Algorithms for Selective Search

Bouke van der Spoel

March 8, 2007

Contents

Chapter 1

Introduction

1.1 History

The field of AI holds tremendous promise: if it succeeds in simulating human thought, all jobs could be automated instantly. It was this promise of automation that lured the first researchers to the field of computer chess. Shannon [25] envisioned that the design of a good computer chess player would "act as a wedge in attacking other problems of a similar nature and a greater significance." Some of those other problems were machine translation, logical deduction and telephone routing.

Among reasons to choose for chess, Shannon notes that "chess is generally believed to require 'thinking' for succesful play; a solution of this problem will force us either to admit the possibility of mechanized thinking or to further restrict our concept of 'thinking'." Today, it seems the second option has come true: A chess computer has defeated the human world champion, but it is still plausibly possible to deny that that particular computer could think. Anyone defending the view that what the computer does is actually 'thinking', would have to admit that it is a very limited kind of thought, almost useless outside the realm of two-player games.

Furthermore, everyone agrees that computer thought differs fundamentally from human thought. The only similarity is that both consider possible moves and evaluate their consequences. But where humans consider a number of moves in the range of several hundreds, the computer considers many millions or even billions of moves. Where humans consider only a few initial moves, computers always consider all initial moves. Also, a human evaluation of a position can sometimes reach a conclusion that would require a search of billions of positions for a computer.

Nowhere can the failure of computer chess be seen clearer than with the game of Go. Although Go is a zero-sum perfect information game, like chess, chess techniques applied to Go have only resulted in an amateur strength player. So computer chess tells us little about human thought and it does not even

generalize to a very similar game.

The causes of this failure can readily be seen from a historical perspective. In the seventies, several championship cycles started, pitting different programs against each other. Accomplishment of a program was measured by its ranking, and programmers quickly found out that a focus on chess-specific tricks and efficient implementation would help their program more than fundamental research on reasoning. In the eighties this tendency worsened. Computers had become cheap enough for consumers to buy and strong programs had a direct marketing benefit over their weaker siblings. Using chess as a vehicle to study human thought virtually disappeared through this period. Afterwards, programs continued to grow stronger, but little knowledge was gained in the proces. In fact, the workings of the best known chess computer, Deep Blue, is shrouded in a veil of secrecy. After playing just one match against the human world champion (and winning it), the project was stopped and the machine was dismantled and put on display.

The problem does not lie with the game of chess per se. Rather the excessive attention to playing strength smothered other scientifically more interesting approaches. The game in itself is still interesting, for the following reasons:

- A large percentage of the population knows the rules, so they don't have to be explained.
- The game has a considerable amount of expert players. This makes it easier to elicit expert knowledge and to compare playing strengths.
- There is a relatively large body of psychological research on human chess thought.
- It is still the most studied game in science, with a very large library of papers on computer chess.

The large difference between number of positions searched strikes us as the biggest difference between human and computer. The difference between accuracy of evaluation is also interesting, but this is of course related to the previous issue: with less positions to be evaluated, more time can be invested in the evaluation itself.

Thus, the question we want to answer is whether viable selective search algorithms can be created. We intend to answer this question by demonstration: by creating and evaluating techniques that can play the same quality chess with fewer move evaluations as other methods.

This question has not been answered yet. In the literature, papers on selective search are quite sparse. In most papers an isolated algorithm is proposed and its effectiveness is evaluated, without much influence from other papers. After publication, most authors move on to other research areas. These are probably some of the reasons that no satisfactory solutions have been found thus far. So, at the moment, in general the dominant non-selective algorithm (alpha-beta) is still unchallenged [19].

1.2 Outline

We start in Chapter 2 with an overview of human chess thinking, which proves that selective search can be highly effective. After that we will give an overview of current chess techniques and current attempts at selective search algorithms, in Chapter 3. In this chapter, most of our attention will be given to the Bayesian search algorithm, because this is the algorithm we intend to improve. Chapter 4 consists of the experiments carried out with the improved algorithm, with a description of the testing problems and comparisons with alpha-beta. Some work is also done on improving synthetic testing models. We finish with conclusions and future work in Chapter 5.

1.3 Relevance to CAI

Cognitive artificial intelligence has, in our view, two research parts. One part is the direct study of human performance, either by cognitive psychology or neuropsychology. These methods give knowledge into a particular form of intelligence, the human form. Another part is the study of performance of algorithms for complex tasks, which are sometimes considered to require 'intelligence' to solve. This approach can be seen as studying intelligence apart from the human form.

Both parts augment each other, as human ways of solving problems can often be implemented in computers. On the other hand, knowledge about algorithms for complex tasks yield knowledge about the underlying problem and the difficulties that can arise in solving it. Such knowledge can help define bounds on the methods possibly used by humans to solve the problem, helping the research for understanding the human mind.

This thesis definitely falls in the second category. It is concerned neither with questions about the nature of human selective search nor with simulating the current models that have been made of it. It is concerned with the problem of selective search in itself and the analysis of the difficulties that arise in it. As such, this thesis can be labelled as 'Advanced Informatics', but inspired by human performance.

Chapter 2

Chess psychology

2.1 Introduction

In order to understand the human and computer chess approach, both must be studied. In this chapter we summarize the current understanding in human chess thinking. Before any theorizing can be attempted, data is needed. The main methods of gathering data in this area are introspective reports of thinking chessplayers, and chessrelated memory tasks. As the scientific understanding of human thought processes is still limited (except perhaps for basic visual information processing), theories in this particular field will most likely be inaccurate. Therefore the emphasis will be more on the data in this chapter.

Introspective reports were the main source of data for de Groot [10], and present some interesting results. These reports were obtained by instructing various chessplayers to think 'aloud' during the analysis of a newly presented chess position. The subjects were six contemporary grandmasters, four masters, two lady players, five experts and five class A through C players. The grandmasters were world-class players, with former world champions Euwe and Alekhine among them. The total number of reports gathered was 46. It is often said that a picture says more than a thousand words, and the same can be said for the reports obtained like this. Therefore we have put one such report from de Groot in Appendix A.

2.2 Statistics from reports

A conspicuous characteristic of the reports is that players tend to return to the starting position many times, and start thinking from there. These returns act as a natural way to divide the thinking process into episodes, and a lot of analysis is aimed at these episodes. One variable obtained from these episodes is the sequence of starting moves in each episode. For example, de Groot's subject C2's thought consisted of 9 episodes, and the first moves of each episode were 1. Nxd5; 1. Nxd5; 1. Nxd5; 1.h4; 1. Rc2; 1. Nxd5; 1.h4; 1.Bh6; 1.Bh6. The

move 1.Bh6 was played.

As can be seen from this data, not every episode starts with a new move: the move 1.Nxd5 is considered three consecutive times. De Groot calls this phenomenon 'immediate reinvestigations', and it occurs an average of 2.4 times in all the reports. The number of first moves that are unique is 4.5 on average.

Sometimes an episode starts with a move that was investigated before, although not immediately before. Named 'non-immediate reinvestigations', it happens on average 1.9 times per subject, but with considerable variance between subjects and positions (e.g. position C had an average of 3.8). De Groot goes to some length to emphasize this "is restricted *not* to certain persons who might have the habit of 'hesitating' and 'going back and forth' from one solving proposition to the other, but rather to situations where the subject - any subject - finds it difficult to come to a decision."[his italics]. Among reasons for non-immediate reinvestigation, de Groot mentions:

- Subject has detected an error in his previous calculations.
- Subject's general expectations have dropped, he is forced to get back and to reconsider other lines.
- Subject may be inspired by some new idea for strengthening or reinforcing the old plan.

In all those cases, it is new information that prompts a subject to reinvestigate.

Another statistic calculated from the reports is the number of unique positions reached when all variations mentioned in the report are played through. De Groot found that this number did not change very much through skill levels, even though skill level was strongly correlated to decision quality. The conclusion was that differences in skill are not due to search thoroughness but rather to evaluation accuracy and/or better focus of search effort, at least in the positions used.

2.3 Planning

The moves that appear in the reports are not random. Most of the times, they are conceived with a certain goal in mind. This goal is acquired in the first stage of thought. The verbal report from this stage is structurally different from the later part, and takes about 20%-25% of the total time. This is more lengthy than in normal games, because the positions are totally new to the subjects. How exactly these goals are acquired is unknown, but it seems clear that better players have better goals than worse players. De Groot notes: 'G5 has a more nearly complete grasp of the objective problems of position B1 [after 10 seconds exposure], than do subjects C2, C5, W2 and even M4 after an entire thought process of ten minutes or more!', where G5 is a grandmaster and the others are not.

The goals that are formulated at the start of the thought process are not set in stone. For instance, in one position G5 at first considers an attack on

the enemy king the best option. When that does not give the desirable results, he considers a new goal, namely blockading enemy pawns, and spends half the analysis with this goal. In the end, this does not yield desirable results either, so he returns to the original plan.

Goals can differ greatly in their concreteness, both in what they hope to achieve and in the means for achieving it. In the protocol in Appendix A, the subject says 'Now let's look in some more detail at the possibilities for exchange'. The goal is quite unclear, it is more a 'let's see what happens' kind of attitude. The means are quite clear though, all exchanging moves. Other times the goal is crystalclear, but the means are not that clear. Quotes from subject M5 in a position with mating threats: 'It must be possible to checkmate the Black King' (singular goal), and 'Lots of possibilities will have to be calculated' (multiple means).

When the means of achieving a goal are unclear, more calculation needs to be carried out to see wether the goal is achievable. Indeed, in the last example, the position with mating threats, the subject was searching for a mate almost his entire analysis. In the end, he did not find the mate (it was possible though), and opted for a quiet move. A natural question to ask is: What is the relative importance between raw search ability and accurate goal-finding for chess skill? De Groot did not find any difference in raw search ability between skill levels, but he did find large differences in the accuracy of the goal. Therefore he naturally attributed more importance to recognition of the correct goals.

This hypothesis was later elaborated into the recognition theory of chess skill. It roughly claims that "what distinguishes chess mastery ... is that masters frequently hit upon good moves before analyzing the consequences of the various alternatives. Rapid move selection must therefore be based on pattern recognition rather than on forward search"[16]. There is quite some evidence in favor of this theory: first, de Groot could not find any macroscopic differences in search behaviour between experts and novices. Second, strict time limits, which hinder deep search mostly, do not impair playing strength much. Gobet and Simon [13] show that Kasparov, the world champion at the time, lost only a marginal amount of ability when playing against 4 to 8 players simultaneously. Third, masters did see solutions of combinatory positions significantly more often than novices when only 10 seconds thinking time was allowed, which is not enough to do any search of consequence.

2.4 Chunking

A more specific version of the theory is given by Chase and Simon [8]. It makes use of the fact that strong chess players appear to store chess positions in chunks, rather than piece by piece. Using chunks to improve memory is well-known in cognitive psychology, but here the chunks are used in another fashion as well. Associated with each chunk is a set of plausible moves, it is theorized. Thus, the set of chunks acts as some sort of production system, with a chess position activating particular chunks, and the chunks in turn activating

particular candidate moves. The extensiveness and quality of the chunks present in long term memory is the deciding factor in chess skill, according to this theory.

Holding discredits this theory by noting that observed chunks in memory tasks do not appear to correspond to important chess relationships or good moves. While this is a valid point, it can only be aimed at a simple version of the theory, a version where these relationships are represented by the chunks in isolation. But pieces can be member of different chunks, or, said differently, chunks can overlap. Under the method used to obtain chess chunks this phenomenon was impossible to detect. Also, sometimes a piece was contained in a 'chunk' of its own (a degenerate case). It seems impossible to derive good moves from just the position of one piece on the board. Therefore, the chunks must interact somehow.

How chunks interact to generate good move candidates, is similar to the problem of how chess pieces interact to generate good move candidates. So it appears the recognition theory explains nothing. This is not necessarily the case, however. The process of chunking is essentially a proces of abstraction. This process may be several layers deep, with chunking as the first layer. The higher layers of abstraction may better reflect the problems of a position, and suggest plans accordingly. This could be translated back down into actual moves. Plausible as it may sound, such a theory has no experimental evidence for it yet, and it seems very hard to get such evidence in the future.

Even though there is much unclear about the nature of chunks, Chase and Simon give an estimate of their number. On the basis of a computer simulation, they estimate that between 10.000 and 100.000 chunks are needed. This figure is repeated without discussion in a lot of publications. However, a number of assumptions are implicitly made in this estimation. First, in the simulation the same configuration at a different location constituted different chunks. For example, two pawns next to each other can be present on 42 positions on the board, and all 42 of these could be different chunks. Also, black and white were considered to have different chunks. Eliminating these two sources of redundance could easily reduce the number to 2.500 or less, as Holding notes. However, if larger numbers of pieces are chunked together by chessplayers, the number of possible chunks could skyrocket again.

So far, chunks have been modeled as a small set of pieces on particular locations. Recently, a totally different conception of chunks has been proposed by Hyötyniemi and Saariluoma [18]. Their inspiration comes from connectionist models and the possibilities these models have for representing knowledge in a different way. They represent the chessboard with 768 bits, one for every possible piece-location combination. In the previous model, a chunk is such a position with only a few pieces present, but here it is a fuzzy representation where any number of pieces can be (partially) present. Because of this fuzzyness chunks can be partially present as well. They present an example where their model has similar performance to humans, and claim that their model could more naturally explain other results as well. Although their investigations are far from complete, their model presents yet another point of view on the chunking debate.

Although the means are unclear, it is clear that better players are better planners. Their plans are more to the point, and this ability seems to contribute much to their skill. The other factors, like search ability and evaluation ability, will be considered next.

2.5 Search

All this attention on the way chess players recognize good moves has occluded another part of the equation, search. Because de Groot did not find differences in the search behaviour between skilled and unskilled players, it was assumed there were none for a long time. However, not finding something does not mean it is not there. Holding notes that the position used in de Groot's research does not require deep search to be solved, and indeed de Groot himself provides 'thought pathway' to a solution to the position that contains only 17 steps.

That a difference in searching ability does exist, was shown by Campitelh and Gobet[7]. They maintained that the position used by de Groot was too simple for his subjects and did not require much search to reach a conclusion. They gathered reports on a more difficult position and noted their subjects that there was a unique solution to the position. Thus motivated, the amount of visited positions (in thought) was much larger than in any previous study, but also highly correlated with skill. The only weakness of the study is that only 4 subjects were tested, but it seems reasonable to conclude that stronger players can search deeper if it is needed.

Although this indicates that better players can search deeper, how much this ability contributes to skill is not known. Gobet does not provide an analysis of this question in the first mentioned paper, but some clues can be found in another study by Gobet[12]. This study is a partial replication of de Groots original study. Due to the bigger sample used, Gobet finds differences between skill levels in more variables, but he notes that "The average values obtained ... do not diverge significantly from de Groot's sample."

The interesting part of the study in this context is the application of statistics. From the reports, a lot of variables were collected, mostly the same variables de Groot collected. For each variable, its power in predicting the quality of the chosen move was calculated. Three such variables (time taken, average depth of search and maximal number of reinvestigations) were found to be significant, and taken together they could account for 35.1% of the variance in quality of the choice of move. This was more than the Elo rating, the generally accepted indicator of chess skill, which accounted for 29.2%. When the three variables were partialled out of the result, the Elo rating still accounted for 17.6% of the variance. Gobet's conclusion is "that search alone does not account for the quality of the move chosen, and that other factors, probably including pattern recognition, play an important role."

2.6 Evaluation

When a goal has been chosen, and a search is being carried out, the last positions reached in the search need to be evaluated. Before we can look in more detail at how humans evaluate positions, the phenomenon of 'evaluation' itself needs to be studied. For this discussion, we will take a more computer-oriented approach, because its more mathematical nature lends itself better for analysis.

The earliest solution to the evaluation problem is due to Shannon [25], who proposed to associate a number to each position. The position with the highest number would be the best, from white's perspective. This idea cannot be entirely attributed to Shannon, as common chess lore assigns numbers to the various pieces which denote their value. However, Shannon formalized the idea and mentioned the possibility of adding many other features of a chess position, each feature having a small decimal value, where 1 is the value of a pawn. There are many possible features, but the following have been used extensively:

- Mobility
- Center control
- King's safety
- Double pawns
- Backward pawns
- Isolated pawns
- Pair of bishops

Many of these features are taken directly from human experience; two of them are even present in the verbal report in Appendix A, isolated pawns and pair of bishops. With so many features, most positions have different evaluations so it becomes possible to choose between them.

With this kind of evaluation, we have created a kind of a definitional problem: chess positions have only 3 definite game-theoretic outcomes. So what do these evaluations mean then? It is surprising that no previous authors have explored this question. Most of them just say the evaluation is an indication of 'quality' of a position, without further explication of the term. Van den Herik [29] even notes there is a definitional problem with the evaluation of the KBNK(King, Bishop and Knight versus King alone) ending (it is always won), but leaves the issue at that.

One possible answer to the question goes as follows. Although each chess position has a definite outcome, the player does not necessarily know what it is. The evaluation is some kind of estimate of this outcome. Therefore, the evaluation should correlate with the probability that the position is actually won, lost or drawn. This means that among the highly evaluated positions, only a few are actually drawn or even lost, and vice versa for the lowly evaluated positions. This option can be formulated mathematically:

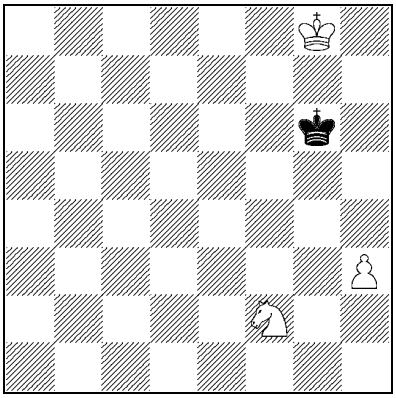

Figure 2.1: A position with two possible winpaths.

$$\sum_{pos \in positions} eval_{pos} \cdot outcome_{pos} > 0$$

where *eval* is an evaluation function that is suitably normalized to a $[-1, 1]$ range, and *outcome* gives the gametheoretic outcome of the position: 1 for a win, 0 for a draw, -1 for a loss.

Another option is that the evaluation is an estimate of the distance to win. This option is mainly useful in endings, where one side may have difficulties in making progress. It can also occur in the midgame, when a player discards a move not because it is bad, but because it does not take him any closer to his goal. In this case both the distant, foreseen position and the current position may be won, but both are still estimated to require the same number of moves for winning. Going to the foreseen position would not bring the player closer to his goal, so the move is discarded.

Still another option is that the evaluation is an estimate of the difficulty of reaching a desired outcome against the current opponent. An example can clarify this proposal: if someone is playing against a much stronger opponent, he may try to keep the position as quiet as possible, with as few tactical possibilities for both players. The reasoning is that the stronger player can better take advantage of them, so if the player wants to reach a draw, this is easier in a quiet position with a small advantage for the opponent, than in a volatile position that is about equal. The stronger opponent may follow the opposite line of reasoning, seeking complications that he would consider bad if he was playing against an equal opponent.

Here follow some examples where different evaluation meanings lead to different results.

In position 2.1, a relatively simple position, several grandmasters gave radically different continuations for white. It is due to van den Herik. The main

subgoal in this position is to move the white king to the white pawn; black will try to prevent this. Most grandmasters proposed 1. Kf8, which is followed by 1. ... Kf6 2. Ke8 Ke6 3. Kd8 Kd6 4. Kc8 Kc6 5. Kb8 Kd6. At the last move, black may not block white at b6 because the white pawn can then walk to promotion unimpeded. After this, white can easily move to his pawn. Another grandmaster proposed 1. Ng4 Kg5 2. Kg7, which will lead to results quicker. Aside from difference in speed, there are other, more subtle, differences in the two approaches. If white plays 1.Ng4, his pawn becomes unprotected. If the pawn is taken it is a draw, so white has to take care to protect it again when the black king attacks it. Though it may seem only a tiny worry, things like these can and have been forgotten if white is in time-trouble. Also, this option requires a bit more computation than simply moving the king around. So, summarizing, both 1.Kf8 and 1.Ng4 are winning strategies, but 1.Kf8 is a little safer, and 1.Ng4 a little shorter.

Another example is the theoretical ending of KBNK. If the stronger player does not give away pieces, it is always won. However, when the stronger player always plays the first move from a list of moves that lead to a won position, and the opponent makes the move that takes longest under optimal play, it is quite possible that the stronger player never wins. Therefore, in this case knowledge of the gametheoretical outcome is not necessarily enough to win.

Sometimes a player chooses to ignore a move that leads to a position the player knows is won. A clear example is when the resulting position is the KNNKp (king and two knights versus king and pawn) ending (see for instance [17]). There are some easy criteria for when such a position is won, but the winning itself can be extremely difficult. As a result, a player can know when such a position is won, but also know that he probably is not able to do so. A position where it is not clear if it is game-theoretically won, but where the player knows how to play, is preferable in this case.

Another example where the estimate of game-theoretical win chances is not the most important feature of a position, is when a player is in time-trouble in a complex position. Consider the case where the player makes the move with the highest estimate of leading to a winning position. If the position remains complex, the player needs time to calculate the consequences of his next moves. But the player is in time-trouble, so he is likely to make a mistake somewhere along the road. A better strategy therefore is to simplify the position so extensive calculation is unnecessary. Even if the estimated probability that the position is game-theoretically won is somewhat lower, the player is much less likely to make a mistake, so this strategy can result in a better overall outcome.

It is especially with this difference in complexity that we will be concerned in the experimental part of this work. It is related to a technique that is already used in almost all chess programs, quiescence search. Although its name suggests a search technique, it can be seen as a hybrid between search and evaluation. The idea is that an evaluation function of the kind given in the start of this chapter is not able to give a sensible evaluation of some positions. For instance, if an exchange of queens is in progress, a pure count of material will put one side a queen ahead. The accompanying evaluation has no basis in the position itself,

but it is very hard to compute an accurate evaluation in this kind of situation statically. Therefore, a small search must be conducted. Only a very limited set of moves is considered, such as capturing unprotected pieces or pieces that are worth more than the capturing one. The outcome of this search is taken as the evaluation of the position.

2.7 Human evaluation

Data on how humans actually evaluate positions is scarce. The most detailed examples of evaluation in the verbal reports go something like "White has the pair of Bishops, at least one very good Bishop. His Knight on g5 is in trouble; weakened position.", "the first thing that strikes me is the weakness of the Black King's wing, particularly the weakness at f6". Perhaps surprisingly, the evaluation of weaker players (in the 1500-2000 Elo range) can be modeled well by computing a linear sum of a variety of positional features. Holding [16] found a correlation of .91 between human judgment and the judgment of CHESS 4.5, a chess computer using just such an evaluation function. It is unknown if this correlation extrapolates to higher skill levels.

More can be concluded about the quality of the evaluation. The position in Appendix A used by de Groot is a good example. For 4 out of 5 grandmasters, the position after 1.Bxd5 exd5 was sufficiently good to decide to make this move. Most experts and class players did not even consider the move (a failure in planning), but E1, who did, made considerable calculations following this move, could not find a satisfactory line and decided on another move. So the grandmasters could choose the best move on the basis of their superior evaluation ability.

Data gathered by Holding [15] shows that evaluation quality steadily rises through skill levels. He asked 50 players, class A through E, to rate 10 test positions. The higher rated players were significantly better at predicting, through their evaluation, which side had the better game. Another result was that higher rated players were more confident in their evaluations.

A complicating factor in these evaluation experiments is the blurry line between pure evaluation and search. Players can not help looking at possible moves when they see a position, and what they see influences their judgment. Holding measured this influence by splitting the dataset on the move choice players made. In a better position, if the move judgment was actually the correct one, the evaluation was significantly higher than if a worse move was chosen. The conclusion is that the evaluation must partially depend on the moves that are seen.

In this chapter, we have seen a glimpse of human thought processes in chess. Relative to computers, humans have very high-quality but costly evaluations. In order to be able to see far enough ahead, humans also have the ability to search selectively without overlooking good continuations most of the time. It seems reasonable to assume the information from the evaluation is used to guide the selective search effectively, but quantitative data about this relation is not

present. The human tendency to formulate subgoals (plans) is probably also important to make selective search possible.

In the next chapter, we will look at how computer programs decide on their move. The two approaches are very different; given the better human performance in domains other than chess, there is still a lot to learn from the human approach.

Chapter 3

Computer chess algorithms

3.1 Conventional algorithms

In this section we will describe some of the most used algorithms in the adverserial search domain. These algorithms are widely available on the world wide web, including pseudocode, so we will not give pseudocode here.

3.1.1 Alpha-beta

Minimax is the algorithmic implementation of the idea to look at all your moves, and then at your opponents' moves after every single move you made, and so on. The name stems from the fact that the algorithm maximizes over the values of the options of the current player and minimizes over the options of his opponent. A common reformulation of this idea is negamax, which always maximizes the negation of these values. The results are the same, but it allows for a simpler implementation. The largest flaw of the algorithm is the exponential complexity. With a branching factor b and a search depth d the complexity is $\mathcal{O}(b^d)$.

The alpha–beta algorithm produces the same results as minimax, but has a much lower complexity. As a result, no programs use minimax anymore. The basic intuition can best be expressed with an example. See figure 3.1. The maximizing player must move from A and has searched node B and found it has a value of 4. It is currently busy evaluating node C. One of its children, node D, has a value of 3. In this situation, the value of node E does not matter anymore. If its value is more than 3, the opponent will choose node D and then the value of C is 3. If its value is less than 3, the opponent will choose node E, and the value of node C will be less than 3 as well. Whatever the value of E, node B is the better choice. Consequently, node E does not have to be searched.

This idea is implemented by keeping track of two values, alpha and beta, which denote the worst-case values for both players. In the example, the worst case value of A for the maximizer is 4. Because the value of C is already lower than that, node E can be safely cut.

This pruning technique works best if the best move happens to be the first to be evaluated. Under optimal circumstances, when the first evaluated move is always the best, this reduces the complexity of the search to $\mathcal{O}(\sqrt{b^d})$, or, alternatively, a search twice as deep in the same time. This theoretical result can be approached very closely, see for instance [24].

The next natural question to ask is: how to get the best move in front? One option is to use various domain-dependent heuristics. In chess, for instance, checking and capturing moves are often better, so it's a good idea to consider them first. Another idea is to do a preliminary search and use the results to order the moves. Due to the use of hashtables (see next paragraph), the overhead of this method is not as big as it might seem. The importance of these node ordering techniques must not be underestimated. Van den Herik [29] says about this: "Chessprogrammers probably even put more energy in this part of their chess program ... than in the functions for evaluating positions" [translated from dutch].

Many positions in the search tree can be reached by more than one sequence of moves. To avoid evaluating such positions again, it is a good idea to keep track of the positions that are already evaluated. For these positions, the evaluation value must be stored. The part of the program responsible for this is called the hashtable, after a data structure that allows $\mathcal{O}(1)$ membership testing and retrieval of the evaluation value. Sometimes they are also called transposition tables, but this is not technically correct most of the times, because the need to evaluate positions more than once can have more causes than transpositions in move sequences. The preliminary search mentioned in the previous paragraph is an example of this.

As already mentioned, a preliminary search can be used to determine the best move ordering. The best preliminary search probably is one that is just one ply shallower than the main search. To provide the preliminary search with a good move ordering, another preliminary search can be made, again just one ply shallower. This can be repeated until the preliminary search is only 1 ply deep. The resulting algorithm is called iterative deepening. At first glance it may seem a very inefficient algorithm, but actually it is not. Even with a very

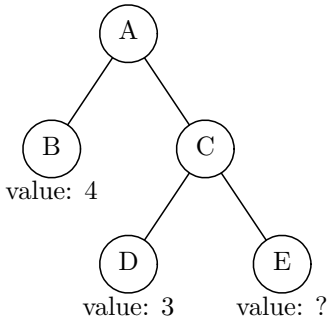

Figure 3.1: An example of $\alpha - -\beta$ pruning

good move ordering in place, the effective branching factor of a search in chess is still about 8. This means that a depth $n-1$ search takes about 8 times less time to complete than a depth n search. An $n-2$ depth search takes even 64 times less time. The overhead of all these preliminary searches is slightly less than $\frac{1}{7}$ of the time of the final search. On average, the speedups from better node orderings are much bigger.

Another trick to reduce the amount of searched nodes is the so-called null-move heuristic. Its assumption is that there is always some move that is better than not to move at all. When searching, we first see what happens if we do not make a move. If the resulting value is good enough to make the node irrelevant to the rest of the search, we can dispense with the real search. This works because usually there is some move that has an even better result than not moving at all. In chess this is the case until deep into the endgame, but in other games it is more problematic.

In essence, these are the techniques that are currently used by the worlds best chess programs [6]. In chess, they are enough to reach world-championship level, but in other games they are just enough to reach amateur level, such as Go. In playing the game the results are good, but these techniques do not contribute anything to models of human chess playing. It is widely agreed that human chess players use completely different methods of deciding their moves. In this sense the chess research program has failed, because it has not yielded new insights into human reasoning or learning, which was the original goal.

3.1.2 Problems with alpha–beta

Junghanns [19] discusses a number of problems with the original alpha–beta algorithm. Many of the techniques from the previous section are aimed at correcting some of these short-comings, although they cannot solve them completely. The problems are:

- Heuristic Error: Alpha-beta assumes evaluations to be correct. As seen in the discussion of the meaning of evaluation, this notion of 'correct' in itself is already problematic. But it generates another problem, in that only one faulty evaluation can cause a wrong decision.

- Scalar Value: All domain-dependent knowledge is compressed into a scalar value at the leaves. As discussed in the meaning of evaluation, there is more relevant information present. Compression into a total ordering of preference (usually implemented by a number) discards potentially usable information.

- Value Backup: Non-leaf nodes are valuated as the simple maximum (or minimum) of their descendants. However, information about other nodes than the best is important as well. If the second-best node is much worse than the best, an error in judgement can have serious repercussions. On the other hand, if there are many approximately equal continuations, one

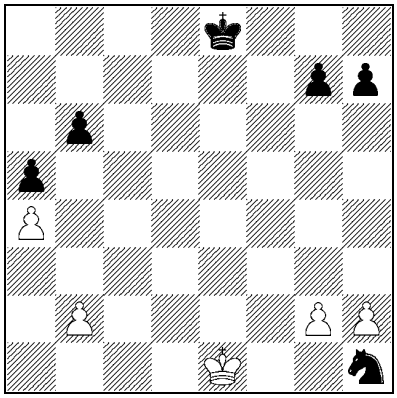

Figure 3.2: A position where the horizon effect can cause problems

incorrect judgement does not have much influence. A simple value propagation rule like taking the maximum cannot take this into account.

- Expand Next: Alpha-beta searches the tree in a depth-first order. It is very hard to use information from other parts of the tree to decide which node to expand next; in fact, only the alpha and beta values can make the algorithm stop searching a certain node, and when such a decision is reached the node cannot be revisited ever again. This is rather inflexible.
- Bad Lines: Alpha-beta gives a guaranteed minimax value of a tree of a certain depth. To be able to do this, even patently bad moves must be searched to that depth. The computations can probably better be spent elsewhere.
- Insurance: This is the opposite of the 'bad lines' problem. As alpha–beta proves the minimax value, it never misses anything within the minimax depth. Selective algorithms can incorrectly judge a move as irrelevant and therefore miss the best continuation. Therefore, insurance is a strong point for alpha–beta and a potential problem for selective algorithms.
- Stopping: The alpha–beta algorithm does not deal with the problem of when to stop searching. Most of the time, a position is searched to a specific depth, so time spent on each position is independent of the importance of the move-choice in that position.
- Opponent: Minimax algorithms assume the opponent plays according to the same algorithm as itself. Therefore, they cannot use any information about known weaknesses of an opponent.

Another problem with minimax search not mentioned by Junghanss is the horizon effect. It is best illustrated with an example, see figure 3.2, where black is moving. Assume black searches 6 plies deep. If he does 'nothing' (e.g.

moving the king around), white will capture the knight. If black moves 1. ... b5 however, he can take the white pawn while white is moving to the knight. The problem is that white can capture both, but he needs more than 3 moves for it, so a 6 ply search will come to the conclusion he can only capture one of them. Because the knight is more valuable, it is best to go for the knight, as far as minimax is concerned. However, if white is not affected by the horizon effect, he will just take the pawn (2. axb5), and the whole process starts again. Black thinks that white will move to the knight, so 2. ... a4, 3. Kf1 a3, 4. Kg1, axb2, 5. Kxh1 is the best black thinks he can do with his moves at this moment. Of course, white would not play 4. Kg1, but 4. bxa3. Note that at move 3, black would see his mistake because white now only needs 2 moves to capture the knight, so now white can capture the pawn as well as the knight. A very devious white player can foresee this and move 3.Ke2 instead of 3.Kf1, just to keep the capture of the knight at the horizon (this is a nice example of using knowledge about weaknesses of the opponent to one's advantage, and this very thing has been known to happen in the early days of computer chess!).

The problem is that black thinks that the only way for white to capture the knight is to move to it right now. Black thinks that any other action by white saves the knight, which of course is not the case. But black acts on the thought and tries to capture some pawns while white is busy capturing his knight.

3.2 Selective algorithms

Over time, a number of selective search schemes has come up, which we will discuss in this section. In essence, the only thing a selective search algorithm needs to do is tackle the 'bad lines' problem. In order to do so many algorithms also address other problems as well.

3.2.1 Pruning or Growing?

All selective search algorithms must decide which nodes not to search. There are two distinct methods of reaching these decisions. First of all, there is the method proposed by Shannon, which is a function that is given a position and a move, and returns a 'yes' or 'no' answer. This method can be seen as 'pruning' the tree, and pruned branches will never grow again. The pruning decision must be reached with information present at the node itself.

Opposed to this method is the method of growing. This method can be seen as a function that takes the entire current search tree as an argument, and returns the node to expand. In contrast with the pruning method, no branch is pruned permanently; other branches just have priority over it, but that may change in the future. The advantage is that much more information can be used to decide where to search next. The disadvantage is that all this information needs to be managed and stored.

An example where the advantage of the growing method can be seen clearly is a position where there are two possible continuations. Both seem very good,

but one of them is just a little bit better, so that one is searched first. A little while into the search, it seems the evaluation of the first move was all wrong, it is much worse than we expected. Therefore we abandon the search and start searching the other move. There things turn out to be even worse than for the first move. At this point, the difference between pruning and growing methods becomes clear. Pruning methods can now only search on at the latest position or stop searching and make the first move. Growing methods can switch back to the first option without any loss.

A different example comes from Conitzer and Sandholm [9], slightly adapted by us. Their main interest was to investigate computational hardness of meta-reasoning problems, and the following problem, in a more generalized version, turns out to be NP-hard. In the problem a robot can dig for precious metals (i.e.make a move) at different sites. To aid its decision where to dig, it can also perform tests wether the metals are present or not (i.e. the search actions). The probability that there is gold at location A is $\frac{1}{8}$, that of finding silver at site B $\frac{1}{2}$, copper at site C $\frac{1}{2}$ and tin at site D $\frac{1}{2}$. Finding gold has value 5, finding silver value 4, copper value 3, tin value 2, while finding nothing has value 0.

If the robot cannot perform tests at all, it should dig at site B for an expected value of 2. If it has time for just one test, it should test for silver at site B. If silver is present, it should dig it up, if not it should try to dig for copper. This strategy has an expected value of $2\frac{3}{4}$. The strategy of searching for gold and digging for silver if there is none has an expected value of $2\frac{3}{8}$. When the robot can do two tests, it becomes more complicated. It is still best to search for silver first, but the next search action depends on the outcome. If silver is found, it is safe to search for gold, because even if no gold is found, the robot can dig up the silver. If no silver is found, it is better to search for copper and if it is not found the robot should dig for tin and hope for the best. This strategy yields an average value of $3\frac{1}{16}$, while just searching for gold and silver regardless of outcome yields $3\frac{1}{32}$.

In this simple example, pruning methods can in principle still determine the correct search order (and stop if search becomes useless), because the conditional search action (search for tin or gold, depending on the outcome of silver) occurs in the same node, the root. If they occurred in children from different parents, pruning methods would not be able to conduct the search in the most efficient way. This is in essence the same phenomenon as the first example. The recurring theme is that pruning methods must either search a move completely or not at all.

We will now look shortly at some specific selective search algorithms, and in-depth at the Bayesian Search algorithm as that is the algorithm we have used for most of our experiments.

3.2.2 Adhoc selectivity

The first computer chess programs used selective searching. There was only one reason for this: there was no time for even a minimal full search (see [29, page 120], page 120). A variety of heuristics was used for this purpose, generally

the same as the move-ordering heuristics for alpha–beta described above. This method was never really successful, and when alpha–beta was invented and more computing power came around, it was quickly abandoned [28].

3.2.3 Probcut

A more successful approach was taken by Buro [5], with the ProbCut algorithm. The idea here is to use a shallow search as an estimate of the outcome of a deep search. With some statistical techniques a confidence interval of this estimate is computed. If this entire interval is outside the alpha–beta interval, the search can be relatively safely stopped. If not, a deep search is carried out anyway. Buro tested this algorithm on othello, where it was very succesfull, but attempts to implement it in chess have failed so far [19]. An important thing to note is that the same confidence interval is used for all positions in the game. No attempt is made to use different intervals for different types of positions. As such, this heuristic is not position-dependent, only game-dependent.

3.2.4 Best First Search

Korf and Chickering [20] introduced Best First Search in 1995. It expands the current principal variation, the 'best' node. The idea is that as the value of this node determines the value of the root, it is important to know its value more accurately. There are a few problems with the Best First approach though. If the evaluation of a node is a serious underestimation of its actual value, the algorithm will never find this out. Hence, it can even terminate without finding the optimal move. Korf and Chickering themselves also note that the algorithm does better with a more accurate evaluation function. Their experiments also suggest that with a less accurate evaluation function it is better to use the algorithm on top of a standard minimax tree.

Another problem is that much of the selectivity characteristics depend on the so-called *tempo* effect. It refers to the fact that the player who is on move can usually improve his position. This changes the value of the node, which has a high probability of changing the principal variation. After all, if the value increases a lot, the minimizer will likely try another move somewhere up the tree. The problem lies in the fact that this tempo effect is actually an inaccuracy of the value function. The value function should reflect the actual value of a position as closely as possible; if the value after an 1-ply search is on average higher than the static evaluation, the evaluation function contains a bias against the player-on-move. If this bias is very high, the search degrades towards a breadth-first search. If there is no bias, the search becomes very selective. Korf and Chickering did not look at the effects of bias, so experimental data on it is not available.

3.2.5 Randomized Best First Search

In order to fix the first problem with Best First Search Shoham and Toledo [26] introduced Randomized Best First Search. In essence this algorithm tries to expand nodes according to the probability that they are the best. This ensures that moves currently viewed as sub-optimal may be expanded as well, according to an estimation of the chance they are actually optimal.

The algorithm for doing this works recursively: at every (sub)tree, the properties of its subtrees are examined and a choice is made in which subtree a node is expanded. In that subtree the same procedure is carried out, until a leaf is encountered, which is then expanded. The properties of the subtree that are examined are the number of direct children it has, the (total) number of expanded nodes in the tree and the current negamax value of the tree. Each combination of these properties is associated with a probability distribution. This probability distribution associated with a subtree is sampled once, and the subtree with the best sample is expanded. This ensures that better subtrees have a higher chance of being examined further.

The last thing the algorithm needs are the probability distributions. For this, they used a large set of positions; each position was searched to depth 10 to determine its 'real' value. They then expanded nodes one by one in each position and recorded the difference between the minimax values and the 'real' values along the way. These differences were fitted by a normally distributed random variable; every node-expansion-count/number-of-moves combination has its own random variable.

Of course, an order in which the nodes are expanded is needed. For this, they used the algorithm itself, initialized with a trivial distribution. This is then repeated a few times, initialized with the resulting random variables, until it converges.

There are a few important things to note about this algorithm. First of all, the only position-specific data that is used for the probability distribution is the number of available moves. In terms of chess, this means that any information regarding pieces that can be captured, checks that can be given, and general (in)stability of the position is not used. The starting position, which has 20 moves, gets exactly the same distribution as some endgame position where white happens to have 20 moves. This is of course a point of the algorithm that can be improved.

Another important thing is that only information about the size of the subtree is used, but no information about the shape of the subtree. The size of the subtree can be used to determine the quality of the backed up minimax value. If a subtree is large, its value is the result of a large search, so the difference between this value and the 'real' value should not be very large. However, in some cases the size can be a misleading indicator of this difference. For instance, say we have a very promising move in the current position. Naturally, the algorithm expands many positions that come after this move, because it looks like it is the best. After searching thousands of positions, we find out that the move is not so good after all, and the principal variation changes to the next best

move, which has received little attention. At this point, the algorithm thinks that the current backed-up minimax value is quite stable. The position has been expanded thousands of times. In reality, however, the value depends on a much smaller amount of nodes.

3.2.6 B*

B* was developed by Berliner [4], and has the most similarities to human chess thinking from the algorithms described in this thesis. It is inspired by the psychological notion of 'threat' in a position.

The driving force for search decisions is not to determine accurate values for all moves, but to prove that the current best move candidate is better than all alternatives. This is done by giving all nodes an optimistic and realistic value, with the optimistic value as the value of the position if all threats are achievable. The rule to backup values from children to their parent is simple. The optimistic value of the parent is the maximum of the optimistic values of the children, the realistic value the maximum of the realistic values.

When the best move candidate has a higher realistic value than the optimistic value of every other move, a condition named 'separation' is achieved (the value range of the best move is separated from the others). When this occurs, no more search is necessary and the move can be carried out. To achieve separation, two strategies are possible. One, called 'ProveBest', attempts to increase the realistic value of the best node. The other, called 'DisproveRest', attempts to decrease the optimistic values of the other nodes. Both strategies work by expanding nodes in best-first manner.

Palay [22] extended this algorithm to use probability distributions instead of simple ranges. This addresses the scalar value-backup problem as described by Junghanns. Here, the left parent node is clearly better than the right. However, if no distributions are used, the backed up values for both are the same. If distributions are used the left one is better. This chance not only influences which move is initially seen as the best, but also changes the way nodes are expanded. The distributions can be used to calculate more precisely which child has the highest chance of achieving separation. This change improved the effectiveness of the algorithm greatly.

At first, Berliner used some simple heuristics to determine optimistic values. Palay had a more principled approach: he gave the player on move two moves in a row, and used the value obtained by such a search as the optimistic value. This value can be seen as the threat value, as that which will happen if the opponent does not respond to the first move. Still a lot of exceptions had to be made in case of checks, but it seems to be a good approach.

The resulting algorithm did quite well on a set of chess problems that was used as a benchmark at the time. Those problems were mainly tactical in nature however, and the more strategic ones were difficult for B*. Therefore, some work must still be done in order to make the algorithm play a decent complete game.

3.2.7 Conspiracy numbers

A conspiracy number of a node was defined by McAllester [21] to be the minimum number of leaves whose value has to be changed in order to change the value of that node by a certain amount. Such change can occur due to further searching. These nodes can be seen as conspiring to change the value, hence the name. As an example, see figure 3.3. The current value of the root is 3, but only nodes F or G need to change to 2 or 1 to change the value of the root to those respective values. In order to change the root to 0, two conspirators are needed. Both a node from the left hand side and the right hand side need to change to 0 in order to change the value of the root to 0. To change the root value to 4, only F needs to increase its value to 4 or more.

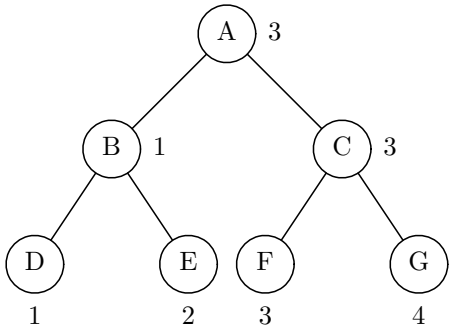

Figure 3.3: An example tree

If the size of the conspiracy to change the root by a small amount is large, it is unlikely that the evaluation of the root is wrong. Therefore, it makes sense to search in such a way to make the smallest such conspiracy larger. McAllester presents an algorithm, conspiracy number search, that implements these ideas. It searches the smallest conspiracies, until the smallest conspiracy is greater than some predefined number.

A problem with this algorithm is that it does not take into account what conspiracies are more likely than others. As can be seen in figure 3.3, all nodes are member of multiple conspiracies of size 2. If those were the smallest conspiracies available, the search would be more or less random.

3.3 Bayesian search

In this section we will describe the Bayesian Search algorithm developed by Baum and Smith [1]. This description has three parts. In the first, we will give the rationale for the algorithm, in the second some algorithmic issues are identified and in the third part we will give a large example of a run of the algorithm.

3.3.1 Philosophy

The point of search is to overcome the inaccuracy of the evaluation function. The standard way of doing this is to search indiscriminately to a certain depth. This method has its merits, but lacks scalability, as was discussed in the previous chapter. It should be possible to do things better.

Even though the inaccuracy of the evaluation function is the reason for searching, some positions are evaluated with a higher accuracy than others. This notion has some validity in chess psychology, but how might it be interpreted game-theoretically? If the interpretation of the evaluation function is a probabilistic estimate of the chance of winning, what does the (in)accuracy of this estimate mean?

One naive answer is to say that an inaccurate estimate should either have been much lower or much higher, and not so for the accurate one. Let's narrow down this line of thought with an example: a position P has an estimated probability of 80% to be a win. This estimation is inaccurate, so it should be either (with a 50% chance) much lower (60% probability of being a win) or much higher (100% probability of being a win). This is useless however: any probability of 80% can be interpreted in this way, there is no way to destinguish the 'accurate' and 'inaccurate' versions of 80%.

A practical way out of this problem is to interpret it as the amount the value will change as a function of further search. In this interpretation, an accurate evaluation means that a search from such a position will not change the value of the position much. An inaccurate one means that large changes are likely. The advantage of this option is that it can be used directly for guiding search: do not search the accurate positions, as it is unlikely such a search will change anything.

There is more to it than just this, however. Imagine there is a very inaccurate position somewhere in a tree. Somewhere above this position one of the players can make another move, which is much better. Searching the inaccurate position does not do much good, because it is not very likely the result will influence the value of the position above. So the shape of the tree needs to be taken into account as well when deciding where to search.

The Bayesian Search algorithm proposed by Baum and Smith deals with all this. It is rather complicated and takes the rest of this section to describe.

Their accuracy function estimates the change in the evaluation of a position after a relatively shallow exhaustive search. The estimate is represented as a discrete probability distribution. For purposes of where to search (or, which nodes to expand), the estimate is assumed to be perfect, meaning that searching the position will indeed result in the given values with the given probabilities. An example initial tree is given in figure 3.4.

The distributions higher up the tree can be calculated from the distributions at the leaves with equation 3.1. Note that the minus is due to the negamax sign convention.

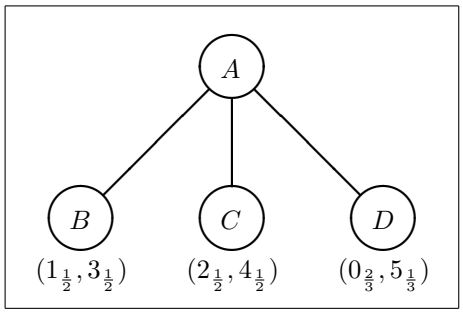

Figure 3.4: Example tree with probabilities at its leaves

$$P(V_{parent} \leq -x) = \prod_{i=1}^{b} \big(P(V_{child_i} \geq x)\big) \tag{3.1}$$

This equation is concerned with cumulative probability functions and not with the given probability distributions, but it is trivial to convert the two to each other. The distributions calculated in this way mean the same thing as the distributions at the leaves, except that all the descendant leaves must be searched to know the exact value of the node, instead of just the node itself.

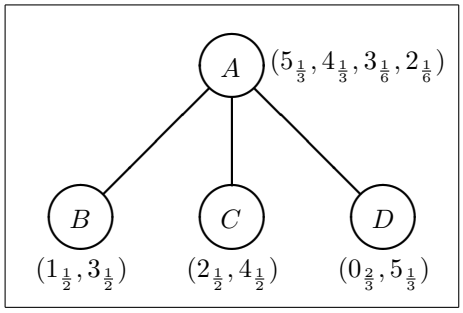

Figure 3.5: The same tree as in figure 3.4, but with probabilities calculated for the non-leaf nodes, i.e. the root.

This also applies to the root. So if all leaves could be searched, the value of the root would be known. The best move would be known as well. However, what move should be taken if there is no more time to search? It is the child with the highest expected value. This position can be searched further next turn, so the exact value will be known then. The expectation of this exact value is by definition the current expected value of this child. This reasoning adds the assumption that we are indeed able to search positions down the tree fully on subsequent turns.

The current concepts yield a natural criterion for stopping search. If the program searches the current tree, the result will be the expected value of the root. If the program does not search, the result will be the maximum expected value of the children. The difference between these two values is named U_{all} by Baum and Smith, short for the utility of expanding all leaf positions. U_{all} can be seen as a measure of the information present in a tree, information that can be gained if the tree is searched. If U_{all} is low, there is little reason to search, and if it is high, search will be profitable.

This does not give any means yet for deciding where to search. For that a measure of the usefulness of a single expansion is needed. Just as it is possible to calculate the usefulness of expanding all leaves, it is possible to calculate the usefulness of expanding any subset of leaves, under the assumption that a move is made right after. Taking a single leaf as the subset, this gives a measure for expansion priority.

This measure has it's problems though. Many programs expand millions or even billions of positions before moving, and only one expansion of them is the last, obviously. So the assumption that the program must make a move after expanding one node is violated. Besides this theoretical problem, there is a practical problem as well, as can be seen in figure 3.6 (reproduced from Baum and Smith). In this tree, only T needs to be expanded according to this expansion strategy. Expanding either V or W doesn't have any immediate influence on the move. Expanding both V and W can have a large influence however, and expanding one of these nodes is more important if the program can make more expansions.

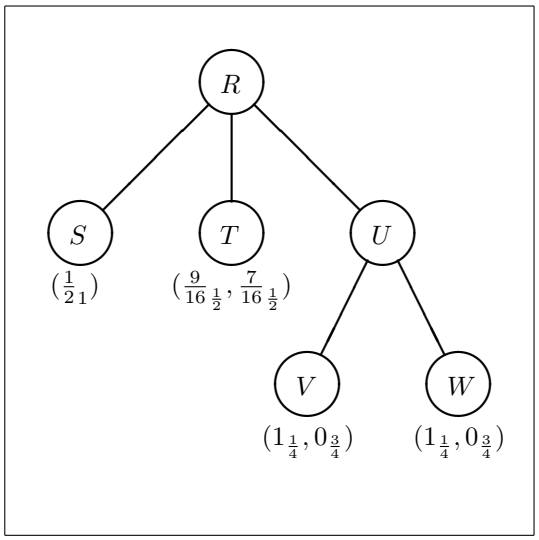

Figure 3.6: Example game tree

As there will most likely be further search after an expansion, a better idea is to look at U_{all}, the measure of the usefulness of search, after the expansion. If it rises, further search will be useful, so this is a good thing. If it falls, further search won't be necessary, which is a good thing as well. Only if it remains relatively equal, the expansion was not useful. So the proposed measure is to take the absolute of the possible changes in U_{all}.

To get a feeling for the effects of this measure, here it will be calculated for figure 3.6. First we need to know U_{all} for our current tree. The expected values of the root and the best child are needed for that: the expected value of the root is $\frac{1}{16} \cdot 1 + \frac{15}{32} \cdot \frac{9}{16} + \frac{15}{32} \cdot \frac{1}{2} = \frac{287}{512}$ and and of S $\frac{1}{2} \cdot 1 = \frac{1}{2}$ respectively, which gives a U_{all} of $\frac{287}{512} - \frac{256}{512} = \frac{31}{512}$. Now there are really just 2 options: expanding either T or V (W has the same result as V).

T will be considered first. If it turns out to be $\frac{7}{16}$, U_{all} becomes $\frac{17}{32} - \frac{1}{2} = \frac{16}{512}$. If it turns out to be $\frac{9}{16}$, U_{all} becomes $\frac{151}{256} - \frac{9}{16} = \frac{14}{512}$. Weighted for the probabilities of either event, the average absolute change in U_{all} is $\frac{16}{512}$.

Now if V is expanded, and found to have a value of 0, U_{all} becomes $\frac{1}{32}$. If it turns out to be 1 however, U_{all} will rise to $\frac{76}{512}$, because the expansion of W becomes vital now. Again weighted for the probabilities, the average absolute *change* is $\frac{45}{1024}$. All these computations are far from complete, but in the end of this chapter a complete run through the algorithm will be presented.

In this example the algorithm would choose to search move V, due to it's interactions with leaf W. The ability to take into account these interactions is the strongest property of this algorithm.

3.3.2 Algorithmics

The previous section described what Baum and Smith's algorithm does, and why it does it. How it does it is a non-trivial question. The (naive) algorithm used in the examples would have a very high complexity. It would calculate the chance distributions for all nodes in the tree, an $\mathcal{O}(n)$ operation, in order to calculate the expected values of the root and its children. It would have to do this for every possible outcome of every leaf, making it an $\mathcal{O}(n^2)$ algorithm (assuming the branching factor is more than one, so the number of leaves is proportional to the number of nodes). This would be enough for only one expansion. n expansions would therefore need $\mathcal{O}(n^3)$ time, whereas conventional search methods like alpha-beta only need $\mathcal{O}(n)$. This is a fatal flaw of any algorithm, but Baum and Smith describe two tricks to get the complexity back to $\mathcal{O}(n)$, both tricks gaining a factor n.

The first trick is rather simple. Instead of expanding one leaf at a time, expand a fixed fraction of all leaves. If $\frac{1}{q}$ is the fraction, $\frac{n}{q}$ leaves will be expanded. The cost of calculating which nodes are most relevant to expand can then be divided among these leaves. This reduces the complexity of the complete algorithm by a factor of n, and increases it by a factor q. But q is a constant factor, so it can be conveniently disregarded. Baum and Smith call this the 'gulp trick', because the algorithm 'gulps' a group of nodes in one go, instead of just one.

The feat the second trick performs is the calculation of all utilities simultaneously, so it only needs $\mathcal{O}(n)$ time to calculate the utilities of n nodes. Two facts make this possible. First, U_{all} is a linear quantity of the probabilities of the root and its children. This means that if one probability rises by δ and another is lowered by δ, the change in the outcome of the function is linear in δ, and can easily be computed. The second fact is that the probabilities of a non-leaf node is a linear quantity of the probabilities of its children. So if the influence of changes of probabilities of a non-leaf node are known, the influence of changes in its children can be computed as well. How this is done exactly is shown in the large example in the next paragraph.

This can be reiterated to the leaves of the tree. Important here is that information in a node higher up in the tree can be used for all its descendants. So every node in the tree needs to be visited only once, which means this part of the algorithm is only $\mathcal{O}(n)$.

All leaves now have information about the influence of changes in their probabilities on the function that needs to be calculated, in this case U_{all}. Remember that expanding a node is modelled as drawing a value from the probability distribution of that node. This can be viewed as a change in the probabilities of the node: the drawn value gets a probability of 1, and the other possible values get probabilities of 0. So for each value the new U_{all} can be calculated, and thus $\delta_{U_{all}}$ as well. Now take the weighted average of these changes in U_{all}, and the calculation is complete.

The last part of the algorithm takes constant time per node, so $\mathcal{O}(n)$ in total. This means the entire algorithm takes $\mathcal{O}(n)$ as well.

3.3.3 Full example

In this section we will give a full example of all the steps in the Bayesian Search algorithm. We start out with the search tree in figure 3.7, and work our way through all the steps taken by the algorithm. Note that the negamax sign convention is used, so all the values in a node are relative to the player who has the move in that node. All the final computed values are shown in tables 3.1 to 3.4, and we explain how these values are computed in the following paragraphs.

The first thing that needs to be done is to propagate the distributions up the tree. We start down in the tree, with node E. Its children have four spikes in total, each of which must be represented. All these spikes are put in a table, flipped (due to negamax), and sorted from high to low. This can be done efficiently because the spikes at the children are sorted as well, so only the last stage of a merge sort is needed.

Now the probabilities that node E actually has one of these values needs to be computed. There is a 50% chance node H has a value of -6, and in that event node E has a value of 6 as well. Therefore the first spike has a 50% chance. The chance that E has a value of 5 is more complicated. This is the case if node G has a value of 5, and no other node has a value higher than 5. The first event has a 50% chance of occuring and the second one as well, and because the probabilities are independent, the total chance is the product of 50% and

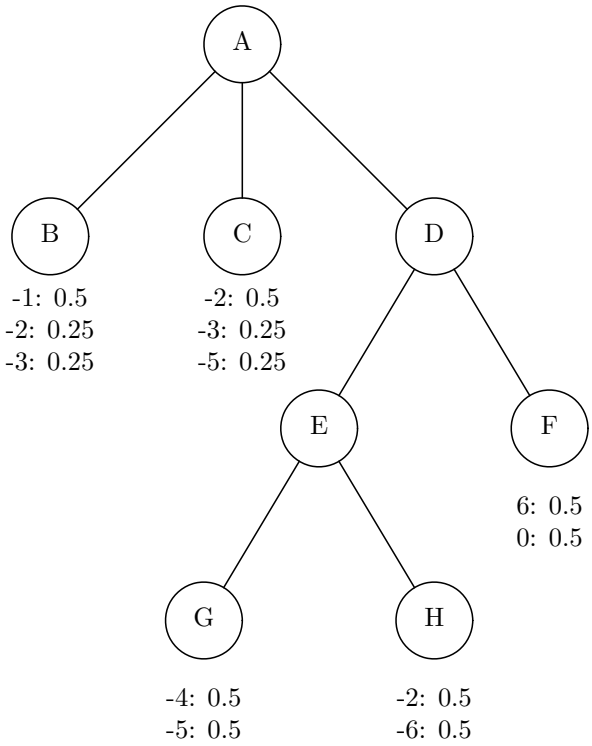

Figure 3.7: An example Bayesian Search tree

50%, 25%. By the same reasoning the chance that E has value 4 is 25% as well. The chance that the value of E is 2 is zero, as there is a 100% chance node G is higher than that. The cumulative probabilities could be computed by equation 3.1, with equation 3.2 the probabilities can be computed directly.

$$P(eval_N = x) = P(eval_c = x) \cdot \prod_{i \neq c} P(eval_i < x) \tag{3.2}$$

the subscribt N refers to the current node, while c refers to the child from which the spike at that location originates.

The same procedure is used to compute the distribution of node D, but here a problem arises: two spikes have the same location. Because neither is smaller than the other, the total probability would drop below 1 if the formula is used exactly. To remedy this problem we consider one of the values as larger than the other anyway, in other words we impose a total ordering on equal spike values. What this total ordering looks like does not matter, as long as the same one is used throughout the algorithm. In our example we treated values from lefthand nodes as higher than values from righthand nodes. If the spikes come from the same node, the ordering from that node is used.

We now have distributions for all nodes in our tree. Our next step is to

compute the expected values of these distributions for the root node and its children. Nodes A,B,C and D have expected values of 4.21875, -1.75, -3 and -2.625 respectively. If we have to make a move now, it would be to node C for an expected value of 3. Note that this is different from minimax that uses as evaluations the expected values at the leaves. In that case there would be no preference between node C and D.

If we can search our tree however, we would find the exact value of A, which would be 4.21875 on average. Therefore, we can expect to gain 1.21875 points (the U_{all} metric) on our evaluation scale if we search further. This is much, so we start the procedure that computes how important each individual node is to expand.

Instead of computing the change in U_{all} on a node-by-node basis, we compute for each spike in the tree their so-called influence coefficient, a number that denotes the change in U_{all} if the spike changes by 1. Of course a spike cannot change by 1, because then the probabilities would sum to more than 1. However, U_{all} is a linear function of the heights of spikes, so if one spike increases by 0.5 and another decreases by the same amount, the change in U_{all} is the sum of the effects of these changes individually.

The influence coefficients at the root are simple, they are the same as the locations of the spikes. For example, if a spike at location 2 increases by 1 in probability, the expected value would go up by 2 as well.

The situation is more complicated for the children. We will use node B as an example. First, what would happen to A if we increase the chance of B being -3 by 1? We'll just use equation 3.2 to recompute the probabilities, disregarding the fact that the probabilities at B sum up to 2. The probabilities of the spikes with a location higher than 3 is doubled, as the chance that B is smaller than them is now 2 instead of 1. This yields an influence of $3\frac{9}{32}$. The probability of the spike at 3 with B as origin goes up by a factor 5 from $\frac{3}{32}$ to $\frac{15}{32}$, because the first factor in the equation goes up by a factor of 5 and the other factor doesn't change. This change has an influence of $1\frac{1}{8}$, so the total influence becomes $4\frac{13}{32}$.

The influence for the next spike of B, at -2, is still more complex to compute. Spikes higher than 3 are still doubled, but spikes lower than 3 and higher than 2 are not. In this case, the factor due to spike B (in the right hand side of the equation) does not go up from 1 to 2, but rather from 0.75 to 1.75. The probabilities of these spikes are increased by a factor of $\frac{4}{3}$, which gives an influence of $\frac{3}{8}$. The probability of the spike at 2 with B as origin behaves the same as the one at 3, so it gives an influence of $\frac{1}{4}$. Adding these numbers to the influence due to the changes at spikes higher than 3 ($3\frac{9}{32}$) gives $4\frac{5}{32}$.

The other spikes at B, C and D are computed in the same way. These influences, however, only measure the change in the expected value of the root. U_{all} is defined as the difference between this expected value and the expected value of the best child. Changing spikes of another child than the best has no influence on the expected value of the best child, only the spikes of the best child have. Therefore, at child C, we need to take into account that changes in its spikes affect the expected value of the root as well as that of its best child. To do so we add the locations of the spikes to the influences.

We have now computed all the influence coefficients, so we can now start to compute the QSS(Q step size, expected absolute change in the utility) for every node expansion. Expanding a node means that it takes a value from its distribution. This means the spike at its location goes up to 1 and the spikes at the other locations go down to 0. The effects of this change on U_{all} can be readily computed with the influence coefficients. Take for example node B. Let's say B gets value -1 upon expansion. The spike at -1 is then increased by 0.5, for an increase of 2.078125 of U_{all}. Both other spikes are decreased by 0.25 however, for a decrease of $4.15625 \cdot 0.25 + 4.40625 \cdot 0.25$. The total is -0.0625. For the other spikes these values are -0.0625 and 0.1875. As we are interested in the absolute amount of change the minusses are removed (if we did not, the utility would be 0 by definition). Now all these values are summed according to their probabilities of occurring, which leads to a leaf expansion utility of 0.09375.

When all the expansion utilities are computed, the leaves are sorted accoringly. A fixed fraction is expanded (the 'gulp trick'), e.g. the best 25% of the leaves, rounded up. As there are 5 leaves, 2 of them will be expanded, C and F. For their children, new spikes are computed and the entire algorithm starts anew, until U_{all} becomes too low to justify further search. At that moment the best move is made.

The choice to expand C and F is intuitatively reasonable. If F turns out to have a value of 0, the probability that node C is the best increases dramatically, so that move can probably be played. If node C turns out to have a value of 5, this probability increases a lot as well. The other nodes do not have so much influence.

A last note of warning to the future implementor: the probabilities of spikes can become very small, in fact so small that numerical difficulties arise, at least when using IEEE floating point numbers to represent the real numbers. In such a case it is advisable to set spikes with very low probabilities (e.g. $< 10^{-12}$) to zero probability. Also care needs to be taken not to divide zero by zero; in such a case the zero factors must be removed beforehand from both numerator and denominator.

Source	Evaluation	Probability	Influence
H	6	1/2	3 1/8
G	5	1/4	3 11/16
G	4	1/4	4 1/16
H	2	0	4 9/16

Table 3.1: Node E

Source	Evaluation	Probability	Influence
F	0	1/2	3 1/8
E	-2	0	3 1/8
E	-4	1/8	4 1/2
E	-5	1/8	5
E	-6	1/4	5
F	-6	0	5

Table 3.2: Node D

Source	Evaluation	Probability	influence
D	6	0	6
D	6	1/4	6
C	5	3/16	5
D	5	3/32	5
D	4	3/32	4
B	3	3/32	3
C	3	3/32	3
B	2	1/16	2
C	2	1/8	2
D	2	0	2
B	1	0	2
D	0	0	2

Table 3.3: Node A

Node	Expansion utility
B	0.09375
C	0.53125
F	1.09375
G	0.09375
H	0.34375

Table 3.4: Expansion utilities of various nodes

Chapter 4

Experiments

4.1 Generating distributions

4.1.1 Decision trees

Baum and Smith [2] used decision trees to generate their parametrized random variable function. The procedure they followed was as follows. First, they handcrafted various features for the domain in question (they used Othello). They fitted the values of these features on a large database of games, with linear regression. After that, they used various tests to remove and change features that were irrelevant. With this evaluation function, they evaluated a set of positions and computed the value after a search of a number of plies. The difference between these two numbers can be seen as the change in opinion that results from deeper search.

These differences are used to create a probabilistic evaluator, in the form of a decision tree. The following recursive procedure was used:

- Repeatedly split the dataset in two sets, set A that has a certain feature, set B does not.
- Do a Kolmogorov-Smirnov test with set A and set B.
- After doing this for all features, check if the feature with largest K-S significance is statistically significant, and if so, add this feature to the decision tree and call this procedure with set A and set B separately.

A Kolmogorov-Smirnov test computes the largest difference between the cumulative probabilities of the outcomes of the two sets.

After this procedure, each leaf in the decision tree had about 200 datapoints in them. The Bayesian Search algorithm needs about 2-10, so they needed a way to reduce this data. This was done by choosing the k spikes in such locations that the first $2k$ non-trivial moments (mean, deviation, skewness, curtosis etc.) are the same. They refer to Golub [14] for details on how to do this.

In our opinion, there are several problems with this approach. First, we like to note that this approach takes a lot of time from the experimenter (programming and selecting features). It also requires some expert knowledge of the game. Second, decision trees have inherent difficulties with non-linear data. This problem can be reduced by using good features, but it does not disappear. Another problem with decision trees is that they do not scale well for increasingly complex problems. For each extra feature that must be taken into account, the tree doubles in size and the dataset must be doubled as well to remain at the same significance level.

4.1.2 Neural networks

Neural networks do not have the drawbacks of decision trees. We experimented with various schemes. In all these schemes, 'neural network' can be equated with a standard 3-layer network that uses a sigmoid activation function for the hidden layer, and a linear activation function for the outputs. We will now discuss some of the issues encountered during the testing of various schemes.

First of all, it is possible to use different networks for different spikes or to use one network for all the spikes at the same time. In the first case each spike has its hidden nodes all to itself, whereas in the second approach there is only one network which shares its hidden nodes among all spikes. Because the first approach is slower, and in preliminary experiments it did not seem to matter much to the quality, we took the second approach in all further experiments.

We recognized two ways in which spikes can be represented. A spike can be represented by two output nodes, one of which denotes the height of the spike, the other denoting the location. Another option is to use just one node, which represents the height. In that case the location is set beforehand.

The learning rule for the first approach is as follows: first, obtain the distribution by running the network on an input. Now determine which of the spikes is closest to the actual outcome. This spike is called the 'winner'. Next, the spikes of the network are changed by supervised learning in the following way: The height of the winning spike is updated towards 1, and its location is updated towards the location of the actual outcome. The heights of the losing spikes are updated towards 0, and their locations not updated at all. At every learning step, all spikes are updated either towards 0 or 1 and the size of the step is dependent on the learning rate. In order to prevent the last few learning steps to dominate the heights of the spikes, the learning rate must be lowered towards 0 during the learning period. This is not specific to this setup, however, and declining learning rates were used throughout our experiments.

Experiments showed that this procedure converged in the non-parametric case (all inputs have the same random variable associated with them). However, in the parametric case problems occurred. It still correctly learned the heights of the spikes, but the locations got stuck. Consider, as example of this problem, a parametrized random variable with two subvariables, a high variance one and a low variance one. Which random variable is actually used in a datapoint depends on its parameters. In this scenario one spike typically tends to represent

an extreme of the high variance random variable. It reaches the correct location before it can recognize which random variable is used, so it will predict this location for both random variables. Even if the network as a whole learns to distinguish between the two subvariables, it will never 'win' again when the low variance random variable is encountered, so its location will not change. This actually occured in our experiments: the height was learned correctly (it should be zero and it was), but the location was stuck on the location of the high variance extremes.

A possible solution to this problem might be to update the locations of a network even if it loses. This will ensure that a network will never get stuck in empty space, but it may also cause convergence to incorrect values. The probabilities generated by such networks will tend to be a bit tighter than warranted by the data. A fix for this might be to stop updating the locations of losers after some time. Because of these problems, we did not pursue this approach any further.

The learning rule for the second approach, with outputs representing heights of spikes at predefined locations, is similar to that of the first. First a winning output is determined, according to the locations of the spikes that were set beforehand. The winner is updated towards 1, the losers towards 0. This eliminates the need to learn locations of spikes, but a lot of outputs are needed to cover all possible outcomes. Most likely there will be more outputs than the desired number of spikes, so we need a method to compress the data. We used a weighted version of the K-Means algorithm for this, with the preset locations as datapoints, and their calculated heights as weights.

The standard K-Means initializes by choosing random datapoints. In our case, this frequently led to bad local optima, so another approach was needed. Fuzzy K-Means could be a solution to this, but it turned out it slowed down the entire algorithm by a degree of magnitude, which was unacceptable. After some experimentation with initialization schemes, we used one where $1/n$ of the data is assigned to each cluster (which is trivial to do in the one-dimensional case, but not in multi-dimensional problems that are usually presented to K-means). It is fast and gives adequate results.

An advantage of this approach is that it is easy to generate differing amounts of spikes. After all, the amount of clusters is just a parameter to K-Means. This makes it easy to generate more spikes for nodes near the root, where the extra precision is more important than the overhead, while generating fewer spikes for nodes down the tree. We did not experiment with this option however.

An option that is present for both schemes is to use a separate value function, and to interpret the values from the spikes as relative to the value calculated by the value function. It has advantages as well as disadvantages:

- It allows the spike generator to specialize on the 'risk' in a position, while the value function specializes on the average quality of the position.

- In the second scheme, it allows for fewer output neurons. Without a value function, the entire range of outcomes must be covered; with a value function, only the deviations from it need to be covered.

- It is possible to only run the value function if no spikes are needed, which is faster.

- Learning is more problematic, because learning of the spikes can only start when the value function has become relatively stable.

- When spikes are needed, two networks must be run, which is slower.

In our experiments, we used both approaches, depending on the domain. In domains with a small set of possible outcomes, fixed locations are used, while in domains with an open-ended set of possible outcomes relative locations are used.

4.1.3 Errorfunctions

All these ways to generate discrete probability distributions need to be compared quantitatively somehow. There is no easy answer, however, on how to compare them. The quality of the decision tree approach by Baum and Smith was not measured at all, because it was implicitly assumed their method is optimal for the current task. Unfortunately it is impossible to use this method ourselves, as it needs a lot of samples from a single random variable to accurately determine its moments, but we only have 1 per random variable. This is due to the fact that we decided not to categorize the positions, but instead let every position have its own random variable.

So we need a method for quantifying the performance of a set of random variables, even though we have only one actual value for each. It is intuitively plausible that two things need to be minimal. First, there must be a spike close to the actual outcome. To get an error measure out of this, the square distance between the actual outcome and the closest spike is calculated. In formula:

$$\int_{-\infty}^{\infty} p(x) \cdot min_i((x - loc_i)^2)dx$$

Secondly, the higher the spike at the closest location is, the better. To measure this aspect of the quality, the following formula is used.

$$\int_{-\infty}^{\infty} p(x) \cdot -\ln(spikeheight_{closest})dx$$

Taking the logarithm may look a little strange, but this measure is actually minimal when the spikes represent the actual probabilities that actual value is closest to that spike. The negative logarithm is the standard error function in use in the field of probability density estimation.

These error functions cannot both be minimal at the same time: The distance function is minimized when there are infinite spikes, one at every real value, and the heightfunction is minimized when there is only one spike. Therefore, any practical function must make a trade-off between these two errors.

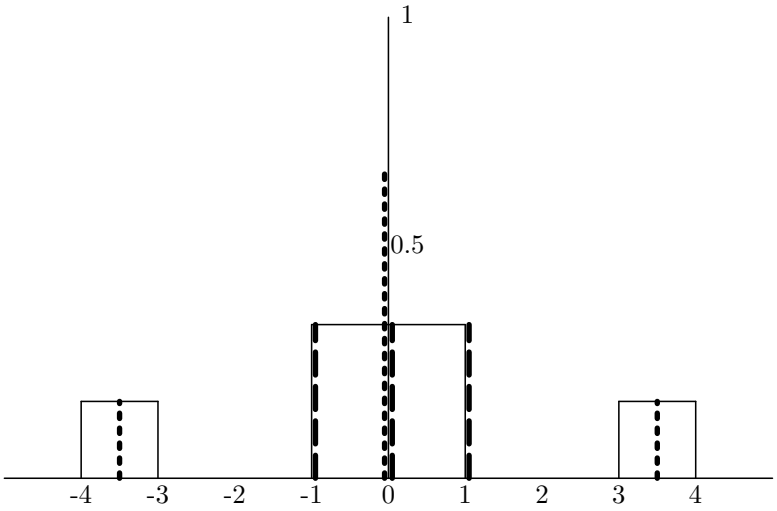

Figure 4.1: A random variable (the boxes) where optimizing different aspects yields different results. The dotted lines minimize distance-error and the dashed lines minimize height-error.

An obvious way to do this is to use a fixed number of spikes. While this eliminates the limiting cases, there is still room for variation. See for instance figure 4.1.

The optimum for the spikeheights is shown as the dotted line, the optimum for distance the solid line. So even with a fixed number of spikes, a choice must be made about the relative importance of the two error functions.

For the Bayesian Search algorithm, it is important that outliers are correctly represented. By their nature, outliers are few and far between. This means that they contribute relatively much to the distance error, because of the large distance, and relatively little to the heighterror, because they are few. Figure 4.1 underwrites this conclusion: when minimizing the distance error, the two outlying areas are accurately represented. Therefore it is probably more important to minimize distance rather than height, but it is likely that some tradeoff between these two options is the best.

4.1.4 Experiments

The two experiments described in this section are intended to show some of the weaknesses of the decision tree approach and that neural networks can do better. In both experiments, the positions consist of a number of uniformly distributed numbers between 0 and 1. If these features are higher than 0.5 on average, the outcome of the position is taken from a high variance distribution, if they are lower than 0.5 on average a value from a low variance distribution was taken. This task is designed to be difficult for decision trees, because the average featureheight depends on most of the features.

The splitting questions the decision tree was allowed to make are of the type 'is feature X higher than 0.5 or not?'. If feature X had been queried before and we are now in the subtree with only high values of X, 0.5 was changed into 0.75, etc. We could have given the decision tree the freedom to split the data on any height of X, but this would have come at considerable computational expense and, given the task at hand, would not have helped anyway.

We worked with 20.000 examples, and the decision tree was allowed to split data until the datasize at a node went below 200 (about the same number as Baum and Smith), after which K-Means was run on the remaining dataset. The results of K-means was used as the probability distribution for that category. Because after each question the dataset is halved, only 6 questions can be posed until the dataset is reduced from 10.000 to under 200. This means that if there are more than 6 features, the decision tree cannot look at all of them. The importance of this is that the required size of the original dataset grows exponentially with the number of features, if all features are relevant. In order for the decision tree approach to be succesfull in environments with many dimensions, it must rely on the human ability to extract a managable amount of relevant features out of all those dimensions, without losing valuable information. In this constructed example it is easy to do just that, of course: just reduce the positions into a single boolean value, which is true if and only if the features of the position are higher than 0.5 on average. But we do not think it is that easy in most other domains.

In our first experiment, the low variance distribution had a 0.5 probability of being 1 and a 0.5 probability of being -1. For the high variance distribution those values were 2 and -2. Both the decision tree and the neural network could output three locations. The specific neural network approach was the one with the varying locations. Using fixed locations would have been unfair because the environment has fixed locations as well. The network had 5 hidden nodes and used an alpha-value of 0.1, which was reduced by 1% after every pass through the training set. The network was trained over 100 passes, because after 100 passes the network was practically converged. Overfitting did not occur.

The results can be seen in table 4.1. It shows the average error over 5 repetitions of the experiment, with as errormeasure the average of the sum of the distance and height errors. All these differences are significant at a degree higher than 0.0001, except for the difference when 6 features were used, where the degree of significance was 0.0184.

Number of features	6	9	12	15	18
Decision tree error	0.123	0.145	0.200	0.186	0.223
Neural network error	0.059	0.021	0.024	0.033	0.039

Table 4.1: Performance on the discrete distribution problem.

In a second experiment both discrete probability distributions were replaced by uniform ones. This increases the distance error a lot, because the values can now be anywhere instead of at four fixed locations. It makes the recognition

task more difficult as well, because positions in the high variance group will have values indiscernible from low variance values 50% of the time. The learning methods for both neural networks and decision trees remained the same. The experiment was repeated 5 times as well.

Again, the values in table 4.2 show that neural networks can handle this kind of task better. The significance of the difference varied more and the difference was not even significant in the 15 feature case, which is due to an outlier in the neural network results. Neural networks are unpredictable (much more so than the decision trees) and do very badly sometimes. It is easy however to filter out these bad apples. If we would have done so, the results would show larger differences.

Number of features	6	9	12	15	18
Decisiontree error	0.386	0.418	0.428	0.441	0.445
Neural network error	0.360	0.372	0.388	0.403	0.387
Significance	0.0021	0.0001	0.0053	0.0972	0.0008

Table 4.2: Performance on the uniform distribution problem.

4.2 Random Game Trees

4.2.1 Introduction

Normal two-player games such as chess, checkers or othello, have several drawbacks for the experimental scientist. The first is that they can take much time to implement, time that can be better spent elsewhere. The second is that evaluation functions for these games are kept secret due to the commercial value of a good game playing program. This hinders the scientific criterion of reproducibility. Finally, the game trees corresponding to these games are fixed. Controlled variation of certain properties, like the branching factor, is not possible. This makes experimentation more difficult in general.

These reasons were put forth by Korf and Chickering[20] as the reason to use incremental random game trees, which do not have these drawbacks. They were first used by Fuller, Gaschnig and Gillogy[11] and a number of other researchers did also use them to evaluate various aspects of search algorithms (see the paper by Korf and Chickering for further references).

In essence, random game trees are pseudorandomly generated trees. In their basic form, the nodes in random game trees have a fixed number of children (except the leaves, of course), and also have a pseudorandom number associated to them. At each leaf node, the numbers of all its ancestors are added together (in this case, a node is ancestor to itself). The resulting number is the outcome of the game in that position. To get evaluations for internal nodes, for use by a search algorithm, the numbers of the ancestors of that interior node are added together. This makes the evaluation of nodes progressively more accurate towards the end of the game, a property observed in many actual games.

In comparison to other games, they are easy to implement. Variation of the properties of this model is also easy, e.g. the branching factor is just a parameter. And if some extra care is taken in the generation of the pseudorandom numbers, the tree is also exactly reproducible, as will be described in the next paragraph.

There are various methods of assigning pseudorandom numbers to nodes. A program could just assign the numbers of a pseudorandom sequence as they are required by the search algorithm (most programming languages contain one, usually under the name "random"). This makes the tree unreproducible, because the search algorithm decides where the next random number will be put in the tree. This is unacceptable. Another option is to determine the number of the node under a breadth-first ordering and use this number to set the seed of the random number generator. These trees are reproducible, but unfortunately not very random. Korf and Chickering found this out experimentally, and reasoned that the sequence of first numbers of random number generators with successive seeds probably does not have the same properties as the sequence produced by the random number generator itself. The solution they finally adopted was to compute the n^{th} number (the number in breadth-first ordering) of the sequence directly. They also provided an algorithm to reduce the complexity of this operation to $\mathcal{O}(\log n)$, so the final tree could be generated fast enough to be practical and had all the desirable characteristics.

4.2.2 Extending random game trees

Our main interest is to investigate methods that generate probability distributions on the basis of the features of a position. In the above description of random game trees, this is not possible, as for evaluation purposes, different nodes are indiscernible. They all have their associated number generated in the same way, just like their descendants. This is fine for search algorithms that are meant to be domain-independent anyway, but it is not when the learning of the evaluation function is the goal. There just is not very much to learn if there really is only one position. Evaluating search algorithms that make use of more knowledge than a scalar evaluation is impossible as well, as there is no more information available in a node.

Therefore, we have extended the random tree model with nodes that have features. In the simplest case, these features are completely random in each node, but have influence on the numbers associated with its children. For instance, if some feature X is high, its pseudorandom number is somewhat higher on average as well, and a higher number means a better position for the maximizer. Mathematically this looks like equation 4.1.

$$number^n = X + \sum_{i \in I}^{I} f_i^n \cdot I_i + bias \tag{4.1}$$

where n is the node under consideration, X means a uniformly distributed random variable between -0.5 and 0.5 (which is generated by a pseudorandom number generator initialized with a hash of the features of the node, so it is

reproducible), f^n is n's feature vector, and I is the influence of each feature on reward, and $bias$ is a normalization factor to ensure the number is 0 on average over the set of all nodes.

These correlations between features and numbers have an influence on the value of a node, and current methods in learning evaluation functions (see [3] for an application of such methods) can estimate this very well, as we confirmed experimentally.

Although the positions are now different with respect to their values, their probability distributions are still the same. A way of changing this is to make the range of the numbers larger if some feature is high. This extends the equation to

$$number^n = X \cdot (\sum_{i \in D}^{I} f_i^n \cdot D_i + bias_{deviation}) + \sum_{i \in I}^{I} f_i^n \cdot I_i + bias \tag{4.2}$$

where D is the influence of each feature on the size of the interval and $bias_{deviation}$ is a baseline value, to ensure a minimal range.

This has no direct influence on the evaluation function, as nodes with large intervals can be either very bad or very good. Nevertheless it would be valuable information for a search algorithm, because the evaluation function is less accurate in such nodes.

In the previous cases, the features of a position only had influence on the numbers of their children. These children themselves have completely random features again. Therefore any influence a feature has is limited to only one number, while the outcome of the game is decided by lots of numbers. Because of this small influence the use of such knowledge to a search algorithm is quite limited. What is needed are features that have influence over a long range of numbers.

To do this the features of a position are not uniformly distributed, but instead depend on the values of the features of its parent. One simple way to do this is to take the value of the parent feature and add a small, uniformly distributed random variable to it. This looks like equation 4.3.

$$f_i^{child} = f_i^{parent} + X \cdot deviation \tag{4.3}$$

Deviation is the amount a feature can drift between moves; e.g. if deviation is 20, the feature can increase or decrease by up to 10. Note that the random variable X is another one than those used in previous equations. It comes from a sequence of random variables associated with each node, so for accuracy it could be subscripted with an index, but for clarity the subscript is omitted.

The changes to the features of a position and its number are linearly dependent of other features. This is not necessarily so. Any kind of update mechanics can be used, which all have a specific influence on the value and the uncertainty of the value of a position. In fact, any game tree can be encoded in this framework, although this would be overly laborious for most real-life games.

The main point however is that the probability distributions of positions can be easily manipulated.

4.2.3 A specific model

Having explained the general layout of random game trees, we now present a specific instantiation. We used it to assess the accuracy of our spike generation methods described in the previous section, and used it to play games between alpha-beta and Bayesian search as well.

In this specific type of game tree, each node in the tree contains 20 features, each ranging from 0 to 1000. The first feature always starts at 0, and increases by 1 after every move. This feature therefore acts as a move counter. Features 2 through 5 are directly correlated with the quality of the position: each feature is multiplied with a constant and the result is added to the number of the position. A correction factor is applied so that if all features would be 500, the next number would be 0 on average. After each move, a number between [-50, 50] is added to each feature, with different numbers for different features.

The rest of the features do not have any direct correlation with the reward. Instead, they are designed to influence the uncertainty of the position. This is done by letting these features decide whether the next position has a chance to be completely random. If the sum of these features is more than 500 times the number of features (i.e. higher than average), this randomization has a 60% chance of occuring, otherwise only a 10% chance. This results in a clear division between certain and uncertain positions.

But perhaps some update equations and pseudocode can make this all a bit clearer. In the next equations, the subscript on f denotes the number of the feature, and the left hand sides always denote values assigned to the new position, while the right hand side is associated with its parent. $Rand(a, b)$ denotes the next pseudorandom number from the sequence, scaled to an [a,b] interval, and rounded if necessary. r denotes the number given to a child.

$$
\begin{aligned}
r &\leftarrow rand(-0.5, 0.5) + (f_2 + f_3 + f_4 + f_5) * 0.001 - 2000 * 0.001 \\
f_1 &\leftarrow f_1 + 1 \\
f_x &\leftarrow f_x + rand(-50, 50) \quad \text{for } x \geq 2 \text{ and } x \leq 5 \\
f_x &\leftarrow f_x + rand(-100, 100) \quad \text{for } x \geq 6 \text{ and } x \leq 20
\end{aligned}
$$

```
if 1/14 (sum_{i=6}^{20} f_i) > 500
    if rand(0,1) < 0.6
        f_x ← rand(0, 1000)    for x ≥ 2 and x ≤ 20
else
    if rand(0,1) < 0.1
        f_x ← rand(0, 1000)    for x ≥ 2 and x ≤ 20
```

To give a sense what kind of game tree this generates, we will now consider some specific positions. Take for instance a position where features 2-5 are on average 800, and features 6-20 are on average 300. For all actions, the reward lies in the [0.7-1.7] range, so this is a good position for the maximizing player. Most likely, the child position is a good position for him as well, because the sum of the features 2-5 is not going to change very much. The sum of the features 6-20 does not change very much as well, so positions will only have a 10% probability of being totally random for a while. There is of course a low chance that this will happen anyway, but it is highly likely that it can be avoided, as the maximizer has 3 actions at his disposal. These properties carry over to all children, which makes it a very good position for him, because he can be sure to keep earning reward for some time.

Now consider a position where features 2-5 are the same, but features 6-20 are on average 700. The first action taken will yield a high reward, but there is a high chance the next position will be completely random. If this happens, there is a 50% chance a quiet position (with low chance of random children) is reached, because the position gets random features and half the positions have a sum of their features that is less than average. This can be either good or bad, but in the other 50% of the cases, the next position will have a high chance of random children. In this case, the search can make a big difference in move quality.

The difficulty in estimating the value of a position with high chance of random children can be illustrated by the value function we generated on the basis of this model. Although high-chance nodes and low-chance nodes occurred equally often, the high-chance nodes were responsible for more than 80% of the generalization error. This shows that there is really something useful to learn for our spike estimator.

4.2.4 Experiments

To get data for our spike estimators, the following procedure was followed. 10.000 random positions were generated, and an 8 ply search was conducted on them. The total of the numbers of the ancestors was used as the evaluation of a position, and the final value of the search was stored as the real value of the position. A neural network value function was fitted on this data. Note that this looks a lot like temporal difference learning (see [27]), a well-known procedure for generating value functions.

The resulting value function can be seen as a predictor of the numbers that can be encountered during the next 8 plies. We deemed this to be too short-sighted: it may lead a searcher to choose a move that has a good value for the next 8 moves, but is bad for the rest of the game. To make the evaluation function more far-sighted, the entire procedure was repeated in a way. We again generated 10.000 positions and conducted an 8 ply search on them, but this time the real value was the total of the numbers of the parent positions *plus* the value of the leaf position, as calculated by the previous value function. In a sense, the estimated effect of another 8 ply search was added on top of the 8 ply search.

This two-step approach to estimate the 16-ply value is necessarily less accurate than directly doing 16-ply searches, but the cost of such searches are prohibitive.

With this data, a spike generator with the following features was learned. It had a neural network to estimate the expected outcome of a position and a separate network to estimate deviations from the expected outcome. This separate network had 20 preset spikelocations, spread out uniformly over an interval that covered almost all occurring deviations.

With the data from an actual game tree, we again compared the performance of our neural network approach with the decision tree approach. The data from this model is reminiscent of the example we used to compare them in the first place: both models have high variance positions and low variance positions. There are differences as well, as the random game tree data tends to be very skewed. The reason for this is that the player on move can choose the best position of three, and taking the best result of three normally distributed random variables is in itself not a normally distributed random variable, but has a quite skewed distribution instead. Another difference is that the data from the random game trees is much more chaotic.

Because the neural network had widely varying quality over different learning runs, the following methodology was used. 10 different networks were trained on a set of 4.000 examples. 3.000 different examples were used to estimate the performance of each network on the actual problem (known as crossvalidation). The network that did best on this second set was the winner, and its performance was again rated on the last 3.000 examples. This procedure is necessary to assure the performance estimate is unbiased. If we took the performance on the second set as our measure, the performance would be better than warranted, because the sampling error is biased. Of course we used the same procedure for decision trees, but because they tend to have about the same quality, it did not improve things much.

As actual errormeasure we again take the simple sum of the performance measures proposed in the previous section. With 20 repetitions of the procedure given in the last paragraph, the average performance of the decision trees was 1.44, with a standard deviation of 0.12, and for the neural network 0.97 and 0.12 respectively. This shows the neural network does significantly better.

We also tried matches between alpha-beta, Bayesian search with decision trees and Bayesian search with neural networks, but this turned out to be problematic. Alpha-beta won almost everything. It turned out there were two problems for Bayesian search in the current setup. First, the value function it used estimates the total of the numbers over the next 16 moves. In positions where the numbers are positive, the moves further down the tree are considered better than those close to the root, because there are more numbers incorporated in the evaluation. In positions with negative numbers this effect is reversed, which is just as bad. In essence, this evaluation function makes nodes at different heights in the tree incomparable. Alpha-beta does not have a problem with this, as it never compares evaluations at different heights.

Another problem is the branching factor of 3 that was used. The low branching factor was initially chosen because it makes the evaluation function less

accurate, so that there are significant deviations from it which the probability distribution estimator can learn. However, when the branching factor is only 3, not much selectivity is possible. In order for the selective search to be meaningful, two reasonable alternatives must exist. This means that only 1 branch can be pruned, which is about the same amount alpha-beta can prune.

4.2.5 Further extensions

To overcome these difficulties, we adapted our model. Three changes were made. First, the branching factor was increased to 20. Second, we abandoned the idea of associating numbers with positions. The outcome is decided at the end of the game and the winner of such an end-position is determined by chance, with the precise probability determined by the features of the position. If all the children of a node are endpositions, and each individual child has a chance of only 10% of being a win, there is still a 88% ($= 1-0.9^{20}$) chance of at least one child being won. This means that the player who can make the last move has a tremendous advantage. Because of this we decided to let the games have variable lengths, the third change. One feature was used to implement this. This feature can only increase after each move made, and when it reaches a certain threshold, positions start having a chance of being an endposition. This chance steadily rises until 100%, after which the game ends. Usually games end much sooner though, because already at 10% there are on average two moves leading to an endposition, one of which is probably won.

This new model has two interesting differences with the previous model. First of all, positions can have only two game-theoretical outcomes. At endpositions, the probabilities for either are precisely specified, and for non-endpositions transition probabilities to other positions are precisely specified. This means it is possible to analytically determine the chance for any position that it is a game-theoretic win. Not only is this useful for selecting positions that have approximately equal chances of winning (for creating a fair game between two algorithms), but this measure can be used as an optimal value function as well. Not optimal in the sense that it returns the game-theoretic outcome, but in the sense that it gives the best possible estimate of the winchance that can be given without doing further search.

The second difference lies in the interpretation of probabilities. In the previous model, positions have real numbers as game theoretical value. A spike located at 0.8, with height 0.5, must be interpreted as 'this position has game theoretical value 0.8 with probability 50%'. In our second model, there are only 2 possible game-theoretical outcomes, but our spikes are not confined to those two values. Now a spike located at 0.8 with a height of 0.5 means 'there is a 50% probability for this position that, after doing a search, I will think it has an 80% chance of being game theoretically won'. If we currently think the position has only a 50% chance of being game theoretically won, the rest of the spikes must give chances averaging 20%, otherwise it would be inconsistent.

To create a value function for this game tree variant, we followed the same general procedure as with the previous one. The 8 ply searches were replaced by

4 ply ones, because 8 ply searches are prohibitavely expensive with a branching factor of 20. We first chose to represent the probabilities with spikes at fixed locations, not relative to a general evaluation function. It turned out however that this approach converged on very bad values, for reasons unknown to us. Therefore we returned to the option of training a neural network evaluation function and another neural network describing the expected deviations from the value provided by the first. The neural network evaluation function did quite nice, on average it deviated less than 15% from the winchance given by our optimal value function.

The deviations were more problematic however. There was one obvious flaw in our early experiments, spikes could lie above 1 (or lower than 0), meaning that there is a chance that such a position is 'better than won', which is impossible of course. To fix this, we put those spikes at 1 (or 0) and changed the other spikes to make sure the expected value of all the spikes together remained the same. Positions where the expected value was higher than 1 (or lower than 0) were given spikes at 0.95 (0.05) and 1.0 (0.0) with probabilities 10% and 90% respectively. This made sure there was at least some utility in expanding those leaves, but not too much. All this made sure Bayesian search would not spend too much time deciding which one of two already won positions would be better.

Even with this change in place, Bayesian search still did worse than alpha-beta. To prove that this was not due to a bug in our program, or to a inherent difficulty in this domain for the Bayesian search program, we decided to hand-craft a spike generator. It works as follows. It has 5 spikes for every position. Two spikes are reserved for the extremes, 0 and 1. One of the remaining three spikes is located at the evaluation of the optimal evaluation function. Next, the minimum of the distances between this spike and the extremes is taken. Another spike is located at the optimal evaluation plus half that distance, and the last at the optimal evaluation minus half that distance. The heights are calculated in the following way. The closer a position is to the end of the game, the more height is given to the two extremes. The heights of the extremes are set in such a way that the expected value of them is the optimal evaluation. The rest of the height is divided over the other three spikes, with the middle spike taking 50% and the two outlying spikes both taking 25% of what is left. Figure 4.2 shows this graphically.

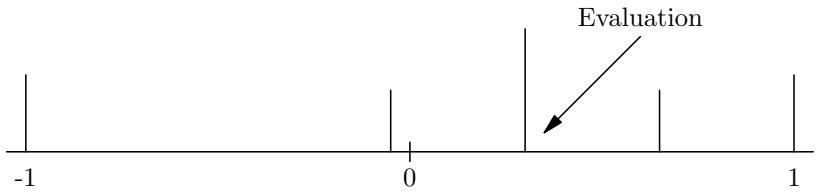

Figure 4.2: Example spikedistribution.

There are several reasons for this approach. The spikes at the extremes can be seen as the chances a definite conclusion can be reached after search. This chance naturally gets higher nearer to the end of the tree. The three other spikes

are representations of the random fluctuations that can occur when searching the tree. Despite the ad-hoc nature, it worked very well, as can be seen in table 4.3.

This table shows the outcomes of 100 games between alpha-beta and Bayesian search with the two spike generation approaches: the neural network and the hand-crafted version. Alpha-beta used the same evaluation functions as Bayesian search, i.e. it used the neural network evaluation function versus the neural network spike generator and the optimal value function versus the hand-crafted version. Bayesian search searched the same amount of nodes alpha-beta did on average. These amounts were 442, 2538, 10648 and 49930 for search depths 3 to 6.

Algorithm	3	4	5	6
Neural network	37	35	47	31
Hand-crafted	59	64	86	89

Table 4.3: Performance of neural network spikes and heuristically generated spikes versus alpha-beta.

The main conclusion that must be drawn from this data is that neural network spike generators in the form used by us do not do well on this task. This raises two important questions. The first is, why do these methods perform so badly, and the second is, does this result generalize to other domains? For the first question we offer an analysis in the next paragraph. To answer the second, we have implemented a chess ending, which is described in the next section. Chess really is a different domain from random game trees: first of all, the game tree is actually a directed acyclic graph, and chess endings have a much richer feature structure which makes it difficult to learn good values.

To understand why the neural network approach performs so badly, it is important to know something about the underlying domain. Most of the positions in the game tree are either clearly won (more accurately: with very high likelihood) or clearly lost. Even if a search starts out from an equal position, the great majority of the searched positions are like that. When such a large amount of positions have nearly but not quite the same chance of winning, small inaccuracies can have large consequences. Bayesian search has more problems with this than alpha-beta, as it assumes there is still useful information to be gained by searching such positions. Alpha-beta just orders the positions differently, which doesn't matter much as most positions are clearly lost or won anyway. Bayesian search is hurt twice by inaccurate value functions, once in deciding which move to take (like alpha-beta) and once in deciding where to search. Apparently the advantage of searching selectively does not overcome this disadvantage in this experimental setup.

4.3 Chess ending

4.3.1 Description

The final experiment was carried out in the game of chess, more specifically, the chess ending of king, knight and pawn on the h-line versus a lone king. There are several reasons this ending was chosen over others. First of all, there are various features that indicate a position is safe, that is, these features are easy indicators that a position is game-theoretically won. One such indicator is the knight protecting the pawn from behind; this is always won no matter the positions of the kings. States that don't have these features may in fact have faster ways of reaching the goal, as was seen in the example in Chapter 2, but they may be lost as well (although the ending technically cannot be lost, we consider failure to win as such). This ensures there is a meaningful notion of 'risk' in a position.

Secondly, there is a natural sequence of subgoals to win the ending, at least for humans. These subgoals are

1. Protect the pawn with the knight,
2. Move the king to the pawn,
3. Move king and pawn to the 6^{th} row (any further and you will stalemate the enemy king),
4. Move the knight to threaten to attack the field of promotion,
5. Move the pawn to the 7^{th} line; if the king moves away, promote him, if he blocks the pawn, checkmate with the knight.

It could be interesting to see to what extent the various algorithms follow this human line of play.

Thirdly, the size of the total ending is small enough to afford the creation of a database of optimal evaluations, yet it is not so small as to make the task trivial.

Lastly, the fact that it has been studied before, by van den Herik[29], makes it more interesting as well. His knowledge-based approach yielded a sufficiently accurate value function for a 3-ply selective search to be enough to win, although a 5-ply search would probably win faster in most cases.

As the third advantage suggests, we made a database of the ending. Our value functions were learned on the basis of this database. This will be impossible in other, more complex cases, because if creation of a database is possible there will be no need for an evaluation function, let alone a search algorithm. But when it is not possible to create a database, temporal difference methods are generally able to make a good evaluation function anyway. By using the database instead, we made sure the outcomes of our experiment are not attributable to the temporal difference method used. It saved a lot of time as well, computer cycles and brain 'cycles' alike.

The database contains the number of moves it takes under optimal play to win a position or marks it as lost if black can prevent a win (these positions are technically drawn, but we refer to them as lost). Using this database we want to train a neural network evaluation function for this ending. There are three things to consider when building such a neural network: the encoding of the chess position into a representation suitable for a neural network, the values of different positions and the encoding of these values in the output of the network.

4.3.2 Experimental setup

We start with the input representation of a position to a network. We chose to represent all fields on the board as different inputs. This means that each piece needs 64 neurons, except for the pawn, as the pawn can only have 6 legal positions at the h-line. We did not handle promotion, when promotion had taken place the correct game theoretical result was available to the algorithm. This results in a total of 198 inputs, but only 4 of them are active at any single time. Implemented correctly, only these four active neurons use up computer time, so applying the network to an input or teaching it some input-output correlation is not very expensive. The only drawback is that the number of learning episodes the network needs is higher than with smaller input encodings, as there are simply more weights to set.

Another possible input encoding is to represent the coordinates of a piece. In this case each piece would need only 16 neurons, 8 for the x-coordinate, 8 for the y-coordinate. In principle, such a network can learn the same functions as the first, but in practice the generalization behaviour is very different. As an example, consider a position with the white king at h2, which is considered good. In our first representation, this generalizes only to other positions with the king at h2, i.e. the evaluation of those positions tends to increase. In the second representation this generalizes to all positions with the king on the h-line and to all positions with the king on the 2^{nd} line. Whether this is a good thing depends on the size and the nature of the actual correlations between these positions. If most of the positions with the king on the h-line are good, it is a good thing. If only the positions with the king on h2 are good, but those with the king on h3 are bad, it would be bad.

It would be interesting to see if these correlations existed, but we did not look at that. The main reason to use the first representation is that it assumes the minimal amount of prior knowledge about the problem. If there are important correlations in our data, our neural network should learn them itself, instead of through a good input representation.

We now move on to values for different positions. Using the number of moves until a position is won would not be a good option, as neural networks do not handle large numbers as inputs or outputs well. This would make it difficult for a network with one output representing the value of the position. To make such evaluation functions possible the following scaling was used: a position won in a single move would have an evaluation of +1. For each extra move required to win, this value was decreased by 0.03. The longest optimal win takes 44 moves,

and so the position it starts in would have an evaluation of -0.29. This is still a long way away from the value given to a lost position, -1.

With the scaling of the output values fixed, the way probabilities of different outcomes are handled needs to be decided. We used the approach of fixed spike locations, not relative to a separate value function. As there are not so many different possible values in this ending (won in 1-44 moves, or lost), we used one output neuron for each of them. This output was then clustered back to 3 outputs by K-means. When testing this probabilistic value function versus a normal value function, it turned out to be roughly as accurate: where the normal value function had an average mean squared error of 0.10, the probabilistic variant had about 0.11. We chose not to compare the quality of the spikes with a decision tree approach. To build the decision tree would have cost us much time, and the resulting comparison would be meaningless anyway, as it totally depends on the amount of effort put into the decision tree.

We did not use all positions to train our evaluation function. Not only would that be too time consuming, but it would also greatly overrepresent the easily lost or won positions. In order to get a more representative set, the following procedure was used. First all the won positions were put into groups according to the number of moves it required to win them. From each of these groups, a thousand positions were taken. All of these positions are won, but lost positions are also needed. To generate these, all positions with white on move were taken and a move was selected which led to a lost position. This results in a set of lost positions with black on move, but it is not possible to do use the same procedure to get lost positions where white must move. After all, in won positions with black moving all moves must lead to a won position, otherwise the position was not won in the first place. Therefore we also generated a set of lost positions two moves away from a won white position, so white has his share of lost positions as well. With this set of positions our evaluation functions were trained.

4.3.3 Comparison between alpha-beta and Bayesian search

We tested the relative performance of alpha-beta and Bayesian search. To do this we used a thousand positions requiring between 20 and 40 moves to win. Both algorithms had to play against the database in these positions and used the same evaluation function. They had a lenient 60 ply move limit, so the algorithms could make quite some suboptimal moves and still win. It turned out they performed very badly. A closer look at the individual games revealed that most of the times the algorithms were repeating moves about 10 plies before the end, roughly the phase where white must move his knight to threaten to attack the field of promotion, h8. The fact that the value function could not give guidance here is probably due to the way the training examples were selected. Although 1000 positions 10 ply from the end where present in the trainingset, the majority of those require white just to move his pawn, because the enemy king is on the wrong side of the board. The cases where the knight is needed are only a minority among the positions won in 10 moves, but it is this minority of positions that need to be played correctly during a game starting with a 20

to 40 ply position.

We could revise our trainingset to accomodate for this phenomenon, but we chose instead to relax the winning criterion. It was no longer required to win a complete game in a position, but the position must be taken 10 ply closer to completion. This decision stems from the fact that only the first part of the game is interesting. Once the king has reached the pawn, all variations are the same in essence.

Depth of search	3	4	5
Alpha-beta	27%	41%	28%
Bayesian Search	67%	59%	71%

Table 4.4: Performance on the KNNp(h)K ending, with the algorithms playing white against optimal play from a database.

In table 4.4 the results of a 1000 games are shown. The listed depths are for alpha-beta, the number of node-expansions available to Bayesian search was set to the amount alpha-beta expanded on average. Over the entire line, Bayesian search performs much better than alpha-beta.

There are a few things to note with this result however. First of all, alpha-beta does not always do better with a deeper search. This is probably due in part to the fact that we use different neural networks for the evaluations of positions with white or black on move[1]. With an odd search depth, black is always on move in evaluated positions, and with an even search depth, it is white. Therefore it seems the evaluation of white positions is better than that of black positions.

The reverse effect present in the Bayesian search data can be explained with the fact that, given a fixed search depth, most of its node expansions occur at the same depth. If this depth is one deeper than alpha-beta, the algorithm would do good on odd depths and worse on even depths, which is what happens.

We must point out however that these analyses are tentative at best. Search algorithms are temperamental beasts and it is in general very hard to figure out why they work the way they work. They do prove however that Bayesian search with neural network spike evaluators are in principle viable. The next question to answer is whether temporal difference methods can train an evaluation function succesfully in this domain, but that question is left for future work.

[1]The reason for this is that the 'on move'-feature is very important for the evaluation. If we would present it to the network as 'just anothe feature', the network would probably not give enough weight to it. For alpha-beta this would not hurt that much, as it does not compare nodes on different levels, but for Bayesian search it would.

Chapter 5

Conclusions and future work

5.1 Conclusions

This thesis made contributions in parametrized probability distribution generation and synthetic model generation. The conclusions that can be drawn from this work will be described in the next to sections.

5.1.1 Generating spikes

In the preceding chapter we have seen some experiments comparing the quality of spikes generated by neural networks and decision trees. The question whether the neural network approach is better we can now answer with a qualified 'yes'. The following issues support this conclusion:

- Decision trees need expert knowlegde to work well. This is both a strength and a weakness. It is a strength because the use of expert knowledge can really improve performance. It is a weakness because expert knowledge can be hard to obtain and hard to implement into the algorithm.
- Our experiments, in random game trees as well as in chess, showed that a neural network does a lot better if no expert knowledge is present.
- Although it is not clear which network topology is best, all the topologies do reasonably well. It is probable that for most tasks a satisfactory topology can be found.
- There is some danger in using our current error measure for assessing the strength of an approach, because this does not always translate to good performance. In our comparison between decision trees and neural networks this does not play a role, however, because the decision trees were effectively uninformed.

- It seems hard to imagine how a complex game like chess can be divided into categories that define their risk. Neural networks can approximate a more general class of functions, so they are bound to have less problems in this respect.

Our conclusion is that neural networks have more potential for complex domains, both due to the inherent problems of decision trees and the fact that neural networks require less tuning.

5.1.2 Random game trees

In the previous chapter we have also seen several versions of random game trees. While they are not important for search algorithms in themselves, in our opinion they represent a methodological advance. They have several advantages over real games, some of which have been mentioned in the literature before, some not. They are:

- It is easy to create domains with easy-to-learn value functions or hard-to-learn value functions.
- It is easy to create domains with varying amounts of uncertainty. The actual cause of these uncertainties can be chosen as well. It can be due to really different outcomes and their probabilities (in our first model), but the uncertainties can also represent possible changes in the opinion of the chances of a particular outcome as a result of further search (our second model).
- Several other characteristics of the game tree are also easily changed: the average branching factor, variance of the branching factor, the depth of the tree, etc.
- Under certain circumstances the optimal value function can be computed by dynamic programming. This is not optimal in the usual sense; it does not tell you the game theoretical value of a node, but the expected value without knowledge of the results of a search. In other words, it gives you the best possible evaluation if no search is possible, but it does not tell you everything. In most practical problems a game theoretic optimal value function can not be created. Some of the knowledge can only come from search. Random game trees can formalize this distinction, and the 'optimal' value function is the best that can be done for the evaluation part.

5.2 Future work

As does most research, this thesis raises more questions than it answers. The next paragraphs discuss the questions we have identified.

5.2.1 Errormeasures

The most important theoretical issue is how the inaccuracy of the spikes affect search quality. In general, there are two ways in which these inaccuracies can influence search. The first one is that the area between spikes is not correctly represented, e.g. there are spikes at 0 and 2, but the actual value is 1 most of the time. This effect only changes the ordering of node expansions, and so only yields some inefficiency. The other one is more serious, namely when the real value lies further than the extreme spikes. Such nodes may have no expansion utility while they can be crucial to the search. To avoid this, we want the spikes to be as far apart as possible, but this reduces the efficiency of the expansion ordering. This is a trade-off, and further research is necessary to see what the best option is.

5.2.2 The covariance problem

In Chapter 2, we introduced the 'value backup' problem. It is concerned with the fact that nodes with multiple good continuations ought to have a higher evaluation than nodes with a single, but equally good, continuation. Simply taking the maximum value does not take this into account. Any attempt to cure this problem is bound to be troubled by the dual problem, the problem that multiple good continuations might not be better than a single one after all.

The problem can be naturally described by looking at the evaluations of positions as random variables. If these random variables are independent, the probability product rule[1] is the correct way to calculate the evaluation of the parent. This rule becomes less accurate as the random variables become more dependent however. Only when the variables are fully dependent[2] the standard maximum rule is the correct way to propagate evaluations.

Either situation can occur and the two models we tested can provide good examples for both. In random game trees, if a position has more than one good continuation, there is an almost 0 chance that the quality of these continuations depends on a common successor state. This means the form of the tree does not introduce any dependence among sibling positions. All properties of a position, including whether it is a win or not, are designed to depend on independent random variables. So the random variables of states looking alike are independent so this cannot be a source of dependence among sibling positions as well. Therefore, the random game tree model is an almost perfect example of independence between positions. In random game trees, if a node has three child nodes with a winchance of 50%, the parent has a 87.5% winchance.

The chess example is another story. Dependence must occur here, due to the shape of the tree and some properties of the task. In a typical position, it is

[1] Taking the product of the winprobabilities of the children as the winprobability of the parent, see [23]

[2] Meaning, if we have nodes with various chances of winning, say 0.6 and 0.2, their actual outcomes depend on one underlying random variable. In this example, if its value is lower than 0.2, both nodes are won, if between than 0.6 and 0.2, only the node with 0.6 chance is won, and all are lost if it is over 0.6.

an important subgoal to move the king to the pawn. If the king needs to move sideways to the pawn, it has 3 options, straight, diagonally up and diagonally down. Most of the times, they lead to the same position after a few moves, so the values of these options are almost completely dependent. If one of them leads to a won position, the others do as well, and if one of them leads to a lost positions, so do the others. So if these three positions would have an estimated 50% winchance by themselves, the parent would have a 50% winchance as well, in contrast to the 87.5% in the random game tree model.

This varying amount of dependence, or covariance, among winchances of nodes has led us to call this problem the covariance problem. In its full form the problem is: how should information about covariance of nodes be estimated, how should it be used in value-backup rules and how should it influence our search decisions?

Alpha-beta, Bayesian search and probability product only have a stance on the second problem, how to use the covariance for the value-backup rule. Alpha-beta assumes complete dependence, probability product assumes complete independence, while Bayesian search assumes a kind of middle ground: short-term changes are seen as independent and long-term changes are seen as dependent. Although Baum and Smith showed experimentally that their approach was better than either alpha-beta or probability product, their approach does not deal with the first and third problems at all, so there is still important work to do with regard to the covariance problem.

5.2.3 Work

Bayesian search assumes a 1-ply search is enough to find the actual value of a node. This is of course a convenient fiction. Sometimes more search needs to be done to find the actual value, sometimes less. Palay[22] already noted this issue and called it the problem of 'work', because sometimes more work must be done to clarify a given position.

The term 'actual value' is actually unwarranted here, because it is defined as the resulting value after a search of a certain size. It is clear that different amounts of search can result in different outcomes, so there is no 'actual value'.

One way of looking at this problem is not to associate a single distribution with a position, but a function mapping search depths to distributions. This change would make search decisions much more difficult, because now combinations of search depths need to be considered. For instance, maybe a shallow search of positions A and B cannot influence the move decision, but a deep search of A and a shallow search of B would. Search decisions need to be made on the basis of not just probability distributions, but sets of probability distributions with costs associated to them. This leads to a combinatorial explosion so it is unlikely that the optimal decision can be found efficiently.

So the question becomes, can a usable approximation be found? And if it can be done, how can the functions mapping search depths to distributions be generated? It seems likely the first problem can only be solved in a limited way, by limiting the number of considered search depths to a small number. In that

case the answer to the second question is not very difficult: just create distinct distributions for every considered search depth, in the same way as before.

5.2.4 Training methods

In our current neural network training methods, we have a winner-take all system. If an actual value falls in between two spikes, the closest spike wins and takes all credit. With fixed spike locations, this can lead to bias, e.g. if there are two spikes located at 10 and at 20, but the actual value is 14 most of the time, the expected value would go to 10, which is too low. If the spike at 10 would 'win' for only 60% and the one at 20 for 40%, this kind of bias would not occur. Whether this approach would work for unfixed locations remains to be seen.

In general, our training procedures are up for revision. They seem to work fine, but they are a bit ad-hoc. It should be possible to either justify our methods theoretically or generate other, more theoretically justified, approaches.

5.2.5 Online learning

In our experiments we first obtained a set of training examples, then we train a value function and only then we test the result on the real problem. In many cases it has been shown to be better to start with the real problem and learn on the job, also known as online learning. In our case this has the following specific advantages:

- There is no overrepresentation of some positions. The positions are encountered in the exact proportion as encountered in the real task (by definition).
- The algorithm can learn from information actually found during search, instead of learning values from a database. This gives more information on the usefulness of a search.
- The algorithm can learn from information it found itself, instead of information generated by alpha-beta. This is risky as well, as it may overlook the best variation, but it gives more accurate information of the possible information gained by search.

More research is needed in order to see whether these advantages occur in practice.

Bibliography

[1] Eric B. Baum and Warren D. Smith. A Bayesian approach to relevance in game-playing. *Artificial Intelligence*, 97:195–242, 1997.

[2] Eric B. Baum, Warren D. Smith, and Charles Garret. Experiments with a Bayesian gameplayer. Technical report, NEC Research Institute, Princeton, 1996.

[3] Jonathan Baxter, Andrew Trigdell, and Lex Weaver. Knightcap: a chess program that learns by combining TD(λ) with game-tree search. In *Proc. 15th International Conf. on Machine Learning*, pages 28–36. Morgan Kaufmann, San Francisco, CA, 1998.

[4] Hans Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12:23–40, 1979.

[5] Michael Buro. ProbCut: An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2):71–76, 1995.

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

[7] Guillermo Campitelli and Fernand Gobet. Adaptive expert decision making: Skilled chess players search more and deeper. *ICGA Journal*, pages 209–216, 2004.

[8] W. G. Chase and H. A. Simon. The mind's eye in chess. In W.G. Chase, editor, *Visual Information Processing*. New York: Academic Press, 1973.

[9] Vincent Conitzer and Tuomas Sandholm. Definition and complexity of some basic metareasoning problems. In *International Joint Conference on Artificial Intelligence*, pages 1099–1106, 2003.

[10] Adriaan D. de Groot. *Thought and Choice in Chess*. The Hague: Mouton, 1946/1978. 2nd english edition, first dutch edition published in 1946.

[11] S.H. Fuller, J.G. Gaschnig, and J.J. Gillogly. *An analysis of the alpha-beta pruning algorithm*. Tech. Rept. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1973.

[12] Fernand Gobet. Chess players' thinking revisited. *Swiss Journal of Psychology*, 57:18–32, 1998.

[13] Fernand Gobet and Herbert A. Simon. The roles of recognition processes and look-ahead search in time-constrained expert problem solving: Evidence from grandmaster level chess. *Psychological Science*, 7:52–55, 1996.

[14] G.H. Golub and J.H. Welsch. Calculation of Gauss quadrature rules. *Mathematics of computation*, 23, 1969.

[15] Dennis H. Holding. The evaluation of chess positions. *Simulation and Gaming*, 10:207–221, 1979.

[16] Dennis H. Holding. *The Psychology of Chess Skill*. Hilldale, N.J., Erlbaum, 1985.

[17] David Hooper and Kenneth Whyld. *The Oxford Companion to Chess*. Oxford University Press, 2nd edition edition, 1992.

[18] Heikki Hyotyniemi and Pertti Saariluoma. Chess - beyond the rules. *Finnish Artificial Intelligence Society*, pages 100–112, 1999.

[19] Andreas Junghanns. Are there practical alternatives to alpha-beta? *ICCA Journal*, 21(1):14–32, 1998.

[20] Richard E. Korf and David Maxwell Chickering. Best-first minimax search. *Artificial Intelligence*, 84:299–337, 1996.

[21] David Allen McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35:287–310, 1988.

[22] Andrew J. Palay. *Searching with Probabilities*. Pitman Publishing, 1985.

[23] Judea Pearl. Heuristic search theory: A survey of recent results. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 24–28, 1981.

[24] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on pattern analysis and machine intelligence*, 11:1203–1212, 1989.

[25] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314), 1950.

[26] Yaron Shoham and Sivan Toledo. Parallel randomized best-first minimax search. *Artificial Intelligence*, 137:165–196, 2002.

[27] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, 1998.

[28] Omid David Tabibi and Nathan S. Netanyahu. Verified null-move pruning. Technical Report CAR-TR-980, CS-TR-4406, UMIACS-TR-2002-39, Center for Automation Research, University of Maryland, 2002. Published in ICGA Journal, Vol. 25, No. 3, pp. 153–161.

[29] H.J. van den Herik. *Computerschaak, schaakwereld en kunstmatige intelligentie*. Academic Service, 1983.

Appendix A

Transcript of an introspective report

This appendix contains a transcript of Max Euwe's own desciption of his thought processes while looking at a position. The instruction was to speak out thoughts as soon as they occured. The time taken in this process was 15 minutes.

First imporession: an isolated Pawn; White has more freedom of movement. Black threatens Qxb2. Is it worthwhile to parry that? It probably is; if he takes, then a3 is also attacked. Can White then take advantage of the open file? Does not look like it. Still again: 2. Nxc6 and then by exchange the pawn at a3 is defended by the Queen. Indirectly in connection with the hanging position of the knight at f6 and possibly because of the overburdening of the bishop at e7. But wait a moment: no, Qxb2 is rather unpleasant after all because the Bishop at a2 is undefended. Can i do something myself? Investigate that first: the pieces on f6 and d5 are both somewhat tied down.
Let us look at the consequences of some specific moves.
1. Nxd5, possibly preceded by
1. Nxc6. Then 1... Rxc6 is probably impossible because taking on d5. Black has a number of forced moves, there may be a possibility to take advantage of that. It's not yet quite clear.
Let us look at other attacks:
1. Bh6 in connection with f7 - but I don't really see how to get at it.
1. b4 in order to parry the threat - but then exchange on c3 will give some difficulties in connection with 2...Bb5 - oh, no, that is not correct, one can take back with the Queen.
So far a somewhat disorderly preliminary investigation. Now let's look in some more detail at the possibilities for exchange: 1.Nxc6 or 1.Nxd5 or maybe 1.Bxd5 or maybe first 1.Bxf6.
1. Nxc6, Rxc6; 2. take on d5; for instance 2.Nxd5, exd5; wins a Pawn, but there may be compensation for Black on b2. But better is 2...Nxd5; then 3.Bxd5, Rxc1 is nearly forced, no, is it not, he can play 3...Bxg5 as well. I see no immediate advantage.
1...exd5 is not forced therefore; and even if it were forced you couldn't be quite sure of winning. It's happened before

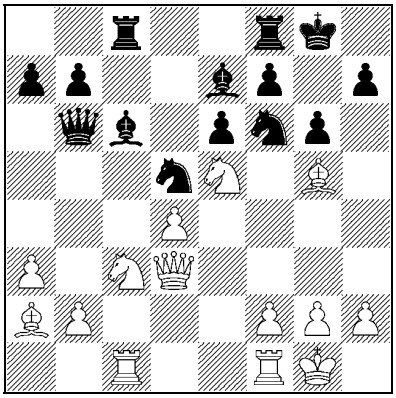

Figure A.1: Position A from de Groot's study.

that such a position proved less favorable than it seemed to be. The point d5 is reinforced by it, that is a disadvantage. 1. taking on d5.

1. Nxc6 at any rate gives the pair of Bishops, if I don't find anything better, I can always do this.

1.Nxd5, Bxd5; is that possible? d7 is free then. 2.Bxf6, Bxf6; 3.Nd7, Qd8 can then be done.

1.Nxd5, Bxd5; 2.Bxf6, Bxf6 will probably yield something. 1...Nxd5 is also possible; maybe better. Then 2. Bxd5, Bxg5 and now there are the possibilities to take on c6, or to play something like f4.; once again:

1.Nxd5, Nxd5; 2. Bxd5, Bxg5 - no, nothing then, 3 Rxc6 does not help any; it is a cute move but at the end of it all everything remains hanging. Something else: 2.Bxe7 - he just takes back. 1...exd5 is very favorable; he won't do that, it needn't be investigated.

1. Nxd5, Nxd5 remains. 2.Bxd5, Bxg5; 3.Bxc6, Bxc1 is then possible. No, can find no way to make anything out of this. 1...Nxd5; Bh6, Rfd8; 3.Qf3 with some threats; if Black now has to play his bishop back to e8, then one gets a good position.

1. Bxd5: this must be looked into. Does that make any difference? 1.Bxd5, Bxd5 is again impossible because of 2.Nd7. That is to say, we will have to look out for 2...Bc4, but that we can probably cope with: the worst that can happen to me is that he regains the exchange, but then I have in any case some gain of time. 1.Bxd5, Nxd5; 2.Same difficulties as just before. No, that is now impossible: 2.Nxd5 wins a piece.

1.Bxd5, Bxd5; 2.Bxf6, Bxf6; 3.Nd7, Qd8. Lets have a closer look at that: 4. Nxd5, exd5 and I'm an exchange to the good: very strong.

1.Bxd5, exd5 is therefore forced. But that's good for white. The knight on f6 is weak, the bishop at e7 hangs - and the bishop on c6 stands badly. On positional grounds one could already decide on 1.Bxd5.

Is there some immediate gain?

1. Bxd5, exd5; it looks bad for black. Probably some more accidents will soon happen. Much is still up in the air. One plays, for instance, 2.Qf3, Defending the knight on f6 is not so easy; 2...Kg7 looks very unpleasant. Yes, I play 1.Bxd5.