Exploring the computational limits of large exhaustive search problems

Diss. ETH No. 14701

Exploring the Computational Limits of Large Exhaustive Search Problems

Dissertation

submitted to the
Swiss Federal Institute of Technology
Zurich

for the degree of
Doctor of Technical Sciences

presented by

Thomas Robert Lincke

Dipl. Informatik-Ing. ETH
born on September 15th, 1969
citizen of
Wyssachen BE and USA

Accepted on the recommendation of
Prof. Dr. Jürg Nievergelt, examiner
Prof. Dr. Jonathan Schaeffer, coexaminer
Prof. Dr. Kai Nagel, coexaminer

2002

Acknowledgments

I would like to sincerely thank the many people who assisted me in working on this thesis.

Jürg Nievergelt as my supervisor made this research possible. He left me complete freedom in my work and taught me how to deliver a message clearly, both in writing and in presentations.

Jonathan Schaeffer as my co-advisor was a competent judge of the technical aspects of this thesis and made many suggestions for improvements. I would also like to thank him for inviting me to spend four weeks as a guest in his GAMES group in Edmonton, Canada.

Kai Nagel also deserves special mention for agreeing to be my co-advisor on such short notice. I particularly appreciate his suggestions for making my writing more understandable.

Martin Müller carefully read and commented on many parts of the final version of the thesis.

Ambros Marzetta always had time for technical discussions. It was one of these discussions, over a pot of Fondue, which led to the highly optimized algorithms for Awari endgame database construction.

Christoph Wirth is a great programmer and deserves the title of Implementor Maximus. As a friend he is always good company and he often helped me to keep smiling even when times were hard.

I would also like to thank those who provided me with game engines for the opening book experiments: *Martin Fierz* (Checkers), *Alvaro Fussen* (Othello), *Ralph Gasser* and *Jörg Kreienbühl* (Nine Men's Morris), *Robert Hyatt* (Chess), *Martin Müller* (Amazons) and *Jack van Rijswijck* (Hex).

The following former and present members of Jürg Nievergelt's research group helped in creating a stimulating and pleasant working athmosphere: *Silvia Ackermann*, *Silvania Avelar*, *Adrian Brüngger*, *Paul Callahan*, *Michele de Lorenzi*, *Ralph Gasser*, *Werner Hartmann*, *Reto Lamprecht*, *Fabian Mäser*, *Ambros Marzetta*, *Martin Müller*, *Matthias Müller*, *Lourdes Pena*, *Nora Sleumer*, *Vincent Tscherter*, *Christoph Wirth*.

Finally, I thank my family and all other friends for their moral support.

Contents

Abstract

Many search problems are so large that their solution on a computer requires huge amounts of main memory or years of computation time or both. One way to get to the solution of such a problem is to sit back and wait until computer hardware with sufficient performance becomes available. Another way is to seek algorithmic improvements which expand the computational limits for search problems with current hardware.

In this thesis we present several such algorithmic improvements. As a testbed for these new algorithms we use the domain of two-player zero-sum games with complete information.

The main contributions of this thesis are:

- An improved retrograde analysis algorithm which improves caching performance of the endgame database construction in Awari by several orders of magnitude.
- An improved expansion strategy for the construction of opening books. The selection of nodes for expansion is not only based on the nodes value but also on the nodes depth in the book. As a result a computer player is much harder to be thrown out of book.
- An improved position-value representation for opening books. The additional values `at-least-draw`, `at-most-draw` and `cycle-draw` propagate more information through the game graph than the traditional game-theoretic values `win`, `draw` and `loss` alone. As a result more positions are solved without additional search and opening book expansion is improved.

All these techniques have been implemented and opening books have been constructed for several games. We also implemented Marvin, the current computer Awari world champion.

Kurzfassung

Viele Suchprobleme sind so gross, dass man für ihre Lösung sehr viel Hauptspeicher oder sehr viel Rechenzeit oder beides benötigt. Eine Möglichkeit solche Probleme zu lösen besteht darin, dass man wartet bis Computer mit genügend grosser Rechenleistung zur Verfügung steht. Eine andere Möglichkeit besteht darin, dass man durch algorithmische Verbesserungen die Menge der sinnvoller Zeit berechenbaren Probleme erweitert.

In dieser Dissertation werden mehrere solche algorithmische Verbesserungen vorgestellt. Als Testumgebung für die neuen Algorithmen wurde das Gebiet der Nullsummenspiele mit perfekter Information ausgewählt.

Die Hauptbeiträge dieser Arbeit sind:

- ein verbesserter Algorithmus für die Rückwärtssuche, welcher durch verbessertes Caching die Anzahl Festplattenzugriffe bei der Berechnung von Awari Endspieldatenbanken um mehrere Grössenordnungen reduziert.

- ein verbesserter Algorithmus zur Berechnung von Eröffnungsbibliotheken. Die Auswahl des nächsten Blattknotens für die Expansion erfolgt nicht nur aufgrund des bisherigen Wertes dieses Blattknotens, sondern auch in Abhängigkeit seiner Tiefe im Baum.

- eine verbesserte Methode zur Darstellung von Positionswerten in Eröffnungsbibliotheken. Durch die zusätzliche Verwendung der Werte `at-least-draw`, `at-most-draw` und `cycle-draw` wird mehr Information durch den Spielbaum propagiert als wenn man nur üblichen Spieltheoretischen Werte `win`, `loss` und `draw` verwendet. Dadurch werden ohne zusätzliche Suche mehr Positionen gelöst und die Expansion der Eröffnungsbibliothek wird verbessert.

Alle beschriebenen Techniken sind implementiert worden und es wurden Eröffungsbibliotheken für mehrere Spiele berechnet. Ausserdem entstand Marvin, der gegenwärtige Weltmeister im Computer Awari.

Chapter 1

Exhaustive Search

The main topic of this thesis is how to solve large exhaustive graph-search problems. A problem is a *graph-search* problem if its solution requires the traversal of a graph. Unless otherwise stated we will use *search problem* as an equivalent to *graph-search problem*.

A search problem is called *exhaustive* if its solution requires us, in general, to visit all its nodes. An exhaustive search problem is called *large* if its solution would require years of computation time, with current computer hardware and current algorithms and tools.

As it is usually the case with hard problems, our focus will be on how to reduce the space and time complexities of the algorithms used to solve large exhaustive search problems. In theoretical computer science our goal is usually to reduce the asymptotic complexity of a problem. Here we are also interested in reducing space and time complexities by a constant factor if this reduction is sufficient to make the solution of a problem feasible.

This Chapter is an introduction to exhaustive search. Section 1.1 defines a general model of a graph-search problem, which is then used to introduce the basic forward search and backward search algorithms. The two search techniques are compared with respect to memory space and computation time requirements. Section 1.2 shows how a certain class of games can be viewed as a special case of a search problem. Section 1.3 discusses work previously done in this field. Sections 1.4 lists the main contributions of this thesis, and Section 1.5 is an overview of the remaining chapters of this thesis.

1.1 Search Problems

A *search problem* is a tuple (S, N, V), where S is the set of states s, N is the neighborhood operator $N(s) \subseteq S$, $\forall s \in S$, and $V(s)$ is the value propagation function $V(s) : \mathbb{R}^{|N(s)|} \mapsto \mathbb{R}$, $\forall s \in S$. $d = |N(s)|$ is called the *degree* of s, and if $d = 0$ then $V(s)$ is constant and s is called a *terminal state*. Otherwise $V(s)$ is a function of the values of other states, $V(s) = prop_s(V(s_1), \ldots, V(s_d))$, and s is called an *interior state*. The states in $N(s)$ are the *successors* of s, and s is a *predecessor* of the states in $N(s)$. The goal is to find the value of one distinguished state $s_0 \in S$, the so called *start state*.

Example 1.1 shows a small search problem. s_5 and s_6 are terminal states, $s_0, \ldots, s_4$ are interior states, and s_0 is also the start state. The propagation function is $max()$ for states s_0, s_3, s_4, and $min()$ for states s_1, s_2.

$$
\begin{array}{ll}
S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\} & \\
N(s_0) = \{s_1, s_2\} & V(s_0) = max(V(s_1), V(s_2)) \\
N(s_1) = \{s_3, s_6\} & V(s_1) = min(V(s_3), V(s_6)) \\
N(s_2) = \{s_3\} & V(s_2) = min(V(s_3)) \\
N(s_3) = \{s_4, s_5, s_6\} & V(s_3) = max(V(s_4), V(s_5), V(s_6)) \\
N(s_4) = \{s_2, s_6\} & V(s_4) = max(V(s_2), V(s_6)) \\
N(s_5) = \{\} & V(s_5) = 3 \\
N(s_6) = \{\} & V(s_6) = 1
\end{array}
$$

Example 1.1: A search problem.

A search problem implicitly defines a directed graph, with states as nodes and arcs between every node and its successor nodes. We will assume that all nodes are reachable from the start node. Otherwise we redefine the search problem as $S' = \{s \mid s \in S \quad \text{and} \quad s \text{ reachable from } s_0\}$, $N'(s) = N(s)$ and $V'(s) = V(s)$, $\forall s \in S'$. Then s'_0 has the same value as s_0. Figure 1.2 shows the graph corresponding to Example 1.1.

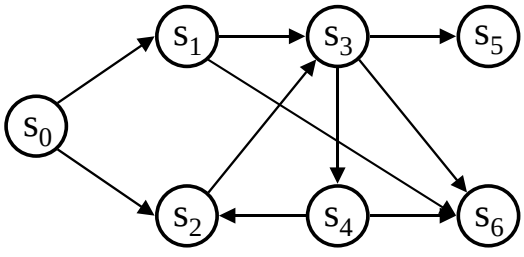

Figure 1.2: Graph representation of Example 1.1

An obvious approach to solving a search problem is to use a *forward search* algorithm. To calculate the value of the start state we first solve all of its successors and then propagate the values. This leads to the following recursive algorithm:

```
int forward1(s) {
  if (isTerminal(s)) {
    return V(s);
  } else {
    return prop_s(forward1(s_1),...,forward1(s_d));
  }/*if*/
}/*forward1*/
```

Unfortunately this algorithm does not work for Example 1.1 because the corresponding graph is cyclic ($s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_2$), `forward1()` runs into an infinite recursion. To avoid this we have to change `forward1()` in two ways: First, we need to detect cycles in the recursion stack. This is done by flagging every state on the path from the start state to the current state. Second, we have to introduce a new state value `cyclic`, different from every other $V(s)$, because the value of a state does not necessarily depend on the value of a terminal state anymore. We also have to adapt $prop_s()$ to accept `cyclic` as one of its inputs. A cyclic search problem can then be solved with the following algorithm:

```
int forward2(s) {
  int value;
  if (isFlagged(s)) {
    return cyclic;
  } else if (isTerminal(s)) {
    return V(s);
  } else {
    SetFlag(s);
    value = prop_s(forward2(s_1),...,forward2(s_d));
    ClearFlag(s);
    return value;
  }/*if*/
}/*forward2*/
```

To solve Example 1.1 we introduce the special value `cyclic` and adapt the propagation function by defining `cyclic` to be smaller than any other value. Then `forward2(`s_0`)` expands the *search tree* shown in Figure 1.3 and finds $V(s_0) = 3$.

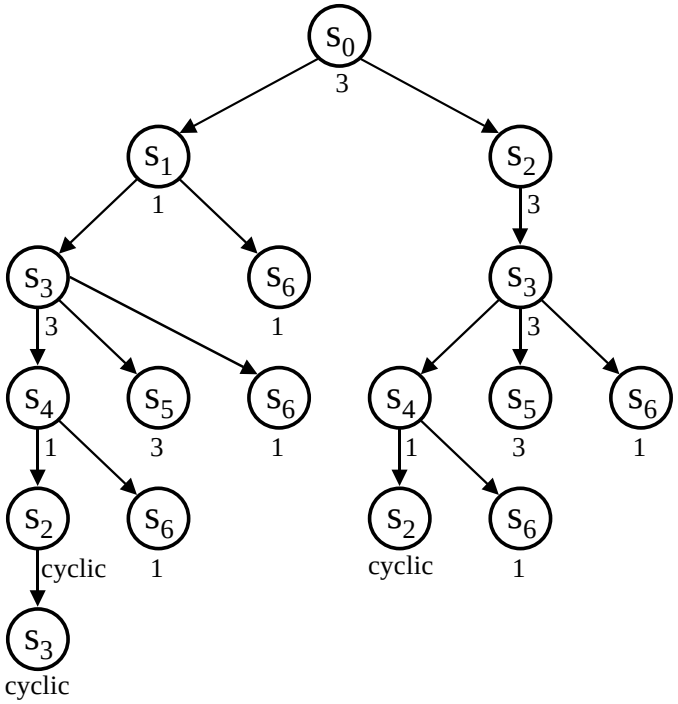

Figure 1.3: Search tree of `forward2()` for Example 1.1. Every node represents a (recursive) call to `forward2()`, with current state number and return value.

Another approach to solving a search problem is to use a *backward search* algorithm (see `backward()` below). The basic idea is that at any point during the calculation only the immediate predecessors of the solved states may be solved in the next step, all the other states can be ignored for the time being. The algorithm starts by initializing an array A of length $|S|$: The values of the terminal states are set to $V(s)$ and the values of the interior states are set to `cyclic`. At the same time the predecessors of the terminal states are stored in C, the set of *candidates*. In the main loop the states are removed from C one by one and checked for a change in value, and if the value did change then it is updated and all predecessors of the state are added to C. The algorithm terminates when C is empty, i.e. nothing can change anymore.

```
void backward() {
  state s;
  set of state C;
  value NewValue;
  array of value A;
  C = {};
  forall (s ∈ S) {
    if (isTerminal(s)) {
      A[s] = V(s);
      C = C + pred(s);
    } else {
      A[s] = cyclic;
    }/*if*/
  }/*forall*/
  while (C != {}) {
    s = ExtractCandidate(C);
    C = C - s;
    NewValue = prop_s(A[s_1],...,A[s_d]);
    if (A[s] != NewValue) {
      A[s] = NewValue;
      C = C + pred(s);
    }/*if*/
  }/*while*/
}/*backward*/
```

`backward()` does not need any special handling for cycles; it is sufficient to initialize the interior states with the value `cyclic`. If a state is not part of a cycle then its value will be overwritten eventually.

Figure 1.4 shows how `backward()` solves Example 1.1. Note that a value might be updated more than once, as happens to the value of s_0 in this example.

To solve a search problem it is in general necessary to evaluate all states at least once, thus we speak of *exhaustive* search problems. Forward search and backward search can both be used to solve search problems, but they differ with respect to time complexity and space complexity. Because they have to evaluate every state at least once, their running time is at least $O(|S|)$, where $|S|$ is called the *size* or *state-space complexity* of the problem. For some search problems there exist search algorithms with expected running times much lower than $O(|S|)$, e.g. alpha-beta search in min-max trees [23].

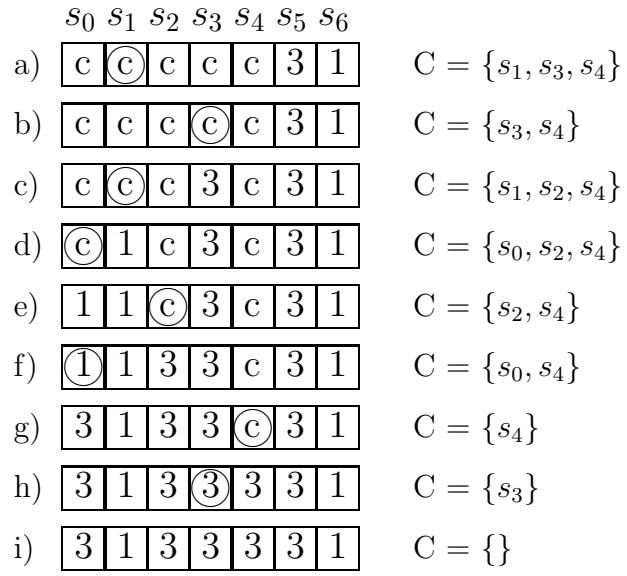

Figure 1.4: One run of `backward()` with Example 1.1. The value 'c' means `cyclic`. Line a) shows the situation right after the initialization, the following lines show the progress at the end of the while loop. In this example, the candidate with the smallest index is chosen for update.

If the graph of a search problem is a proper tree, then forward search will visit every state exactly once. But in the general case some states are reachable from the start state in more than one way. This is called a *transposition* and it means that the corresponding state and all of its successors will be evaluated more than once, thus the number of calls to `forward2()` will be significantly larger than $|S|$. This so-called *search-tree complexity* can be approximated by estimating the average degree D of the interior states and average height H of the search tree. Then the search-tree complexity is approximately D^H. In Example 1.1 the exact value for the search-tree complexity is 17, the number of nodes in Figure 1.3, and $D^H = (16/8)^{(31/9)} = 10.9$. Because `forward2()` needs to store a flag for every state on the path from the start state to the current state, it also needs $O(H_{max})$ space. In the worst case $H_{max} = |S|$.

Backward search does not suffer from the transposition problem. The initialization takes exactly $|S|$ steps. The number of iterations in the main loop depends on how many times the value of a state can change until it reaches its final value. It is often possible to give an upper bound for the number of value changes: In Example 1.1, all states have one of the three values $\{\texttt{cyclic}, 1, 3\}$ and are initialized with `cyclic`. The value of a $max()$

state will only go up, and only when at least one of the successor values becomes larger than its current value, thus it will change at most twice. The value of a $min()$ state will also only go up, and only when all of the successor values become larger than its current value. Therefore this problem is solvable with backward search in at most $|S| + 2|S|$ iterations. In general the runtime is $O(|S|)$ and the required space is also $O(|S|)$.

The memory requirements of backward search may be reduced if the state space can be partitioned into an acyclic graph whose nodes are subsets of the original graph. In Figure 1.5 the states of Example 1.1 have been partitioned into subsets S_A, S_B and S_C, which form an acyclic graph. Now the subsets can be solved independently one by one in the order S_C, S_B, S_A. More generally, if S can be partitioned into subsets S_i, then the required space for backward search is $O(max(|S_i|))$.

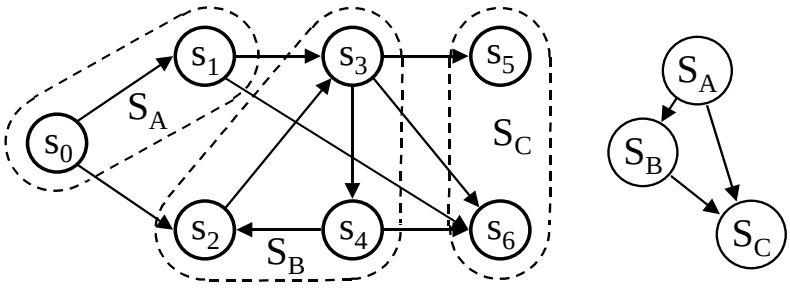

Figure 1.5: The partitioned state-space of Example 1.1.

We conclude that, in general, forward search has a larger time complexity than backward search, but requires much less space during the calculation. Another difference is that backward search returns a value for every state, whereas forward search returns only the value of the start state.

As shown in the previous paragraphs several algorithms exist for solving search problems. However, despite the continuing increase in computation power, we will always be confronted with *large* search problems which are effectively intractable with current technology. By *large* we mean that the running time of the solver would be in the order of magnitude of years, and/or the state-space complexity is significantly larger than the size of the main memory of commonly available workstations.

The main topic of this thesis is how to solve such large exhaustive search problems. From the previous discussion we can conclude that the two main questions are: How can we reduce the space requirements of backward search? And how can we reduce the number of nodes expanded by forward search?

1.2 Games as Search Problems

Games are a special class of graph search problems. The states are defined by the set of positions, and $V(s)$ is defined by the set of moves from position s. The terminal positions have one of the three values `win`, `draw` or `loss`. The propagation function is the so-called *negamax* function: the value of a position equals the maximum of the inverse values of its successor positions. In other words: If the value of any successor of s is `loss`, then $prop_s() =$ `win`, otherwise if the value of any successor of s is `draw`, then $prop_s() =$ `draw`, otherwise $prop_s() =$ `loss`. A cycle in the game graph corresponds to a position repetition in the game. The usual rule for a position repetition is that the game is a `draw` (see Table 4.4), therefore we do not need a new special value because `cyclic` = `draw`.

Although the algorithms presented in the previous section can be used as solvers, games have the *value dominance* property, which can be exploited to significantly speed up the calculation. Value dominance means that it is sufficient to find one successor with a `loss` value to prove that the current position is a `win`, therefore we can ignore, or *prune*, all other successors as soon as we find one with a `loss` value. In the case of forward search this means that the number of evaluated nodes also depends on the order in which the successors are evaluated. Knuth and Moore [23] showed that in a uniform tree the alpha-beta search algorithm only visits $O(\sqrt{N})$ states, N = search-tree complexity, if the successors are evaluated in optimal order. To have optimal order it is sufficient that one of the successors which has the highest value is evaluated first.

Instead of using the *depth-first* algorithm `forward2()` together with local move ordering in every node, we can also exploit the value dominance property by using a *best-first* algorithm. A best-first algorithm stores the whole search tree of the nodes that have been visited so far and, among all the unsolved leaf nodes of that search tree, it chooses the "best" node for expansion, where "best" is measured by a heuristic function which estimates $V(s)$. `forward3()` outlines a best-first search algorithm. Examples of best-first search algorithms are SSS* [49], B* [8], Conspiracy-Number-search [31, 43] and Proof-Number-search [4].

```
void forward3() {
  graph of states G;
  G = s0;
  while (s0 is not solved) {
    follow a path of 'best' moves from s0 to a leaf node s;
    for (all successors si of s) {
      G = G + si;
      if (isTerminal(si)) {
        G(si).value = V(si);
      } else {
        G(si).value = EstimateValue(si);
      }/*if*/
    }/*for*/
    'update the values of all nodes which depend on s'
  }/*while*/
}/*forward3*/
```

In principle, backward search is always possible for games. However, it is practical only if the state-space can be partitioned into small enough subsets, and if the game is *convergent* [4]. A game is called convergent if the size of the reachable state-space decreases during a game. For games, a solved sub-state-space is called an *endgame database* or *tablebase*.

In the previous section, a search problem was solved as soon as the value of the start state was known. Analogously, a game is solved when the value of the start position is known. However, we may also be interested in playing a game, but the knowledge of the value of the start position does not tell us which move is best. Therefore we need a more fine-grained notion of "solved" in games (see also [4, Section 1.5]):

ultra-weakly solved A game is called *ultra-weakly solved* if only the value of the start position is known. This corresponds to the general definition of a solved search problem. For example, there is a well-known proof (John Nash, 1949) that the game of Hex on a nxn board is a `win` for the first player for any n. So far, winning strategies are only known for $n \leq 9$ [61, 62].

weakly solved A game is called *weakly solved* if the value of the start position is known and a strategy is known which guarantees that the first player can achieve that value. This means that for example if a game is known to be a `draw`, then the first player will never lose. If the opponent makes a mistake, the first player does not necessarily know a strategy which achieves the `win`.

strongly solved A game is called *strongly solved* if the best move can be found (within reasonable time) from every position in the game. This is only possible for small games and for subspaces of large games (the subspace of Chess for some positions with 6 pieces and for all positions with 5 or less pieces on the board has been strongly solved [54, 34, 59, 55, 22, 52]).

When we look at games as search problems, then our goal is not necessarily to solve them. The state-space of many games is so large that their solution is still far out of reach. Instead of solving a game we try to find heuristics which approximate the exact values of the positions. But even if a game is not solvable, we can use forward search and backward search as a kind of preprocessing for tournament game playing. For example, in Chess we can use backward search to build endgame databases, which can improve the values found by forward search. From the other end we can use forward search to build opening books, which improve and accelerate game play during the first few moves into a game. This allows us to divide a tournament game into three phases: The opening phase, where only the precalculated (heuristic) values of the opening book are used; the middlegame phase, where values are calculated in real time using a heuristic; and the endgame phase, where exact values are read from the endgame database. The more computation power we can put into the preprocessing of a game, the longer the opening and endgame phases will be. Whenever we manage to close the gap between the opening book and the endgame databases, we will have successfully solved a game. In the rest of this thesis we will focus on algorithms for the first and the last game phase, but we will keep an eye on how the results can be used to improve the winning chances in tournament games.

1.3 State of the Art

Both backward search and forward search have been successfully applied to solving large exhaustive search problems.

Qubic An upper bound for the state-space complexity of Qubic is $3^{64} \approx 10^{30}$. The game has no cycles and contains many forced moves. It was first solved 1977 by Patashnik [36]. He used standard alpha-beta search to try to solve positions with a `win`. If a position was not solvable within a certain time limit and if no forced white move was available, then human expert knowledge was used to choose a suspected winning move, and then the successors were again checked with alpha-beta search. A total of 2929 moves had to be chosen by hand to solve the game.
The game was solved again in 1992 by Allis [4, 2]. He manually chose a set of 195 4-ply starting positions and then solved them one by one using a combination of db-search and pn-search. A total of 3254 non-forced white moves were selected by pn-search to solve the game.

Connect-four Connect-four has an estimated state-space complexity of 10^{14}. It contains no cycles and was independently solved by Allen [1] and Allis [56] in 1988. Both took the forward search approach with alpha-beta search, combined with several sophisticated, game-specific forward-pruning rules. These pruning rules reduced the size of the search tree significantly, and allowed the game to be solved in 300 and 1000 hours of computing time respectively.

Nine Men's Morris Nine Men's Morris has a state-space complexity of 10^{10}, it contains cycles in the middle- and endgame, and was solved in 1993 by Gasser [15]. It was the first game that was solved with a combination of backward and forward search: all positions after the opening phase were solved with retrograde analysis, and then the start position was solved with alpha-beta search in the opening. The forward search was simplified by the fact that the opening phase of the game has a fixed length of 18 plies and does not contain any cycles.

Go-moku An upper bound for the state-space complexity of Go-moku is $3^{225} \approx 10^{105}$. The game has no cycles and contains many forced moves. It was solved by Allis [4] in 1994 with a combination of db-search and pn-search.

Kalah Kalah has a state-space complexity of 10^{13} and contains no cycles. It was solved by Irving, Donkers and Uiterwijk in 2000 [20]. The commercial version of the game has six pits per side and starts with four

stones per pit (Kalah(6,4)). The authors solved all Kalah(m,n) with $m, n = 1 \ldots 6$, except Kalah(6,6). A combination of backward and forward search was used. Forward search was simplified by the fact that the game length is relatively short.

Amazons 5x5 Amazons on a 5x5 board has a state-space complexity of $1.2 * 10^{12}$ and contains no cycles. It was shown to be a first player `win` by Martin Müller in 2001 [32]. The solution was found using forward search together with sophisticated pruning rules.

Moreover, the calculation of endgame databases in Checkers [44] and Awari [30] was a decisive factor for the creation of world-champion-level tournament programs. In Chess, some positions with 6 pieces and all positions with 5 or less pieces have been solved [54, 34, 59, 55, 22, 52]).

An interesting observation about the above list of solved games is that the year when a game was solved and the size of the game do not seem to be related. What one might expect is that with increasing computation power the size of the solvable games would increase accordingly. Instead, it seams that the solvability of a game is strongly influenced by game specific properties that simplify or complicate the solution process:

Forced moves/threat moves: Qubic, Connect-four and Go-moku are called *n-in-a-row* games, where each player tries to place n of his own stones in a connected straight line. If one player already has $n - 1$ stones placed, then the opponent is forced to block the n-th field with one of his own stones. If a player already has $n - 2$ stones placed then creating a threat by adding the $n - 1$'th stone is often a good move. Both forced moves and threat moves simplify forward search because they provide a good heuristic for move ordering, and thus significantly reduce the size of the search tree.

Game-tree complexity Game-tree complexity [4, p.160] is defined as the number of leaf nodes in the solution tree of the initial position of the game. Most solved games have a relatively low game-tree complexity [4, Figure 6.1].

Cycles: With the exception of Nine Men's Morris, all games solved so far are acyclic. In Nine Men's Morris, only the subspace after the opening is cyclic; that part of the game was solved with backward search. We conclude that the existence of cycles makes it harder to solve games.

1.4 Contributions

The main motivation for the presented work was to solve one large exhaustive search problem: the game of Awari. As a result of this quest we present three contributions to the domain of large exhaustive search problems. First, a backward search algorithm, which allows the efficient construction of endgame databases many times larger than the main memory; second, a new opening book expansion strategy, which provides a systematic way for a space and time efficient forward search; and third, a refined value representation for opening books:

Backward search: During backward search it is necessary to store the entire state-space. For large search problems the state-space does not fit into main memory, therefore some portion of the state-space has to be cached on disk. As a consequence the computation becomes much slower because of the slow disk accesses. In Chapter 2 we introduce the new dual-indexing algorithm. It is applicable to any search problem, and in the case of Awari it reduces disk-I/O by three orders of magnitude, with good performance even when only 10% of the state-space fits into main memory.

Forward search: In the past, both depth-first and best-first algorithms have been used to solve games. Our new opening book construction algorithm combines the two search paradigms and takes advantage of the space efficiency of depth-first search and the time efficiency of best-first search. Because an opening book can be viewed as a (partial) solution graph of a search problem, the new algorithm is useful both for game solving and game playing.

Value representation: A `win` is always better than a heuristic value, a `loss` is always worse than a heuristic value, but a `draw` is not comparable to heuristic values. During opening book construction both exact and heuristic values occur in the book, which requires a more refined value representation than what is usually used in forward search.

Further contributions that resulted from this work are:

Drop-out diagrams: A drop-out diagram allows the visual inspection of the quality of an opening book. It shows whether or not the opponent can force us out (or drop us out) of the book at a certain depth with a certain positional value. The diagram provides an excellent tool to measure the quality of book expansion strategies.

Drop-out expansion: When we build an opening book we have to decide which positions should be part of the book and which shall not. Given the current book, the drop-out expansion heuristic selects a "best" position to be included in the book, where "best" is a linear function of the depth of the position and its value.

Although the game of Awari remains unsolved, our work made possible the construction of Marvin, a strong Awari program which is the current computer Awari world champion [30]. We estimate that the solution of Awari would require the completion of at least the 44-stone database, which means that more than 50% of the state-space of about $0.9 * 10^{12}$ positions would have to be enumerated completely.

1.5 Thesis Overview

Chapter 2 describes backward search with Awari as an example. First it is shown how the space requirements during calculation can be reduced to one bit per position, and then the new dual-indexing algorithm is introduced, which allows the efficient calculation of endgame databases several times larger than main memory.

Chapter 3 addresses the problem of automatic opening book construction. Drop-out diagrams are introduced to measure the quality of opening books, and the new drop-out expansion strategy for opening book construction is presented.

Chapter 4 introduces a new model for position value representation in opening books. This refined position value representation helps to give a better estimate on how close a position is to being solved.

Chapters 5 and 6 describe OPLIB, our implementation of an opening book construction tool, and show experimental results with several games.

Chapter 7 concludes this thesis with a summary.

Chapter 2

New Techniques in Retrograde Analysis

This Chapter presents the new dual-indexing algorithm, which significantly improves the disk-I/O efficiency of backward search when the state space of the search problem does not fit into main memory. We show how for Awari the memory requirements of backward search can be reduced from 1 byte per state in a straightforward implementation to 1 bit per state, and then we show how caching efficiency can be improved with dual-indexing when even 1 bit per state is too much to be stored in main memory. Although Awari is used as an example, dual-indexing can be generalized to any game. The rules of Awari can be found in Appendix A.

The Chapter starts with an overview of retrograde analysis in Section 2.1. Section 2.2 discusses the properties of the state space of Awari in relation to retrograde analysis. Sections 2.3 and 2.4 show how retrograde analysis for Awari can be done with only 2 bits per position and 1 bit per position in memory, respectively. Section 2.5 introduces the new dual-indexing algorithm, and Section 2.6 summarizes the Chapter.

2.1 Overview of Retrograde Analysis

The basic retrograde analysis algorithm was introduced by Ströhlein [50], who was the first to calculate endgame databases for Chess. Due to the popularity of the game, most of the research for retrograde analysis was done on Chess. This culminated in the calculation of all 5-piece databases and many 6-piece databases [53, 54, 59, 34, 55, 22]. In other games, the influence of retrograde analysis on the playing strength of computer programs was much more significant. In Nine Men's Morris [15] and Kalah [20], retrograde analysis played a crucial role for solving the games. In Checkers [25, 44] and Awari [30], the endgame databases were a significant contribution to building a world-champion level computer program.

One recurring topic in retrograde analysis research is the main memory bottleneck. The size of the databases increases exponentially with the number of pieces on the board and it is not feasible to calculate databases which are larger than the size of main memory, because the disk accesses incurred by caching part of the database slows the calculation down by two or three orders of magnitude. Several ideas have been suggested and tried out to solve this problem:

Memory space per position To store game-theoretic values (`win`, `loss` or `draw`) at most 2 bits per position are required. To store additional information like the number of moves until the next capture (distance-to-conversion, DTC) or the number of moves until mate (distance-to-mate, DTM) usually requires one byte per position, sometimes even more [55]. Algorithms are known which efficiently calculate DTC databases [53] and DTM databases [60] and which require only one bit of main memory during calculation.

Indexing function An indexing function maps a game position to an index in the database. Depending on the game, an indexing function can be quite complicated, and sometimes it maps unreachable or even illegal positions into the database. See [16] for an overview of indexing functions in Chess.

Support databases In Chess the state-space is usually partitioned into databases with a specific set of pieces. This reduces the maximum size of a database, but we still need to have read access to the databases which are reached by capture or conversion moves. Heuristics have been used [59] to avoid the read access to the support databases, without introducing too many errors in the resulting database.

Distributed or parallel system In a distributed or parallel system the amount of available main memory is equal to the sum of the main memory available to all processors. In a distributed system a disk access is then replaced by network communication. [48] uses a Connection Machine with 32K processors (each with 64K bits of RAM) to solve all 5-piece Chess databases without pawns. [25] use a network of workstations to calculate databases for Checkers. [6] also use a network of workstations to calculate databases for Awari. [57] uses a 16 GBytes shared memory parallel machine with 64 processors to calculate Awari databases. [41] recently solved Awari using a network of 72 workstations with a total of 72 GBytes of memory.

2.2 Retrograde Analysis for Awari

From the rules of Awari it follows immediately that the number of stones on the board decreases monotonically, therefore we can use retrograde analysis to calculate the values of all positions starting with $0, 1, 2, \ldots$ stones on the board, and so on.

Awari endgame databases usually store stone-difference values of configurations instead of `win`/`loss`/`draw` values of positions [3]. A configuration is defined as the distribution of the stones in the twelve pits, ignoring the stones already captured. The stone difference is the number of stones the player to move can capture, minus the number of stones the opponent can capture. For example, the stone difference (or database value) in Figure 2.1 is -2 for South-to-move, which means that with optimal play of both sides, South will capture 8 and North 10 stones. Therefore the value of the position is a `win` for South with a score of 28–20.

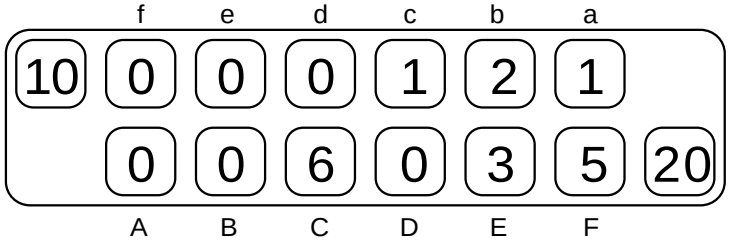

Figure 2.1: The configuration value for this position is -2 (South-to-move). If both sides always play a best move then North will capture 10 of the 18 stones on the board. The position value is therefore a `win` for South with a final score of 28–20.

For Chess endgame databases it is usual to calculate distance-to-win (DTM) or distance-to-conversion (DTC) values. This stems from the fact that it is often hard to win a game even if it is known that a position is a `win`. The DTM and DTC information helps to guarantee a `win` in a finite number of moves. For Awari this additional information does not seem to be necessary. In practical play, it is easy to guarantee a `win` even if only the stone-difference values are known. This is probably a consequence of the low branching factor of the game: most of the time there is only one optimal move anyway, so there is no room for error.

Storing stone-difference values is more efficient than storing `win`/`loss`/`draw` values. The stone difference of a N-stone configuration is always in the range $-N \ldots N$, thus we need $\lceil \log_2(2N+1) \rceil$ bits of storage space for one configuration. With an upper bound of 48 for N, at most 7 bits are required, but to simplify index calculations, 8 bits are used for all databases. To store `win`/`loss`/`draw` values would require $(48 - N + 1) * 2$ bits per configuration ($48 - N$ is the number of captured stones and there are $48 - N + 1$ ways to distribute them between South and North), which is, in general, larger than the 8 bits for stone differences.

The number of configurations with a fixed number N of stones is $\binom{N+11}{11}$, and the total number of configurations with up to 48 stones is about 1.4×10^{12} [3]. This number contains many unreachable configurations. For example, configurations with 47 stones are not reachable because we always have to capture at least two stones. Furthermore, all configurations where all pits of the non-moving player are occupied are unreachable (with the exception of the starting position) because any move empties one pit of the player. There are $\binom{N+5}{11}$ such configurations. In our calculations, we have not excluded unreachable configurations because doing so would make the indexing function more complicated. However the ratio between unreachable and reachable positions grows with N, and it may be that the benefit from excluding them becomes greater than the penalty from the more complex indexing function.

Without the unreachable configurations, we obtain an upper bound for the number of states in Awari of $889,063,398,406$, or about 0.9×10^{12} states. Fortunately, it is not necessary to calculate all these states in one database, instead we can split the state space by the number of stones in the configuration into 48 separate databases (0,...,46,48).

With one byte per configuration, the databases soon become larger than today's workstation memory sizes. For example the 27-stone database has 1.2G states, and the 34-stone database has 10G states.

In other games, compression and symmetry are used to reduce memory requirements. In Awari, no efficient compression algorithm has been found so far; and there is no board symmetry, i.e., we cannot mirror the board by a

vertical or horizontal axis, because the pits form a directed cycle. However, there is a color symmetry because for every configuration with South-to-move there is an equivalent configuration with the same value and North-to-move. Thus it is sufficient to calculate the configuration values for South-to-move and to rotate the board 180 degrees whenever it is North's move.

2.3 Reducing Memory Requirements to Two Bits per Configuration

While it is possible to calculate large Awari databases with small memory, the performance of a caching algorithm drops sharply because of the randomness in the order of database accesses. During a database calculation we often make a sequential pass over all configurations and access all successors to determine the value of the configuration. Ideally, an indexing function should preserve the locality of two configurations with similar indices when we access their successors. For example the first successor of one configuration should have an index near the index of the first successor of the next configuration. In that case caching would show good performance, but, unfortunately, we were unable to find such an indexing function for Awari so far.

If a database is too large for main memory and caching is inherently inefficient, how can we still compute such a database efficiently? The solution is to map the problem of calculating one database with a value range of $-N \ldots N$ to the problem of calculating N databases with three composite value ranges $-N \ldots -k$, $(-k+1) \ldots (k-1)$ and $k \ldots N$, where $1 \leq k \leq N$.

For example, let $N = 5$ (5-stone database). The configurations in this database have a value in the range $-5 \ldots 5$. With $k = 2$ we separate the configurations into three groups: configurations with values in the range $-5 \ldots -2$, configurations with values in the range $-1 \ldots 1$ and configurations with values in the range $2 \ldots 5$.

It is easy to see that, with respect to the propagation rules, there is an isomorphism between these composite value ranges and the values `loss`, `draw` and `win`. A database which only calculates `win`/`loss`/`draw` values can be calculated with 2 bits per position [25]. Therefore we have transformed the problem of calculating one database with 1 byte per configuration into the problem of calculating N WLD(k) databases with 2 bits per configuration. Figure 2.2 shows how the values from the range $-N \ldots N$ are mapped to the values `win`, `loss` and `draw`.

If we use two bits per configuration, the implementation of a WLD algorithm is straightforward (see Figure 2.3). `Preprocess()` sets the values

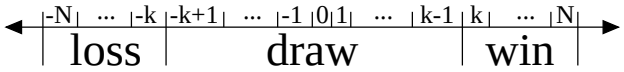

Figure 2.2: The values from the range $-N \ldots N$ are mapped to the values `win`, `loss` and `draw`.

of terminal nodes, propagates values for capturing moves and sets the other values to unknown. `Postprocess()` sets all remaining unknown values to `draw`. It should be pointed out that since these WLD(k) databases can be calculated independently, this splitting allows us to exploit an N-way parallelism. However, there is also redundancy between the WLD(k) databases. For example, if a configuration is evaluated as `draw` for one database, then we already know that it will be a `draw` for all databases with a larger k. The same is true for smaller ks and `win` and `loss` (Figure 2.4). Thus if we calculate the databases sequentially we can use the results of the previous database to initialize the current one. As an additional optimization we precalculate the capture move values and store this information in a support database. This avoids repeated random accesses to the smaller databases.

```
void CalcWLD() {
  Preprocess();
  while (any change possible) {
    for (all configurations) {
      if (value == unknown) {
        if (all successor values are wins) {
          set value to loss;
          set all predecessor values to win;
        }/*if*/
      }/*if*/
    }/*for*/
  }/*while*/
  Postprocess();
}/*CalcWLD*/
```

Figure 2.3: Basic algorithm for WLD database calculation.

With this algorithm, databases 29 (2.3 GBytes) and 30 (3.1 GBytes) were calculated on a 1 GByte RAM machine. But database 31 is larger than 4 GBytes and therefore too large even with two bits per configuration.

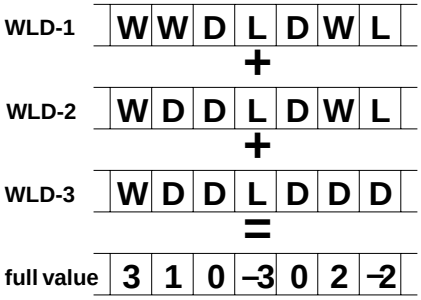

Figure 2.4: An example how the values of a 3-stone database can be represented by three WLD(k) databases. If a value is W in WLD(k) and D in WLD(k+1), then the full value is k. If a value is L in WLD(k) and D in WLD(k+1), then the full value is -k.

2.4 `Win/Loss/Draw` Database Calculation Using One Bit in Memory

For Chess endgame database construction, [53] described an algorithm to calculate distance-to-mate and distance-to-conversion databases which requires one bit per position. We adopted the method and developed an analogous algorithm for the simpler WLD databases.

We use a total of three bits per configuration, two bits on disk and one bit in main memory. The two bits on disk store one of the values `win` (W), `loss` (L), `draw` (D) or `unknown` (U). The bit in memory stores one of the values `win` (W) or `unknown` (U). These two values together determine the value of the configuration, as shown in Figure 2.5. (Only cases which occur during the calculation are shown.)

memory value	W	W	W	U	U	U
disk value	W	D	U	D	L	U
configuration value	W	W	W	D	L	U

Figure 2.5: Calculating the configuration value from memory and disk values. If the memory value is `win`, then the configuration value is also `win`. Otherwise the configuration value is the same as the disk value.

The algorithm is divided into three phases, a preprocessing, a calculation, and a postprocessing phase. Figure 2.6 illustrates the three phases and all state transitions.

In the preprocessing phase we examine every configuration exactly once, determine the values of terminal nodes and propagate the values from cap-

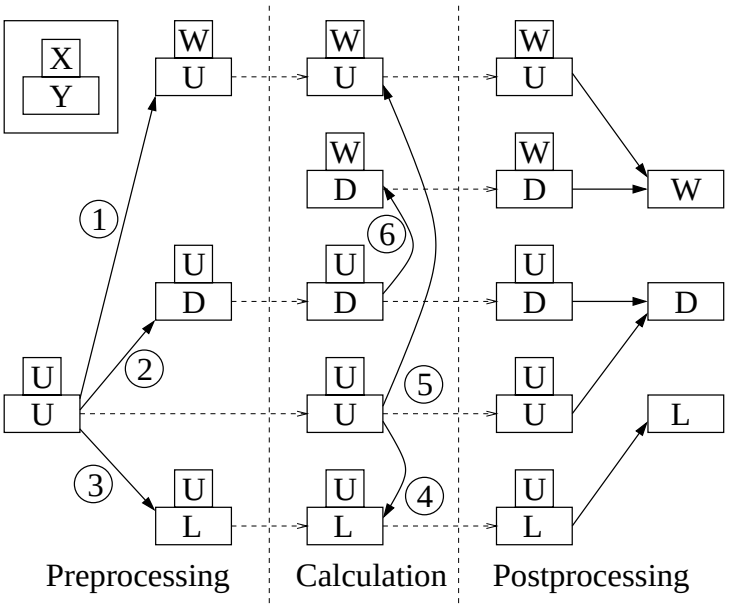

Figure 2.6: State transitions during calculation. Every configuration is represented by a pair of values (X, Y), where $X \in \{W, U\}$ is stored in memory and $Y \in \{W, L, D, U\}$ is stored on disk.

turing moves (i.e., the values from previously calculated databases with fewer stones). This is the only phase where we have to consider values of capturing moves. The following six state transitions might occur.

① If a configuration is a won terminal configuration, or if there is a winning capturing move, we set the memory value to W. (Optionally we might set the disk value to W too, but we do not have to because the memory value overrides the disk value.)

② If a configuration is a terminal configuration with a `draw` value, or if at least one capturing move leads to a configuration with a `draw` value, then we set the disk value to D. In the second case the configuration might have some non-capturing moves that may turn out to be winning moves later. The correct value for such a configuration would therefore be `at-least-draw` instead of `draw`. The reason why we can get away with encoding them both to D will be explained below in connection with state transition ⑥.

③ If a configuration is a lost terminal configuration, or if all moves are losing capturing moves, then we set the disk value to L. Additionally, we set the memory values of all predecessors of the configuration to W (①).

In the calculation phase we make multiple passes over all configurations until no change occurs in one pass. During each pass, we examine every configuration which has a disk value of U. If all non-capturing moves lead to won configurations, then the disk value is set to L (transition ④). Additionally, we set the memory value of all predecessors of the configuration to W (⑤ or ⑥).

Now it also becomes clear why we can ignore `at-least-draw` values: if the configuration ever becomes a `win`, then this will be accomplished by backpropagation, no matter how the configuration was initialized.

In the postprocessing phase we examine every configuration once again, and merge the corresponding memory and disk values into the configuration value.

The most important aspect of this algorithm is that all accesses to the disk values are made in sequential order of the configurations. This sequential access scheme limits disk I/O to a reasonable amount, because disk values can be read in blocks. All the random order accesses caused by the one-ply-forward look and the backpropagation of W values are made in memory.

The algorithm as described above can be applied to any game. For Awari, some improvements are possible. As mentioned in Section 2.3, some of the information in the WLD databases is stored in several databases, redundantly. Thus we can, for example, use the preprocessing phase to set all configurations in WLD(k) to D if they were solved to D in WLD(k-1). This reduces the calculation time significantly. A second improvement is to calculate and store the influence of the capturing moves once before the first WLD(k) database is started. Otherwise we would have to make the same accesses to the smaller databases N times.

With this algorithm, databases 31 (4.3 GBytes), 32 (5.7 GBytes) and 33 (7.7 GBytes) were calculated on a 1 GByte RAM machine. But database 34 is larger than 8 GBytes and therefore too large even with one bit per configuration.

2.5 Beyond One Bit per Configuration

If a database is too large for the main memory even with only one bit per configuration, we have to go back to caching. In Section 2.3 we explained that the Awari indexing function makes any caching algorithm perform poorly. So the question arises as to whether or not there is a way to efficiently calculate databases when caching is required.

It turns out that our one bit algorithm provides us with a prime opportunity to increase the locality of our successor indices. The problem with the indexing function was that we rotate the board after every move. This causes a random distribution of the indices of the successor configurations, because the rotation of the board changes the order in which the pits are encoded in the indexing function.

In the one bit algorithm the value of a configuration is stored in two different places, once in memory and once on disk. During the calculation phase, corresponding values are never accessed together. Instead, for every configuration on disk we read or write the memory values of its successors and predecessors only (see also Figure 2.3). So far we had implicitly assumed that we use the same indexing function for the memory values and the disk values, but we do not have to! Instead we can use one indexing function for the disk values, and another indexing function for the (cached) memory values. For Awari we chose to encode the memory values for North to move (the disk values are still encoded for South to move). This means that the board does not need to be rotated between moves, and as a consequence the indices of the successors are more likely to be close together.

The impact of using this dual-indexing algorithm is shown in Figure 2.7. The number of cache misses was measured for one pass over the 25-stone database. The cache block size was 16 kBytes, an LRU algorithm was used and the cache size varied from 0 to 100 percent of the database size. Dual indexing reduced the number of cache misses by at least three orders of magnitude, with good performance even when only 10 percent of the memory values are actually in memory.

The disadvantage of dual indexing is that the preprocessing and postprocessing phases are slowed down, because there the memory value and the corresponding disk value are accessed at the same time for initialization or to determine the final value of a configuration. For WLD(1) the preprocessing and postprocessing phases together usually take up to 20% of the total computation time. For larger k, they can take up to 50% of the total computation time, because more configurations can be solved during preprocessing and therefore the calculation phase is much shorter.

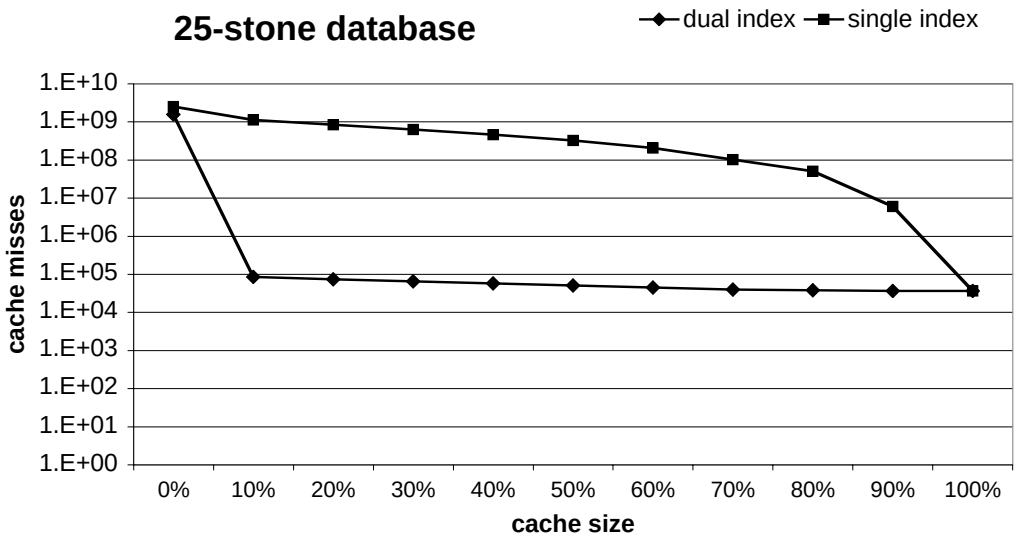

Figure 2.7: The number of cache misses during one pass over the 25-stone database. Cache size is relative to the database size.

With this algorithm, databases 34 (10 GBytes) to 40 (47.6 GBytes) were calculated on a 1 GByte RAM machine.

2.6 The Choice of an Algorithm depends on Memory Size

It is always a challenge to solve large search problems with limited resources. In this paper, we presented algorithms for Awari endgame database calculation which require only two bits and one bit in memory, respectively. An improved version of the one bit algorithm uses dual indexing to calculate efficiently databases with a size of eighty times the main memory. This expands the horizon of solvable endgame databases far beyond the limits of today's memory sizes.

We propose the following rule for algorithm selection: if the database fits into main memory, use a one-byte algorithm. Otherwise, if the memory is sufficiently large to store 1/4 of the database, use the two-bit algorithm. Otherwise, if the memory is sufficiently large to store 1/8 of the database, use the one-bit algorithm. Otherwise, if the memory is sufficiently large to store about 1/80 of the database, use the one bit algorithm with caching and dual indexing.

The Awari databases and their statistics can be viewed online [28].

Chapter 3

Opening Book Construction

This chapter introduces the new drop-out expansion algorithm for the automatic construction of opening books. The algorithm differs from other approaches in that it uses not only position values for node selection, but also takes the depth of leaf nodes into account to make it harder for the opponent to throw the book player out of the book.

For Chess, the most widespread method of opening book construction is the manual or semi-automatic compilation of human opening book knowledge [5, 17, 44, 11]. This requires a Chess expert to select moves that are both favorable and suited to the playing style of the program. Besides the fact that this is tedious work, this method has other drawbacks. For example, there is no guarantee that the human knowledge is free of errors; every move should also be checked by the computer. Another drawback is that the method is only applicable if there is any human opening book knowledge around in the first place.

Our main interest for opening book construction was to create a strong computer Awari program. In Awari there is hardly any human literature on the game, and the computer programs have surpassed the level of human play anyway. The only way to create an opening book was therefore to construct it automatically using the Awari search engine. Sections 3.1 and 3.2 motivate our work on opening books and introduce the basics. Section 3.3 introduces drop-out diagrams, a method for the visualization of opening books. In Section 3.4 the new expansion strategy is introduced and compared to best-first expansion [10].

3.1 Introduction

Games are usually divided into three phases: opening, middlegame and endgame. In any of the three phases, the default action of a computer program is to start a search for a "best" move. Since such a search has to be performed within a limited time, it can only examine nodes down to a certain depth, which means that the calculated value is only a heuristic approximation of the game-theoretic value.

However, in endgames a different approach is possible if the game has the convergence property [4], i.e. the number of pieces on the board decreases monotonically. In this case we are able to construct an endgame database of precalculated game-theoretic values for positions with a sufficiently small number of stones [53]. A computer program can benefit from such a database in two ways: First, the values of the endgame positions can be retrieved in one operation instead of performing a lengthy search. Second, the retrieved values are exact game-theoretic values of the positions instead of approximations, which reduces the error in the heuristic search. The disadvantage is that we need additional space to store the database, and if the database is stored on disk then retrieving database values causes disk-I/O.

For openings an analogous approach is possible. Since the state-space up to a few plies into a game is small, we can precalculate a database (called "opening book", or just "book") of values for positions that are likely to occur at the beginning of a tournament game. If the opening book is stored as a directed graph, with positions as nodes and moves as arcs, then the values can be propagated within the book. This way the computer program not only saves time compared to search, but also obtains better values, assuming the search depth used for precalculation is greater than the one used during play.

Although the benefits of using endgame databases and opening books are similar, algorithms for the construction of endgame databases have received more attention up to the present. The reason for this seems to be twofold: values in endgame databases are exact game-theoretic values, whereas values in the opening book are (mostly) heuristic values. This makes it hard to judge the usefulness of an opening book, and the non-trivial truth of the endgame database provides a basis for data mining. Another reason is that efficient indexing functions for complete enumeration can be constructed for positions in endgame databases, whereas for positions in the opening book no efficient indexing functions exist, due to the fact that the relevance of a position for the opening book stems from the position value and not from the configuration of stones on the board.

3.2 Opening Book Basics

3.2.1 Book Representation

An opening book is represented as a directed graph. The nodes of the graph represent positions, and an arc between two nodes represents a legal move. One node, called the start node, represents the start position, and all other nodes must be reachable from it. If a node has an edge for each of its moves, then it is called an *interior node*, otherwise it is called a *leaf node*.

Every node i has two attributes: the heuristic value h_i and the propagated value p_i. The value h_i is computed by the search engine. For interior nodes, p_i is the negamax value of p_{s_j} of all successor nodes s_j. For leaf nodes, p_i is equal to h_i.

$$p_i = \begin{cases} \max(-p_{s_j}) & \text{for all successors } s_j \quad \text{(interior nodes)} \\ h_i & \text{(leaf nodes)} \end{cases}$$

For book construction it is not actually necessary to keep h_i once i has become an interior node. However, for testing the book expansion strategies, it is useful to be able to compare h_i and p_i during the calculation. Unless explicitely stated otherwise, let *value* mean *propagated value*.

Figure 3.1 shows an example of a node with three successors. To improve readability we will use max-propagation instead of negamax-propagation in the Figures. Node n is a max-node, therefore $p_n = max(p_{s1}, p_{s2}, p_{s3}) = 3$.

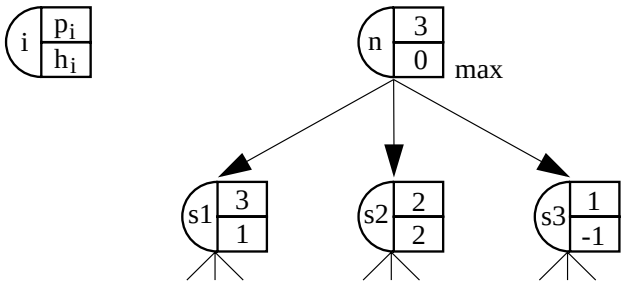

Figure 3.1: Node representation

3.2.2 Book Expansion

When an opening book is created, it contains only the start node. Expansion is done in three steps:

1. Choose a leaf node and add all successors to the book.
2. Calculate the heuristic values of the new successors.
3. Propagate the (heuristic) values up to the start node.

This chapter deals with the first step: how to choose the next node for expansion. Step two, the calculation of a heuristic value, is a topic of ongoing research in the field of computer games. We simply assume that a state-of-the-art search engine is available. Once the value is calculated, it will be negamax propagated through the graph in step three.

3.2.3 Goals

When is an opening book good? Of course, the ultimate goal in every tournament game is to win. Since it will rarely be possible to win a game straight from the book, its main benefit is that during the opening phase of the game search time is saved, which may be used later in the game to outsearch the opponent. Therefore, the primary goal is to maximize the expected number of moves one can play within the book.

Which nodes should we expand to achieve this goal? A naive approach that guarantees that a minimal number of moves can be played from the book is to enumerate all positions at depth 1 from the start position, calculate their values and store them. Next, the same is done for all positions at depth 2, and so on. This will, however, waste a lot of time and space on positions which are very unlikely to occur in a tournament game, because to reach them some player would have to make an obviously bad move. Therefore the secondary goal must be to achieve the primary goal with as few expansions as possible.

3.3 Drop-out Diagrams

As a starting point for the solution of our problem we introduce a graphical representation of an opening book. A *drop-out diagram* shows the depth and value of all leaf nodes that can be reached, under the assumption that the book player only makes best moves and that the opponent is allowed to make any move. For example, Figure 3.2 shows the reachable leaf nodes of a small

opening book. The value of the start node of the book is 0.9. A leaf node with this value is only reached if the opponent only plays best moves too. If the opponent makes a mistake, then leaf nodes with higher values (from the point of view of the book player) may be reached.

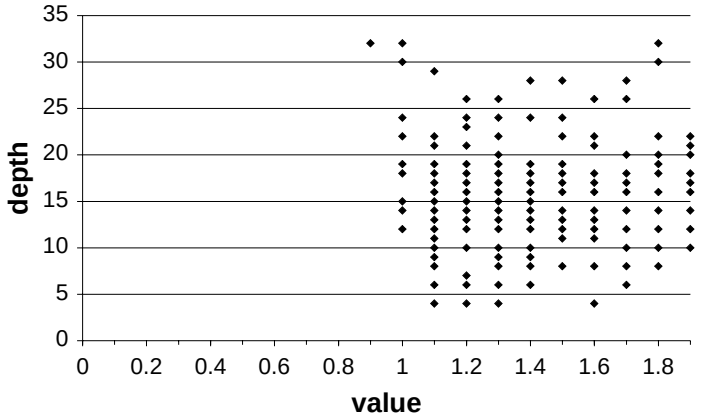

Figure 3.2: Drop-out diagram of a small opening book. The values are shown from the point of view of a max-player, i.e. the greater a value the better it is. In this example the smallest value that shows up is 0.9, which is the value of the start node. Leaf nodes with higher values are only reachable if the opponent (the min-player) makes a mistake.

Each dot in the diagram represents one or more candidate leaf nodes for expansion: Which one should we choose? Or, to reformulate the question, if you were the opponent of the book player: Which leaf node would you try to reach? Obviously, if the opponent only plays best moves, the book player will be able to play at least 32 plies from the book. However, as the plot shows, there is a line in the book which ends at ply 4, with a value of 1.1, which is only 0.2 points worse than the best value. If any opponent of this book player ever finds out about this line, he will probably decide that it is worth to take the risk of a slightly worse position in return for getting the book-player out of book. Such deliberate deviation from best moves is a favorite strategy for humans (and sometimes computers) against computers. Therefore that node at a depth of 4 plies is probably the best candidate for expansion.

3.4 Expansion strategies

The goals in Section 3.2.3 suggest that an expansion strategy should choose nodes for expansion according to the likelihood of their occurrence in a game. Since we can assume that good moves are more likely to occur than bad moves, the most straightforward strategy is best-first. This strategy was implemented in [10]; it is used here as a reference against which we want to compare the new strategy.

For the following discussion of expansion strategies we will assume that the game engine using the book will only play best book moves. This is a reasonable assumption because there is no point in constructing an opening book and then to ignore it. However there are also good reasons to choose other moves now and then, for example to reduce predictability of the game engine.

3.4.1 Best-First Expansion

The rule for best-first expansion is: Expand the leaf node that is reached by following a path of best moves from the start node. If an interior node has more than one best move, choose one of them at random. See Figure 3.3.

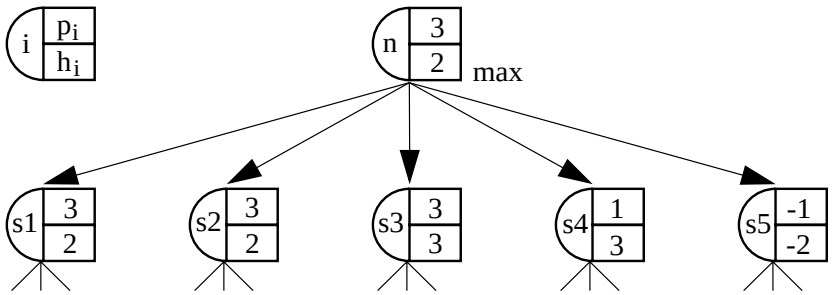

Figure 3.3: Example for best-first expansion. The best value among all successors of n is 3, thus the candidate set for expansion is $\{s_1, s_2, s_3\}$.

This strategy is simple and ignores bad moves, but it has a major flaw: suppose we just expanded the start position of Chess, and the engine returned 0.1 (measured in pawns) for e2–e4, and 0 for all other moves. Expansion will now continue along e2–e4, and it is easily possible that the value for that move will always be > 0, so all other moves from the start node will be ignored forever. This violates the goal of maximizing the expected number of book moves played in a game, because for example the common move d2–d4 by the opponent leads to a position that is not in the book, i.e. we "drop out" of the book.

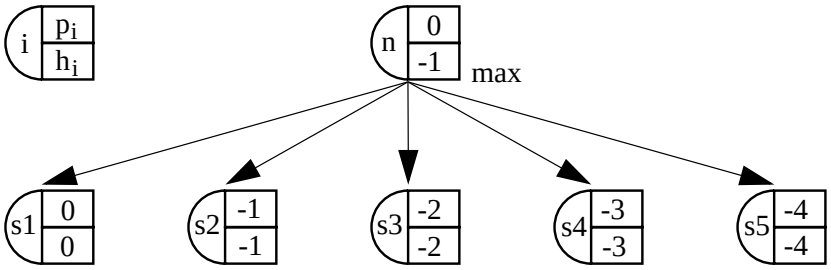

Figure 3.4: Successor s_1 is the only best move.

Of course the assumption that the value of the best move will only increase is weak. But even if we assume that the propagated value of a node will oscillate in an interval around its original heuristic value, the problem remains.

Take, for example, the situation in Figure 3.4, where successor s_1 currently is the only best move. Assume that during further expansion the propagated values remain in the range $[h_i - 2, h_i + 2]$. Then s_2 and s_3 may eventually have values larger than that of s_1, the favorite candidate, and thus become eligible for expansion. However if their values ever become less than -2 they will never be selected again, because the value of s_1 will never go below -2. At some point, a situation similar to Figure 3.5 will be reached, where the values of the alternative moves get stuck just below the lowest value which the optimal move has ever reached. The resulting successor values are misleading, because they are biased to look worse than what they are. This again leads to early drop-out, because the depth of the book after move s_2 will remain shallow, but it is still likely to be played by an opponent.

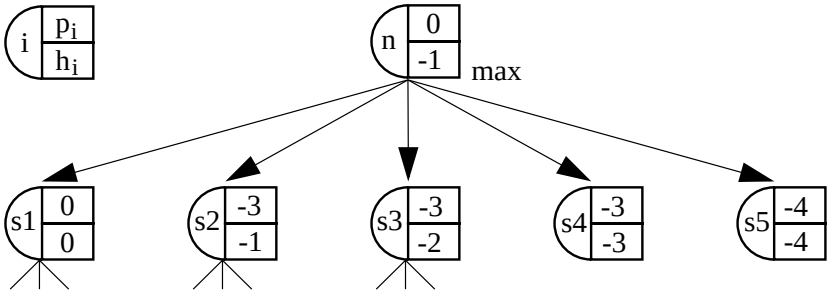

Figure 3.5: The successors s_2 and s_3 are expanded until their values fall below -2.

The heart of the problem with early drop-out is the fact that we use the same engine to search for values for both the first and the second player. We

implicitly assume that the opponent uses the same evaluation function as we do, which is not necessarily true. It would be more reasonable to assume that the evaluation function of the opponent is only similar, and to expand some of the inferior move alternatives in a controlled way.

There is another minor flaw in best-first expansion. Because we make a random choice if more than one best move is available, the probability of choosing a specific leaf node on a best path depends on the number of best moves at any node along the path. This means that the book grows faster along some paths. We could solve this if we first made a list of all leaf nodes reachable with best moves, and then chose one at random, but that would be both time and space consuming. Instead we prefer to have a strategy that solves this by making a series of local decisions.

3.4.2 Drop-out Expansion

The problem with best-first expansion was that in situations as in Figure 3.5 only the best move is considered for expansion, whereas it is not unlikely that an opponent will play the second-best move. To solve this we now consider all moves for expansion, and give each move a priority depending on the depth of the book following the move, and the difference between the best value and the value of the move. A successor has high expansion priority if it is a good move and/or it has a shallow subbook, and a successor has low expansion priority if it is a bad move and/or it has a deep subbook. To calculate the expansion priorities we add two new attributes, epb_i and epo_i, to the nodes.

$$epb_i = \begin{cases} 1 + \min(epo_{s_j}) & \left(\begin{array}{l}\text{for all best successors } s_j,\\ \text{interior nodes}\end{array}\right) \\ 0 & \text{(leaf nodes)} \end{cases} \tag{3.1}$$

$$epo_i = \begin{cases} 1 + \min(epb_{s_j} + \omega(p_i - p_{s_j})) & \left(\begin{array}{l}\text{for all successors } s_j,\\ \text{interior nodes}\end{array}\right) \\ 0 & \text{(leaf nodes)} \end{cases} \tag{3.2}$$

epb_i is the expansion priority for when it is the book player's move (Equation 3.1). It is initialized to zero in leaf nodes, and depends only on the expansion priority of the best successors. The +1 is the depth penalty. It guarantees that shallow nodes have higher priorities.

epo_i is the expansion priority for when it is the opponent's move (Equation 3.2). It is initialized to zero in leaf nodes, and depends on the expansion priority of all successors. Besides the depth penalty (+1), inferior moves

get an additional penalty which depends on the value difference to the best move.

$\omega \geq 0$ is the weight for the difference $p_i - p_{s_j}$ between the best value and the value of successor s_j. The right choice of ω is game specific and depends on the heuristic value resolution, i.e. +1 may mean "one piece ahead" or "0.01 pieces ahead". A low value for ω means higher priority for inferior moves. If $\omega = 0$ then all successors will be expanded to the same depth, regardless of their values. On the other hand, if $\omega \to \infty$ then drop-out expansion degenerates into best-first expansion because only best moves will be expanded.

Figure 3.6 uses drop-out diagrams to show a graphical interpretation of the influence of the choice of ω on the expansion strategy. With $\omega \to \infty$ we get best-first expansion, and leaf nodes are expanded from left to right. With $\omega = 0$ we expand the shallowest nodes first, going from bottom to top. Drop-out expansion allows expansion from bottom-left to top-right at an arbitrary angle.

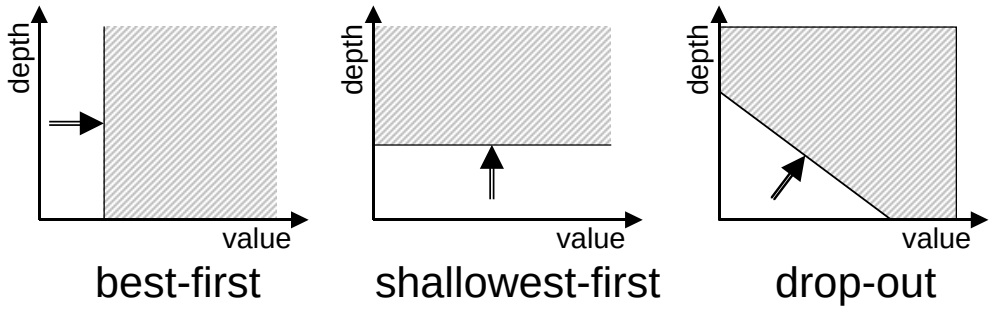

Figure 3.6: The influence of ω on expansion. For $\omega \to \infty$ we get best-first expansion and for $\omega = 0$ we get shallowest-best expansion. Drop-out expansion is the generalized expansion strategy.

Figures 3.7 and 3.8 show two examples of Othello opening books with 10,000 nodes each, calculated with different values of ω.

Figure 3.9 shows the pseudo-code for drop-out expansion. The recursion is started by calling either `CalcEpb(`n_{root}`)` or `CalcEpo(`n_{root}`)`, depending on whether the book should be expanded from the first player's point-of-view or from the second player's point-of-view. The function `Select()` selects a node for expansion. The pseudo-code ignores the handling of exact values.

What do we gain if we use drop-out expansion? Obviously a move that is only slightly worse than the best move will not be ignored forever, even if the best value never decreases. With increasing depth of the best move, the priority for the expansion of suboptimal moves will increase too and

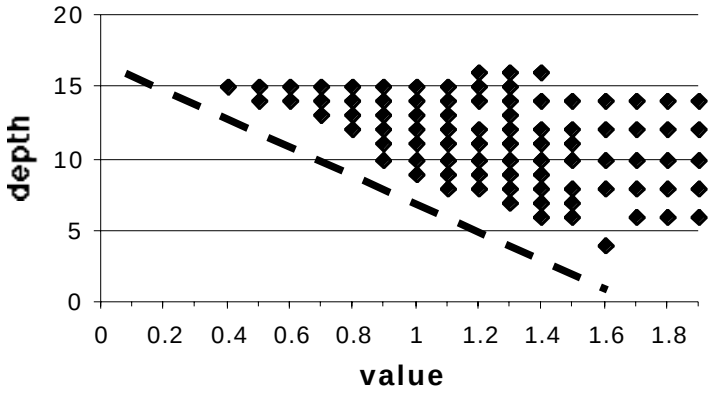

Figure 3.7: Drop-out diagram of an Othello book with 10,000 nodes calculated with $\omega = 1.0$.

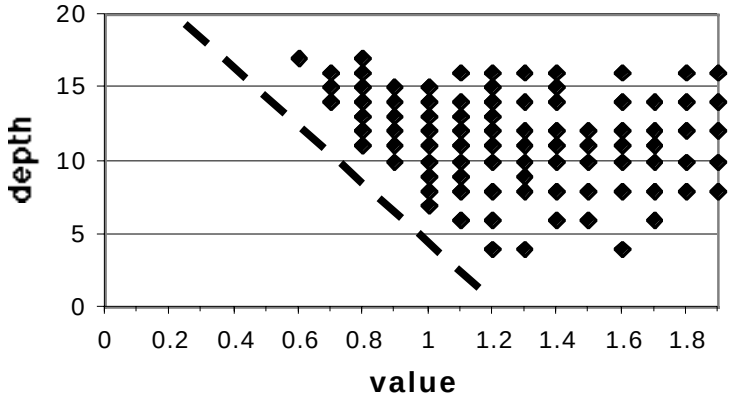

Figure 3.8: Drop-out diagram of an Othello book with 10,000 nodes calculated with $\omega = 2.0$.

```
void CalcEpb(Node n) {
  if (isLeaf(n)) {
    Select(n);
  } else {
    BestValue = BestSuccessorValue(n);
    epb_max = +∞;
    for (all successors s_i) {
      if (s_i.value == BestValue) {
        if (1 + s_i.epo < epb_max) {
          epb_max = 1 + s_i.epo;
          BestMove = i;
        }/*if*/
      }/*if*/
    }/*for*/
    CalcEpo(s_BestMove);
  }/*if*/
}/*CalcEpb*/

void CalcEpo(Node n) {
  if (isLeaf(n)) {
    Select(n);
  } else {
    BestValue = BestSuccessorValue(n);
    epo_max = +∞;
    for (all successors s_i) {
      if (1 + s_i.epb + ω(BestValue − s_i.value) < epo_max) {
        epo_max = 1 + s_i.epb + ω(BestValue − s_i.value);
        BestMove = i;
      }/*if*/
    }/*for*/
    CalcEpb(s_BestMove);
  }/*if*/
}/*CalcEpo*/
```

Figure 3.9: Pseudo-code for drop-out expansion. The function `Select()` selects a node for expansion. The recursion is started by calling either `CalcEpb()` or `CalcEpo()` with the start node.

this will eventually lead to their expansion. For the same reason it will not happen that a suboptimal move gets stuck with a bad value, as was shown in Figure 3.5 for best-first strategy. Thus all the problems observed with best-first expansion have been solved.

An additional benefit from drop-out expansion is that the parameter ω can be used to control the shape of the opening book. The user can choose any shape between full expansion of every line and best-first expansion.

The benefit from drop-out expansion may also be understood as a kind of insurance: if the opponent wants to force a drop-out, he has to pay with a move that is so bad that it has not been considered for expansion yet. If he keeps playing good moves, we will not drop out of the book early. So, at the end of the opening, we have either a good position, or we have saved lots of time, or a combination of both.

3.4.3 Further Enhancements

So far our reasoning has led us to consider not only the values of successor nodes, but also the depths to which they have already been expanded. The subset of leaf nodes considered for expansion was changed to avoid the problems found with best-first expansion. However, the candidate set of leaf nodes changes most radically when the value of the start node changes. It would be of advantage if we could settle that value first.

This is exactly what conspiracy number search [31, 43] can do: Expand a leaf node that is reached by following a path of best moves from the start node and which is most likely to change the value of the start node. Two new attributes need to be added to each node, one to count the number of leaf nodes that have to change to increase the value of the node, and one to count the number of leaf nodes that have to change to decrease the value.

We implemented conspiracy number expansion as an option which can be turned on if the start node is unstable. However the expanded nodes are often in deep remote lines and of limited use for the opening book.

Another enhancement is the use of fractional depth. The depth of the successor nodes is not incremented by 1 as in Equations (3.1) and (3.2), instead a value within the range $0.01 \leq fractd_i \leq 1$ is used as an increment. $fractd_i$ depends on the difference between the best and the second best value. If the difference is large (or if there is only one move), then the fractional depth is small, otherwise large.

This favors the expansion of lines with unique or almost unique moves. The increased expansion of these moves is justified by the fact that they do not contribute to the exponential growth of the book as the other moves do.

3.4.4 Other Considerations

The task of opening book construction would be simplified if we had a good model of the opponent, i.e. if we could predict the opponent's moves with high accuracy. For drop-out expansion we proposed a linear function, $\omega(p_i - p_{s_j})$, for the value-dependant penalty. This is a very simple opponent model, where ω is the estimated similarity of the book player's engine and the opponent's engine. We also considered some non-linear functions, but abandoned the idea because of a lack of efficient implementations.

3.5 Conclusions

We have shown that, for opening book construction, best-first expansion has certain deficiencies. For instance, it may completely ignore alternative moves with values only slightly inferior to the best value, and it has a tendency to stop expansion of inferior moves with a misleadingly low value. In both cases, a lucky or an informed opponent can force us to drop out of the book with only a small penalty for him. Both problems are related to the implicit assumption that the opponent uses the same evaluation function.

We propose a new strategy, drop-out expansion, which, in a user controlled way, also considers inferior moves for expansion. This not only avoids the problems with best-first strategy, but also gives the user the flexibility to control the growth of the opening book between full-breadth expansion and best-first expansion. For best results, drop-out expansion can be combined into a mixed strategy with conspiracy number expansion.

The flexibility of drop-out expansion can also be used to tune a book to an opponent: if the opponent is known to play similar moves, then parameters can be chosen to construct a best-first like opening book. If the opponent is known to play differing moves often, then parameters can be chosen to expand more alternative moves.

Chapter 4

Position-Value Representation

The problem of position-value representation is similar for game search engines and opening books. In both cases, leaf nodes have either an exact or a heuristic value, and in both cases these values must be propagated to the start node.

However, the requirements for propagation speed and value accuracy differ. For a game search engine we want to use a position-value representation which allows for fast propagation, because we need to search as many nodes as possible. We are not interested in the value of the start node; we are only interested in the best move. For example the B* algorithm [8] terminates as soon as one move is proved to be better than any other move, possibly without calculating an exact value for that move.

For an opening book we do not necessarily need the fastest propagation algorithm, because the book is constructed offline. We want to know, as accurately as possible, the value of each node in the book, because the values of the successor nodes are used to guide the expansion of the opening book (see Section 3.4.2).

The question of how to mix heuristic and exact values in a search tree was also addressed in [7]. The author shows that, by propagating bounds on exact values, nodes are solvable even if the exact values of some successor nodes are unknown. He then extends alpha-beta search to propagate lower and upper bounds on the exact value in addition to the heuristic value.

Section 4.1 discusses the problem of incomparable values and how it is dealt with in search engines. In Section 4.2 we solve this problem for opening books by introducing two new value types, `at-least-draw` and `at-most-draw`, which are bounds on exact values. And in Section 4.3 we extend the idea of attributed values by introducing another value type `cycle-draw`.

4.1 Incomparable Values

The only values we would like to deal with are `win`, `loss` and `draw`. Unfortunately it is not always possible to determine the exact value of a position within a given amount of time. In that case we use so-called *heuristic values* to calculate an estimate of the exact value. A heuristic value is an integer h in a range $h_{min} \leq h \leq h_{max}$. If a heuristic value is close to h_{max} then we say that the corresponding position is likely to be a `win`. If a heuristic value is close to h_{min} then we say that the corresponding position is likely to be a `loss`. If a heuristic value is close to 0 then we say that the corresponding position is likely to be a `draw`.

While the introduction of heuristic values allows us to estimate position values in a limited amount of time, it also gives rise to a new problem. We cannot, without information about the context of a game, tell whether a `draw` is better than a heuristic value or not: the `draw` and the heuristic values are *incomparable*.

For example, imagine your are playing a game in the last round of a tournament. In your current position you have two moves: one leads to a `draw`, but for the other move you have only a heuristic value. Which one should you play? Now assume that a `draw` is sufficient to win the tournament. In that case it is obviously better to play safe and to choose the move to the position with a `draw` value. On the other hand, it might be that you will at least end up on second place, regardless of the outcome of the last game, but a `win` might allow you to win the tournament. In this case it is obviously better to play for 'all or nothing' and to choose the move with the heuristic value.

4.1.1 Partially Ordered Sets

Because the `draw` value and the heuristic values are incomparable the set of game-position values forms a *partially ordered set* [47, Chapter 3].

A partially ordered set P is a set together with a binary relation $\leq$ satisfying three axioms:

1. For all $x \in P$, $x \leq x$. (reflexivity)
2. If $x \leq y$ and $y \leq x$, then $x = y$. (antisymmetry)
3. If $x \leq y$ and $y \leq z$, then $x \leq z$. (transitivity)

We use the obvious notation $x \geq y$ to mean $y \leq x$, $x < y$ to mean $x \leq y$ and $x \neq y$, and $x > y$ to mean $y < x$. We say two elements x and y of P are comparable if $x \leq y$ or $y \leq x$; otherwise x and y are incomparable.

If P is a partially ordered set, and if $x, y \in P$, then we say that y covers x if $x < y$ and if no element $z \in P$ satisfies $x < z < y$. The *Hasse diagram* of a partially ordered set P is the graph whose vertices are the elements of P, whose edges are the cover relations, such that if $x < y$ then y is drawn "above" x. Figure 4.1 shows the Hasse diagram for a game-value set with $h_{min} = -2$ and $h_{max} = 2$.

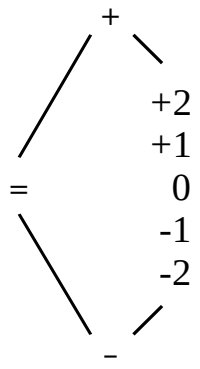

Figure 4.1: Hasse diagram for a game-value set with $h_{min} = -2$ and $h_{max} = 2$. The `draw` value and the heuristic values are incomparable.

4.1.2 Workaround for Search Engines

For search engines the problem of incomparability between `draw` and heuristic values is usually solved by treating `draw` and 0 as equal. This makes sense, because 0 is the heuristic estimate of `draw`, and because this can be implemented efficiently. Table 4.1 shows the propagation rules and `propagate1()` in Figure 4.2 is the corresponding propagation function.

$$\begin{array}{rcc} propagate(+, x) & \rightarrow & + \\ propagate(-, x) & \rightarrow & x \\ propagate(=, h) & \rightarrow & max(0, h) \\ propagate(h_1, h_2) & \rightarrow & max(h_1, h_2) \end{array}$$

Table 4.1: Simple max-propagation rules. + means `win`, − means `loss`, = means `draw`, h means heuristic value and x means any type of value.

Another way to solve the incomparability problem is to provide an additional parameter h_{draw}, the draw-threshold, to the propagation function, where $h_{min} \leq h_{draw} \leq h_{max} + 1$. h_{draw} represents a player's willingness to

```
int propagate1(int v1,int v2) {
  if ((v1 == draw) && (v2 != draw)) v1 = 0;
  if ((v2 == draw) && (v1 != draw)) v2 = 0;
  return max(v1,v2);
}/*propagate1*/
```

Figure 4.2: Propagation function with proven `draw`. It is assumed that `loss` $<$ `draw` $<$ `win`, `loss` $< h_{min}$ and $h_{max} <$ `win`.

end the game with a `draw` and is chosen separately for every game, or even for every search. Whenever we compare a `draw` with a heuristic value h, we propagate `draw` if $h < h_{draw}$, and we propagate h if $h \geq h_{draw}$. This means that the higher the value of h_{draw}, the higher is our preference for a `draw`. This effectively turns the partially ordered set into a totally ordered set (see Figure 4.3).

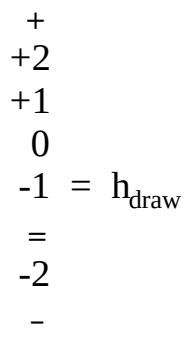

Figure 4.3: Hasse diagram for a game-value set with a draw-threshold. This set is totally ordered.

Table 4.2 shows the propagation rules and `propagate2()` in Figure 4.4 implements the corresponding propagation function. This solution is cheap and useful for search engines, but not usable for the construction of opening books, because at the time of construction we do not know the value of h_{draw}.

$$
\begin{array}{ccc}
propagate(+, x) & \rightarrow & + \\
propagate(-, x) & \rightarrow & x \\
propagate(=, h < h_{draw}) & \rightarrow & = \\
propagate(=, h \geq h_{draw}) & \rightarrow & h \\
propagate(h_1, h_2) & \rightarrow & max(h_1, h_2)
\end{array}
$$

Table 4.2: Max-propagation rules with draw-threshold h_{draw}.

```
int propagate2(int v1,int v2,int hdraw) {
  if ((v1 == draw) && (v2 >= hmin) && (v2 <= hmax)) {
    if (v2 < hdraw) return draw; else return v2;
  }/*if*/
  if ((v2 == draw) && (v1 >= hmin) && (v1 <= hmax)) {
    if (v1 < hdraw) return draw; else return v1;
  }/*if*/
  return max(v1,v2);
}/*propagate2*/
```

Figure 4.4: Propagation function with draw-threshold. It is assumed that `loss` $<$ `draw` $<$ `win`, `loss` $< h_{min}$ and $h_{max} <$ `win`.

4.2 `At-least-draw` and `At-most-draw`

In a situation like in Figure 4.5, the propagation rule of Table 4.1 propagates the value 0. If we want a fast propagation and are only interested in the best move at the start position then this is a good solution. However this propagation function loses information, namely that the player to move has the option of a guaranteed `draw`. In an opening book we want to keep this kind of information to make better decisions for book expansion. We can do this by introducing a new value type, the `at-least-draw` value, denoted by '$\geq$', and its inverse, the `at-most-draw` value, denoted by '$\leq$'. This leads to Figure 4.6.

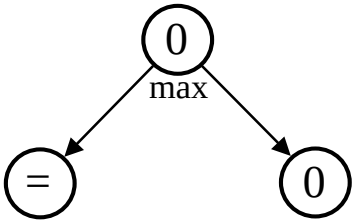

Figure 4.5: Simple propagation.

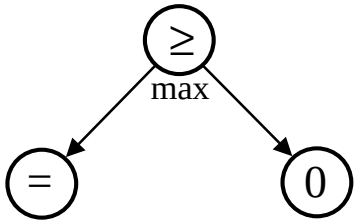

Figure 4.6: Propagation with `at-least-draw` values.

That these new value types indeed do keep more information and even help to solve positions is demonstrated in Figures 4.7 and 4.8. With the old propagation rule the value of the start position is a heuristic 0. With the new propagation rule we are able to prove that the value of the start position is a `draw`.

We can still do better than just use the `at-least-draw` value. Consider for example the situation of Figure 4.9. The two start positions have the same value, but the left one is probably a `draw` and the right one is probably a `win` (assuming that +9 is a real good heuristic value). What we would like to do is to keep the information about the value of the best heuristic value of the successors of a position with an `at-least-draw` value. We achieve this by introducing the attributed `at-least-draw` value, written $\geq|h$, where h is the value of the best heuristic successor ($h_{min} \leq h \leq h_{max}$). For `at-most-draw` values, $\leq|h$ is defined analogously. Figure 4.10 shows the same situation as Figure 4.9, but now with attributed `at-least-draw` values.

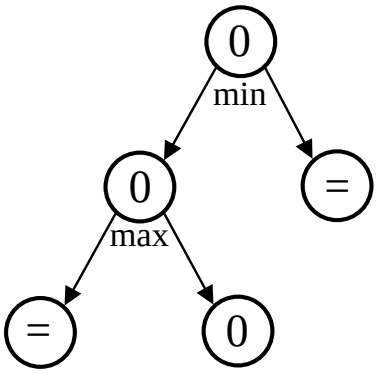

Figure 4.7: With simple propagation the start position of this graph is not solved.

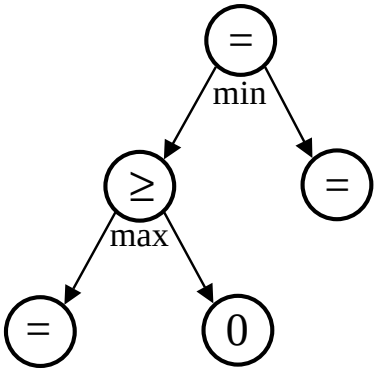

Figure 4.8: With `at-least-draw` / `at-most-draw` values the start position of this graph is solved.

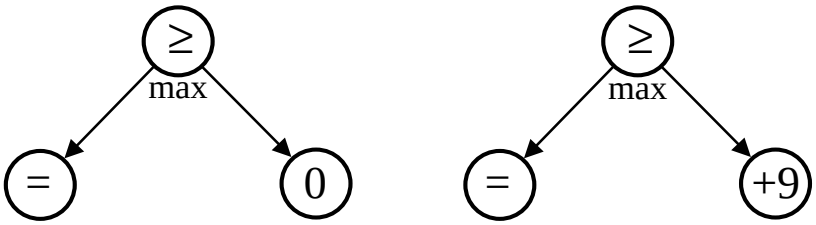

Figure 4.9: The start position on the left is probably a `draw`, the start position on the right is probably a `win`, but their values are the same.

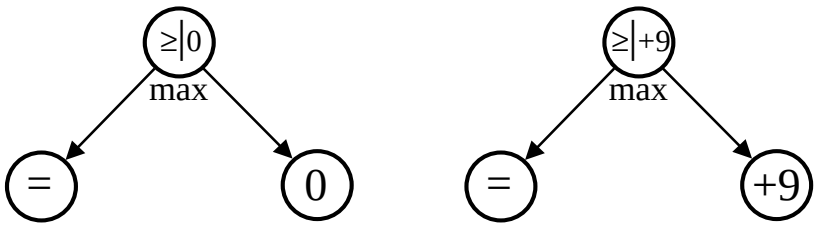

Figure 4.10: The same graphs as in Figure 4.9, now with attributed `at-least-draw` values.

Figure 4.11 shows the Hasse diagram of a value representation with attributed `at-least-draw` and `at-most-draw` values, and with $h_{min} = -2$ and $h_{max} = +2$. $\geq|h$ means that the player to move has a forced `draw`, but has the option to move into a position with value h instead. $\leq|h$ means that the opponent has a forced `draw`, but has the option to move into a position with value h. The values $[\geq|h_{min}, \ldots, \geq|h_{max}]$ and $[\leq|h_{min}, \ldots, \leq|h_{max}]$ are continuous ranges of values between `draw` and `win` and `loss` and `draw` respectively, just in the same way as $[h_{min}, \ldots, h_{max}]$ is a continuous range of values between `loss` and `win`. Therefore these new values give us a more accurate estimate of the exact value than what we had before.

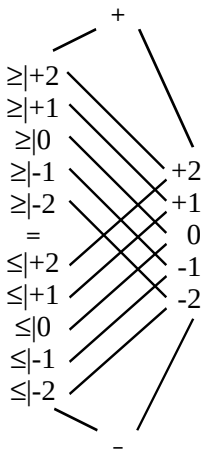

Figure 4.11: The Hasse diagram for a value representation with attributed `at-least-draw` and `at-most-draw` values, and with $h_{min} = -2$ and $h_{max} = +2$. $\geq|h$ means that the player to move has a forced `draw`, but has the option to move into a position with value h instead. $\leq|h$ means that the opponent has a forced `draw`, but has the option to move into a position with value h.

The term 'at least draw' is sometimes used with a different semantics. For example Chess positions with a king and a pawn against a king (KPK) are said to be 'at least draw' for the player with the pawn: whatever he does, no mistake is bad enough to make him lose.

Our definition of `at-least-draw` is weaker. The value $\geq|h$ must be read as '`draw` exclusive-or h', and if a player chooses h then he might still lose. However, the name `at-least-draw` is still justified, because whatever the exact value of the position with the currently heuristic value h will be, the exact value of $\geq|h$ will be either `win` or `draw`.

Although the problem of incomparability between `draw` and heuristic values is now solved, Figure 4.11 shows that now two new types of incomparability call for a propagation heuristic:

- $\geq|a$ and h are incomparable if $a < h$. This case is simple to solve. Remember that $\geq|a$ means that the player to move can force a `draw`, but has the option to move to a position with a heuristic value a (and a is the best heuristic value that is reachable). Therefore it is natural to propagate $\geq|max(a, h)$ in this case.

- $\leq|a$ and h are incomparable if $a > h$. We know that the propagated value of $\leq|a$ and h must be a heuristic value, because we cannot derive any bounds on the exact value from $\leq|a$ and h. Now $\leq|a$ means that the opponent can force a `draw`, but has the option to move to a position with a heuristic value a (and a is the best heuristic value that is reachable). Therefore when we compare $\leq|a$ and h, a is an option for the opponent and h is an option for the player to move, the two cannot be related to each other. To solve this problem we devised the following propagation heuristic: we assume that the opponent will always play the forced `draw` if $a \geq 0$, but will always avoid the `draw` if $a < 0$. The propagated value will be $max(h, min(a, 0))$.

Table 4.3 summarizes the propagation rules for a value representation with attributed `at-least-draw` and `at-most-draw` values.

$propagate(+, x)$	$\rightarrow$	$+$
$propagate(-, x)$	$\rightarrow$	x
$propagate(h_1, h_2)$	$\rightarrow$	$max(h_1, h_2)$
$propagate(=, \geq\|a)$	$\rightarrow$	$\geq\|a$
$propagate(=, \leq\|a)$	$\rightarrow$	$=$
$propagate(\geq\|a_1, \geq\|a_2)$	$\rightarrow$	$\geq\|max(a_1, a_2)$
$propagate(\geq\|a_1, \leq\|a_2)$	$\rightarrow$	$\geq\|a_1$
$propagate(\leq\|a_1, \leq\|a_2)$	$\rightarrow$	$\leq\|min(a_1, a_2)$
$propagate(=, h)$	$\rightarrow$	$\geq\|h$
$propagate(h, \geq\|a)$	$\rightarrow$	$\geq\|max(h, a)$
$propagate(h, \leq\|a)$	$\rightarrow$	$max(h, min(a, 0))$

Table 4.3: Propagation rules with `at-least-draw` and `at-most-draw` values. The last three rules define the propagation for incomparable values.

4.2.1 `At-least-draw`, `At-most-draw` and Opening Book Construction

There are two goals in opening book construction: to improve the playing strength at tournament games, and to solve a game. In a situation like in Figure 4.12 we have a conflict of interests.

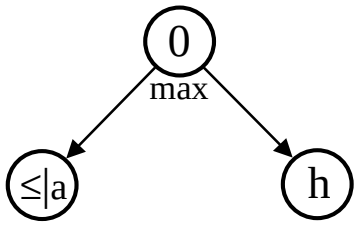

Figure 4.12: Conflicting expansion priorities (for $h < 0$). To improve the playing strength use drop-out expansion. To solve the start node use conspiracy numbers.

Improve playing strength The `at-most-draw` position is the most likely move and should get the higher priority for expansion. This is exactly what drop-out expansion does: $\leq|a$ is treated as 0, and the expansion priority of the two moves depends on h and the depths of the subtrees. However, the value of the start node does not depend on a, therefore expanding the left node is unlikely to influence the value of the start node.

Solve the game There are several ways how the start node of Figure 4.12 can be partially or completely solved. For a partial solution we can prove the left successor to be a `draw`, or prove that the right successor is either `at-most-draw` or a `loss`. For a complete solution we can either prove the right move to be a `win` or a `draw`. What we would like to do is to minimize the number of nodes that have to be expanded. This is best done with the conspiracy numbers heuristic [31, 43].

4.3 Cycles

As explained in Section 1.1, graphs of search problems are in general cyclic. For games, the most common rule for handling position repetitions is that the game is declared to be a `draw`. See Table 4.4 for an overview of cycle handling in various games.

The first problem of cycle handling is cycle detection, see Figure 4.13. Any time a move is added to the book, we have to find out if this new move closes a cycle, and update the values accordingly. Unfortunately it is impossible to detect a closed cycle locally, in general we have to search the whole graph to detect cycles. Our implementation of cycle detection is discussed in Section 5.2.2.

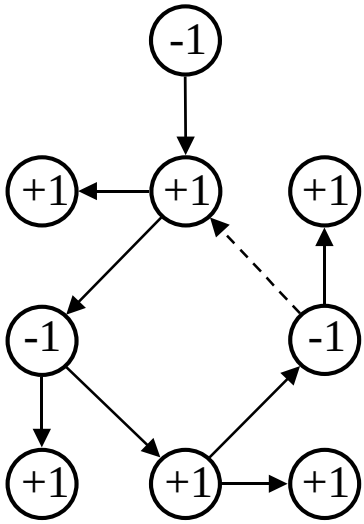

Figure 4.13: [negamax propagation] The cycle problem: when the dashed edge is added, a cycle is created. The cycle cannot be detected locally, at the node where it was closed, instead a global search is necessary. Moreover, all the values in the cycle are locally consistent, but should be set to 0 because the positions are most likely `draw`.

4.3.1 `Cycle-draw`

The second problem of cycle handling is cycle value propagation. Again our aim is to keep as much information about the value as possible. The simplest solution is to assign to every position in the cycle the heuristic value 0, see Figure 4.14. However, this loses some information about the

game	cyclic?	cycle value	cycle breaking moves
Abalone	yes	`draw`	capture
Amazons	no		
Awari	yes	The remaining stones are split between the players, thus the configuration has value 0, the position value depends on the distribution of captured stones.	captures
Checkers	yes	`draw`	captures, conversions, checker moves
Chess	yes	`draw`	captures, conversions, pawn moves, castling
Chinese Checkers	yes	`draw`	none
Chinese Chess	yes	`draw` (Perpetual checks and perpetual threats to capture are forbidden.)	captures, forward moves by pawns
Connect-four	no		
Go	no		
Gomoku	no		
Hex	no		
Nine Men's Morris	yes	`draw`	opening moves, captures
Othello	no		
Qubic	no		
Shogi	yes	`draw` (If the repetition is forced by a sequence of checking moves of one player, then that player loses.)	none

Table 4.4: Cycle handling for some games.

value of the position, as shown in Figure 4.15: the same value is propagated for positions with a value almost identical to a 0 and for positions with a value almost identical to a `draw`. As in the case with the `at-least-draw` and `at-most-draw` values we solve this by introducing a new attributed value type, the `cycle-draw`, denoted $0|+h_1|-h_2$, which has two attributes, $+h_1$ and $-h_2$. $+h_1$ is the smallest heuristic value with which the opponent can move out of the cycle, in other words the opponent either stays in the cycle and settles for a `draw`, or plays a move that has a value of at least $+h_1$ for the current player. $-h_2$ is the largest heuristic value with which the current player can move out of the cycle, in other words the current player either stays in the cycle and settles for a `draw`, or plays a move that has a value of at most $-h_2$ for the current player.

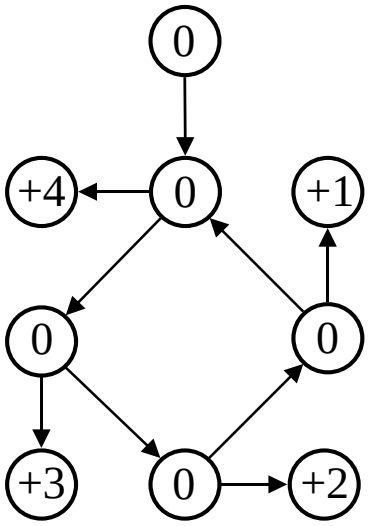

Figure 4.14: [negamax propagation] A cycle using the heuristic value 0 as cycle value.

As the use of the + and − sign in the notation for the `cycle-draw` suggests, the first attribute is always strictly positive and the second attribute is always strictly negative. Otherwise one player would have a cycle-leaving move which is at least as good as any cycle-preserving move. Then the value of the cycle-leaving move is assumed to be better and the corresponding heuristic value is propagated. Figure 4.16 shows the same situation as Figure 4.15, now using attributed `cycle-draw` values.

When a game has a heuristic value range $h_{min}, \ldots, h_{max}$, the value ranges for the attributes of the `cycle-draw` are $1, \ldots, h_{max}$ for the first attribute and $h_{min}, \ldots, -1$ for the second attribute. However it is possible that all the cycle-

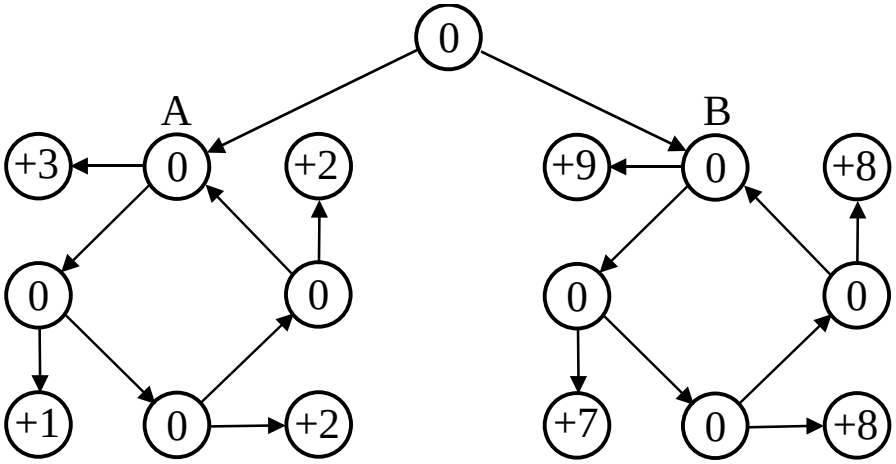

Figure 4.15: [negamax propagation] Using heuristic 0 as cycle value loses information. In this graph, the two successors of the start position look exactly the same. However, the penalty for leaving the right cycle is higher for both players, therefore the right cycle is closer to a `draw` than the left cycle.

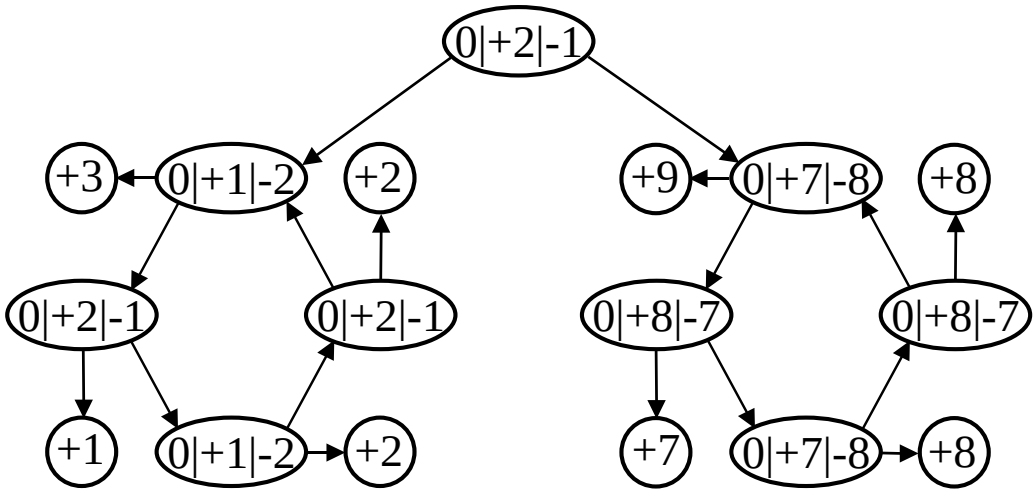

Figure 4.16: [negamax propagation] The same graph as in Figure 4.15, but now with attributed `cycle-draw` values.

leaving moves for one or both players have `loss` values. We use $0|+|-h$ for the case where the opponent has only losing cycle-leaving moves, we use $0|+h|-$ for the case where the current player has only losing cycle-leaving moves, and we use $0|+|-$ for the case where both players lose if they play a cycle leaving move. Obviously, the values $0|+|-h$, $0|+h|-$ and $0|+|-$ are equivalent to $\geq|-h$, $\leq|+h$ and `draw`, respectively. We conclude that `cycle-draw` values are a generalization of the previously introduced `at-least-draw`, `at-most-draw` and `draw` values.

Table 4.5 shows the propagation rules for a value representation with `cycle-draw` values.

$propagate(+, x)$	$\rightarrow$	$+$
$propagate(-, x)$	$\rightarrow$	x
$propagate(h_1, h_2)$	$\rightarrow$	$max(h_1, h_2)$
$propagate(=, \geq\|a)$	$\rightarrow$	$\geq\|a$
$propagate(=, \leq\|a)$	$\rightarrow$	$=$
$propagate(\geq\|a_1, \geq\|a_2)$	$\rightarrow$	$\geq\|max(a_1, a_2)$
$propagate(\geq\|a_1, \leq\|a_2)$	$\rightarrow$	$\geq\|a_1$
$propagate(\leq\|a_1, \leq\|a_2)$	$\rightarrow$	$\leq\|min(a_1, a_2)$
$propagate(=, h)$	$\rightarrow$	$\geq\|h$
$propagate(h, \geq\|a)$	$\rightarrow$	$\geq\|max(h, a)$
$propagate(h, \leq\|a)$	$\rightarrow$	$max(h, min(a, 0))$
$propagate(0\|+h_1\|-h_2, 0\|+h_3\|-h_4)$	$\rightarrow$	$0\|min(h_1, h_3)\|max(-h_2, -h_4)$
$propagate(h, 0\|+h_1\|-h_2)$	$\rightarrow$	$max(h, 0)$
$propagate(=, 0\|+h_1\|-h_2)$	$\rightarrow$	$\geq\|0$
$propagate(\geq\|a_1, 0\|+h_1\|-h_2)$	$\rightarrow$	$\geq\|max(a_1, 0)$
$propagate(\leq\|a_1, 0\|+h_1\|-h_2)$	$\rightarrow$	0

Table 4.5: Max-propagation rules with `cycle-draw` values. When compared to other types of values, the `cycle-draws` are treated like heuristic 0 values.

4.3.2 `Cycle-draw` and Book Expansion

The attributed `cycle-draw` values provide a measure for estimating how close we are in proving that a position is a `draw` by repetition. When we expand an opening book, we are usually interested in expanding a cycle-leaving move with the smallest absolute value. For example, in Figure 4.16, the candidates for expansion are the leaf nodes with values +1 and +2 at the bottom in the left cycle, because these are the values on which the attributes of the start node depend.

In general these candidates cannot be found by local decisions on a path from the start node. At position A in Figure 4.17, look-ahead is necessary to decide which path should be taken to reach the candidate leaf nodes. We solved this problem by using the drop-out expansion strategy, where the expansion priority of a leaf node depends not only on its value, but also on its depth, see Section 3.4.2. With drop-out expansion, the leaf node with highest expansion priority can still be found by making local decisions.

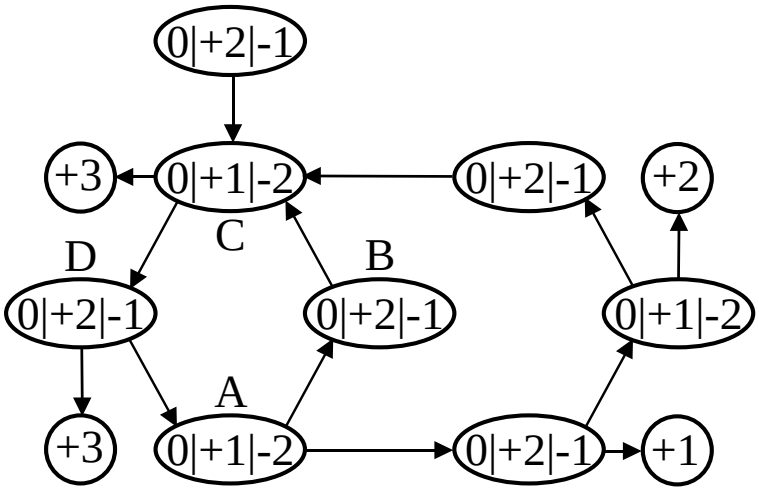

Figure 4.17: [negamax propagation] It is not possible to find the best cycle-leaving moves by only looking at the successor values. For example, if we are at position A, we cannot decide locally which move leads to the best cycle-leaving moves (the leaf nodes with values $+1$ and $+2$).

4.3.3 `Cycle-draw` and Tournament Play

Like for book expansion, the `cycle-draw` values are useful in tournament play to estimate how close we are in proving that a position is a `draw`. However, when a position has the value $0|+h_1|-h_2$ it only means that a move *exists* such that the current player can leave the cycle with value $-h_2$. Whether such a move is reachable may depend on the moves of the opponent. Assume for example that in Figure 4.17 the first player in the start node decides that he is willing to leave the cycle with a penalty of -1, and moves into the cycle. But at position A, it is the opponent's move, and if the opponent moves to position B, the game is forced into a `draw`.

Note that we cannot simply conclude that position A is at least a `draw` for the opponent; this depends on where the cycle was entered. Had the

current player entered the cycle at position A, then a move to B would not force a `draw`, but the first player could still leave the cycle with a penalty of -3 at position C.

We conclude that `cycle-draw` values are useful for estimating the solvedness of a position and to guide the expansion of opening books. In a tournament game we still need look-ahead and opponent modelling to find out which leaf nodes are reachable.

Chapter 5

OPLIB: Architecture and Implementation

This chapter describes OPLIB, a game independent tool for opening book construction and game solving. The motivation for the implementation of OPLIB was twofold: First, because of the lack of human expert knowledge in Awari, we needed a tool for the automatic construction of an opening book. Second, because of the observation that the previous solution of games always involved some large scale forward search. Two different strategies were used for game solving with forward search:

Single search Nine Men's Morris [15] and Kalah [20] were solved by a single search from the start position.

Multiple searches Gomoku and Qubic [4] were solved by multiple searches. Whenever a position turned out to be unsolvable within given resource constraints, then the solution task was broken down recursively into smaller pieces by solving the successor positions one by one.

The second strategy has several advantages. For example it is more general because it can be applied even if the resources are insufficient for the first strategy. It also allows to keep a better eye on the progress of the solution process. However, it also requires a certain maintanence overhead because we need to keep track of the recursive work breakdown and the reconstruction of the results.

For Awari we decided single search is not feasible: the runtime is unpredictable, and we want to add endgame databases to the search as they become available. To apply the multiple-search approach to Awari we decided to automatize the search management in the form of an opening book tool.

5.1 Architecture

Figure 5.1 shows the architecture of OPLIB. OPLIB comes in two versions: An interactive, single machine version and a background version for distributed book construction. The interactive version is used to manually browse an opening book. It provides a shell which allows the user to navigate the book, to add and edit nodes, and to run the search engine for the current node. The background version is used for automatic book construction on a cluster of workstations. It acts as a server which hands out positions to the clients and updates the book with the values calculated by the clients. It also acts as a server for web interfaces.

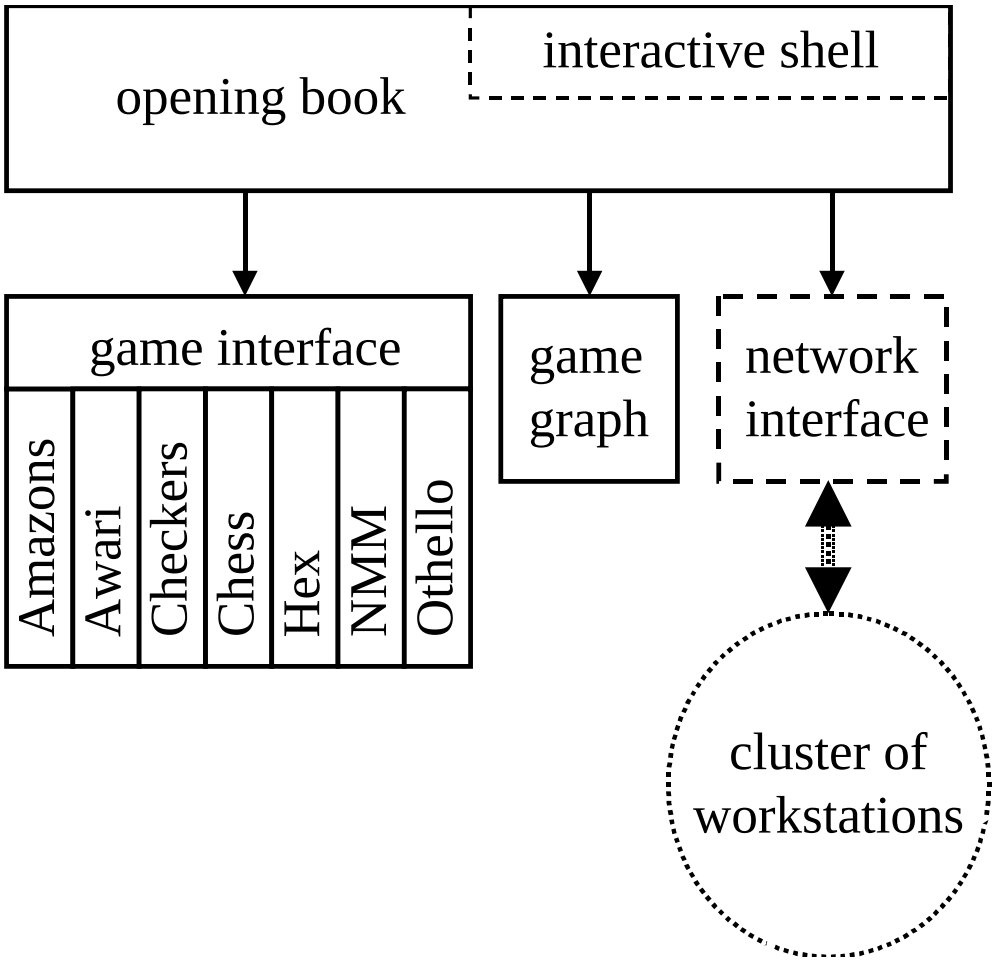

Figure 5.1: The main modules of OPLIB. The interactive shell is only available in the single machine version, the network interface is only available in the background version.

OPLIB consists of the following main components:

Opening Book The main module is responsible for book management. In the interactive version it also provides a text-based user interface.

Game Graph At the core of OPLIB is an implementation of a directed graph. Both the nodes and the edges are configurable to have attributes. This flexibility simplifies experimenting with different expansion strategies, as every strategy requires its own set of node attributes.

Game Interface OPLIB defines an interface of about a dozen game specific functions. Currently these have been implemented for Amazons, Awari, Checkers, Chess, Hex, Nine Men's Morris and Othello.

Network Interface A network interface is used for the background version. It uses a simple string-based protocol to communicate with the clients.

5.2 Implementation

During the implementation of OPLIB several design decisions had to be made that influenced performance and usability of the system. The following sections exemplify three such design issues, how they where solved and how they influenced the system.

5.2.1 Main Memory versus External Memory

At the very beginning of the project the question arose as to whether the opening book should be kept in main memory or on disk during operation. Keeping the book in memory has the big advantage of fast access to the data structure. However we decided to keep the book on disk for the follwing reasons:

- A tournament program that uses the book only needs information about the current position and its successors. The time necessary to load this information from disk is negligible, whereas the amount of memory that would be required to store the book is significant.

- Our estimates showed that it was likely that some books would grow beyond the size of the main memory of a standard workstation (a few hundred megabytes). Hard disks are up to a hundred times larger than main memories.

- The majority of the book operations only require access to one position or a path of positions. For example the drop-out expansion strategy described in Chapter 3 accesses all positions on a path from the start position to a leaf position. The number of positions on this path is exponentially smaller than the number of positions in the whole book.

- For a large opening book the time required to load the book into memory would be significant. Keeping the book on disk makes the startup time of the system independent of book size.

- For some applications and systems it is preferable to keep the data in memory and to use the swap mechanism of the operating system to handle the details of loading and unloading data. The drawback of this approach is that the operating system does not know anything about the semantics of the data. For example it accesses the files blockwise, whereas in the case of an opening book it is preferable to use position-wise caching because the book is accessed at the granularity of positions.

With hindsight we can say that the decision to store the book on disk was justified. Not having to bother about main memory size improves the usability of the system, for example we can run background servers for all games on a single machine. The drawback is that some operations like calculating drop-out diagrams become increasingly time consuming for larger books.

5.2.2 Transposition Detection and Cycle Detection

Transposition detection is done every time a new successor is about to be added to the opening book. If the successor already is in the book, then a new edge is added between the leaf node and the successor. If the successor is not yet in the book then a new node and a new edge are added to the book. To check for the existence of a node we store every node in an extensible hash table [12]. Extensible hashing has the advantage that every lookup can be done with two disk accesses.

Cycle detection is a classical problem from graph theory. We need to find the cycles in a game graph so that we can assign the proper values to the positions in the cycle, see Section 4.3.

In a first attempt we tried an incremental solution: Every time an edge was added to the graph we immediately checked if the new edge had closed a cycle. This turned out to be a major performance bottleneck, the system started to spend most of the time in cycle detection. Moreover, most of the time the result of the check was negative.

Therefore we decided to use a global cycle detection algorithm: When an edge is added to the graph we do not check for cycles. Instead we do cycle detection on the whole book, see Figure 5.2. This cycle detection is invoked manually, whenever the book seems to suffer from too many undetected cycles. For most games this change is irrelevant, because they either have no cycles, or cycles only play a minor role for opening books, see Table 4.4. Currently the only game which has an abundance of cycles in the opening is Chess.

5.2.3 Distributed Expansion

The idea of distributing the work for forward search to a cluster of workstations is straightforward. One server which manages the book can keep several clients busy which run searches on expanded positions in parallel.

The code for the network interface is relatively simple, consisting of about 100 lines of code. However, the whole system also requires some adaptions for distributed expansion.

Efficiency When a client connects to the server we want to send it a job as quickly as possible. Therefore we keep a short queue of jobs ready for search, and we refill that queue in the idle time between client connections.

Re-scheduling Once a job has been sent to a client, we have to make sure that the same job is not sent to another client. Therefore we keep a history of recently sent jobs. Any job in the queue is assumed to be running on a client already.

Robustness There are various issues of robustness in connection with distributed computing. The most important issue here is client death: there is no guarantee that the result of a job sent to a client is eventually returned, be it because of network problems or because the client process has been killed. If the result of a job is never returned then the job would remain in the history forever. To avoid this we use history aging: When a job remains in the history for too long then it is allowed to be resent to a client.

These adaptions also influence the process of book expansion. Because of the queue we do not only expand the 'best' node but the 'best n' nodes. Therefore the single machine version and the distributed version do not necessarily calculate the same opening book. Moreover the distributed version is nondeterministic because the order in which the results are entered into the book

```
void cycledetection() {
  Node n;
  Value NewValue;
  boolean Changed;
  for (all unsolved interior nodes n) {
    n.value = 0|+∞|−∞;
  }/*for*/
  Changed = true;
  while (Changed == true) {
    Changed = false;
    for (all unsolved interior nodes n) {
      NewValue = prop_n(s_1.value, ..., s_d.value);
      if (NewValue != n.value) {
        n.value = NewValue;
        Changed = true;
      }/*if*/
    }/*for*/
  }/*while*/
  for (all unsolved interior nodes n) {
    if (n.value == 0|+∞|−∞) {
      n.value = draw;
    }/*if*/
  }/*for*/
}/*cycledetection*/
```

Figure 5.2: The global cycle detection algorithm. First set all node values to the new artificial value $0|+\infty|-\infty$, which is a synonym for `draw`. Then propagate until no changes occur. All nodes which are not part of a cycle will receive their former value. Nodes which are part of a cycle with unsolved cycle-leaving moves will end up with a value of the form $0|+a|-b$. If all the values of cycle-leaving moves are `loss` then the values of the nodes in the cycle will be $0|+\infty|-\infty$, these nodes are `draw`.

is non-deterministic. Therefore two runs of the distributed version do not necessarily calculate the same opening book either.

There also is a trade-off in the choice of history-aging speed. If the history ages too fast then too many jobs are solved twice. If the history ages too slow then it may take too long until the result of the most interesting position is returned to the server. We usually tune the history-aging speed in such a way that only about every thousandth position is calculated twice.

Chapter 6

Experiments with Opening Book Construction

This Chapter gives an overview of the current state of the opening books for the games that were implemented for OPLIB. Every Section starts with a short discussion of game properties and the engine used for book expansion. Then follows a Section with book statistics, which includes a diagram and a table with node distributions and, if applicable, a drop-out diagram.

The diagrams give an impression of some graph properties of the games. For example it is shown how the branching factor varies during the course of a game, and the average number of predecessors of the nodes is an estimate of the connectivity of the graph. The statistics also give an impression of how far the games are from being solved.

6.1 Amazons

Game Properties

The tournament version of Amazons is played on a 10x10 board with four queens for each player. The game is scalable to other board sizes; reasonable games can be played from 5x5 upward. Scalability makes Amazons a nice testbed for game solving tools.

An upper bound for the size of the state-space of Amazons on a nxn board is $\binom{n^2}{4}\binom{(n^2-4)}{4}2^{n^2-8}/8$, see Table 6.1. The branching factor is very high at the beginning of the game, but drops significantly in the first couple of moves. The game graph contains no cycles, but transpositions are possible.

board size	state-space	first move branching factor
5x5	$1.2 * 10^{12}$	260
6x6	$7.1 * 10^{16}$	544
7x7	$8.7 * 10^{21}$	812
8x8	$2.8 * 10^{27}$	1232
9x9	$2.7 * 10^{33}$	1700
10x10	$8.1 * 10^{39}$	2176

Table 6.1: Amazons state-space sizes for different boards. The number of moves from the start position is very high compared to other games with similar state-space sizes.

Search Engine

As a search engine we used Arrow (Macintosh version available at [33]), a strong Amazons program written by Martin Müller. Arrow was able to prove that 5x5 Amazons is a first player `win` [32].

Opening Book Statistics

Figure 6.1 shows the distribution of the depths of the nodes in the book. About 2,000,000 nodes were expanded in the depth range from 0 to 9 with a peak of over 1,000,000 nodes at depth 4. Table 6.2 shows more details of the book statistics. The 'Nodes' column shows the number of nodes at each depth. The 'Solved' column shows the number of nodes that were solved. The start position has been prooven to be a `win`, two winning moves are

known. The 'Predecessors' column shows that, on average, every position in the book has about 1.5 predecessors which are also in the book. This means that Amazons has many transpositions. The 'Degree' column shows the average number of possible moves at each depth. The branching factor is very high at the beginning of the game, but drops significantly during a game.

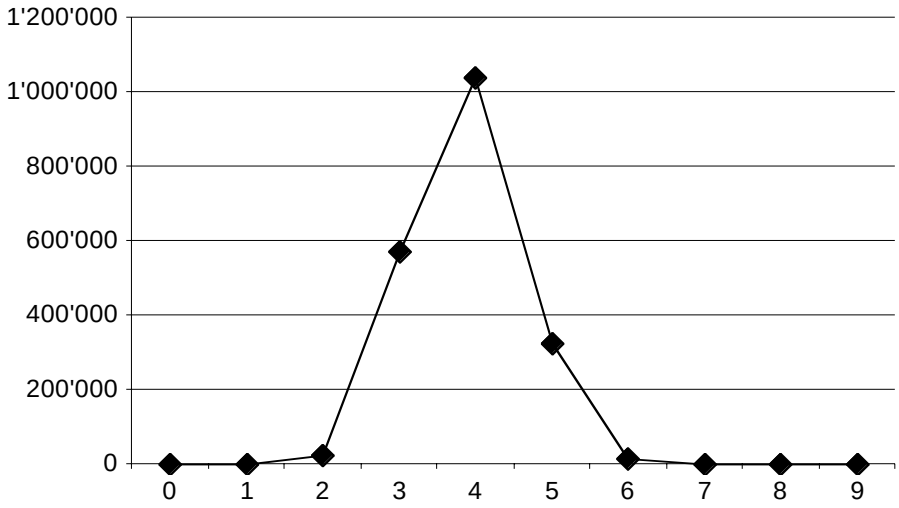

Figure 6.1: The depth distribution of the nodes in the Amazons 5x5 opening book.

6.2 Awari

Game Properties

An upper bound for the number of states in Awari is $889,063,398,406$ (see Section 2.2). [4] gives a branching factor of 3.5 and a search-tree complexity of 10^{32}. In our opening book, the average branching factor is about 5 6.3. This is due to the fact that the vast majority of the positions in the book have 40 or more stones on the board. The game graph contains transpositions and cycles, but both are rare, and to the best of our knowledge the current opening book does not contain a single cycle.

Depth	Nodes	Solved	Predecessors	Degree
0	1	1	0	260.0
1	130	2	260	193.9
2	22,872	824	25,211	158.6
3	570,854	2,949	862,755	118.9
4	1,036,913	63,123	1,401,896	93.5
5	323,503	84,437	586,791	57.2
6	15,496	11,442	18,182	62.7
7	61	61	61	35.3
8	35	35	35	23.9
9	25	25	25	21.0
Total	1,969,890	162,899	2,895,216	95

Table 6.2: Book statistics for Amazons 5x5.

Search Engine

As a search engine we used Marvin, a strong Awari program written by the author. Marvin has won a silver medal in international computer Awari tournaments twice (1992 [9] and 1995 [58]), has won the gold medal once (2000 [30]) and currently is the reigning computer Awari world champion. For opening book construction, Marvin was configured to search to a fixed depth of 20 plies. Endgame databases with up to 40 stones were used, but only on the first 8 plies of the search, because accessing the databases in deeper positions would increase the search time without noticable benefit.

Opening Book Statistics

Figure 6.2 shows the distribution of the depths of the nodes in the book. About 1,100,000 nodes were expanded in the depth range from 0 to 93 with a broad peak in the range from depth 17 to 50. Tables 6.3 and 6.4 show more details of the book statistics. The 'Nodes' column shows the number of nodes at each depth. The 'Solved' column shows the number of nodes that were solved. Thanks to the endgame databases an average of about 25% percent of all nodes are solved. The 'Predecessors' column shows that, on average, every position in the book has about 1.02 predecessors which are also in the book. This means that cycles and transpositions are rare in Awari. The 'Degree' column shows the average number of possible moves at each depth. The degree is remarkably stable during the course of a game.

Depth	Nodes	Solved	Predecessors	Degree
0	1	0	0	6.0
1	6	0	6	6.0
2	36	0	36	5.3
3	81	0	81	5.3
4	244	0	244	5.1
5	437	0	437	5.2
6	932	0	944	5.1
7	1,711	4	1,749	5.2
8	2,880	7	2,921	5.1
9	4,592	46	4,644	5.1
10	5,959	116	6,046	5.2
11	7,905	183	8,027	5.1
12	9,494	257	9,655	5.1
13	12,220	334	12,471	5.1
14	13,469	501	13,813	5.1
15	16,583	589	17,020	5.1
16	17,187	779	17,680	5.1
17	19,479	857	20,117	5.0
18	19,057	1,128	19,671	5.1
19	21,352	1,284	22,001	5.0
20	19,356	1,578	20,048	5.0
21	20,916	1,817	21,571	5.0
22	19,967	2,210	20,679	5.0
23	22,336	2,402	22,985	5.0
24	21,407	2,838	22,045	5.0
25	24,067	3,149	24,735	4.9
26	22,924	3,484	23,615	5.0
27	25,516	3,945	26,142	4.9
28	24,127	4,341	24,785	5.0
29	26,703	4,830	27,334	4.9
30	24,635	4,924	25,186	5.0
31	27,381	5,670	27,946	4.9
32	25,676	5,541	26,288	4.9
33	26,859	5,865	27,473	4.9
34	25,043	5,581	25,583	4.9
35	26,054	5,956	26,548	4.9
36	24,655	5,976	25,140	4.9
37	24,847	6,348	25,325	4.9
38	23,713	6,278	24,152	4.9
39	24,055	6,664	24,519	4.9
40	23,746	6,839	24,214	4.9
41	23,985	7,325	24,460	4.9
42	23,628	7,663	24,075	4.9
43	23,075	7,503	23,492	4.9
44	22,950	7,696	23,350	4.9
45	22,646	7,743	23,025	4.9
46	22,651	8,115	23,025	4.9

Table 6.3: Book statistics for Awari, part 1.

Depth	Nodes	Solved	Predecessors	Degree
47	23,051	8,463	23,437	4.9
48	21,661	8,207	22,034	4.9
49	21,035	7,885	21,397	4.9
50	19,895	7,538	20,252	4.9
51	19,203	7,084	19,553	4.9
52	18,696	6,882	19,030	5.0
53	17,914	6,479	18,236	4.9
54	16,853	6,011	17,170	5.0
55	16,506	5,820	16,787	4.9
56	15,658	5,441	15,954	5.0
57	14,136	5,000	14,410	4.9
58	12,365	4,333	12,570	5.0
59	11,402	3,792	11,602	4.9
60	10,154	3,514	10,339	4.9
61	8,936	3,078	9,103	4.9
62	7,682	2,671	7,826	4.9
63	6,288	2,292	6,386	4.9
64	5,281	1,911	5,397	4.9
65	4,514	1,804	4,575	4.9
66	4,032	1,772	4,086	4.9
67	3,667	1,705	3,712	4.9
68	3,437	1,720	3,484	4.9
69	3,350	1,633	3,391	5.0
70	3,190	1,691	3,232	5.0
71	2,916	1,553	2,953	5.0
72	2,620	1,477	2,653	5.0
73	2,278	1,271	2,308	5.0
74	1,990	1,128	2,010	5.0
75	1,635	939	1,649	4.9
76	1,436	833	1,456	4.9
77	1,278	767	1,297	5.0
78	1,005	588	1,017	4.9
79	792	478	800	4.9
80	647	384	662	4.9
81	475	287	483	4.9
82	377	211	380	4.8
83	278	154	283	4.9
84	218	100	218	4.9
85	128	60	129	5.0
86	151	72	152	4.8
87	99	60	102	5.0
88	89	46	90	5.0
89	112	66	114	4.9
90	59	36	60	4.9
91	44	23	44	4.9
92	30	22	30	5.3
93	11	4	11	4.7
Total	1,104,117	265,651	1,128,167	4.9

Table 6.4: Book statistics for Awari, part 2.

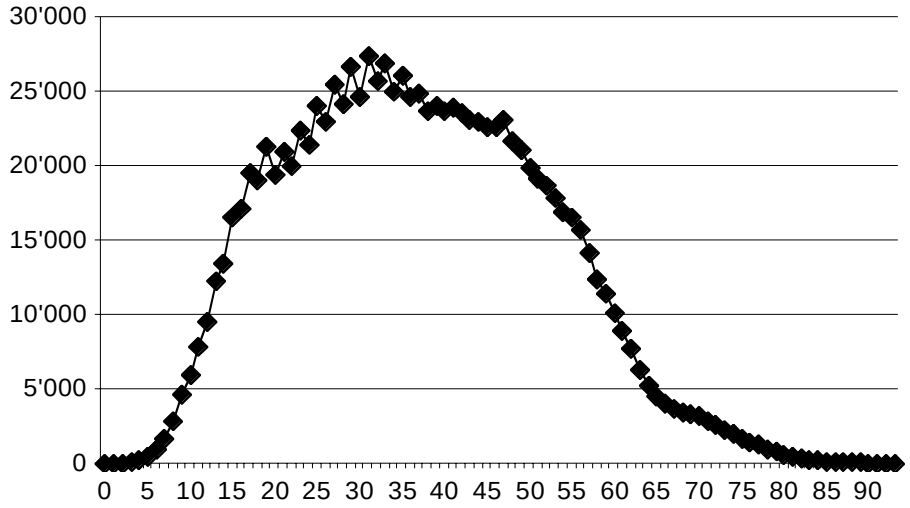

Figure 6.2: The depth distribution of the nodes in the Awari opening book.

Figure 6.3 shows the drop-out diagram of the Awari opening book. The dashed line is the angle of the expansion frontier as defined by our choice of the parameter ω. At this moment, nodes with value 0 at depth 22 have the highest expansion priority.

Opening Book Evaluation

Figure 6.4 shows the first three plies of the Awari opening book. On their first move, both players have only one good move, F4 and f4, respectively. All other moves seem to lose at least two stones. After that the first player has a choice of four moves which are all drawish.

The construction of the Awari opening book started when the 30-stone database was available and continued until the 40-stone database was available. The question arose what to do with the book when new databases are added: should we just continue with the current book so that we don't lose our previous search results, or should we throw the book away and start all over because we can get better values with new database?

In the case of Awari throwing away the whole book seemed to be a waste of time, because the heuristic evaluation function doesn't seem to change too much when a new database is added. Therefore we decided to throw away only those positions which are now solved in the database. For example, when the 39-stone database became available all positions with 39 or fewer stones where removed from the book. This means that some positions had

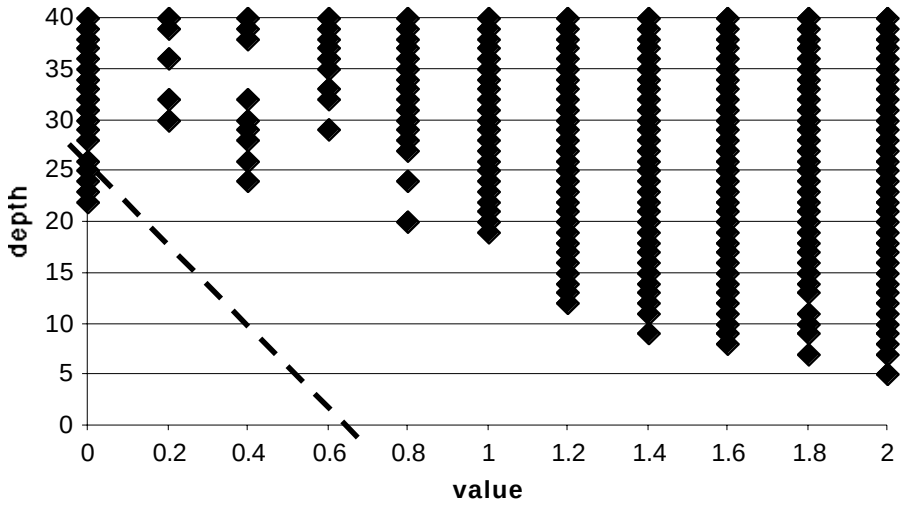

Figure 6.3: The drop-out diagram of the Awari opening book. The value 1 is equivalent to one stone.

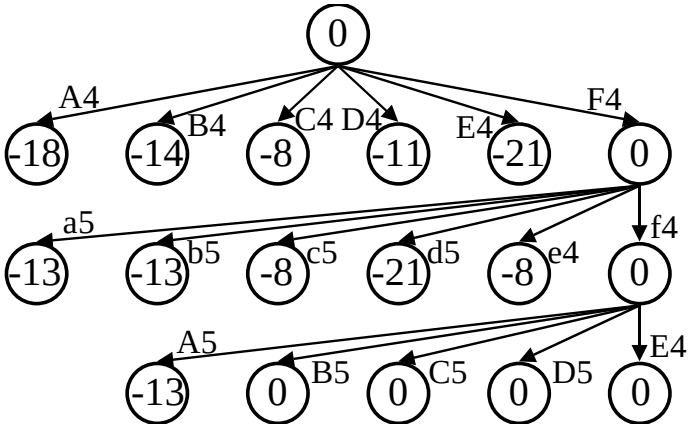

Figure 6.4: The first three plies of the principal variations of the Awari opening book. A value of 5 is equivalent to on stone.

to be searched several times because they got removed every time a new database was added. But the overhead of doing that is small because the reevaluation is done with one database lookup.

The opening book was an important factor in winning the computer Awari world championship [30].

6.3 Checkers

Game Properties

Checkers has an estimated state-space complexity of 10^{18} [46], an estimated search-tree complexity of 10^{31} [4] and an estimated average branching factor of 1.1 in capture positions and 7.8 in non-capture positions [45]. The game graph contains cycles, but only after both players have converted at least one checker to a king. As a consequence, cycles are rare in the opening book.

A speciality of the Checkers games rules are the so-called 3-move-ballots. In human tournaments the game is not played from the start position, because from there it is too easy for the weaker side to keep a `draw`. Instead the first three plies are picked at random, and then two games are played with each player playing black once.

To take 3-move-ballots into account OPLIB was configured to ignore the expansion priorities on the first three plies. In other words, all 216 positions reachable in three plies are treated as start positions with equal expansion priorities.

Search Engine

As a search engine we used Cake++ [13], a strong Checkers program written by Martin Fierz. Cake++ is freely available, together with the CheckerBoard GUI. For opening book construction, Cake++ was configured to search about 30 seconds per position and used the 6-stone endgame databases available from the Chinook project [40].

Opening Book Statistics

Figure 6.5 shows the distribution of the depths of the nodes in the book. About 1,834,000 nodes were expanded in the depth range from 0 to 28 with a peak of more than 200,000 nodes at depth 14. Table 6.5 shows more details of the book statistics. The 'Nodes' column shows the number of nodes at each depth. The 'Solved' column shows the number of nodes that

were solved. Obviously the engine together with the 6-stone database cannot solve positions in the early opening. The 'Predecessors' column shows that, on average, every position in the book has about 1.2 predecessors which are also in the book. The 'Degree' column shows the average number of possible moves at each depth. The degree is remarkably stable during the course of a game.

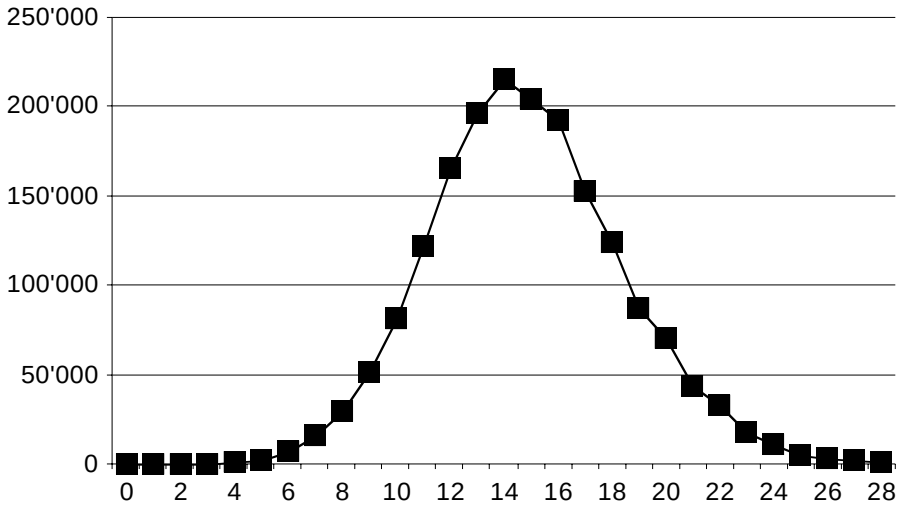

Figure 6.5: The depth distribution of the nodes in the Checkers opening book.

Figure 6.6 shows the drop-out diagram of the Checkers opening book. The dashed line is the angle of the expansion frontier as defined by our choice of the parameter ω. At this moment, nodes with value 8 at depth 13 have the highest expansion priority.

6.4 Chess

Game Properties

Chess has an estimated state-space complexity of 10^{50}, an estimated search-tree complexity of 10^{123} and an estimated average branching factor of 35 [4]. The game graph contains many transpositions and cycles, because almost every position can by part of a cycle of length four by just moving some piece forth and back. Therefore the game is an ideal testbed for the `cycle-draw` values presented in Chapter 4. Moreover the large literature on Chess openings allows us to compare our book to human opening knowledge.

Depth	Nodes	Solved	Predecessors	Degree
0	1	0	0	7.0
1	7	0	7	7.0
2	49	0	49	6.2
3	216	0	302	4.7
4	805	0	1,025	4.7
5	2,406	0	3,426	4.7
6	6,663	0	9,319	4.7
7	15,758	0	22,956	4.6
8	29,603	0	40,977	4.8
9	51,516	0	71,481	4.9
10	81,209	0	106,757	5.2
11	121,744	0	159,973	5.4
12	166,078	0	212,014	5.5
13	196,675	0	253,591	5.6
14	215,052	0	264,314	5.8
15	203,939	0	252,680	5.9
16	192,647	1	226,408	6.0
17	152,652	1	181,680	6.1
18	124,358	0	140,343	6.1
19	87,159	7	101,706	6.1
20	70,204	3	78,776	6.1
21	44,004	8	50,781	6.1
22	32,646	4	35,877	6.1
23	18,292	36	20,890	6.1
24	10,683	10	11,444	5.9
25	4,863	25	5,219	5.9
26	2,781	2	2,870	5.4
27	1,621	35	1,736	5.8
28	1,144	9	1,193	5.2
Total	1,834,775	141	2,257,794	5.8

Table 6.5: Book statistics for Checkers.

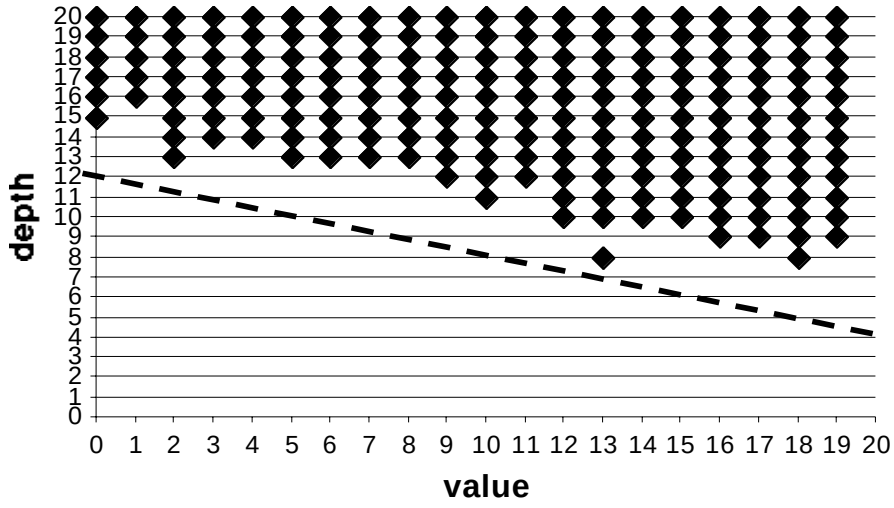

Figure 6.6: The drop-out diagram of the Checkers opening book. The value 100 is equivalent to one checker. The shallowest line of the principal variations is 15 plies deep.

Search Engine

As a search engine we used Crafty [19], a strong Chess program written by Robert Hyatt. Crafty is available in source code [18], and has about 2500 Elo on the *Swedish Rating List* [21].

For opening book construction, Crafty was configured to search to a fixed depth of ten plies and the built-in options for using an opening book or endgame databases were turned off. The heuristic values returned by Crafty are mapped to a range of $-60 \ldots 60$, where $+10$ is equivalent to one pawn, $+30$ is equivalent to a knight/bishop and $+60$ is a near `win`.

Opening Book Statistics

Figure 6.7 shows the depth distribution of the nodes in the book. About 950,000 nodes were expanded in the depth range from 0 to 11, with a distinctive peak at plies 7 and 8. Table 6.6 shows more details of the book statistics. The 'Nodes' column shows the number of nodes at each depth. The 'Solved' column shows the number of nodes that were solved, their number is insignificant compared to the total number of nodes in the book. The 'Predecessors' column shows that, on average, every position in the book has about 2 predecessors which are also in the book. This proves the fact that the game graph of Chess looks more like a mesh than like a tree, with many

transpositions and cycles. The 'Degree' column shows the average number of possible moves at each depth.

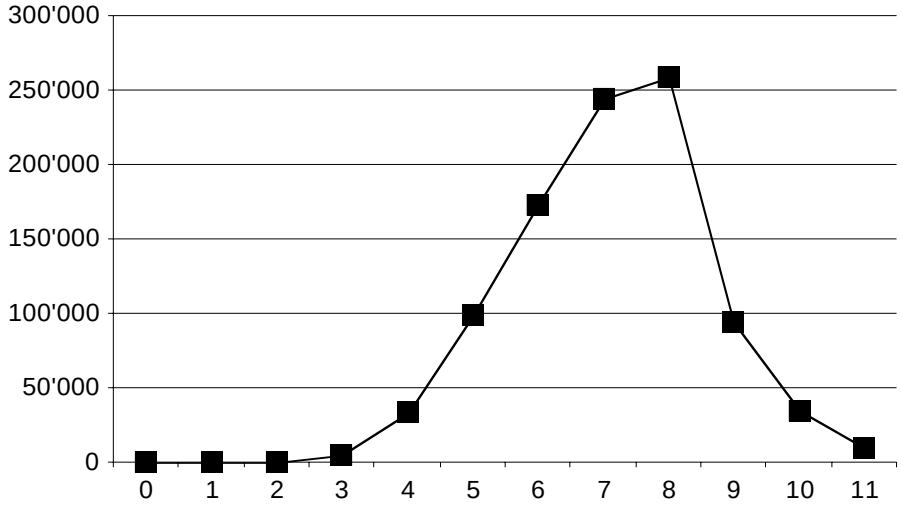

Figure 6.7: The depth distribution of the nodes in the Chess opening book.

Figure 6.8 shows the drop-out diagram of the Chess opening book. The dashed line is the angle of the expansion frontier as defined by our choice of the parameter ω. Nodes with value 0 at depth 9 and nodes with value 1 at depth 5 have the highest expansion priority.

Opening Book Evaluation

First we compare our book with about 950,000 positions to human Chess opening knowledge. As a measure of the size of human Chess opening knowledge we chose the "Moderne Eröffnungstheorie", a series of 24 volumes written by several russian Chess masters. Each volume contains about 240 pages of annotated Chess variants, and by sampling 40 pages from 4 different volumes [37, 38, 39, 51] of the series we found that about 74 positions occur on every page. We conclude that the Chess opening knowledge in this book series contains about 426,000 positions.

On first sight it might seem that our book is about two times larger than the human book knowledge. However the number of positions in our opening book and the number of positions in the human literature cannot be compared directly with each other, because the knowledge as presented in the human literature is already heavily filtered; the human literature contains only positions which are considered to be part of a principal variation. The

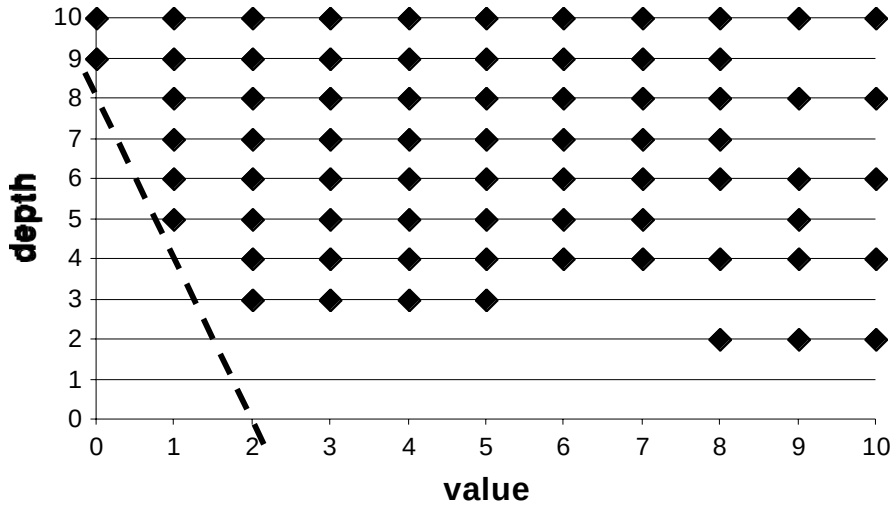

Figure 6.8: The drop-out diagram of the Chess opening book. The value 10 is equivalent to one pawn. The shallowest line of the principal variations is 9 ply deep.

Depth	Nodes	Solved	Predecessors	Degree
0	1	0	4	20
1	20	0	114	20
2	400	0	1,305	22
3	4,474	8	11,082	22
4	33,140	85	67,794	25
5	99,211	127	231,843	27
6	173,213	60	348,577	29
7	244,530	104	545,928	30
8	258,007	21	494,721	31
9	94,107	18	196,336	31
10	34,181	34	49,012	33
11	9,392	0	13,911	33
Total	950,676	457	1,960,627	29

Table 6.6: Book statistics for Chess.

positions which are only reached with at least one bad move are not mentioned, the reader is assumed to be able to figure out for himself if a move immediately loses a queen. On the other hand, our book contains many positions which are not part of a principal variation. To find out that a certain move is bad, our book has to add the position after that move to the book and then let the engine evaluate the position. Therefore an automatically generated opening book contains many positions which are only reached with one bad move.

To compare the sizes of our book and human opening knowledge we have to use the additional observation that the branching factor of principal variations in the human literature is close to one. Therefore an automatically constructed opening book in Chess has to contain about $30 \times 426,000$ positions to be equal to human Chess opening knowledge. In other words, our book corresponds to about $32,000$ positions of human Chess knowledge. From this relatively small number and from the shallowness of the book it is obvious that the book is not very useful (neither for computers nor for humans).

As mentioned earlier the game graph of Chess contains many cycles. To show an example of `cycle-draw` values we manually forced the game Sax 2610 – Seirawan 2595 (Bruxelles, 1988 [42]) into the book. Figure 6.9 shows the position after the 10th move of black, and Figure 6.10 shows the game graph starting from the same position. All nodes in the graph are

labeled with their value. For better readability the graph contains only the best moves deviating from the game.

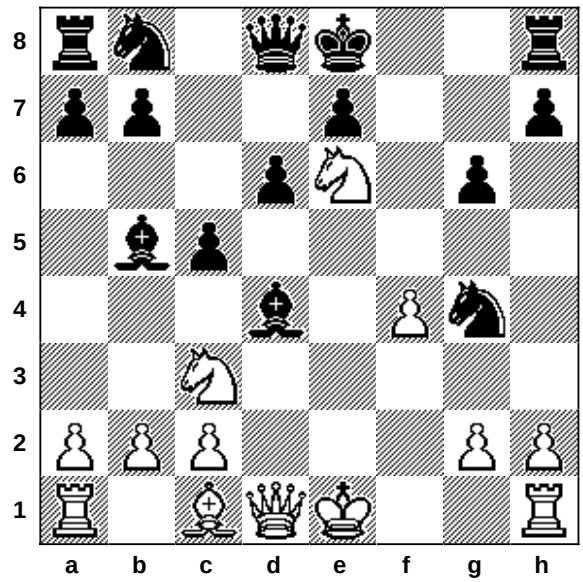

Figure 6.9: Sax – Seirawan (Bruxelles, 1988): 1.e4 d6 2.d4 Nf6 3.Nc3 g6 4.f4 Bg7 5.Nf3 c5 6.Bb5 Bd7 7.e5 Ng4 8.e6 fxe6 9.Ng5 Bxb5 10.Nxe6 Bxd4 **(Diagram)**11.Nxd8 Bf2+ 12.Kd2 Be3+ 13.Ke1 Bf2+ 1/2–1/2.

As the example shows, the `cycle-draw` values are an intuitive aid to model the human way of thinking about such positions: the positions in the cycle are not solved in the strong mathematical sense, but they are solved for all practical purposes. It is quite possible that the game of Chess will turn out to be a `draw`, and that the principal variations will all end in a position repetition. In that case we conjecture that the game will first be solved "for all practical purposes" before any proven value will be known.

6.5 Nine Men's Morris

Game Properties

Nine Men's Morris has a state-space complexity smaller than 10^{10} [15], and an estimated search-tree complexity of 10^{50} [4]. On the first 18 plies (opening phase) of the game the graph is acyclic, after that (middle- and endgame phase) the graph contains cycles. The game was solved and proven to be a `draw` by Ralph Gasser in 1993 [15].

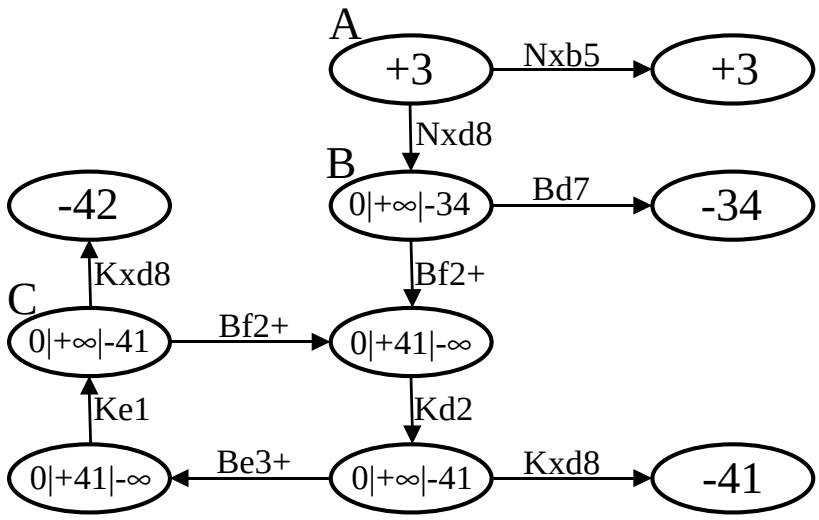

Figure 6.10: Node A represents the position in Figure 6.9. In every node only the best move deviating from the game is shown. Nodes B and C represent different positions because in B White still has castling rights. Inside the cycle White always has only one move. The best Black move leaving the cycle is always Kxd8, once with a value of -41 and once with a value of -42 (+10 being the value of a pawn).

Search Engine

The search engine was written by Jörg Kreienbühl as part of his diploma thesis project [24]. The engine uses the endgame databases of Ralph Gasser [15].

Opening Book Statistics

Figure 6.11 shows the depth distribution of the nodes in the book. About 260,000 nodes were expanded in the depth range from 0 to 10. Most of the leaf nodes are at depth 9, because the search engine was configured to solve these positions with 9 ply searches into the endgame databases. For experimental purposes some lines were expanded to depth 10. Table 6.7 shows more details of the book statistics. The 'Nodes' column shows the number of nodes at each depth. The 'Solved' column shows the number of nodes that were solved. The nodes at depths 9 and 10 were solved by the search engine, the others through backpropagation. Opening book expansion was stopped when the start node was proved to be a `draw`. The 'Predecessors' column shows that, on average, every position in the book has about 3 predecessors which are also in the book. This high ratio of transpositions is explained by the fact that most of the times two consecutive moves of one player can also be played in the inverse order. The 'Degree' column shows the average number of possible moves at each depth. At the beginning of the game the degree equals the number of empty fields on the board, later on the degree is slightly larger because in capture moves there is a choice in which opponent stone should be captured.

Opening Book Evaluation

The main purpose of our work on Nine Men's Morris was to construct an opening book. As a side effect we confirmed the previous result of Ralph Gasser that the game is a `draw` [15]. Because the correctness of the endgame databases has been verified by Fabian Mäser [35], and because we solved the opening phase with a different engine, all parts of the proof have now been double-checked independently. In addition we also strengthen the result: now the game is 'weakly solved' instead of 'ultra-weakly solved', because the first player can determine an optimal move from any position within a few seconds.

Our solution of Nine Men's Morris also demonstrates the ability of an opening book tool to efficiently close the gap between a start position and endgame databases: while there are about 10^{10} opening phase positions, only 260,000 of them had to be added to the book to solve the game.

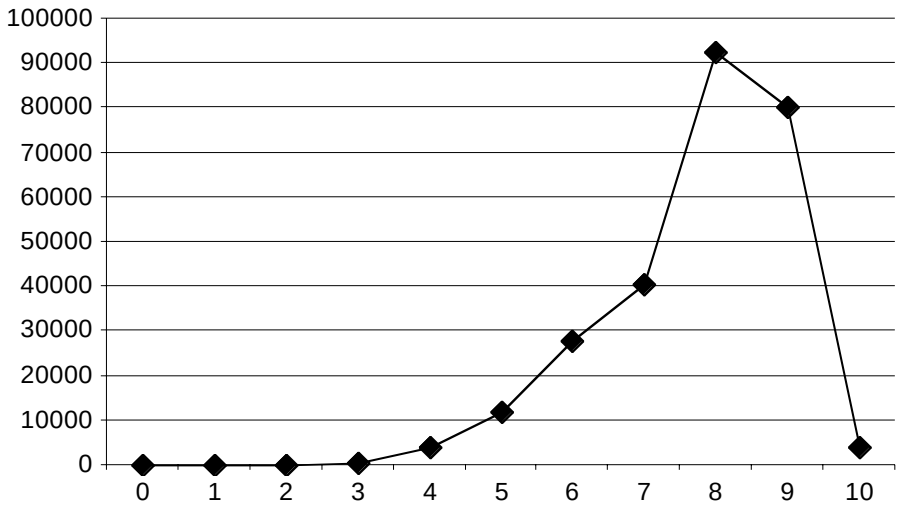

Figure 6.11: The depth distribution of the nodes in the Nine Men's Morris opening book.

Depth	Nodes	Solved	Predecessors	Degree
0	1	1	0	24.0
1	4	1	24	23.0
2	46	2	92	22.0
3	428	15	1,012	21.0
4	4,054	24	8,694	20.2
5	11,885	373	37,970	19.3
6	27,889	621	76,119	18.8
7	40,446	6,185	137,293	18.0
8	92,288	10,832	199,492	17.5
9	80,147	79,941	258,961	16.7
10	3,831	3,827	10,704	16.4
Total	261,019	101,822	730,361	18.0

Table 6.7: Book statistics for Nine Men's Morris.

6.6 Othello

Game Properties

Othello has a state-space complexity of about 10^{28}, an estimated search-tree complexity of 10^{58} and and average branching factor of 10 [4]. The game graph is acyclic and at most 60 plies deep.

Search Engine

As a search engine we used Wolpers, written by Alvaro Fussen [14]. The engine was configured to do full endgame searches from ply 43.

Opening Book Statistics

Figure 6.12 shows the depth distribution of the nodes in the book. About 1,500,000 nodes were expanded in the depth range from 0 to 42, with a distinctive peak around ply 20. Table 6.8 shows more details of the book statistics. The 'Nodes' column shows the number of nodes at each depth. The 'Solved' column shows the number of nodes that were solved. These values are all zero because nodes are only solved by endgame searches, and the book is not deep enough for that. The 'Predecessors' column shows that, on average, every position in the book has only slightly more than one predecessor which is also in the book. This proves the fact that the game graph of Othello does not contain many transpositions. The 'Degree' column shows the average number of possible moves at each depth.

Figure 6.13 shows the drop-out diagram of the Othello opening book. The dashed line is the angle of the expansion frontier as defined by our choice of the parameter ω. Several nodes at various depths are candidates for further expansion.

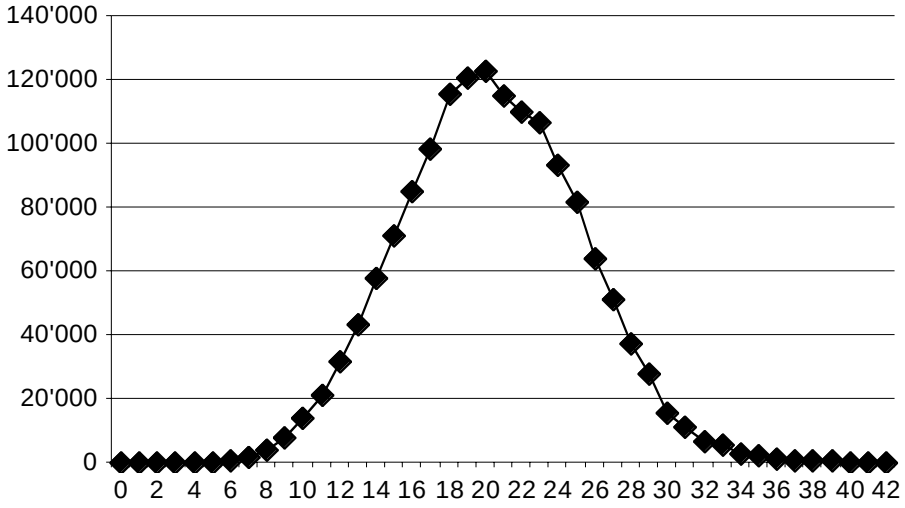

Figure 6.12: The depth distribution of the nodes in the Othello opening book.

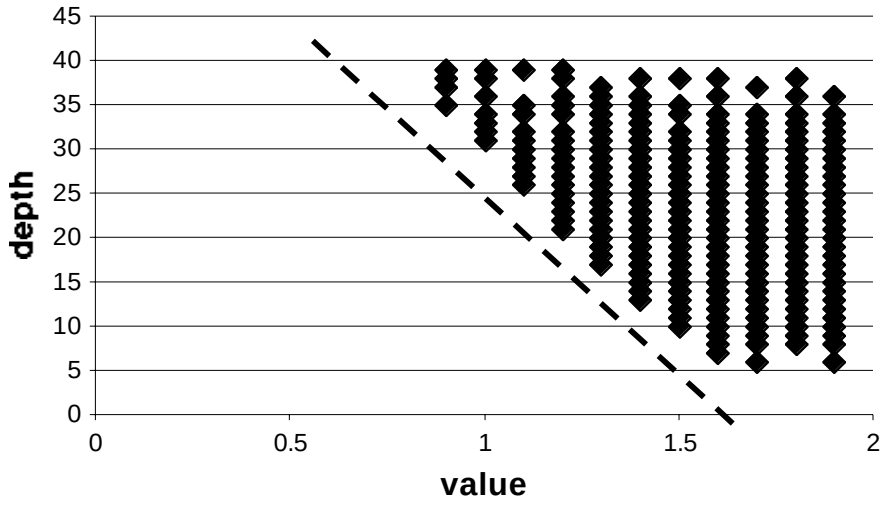

Figure 6.13: The drop-out diagram of the Othello opening book. The value 1 is equivalent to one piece. The shallowest line of the principal variations is 35 plies deep.

Depth	Nodes	Solved	Predecessors	Degree
0	1	0	0	4.0
1	1	0	4	3.0
2	3	0	3	4.7
3	14	0	14	4.4
4	60	0	61	5.7
5	180	0	195	5.9
6	631	0	657	7.1
7	1,743	0	1,957	7.3
8	3,873	0	4,227	8.5
9	7,574	0	8,414	8.7
10	13,794	0	15,052	9.6
11	21,300	0	23,076	10.0
12	31,717	0	34,407	10.5
13	43,153	0	46,142	11.0
14	57,658	0	62,251	11.3
15	71,201	0	76,382	11.8
16	85,115	0	91,449	12.0
17	98,150	0	105,314	12.3
18	115,345	0	124,032	12.7
19	120,516	0	129,443	12.8
20	123,041	0	132,374	13.3
21	115,097	0	123,973	13.1
22	109,766	0	118,307	13.6
23	106,424	0	114,325	13.2
24	93,372	0	100,877	13.9
25	81,636	0	87,721	13.2
26	63,639	0	68,547	14.0
27	51,232	0	54,867	13.1
28	37,175	0	39,838	13.8
29	27,562	0	29,509	12.9
30	15,630	0	16,496	13.3
31	11,304	0	11,759	12.6
32	6,631	0	6,867	13.1
33	5,334	0	5,500	12.2
34	3,004	0	3,107	12.6
35	2,253	0	2,301	11.5
36	1,171	0	1,199	11.9
37	782	0	788	10.3
38	544	0	561	11.8
39	315	0	330	9.5
40	72	0	72	10.6
41	29	0	29	9.4
42	9	0	9	8.1
Total	1,528,051	0	1,642,436	12.3

Table 6.8: Book statistics for Othello.

Chapter 7

Conclusions

To a large degree the advances in exhaustive search are triggered by advances in technology. Memory chips with larger capacities mean we can solve larger problems, and microprocessors with higher clock frequencies mean that we can solve them faster. When confronted with a large exhaustive search problem, one solution is to simply wait until technology catches up. However, there will always be a search problem for which the solution seems to be just out of reach, which gives us the motivation to look for algorithmic improvements that make that problem solvable even with today's technology.

In this thesis we presented such algorithmic improvements both for backward and forward search. In backward search, the main bottleneck for solving large exhaustive search problems with retrograde analysis is the size of main memory. If the whole problem does not fit into main memory then we have to store part of the problem space on a disk. The resulting latency in accessing the data on disk may increase the running time of the algorithm by a factor of 100, thus making the solution of the problem infeasible. In Chapter 2 we presented our new dual-indexing algorithm. When applied to retrograde analysis in the game of Awari, the new algorithm significantly reduced the number of disk accesses with good performance even when only 10% of the state space fit into main memory. The new algorithm was used to calculate a 47.6 GBytes database on a 1 GByte machine.

For forward search we implemented an opening book construction tool called OPLIB. It was used to implement and test the new opening book expansion strategy presented in Chapter 3. Drop-out expansion is a best-first expansion strategy which strikes a balance between expanding leaf nodes with good values and expanding leaf nodes at shallow depths. This ensures the usefulness of the opening book no matter how the opponent plays the opening. If the opponent plays only good moves, then we stay in the book longer, and therefore save time which can be used later for deeper searches.

If the opponent plays bad moves, then we might drop out of the book earlier, but we will leave the opening book in a superior position.

For the representation of position values in opening books we proposed new types of attributed values in Chapter 4. The `at-least-draw`, `at-most-draw` and `cycle-draw` values give us a more accurate estimate of the game-theoretic value of a position. This allows for a better control of book expansion and game solving.

The techniques presented in this thesis and their combined application to the game of Awari played a significant role for the implementation of a competitive computer Awari program. As a result, the author's Awari engine Marvin won the computer Awari world championship [30]. Moreover, as a result of the use of opening book expansion in other games we confirmed and strengthened various previous results, see Chapter 6.

Appendix A

Awari Rules

Awari is a simple board game for two players, usually called *North* and *South*. The board contains two rows of six pits and each player is in charge of one row. An extra pit for each player is used to collect captured stones. The game starts with 48 stones on the board, as shown in Figure A.1. South makes the first move. The player who captures more than 24 stones is the winner. If both players capture 24 stones the game is a `draw`.

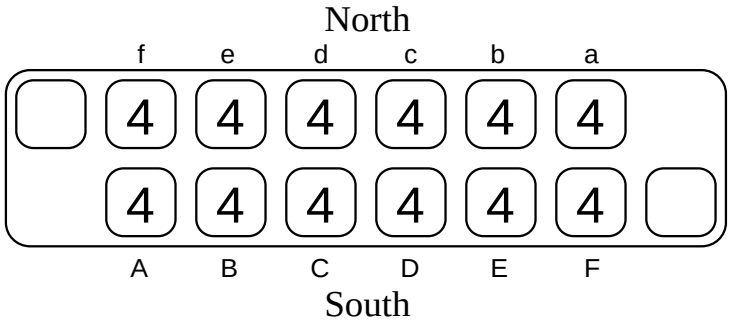

Figure A.1: The start position.

To make move, a player chooses any non-empty pit on his side of the board, takes out all the stones and drops them one by one counter-clockwise into the other pits, starting with the next pit. Whenever a move starts with twelve or more stones, the pit from where the move started is skipped in the sequence of dropping stones. In Figure A.2, F5 would be a legal move for South ('F' is the start pit and '5' is the number of stones moved). After the move, pit F would be empty and pits a,b,c,d and e each would contain one additional stone.

If the last stone of a move falls into a pit of the opponent containing two or three stones (including the ones that just have been dropped), these stones

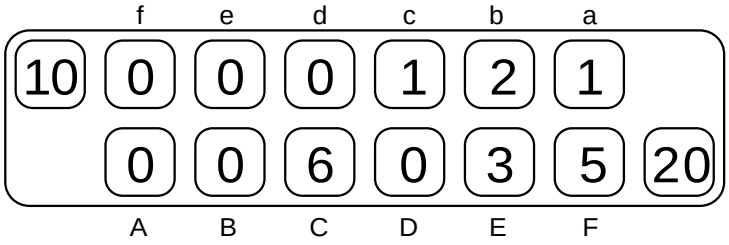

Figure A.2: South has two legal moves: F5 and E3×5. C6×7 is illegal.

are captured, as well as all stones from the preceding pits, as long as they belong to the opponent and contain two or three stones. In Figure A.2, the move E3×5 would capture 5 stones.

If a player runs out of moves, the opponent captures all the remaining stones. If a position is repeated, the remaining stones are divided between the players (including an occasional odd stone). If the player to move has a choice, then he must make a move in such a way that the opponent does not run out of moves. For example, in Figure A.2 the move C6×7 is illegal.

Appendix B

Awari Endgame Database Statistics

The rows are labeled with the difference between the number of stones captured by the first player and the number of stones captured by the second player. The columns are labeled with the number of stones left on the board. For example the value 6,828 in the cell -4/8 means that there are 6,828 8-stone positions where the first player captures 2 and the second player captures 6 stones. (Assuming optimal play by both sides.)

	0	1	2	3	4	5	6	7	8
-8									12,370
-7								8,948	
-6							4,242		228
-5						1,892		3	
-4					532		116	75	6,828
-3				177		2		2,948	
-2			40		110	85	772		9,977
-1		7		1		224		442	61
0	1		13	63	143		1,548	4,487	8,351
1		5		7		121		491	236
2			25		165	184	550		10,182
3				116		22	62	2,294	
4					415		302	239	6,972
5						1,838		145	168
6							4,784		507
7								11,752	
8									19,702

Table B.1: Awari endgame database statistics, part 1.

	9	10	11	12	13	14	15	16
-16								34,460
-15							34,835	
-14						27,769		1,114
-13					36,074		1,213	709
-12				29,152		1,176	715	46,499
-11			34,321		889	893	13,358	
-10		22,941		770	493	11,508		90,304
-9	24,408		790	574	19,911		69,533	26,185
-8		211	329	15,547		81,311	34,106	58,024
-7	60	34	24,144		31,986	24,031	50,127	41,199
-6	192	13,906		42,517	31,669	58,180	68,751	438,669
-5	15,183		20,784	15,404	41,136	32,038	238,847	200,956
-4		25,783	22,017	49,215	68,851	300,409	248,812	532,060
-3	7,037	3,938	25,218	12,818	159,908	132,793	440,305	450,420
-2	10,080	33,687	58,269	229,747	185,455	381,555	549,991	1,686,085
-1	5,407	752	54,813	43,954	242,574	228,981	918,495	687,719
0	20,308	93,743	92,787	197,894	312,251	905,779	591,903	1,445,158
1	4,287	738	60,937	50,657	282,308	271,478	1,077,291	796,128
2	13,081	38,773	72,301	287,396	233,025	520,046	701,647	2,148,284
3	6,942	7,722	36,111	27,892	264,506	212,762	726,614	741,567
4		34,896	30,814	84,465	112,279	518,286	400,852	977,733
5	18,932		35,658	29,073	102,112	92,458	582,414	433,693
6	458	25,402		80,045	59,923	150,334	156,984	949,245
7	390	298	51,041		88,712	65,349	185,874	164,754
8		810	874	55,585		186,835	97,298	237,137
9	41,195		1,533	1,246	85,393		216,425	113,209
10		49,082		1,816	1,495	96,016		316,761
11			82,691		2,718	2,910	121,523	
12				96,311		3,982	3,018	187,810
13					132,476		5,586	4,451
14						150,521		6,983
15							189,643	
16								220,579

Table B.2: Awari endgame database statistics, part 2.

	17	18	19	20	21	22	23
-23							243,099
-22						253,255	
-21					254,025		1,260
-20				191,172		829	1,251
-19			144,669		933	545	282,445
-18		86,123		841	410	295,383	
-17	42,905		1,217	740	266,450		307,605
-16		985	540	177,281		252,492	42,746
-15	1,493	774	59,116		150,160	28,929	453,083
-14	590	76,827		105,949	26,949	386,473	55,219
-13	76,859		113,092	24,178	175,806	48,454	871,469
-12		118,957	25,937	175,793	56,504	640,974	288,839
-11	69,399	24,163	192,896	48,708	635,342	252,487	1,174,136
-10	28,839	138,366	54,510	642,857	273,871	1,079,746	761,939
-9	64,013	42,978	467,274	270,978	1,112,744	732,323	3,760,360
-8	63,747	484,703	293,617	890,816	788,118	3,584,717	1,934,855
-7	379,260	254,513	793,781	713,691	3,119,656	1,882,065	6,468,322
-6	285,746	673,845	771,474	3,011,026	1,895,376	5,606,957	4,384,820
-5	604,115	623,720	2,571,382	1,722,876	5,180,632	4,059,524	13,384,230
-4	693,040	2,510,685	1,637,473	4,458,973	3,820,632	12,214,084	7,770,039
-3	1,786,369	1,274,500	3,780,934	3,059,399	9,768,828	6,879,520	20,472,752
-2	1,143,933	2,878,883	2,797,775	8,546,121	5,793,223	16,697,549	12,790,789
-1	2,232,281	1,839,168	5,863,824	4,459,475	13,371,559	9,798,515	29,321,280
0	1,609,187	4,768,086	3,310,844	9,263,057	7,877,744	22,897,612	15,354,066
1	2,542,408	2,118,595	6,638,702	4,873,157	14,461,718	10,788,634	32,291,319
2	1,423,911	3,683,748	3,454,533	10,400,148	7,004,962	20,129,882	15,414,994
3	2,705,626	1,815,924	5,291,792	4,353,606	13,516,752	9,281,073	27,163,537
4	1,170,387	4,053,817	2,563,688	7,021,373	5,910,671	18,323,654	11,740,549
5	1,287,552	1,333,837	4,921,937	3,145,834	9,062,415	7,207,310	22,600,192
6	615,430	1,601,036	1,718,101	6,310,717	3,859,539	11,063,487	8,622,147
7	1,092,262	716,356	2,055,589	1,965,644	7,485,818	4,473,208	13,766,879
8	216,056	1,456,782	889,144	2,503,309	2,340,451	9,136,900	5,160,163
9	300,233	237,249	1,721,107	1,017,008	3,174,547	2,620,350	10,639,880
10	142,239	434,877	288,293	2,216,108	1,165,964	3,669,143	3,005,089
11	370,137	169,533	576,043	324,822	2,589,801	1,311,973	4,398,254
12		513,741	202,487	679,125	381,331	3,046,691	1,508,672
13	251,878		610,431	239,700	827,478	418,419	3,822,854
14	5,218	300,010		720,336	279,751	1,236,809	473,355
15	9,558	6,857	339,718		915,600	319,476	1,498,643
16		9,893	7,672	551,255		1,153,625	364,308
17	259,509		12,785	9,603	733,474		1,383,768
18		347,759		13,740	9,869	930,456	
19			454,923		16,097	11,410	1,157,672
20				562,899		18,187	13,208
21					719,280		20,505
22						803,600	
23							927,168

Table B.3: Awari endgame database statistics, part 3.

	24	25	26	27	28	29
-29						461,324
-28					347,309	
-27				297,299		1,203
-26			261,189		1,267	860
-25		244,084		1,618	977	380,486
-24	235,353		1,288	1,189	325,575	
-23		1,307	1,169	215,966		427,433
-22	890	2,011	183,254		377,746	39,485
-21	870	180,775		324,228	42,618	620,415
-20	228,866		278,102	43,175	475,031	83,999
-19		271,234	49,222	404,600	75,210	1,270,433
-18	328,160	54,013	397,792	74,995	1,128,382	312,755
-17	47,059	416,693	77,916	1,074,397	320,393	1,624,515
-16	450,764	68,213	1,127,524	320,193	1,500,391	733,230
-15	88,938	1,090,063	347,647	1,607,556	732,524	4,230,434
-14	995,558	355,539	1,510,647	798,548	4,342,897	1,826,143
-13	356,169	1,536,911	752,748	4,304,687	1,961,417	7,695,531
-12	1,414,692	885,779	4,594,263	2,038,420	7,727,382	4,551,984
-11	781,492	4,467,176	2,151,966	8,098,784	4,696,825	18,712,693
-10	4,064,136	2,142,064	7,977,335	5,055,625	19,360,464	9,614,659
-9	1,957,512	7,253,524	4,818,960	19,132,541	10,068,795	34,284,855
-8	6,601,638	4,613,768	17,691,926	9,720,977	33,366,536	19,997,762
-7	4,421,066	16,520,574	9,256,306	31,312,981	19,404,874	67,035,493
-6	15,353,510	8,877,654	28,535,230	18,548,350	62,790,716	35,834,914
-5	8,563,115	26,690,922	17,615,921	57,651,735	34,066,512	109,166,720
-4	23,304,597	16,503,750	52,332,367	31,530,521	98,228,502	60,017,087
-3	14,604,434	44,431,147	28,975,242	87,373,503	54,880,375	169,257,602
-2	38,236,499	24,754,815	72,892,024	48,422,228	145,475,257	87,819,805
-1	20,341,031	59,677,862	39,942,586	117,211,310	74,915,275	223,185,818
0	44,357,445	31,717,625	91,990,780	58,669,738	172,674,564	109,030,172

Table B.4: Awari endgame database statistics, part 4.

	24	25	26	27	28	29
0	44,357,445	31,717,625	91,990,780	58,669,738	172,674,564	109,030,172
1	22,117,967	64,592,557	43,587,682	127,507,635	81,029,791	239,626,083
2	45,603,797	29,617,409	86,442,306	57,448,687	170,526,617	103,654,645
3	19,641,398	58,677,777	38,060,622	112,593,824	71,824,851	216,937,797
4	34,506,278	24,433,005	75,107,858	46,118,073	139,398,973	86,626,451
5	14,453,288	43,335,085	29,298,512	91,388,510	55,082,367	169,371,315
6	28,085,753	16,875,753	51,542,064	34,148,128	109,017,715	63,798,616
7	9,970,443	33,480,187	19,603,597	61,647,591	39,365,173	128,095,459
8	15,927,689	11,503,239	39,347,743	22,509,411	71,992,791	44,513,129
9	5,852,540	18,992,568	13,111,100	46,315,459	25,413,943	81,846,374
10	12,508,979	6,731,389	22,440,013	14,855,454	52,446,685	27,835,738
11	3,394,848	14,749,300	7,517,971	25,498,902	16,161,806	58,529,637
12	5,415,335	3,888,927	16,833,708	8,255,062	28,601,989	17,827,459
13	1,732,680	6,269,652	4,172,887	18,836,248	9,097,713	32,066,853
14	4,452,714	1,918,235	7,183,052	4,684,801	21,227,668	9,753,318
15	546,376	5,227,590	2,099,935	8,244,834	4,999,222	23,212,449
16	1,822,448	558,637	5,989,775	2,256,756	8,982,467	5,367,361
17	393,873	2,149,286	616,123	6,618,995	2,428,643	10,131,504
18	1,650,134	429,225	2,446,987	654,715	7,524,351	2,596,893
19		1,845,759	456,961	2,845,888	695,554	8,482,044
20	1,330,211		2,154,054	476,633	3,283,203	761,530
21	13,858	1,552,314		2,472,541	503,314	3,840,171
22	22,208	16,525	1,832,103		2,802,535	518,004
23		24,109	15,895	2,139,695		3,169,951
24	1,049,289		24,753	16,677	2,577,097	
25		1,179,265		25,444	16,999	2,927,746
26			1,343,047		24,759	16,678
27				1,527,161		24,179
28					1,742,004	
29						2,052,246

Table B.5: Awari endgame database statistics, part 5.

	30	31	32	33	34
-34					948,306
-33				881,583	
-32			824,833		1,583
-31		697,823		1,785	1,066
-30	578,961		841	1,231	819,219
-29		970	973	852,875	
-28	897	1,256	830,449		782,095
-27	839	685,224		744,479	54,995
-26	486,063		617,105	42,522	1,396,500
-25		526,717	39,874	1,259,129	89,589
-24	474,580	37,413	1,072,137	83,741	2,505,270
-23	38,973	905,955	86,585	2,161,336	360,652
-22	809,137	67,834	1,835,945	352,978	3,079,719
-21	103,160	1,678,356	352,222	2,828,631	930,411
-20	1,461,525	326,470	2,586,583	817,952	6,122,301
-19	319,432	2,307,867	802,523	5,576,279	1,982,494
-18	1,777,595	779,690	5,098,871	1,890,790	9,146,641
-17	700,221	4,582,031	1,782,126	8,592,354	4,072,232
-16	4,434,651	1,730,336	7,643,162	4,080,081	17,439,746
-15	1,793,765	7,477,427	4,023,486	16,988,447	8,115,057
-14	7,285,983	4,142,783	17,064,474	8,272,515	29,903,322
-13	4,258,789	17,300,690	8,667,983	30,295,809	16,334,978
-12	18,221,822	8,833,931	31,068,870	17,034,747	58,368,912
-11	9,479,031	32,687,170	17,999,663	61,337,315	31,750,330
-10	33,391,939	19,143,456	64,843,303	33,296,722	106,843,891
-9	19,535,652	67,029,382	35,595,732	113,709,165	60,291,029
-8	68,437,188	36,157,205	117,276,131	64,339,968	202,342,100
-7	36,898,814	119,735,086	65,931,743	210,947,279	112,582,875
-6	116,284,342	66,544,140	213,681,647	114,856,020	362,549,635
-5	64,274,682	206,624,395	114,681,026	362,957,495	198,056,654
-4	190,630,010	109,075,681	344,639,679	193,941,924	613,898,148
-3	99,885,149	312,203,168	179,933,810	567,370,580	312,835,808
-2	268,380,775	159,407,509	499,926,160	282,000,952	885,516,383
-1	135,382,286	414,067,129	239,429,941	750,409,683	422,402,448
0	329,539,527	192,547,212	592,741,535	339,869,158	1,077,956,188

Table B.6: Awari endgame database statistics, part 6.

	30	31	32	33	34
0	329,539,527	192,547,212	592,741,535	339,869,158	1,077,956,188
1	146,382,668	445,501,970	257,801,186	804,135,425	454,051,201
2	312,680,181	187,061,985	580,529,439	329,345,133	1,028,504,726
3	129,216,314	396,284,378	231,558,571	721,751,925	401,115,068
4	267,079,494	155,368,326	480,873,852	274,582,719	859,193,573
5	102,206,122	318,447,533	179,875,691	559,231,434	311,368,891
6	198,872,990	116,502,190	364,145,925	200,585,232	626,239,246
7	72,388,600	226,840,564	129,004,903	405,148,088	220,362,778
8	145,241,196	79,889,179	252,198,425	141,462,865	441,691,198
9	49,018,609	161,661,902	87,968,563	276,790,147	150,957,608
10	91,195,051	53,809,812	177,365,838	93,721,119	295,358,274
11	30,723,950	100,739,637	57,458,154	189,830,866	99,499,636
12	64,965,929	32,814,431	108,505,776	60,960,621	201,802,772
13	19,203,793	70,268,247	35,358,865	117,035,177	64,429,679
14	34,726,226	20,642,105	76,358,362	37,480,333	125,320,833
15	10,529,295	38,268,578	22,010,423	82,047,001	39,623,776
16	25,666,321	11,183,313	41,488,732	23,347,297	87,990,324
17	5,716,944	28,057,308	11,925,609	45,620,736	24,611,922
18	11,234,920	6,120,104	30,903,106	12,707,575	49,499,851
19	2,792,942	12,857,951	6,495,797	33,893,922	13,497,191
20	9,574,592	2,968,223	14,347,211	6,846,475	37,163,673
21	825,675	10,838,209	3,162,539	16,056,576	7,331,086
22	4,481,090	830,086	12,126,133	3,334,639	17,976,393
23	544,505	5,084,600	920,841	13,793,819	3,526,943
24	3,543,891	567,483	5,913,194	955,488	15,612,419
25		3,966,308	599,437	6,842,812	1,011,835
26	3,382,628		4,494,214	635,791	7,860,269
27	17,371	3,957,524		5,089,031	681,523
28	23,667	18,149	4,473,687		5,680,254
29		24,192	19,290	5,031,537	
30	2,361,216		24,697	21,923	5,617,946
31		2,682,783		26,915	23,206
32			3,016,477		27,687
33				3,230,986	
34					3,483,552

Table B.7: Awari endgame database statistics, part 7.

	35	36	37	38	39	40
-14	106,433,189					
-13	56,307,971	192,402,777				
-12	29,697,687	95,059,460	333,084,328			
-11	100,799,622	52,006,042	161,140,295	559,306,553		
-10	56,016,351	174,463,098	88,617,647	265,847,982	916,421,450	
-9	188,779,461	97,678,963	294,492,420	150,537,480	447,273,434	
-8	105,629,019	323,436,150	167,856,412	504,729,720	254,581,825	
-7	348,198,523	182,972,378	560,294,086	286,839,884	855,228,661	
-6	196,094,310	605,312,549	315,052,775	957,187,874	492,369,013	
-5	624,551,247	333,557,304	1,027,362,150	537,403,909	1,644,075,038	
-4	330,768,201	1,037,939,616	557,039,062	1,727,385,463	909,122,797	
-3	988,897,846	534,929,234	1,689,562,105	908,417,444	2,814,358,086	
-2	489,553,590	1,555,836,520	844,245,880	2,653,356,924	1,430,578,535	
-1	1,337,655,782	738,002,652	2,335,460,580	1,269,402,934	4,011,201,003	15,885,256,635
0	602,028,938	1,898,144,926	1,043,646,738	3,315,952,673	1,810,860,804	5,697,177,482
1	1,436,469,545	792,758,539	2,511,910,933	1,368,668,443	4,331,469,299	26,043,582,853
2	571,113,849	1,813,852,483	988,673,102	3,116,239,469	1,686,748,696	
3	1,259,854,492	688,025,632	2,174,289,050	1,175,721,928	3,662,571,578	
4	469,045,954	1,468,121,199	794,644,058	2,479,272,464	1,317,733,511	
5	975,447,519	526,827,700	1,631,808,113	864,872,842	2,679,810,595	
6	344,261,322	1,067,303,647	564,214,359	1,737,542,126	910,250,725	
7	682,297,614	366,041,834	1,130,697,259	590,259,694	1,794,374,309	
8	234,909,527	724,098,607	383,475,918	1,170,366,419	603,314,853	
9	469,784,997	246,875,120	753,445,114	393,409,640	1,187,427,653	
10	158,723,099	491,480,370	254,262,022	769,914,442	3,593,966,935	
11	312,087,486	164,773,205	506,730,229	2,333,279,957		
12	103,904,986	325,725,938	1,483,195,733			
13	213,013,077	919,507,674				
14	548,457,992					

Table B.8: Awari endgame database statistics, part 8. From the 35-stone database on we only calculated the value ranges which are relevant to determine `win`, `loss` and `draw`. For example in the 39-stone database, all configurations with a value ≥ 10 are wins, because the opponent has captured at most 9 stones. The 40-stone database is incomplete.

Bibliography

[1] J. D. Allen. A Note on the Computer Solution of Connect-Four. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, pages 134–135, Chichester, England, 1989. Ellis Horwood.

[2] L. V. Allis and P. N. A. Schoo. Qubic Solved Again. In H. J. van den Herik and L. V. Allis, editors, *Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad*, pages 192–204, Chichester, England, 1992. Ellis Horwood.

[3] L. V. Allis, H. J. van den Herik, and M. van der Meulen. Databases in Awari. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2: The Second Computer Olympiad*, pages 73–86, Chichester, England, 1991. Ellis Horwood.

[4] L. Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, The Netherlands, 1994.

[5] Henry Baird and Ken Thompson. Reading Chess. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(6):552–559, 1990.

[6] Henri Bal and Victor Allis. Parallel Retrograde Analysis on a Distributed System. In *1995 ACM/IEEE Supercomputing Conference*, pages 2010–2041, 1995.

[7] D. F. Beal. Mixing Heuristic and Perfect Evaluations: Nested Minimax. *ICCA Journal*, 7(1):10–15, 1984.

[8] H.J. Berliner. The B* Tree Search Algorithm: A Best-First Proof Procedure. *Artificial Intelligence*, 12:23–40, 1979.

[9] D. M. Breuker and J. Gnodde. The AST 4th Computer Olympiad. *ICCA Journal*, 15(3):152–153, 1992.

[10] Michael Buro. Toward Opening Book Learning. *ICCA Journal*, 22(2):98–102, 1999.

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

[12] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extensible Hashing – A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3):315–334, 1979.

[13] Martin Fierz. Cake++ Homepage, 2002. http://www.fierz.ch /checkers.html.

[14] Alvaro Fussen. Personal communication, 1999.

[15] Ralph Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search*. PhD thesis, ETH Zürich, 1995.

[16] Ernst A. Heinz. Endgame Databases and Efficient Index Schemes for Chess. *ICCA Journal*, 22(1):22–32, 1999.

[17] Feng-hsiung Hsu, Murray S. Campbell, and A. Joseph Hoane Jr. Deep Blue System Overview. In *1995 International Conference on Supercomputing*, pages 240–244. ACM Press, 1995.

[18] Robert Hyatt. Crafty ftp site, 2002. ftp.cis.uab.edu/pub/hyatt.

[19] Robert Hyatt. Homepage Hyatt/Crafty, 2002. http://www.cis.uab.edu /info/faculty/hyatt/hyatt.html.

[20] Geoffrey Irving, Jeroen Donkers, and Jos Uiterwijk. Solving Kalah. *ICGA Journal*, 23(3):139–147, 2000.

[21] Thoralf Karlsson. The Swedish Rating List. *ICGA Journal*, 24(3):199, 2001.

[22] Peter Karrer. KQQKQP and KQPKQP≈. *ICGA Journal*, 23(2):75–84, 2000.

[23] Donald E. Knuth and R.W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[24] Jörg Kreienbühl. A Robot that Plays Nine Men's Morris. Eidgenössische Technische Hochschule, Zürich, Switzerland, 2001. Diploma thesis.

[25] Robert Lake, Jonathan Schaeffer, and Paul Lu. Solving Large Retrograde Analysis Problems Using a Network of Workstations. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess VII*, pages 135–162, Maastricht, the Netherlands, 1994. University of Limburg.

[26] Thomas R. Lincke. Perfect Play using Nine Men's Morris as an Example. Eidgenössische Technische Hochschule, Zürich, Switzerland, 1994. Diploma thesis.

[27] Thomas R. Lincke. Strategies for the Automatic Construction of Opening Books. In Tony Marsland and Ian Frank, editors, *LNCS 2063: Computers and Games*, pages 74–86. Springer, 2001.

[28] Thomas R. Lincke. Awari Endgame Database Statistics, 2002. http://www.shortestwin.com/awari/.

[29] Thomas R. Lincke and Ambros Marzetta. Large Endgame Databases with Limited Memory Space. *ICGA Journal*, 23(3):131–138, 2000.

[30] Thomas R. Lincke and Roel van der Goot. Marvin wins Awari Tournament. *ICGA Journal*, 23(3):173–174, 2000.

[31] David A. McAllester. Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, 35:287–310, 1988.

[32] Martin Müller. Solving 5×5 Amazons. In *The 6th Game Programming Workshop (GPW 2001)*, number 14 in IPSJ Symposium Series Vol.2001, pages 64–71, Hakone (Japan), 2001.

[33] Martin Müller. Arrow Homepage, 2002. http://www.cs.ualberta.ca/~mmueller/amazons/index.html.

[34] Eugene Nalimov, Christoph Wirth, and Guy Haworth. KQQKQQ and the Kasparov–World Game. *ICCA Journal*, 22(4):195–212, 1999.

[35] Jürg Nievergelt, Ralph Gasser, Fabian Mäser, and Christoph Wirth. All the Needles in a Haystack: Can Exhaustive Search Overcome Combinatorial Chaos? In Jan van Leeuwen, editor, *LNCS 1000: Computer Science Today*, pages 254–274. Springer, 1995.

[36] O. Patashnik. Qubic: 4x4x4 Tic-Tac-Toe. *Mathematics Magazine*, 53:202–216, 1980.

[37] Lew Polugajewski. *Königsindisch*. Number 16 in Moderne Eröffnungstheorie. Sportverlag Berlin, 1984.

[38] Lew Polugajewski. *Sizilianisch*. Number 3 in Moderne Eröffnungstheorie. Sportverlag Berlin, 1987.

[39] Lew Polugajewski. *Sizilianisch*. Number 4 in Moderne Eröffnungstheorie. Sportverlag Berlin, 1987.

[40] Chinook project. 8-stone checkers endgame databases, 2002. http://www.cs.ualberta.ca/~chinook/Software/software.html.

[41] John Romein. Personal Communication.

[42] Sax 2610 - Seirawan 2595. Chess Informant 46, VII-XII 1988, p. 9.

[43] Jonathan Schaeffer. Conspiracy Numbers. *Artificial Intelligence*, 43:67–84, 1989.

[44] Jonathan Schaeffer. *One Jump Ahead*. Springer-Verlag, 1997.

[45] Jonathan Schaeffer. Search Ideas in Chinook. In H. J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 19–30, 2000.

[46] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. Reviving the Game of Checkers. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2: The Second Computer Olympiad*, pages 119–136, Chichester, England, 1991. Ellis Horwood.

[47] Richard P. Stanley. *Enumerative Combinatorics Vol. 1*. Number 49 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1997.

[48] L. Stiller. Parallel Analysis of Certain Endgames. *ICCA Journal*, 12(2):55–64, 1989.

[49] G. Stockman. A Minimax Algorithm Better than Alpha-beta? *Artificial Intelligence*, 12:179–196, 1979.

[50] T. Ströhlein. *Untersuchungen über kombinatorische Spiele*. PhD thesis, Fakultät für allgemeine Wissenschaften der Technischen Hochschule München, Munich, 1970.

[51] Mark Taimanow. *Nimzowitsch–Indisch*. Number 12 in Moderne Eröffnungstheorie. Sportverlag Berlin, 1983.

[52] J. Tamplin and Guy Haworth. Ken Thompson's 6-Man Tables. *ICGA Journal*, 24(2):83–85, 2001.

[53] Ken Thompson. Retrograde Analysis of Certain Endgames. *ICCA Journal*, 9(3):131–139, 1986.

[54] Ken Thompson. 6-Piece Endgames. *ICCA Journal*, 19(4):215–226, 1996.

[55] Ken Thompson. The Longest: KRNKNN in 262. *ICGA Journal*, 23(1):35–36, 2000.

[56] J. W. H. M. Uiterwijk, H. J. van den Herik, and L. V. Allis. A knowledge-based approach to connect-four. the game is over: White to move wins! In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, pages 113–133, Chichester, England, 1989. Ellis Horwood.

[57] Roel van der Goot. Awari Retrograde Analysis. In Tony Marsland and Ian Frank, editors, *LNCS 2063: Computers and Games*, pages 87–95. Springer, 2001.

[58] Eric van Riet Paap. World Championship Computer Awari, 1995. http://groups.google.ch/groups?q=awari+marvin+lincke.

[59] Christoph Wirth and Jürg Nievergelt. Exhaustive and Heuristic Retrograde Analysis of the KPPKP Endgame. *ICCA Journal*, 22(2):67–80, 1999.

[60] R. Wu and D. F. Beal. Fast, Memory-Efficient Retrograde Algorithms. *ICGA Journal*, 24(3):147–159, 2001.

[61] Jing Yang, Simon Liao, and Mirek Pawlak. A Decomposition Method for Finding Solution in Game Hex 7x7. *International Conference On Application and Development of Computer Games in the 21st Century*, pages 96–111, 2001.

[62] Jing Yang, Simon Liao, and Mirek Pawlak. A New Solution for 7x7 Hex Game. *to appear*, 2002.

Curriculum Vitae

Name	Thomas Robert Lincke
Email	thomas@lincke.ch
Date of birth	September 15, 1969
Place of birth	Milwaukee, USA
Nationality	Switzerland/USA

1982 – 1988	C-Maturity (high school diploma), Kantonsschule Im Lee, Winterthur, Switzerland.
Oct '88–Feb '89	Internship as programmer at <Winterthur> Life Insurances.
1989 – 1994	Diploma Informatik Ingenieur (Software Engineer), ETH Zürich, Switzerland. Major in Computer Science and minor in Robotics.
1990 – 1994	Freelance IT consultant for SIK (Schweizerisches Institut für Kunstwissenschaft), Zürich.
Nov '92–Jan '93	Internship as programmer at Rieter AG, Winterthur.
1994 – 1998	Teaching Assistant at the department of computer science, ETH Zürich.
1998 – 2002	Ph.D. student at the department of computer science, ETH Zürich. Supervisor: Jürg Nievergelt.