by White. In fact, the state of the game does not improve at each step. The graph shows how the evaluations of the reference board change during a match won by White: the value of metapositions does not increase at each pseudomove, but at some unplanned stroke of luck for White. Thus, the program basically wins by chance, that is by exploiting lucky metapositions or its opponent's mistakes.

We conclude that our program is able to progress to victory when we deal with pieces able to divide the board into separate areas, which can then be reduced to trap the Black King; whereas, when we use a piece without this power, like the Knight, the behavior of the program is not fully satisfactory.

6.4.4 Tests against humans

Practical findings reveal that it is very important for a Kriegspiel player to know how to play the endgame correctly. As the opponent is not aware of what pieces one has, they may not be as inclined to resign as they would in

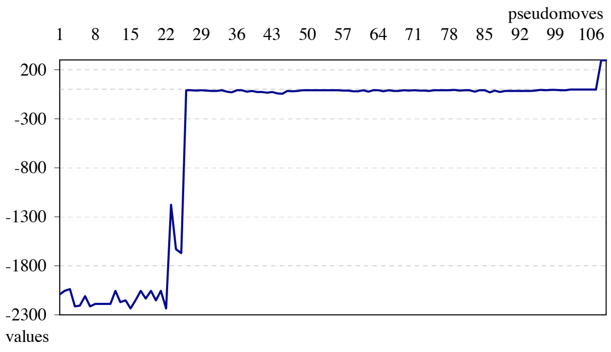


Figure 6.12: Trend of evaluations assigned to metapositions crossed during KBNK.

Chess, even in the presence of overwhelming material. Moreover, all mates are considerably more complex in Kriegspiel than their Chess equivalents and players tend to make many mistakes. We base our claim on a database of about 12,000 Kriegspiel games played on the Internet Chess Club. The most frequent endgames and their outcomes are shown in Table 6.1. As it can be seen, complex endgames such as one with the bishop and knight is in the list of the top 15 drawn endgames, and is almost never won in practice. Even situations that should always be won, such as KQK and KRK, turn out to be draws with surprising frequency. For this reason, even human play can draw considerable benefit from computer analysis.

We have developed a software player, called Darkboard, that can play a full game of Kriegspiel over the Internet Chess Club. We have played about 6,000 games since 2007, using the algorithms presented in this paper for the four endings KRK, KQK, KBBK and KBNK. The outcomes for these endings are summarized in Table 6.2. As it can be seen, our program still does not win every time, though its performance is much better than the average human's. The draws in KRK are due to the 50-move rule, losing the Rook on the first move and timeouts. The few draws in KQK are due to timeouts. Not much can be said about KBBK and KBNK because of their very low frequency, though the program never managed to checkmate a human player with a Bishop and a Knight. We note that, on average, endings against human players last longer than against the King's algorithm used in the previous tests. Humans put up a better defense, which results

Wins		Mates		Draws	
KQK	347	KQK	184	KK	273
KQQK	180	KQQK	144	KBK	121
KQRK	162	KQRK	118	KQK	121
KRK	138	KRK	54	KNK	103
KQRPK	92	KQRPK	54	KPK	44
KQBK	88	KQQBK	47	KRK	41
KQPK	87	KQBK	45	KQBK	27
KPK	65	KQRBK	41	KQNK	21
KQNK	62	KQQPK	37	KQQK	20
KQRBK	60	KQRNK	37	KQBNK	19
KQQBK	57	KQNK	33	KBKB	16
KRPK	56	KQQRK	30	KNKN	16
KQQPK	49	KQRBPK	29	KQRK	15
KQRNK	49	KQQNK	27	KBNK	12
KQRBPK	43	KQQQK	25	KQBPK	10

Table 6.1: Most frequent endgames in a database of Kriegspiel games.

Ending	Wins	Mates	Draws
KRK	81	32	15
KQK	154	73	3
KBBK	7	2	5
KBNK	7	0	7

Table 6.2: Performance of our program against humans.

in a higher ratio of games over the 50-move limit. A human expert will try to move back and forth between two squares and ask for a draw by threefold repetition on every move. Also, the starting distribution in real games is not uniform, and the Black King will often be in the middle of the board, where mates take the longest time.

Chapter 7

Perfect play with retrograde analysis

Retrograde analysis is a tool for reconstructing a game tree starting from its leaves; with these techniques one can solve specific subsets of a complex game, achieving optimal play in these situations, for example a chess endgame. Position values can then be stored in "tablebases" for instant access, as is the norm in professional chess programs. This chapter and the next show that a very similar approach can be used to solve subsets of certain imperfect information games such as Kriegspiel endgames. Using a brute force retrograde analysis algorithm, a suitable data representation and a special lookup algorithm, one can achieve perfect play, with perfection meaning fastest checkmate in the worst case and without making any assumptions on the opponent. We investigate several Kriegspiel endgames (KRK, KQK, KBBK and KBNK), building tablebases and casting light on some long standing problems.

7.1 Overview

In a zero-sum game of perfect information, Zermelo's theorem [Zermelo, 1913] ensures that there is a perfect strategy allowing either player to obtain a guaranteed minimum reward. In many games, discovering the perfect strategy seems to be synonymous with exploring a major portion of the game tree, which is unfeasible under current and foreseeable computer technology. On the other hand, it is possible to explore significant subsets of the game tree in such a way that, if a particular position is encountered during gameplay, its value has already been computed and the best strategy is immediately available. Most serious programs for playing chess include a so-called "endgame tablebase". Unlike opening books, the same tablebase can freely be used by any number of programs even under tournament conditions, on the basis that it contains no creative work but simply large amounts of processor time.

Currently, tablebases exist for all six-piece chess endings, with seven-piece positions in the process of being computed for the next few years. In many cases, the perfection of tablebase-powered play is unapproachable by even the strongest evaluation function, or indeed the strongest human player. Positions that most experts would have considered draws turn out to be mates in 300 or 500 moves, and seemingly hopeless games can be drawn by repetition. Tablebases are usually obtained through retrograde analysis. Analysis starts from final nodes, the leaves in the game tree corresponding to checkmates and stalemates, and then moves backwards in time to find out predecessors to those positions, until all possible layouts of the desired type have been explored. The concept has been widely studied since the '60s, so there is a large bibliography devoted to chess tablebases and their creation. Bellman's seminal paper [Bellman, 1965] first showed that playing the King and Pawn vs. King (KPK) endgame in chess was a dynamic programming problem in which values for earlier positions could be recursively computed from later positions. At the time of writing, computers were not powerful enough to tackle any other chess ending, but the author suggested that common endgames would be solved within ten years.

The prophecy turned out to be correct, and five years later [Ströhlein, 1970] solved some simple endings with three or four pieces on the board, including KRK, KQK and KQKR. The method was a more elaborate version of Bellman's original algorithm; to find all mates in 1, then find all predecessors of those positions, which would become mates in 2, and so on. At the end of the computation, all positions not in the database are necessarily draws. This thesis marked the start of an arms race towards larger and more comprehensive databases for playing chess endings; in the present day, 45 years since the first paper, retrograde analysis is still being conducted for the game of chess. We cite, among many others, [Stiller, 1991] as one of the best known examples of massively parallel retrograde analysis.

Today, the so-called Nalimov tables or EGT [Nalimov et al., 2000] have become the de facto standard for chess tablebases and are used by most serious programs as well as in human analysis. These tables and the code for accessing them are freely available and all programs can use them in computer tournaments; there are also online interfaces for querying them manually. The indexing scheme of the EGTs is highly optimized, exploiting all possible symmetries and rules of chess in order to exclude redundant or impossible positions. The number of entries in the five-piece table is in the order of the tens of billions.

Being able to play the endgame perfectly has had important repercussions on chess theory itself. Through retrograde analysis it is possible to solve problems that are far beyond the ability of humans and computer programs alike to analyze. Some positions that were widely believed to be draws turned out to be mates in hundreds of moves, and conversely positions that were considered won actually had a successful defense leading to a draw. The "Kasparov vs. the World" game, played on the Internet, has probably become the most famous instance of a game whose outcome could have been different with larger endgame tablebases.

We also recall that retrograde analysis has since been employed in several games other than chess. Checkers pushed it so far that the whole game is now solved [Schaeffer et al., 2007], proving that it is a draw just like tic-tac-toe; its exploration required a cluster of supercomputers running for about 18 years. Some other games, like Nine Men's Morris [Gasser, 1996] and Awari [Romein and Bal, 2003] have been solved, whereas others have made about as much progress as their chess counterparts, like Shogi; see, for example, [Iida et al., 1998] for a classical Shogi endgame analyzed in such a fashion.

The aim of this chapter is to show that the same concept can be successfully applied to a game of imperfect information, as well. Specifically, it can be applied to games which can be somehow transformed into perfect information games in a meaningful way. Unlike other game-theoretical methods, this is only limited to finding positions where a player can force victory with probability 1, but these positions, once found, can be played optimally. In particular, we give an algorithm for solving Kriegspiel endings that have so far only been approached with approximated or heuristic methods, and use it to build Kriegspiel tablebases for several endings, including KRK, KQK, KBBK and KBNK.

7.2 Retrograde analysis under imperfect information

Retrograde analysis works for perfect information games, in which Zermelo's theorem and the Minimax theorem [von Neumann, 1928] hold. Intuitively, for it to work in an imperfect information scenario, we must reduce it to the perfect information case. We do so essentially by abstracting the black king's moves so that multiple "virtual" black kings may exist on the board at the same time. What we get is the merging of several hypothetical states which evolve depending on observations, that is, referee's messages. While the actual message we hear upon trying a move is usually unpredictable, the possible locations of the black king following that message are entirely computable. At this point, it suffices to imagine that the black player decides which message is returned by the referee among the legal ones, and the whole game becomes one of perfect information, albeit one played with different states than chess. If we can always mate even with Black deciding the outcome of our moves (i.e. we can beat an omniscient oracle) then there exists a pure winning strategy that can be stored in a tablebase. The maximum number of moves it takes to mate is also fixed and corresponding to the oracle's best defense.

Throughout our analysis, we will always suppose that the black king not to have any allies on the board. Although considering the king alone makes several tasks easier, this does not stem from any particular limitation in the algorithm; simply, under standard Kriegspiel rules, victory can almost never be guaranteed if there are more black pieces on the board, and as such building a tablebase would be quite meaningless. If the referee were modified to provide more information to the players, then more scenarios would likely become worth investigating. The same theory applies to any subset of an imperfect information game in which one player can achieve certain victory.

The aim of this section is to prove that we can create an algorithm for playing Kriegspiel endings optimally through a tablebase. We will be working

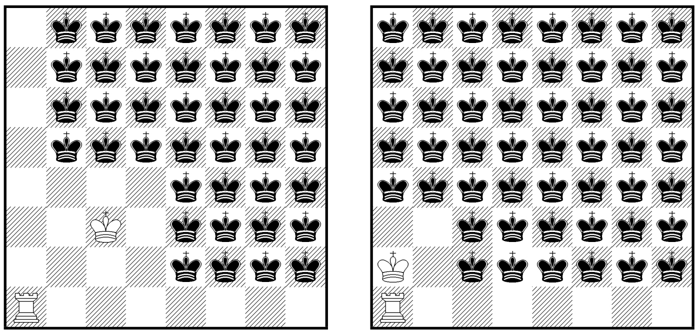


Figure 7.1: Minimum and maximum number of possible kings on a maximal board in KRK.

with diagrams containing actual white chessmen and “virtual” black kings representing possible game states. These diagrams have been called in various ways by different authors. We will continue to use the term ’metaposition’ introduced in chapter 6, though we will provide a more accurate formalization of their structure here in order to prove the correctness and completeness of our method. However, we will occasionally refer to these diagrams simply as ’positions’ or ’situations’ for brevity, when the distinction is obvious from its context. Figure 7.1 shows two examples of maximal metapositions for the KRK endgame (they are maximal because nothing is known about the black king).

Let us begin by defining the sets and functions on which we are operating. Let $Sq = \{\mathrm{a1}, \ldots, \mathrm{h8}\}$ be the set of squares on the chessboard. A *disposition* is a way of arranging the existing, visible white pieces, and we can represent it as unordered piece sequences of the form [Ka1, Rb1], meaning the white king in a1 and a white rook in b1. The *disposition set* D is then the set of all possible dispositions for a given piece set. Calculating the cardinality of D is a simple combinatorial exercise; for example, $|D_{KRK}| = 64 \times 63 = 4032$. For the purpose of our algorithm, however, we can make use of mirroring just like we would in chess. All dispositions can be obtained by mirroring another along the x or y axes, the right diagonal, or combination thereof. Obviously, this would not hold true in endings with pawns, but for the purpose of our scenarios this will always be the case. Mirroring reduces the cardinality of D by a large factor; with only ten king positions to keep track of, we can define a mirrored disposition set D^m that contains fewer redundant dispositions. In this way, $|D^m_{KRK}| = 10 \times 63 = 630$. There is still some redundancy: dispositions in which the king lies on the main diagonal could be halved in size by checking the positions of the other pieces. However, we will be using this incomplete mirroring scheme for the sake of simplicity.

It is easy now to define a *metaposition* in this context as a pair $(d, S), d \in D, S \subseteq Sq$. The rules of chess define a legality function $lgl : D \to P(Sq)$ that accepts a disposition as its input and returns a set of legal squares (a member of the power set of Sq). This function represents the legal locations for the black king, assuming it is white's turn. This is an assumption that we are going to make throughout the chapter - all diagrams show situations in which the white player is to move. This means the black king cannot be in check as we start. This allows us to define the set of legal metapositions

$$L = \{(d, S) : d \in D, S \subseteq lgl(d), S \neq \emptyset\}.$$

The legality function defines the maximum number of black kings that can appear on the board at any given time. That, of course, depends on the particular disposition: Figure 7.1 shows the minimum and maximum sets returned by lgl in the KRK ending: 40 and 52, respectively. These two numbers alone provide a rough estimate of the cardinality of L. Since S can be any subset of $lgl(d)$, which ranges between a given x_1 and x_2, we have that $(2^{x_1} - 1)|D| \leq |L| \leq (2^{x_2} - 1)|D|$. Again, using KRK as an example and only counting the mirror dispositions in D^m_{KRK}, we can place the cardinality of L between $7 \cdot 10^{14}$ and $3 \cdot 10^{18}$; more accurate estimates would place it at around 10^{17}. It is certainly a huge number, especially next to the 24324 positions required to solve KRK in chess, but as we will see this number is not all that significant.

We need two special subsets of L, one representing metapositions that we can always win, and then a smaller set B of "best", maximal metapositions that are our true objective and the only ones required to play the whole endgame optimally. In order to represent these two, we need to formalize the white player's moves and the referee's role. We can define a *move* as an ordered pair of squares, that is, $(s_1, s_2) \in Sq^2$. Chess rules provide us with a *pseudolegal* move function $lglmv : L \to P(Sq^2)$ that returns a set of moves that have a chance of being legal in the current metaposition.

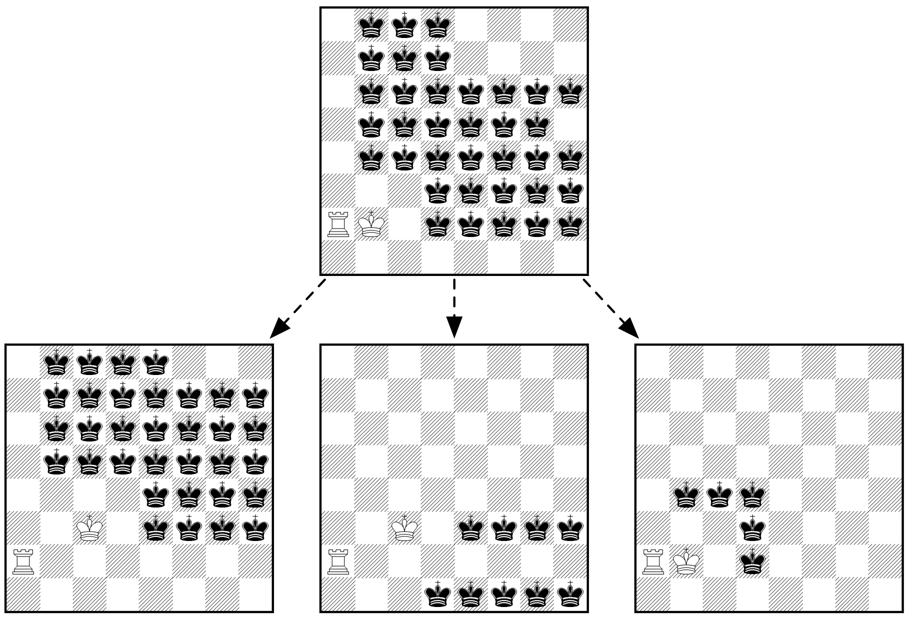


Figure 7.2: If white plays Kc3, he can hear "legal", "rank" or "illegal" from the referee.

Making a move is formalized as follows. There is a set of messages Msg, which models the possible answers from the referee in the particular Kriegspiel version being played. In our case, under ICC rules, $Msg = \{\text{legal}, \text{illegal}, \text{checkmate}, \text{draw}, \text{rank}, \text{file}, \text{short\_diagonal}, \text{long\_diagonal}, \text{knight}, \text{double\_check}\}$. There is one difference between this referee and the one used in an actual game: this one is a worst-case referee and will output a "draw" message if there is the slightest chance of the game being a draw. For example, in Figure 7.1 above, trying Rh1 in either metaposition will result in a "draw" message from the referee, as white might indeed lose the rook from that move. Since we are looking for positions that we can certainly win, we must always consider the most favorable case for black. This is equivalent to stating that in the worst case Black can select the referee's message just like he would select his move in chess. Under these premises, the game turns into one of perfect information.

From a metaposition and a referee's message it is easy to generate a new metaposition that reflects the consequences of that message. Clearly, metapositions will not allow all messages and a majority of them will only be compatible with a few. For example, only king moves can be illegal, and only knight moves can give a knight check. Figure 7.2 shows a metaposition that allows three different outcomes to the same move. It is to be noted that, while in the first two there has been a black move following Kc3 (as seen, for example, in the black kings spreading towards the top right corner), in the event of an illegal move black did not get an opportunity to move. Of course, as long as it is still white's turn and the metaposition is legal, our formalism is satisfied. We can represent this through an *evolution* function

$$ev : (L \times Sq^2 \times Msg) \to (L \cup \emptyset),$$

which accepts a legal metaposition, a move and a message, returning a legal metaposition or the empty set if the message is impossible in this context. In the case of a game-ending message such as "checkmate" or "draw", *ev* returns the same metaposition it received as an input if the message is possible. If "draw" is possible, then no other message is, as explained above. The definition of *ev* is trickier than its actual meaning: it works by erasing the black kings that are incompatible with the message, and if the message is not "illegal" it moves the black kings to any location a real black king could visit from the current positions.

A metaposition $m \in L$ is *won* if it satisfies either one of these conditions:

- $\exists x \in Sq^2, y \in Msg : ev(m, x, y) \neq \emptyset \iff y =$ "checkmate"; that is, there is a certain checkmating move (mate in 1);
- $\exists x \in Sq^2 : ev(m, x, "draw") = \emptyset, \forall y \in Msg : ev(m, x, y) \neq \emptyset \Rightarrow ev(m, x, y)$ is won; that is, there is a move that does not lead to a draw and whose possible outcomes are all won.

This definition of victory excludes probabilistic wins through mixed strategies: it only includes metapositions that are won against an omniscient adversary starting on his most favorable square and possessing foresight of our own strategy. At this point, one can define

$$W = \{m \in L : m \text{ is won}\},$$

the set of all won metapositions. This set, while smaller than L, is ostensibly still very big; it suffices to consider that KQK is virtually always won because

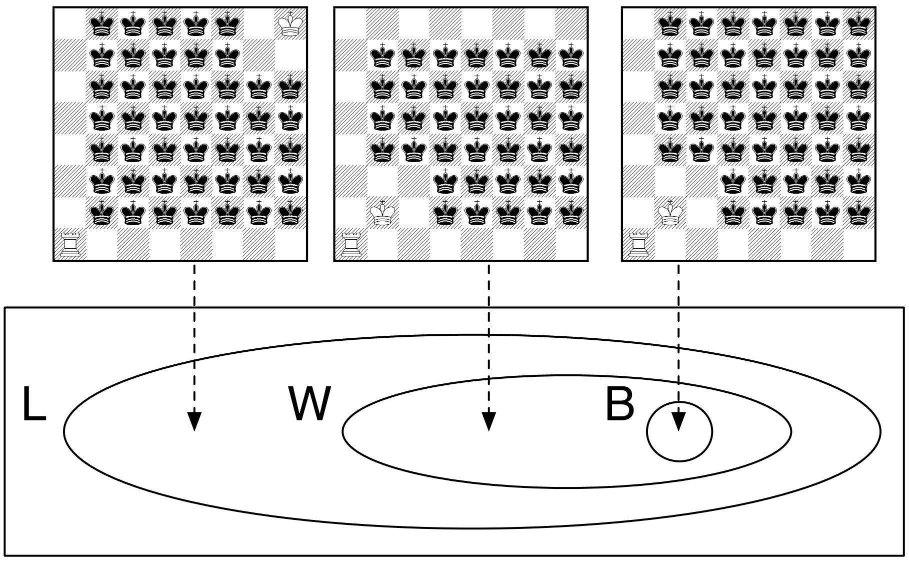


Figure 7.3: L, W, B and a sample element from all three.

the queen cannot be actively attacked by the black king. Our main interest, then, is a small subset of W: the set of *largest metapositions with the same optimal strategy.* In order to define this, we need to define a function $dist : W \rightarrow N$ returning the maximum distance to mate for each won metaposition, expressed as the number of actual white moves (not tries) required to win. We can now define the new set

$$B = \{(d, S) \in W : \exists (d, S^*) \in W, S \subset S^* \Rightarrow dist((d, S)) < dist((d, S^*))\}.$$

This definition should be read as follows: a won metaposition will be in B if any larger won metaposition (a superset of it, with all its black kings and then some) requires more moves in order to achieve victory. In other words, metapositions in B are optimal in the number of moves to checkmate; if any black king is added, the metaposition is either not won anymore, or it is won but it takes more moves to do so. If the metaposition is the largest possible one for its disposition, it is still included in B because the left-hand side evaluates to false.

Figure 7.3 illustrates L, W and B, showing an element from all three sets. The first metaposition is legal, but clearly not won: all moves leave the rook in danger of being captured. The second position can be won with

Boyce's algorithm, but it is possible to win a larger one – the third – with the same number of moves, and indeed with the same strategy. The reason is that the information about the black king not being on the eight rank is useless: the only move that might get a different outcome because of it is Ra8, which is unsafe and must be discarded. All other moves generate the same metapositions in the two cases, so the smaller one can be sacrificed without loss. Thankfully, most elements of W are like this, and we are entitled to hope that B may contain a small, computationally feasible fraction of the total. It is readily seen from this example that there are $2^7 = 128$ elements of W that are like the second metaposition but with any combination of black kings on the eight rank, and hence are not in in B. A deeper investigation would reveal that we can take away even more squares with no consequences after the first move, thus excluding tens of thousands of elements from B.

At this point, one might wonder about the usefulness of B and the reason for its definition. Why not just define it as the set of all won positions that are not a strict subset of any other won position? The reason is a practical one, and it is best demonstrated with a practical example. Figure 7.4 shows two metapositions, with (a) being a subset of (b), but both their distances to mate and correct strategies are different. In (a), keeping the king confined to just one file is the optimal strategy, which is obviously not possible in (b). If B is to capture all "important" metapositions, it obviously has to contain both (a) and (b). If both could be solved in the same amount of moves, one could simply use (b)'s strategy for (a), as well – the additional information in (a) would be ignored at no cost, and we would not need to have (a) in B.

More specifically, it is our aim to build an algorithm that *exhaustively computes B given a set of white pieces.* Also, the algorithm will associate to every element b of B its corresponding $dist(b)$, as well the optimal move (or sequence thereof, should the first be illegal) to try from there. This will be our endgame tablebase and it is enough to play any won metaposition optimally, as shown in the following

Theorem. *Let B be available for a given set of white pieces, and let $dist(x)$ be known for all $x \in B$. Also, let all $x \in B$ have an optimal sequence of moves $m_{x1}, \ldots, m_{xn} \in Sq^2$ such that playing such a sequence will lower the distance to mate by at least 1. Then, it is possible to:*

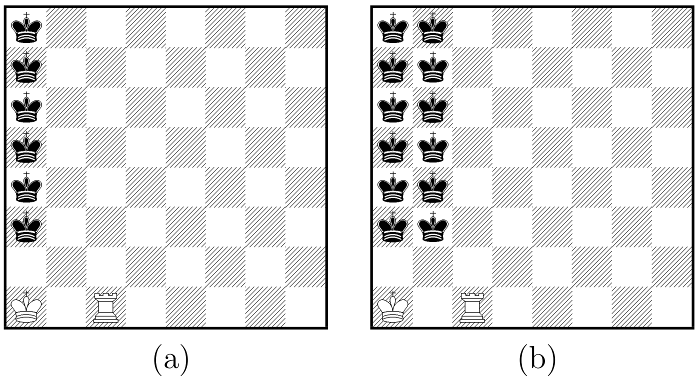


Figure 7.4: (a) is a mate in 9, play Rb1; (b) is a mate in 13, play Kb2, or if illegal Kb1; however, (b)'s strategy can solve (a) as well.

- *determine whether any legal position $l \in L$ is won ($l \in W$?)*
- *if it is won, determine $dist(l)$ and the optimal strategy leading to checkmate.*

Proof. We prove this theorem constructively, that is, we provide a strategy for querying B like a tablebase. This is a very simple strategy that can be used in a real-world program such as Darkboard. In this context, playing "optimally" means that at any given time we can checkmate in $dist(x)$ moves at most, even against an omniscient opponent. We do not make any assumptions on the nature or play patterns of the enemy; we simply consider worst-case performance, much like a chess tablebase.

Suppose we need to solve a legal metaposition $l = (d, S) \in L$. The querying algorithm is as follows:

- Look for all $(d, S^*) \in B$ such that $S \subseteq S^*$. If none exist, meaning that the metaposition has no supersets in B, then l is not won.
- On the other hand, if it is won, select the (d, S^*) with the shortest distance to mate. Play the corresponding sequence of moves.

We need to prove that both steps are correct. The first step requires us to prove that $(d, S) \in W \iff \exists (d, S^*) \in B, S \subseteq S^*$.

- $\Rightarrow$: obvious by construction. If a metaposition is won, it either is in B or its superset is.

- $\Leftarrow$: $B \subseteq W$, so (d, S^*) is won. Any subset of a won metaposition is also won – one can simply pretend not to know the additional information. A strategy that solves (d, S^*) also solves (d, S), hence (d, S) is won.

The second step requires us to prove that the selected strategy is valid and optimal. Obviously, the strategy is valid because of the same argument as before: a strategy that solves a metaposition also solves any of its subsets. This does not guarantee that it will do so optimally, however: as seen in Figure 7.4, one can solve (a) with (b)'s strategy, but doing so requires 13 moves instead of the optimal 9. On the other hand, suppose that the selected strategy is indeed sub-optimal, that is, $dist(d, S) < dist(d, S^*)$. But this means (d, S) should have been an element of B, as well: by construction, $B = \{(d, S) \in W : \exists (d, S^*) \in W, S \subset S^* \Rightarrow dist((d, S)) < dist((d, S^*))\}$. Since $dist(d, S^*) > dist(d, S)$ is the minimum distance for a superset of (d, S), then (d, S) meets all requirements for being in B, and it should have been returned by the algorithm. Hence the selected strategy is both valid and optimal.

The theorem above proves a very important point: that it is possible to play all won metapositions optimally while only knowing a small subset $B \subset\subset W$. The next sections describe the actual retrograde analysis algorithm that builds B and computes the data stored with its elements, thus making a complete endgame tablebase that can play Kriegspiel endings optimally.

7.3 A perfect play algorithm

We need to construct B iteratively. If $B_x \subseteq B = \{b \in B : dist(b) \leq x\}$ represents the subset of metapositions in B that can be won in at most x moves, it is clear that, by increasing x, at some point $\exists k \in N : B_x = B \; \forall x \geq k$, because B is not infinite. Thus, a simple inductive reasoning shows that what is really needed to construct B is:

- an algorithm for building B_1;

- an algorithm for constructing B_{x+1} from B_x.

Intuitively, B_1 is not hard to construct: it suffices to try every move for each disposition of white pieces, and see which locations of the black king would be a mate in 1 on a regular chessboard. The latter algorithm is obviously much more challenging. The simplest way to think of the solution is: if, given a metaposition and a move, we can establish that all outcomes of that move are either in B_x or subsets of elements in B_x, then we know that such a metaposition can be won in at most $x+1$ moves. The actual problem is making the metaposition maximal, that is, finding a metaposition such that none of its supersets can be won in $x+1$ moves. Still relying on intuition, rather than building random metapositions and testing them against the elements in B_x, the solution will be constructed from the elements in B_x themselves.

The key observation to be made here is that given a metaposition $b \in B$, a move $v \in Sq^2$ and an *assignment set* associating metapositions to referee's messages as in $A = \{(b_1, m_1), \ldots, (b_n, m_n)\}$, $b_k \in B$, $m_k \in Msg$, m distinct, it is possible to construct $b^* \in L : \forall (b_x, m_x) \in A \; ev(b, v, m_x) \subseteq b_x$; that is, b^* is a legal metaposition whose outcomes will be contained in the metaposition associated with each message. Moreover, if the assignment is exhaustive, $b^* \in W$; the metaposition is won because there is a move whose outcomes are all won. If one tries all possible assignments, sooner or later the maximal elements belonging to B will be generated. The method for doing this operation is lengthy, but rather trivial. The problem merely becomes one of exploring millions of assignments and storing the best ones.

Figure 7.5 contains a skeletal version of the algorithm. It accepts two parameters as its input: `entryList`, which is basically B_x, and the depth level $x+1$. What it does is create all possible dispositions of white pieces, and for each of those all pseudolegal moves are considered. All compatible assignments are tried: this means that the same metaposition in entryList will be tested for each referee's messages, and the same metaposition can appear more than once in the same assignment set for different messages – even all of them. After all, the meaning of the assignment set is "if message x happens, the problem reduces to previously solved metaposition y", so it is perfectly possible for a single metaposition to cover several messages.

```
kriegRetrograde(entryList,depth)
begin
 added = false;
 for each disposition of white pieces P do
  for each pseudolegal move M do
   messages = possibleMessages(P,M);
   for each assignment A of entries from entryList to messages
do
    if (generateAndAdd(entryList,P,A)) added = true;
   od
  od
 od
 if (added) kriegRetrograde(entryList,depth);
 else if (entriesWithDepth(depth+1)) kriegRetrograde(entryList,depth+1);
 return entryList;
end
```

Figure 7.5: Pseudocode listing for main retrograde function.

The method `generateAndAdd` creates a metaposition from the assignment set and checks if it is a subset of something already in B. If it is new, it adds it to `entryList` and returns true. If all assignments are tested without any additions to the database, this depth level is considered exhausted, with `entryList` now representing B_{x+1}: the algorithm now starts over at the next depth, but if there are no suitable metapositions of the right depth, that is $B_x = B_{x+1}$ it terminates execution, having found B in its entirety. On the other hand, if it found new metapositions, it starts a new iteration at the same depth; this is necessary because of illegal moves. Metapositions of depth $x + 1$ found during the first iteration can create more metapositions of the same depth in the second iteration by being associated to the illegal message in the assignment set, and so on. In most cases, it takes 3-5 iterations to completely clear a depth level; these correspond to the 2-4 illegal moves the white king might make, at most, before a legal move is found.

This algorithm requires exponential time in the number of pieces and possible referee's messages; KQK takes the longest time to compute because the queen can check in four different directions. The space required by the algorithm only depends exponentially on the amount of white pieces, not the messages. The size of B is bounded by the size of L, which can be estimated as shown in the previous section.

7.3.1 Computational complexity

The pseudocode listing can be misleading as to the true complexity of the algorithm, also because it is easy to dismiss several key elements of the problem as mere constants. In fact, whenever such statements as “try all assignments” are made, even small constants can turn out to be very troublesome. In this case, the algorithm runs in exponential time depending on at least two factors: the number of white pieces on the board and the number of possible referee’s messages. The former affects the number of dispositions: moving from two white pieces (as in KRK) to three (as in KBNK) marks a major increase. The latter affects the number of assignment sets, making it grow exponentially. For example, a move in KRK can only have three outcomes at most: legal, illegal and check (rank and file are mutually exclusive). By contrast, in KQK the queen can generate four different checks with the same move, plus the legal message. As with all exponential algorithms, two messages can make the difference between an iteration taking hours or weeks or even months.

The space required by the algorithm only depends exponentially on the amount of white pieces, not the messages. The size of B is bounded by the size of L, which can be estimated as shown in the previous section.

7.3.2 The lookup algorithm

Once the algorithm has finished, it returns a full list of metapositions, each having a best move and a distance to mate in the worst case. The tablebase goes through a series of post-processing steps and is finally stored as a text file in which every line represents a single entry. A sample line from the KRK tablebase reads as follows:

```
kkkkkk2/7R/kkk5/8/4k3/2K1k3/4k3/3k4 26 Kc3-d3 Kc3-d4 Rh7-h5
```

The metaposition itself is represented through standard FEN (Forsythe-Edwards Notation), with multiple black kings on the board. This board representation is followed by the maximum distance to mate and the sequence of moves that the white player should try from here, starting from the first and moving to the next should it be illegal.

This database is used as indicated in the proof to the Theorem. The player searches it with a metaposition representing the current state of the game. All entries that are supersets of it are returned, and if none exist, it means that it is not possible to force a mate from here. Among these entries, the player selects the one with the shortest distance to mate; in the event of a tie, he will pick the one with the lowest number of states (black kings). He will then proceed to play the corresponding move, which is optimal.

7.3.3 Validation

The problem of proving that the final database is correct and complete has no simple practical solution, just like there is none for the perfect information case. [Nalimov et al., 2000] provides the following reasoning: one can run a self-consistency test to make sure the tablebase is correct, i.e. entries do not contradict each other, but time is as good an optimality test as we get. The paper goes on to mention that chess programs have used Nalimov tables for years without reporting any mistakes or non-optimal mate sequences; this is a reasonable indicator of their correctness. As noted, there have been cases (albeit rare) of retrograde analysis conflicting with previous retrograde analysis due to some error or bug in the implementation. These programs are necessarily full of tricky details and performance optimizations, and there is always a small risk of making a mistake somewhere.

Since the Kriegspiel tablebase code is too large and complex (about 10,000 lines of Java code in its final version) for formal analysis, we are also limited to checking the database for internal consistency, and awaiting the ultimate judgment of time about its optimality. All the endgames studied with this method have improved existing algorithms and solutions by several moves, sometimes by a large margin. The impression one gets from looking at the database entries is that they are highly efficient and further improvement seems unlikely.

It is, however, quite easy to perform a consistency test to prove that, at the very least, the database offers consistent solutions. The process is more or less identical to how a chess tablebase would be tested. Each metaposition in the tablebase is examined with each possible pseudolegal move. The following must hold true:

- there is a move whose outcomes are all contained in the database, unless it is a trivial mate;
- if the metaposition being tested is a mate in n moves, the worst-case outcome of this move is a mate in $n - 1$, or n if the outcome is the illegal message;
- in the latter case (illegal move), the resulting metaposition satisfies the same constraints (this prevents a metaposition from self-validating since the outcome of an illegal move is a subset of the original entry);
- no moves with shorter distance to mate exist.

These steps are taken as part of the **post-processing** phase that follows the exploration phase and exports the database into a text file. Actually, it is at this stage that optimal strategies are reconstructed for each metaposition, as optimal moves are not stored during the algorithm itself in order to save memory space.

7.4 Implementation

The retrograde analysis algorithm was implemented in the Java programming language. The main reason behind this choice was that the code could benefit from a number of Kriegspiel-related primitives defined elsewhere in the Darkboard project, which undoubtedly sped up development of this component. In retrospect, Java might not have been the most suitable programming language for such a memory-intensive application, but it managed to run all test scenarios on a high-end machine.

Simplicity was the primary concern behind all design choices, and straightforward classes and data structures were preferred over faster but more complicated entities – for example, simple two-dimensional matrices represent the board instead of more complex structures such as bitboards. The program is comprised of only seven classes, modeled in the UML class diagram depicted in Figure 7.6. Their purpose is as follows.

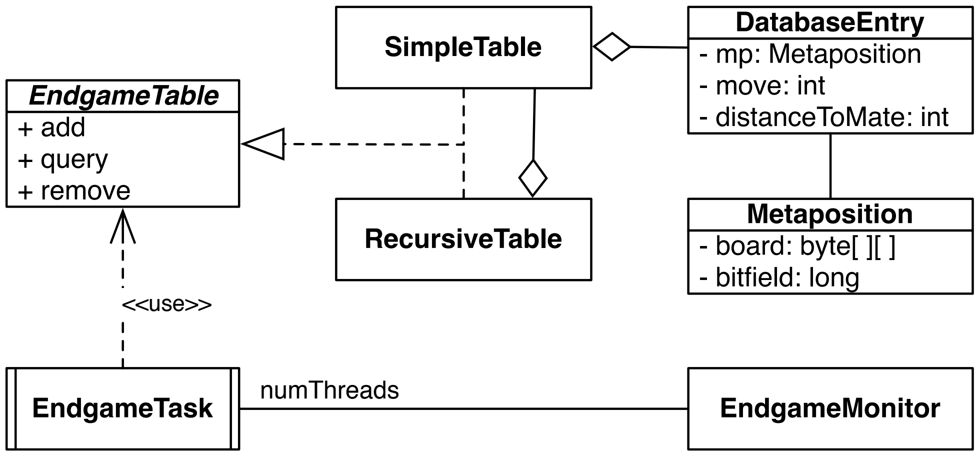


Figure 7.6: UML class diagram for the component that creates Kriegspiel tablebases.

- **Metaposition**: the class representing a single metaposition. As seen in the diagram, the metaposition has two different but equivalent representations: an explicit two-dimensional 8x8 array in which each element is a square of the board, and a long (64-bit) integer bitfield in which each bit simply indicates whether a square can hold an enemy king or not. The latter form does not store information about allied pieces, and therefore is not complete on its own, but it is very useful to perform certain computations much faster than we would be able to with the array form. The most important of these tasks is the *comparison* between metapositions. Since a large portion of the algorithm consists of comparing metapositions to establish if one if a subset of another, such an operation must be carried out as quickly as possible. With the 8x8 matrix alone, we would only be able to perform a naive square-by-square test, but with the bitfield only three bitwise operations are required to compare metapositions: one XOR and two ANDs. Figure 7.7 shows the comparison code used in the program: the XOR highlights the differences between the two metapositions, and the ANDs check which one has the bit set to 1. If Java is running in 64-bit mode, as it always did in our scenarios, this snippet is particularly fast as the long integer can be loaded into a register in one go.

```
compare(mp1,mp2)
begin
 long compField = (mp1.bitfield ^ mp2.bitfield);
 if (compField==0) return COMP_EQUAL;
 long c1 = mp1.bitfield & compField;
 long c2 = mp2.bitfield & compField;
 if (c1!=0 && c2==0) return COMP_BIGGER;
 if (c1==0 && c2!=0) return COMP_SMALLER;
 return COMP_NOT_COMPARABLE;
end
```

Figure 7.7: Using bitfields for quickly comparing metapositions.

- **DatabaseEntry:** the container class forming the building block of the tablebase. It holds a metaposition, the optimal move and the distance to mate if the optimal move is played. In later, more optimized versions of the program the move is not saved in order to save space: when the database is complete, it is very easy to reconstruct the optimal move or move sequence by just querying the database in a post-processing phase.

- **EndgameTable:** a generic interface for using an endgame tablebase, it is implemented by two different classes, one of which provides simple linear access to its contents, whereas the other works in a more complex, scalable way.

- **SimpleTable:** the simpler implementation of EndgameTable, this is basically a wrapper class for a Vector of DatabaseEntry objects. The basic constraints for this class to work is that all of its contents belong to the same disposition, that is, the white pieces are on the same squares. While this is a fast class, it is obviously not very scalable to arrays with millions of elements, but it is used throughout the program.

- **RecursiveTable:** this is the class representing a full tablebase. It recursively contains more RecursiveTable objects in a hierarchical tree structure. A query is redirected to the correct child depending on the positions of the white pieces: the top level switches according to the location of the first piece, and so on; the hierarchy has as many levels as there are white pieces. The bottom tier is formed by SimpleTable objects storing the actual positions.

- **EndgameTask:** the active class that runs the actual algorithm described in the previous sections.

- **EndgameMonitor:** a monitor class handling synchronization of several EndgameTasks running in parallel. This class launches any number of tasks, each running in a separate thread, and then assigns units of work to each, collecting the finished results.

7.4.1 Parallelization

Parallelization is illustrated in Figure 7.8. Retrograde analysis lends very well to parallelization, and this algorithm is no exception: the result is a simple version of the producers-consumers problem. EndgameMonitor creates a number of EndgameTask object, each running in its own thread. Tasks request an assignment from the monitor, which responds by transmitting a disposition of white pieces as well the current database. The thread will have to compute all new metapositions of the specified type, and then send the resulting list back to the monitor before requesting the next assignment. The new metapositions are not sent directly to the database to avoid read-write conflicts with other ongoing tasks as well as to keep the computations in each thread unaffected by the others. Throughout this part of the algorithm, the database is therefore read-only.

Only after the iteration is complete, does the monitor update the database with all the new entries that the tasks have computed. Because interaction between the various tasks is virtually non-existent, it is possible to achieve a near-linear speedup until this point, though this updating phase is obviously not as efficient. Thankfully, the update list is not large enough to cause a major bottleneck in the scenarios tested so far, but a more sophisticated method may have to be used for larger endings such as the ones with five pieces on the board.

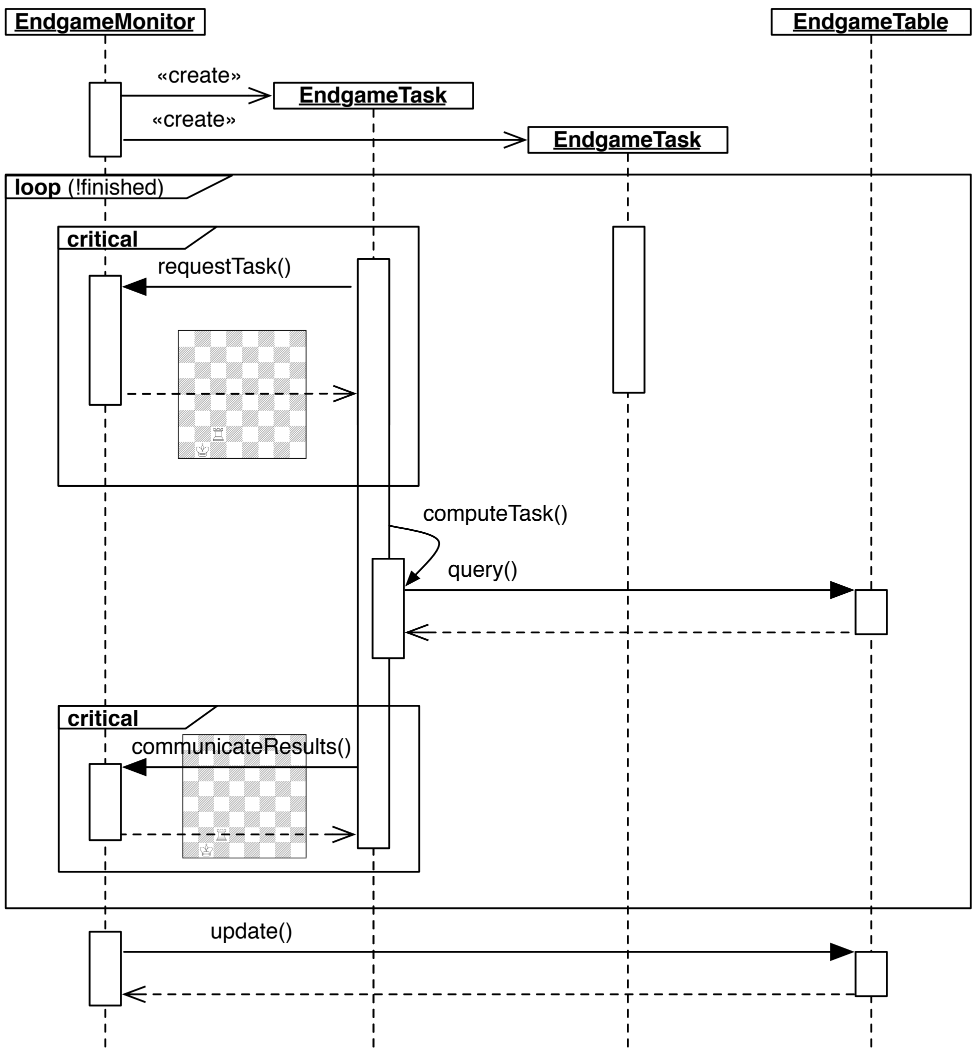


Figure 7.8: UML sequence diagram showing parallelization of the main algorithm.

7.4.2 Optimization

The basic version of the algorithm is unable to solve any but the simplest endgame, KRK, in reasonable times (spanning days rather than weeks and months). There are several sources of inefficient behavior in the code, but the single most important reason stems from the database itself and was unforeseen before the first testing phase. *Kriegspiel tablebases do not have*

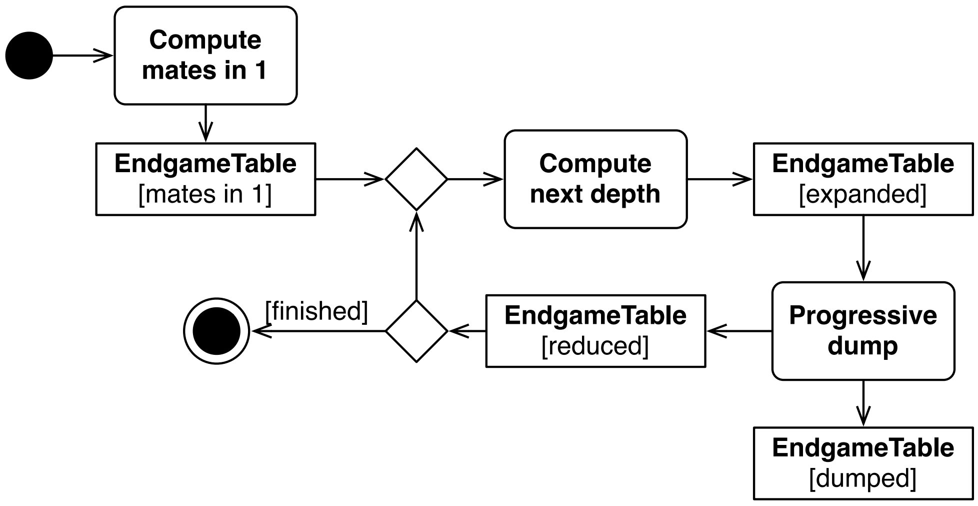


Figure 7.9: UML activity diagram of the main algorithm with progressive dumping.

a uniform structure with respect to the white pieces. The opposite is true; some dispositions of the white pieces have many more metapositions in the database than others. In particular, the size and complexity of the tablebase increases greatly as the white king gets closer to the center of the board. In other words, positions are much more complex and subtle – easily by a factor of 10 or more – when the king is in the middle.

This is far from a minor detail for a combinatorial algorithm. It means that computing metapositions in which the king is near the center of the board takes much longer than if the database was uniformly populated. Suppose the database is uniform and each disposition of the white pieces has 50 metapositions. Let us also assume that each move generates on average 3 referee's messages; then, given any metaposition and a pseudolegal move, we expect to examine about $50^3 = 125000$ combinations. Now suppose that the database contains 10 metapositions with the king in and near the corner, and 100 metapositions with the king near the center. Under these assumptions, a move near the edge might require us to examine only $10^3 = 1000$ combinations, but a move near the center will generate $100^3 = 1000000$ combinations. Since the actual numbers can get even worse than this example, optimization is necessary for the computation to run at acceptable speed.

Most optimizations revolve around two simple ideas: caching whenever possible, and avoiding exploration of useless combinations, that is, assignments that are guaranteed not to yield any new metapositions. If possible, the two approaches should be enforced simultaneously, so as to narrow down the number of useful combinations and save this information for later. The single most significant improvement is **progressive dumping**. It is a filtering procedure that takes place at the end of each depth level, discarding a portion of the accumulated database. The process is illustrated in Figure 7.9. Progressive dumping splits the database into two parts after a new depth has been conquered. *Metapositions that are subsets of others are dumped, regardless of their distance to mate.* This happens for a simple reason: those metapositions will not further contribute to the database. Even though their distance to mate may be shorter than that of their supersets, this is irrelevant at the next depth levels to be computed. When computing depth 20, the algorithm is looking for positions for which at least one outcome yields a mate in 19 (or 20 if this outcome is the illegal move). Whether the other outcomes are mates in 2 or 18 does not change the distance to mate, so we may as well only keep the largest metapositions. The full endgame tablebase is therefore the union of all dumped tables from each step of the algorithm, as well as the positions left in the database after the last step, which are no subsets of any other metapositions in the database.

Progressive dumping is a very effective strategy: depending on depth and the specific endgame type, it dumps 10-25% of the tablebase after each new depth. The ratio of dumped positions increases steadily with depth as metapositions become larger and more inclusive. Since the process is repeated over dozens of depths, it allows the algorithm to only focus on a small fraction of the full tablebase.

Component caching is another important optimization: it stems from the observation that, while we need to examine many combinations, they are formed by the same components being checked over and over again. Component caching saves such metapositions as Figure 8.4, (b) in a separate table instead of recalculating it each time (a) is being examined. Even more importantly, this is yet another case in which we can discard components that are subsets of others (provided their distance to mate is also equal or higher), which helps the computation immensely as these metapositions are more regular-looking than the elements they combine to form, and thus a high amount of them can be discarded.

Finally, **active entry marking** allows to save time during the second and later iterations of a given depth. These iterations collect all the metapositions that can be obtained with multiple illegal moves, but clearly the amount of new entries decreases sharply with each iteration (on average, by a factor of 10). Entry marking is accomplished by storing the iteration number in which a new entry was found. It is obvious that metapositions found on the n-th iteration will have to make use of one or more metapositions found on the $n-1$-th iteration, or else they would have been discovered earlier. Therefore, combinations without one of these newer metapositions in their components are not checked and immediately discarded. This makes iterations progressively faster as fewer entries are marked as new.

Chapter 8

Perfect play results

In this chapter, we list and analyze some noteworthy results obtained through our retrograde analysis algorithm, with respect to four interesting Kriegspiel endgames: king and rook, king and queen, king and two bishops, and finally king, bishop and knight. In theory, it is possible to run and solve any lone-king scenario with this algorithm; however, the current implementation will need better memory management before it can run larger endgames such as KRRK, KQRK and KNNNK. These endings deal with more metapositions than our algorithm can handle on the current test machine. The first two also offer the additional problem of rare, but possible instances in which White might want to sacrifice one of his pieces to achieve checkmate faster.

Before considering each of the four tested endgames in turn, we devote Section 8.1 to a general analysis of tablebase size and distributions, as well as giving a rough estimate of its compression power.

8.1 Test cases

The tablebase-building algorithm was implemented in Java, modified for parallelization and run on an eight-core machine. Four scenarios have been executed so far: KRK, KQK, KBBK and KBNK. Execution time ranged from about six hours for KRK to seven days for KQK, with KBBK taking about three days and KBNK taking five. The dimensions of the resulting databases are radically different, as seen in Figure 8.1. The figure shows distributions by distance to mate, and provides visual information as to how large a database

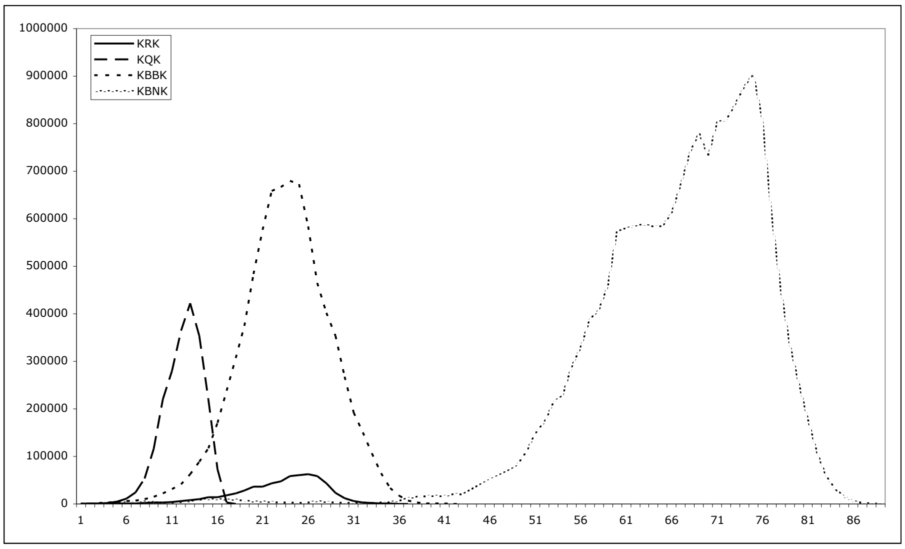


Figure 8.1: Metaposition distributions by distance to mate in the four tablebases.

is; moreover, checkmates for the general positions usually encountered in practical gameplay (that is, positions in which White knows nothing about the black king) are mostly located near the distribution's peak. Entries to the right of the peak are riddle-like and require the white player to spend moves protecting his pieces and reaching a stable configuration. It should be noted that KBNK is unique among the four in its irregular development. In particular, very few entries exist before depth 35, after which the database explodes. The fact it takes so long to find general strategies for KBNK is probably the main reason why a general pure strategy for this endgame was never found through manual analysis.

Figure 8.2 represents distributions by the amount of black kings on each entry. If the database contained every possible legal metaposition, the resulting graph would resemble a Gaussian, being the sum of binomial distributions with similar coefficients (each king either is or is not present on the board). The actual databases all show a skew towards entries with fewer kings; the longer the endgame, the larger the skew. This fact does not immediately prove anything about the database's compression power, that is, the ratio of database entries compared to all won metapositions. In this sense, KQK is

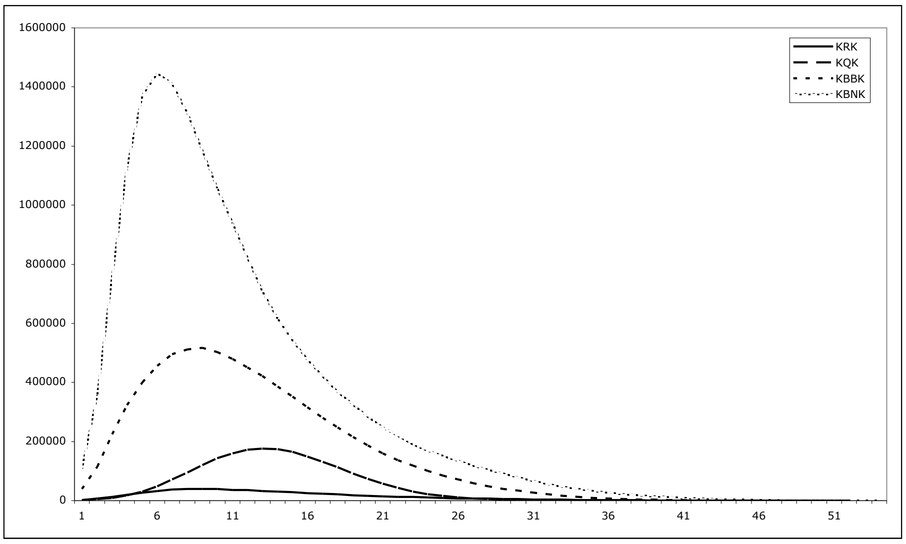


Figure 8.2: Metaposition distributions by number of black kings on the board.

the easiest case to compute since almost every game can be won with probability 1, the queen being safe even when the white king is far away. The KQK database is slightly over two million entries, with about 10^{16} possible metapositions. This means that the database contains two in 10^{10} elements, having a compression power of approximately 99.99999998%. The other endgames are less straightforward, since they all contain entire classes of situations that cannot be won with absolute certainty. If the king starts out separated from the other white pieces, victory will not be guaranteed in a majority of cases. For KRK, it can be argued that compression ratio is even higher than KQK, because it is less than one third the size and roughly half legal metapositions can be won (if the rook starts on the same or adjacent rank or file to its king, the game is almost always won).

8.2 KRK

KRK is arguably the simplest Kriegspiel endgame in which victory can always be obtained from a sizeable amount of initial configurations and no information on the black king. Using mirroring on the x, y and diagonal

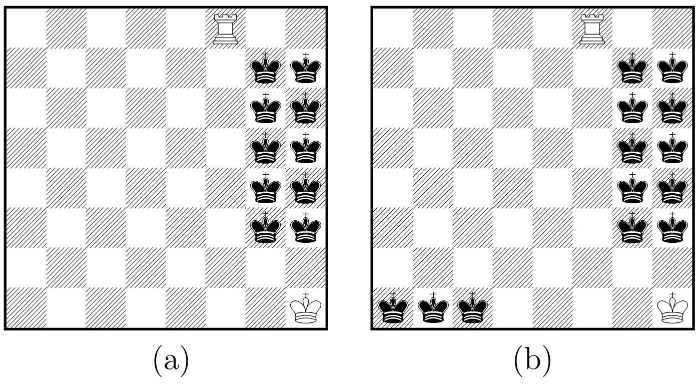


Figure 8.3: Sample tablebase lookup in KRK: (a) is entered, (b) is found: mate in 14 with Rf1.

axes, the problem of KRK in Kriegspiel is described with a tablebase of 635,968 metapositions. KRK in chess is fully described with about 23,000 positions, making the equivalent Kriegspiel problem about 30 times as complex. Results may vary to a degree, depending on optimizations and storage policies for metapositions that are subsets of other entries; for instance, in our tablebase boards with the white king on the main diagonal are not further mirrored with the position of the rook. There are 2207 entries with only one black king on the board. These metapositions look exactly like chess positions, and their presence is roughly equivalent to saying that roughly 10% of the time there is a specific, optimized strategy for checkmating the black king that can only be applied if its initial position is known with certainty. The remaining 90% are subsumed inside larger metapositions with two or more kings.

Figure 8.3 shows an example of a KRK tablebase query. The lookup algorithm returns the superset of (a) with the shortest distance to mate, which happens to be (b). It is easy to see that the best move to play here is Rf1, and it is also clear that such a move works well in the three additional cases included in (b). In event of a check, Rf3 confines the black king to two ranks (note this would not be possible with a king originally in d4). Obviously, if those three kings were the only ones, the tablebase would return a much faster mate with Rf2.

Figure 8.4 contains three more examples of tablebase entries with wildly

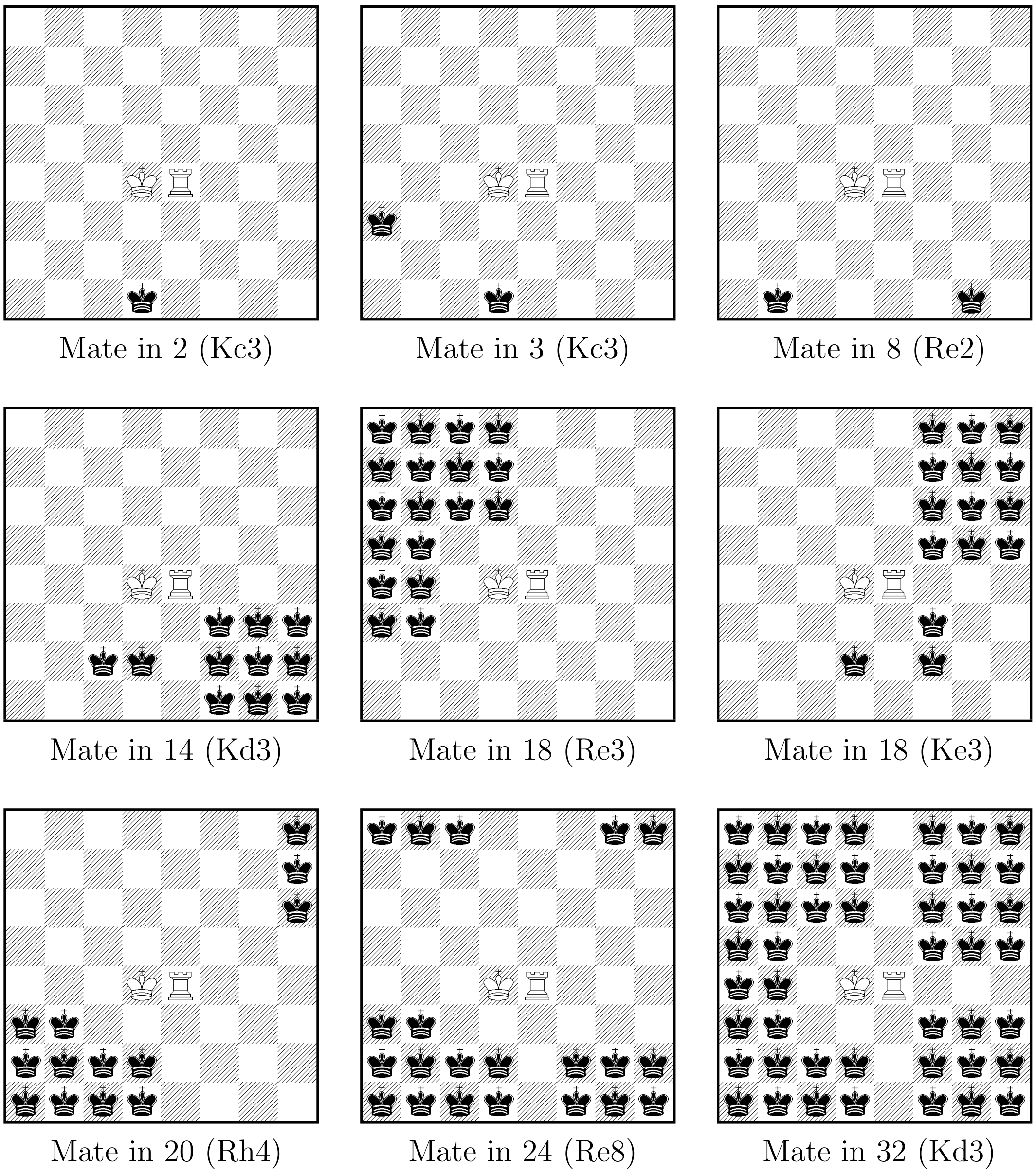
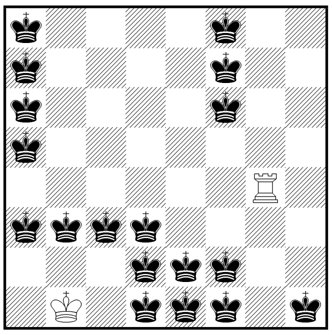
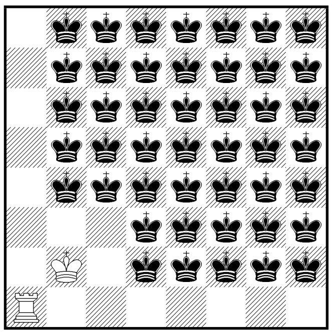
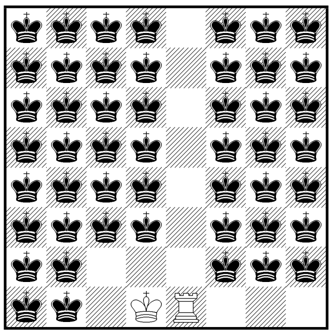


Figure 8.4: A small selection of metapositions from the KRK database with king in d4 and rook in e4 (out of 2483).

different depths and complexities but with the white pieces in the same squares. Since there are 2483 such entries in the tablebase, one currently to search through them all during the lookup algorithm. While a single

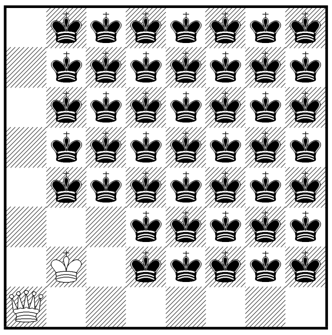
(a) (b) (c)

Figure 8.5: (a): mate in 37, longest forced sequence in KRK; (b): Boyce's starting position, mate in 26; (c): Magari's starting position, mate in 30.

lookup will not affect performance much, a better indexing or classification method might be needed for particularly large endgames.

The longest forced mate sequences in Kriegspiel KRK are 37 moves long, making the 50-move rule irrelevant in this endgame. There are 50 entries for this depth in the tablebase. Figure 8.5, (a) is one such entry: White needs to spend several moves escorting his rook to safety. In the worst case, assuming the referee is always silent, this task requires eight moves: Rf4 Rf8 Kc2 Kd3 Rg8 Rh8 Rh1 Rd1.

Boards (b) and (c) represent situations that are much more likely to happen in a real game. In particular, board (b) is the starting position for Boyce's algorithm given in [Boyce, 1981], and board (c) is the starting position for Magari's algorithm, from [Magari, 1992]. Boyce's directives are based on trapping the king in a single quadrant of the board with the rook and then using the king to push back the opponent. Magari's method consists of starting from (c) and isolating the king on one side of the board by playing Kd2 Re2 Kd3 Re3 and so on, scanning the board until a check reveals the location of the enemy king. The tablebase shows that Boyce's method is a better approximation of the shortest mate. Boyce's position is a mate in 26, four moves shorter than Magari's position. Thus, Boyce has a good understanding of a convenient starting position: from here, optimal play is Kc3 (longest mate is 26 regardless of whether this is legal) Ra2 Kd4 Rb2 Rb3. Additionally, optimal play from (c) does not follow Magari's algorithm;


(a)

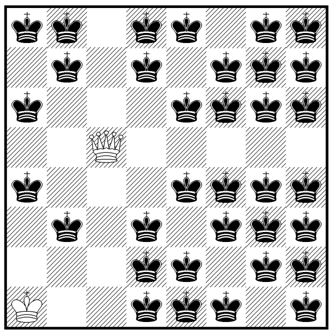
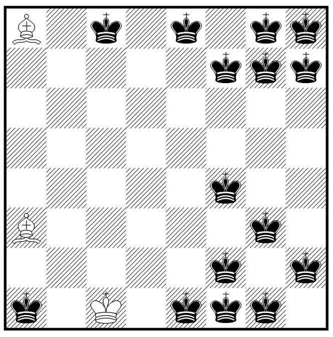

(b)

Figure 8.6: (a): Mate in 14; (b): longest KQK mate, 18 moves.

instead, it is more convenient to play Ke2 Kf3 Re2, followed by the unusual Re4 Re2 which can clear the black king at h3 without White having to move his own king. This allows him to safely play Rh2 and reach a Boyce-like scenario that is a mate in 24.

8.3 KQK

The king and queen vs. king endgame is certainly the fastest to win, yet one of the slowest to compute because of the larger number of referee's messages that most moves can generate. At 2,150,833 entries, it is over three times as large as KRK, and can be won roughly twice as fast with similar strategies and comparatively fewer illegal moves. The equivalent of Boyce's starting position, shown in Figure 8.6, (a) is a mate in 14 following an almost identical strategy: Kc3 Kd4 Qb2 Ke5 Qc3. The main difference from KRK is that the king pushes for the center more aggressively, and the queen follows from a distance. The longest forced mate in KQK takes 18 moves in the worst case; there are 33 such instances in the tablebase, one of which is depicted in Figure 8.6, (b). Interestingly, one might wonder why this mate requires 18 moves instead of 17, since it appears to be only three moves from the situation in (a). In fact, the tablebase correctly recognizes that Kb2 Qa3 Qa1 is the wrong strategy, as it leads to stalemate after the first move if the black king is on a4. The correct strategy is Kb1 Qc2 Qa2 Kc2 Kd3, etc.


(a)

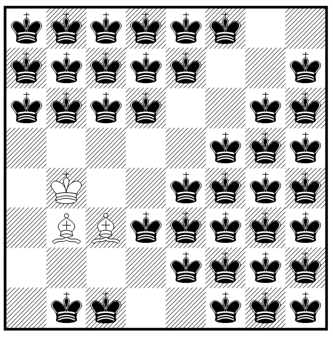

(b)

Figure 8.7: (a): Mate in 43, play Bd6; (b): mate in 32 with Ferguson's method, actually mate in 27.

8.4 KBBK

The KBBK database contains 7,887,296 entries, with the longest forced mate spanning 43 moves; there are only 5 entries at this depth, one of which is shown in Figure 8.7, (a). Obviously, because a game of Kriegspiel is also a valid game of chess, the tablebase only contains positions with the two bishops standing on differently colored squares.

This endgame, together with KBNK, is particularly interesting because there is existing research to compare the tablebase with. KBBK is studied by Ferguson in [Ferguson, 1995], which correctly points out that it cannot be won for every starting position, even if the white pieces are initially safe. This is because the two bishops cannot directly protect each other; they can only stand side by side and block the enemy king from the front and back, but not the flanks. When the white king moves to clear one quadrant, it leaves a bishop unguarded, therefore the game cannot be won with probability 1. However, if the pieces start out close enough to the edge of the board, the king needs only protect one flank, and victory is guaranteed. Figure 8.7, (b) depicts a good starting position for White, and is the top level example provided by Ferguson. His estimate of 32 moves is cut by five in the tablebase, setting the distance to mate to 27. Clearly, both sources agree that Kc4 is the correct move to play in this context as it is basically the only plausible option.

Position	**Ferguson**	**Tablebase**
k6B/k1K5/8/8/2B5/8/8/8	4 (Kb6)	4 (Kb6)
k7/k1K5/3B4/8/2B5/8/8/8	5 (Be7)	5 (Be7)
k1B5/k7/8/1K2B3/8/8/8/8	7 (Kc6)	7 (Kc6)
kk6/1k1K4/8/8/2BB4/8/8/8	9 (Kc6)	9 (Kc6)
8/8/k1B5/k1B5/2K5/8/8/8	12 (Bd4)	11 (Kd5)
kkkkk3/8/2K5/2B5/2B5/8/8/8	16 (Be6)	14 (Kd7)
8/8/k7/kk1B4/kk1B4/k2K4/8/8	18 (Kc3)	14 (Kc3)
8/k7/kk1B4/kk1B4/k2K4/8/8/8	14 (Kc4)	13 (Kc4)
kkkk4/1kkk1K2/2kk4/8/2BB4/8/8/8	20 (Ke7)	16 (Ke7)
kk6/kk1K4/kk6/1kk5/8/1BB5/8/1kk5	20 (Kc6)	15 (Kc6)

Table 8.1: Comparison between Ferguson's KBBK analysis and tablebase findings for several positions, as well as suggested moves.

Table 8.1 shows a comparison of Ferguson's analysis and our KBBK tablebase. It can be seen that, for simpler mates, results and strategies are more or less identical, but manual analysis starts to fall behind retrograde analysis as positions become larger and more complicated. Interestingly, strategies do not differ in a majority of cases (though they differ more when it comes to reacting to illegal moves), but sub-optimal behavior in a small number of situations seems to be enough to slow down checkmate by as much as 33%. The positions considered by Ferguson to be impossible to win with certainty are indeed not found in the tablebase, though the tablebase contains many slightly more restrictive entries that can always be won. These positions actually form the bulk of its nearly eight million entries.

8.5 KBNK

KBNK is probably the most interesting among the four endgames we examined with retrograde analysis. At 17,508,207 entries, it is also the largest tablebase as well as the one with the longest distances to mate: up to 89. In order to win a position in this scenario, it may be necessary to disable the 50 move rule. Of further interest is the fact that it is the only instance of three different pieces collaborating towards checkmate. Moreover, it is the only endgame for which, prior to the present research, it was unknown whether it

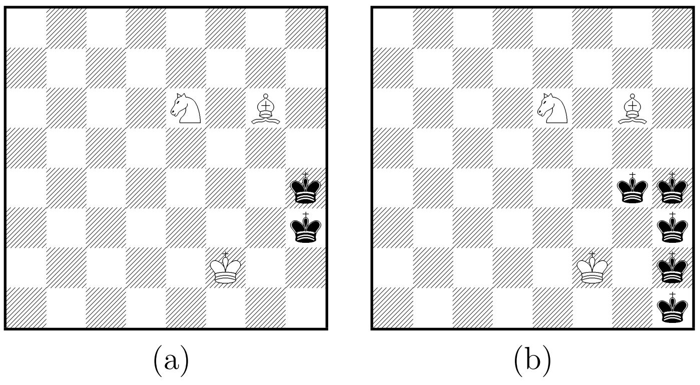
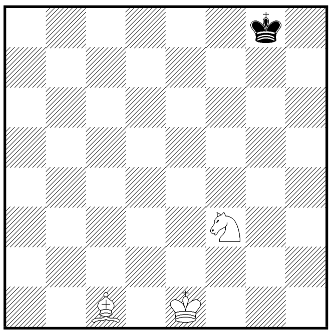


Figure 8.8: (a): a position requiring a randomized strategy (Kg2,Kf3) according to [Ferguson, 1992]; (b): its strategically optimal superset in the tablebase, which can be won in 21 with a pure strategy Kf3.

could be won with a pure strategy in a fixed number of moves. The first claim that such a strategy existed was given in [Isham, 1926] over eighty years ago on a chess variant magazine, *The chess amateur*, by a group of Kriegspiel enthusiasts at the Los Angeles Chess Club. However, they never published the full proof to their statement. Recently, [Beasley, 2005] supported the original claim, intuitively showing that it is likely for White to be able to checkmate, but admitting that the problem was too long and tedious to fully analyze.

On the other hand, Ferguson studied this endgame in [Ferguson, 1992] and concluded that it could be won 100% of the time starting from stable positions, but believed that doing so required some sort of randomized strategy at several key points throughout the algorithm. Figure 8.8, (a) depicts the simplest case, which was thought to require White to play Kg2 with probability θ and Kf3 with probability $1 - \theta$. The problem would then turn into a recursive game, which allows one to calculate an upper bound to the expectation of the distance to mate: 26, in this case.

The tablebase shows that the Los Angeles players were correct in their claim. As highlighted in diagram (b), retrograde analysis shows that diagram (a), and even a more general one, can be won in 21 moves with a pure strategy that always plays Kf3. Unfortunately, the very large number of cases and subcases makes it impossible to list them all; however, the core


(a)

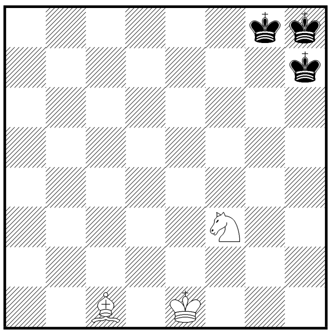
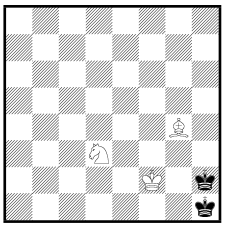
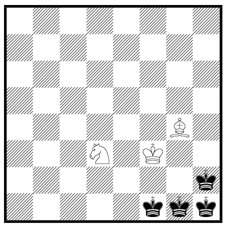

(b)

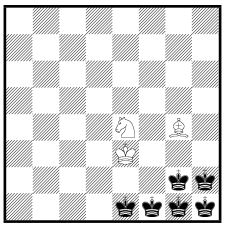
Figure 8.9: (a): Problem given by Shapley (1973) as a mate in 22; (b) its tablebase solution, found as a mate in 21.

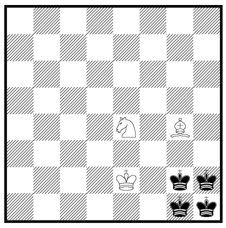
of the optimal strategy for (a) begins with Kf3 Bf7 Kg2 Bg6 Kf3 Nd4 Kf4 Nf5 Bh5. Of these, only Kg2 can be illegal; in this case one should play Nf4 instead, which actually shortens the mate by one move. Other randomized positions are likewise shown to be possible to win with pure strategies, so the whole endgame is. Depending on the starting position, most scenarios can be won in 70-80 moves in the worst case.

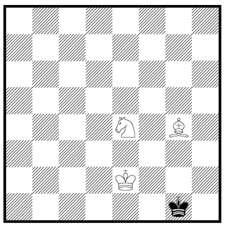
A remarkable (and possibly the most famous) KBNK problem for Kriegspiel was given by L. Shapley in 1973. It is reproduced in Figure 8.9, (a). Originally solved in 22 moves, the tablebase contains a strategy for winning its superset (b) in only 21 moves. That the solution could be expanded to three enemy kings is obvious even without automatic analysis, since White's initial moves trap the black king in the same six-square territory regardless of whether (a) or (b) is used. The optimal strategy, as found by retrograde analysis, starts with Ne5 Bg5 Ke2 Ke3 Ke4 Kd5 Kd6, which is almost identical to the manual solution (Kd6 instead of Ke6, but the two moves are actually equivalent). The problem is with a single 15-move case considered by Ferguson from this point, namely after the next move Ke7 is declared illegal. The stated strategy, a temporary retreat with Kd7, is a wasted move; the (rather surprising) optimal move is Bf6, which leads to a mate in 12. From here, Nd3 is the best follow-up. Roughly speaking, White maneuvers the knight around while trying to push forward the king every other move, which leads to a faster mate if legal.

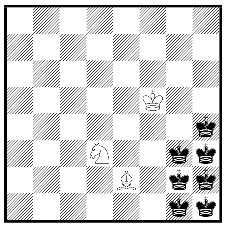

2A. F: 5, T: 5

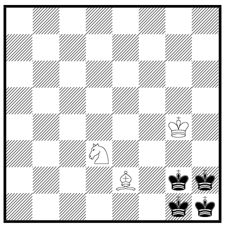

2B. F: 7, T: 7

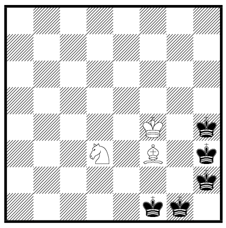

3A. F: 15, T: 14

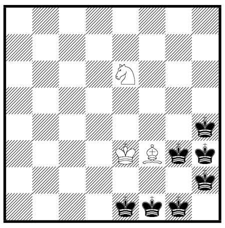

3B. F: 13, T: 13

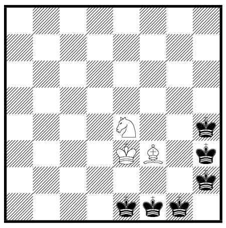

3C. F: 8, T: 8

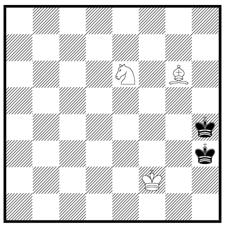

3E. F: 15, T: 13

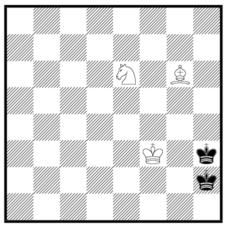

3F. F: 12, T: 11

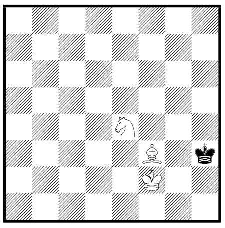

3I. F: 15, T: 10

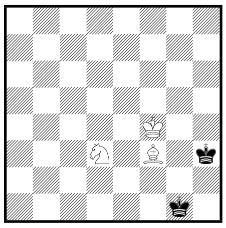

3J. F: 24, T: 23

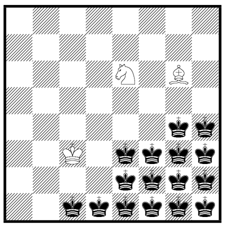

3K. F: 19, T: 18


4A. F: < 26, T: 21


4B. F: 24, T: 18


4C. F: 14, T: 14


4D. F: 10, T: 10


4E. F: < 42, T: 36

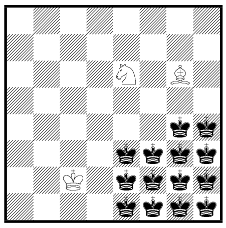
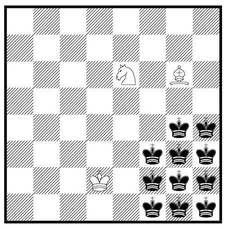
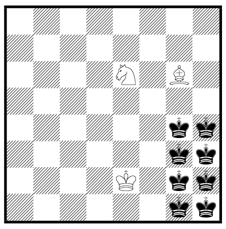
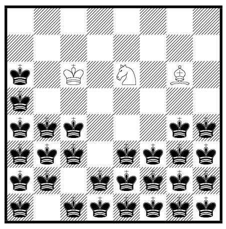

4F. F: < 36, T: 33

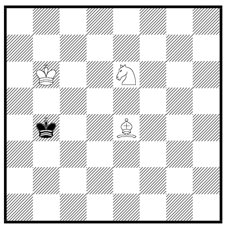
Table 8.2: Comparison of distances to mate between Ferguson's KBNK analysis and tablebase findings using the original diagram identifiers, first part. Use of $< X$ denotes a probabilistic mixed strategy with an estimated upper bound.

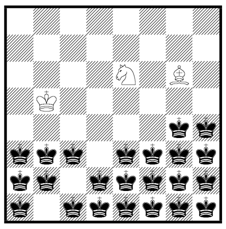
Once again, even one sub-optimal line of play can raise distances to mate considerably. The effect in KBNK is more evident than in KBBK because of the greater complexity of this endgame. A full comparison between the results in [Ferguson, 1992] and tablebase computations is given in Tables 8.2

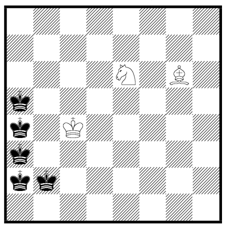

4G. F: < 31, T: 28

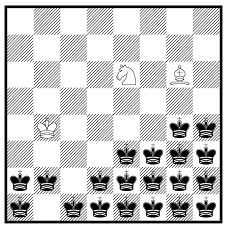

4H. F: < 29, T: 26

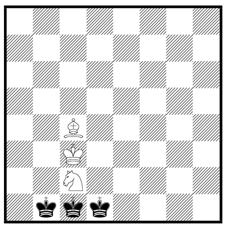

5A. F: < 72, T: 52

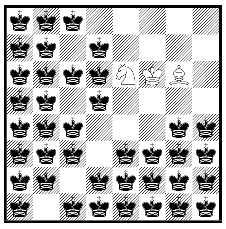

5B. F: < 70, T: 52

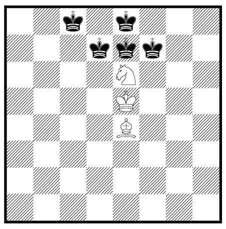

5C. F: < 62, T: 47

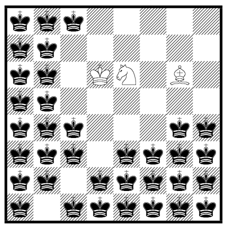

5D. F: < 60, T: 45

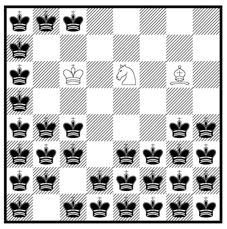

5E. F: < 56, T: 45

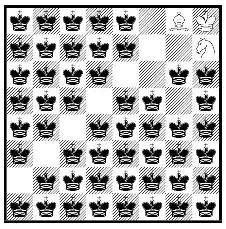

5F. F: < 52, T: 42


6A. F: < 86, T: 77


6B. F: < 72, T: 51


6C. F: < 81, T: 75


6D. F: < 80, T: 75


7A. F: < 95, T: 79

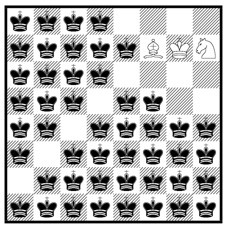

7B. F: < 91, T: 78

Table 8.3: Comparison of Ferguson's KBNK analysis and tablebase findings, continued.

and 8.3. Interestingly, the difference between estimated upper bounds and actual distance to mate according to the tablebase does not grow uniformly with the length of the problem. Some of the largest gaps are found in medium depth entries, whereas there are smaller discrepancies in some later metapositions. Probabilistic strategies that differ less from the tablebase findings are more effective than the others and considerably simplify the game, thus partly compensating for the computer's merciless analysis.

Chapter 9

Conclusions and future developments

9.1 Conclusions

This thesis has approached a single, overarching problem – playing a complex, imperfect information game such as Kriegspiel – from several angles, including heuristic-driven search, statistical sampling and exhaustive game-tree exploration. So far, none of these approaches can conquer the whole game by itself. Indeed, there are situations that all of them fail to address elegantly in the general case, the opening being the most glaring example. In a way, the very first conclusion we can draw is that Kriegspiel is indeed a difficult game for computers to master.

Still, we have reduced the gap between the best computer and the best human, building software that can consistently rank among the top 20 players on the Internet Chess Club. This is even more remarkable in that, unlike some top humans who only play low-risk opponents to protect their rankings, Darkboard accepts challenges from anyone. Moreover, unlike some ancient Kriegspiel bots that tended to play on very short time settings to mask their weakness, Darkboard plays on any time limit except the shortest ones. In other words, humans can challenge our program on their own terms: obviously, Darkboard does not mind ignoring Sun Tzu's millennia-old advice.

All the algorithms described in this work have a practical use within the program. While Darkboard 2.0 is stronger than its predecessor, it still uses 1.0's metaposition search to play the endgame as it proved more effective than Monte Carlo in handling scenarios with the king alone. The tablebase is perfect for playing specific endgames, but it only covers a few of the more frequent ones, and it will not help when victory is not guaranteed.

There are several key points following logically from the present research:

- Information and modeling are an excellent replacement for domain knowledge in Kriegspiel (and presumably, other games of its kind). The ability to model the referee (as an easier proxy for the opponent) with some degree of accuracy makes specific Kriegspiel knowledge not crucial to the program.

- Except in cases where there is very little uncertainty, reasoning on individual "chess" states seems to be less effective than reasoning on a group of states as a whole with some kind of model or approximation. We do not have the computational power or storage capacity to produce a truly meaningful sample of mid-game Kriegspiel states with our current and foreseeable technology.

- Humans are better than computers at Kriegspiel because they can merge states effortlessly and reason on their abstracted vision of the game. Their model is sophisticated enough to give them purpose, which in turn allows them to see ways to make progress through the game. For a human, Kriegspiel has a faster learning curve than chess because there are effectively much fewer states. This has an interesting parallel in the tremendous compression factor of our Kriegspiel tablebases. It is as if the game were telling us that most of its states are redundant, if only we could understand how.

- If computers can replicate the same sense of purpose and progress, we can expect them to outperform humans with their better statistical analysis.

- Once you assume a best defense model for the opponent, Kriegspiel looks much more like chess, as shown in the endgame tablebases. Of course, it is a pointless type of chess if applied to the entire game, as the omniscient player has an almost guaranteed win, but it can be highly informative if even the oracle can be beaten.

9.2 Future developments

Despite the noteworthy improvements, all current programs are still missing something – a spark of intelligence – before they can challenge a human champion successfully over a long series of matches. No doubt, Darkboard 2.0 has not reached its full potential yet; it has not undergone extensive optimization and the referee modeler could be made more accurate, thus correcting some of its tactical blunders. Observing the program in action shows that, while it seems to have purpose in attacking the opponent, it is overly optimistic in its assumptions: often, it thinks it has all the time in the world to organize an attack, and it does not mind jeopardizing some pieces in the process. If curbed manually, such overconfidence quickly turns into fear, which leads the program to maintain position and stop attacking. In Kriegspiel, defense may win the game against a poor opponent, but will generally lose to a good one.

The missing spark to create a champion-level program can come from several places. Better simulations can certainly be of help, as can a series of Monte Carlo Tree Search optimizations already experimented in other games. In Go, MCTS is more and more often combined with game-specific heuristics that help the artificial player in the selection and simulation tasks. Since Monte Carlo methods are weak when they are short on time, these algorithms drive exploration through young nodes when there is little sampling data available on them. An example of such algorithms is the two progressive strategies described in [Chaslot et al., 2008]. 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. There are several other optimizations borrowed from Go that might be useful under imperfect information, such as the all-moves-as-first heuristic.

It is, however, entirely possible that such optimizations alone will not be enough for the computer to achieve champion levels of play (and, given the element of chance in the game, it remains to be seen whether it is even possible for a computer to reach overchampion level). Perhaps this will only happen when planning methods are integrated into a program, adding another layer of complexity. Human Kriegspiel players make most of their decisions basing themselves on a number of plans of varying difficulty and effectiveness.

The main problem here consists of representing these plans, as they are more than just sequences of repeated moves. Human players adapt their plans to the specific game; we once tried to copy human "power moves", but without more sophisticated knowledge, the move turns out to be weak more often than not. Preliminary research on very simple plans (basically targeting a square and attacking it *en force*), whose output was used to provide progressive unpruning in Monte Carlo, yielded small but measurable improvements; however, research in this area is not yet mature enough.

Playing the opening deserves a separate mention. Currently, all versions of Darkboard make use of an opening book taken from the initial moves of the top human players. This occasionally becomes predictable by the best humans, leading to a few very early checkmates. Research is still ongoing on at least two fronts: first, using Monte Carlo sampling to select the best opening from the book in the presence of an opponent model, and secondly, using it to create new openings for Darkboard to use, tailored for the specific opponent. From a simulation standpoint, the opening is the best time to use Monte Carlo, as there is no uncertainty and even textbook MCTS (what we called approach A) may help.

As far as the endgame tablebase is concerned, indexing and compression seem to be the next main problem to be faced. Currently, there is no indexing to the tablebase, forcing the algorithm to look up a large number of entries in response to a query. Also, tablebase are quite large, with only a tiny fraction of their entries being actually useful in a real game. This leads to the problem of compressing a tablebase by removing positions that are unlikely to occur, at the risk of selecting a sub-optimal strategy if those positions actually occur. A separate problem is that of expanding the tablebases to find positions that are won with probability arbitrarily close to 1 through recursive mixed strategies. Such expanded tablebases would no longer be directed acyclic graphs, however.

Appendix A

Official Kriegspiel rules

These are the rules for playing on the Internet Chess Club, where this chess variant has been active since 1996. This ruleset is the closest to becoming a standard for Kriegspiel, being the one enforced at the Computer Olympiads.

Kriegspiel (wild 16) is a chess variant in which you cannot see your opponent's pieces. You can only see your own pieces, and you have to guess where your opponent's pieces are. When you try to make a move, ICC may tell you that your move is illegal, in which case you should make another move instead.

To play Kriegspiel, just match someone for a wild 16 game:

```
match Fred 2 12 kriegspiel
```

The referee makes the following announcements where appropriate:

```
"White's move"
"Black's move"
"Pawn at <square> captured"
"Piece at <square> captured"
"Rank check"
"File check"
"Long-diagonal check"
   (the longer diagonal from the king's point of view)
"Short-diagonal check"
   (e.g. for a king on e1, the short diagonal is e1 to h4)
"Knight check"
"<number> pawn tries"
```

```
(number of legal capturing moves using pawns)
```

When you try an illegal move, you are simply told "Illegal move", whether it is moving into check or moving through an enemy piece. Your opponent is not told anything when you try an illegal move.

Moves must be entered in dumb-computer format, e.g. "e2e4". Input strings such as "nxq" which might be interpreted differently depending on the enemy position are not allowed, with one exception: "px" is allowed, to save you the trouble of trying a dozen possible diagonal pawn moves when you know that precisely one of them is legal. Other acceptable forms include "e2-e4", "o-o", and "f7g8=N". Moves like "Rd3" are currently not accepted (because there are a few cases where they could be context-dependent). Most graphical interfaces generate strings like "e2e4" for you when you make moves with the mouse. (But see the note below about pawn captures under xboard and slics).

Opponent's moves show up in the form "?" or "?xf3". This might break some interfaces.

You cannot observe rated Kriegspiel games. This is to prevent people from logging in with a second account, and seeing all the pieces while they are playing on another account! You *can* observe unrated Kriegspiel games, if you're registered.

In examine mode you can see all the pieces and moves. E.g. if you have examine=1, at the end of the game you'll be able to see your opponents pieces (you may have to "refresh"), review the move history, etc. The illegal moves tried are not recorded.

The clients xboard 3.4, slics 22f, and probably some other interfaces will not allow you to even attempt a diagonal pawn move with the mouse when they can't see a piece to take. The move is not even being transmitted to the server in this case; it's just being rejected by the client. So with those interfaces, you should type in pawn capturing moves, e.g. "px" or "d5c4". Naturally these problems are not the fault of the interface writers! We sprang kriegspiel on them with no warning (sorry guys). Ziics happens to work well as is.

Notes and known problems:

1. Pawn captures must be done by keyboard on some interfaces.

2. Channel 116 is the Kriegspiel channel.
3. Kriegspiel with lag is painful. Even with timestamp, your clock will run while you wait to hear that your move is illegal.
4. If you start a game and your opponent complains that he can't see your pieces, offer to abort, and suggest that he read this file.
5. If you disconnect or get disconnected during a Kriegspiel game, the game is a loss. Kriegspiel games are not adjourned.

Appendix B

A notation for Kriegspiel games

In order to record the transcript of a chess game, there exists a file format specification called PGN (Portable Game Notation). This well-known format was designed to be both easily read by a human and easily parsed by a computer program, and consists of a series of tag-data pairs, some of which are mandatory, as well as a sequence of moves given in standard Algebraic Chess Notation. Actually, there are two formats to PGN, one being the relatively more lax "import" format, which can correctly parse human-created PGN files, and a strict "export" format, which a computer can generate to best adhere to the standard. A more thorough description is given in [`http://en.wikipedia.org/wiki/PGN`] and the full format specification can be found in [Edwards, 1994].

PGN can be used to record chess variants as it supports the "Variant" tag (not to be confused with the "Variation" tag, which denotes a peculiar variation on a given chess opening). While some variants are harder to encode into PGN due to radical rule changes with respect to orthodox chess (because of different piece types, multiple turns, etc.), Kriegspiel is virtually identical to chess as far as the final product is concerned that is, the resulting game is a full-fledged game of chess that can be exported as PGN without any modification. However, this solution is lacking in two aspects: firstly, because we may not have access to the umpire's information, and secondly because PGN is not designed to record umpire messages and illegal move attempts. The latter is the more serious shortcoming, because umpire messages can be reconstructed at runtime but illegal moves are a priceless source of information on a player's style and decision process.

The best solution offered so far is to define an extended PGN notation for Kriegspiel that includes the missing information as comments in the PGN file. This idea originates from the Berkeley University Kriegspiel project [Wolfe]. Surrounded by curly braces, comments are a legal part of the PGN specification, meaning that a compliant piece of software can make use of the additional data, otherwise it will be safely ignored. They are found between two moves in the PGN file, and in the extended notation they contain information on the move they follow. The format is format

{ *umpire-messages : illegal-move-list* },

where both *umpire-messages* and *illegal-move-list* are comma-separated lists. Umpire messages are deterministic and could be easily reconstructed, but their presence is for the convenience of a human reader. Allowed codes include:

- **Px**, where **x** is the number of available pawn tries in the next move (for the opponent to take advantage of). If the Kriegspiel variant does not allow pawn tries, this code will simply never appear, or can be replaced with **Ax** to mean that a player asked "Are there any?" and the umpire gave the answer **x**.

- **Cx**, where **x** is an uppercase letter, means that, following this move, the opponent's King is in check. Legal character codes are **R** (rank check), **F** (file check), **L** (long diagonal check), **S** (short diagonal check) and **N** (knight check). In the event of a double check, two check entries appear in the list rather than two letters following the **C** code.

- **Xa1**, where **a1** represent any square on the chessboard, means that a capture took place on the given square. This is, in most cases, the same square where the current player moved his piece to, the only exception being *en passant* captures.

Moreover, extended PGN games can be filtered to only include the moves of one player, but not both, as if that player was narrating the game from his point of view. Man-made Kriegspiel problems can include more or less information, for the purpose of creating an interesting riddle. The extended format simply instructs to replace unknown facts such as moves and umpire messages with the ?? placeholder.

Another technical, but interesting point which needs to be raised concerns notation. Algebraic notation aims to shrink moves down to a more compact format whenever possible; “Rg4” is used instead of “Rh4-g4” unless the starting square (or just one coordinate of it) is required to disambiguate the move, for example if two rooks are on the same rank. When one can reason with perfect information, disambiguation is a trivial matter; however, the presence of hidden enemy pieces makes moves that are illegal from the umpire’s perspective, potentially legal. Therefore, a shortened move transcript that is perfectly sound for the umpire, might seem ambiguous to the player who acts on imperfect information. The rule of thumb is that moves are interpreted with the knowledge of the party describing the game. For unfiltered games, this is the referee and disambiguation works as usual, but for partial information transcripts the point of view is that of the player, and moves are judged ambiguous depending on his point of view even though the rest of the information is available elsewhere.

Bibliography

S. Akl and M. Newborn. The Principal Continuation and the Killer Heuristic. In *Proc. 32nd ACM National Conference*, pages 466–473, New York, NY, USA, 1977. ACM Press.

L. Allis. A knowledge-based approach to connect-four. The game is solved: White wins. *Master's thesis, Vrije Universiteit*, 1988.

L. Allis. *Searching for solutions in games and artificial intelligence*. PhD thesis, Doktorat, Universiteit Maastricht, 1994.

L. Allis, M. Meulen, and J. van der Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

G. Anderson. *Are There Any?* Straud, 1959.

B.W. Ballard. The *-Minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

B. Banerjee and P. Stone. General game learning using knowledge transfer. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 672–677, Hyderabad, India, 2007.

J. Baxter, A. Tridgell, and L. Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000. URL `citeseer.ist.psu.edu/baxter00learning.html`.

J. Beasley. Bishop endings in Kriegspiel. *Chess Variant*, pages 66–67, 2005.

R. Bellman. On the application of dynamic programing to the determination of optimal play in chess and checkers. *Proceedings of the National Academy of Sciences*, 53(2):244–247, 1965.

E. Berlekamp, J. Conway, and R. Guy. *Winning ways for your mathematical plays.* AK Peters Ltd, 2003.

D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134:201–240, 2002.

D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. Holte, J. Schaeffer, and D. Szafron. Game-tree search with adaptation in stochastic imperfect-information games. *Lecture Notes in Computer Science*, 3846: 21–34, 2006.

J. Blair, D. Mutchler, and C. Liu. Games with imperfect information. In *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning, AAAI Press Technical Report FS93-02, Menlo Park CA*, pages 59–67, 1993.

A. Bolognesi and P. Ciancarini. Computer programming of Kriegspiel endings: the case of KR vs K. In J. van den Herik, H. Iida, and E. Heinz, editors, *Advances in Computer Games 10*, pages 325–342, Graz, Austria, 2003. Kluwer.

A. Bolognesi and P. Ciancarini. Searching over metapositions in Kriegspiel. In J. van den Herik, Y. Björnsson, and N. Netanyahu, editors, *Computer and Games 04*, volume 3846 of *Lecture Notes in Computer Science*, pages 246–261, Ramat-Gan, Israel, 2004. Springer.

A. Bolognesi, P. Ciancarini, and G. Favini. Moving in the dark: progress through uncertainty in Kriegspiel. *Preprint*, 2009.

J. Borsboom, J. Saito, G. Chaslot, and J. Uiterwijk. A comparison of Monte-Carlo methods for Phantom Go. 2007.

B. Bouzy and T. Cazenave. Computer go: an AI oriented survey. *Artificial Intelligence*, 132:39–103, 2001.

M. Bowling, J. Fürnkranz, T. Graepel, and R. Musick. Machine learning and games. *Machine Learning*, 63:211–215, 2006.

J. Boyce. A Kriegspiel endgame. In D. Klarner, editor, *The Mathematical Gardner*, pages 28–36. Prindle, Weber & Smith, 1981.

J. Bryan, P. Gmytrasiewicz, and A. Del Giudice. Particle filtering approximation of Kriegspiel play with opponent modeling. In *AAMAS 2009 Workshop on Multi-agent Sequential Decision-Making in Uncertain Domains*, 2009.

J. Burger. UMPIRE: An automatic kriegspiel referee for a time-shared computer. In *Proc. 22nd ACM National Conference*, pages 187–193, Washington, USA, 1967. Association for Computing Machinery.

R. Callois. *Man, Play and Games.* Free Press, 1961. ISBN 0029052009.

T. Cazenave. A phantom go program. In J. van den Herik, editor, *Advances in Computer Games 11*, pages 120–126, Taipei, Taiwan, 2005. Springer.

W.G. Chase and H.A. Simon. Perception in chess. *Cognitive psychology*, 4 (1):55–81, 1973.

G. Chaslot, M. Winands, J. van der Herik, J. Uiterwijk, and B. Bouzy. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, 4(3):343–352, 2008.

K. Chellapilla and D. Fogel. Evolving neural networks to play checkers without relying on expert knowledge. *IEEE Transactions on Neural Networks*, 10(6):1382–1391, 1999.

M. Chung, M. Buro, and J. Schaeffer. Monte Carlo planning in RTS games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2005.

P. Ciancarini. La scacchiera invisibile: Introduzione al Kriegspiel. `http://www.cs.unibo.it/~cianca/wwwpages/chesssite/kriegspiel/scacchierainvisibile.pdf`, 2004.

P. Ciancarini and G. Favini. Representing Kriegspiel states with metapositions. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2450–2455, Hyderabad, India, 2007a.

P. Ciancarini and G. Favini. A program to play Kriegspiel. *ICGA Journal*, 30:3–24, 2007b.

P. Ciancarini and G. Favini. Monte carlo tree search techniques in the game of Kriegspiel. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 474–479, Pasadena (CA), USA, 2009a.

P. Ciancarini and G. Favini. Solving Kriegspiel endings with brute force: the case of KR vs. K. In *Advances in Computer Games 12*, page to appear, 2009b.

P. Ciancarini and G. Favini. Playing the perfect Kriegspiel endgame. *Preprint*, 2009c.

P. Ciancarini and G. Favini. Monte Carlo tree search in Kriegspiel. *Preprint*, 2009d.

P. Ciancarini, F. DallaLibera, and F. Maran. Decision making under uncertainty: a rational approach to Kriegspiel. In J. van den Herik and J. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 277–298. Univ. of Rulimburg, 1997.

M. Clarke. A quantitative study of king and pawn against king. *Advances in Computer Chess*, 1:108–118, 1977.

R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. *Lecture Notes in Computer Science*, 4630:72–83, 2007.

A. Del Giudice, P. Gmytrasiewicz, and J. Bryan. Towards strategic kriegspiel play with opponent modeling. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 1265–1266, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-7-8.

D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 31–36, Chicago, USA, 1994. AAAI Press.

D. Edwards and T. Hart. The alpha-beta heuristic. *HIT Artificial Intelligence Memo*, 30, 1963.

S. Edwards. Portable Game Notation Specification and Implementation Guide, 1994.

J. Feinstein. Amenor wins World 6x6 Championships. *British Othello Federation Newsletter, July*, pages 6–9, 1993.

T. Ferguson. Mate with bishop and knight in Kriegspiel. *Theoretical Computer Science*, 96:389–403, 1992.

T. Ferguson. Mate with two bishops in Kriegspiel. Technical report, UCLA, 1995.

T. Ferguson. On a Kriegspiel problem of Lloyd Shapley, 2009.

G. Ferrer and W. Martin. Using genetic programming to evolve board evaluation functions. In *1995 IEEE Conference on Evolutionary Computation*, pages 747–752, Perth, Australia, 1995. IEEE Press.

R. Fikes and N. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

D. Fogel, T. Hays, S. Hahn, and J. Quon. A self-learning evolutionary chess program. In *Proceedings of 2004 Congress on Evolutionary Computation*, pages 1427–1432, Piscataway, NJ, 2004. IEEE Press.

I. Frank and D. Basin. Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100:87–123, 1998.

I. Frank and D. Basin. A theoretical and empirical investigation of search in imperfect information games. *Theoretical Computer Science*, 252:217–256, 2001.

R. Gasser. Solving Nine Mens Morris. *Games of No Chance*, pages 101–114, 1996.

M.L. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, volume 16, pages 584–593. Morgan Kaufmann, San Francisco, USA, 1999.

F. Giunchiglia and P. Traverso. Planning as model checking. In *Proceedings of the 5th European Conference on Planning*, pages 1–19, Durham, UK, 1999.

F. Gobet, P. Lane, S. Croker, P. Cheng, G. Jones, I. Oliver, and J. Pine. Chunking mechanisms in human learning. *Trends in Cognitive Sciences*, 5 (6):236–243, 2001.

T. Hauk, M. Buro, and J. Schaeffer. Rediscovering *-Minimax search. *Lecture Notes in Computer Science*, 3846:35–50, 2006.

C. Hoekstra. Adaptive artificially intelligent agents in video games: A survey, 2006.

`http://en.wikipedia.org/wiki/PGN`. Pgn information on wikipedia.

H. Iida, J. Yoshimura, K. Morita, and J. Uiterwijk. Retrograde analysis of the KGK endgame in shogi: Its implications for ancient Heian shogi. *Lecture notes in computer science*, pages 318–336, 1998.

H. Isham. More Kriegspiel problems. *The Chess Amateur*, page 44, 1926.

W. Jamroga. A defense model for games with incomplete information. In *Proceedings of the Joint German/Austrian Conference on AI: Advances in Artificial Intelligence*, pages 260–274, Vienna, Austria, 2001. Springer-Verlag.

A. Jansen, D. Dowe, and G. Farr. Inductive inference of chess player strategy. In *Pacific Rim International Conference on Artificial Intelligence*, pages 61–71, Melbourne, Australia, 2000.

L. Kaebling, M. Littman, and A. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

H. Kaindl. Searching to variable depth in computer chess. In *Int. Joint Conf. on Artificial Intelligence (IJCAI83)*, pages 760–762, Karlsruhe, West Germany, 1983.

M. Kalos and P. Whitlock. *Monte Carlo methods.* Wiley-VCH, 2008. ISBN 352740760X.

L. Karlsson. Conditional progressive planning under uncertainty. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 431–438, Seattle, USA, 2001. Morgan Kaufmann.

S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. *Lecture Notes in Computer Science*, 4212:282–293, 2006.

R. Korf. Depth-first iterative-deepening. *Artificial intelligence*, 27(1):97–109, 1987.

J. Koza. *Genetic Programming: On the programming of Computers by Means of Natural Selection.* MIT Press, 1992.

H. Kuhn. Extensive games and the problem of information. pages 193–216, 1953.

N. Lassabe, S. Sanchez, H. Luga, and Y. Duthen. Genetically programmed strategies for chess endgame. In *Proceedings of the 8th annual conference on genetic and evolutionary computation*, pages 831–838, Seattle, USA, 2006. ACM Press.

M. Leoncini and R. Magari. Manuale di scacchi eterodossi. *Tipografia Senese, Siena*, 1980.

D. Li. *Chess Detective: Kriegspiel Strategies.* Premier Publishing, 1995. ISBN 0963785249.

R. Magari. Scacchi e probabilità. In *Atti del Convegno: Matematica e scacchi. L'uso del Gioco degli Scacchi nella didattica della Matematica*, pages 59–66, Forlì, Italy, 1992.

N. Metropolis and S. Ulam. The Monte Carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

D. Michie. Game-playing and game-learning automata. *Advances in programming and non-numerical computation*, page 183, 1966.

D.E. Moriarty and R. Miikkulainen. Discovering complex Othello strategies through evolutionary neural networks. *Connection Science*, 7(3):195–210, 1995.

E. Nalimov, G. Haworth, and E. Heinz. Space-efficient indexing of chess endgame tables. *ICGA Journal*, 23(3):148–162, 2000.

M. Nance, A. Vogel, and E. Amir. Reasoning about partially observed actions. In *Proceedings of 21st National Conference on Artificial Intelligence (AAAI '06)*, pages 888–893, Boston, USA, 2006.

N. Onder and M. Pollack. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 577–584, Orlando, USA, 1999.

A. Parker, D. Nau, and VS. Subrahmanian. Game-tree search with combinatorially large belief states. In *Int. Joint Conf. on Artificial Intelligence (IJCAI05)*, pages 254–259, Edinburgh, Scotland, 2005.

A. Parker, D. Nau, and VS Subrahmanian. Overconfidence or paranoia? search in imperfect-information games. In *Proc. 21st National Conference on AI*, pages 1045–1050, Boston, USA, 2006.

D. Parlett. *The Oxford History of Board Games.* Oxford University Press, 1999. ISBN 0192129988.

P. Perla. *The Art of Wargaming.* Naval Institute Press, 1990. ISBN 0870210505.

A. Plaat, J. Schaeffer, W. Pijls, and A. De Bruin. A new paradigm for minimax search. Technical Report CS-94-18, Dept. of Computing Science, Univ. of Alberta, 1994.

A. Plaat, J. Shaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artif. Intell.*, 87(1-2):255–293, 1996. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(95)00126-3.

M. Pollack. The uses of plans. *Artificial Intelligence*, 57:43–68, 1992.

M. Ponsen, J. Ramon, T. Croonenborghs, K. Driessens, and K. Tuyls. Bayes-relational learning of opponent models from incomplete information in no-limit poker. In *Twenty-third Conference of the Association for the Advancement of Artificial Intelligence (AAAI-08)*, pages 1485–1487, 2008.

A. Reinefeld. Spielbaum Suchverfahren. Volume Informatik-Fachberichte 200, 1989.

J. Romein and H. Bal. Solving the game of awari using parallel retrograde analysis. *IEEE computer*, 36(10):26–33, 2003.

S. Russell and J. Wolfe. Efficient belief-state AND-OR search, with application to Kriegspiel. In *Int. Joint Conf. on Artificial Intelligence (IJCAI05)*, pages 278–285, Edinburgh, Scotland, 2005.

E. Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier Publishing Company, New York, 1977.

A. Sadikov and I. Bratko. Learning long-term chess strategies from databases. *Machine Learning*, 63:329–340, 2006.

M. Sakuta. *Deterministic Solving of Problems with Uncertainty.* PhD thesis, Shizuoka University, Japan, 2001.

M. Schadd, M. Winands, M. Bergsma, J. Uiterwijk, and H. van den Herik. Best play in Fanorona leads to draw. *New Mathematics and Natural Computation*, 4(3):369–387, 2008.

M. Schadd, M. Winands, and J. Uiterwijk. CHANCEPROBCUT: Forward pruning in chance nodes, 2009.

J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.

J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518, 2007.

N. Schraudolf, N. Dayan, and T. Sejnowski. Temporal difference learning of a position evaluation in the game of go. In Tesauro Cowan and Alspector, editors, *Advances in Neural Information Processing Systems, volume 6*, pages 817–824, Denver, USA, 1994. Morgan Kaufmann, San Francisco.

C.E. Shannon. Programming a computer for playing chess. *Philosophical Magazine (Series 7)*, 41(314):256–275, 1950.

L. Shapley. The Invisible Chessboard, 1987.

B. Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 134(1-2):241–275, 2002.

S. Smith and D. Nau. *Strategic planning for imperfect-information games*. University of Maryland, 1993. URL `citeseer.ist.psu.edu/smith93strategic.html`.

S. Smith, D. Nau, and T. Throop. Computer bridge - a big win for AI planning. *AI Magazine*, 19(2):93–106, 1998.

F. Southey, M. Bowling, B. Larson, C. Piccione, N. Burch, D. Billings, and C. Rayner. Bayes bluff: Opponent modelling in poker. In *Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 550–558, 2005.

J. Stankiewicz. Opponent modeling in Stratego, 2009.

D. Stern, R. Herbrich, and T. Graepel. Bayesian pattern ranking for move prediction in the game of Go. In *Proceedings of the 23rd international conference on Machine learning*, pages 880–887. ACM, 2006.

L. Stiller. Some Results from a Massively Parallel Retrograde Analysis. *ICCA Journal*, 14(3):129–134, 1991.

T. Ströhlein. *Untersuchungen über kombinatorische Spiele*. PhD thesis, Dissertation, Fakultät für Allgemeine Wissenschaften der Technischen Hochschule München, 1970.

R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

G. Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

G. Tesauro and G.R. Galperin. On-line policy improvement using Monte Carlo search. In *Proc. of NIPS*, pages 1068–1074, 1996.

K. Thompson. Retrograde analysis of certain endgames. *ICCA Journal*, 9 (3):131–139, 1986.

J. Tromp and G. Farneback. Combinatorics of go. *Lecture Notes in Computer Science*, 4630:84, 2007.

H. van den Herik, J. Uiterwijk, and J. van Rijswijck. Games solved: now and in the future. *Artificial Intelligence*, 134(1–2):277–311, 2002.

C. Van Wyk and D. Knuth. A programming and problem-solving seminar. Technical Report STAN-CS-79-707, Computer Science Department, School of Humanities and Sciences, Stanford University, 1979.

J. von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928.

J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944. ISBN 978-0-691-13061-3.

S. Walczak. Knowledge-Based Search in Competitive Domains. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):734–743, 2003.

S. Walczak and D. Dankel II. Acquiring tactical and strategic knowledge with a generalized method for chunking of game pieces. *International Journal of Intelligent Systems*, 8:249–270, 1993.

C.S. Wetherell, T. Buckholtz, and K.S. Booth. A Director for Kriegspiel, A Variant of Chess. *The Computer Journal*, 15(1):66–70, 1972.

C.S. Wetherell, T.Buckholtz, and K.S. Booth. A Program to Referee Kriegspiel and Chess. *The Computer Journal*, 18, 1975.

D. Wilkins. Using patterns and plans in chess. *Artificial Intelligence*, 14: 165–203, 1980.

S. Willmott, J. Richardson, A. Bundy, and J. Levine. Adversarial planning in complex domains. In *Computers and Games*, pages 93–112, Tsukuba, Japan, 1998. URL `citeseer.ist.psu.edu/article/willmott98adversarial.html`.

S. Willmott, J. Richardson, A. Bundy, and J. Levine. Applying adversarial planning techniques to Go. *Theoretical Computer Science*, 252(1–2):45–82, 2001. URL `citeseer.ist.psu.edu/willmott01applying.html`.

J. Wolfe. Kriegspiel PGN notation.
`http://www.cs.berkeley.edu/~jawolfe/kriegspiel/notation.html`.

E. Zermelo. Uber eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In *Proceedings of the Fifth International Congress of Mathematicians*, volume 2, pages 501–4, Cambridge, UK, 1913.