TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER'S THESIS

The Design and Implementation of the Rookie 2.0 Chess Playing Program

by
M.N.J. van Kervinck

Supervisor: dr. ir. T. Verhoeff

Eindhoven, August 2002

Contents

Chapter 1

Introduction

1.1 Computer chess

Half a century of computer chess has produced remarkable results. Early with the advent of the first electronic digital computers attempts started to implement programs that would play chess [Shannon50]. What first became a *Drosophila* for Artificial Intelligence, finally evolved into a technology arms race. Competition drove a fast-paced engineering effort of refining brute-force techniques. Eventually, the human world champion was beaten in a match by a hurd of chess-specific VLSI circuits, IBM's Deep Blue, leaving the question about A.I. mostly unanswered.

With the increasing availability of ever faster and cheaper computers, writing one's own chess program has become a relatively popular, albeit somewhat peculiar, challenge for both the amateur and the professional programmer. Although the world's attention has diminished since Deep Blue, the competition among software implementations still continues. Chess programs that play at the grandmaster level are commercially available for the personal computer and so are master-level programs for portable digital assistants.

The improvement in playing strength can be attributed to two developments. First, the continuous growth in computing power that allows programs to search through greater portions of the game tree. Second, the development and refinement of search and evaluation techniques that allow programs to

utilize their time and memory resources in more efficient ways.

Over the past twenty years most successful programs have been based on the same principles as their predecessors. Improvement came from extending and refining existing techniques. Radically different and revolutionary approaches have never been able to catch up with this process of steady progress, although at times they provide ideas and understanding on how to improve upon the state of the art. This long engineering history also has made the task more difficult for new programmers who aspire to write a program that can compete with the best programs today. One has to understand the history of the field and one must be prepared to repeat this evolution in an accelerated fashion. Within the established framework, however, there is still plenty of room for experiment and new ideas. This, together with the competitive nature of chess, is likely the reason why new programs are still being made.

1.2 Rookie 2.0

This report describes the design of one such chess program: 'Rookie'. The youngest version, Rookie 2.0, was written from scratch in about a year of work. It is based on five years of experience obtained by building two earlier versions of a program with the same name. Each version represents a rewrite of the core data structures and functions. These earlier versions were mostly implemented in assembly language[1]. The oldest version played its first computer tournament games in 1993. It evolved from a mate-in-2 problem solver into a playing program. The current version is written entirely in the programming language 'C' [Kernighan88]. It consists of approximately 25,000 lines of code plus a large database of book knowledge. The emphasis of the design is on the engine's game playing strength and its position analyses capabilities. Therefore, the program has no graphical user interface: a command line operator interface is sufficient[2].

[1] For the Motorola 680x0 CPU series.

[2] However, it is compatible with the xboard program, which can give Rookie a graphical chess board (http://www.tim-mann.org/xboard.html).

1.3 This report

This report serves to document the design considerations and the implementation decisions of our program. We describe and evaluate the most important techniques and innovations. Where needed for better understanding we will briefly discuss the required background from the computer chess field.

Chapter 2 is devoted to tree searching. Topics like pruning, move ordering, null moves, quiescence search, transposition tables and iterative deepening all fall into this category. We provide some quantitative evidence of the effectiveness of these techniques.

Chapter 3 describes the evaluator in Rookie. While being conventional in the type of chess knowledge that is contained in the program, the actual implementation of this knowledge is rather unconventional. The purpose is processing speed. The program uses carefully designed data structures that enable very fast processing of common patterns of chess knowledge.

Chapter 4 deals with book knowledge. The program has access to a database containing the moves of thousands of games played by masters and grandmasters. These games are used to provide the program with a broad opening repertoire. We describe our experiments in opening learning.

Chapter 5 concludes this report by summarizing the main results and conclusions.

1.4 Theory, terminology and notation

For those who are unfamiliar with the subject we explain the general foundation on which computer chess is built.

1.4.1 Heuristics

The game of chess has an estimated state-space size of 10^{50} positions [Allis94]. Also, the rules of chess [FIDE97] generate a complexity that is too great to simplify into some closed form that always yields optimal play. Thus there is no technology that can analyze an arbitrary chess position deep enough

to establish the game-optimal result. Therefore, the only practical way to build a strong program is to use a heuristic algorithm. The approach is to construct a game tree in which the nodes represent chess positions. The root node corresponds to the actual position on the chess board[3] and the branches represent single chess moves. We then assign heuristic scores to the leaf node positions. These scores evaluate the positions and gives them a linear ordering: the 'better' the position, the higher the score. From such a tree with scores we can work out the path of moves that brings us to the leaf node with the highest achievable score, assuming that the opponent attempts to do the opposite.

The heuristic nature of chess programs is in the way they select the subtree and in the evaluation function. The depth of a tree is expressed in a unit called *ply*. This is the same a move by one player. The term 'move' is avoided because to chess players this often means 'full move': a move by both sides (two plies).

1.4.2 Evaluation

The evaluation is based on detecting known patterns on the board, associating them with a value and summing these. It could be expressed as a linear combination of terms

$$\text{evaluate}(pos) := \sum_i W_i \cdot P_i(pos)$$

where P_i detects the presence or multitude of a pattern and W_i represents its weight or value. Examples of patterns are 'the number of white pawns' or 'the distance of the black queen to white's king file[4]'.

In computer chess, the standardized scaling is to give +1.0 to the value of a pawn. The other pieces are worth more than a pawn. Most remaining chess patterns have values that are much smaller. In Rookie 2.0 a score can be expressed as accurately as 1/250th of a pawn (ϵ). Because of that it can be represented internally with signed 16-bit integers without fear of overflow in extreme positions where the material score could go up to the equivalent

[3] A position includes the 'invisible' status parameters, such as side to move, castling rights and en-passant information.

[4] A *file* is one of the eight vertical lines on a chess board. Horizontal lines are called *ranks* and the lines along which bishops move are *diagonals*.

of 100 pawns. For game-theoretical, non-heuristic, scores (distance to mate, distance to loss, or draw) we reserve three exclusive intervals in this scale. Internally, 'mate-in-n' is $+32768 - n$, loss-in-n is $-32768 + n$ and a draw is 0. To keep the 0 reserved for real draws, the heuristic evaluation s is represented as $-250s$ (for $s < 0$) or $1 + 250s$ (for $s \geq 0$). In external presentations we use symbolic representations such as `+1.824`, `+mate3` and `=draw=`

1.4.3 Search and speed

For chess it has been experimentally established that a deeper overall search invariably translates into significantly stronger play [Hsu90]. The result is that one must be cautious when adding evaluation patterns if that slows down the program: the end effect may be a more knowledgeable but weaker program due to the lost depth. At the same time it means that quality of implementation matters. Efficient data structures that help minimize the number of instructions per node are mandatory. To give an indication, our current program visits well over 100,000 nodes per second on a 350MHz Pentium-II system. We have reasons to believe (based on Rookie 1.0) that an optimal assembler implementation of the same program would be about three times faster. This means that the program should be designed to use much less than 1000 instructions per node. This includes move generation, move validation, move making, move unmaking and position evaluation.

Chapter 2

Tree Search

The tree search contains the *dynamic* knowledge of a chess program. Its purpose is to find the most favorable path in the game graph by traversing and evaluating an as large as possible part of the state-space (see section 1.4). The tree search techniques in Rookie 2.0 are similar to those found in most other chess programs. The differences are in the way some of the details are refined and implemented. Our search strategy can be classified as consisting of:

- an iterative deepening PVS[1] with aspiration window and confidence search,
- SEE[2]-based quiescence search and move sorting,
- 8-probe transposition table using a sub-tree size replacement strategy,
- extensions with fractional depths,
- variable-R null-move pruning with verification search, and
- additional move sorting with hint moves (killer moves, counter moves) and the history heuristic.

[1] PVS is addressed in section 2.1.6
[2] SEE is explained as part of section 2.2

Of these, the confidence search, the fast static exchange evaluator (SEE) implementation, the transposition table replacement scheme and the validation of hint moves are the areas where we have made the most interesting new developments. What this means for Rookie is the topic of the remainder of this chapter.

2.1 Basic search

This section briefly describes the basic framework that is common to all chess programs.

2.1.1 Brute-force and full-width

The core search strategy is *brute-force*: From the current board position, the root, we search a tree of moves as deeply as possible. The tree will be *full-width*, meaning that we expand *all* of the moves in the inner nodes. The alternative would be a selective search that attempts to reduce the tree size by exploring just candidate good moves. This has never been at the heart of a strong program since CHESS 4.5 has demonstrated the superiority of full-width in the domain of chess [Slate77][3]. Usually we cannot search deeply enough to reach the end the game and then we have to restrict ourselves to searching some subtree of the whole game tree. The leaves are then called the *horizon*. Figure 2.1 sketches such a tree where all variations have been expanded until a 4-ply deep horizon[4].

We apply the position evaluator (chapter 3) in all of the end nodes, and propagate and combine their scores towards the root node using minimax or negamax. 'Minimax' means the alternating maximizing and minimizing of scores in inner nodes, depending on whether it is white's or black's turn in the node. 'Negamax' is the commonly used equivalent that only maximizes and reverses the sign on the way down. Negamax therefore interprets the

[3]But beware, in many programs, variants of selective search come back in the form of deviations from the pure full-width basis strategy. Null-move (see section 2.5) is a modern example of that.

[4]For simplicity, the tree figures in this report always have a much lower branch count per node than the 30 to 40 moves that chess positions normally have.

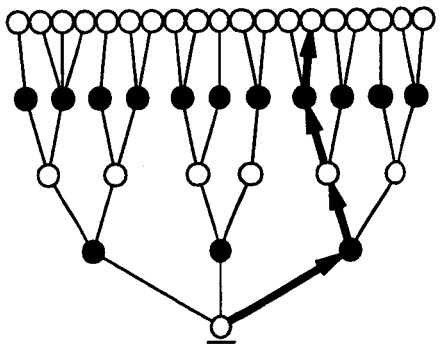

Figure 2.1: A 4-ply full-width tree with principal variation

score from perspective of the side to move. Its pseudo-code is given in figure 2.2.

```
1   fun negamax(pos, depth)
2   {
3       var score, best, move_list, new_pos
4
5       if (depth=0 ∨ game_over(pos)) return evaluate(pos)
6       move_list := generate_moves(pos)
7       best := −∞
8       for (move in move_list) {
9           new_pos := make_move(pos, move)
10          score := −negamax(new_pos, depth−1)
11          if (score > best) best := score
12      }
13      return best
14  }
```

Figure 2.2: The negamax algorithm

The *principle variation* (pv) is the path from the root to the end node whose score has propagated back to the root. It consists of the best moves against the best possible counter play[5]. The first move of the pv is the move that the program should play. Figure 2.1 shows a hypothetical pv as a chain of fat arrows.

[5]The meaning of 'best' here is to be taken within the scope of the subtree.

2.1.2 Alpha-beta pruning

Alpha-beta is a branch-and-bound algorithm that can perform a minimax or negamax search without need to visit all of the nodes in the tree [Knuth75]. It is based on maintaining lower and upper bounds (*alpha* and *beta*) throughout the search, describing the interval within which the score must fall to have an effect on the root score. When in an inner node, after searching one or more moves, the partial score falls outside of this interval, it can safely discard searching the remaining moves in that node. This interval is called the alpha-beta *window*. Figure 2.3 lists the pseudo-code.

```
1    fun alphabeta(pos, depth, alpha, beta)
2    {
3        var score, best, move_list, new_pos
4
5        if (depth=0 ∨ game_over(pos)) return evaluate(pos)
6        move_list := generate_moves(pos)
7        sort(move_list)        # (see section 2.6)
8        best := −∞
9        for (move in move_list) {
10           new_pos := make_move(pos, move)
11           score := −alphabeta(new_pos, depth−1,
12                   −beta, −max(alpha,best))
13           if (score > best) best := score
14           if (best ≥ beta) break
15       }
16       return best
17   }
```

Figure 2.3: The alpha-beta algorithm [Knuth75]

For depth ≥ 0, alpha $<$ beta, ab = alphabeta(pos, depth, alpha, beta), and n = negamax(pos, depth), we have

$$\begin{aligned} \text{ab} \leq \text{alpha} &\Rightarrow \text{ab} \geq \text{n}, \\ \text{ab} \geq \text{beta} &\Rightarrow \text{ab} \leq \text{n}, \\ \text{alpha} < \text{ab} < \text{beta} &\Rightarrow \text{ab} = \text{n}. \end{aligned} \tag{2.1}$$

When the window is fully open, alphabeta computes the true negamax score:

$$\text{negamax}(\text{pos}, \text{depth}) = \text{alphabeta}(\text{pos}, \text{depth}, -\infty, +\infty)$$

In a way, alphabeta can be considered to be a kind of branch-and-bound algorithm, albeit with a 2-dimensional bound. As such, the number of skipped

nodes depends on how well-ordered the tree is. The more of the best moves are searched first in their node, the better the gain. Figure 2.4 illustrates what the same tree may look like under alpha-beta pruning with different internal move orderings. In this small tree we see that the search in the well-sorted tree visits only 8 leaves while the unsorted tree needs 12. The gain becomes significantly bigger with wider and deeper trees. With near-perfect move ordering, alpha-beta can negamax a tree approximately twice as deep with the same effort [Knuth75]. Section 2.6 explains how we achieve a near-perfect move ordering in Rookie.

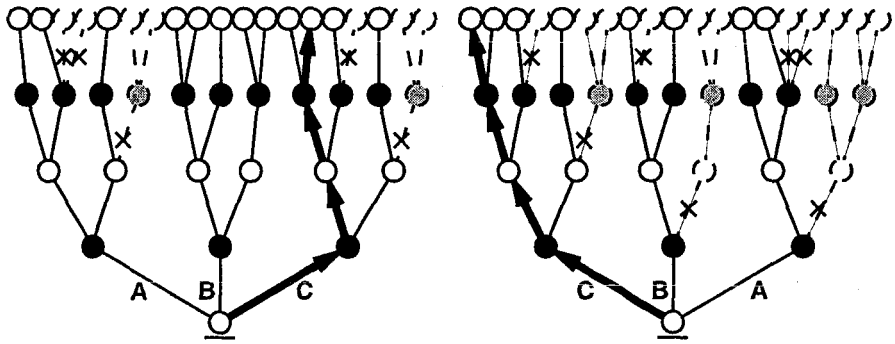

Figure 2.4: Alpha-beta pruning, unsorted (left) and well-sorted (right)

2.1.3 Iterative deepening

Iterative deepening is one of the oldest techniques to help sorting the tree. It works by the observation that the results of a search to depth $d-1$ can act as a good starting point for search to depth d. Due to the exponential tree growth the shallow search represents just a small part of the total effort and the savings in the deeper tree are expected to exceed this overhead: most often what is a good move in the small tree is also good in the bigger one. This logic can be applied repeatedly, starting from depth 1 and working upwards to d in steps of one ply. Figure 2.5 illustrates the idea for 4 iterations.

In its simplest form only the root moves are re-ordered between iterations. It is better also to remember and try the previous principle variation first. The greatest gain can be obtained if we have some way to store and re-use all of the previous search tree. Naive attempts to accomplish this could

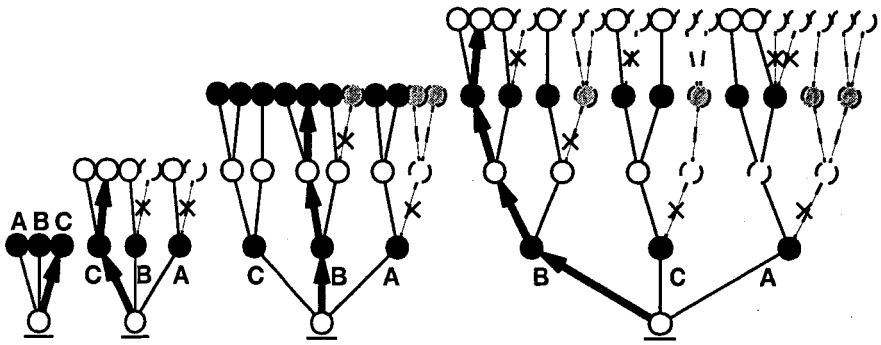

Figure 2.5: Iterative deepening

lead to complex tree-shaped data structures that will pose problems to the programmer when memory runs out. Fortunately there is a simple and effective alternative: with transposition tables (discussed in section 2.3) we have a fast mechanism that keeps important parts of the tree in memory. It also exhibits graceful degeneration when the trees grow bigger than the main memory can handle. We give quantitative evidence for the efficiency of iterative deepening in section 2.6.4.

In Rookie 2.0 we fully rely on the transposition tables to keep both the principle variation and the tree. In the root node, between iterations, we sort the non-optimal moves in decreasing order of the subtree sizes that are behind these moves. The heuristic that is behind this is that the dumber the move, the easier it is for the opponent to demonstrate that it is bad, often generating fewer nodes. Consequently, the best candidate moves to replace your current best move are likely the ones that generate large subtrees. Within an iteration, we want to try these candidates as early as possible after the best move.

An added benefit of iterative deepening is that it helps in time management. When it is the program's move, it does not have to determine a target search depth beforehand. It can be hard to predict how much time is needed for a search to a certain depth: unexpected hard positions can appear within the search, slowing it down. Mis-calculating the final depth by one ply easily causes an exceeded time usage by several factors. With iterative deepening we can prevent such anomalies because the program always has a reasonable move to play when it discovers that it is running out of time. This can

be done between iterations, but also in the middle of an iteration if time suddenly runs out.

There is more. Under certain conditions we detect 'easy' moves. That are moves that already looked good without search (by using a static move evaluator) and that stay best along all iterations. For easy moves we decide to make the move when a fraction, 25%, of the time limit is reached. In the same way we can detect when the program is getting in trouble: If the best move is repeatedly refuted in a later iteration it is wise to spend extra depth and extra time before it is too late in the game.

2.1.4 Aspiration window

From formula 2.1 we can derive that alphabeta($pos, -\infty, +\infty$) will always give an exact result. So it seems logical to use that for the first move in an iteration. Yet we can do better. We can assume that in most cases the new score will not deviate much from that computed in the previous iteration. We can anticipate this score by narrowing the window of the first root-move around the earlier score. In Rookie we have an *aspiration window* of $(best - 0.3, best + 0.3)$, where 1.0 is the nominal value of a pawn.

With the narrow window alphabeta will search fewer nodes. This wins time at the expense of risking to get a score that is outside the window. If $score \geq beta$, we say it *failed high* while at $score \leq alpha$ we say the search *failed low.* In the first case the best move turns out to be even better than we expected. This is good news and all we do is re-search the move with a wider window to get its true value. In Rookie we always perform such re-searches using a full window of $(-\infty, +\infty)$[6].

In the case of a fail-low we should be careful: our best move may not be that good at all and the opponent may have a much better position than expected. First we perform a re-search to establish the magnitude of the damage. Then we continue with the remaining moves in hope to find one

[6]Earlier versions tried gradual window expansion, usually just into the direction of the failure. This gives all kinds of trouble such as repeated fails, or even worse: opposite failures due to search anomalies. So finally we simply resorted to a full-window research and accept its outcome as the truth. The transposition tables (section 2.3) will recall the initial search anyway and ensure that the tree-traversal during the re-search is close to optimal.

that brings the score back to the old level. At the same time we flag the fail-low situation and will not make a move until the current iteration is completed.

2.1.5 Confidence search

A trick is used in the root search that we invented for Rookie 2.0: the *confidence search*. Normally one would make the decision to make a move after completing an iteration. In early versions of Rookie 2.0 we regularly observed that the program would make a move, and then, while pondering in the opponent's time, suddenly discover that it was in fact a very bad move! The observation that we made while watching iterative deepening in tournament games was that, in a most cases, such discoveries appeared to be instant: within a couple of seconds. If only we had searched just one more ply before making the move, we would have recognized the problem in time to search an alternative.

To remedy this we introduced the confidence search. Instead of making the best move after completing iteration d, we continue searching for a small amount (25%) of time on the next iteration, $d+1$. Two things could happen: normally the additional time runs out before the $d+1$ search on the first move returns. In that case we assume the move is safe and play it. Otherwise the first search is completed within the extra time. If it is a fail-low, we panic and allocate extra time to find a remedy. Otherwise we just play the move.

Normally the first move in an iteration takes about 50% of the time: the refutation of the remaining moves is usually very fast. Fail-lows on the first move are an exception to that pattern: they are often fast to return. The reason for this can be understood from the shape of a well-ordered alpha-beta tree: it only takes one opponent move to cause a fail-low on the current-best root move. The size of the corresponding subtree is comparable to a $d-1$ subtree, so it represents just a fraction of the just performed d search. With the confidence search one could say that we add an extra 'virtual' ply of search using only a fraction of the time.

2.1.6 Principle variation search

Principle variation search, PVS, is a refinement on alphabeta that works best in well-sorted trees. It is an opportunistic algorithm in the way that it presumes that the best move in a node is always known. Within a node, it only tries the first move with the original alphabeta window. All other nodes are further searched with a null-window of $(best, best + \epsilon)$, where ϵ is the smallest evaluation unit. It expects these moves to return below or at *best*. Only when this assumption proves wrong (a fail-high on the null-window) does it restore the original window and perform a re-search to establish the new best value. In practice, within well-sorted trees, just a hand full of positions are required to be searched with the normal 'open' window[7], and the bulk of the search is done with a null-window. This property makes PVS a big node saver. PVS is well-described in text books, for example [Kaindl90].

2.2 Quiescence search

If we reach the horizon in the middle of a piece exchange sequence, we must be careful evaluating the resulting leaf position. After all, the material as present on the board will immediately change with the next (re)capture and that will alter the score significantly. Consider the position in figure 2.6 where black has just captured a white pawn on e4. A static evaluation would count material and conclude that white is down by a pawn. However, white can safely recapture the black pawn with its bishop, balancing the score. We say that such positions are *unstable* or not *quiet*. If we do not recognize such temporary imbalance, the search will suffer from severe mis-evaluations.

We will see in chapter 3 that our evaluation function does not take immediate material changes into account. In Rookie we delegate the responsibility to handle this to the *quiescence* search. Its purpose is to extend the search, if needed, to 'quiet' positions where the evaluator can be applied safely. Conceptually this means that alphabeta does not call evaluate directly in the leaves, but a quiescence search function instead. This function first calls the evaluator and then checks if any of the legal moves are likely to invalidate its score. If none, it returns the evaluation to the caller. Otherwise, it

[7]These nodes are the candidate principle variations, which explains its name.

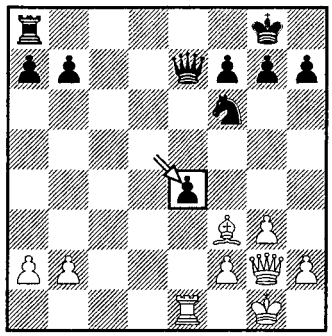

Figure 2.6: Unstable position after dxe4.

searches these moves deeper, using itself recursively. If this proves to be an improvement over the evaluation, it replaces it. Its pseudo-code is given in figure 2.7.

```
1    fun qsearch(pos, alpha, beta)
2    {
3        .var score, best, move_list, new_pos
4
5        if (game_over(pos)) return evaluate(pos)
6        best := evaluate(pos)
7        if (best ≥ beta) return best
8        move_list := generate_moves(pos)
9        filter_and_sort(move_list)    # keep moves that may improve best
10       for (move in move_list) {
11           new_pos := make_move(pos, move)
12           score := −qsearch(new_pos, −beta, −alpha)
13           if (score > best) best := score
14           if (best ≥ beta) break
15       }
16       return best
17   }
```

Figure 2.7: The quiescence search

Figure 2.8 shows how this may generate a couple of extra moves beyond the horizon.

Of course the quiescence search also has a responsibility to terminate the search quickly. By applying alpha-beta pruning in the resulting extended tree and by carefully selecting the candidate captures, the quiescence search

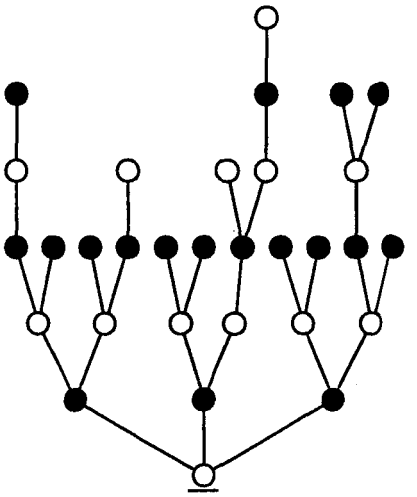

Figure 2.8: A 3-ply full-width tree with quiescence searches

can be kept sufficiently small (that is, far fewer than 1 extra node per leaf on average). In Rookie we restrict ourselves to captures[8].

Not all capture moves are candidates for extension in the quiescence search. If we allow that, then the search beyond the horizon will explode into many futile capture sequences. Even alpha-beta pruning and good sorting does not remedy this. What is needed is a method that can be applied in the capture generator to detect which capture moves potentially increase the score. This is called a *static exchange evaluation*, or SEE. The SEE-value of the captures is then used to extend the search.

In Rookie 2.0 we implemented a fast SEE that performs remarkably well. Captures are generated by iterating over the enemy pieces and checking if they are under attack. For each attack a capture move is generated. At the same time it collects the number and type of attacking pieces to that square. For sliding-piece[9] attacks the generator checks if the attacking piece is defended by another slider that can move in the same direction: one attacker behind another. In the example of figure 2.6 we would thus find the white queen. This way we generate a list of captures of the enemy piece together with a piece list that includes attackers hidden behind another.

[8]Or all check-evasion moves for horizon positions where the player is in check.

[9]Bishop, rook or queen.

Along the same lines we generate a list of defending pieces. When the list of attackers and defenders is complete we revisit the just generated captures and compute a SEE-value for each of them.

This SEE value depends on the value of the first-captured piece, the value of the capturing piece, and the remaining piece lists. A straight-forward implementation of SEE would quickly perform a simplified search on the two opposing sets to determine when the exchange would stop. This works on the observation that for a recapture a player would always recapture with its least-valued piece first. Thus this 'search' is just a linear run through the lists. In some positions this is not sufficient: if in the example position the white bishop and queen locations were swapped, then white could not capture with the bishop first.

Such exceptions make SEE troublesome to implement. In our program we decided to relax the requirements for SEE a bit: it does not have to be exact when it comes to batteries[10], but it should never exclude a capture due to misunderstanding order-constraints. With this relaxation we can afford to ignore piece order in the two attack and defense lists. For the attacking set this means that the SEE will be a little too optimistic: some extra captures will be generated in cases where weak pieces are in batteries behind strong ones. For the defense set we must perform a special technique if a weaker sliding piece (bishop or rook) is behind a stronger piece (queen), we *promote* it to the stronger piece in the defense set. This effectively weakens the SEE's notion of the defense.

Now, the SEE-function is reduced to repeatedly playing out the weakest piece from the alternating sets. This exchange sequence stops either when a side runs out of pieces or when further exchange would lose material continuing. Figure 2.9 gives pseudo-code.

The point of strengthening the attack set and weakening the defense set is that it removes the piece order in batteries and therefore reducing the SEE's search space. Under the most pathological circumstances, one piece set consists of at most 3 pawns, 12 bishops or knights, 10 rooks, 9 queens and/or 1 king. Thus there are at most $4 \cdot 12 \cdot 11 \cdot 10 \cdot 2$ possible sets. These possibilities can be encoded in 13 bits[11].

[10] Batteries are multiple sliding pieces behind one another.

[11] The 2-log of the number is slightly higher than 13. But not all combinations are possible in a valid position, making 13 bits sufficient to represent the piece set.

```
1    fun SEE(victim, attackers, defenders)
2    {
3        var score, next
4
5        if (is_empty(attackers)) return 0
6        next := weakest_piece(attackers)
7        score := value(victim)
8        score := score − SEE(next, defenders, attackers − {next})
9        if (score < 0) score := 0
10       return score
11   }
```

Figure 2.9: The SEE algorithm

In the implementation we thus collect the defense and attack sets during move generation, where we represent them as two short integers. We complement the defending set with two bits that indicate what piece currently occupies the target square. That piece cannot capture but is subject to capture itself. By combining the two set representations with an **xor**-operator we obtain a well-distributed hash key in a lookup table that caches earlier results of the SEE-function. This works well enough that the whole SEE-computation in Rookie can be considered to consist of a mere table lookup.

For completeness we note that when the target square is on the 1^{st} or 8^{th} rank we have a dedicated SEE variant that understands pawn promotion. Its results are not cached, but the moment the sets are free of pawns it falls back to the regular SEE.

2.3 Transposition table

2.3.1 Concept

The *transposition table* is the place to store partial search results in the hope to generate future savings. Its name is derived from its capability to detect identical positions that are reached through different move sequences. There is no need to search such repeats multiple times if the earlier search result is still available. Especially in the endgame, where the tree is thinner, transpositions reduce the node count and increase the search depth. In other positions, the role of the table is more to help the iterative deepening

algorithm remember the tree's internal best moves of previous iterations (see section 2.1.3).

The transposition table is the largest structure in chess programs. It should scale with the speed of the search. To save time and space, the table stores hash values of positions instead of full position representations[12] For this purpose, the move making functions update a 64-bit hash-key with each move. With this we can index the table to find the slot that should belong to a position. The slots themselves hold 32 bits of the hash. In our current scheme, we reserve 16 bytes for each slot.

- `hash`: 32-bits of the full hash key,
- `vector, owner`: two 8-bit redirection fields
- `score`: a 16-bit score,
- `move`: a 16-bit best move representation,
- `nodes`: a 16-bit node counter,
- `depth`: the 8-bit tree depth,
- `flags`: 8 flag bits
- two bytes are unused

The score, move and depth fields hold the search result. The flags can signal if the score is to be interpreted as an exact score, an upper bound or a lower bound. Entries are written just before returning from a recursive alphabeta call and probed at function entry. If the position is found, its move is always searched first. If the depth field is at least as large as the depth parameter of the call, the stored score can be used to tighten the alpha-beta window.

2.3.2 Hashing errors

With an inherently imperfect hashing function to represent chess positions in the table, there is room for error: when two different positions map to

[12]The consequence of this is that we introduce the potential of *false positives*, examined in more detail in section 2.3.2.

the same hash code, they may be confused for one another. This may result in a false evaluation of the position. The probability for such an error can be modeled: for each supposedly unsuccessful probe on a slot in the table, we run the risk of mis-interpreting it as a hit. Say that, on average, we search 100 seconds at 100,000 nodes per second ($n = 10,000,000$ nodes), where 75% of all nodes are new and not present in the table. Furthermore, assume the table is already filled with previous results and that $b = 32$ bits of hash-key are kept in the slots. Then, the probability P of making at least one identification error is[13]:

$$P = 1 - (1 - 2^{-b})^n,$$

or 0.002 for the values given above. The number ways in which such a occasional error could influence the outcome of the search are even smaller.

There is also an experimental approach to approximate the frequency of these errors: we can adjust the table lookup code and add a side-effect that collects statistics for each failed lookup. For these events we count the minimum number of bits that were needed to distinguish the position from the one that was present in the slot. For example, if $b = 4$, we expect one out of 16 misses to be falsely attributed as a hit. We ran the experiment using 24 test positions[14]. For each position we had Rookie search for 5 minutes, while recording the number of discriminating bits ($1 \ldots 32$) for all table misses. With this we can count the number of errors for any $b < 32$, as if b were less than 32, instead of equal to it. The total results are plotted in figure 2.10.

The experiment represents two hours of search and 900 million nodes, of which 650 million were misses (and thus potential errors). As expected, we see the number of errors halve with each extra bit. When extrapolating the graph we can safely assume that not a single 32-bit error was made in this run.

[13]This model differs from that in [Breuker98]. The explanation that we can give for that is that our model assumes a saturated table: under such conditions, each new update causes an entry to be forgotten. The empirical evidence shows that this model is valid. As a result, the error rate does not depend on the size of the table, but only on the number of hash-bits stored with each entry. Also, the *birthday paradox* is not applicable any longer.

[14]The Bratko-Kopec test set.

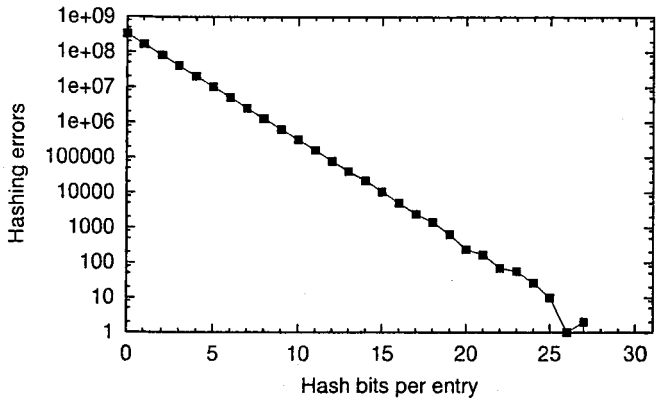

Figure 2.10: Hashing errors

2.3.3 Multi-probe and replacement

The redirection fields are special for Rookie. They implement a *multi-probe* scheme. When the table is too small in comparison to the search tree, too many useful entries will get overwritten by entries that represent little search effort. To remedy this it is a good idea to associate more than one (n) slot with each hash index [Breuker94]. Then, when writing to the table, we can select from these n candidates the slot that represents the least effort, preserving more expensive results. In Rookie we settled for an $n = 8$ scheme, where the probed slots are given as

$$\text{table}[index + 2^i - 1], \text{for } 0 \leq i < n.$$

We call table[index] the *primary* entry and table[index+$2^i - 1$] the *actual* entry. To avoid probing all n slots at function entry, each primary entry has the 8-bit field `vector`. This field contains the set of actual entries that are in use by the primary entry. For $n = 8$ it can be conveniently stored as one byte. Since at a given moment, each slot is in use by only one primary slot, the average vector field has just one reference. Therefore a table lookup consists of one physical access as well[15]. The `owner` field is used for updating `vector` and contains a back-reference from the actual slot to its primary slot.

The replacement criterion that we use is the size of the subtree that the entry

[15]When storing an entry we still need to investigate all n slots to find the best replacement candidate.

represents. This replacement strategy is called BIG in [Breuker94]. To save some space, the node count is squeezed into 16-bits. The representation is a tuple (s, v). We have 6 bits for s and 10 bits for v. This tuple represents the value $v \cdot 2^s$ as an approximation the real subtree size. This scheme sacrifices some accuracy for a lot of space. It also has the nice property that the squeezing operation is monotonic: we can use 16-bit unsigned integer comparisons on the squeezed representation as if we are comparing the real node counts.

2.3.4 Performance

To show how Rookie's multi-probe transposition table performs, we repeated the experiments in [Breuker96] and extended them to $n > 2$. From figure 2.11 we confirm the observation that using a two-level scheme is better than a one-level scheme. We also observe that with increasing search depth, and thus with higher table load, extra probes keep increasing the table's effect on the search.

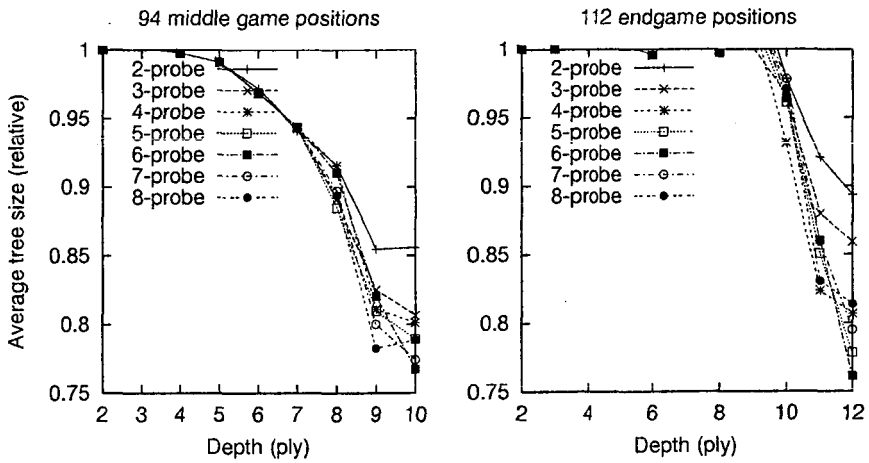

Figure 2.11: Multi-probe

Finally, we measured the search performance with greater memory sizes (figure 2.12). We see that, with the $n = 8$ replacement scheme, Rookie 2.0 needs at least 8 million entries (128MB) on the current hardware to make optimal use of the table. More memory is overkill for its speed.

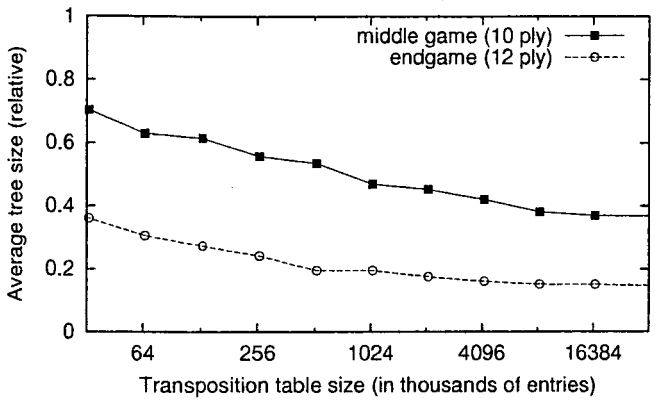

Figure 2.12: Transposition table savings

2.4 Search extensions

2.4.1 Concept

Search extensions make selected paths from the root to the horizon longer than nominal, breaking with the idea of a uniform-depth horizon. The purpose is to follow certain 'interesting' tactical lines deeper in the expectation to find their real outcome sooner: in some earlier iteration with less nominal depth. The observation is that many moves in tactical sequences are more or less forced. Extensions are implemented in a search by replacing the constant depth reduction by something smaller than 1. Figure 2.13 illustrates a tree with extensions of 1 ply applied along three of its branches, marked with exclamation marks.

Of course one has to be careful when adding extensions to a search tree: one should not introduce too many of them, because that will cause an overall loss of depth. Also one must be careful not to extend too aggressively. For example, a 1-ply extension effectively does not reduce the 'depth' parameter of alphabeta. So there lies some danger to not reach a horizon at all. Termination must then be guaranteed by taking the game rules into account, such as if a move is irreversible or leads to repetition.

In Rookie we have *pure* extensions. A pure extension depends solely on the node's position, the move to search and possibly the remaining depth. For

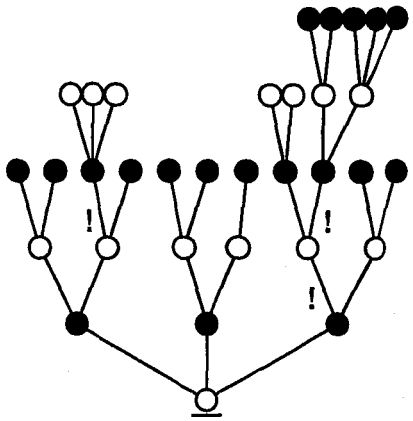

Figure 2.13: Search extensions

example, it does not depend on the variation leading to the position nor on its current alpha-beta window. This is to prevent anomalies with transposition tables and PVS re-searches that otherwise appear. Therefore extensions can be understood in terms of a function, new_depth(depth,pos,move), that replaces the expression 'depth − 1' in recursive search calls. The extension is then defined as

$$\text{extension} := \text{new\_depth}(d, p, m) - d + 1,$$

which should not be negative. Otherwise we have forward pruning and lose the ability to guarantee a certain full-width beforehand. This would be bad for the ability to solve chess puzzles[16].

2.4.2 Implementation

The notion of depth in our implementation is not restricted to whole plies. It is a fixed-point number with $1/8^{\text{th}}$ ply precision. This way we obtain the possibility to have extensions smaller than a 1 ply. Sometimes an extra full

[16]Actually this constraint can be relaxed a bit. Fractional pruning is allowed if it does not 'cross' a whole-ply boundary. This could be useful when experimenting with gradually fading out the effect of an extension deeper along a variation. Such a 'skew' also helps maximizing the number of transposition table hits for which the depths match. We experimented with this in Rookie, but normally leave it commented-out.

ply is just too much. With these fractional plies it takes at least two half-ply extensions along a variation to result in a deeper search[17]. The presence of the depth-parameter in de new_depth function allows us to have larger extensions near the horizon and smaller closer to the root. When we do this, we do need to make sure that new_depth is monotonic:

$$\text{new\_depth}(d, p, m) \leq \text{new\_depth}(d + n, p, m),$$

This guarantees that an increased d yields a tree that at least comprises the old one, preventing unpleasant surprises.

For Rookie 2.0 we currently have three extensions: check evasion (7/8 ply), single move (15/8 ply), and piece exchange (5/8 ply). Check evasion is any legal move out of check. A single-move extension is for nodes where the player has just one legal move. Piece exchanges also often consist of forced moves. Ideally we would like to extend just a re-capturing move in an exchange, because that is the forced move. In general, one cannot determine if a capture is a re-capture without making the extension impure. We therefore opt for an alternative and apply this extension to any capture for which the SEE-score predicts an even outcome: the first move of an exchange[18]. When multiple extensions are applicable we pick the largest. For each ply above depth=6 we reduce the extension by 1/8 ply.

2.4.3 Example

Figure 2.14 demonstrates the power of extensions. During the 5-ply iteration, within 14,000 nodes, or 130 milliseconds on its current hardware, Rookie reports an eight-move (15-ply) mating combination starting with a queen sacrifice on h7. After 2.6 seconds, in the 8-ply iteration, the shortest mate is found which is one move faster. Without extensions the program would need more than one day of computation before it reaches the 13-ply iteration and finds the solution. Until that point it is unlikely to produce the winning first move.

[17] In the case where we use iterative deepening over whole plies. We can vary a little by iterating over $n + 1/2$ plies, effectively shifting the horizon a bit. Then the first half-ply extension already deepens the search, but it takes two more to deepen it again.

[18] Unless we restrict ourself to captures for which a SEE-score predicts to bring back the overall material balance to, say, 0. We have not experimented with this variant yet.

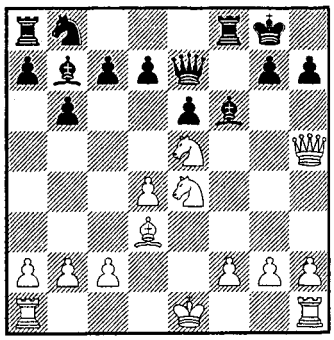

Figure 2.14: Extensions finding a deep mate (Lasker-Thomas, 1912)

2.5 Null-move pruning

The *null-move* technique is an aggressive domain-dependent forward-pruning technique [Goetsch90][Donninger93]. In a full-width search a lot of the variations are necessarily nonsense and include variations with multiple blunders. Null-moves discard many of such futile branches by testing if a position still makes sense in a certain way.

2.5.1 Concept

The null-move attempts to establish a lower bound for a node's score by searching the position from the opponent's point of view first. Making such a hypothetical 'passing move' or 'null move' is not legal in chess. Even if a player were given the option to pass it is often not good: under normal circumstances there will always be a better move than doing nothing. The null-move heuristic assumes that such a lower bound could just as well be estimated using a search with reduced search depth. When the null-move score exceeds *beta* it will assume that one of the other moves would fail-high as well. It then simply returns this score without searching any of the other moves at all. Figure 2.15 illustrates such a cut-off after a null-move search with a depth reduction of 1 ply (R=1).

In principle the null-move heuristic can be applied recursively within the null-move searches themselves. Yet we do have a couple of constraints. It is not performed twice in a row or in positions where the king is in check: Pass-

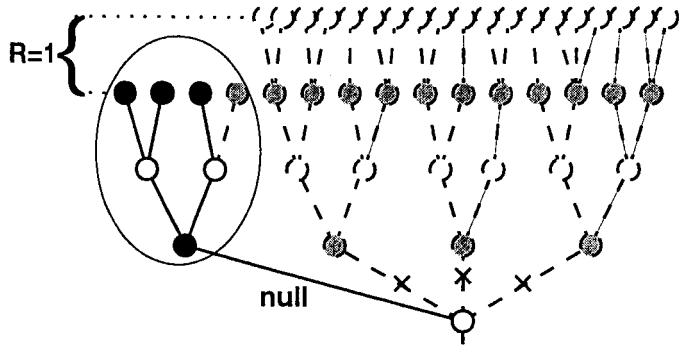

Figure 2.15: Conventional null-move pruning

ing with one's king exposed would yield a position where the opponent can capture the king, which makes no sense within the rules of chess [FIDE97]. Further we do not apply it in the candidate principle variation nodes of PVS (section 2.1.6), but only in the null-window nodes.

Finally, the depth reduction R is made to depend on depth d, the distance to the horizon. After some tuning in Rookie we decided to have R range over $[1.5 \ldots 2.5]$, using fractional depths, and we settled for

$$R := \mathbf{min}(1.5 + d/8, 2.5).$$

Figure 2.16 shows the average tree sizes for searches with and without recursive $R = 1$ null-move pruning. The node savings are considerable and the resultant extra depth introduced by null-move pruning is worth about two full plies of additional search. The 206 test positions we used here are take from from top-level players and described in [Breuker96].

2.5.2 Risks

The extra plies of depth come with the price of extra risks. First one has to be careful using null-moves in combination with transposition tables and, especially, repetition detection. A somewhat careless implementation will easily confuse legal variations with variations containing a null-move. Such subtle interactions may be hard to detect in games or test sets, yet result in suboptimal overall play.

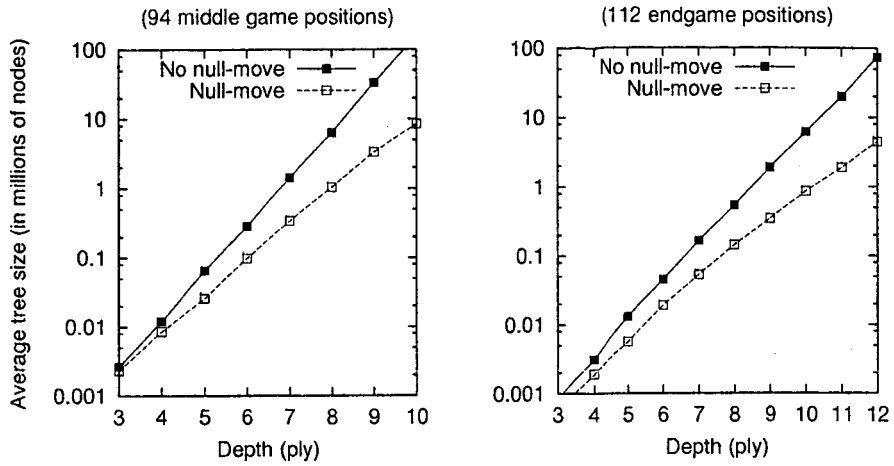

Figure 2.16: Node count comparisons for null-move pruning

A more fundamental risk is that, being a heuristic, its assumptions may not hold. There are two types of such failures. The first type has to do with the depth reduction causing horizon effects. Such mistakes will always be resolved in a later iteration. The second type of failure occurs when the null-move would actually be the best move in the node. To chess players, such positions are known as *zugzwang* positions. While rare during the opening and in the middle game, they are common in endgames. Here a deeper iteration will not reveal the true nature of the position, no matter how deep we search: the crucial parts of the tree are always skipped by null-move search, resulting in gross mis-evaluation of the position.

2.5.3 Verification search

In zugzwang positions we need to contain the risk of making such errors. The traditional approach is to turn off the heuristic in endgames. We choose not to do that in Rookie because that would throw away the depth advantage in these positions and because such a rule is a heuristic itself that can fail. Recall that our second design goal was that the program would be used as a stand-alone analyzer and, in principle, be capable of finding a position's true outcome. Permanent blind spots do not fit this.

Instead we redefined the way null-move prunes the tree. In conventional null-move pruning, after a fail-high on the null-move search all of the real moves are skipped. In Rookie 2.0 we *do* search these moves, but with a reduced depth of $d - R$: the verification search[19]. Of course, we normally expect one of the first moves to fail-high, so the extra node count is in the order of that of the null-move search itself. Figure 2.17 shows the tree with such a verification search.

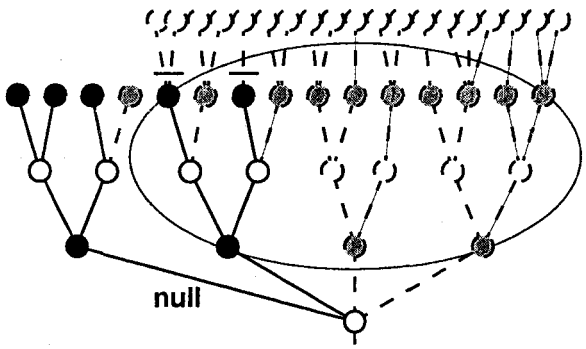

Figure 2.17: Null-move pruning with verification search

As we can see this still gives us a considerable node reduction over searching the full d plies, yet it overcomes the blind spots of zugzwang. In such positions we will not fail-high on the verification search but return a genuine score based on a true search result. Since such positions are forceful in nature, the loss of depth is still an issue: normally we prefer to search forced lines deeper instead of shallower (section 2.4). Therefore we added an extra criterion that turns off the null-move in nodes with a high zugzwang probability: it is not applied when the side to move has no rooks or queens and at most one bishop or knight. This is a more specific heuristic than turning it off in all endgame positions.

2.6 Move sorting

In section 2.1.2 the importance of good move sorting within the search tree is mentioned. Especially in the fail-high nodes we want to optimize the

[19]The concept of a verification null-move was independently developed and published in [Plenkner95].

likelihood of searching the best move first. More precisely: we like to search a move first that is good enough to raise the score above *beta*. To get a good move ordering we must define a suitable *pre-score* for each move in the node and sort the moves on those before continue searching. There are two types of pre-scores: *static* and *dynamic*. Static pre-scores are defined just in terms of the moves themselves and the positions they are made in. Dynamic pre-scores depend on state-space information that was collected during the search. Both must be combined to get a single value.

2.6.1 Dynamic ordering

In Rookie we use the transposition table moves, killer moves, counter moves and the history heuristic as dynamic ordering techniques. These all work throughout the tree. Iterative deepening (section 2.1.3) is also a dynamic sorting technique, albeit that it operates only in the root.

Transposition table moves

The most important dynamic sorting heuristic is starting with the move, if any, retrieved from the transposition table. When we find a move there it was the best move in this position before, so it is likely good enough now as well.

Killer moves

Killer moves are moves that were good in sibling nodes. It is one of the older sorting techniques in computer chess [Slate77]. In chess, often the same defense works against multiple moves, so if we remember what was good against one move in a node it is good to try the same reply to the next move. That is, if the move is legal in the new position.

In Rookie we keep slots for two killer moves in the parent of the sibling nodes that use them. We associate two popularity counters with each of them[20]. After the completion of each sibling we update the slots: either by increasing its counter if the move is already present, or by replacing the

[20] Actually implemented as one counter containing the difference between the two.

second-ranked move with the new one. Each node can read from and react to at most three killer moves: the two that are kept in the parent plus the best one in grand-grand-parent. This latter is added to anticipate delaying moves somewhat.

Counter move

The counter move heuristic works by observing that killers need not be constrained by sharing the same parent node. Often, a single move, defined as a (from,to)-tuple would generate the same refutation [Uiterwijk92]. Counter moves are easily implemented using a 64×64 table[21] that can hold the counter moves and that is indexed with the previous move. When we first added this to Rookie it gave an immediately visible node count reduction in most positions. (But see section 2.6.4 for its overall effect in the completed program.)

History heuristic

The history heuristic [Schaeffer83] is the last dynamic technique in Rookie. Like the countermove it uses a 64×64 table indexed with moves. Instead of moves it holds a popularity counter for each move. Within a node the moves are sorted according to these counters. After each node the slot that belongs to its best move is incremented, preferably with an increment that increases progressively with the remaining search depth d. In Rookie we use $d \cdot d$ for this. When an overflow occurs the whole table is scaled down by dividing all entries with some constant.

2.6.2 Static ordering

The only static ordering that we apply among moves in Rookie is based on the static exchange evaluator that we described in section 2.2. For capture moves this value is already computed by the capture move generators as described before. For each non-capturing move we collect the attacks and defenses on the target square and feed them to the same SEE-function. The

[21] Common variations use separate tables for both sides, or even for each piece type.

move can then be thought of as capturing a valueless piece, so its SEE result could at best become zero[22]. The effect is that moves to squares where they are easily captured automatically end up being searched last in the node.

A common additional static sorting technique could be obtained from computing the difference of the piece-square table values for the move's from-to squares (see 3.3.1). In our program, however, we cannot use the piece-square tables for this purpose. The reason is that, as we will see in more detail in chapter 3, our evaluator may not have them available for every node in the tree.

2.6.3 Implementation

There are two issues when implementing move sorting: Testing the legality of the suggested moves and combining the different techniques into one pre-score.

The suggested killer moves and counter moves must always be validated for legality. As they originate from other positions we can expect that often they will not be valid moves. The same holds for the transposition table move: after all, it could come from a different position due to a full hash-key collision. The problem is that an explicit legality test is expensive, even without testing for check. Also, for speed, the move generator does not just generate (from,to)-tuples, but also pointers to dedicated move making functions. We must generate these as well.

The solution is to do the testing *implicitly*. For this we use a 64×64 *priority* table that is initially filled with zeroes. Before move generation starts we place priorities in the entries for the moves that are suggested by the ordering heuristics. During move generation, for each move, we probe this table and fold-in the value found in the pre-score. After the move generation we clear the entries again for later re-use. Simple and effective, because illegal moves are not generated and thus not probed from the table. By storing the priorities in increasing order we can assure that the highest priority move will get the highest pre-score.

The pre-score is a 3-tuple in Rookie that is represented by a 16-bit unsigned word. Figure 2.18 shows the pre-score representation.

[22]Meaning that the piece will not be captured on the destination square.

bits	13...15	8...12	0...7
value	7 = transposition move 5 = first killer move 4 = second killer move 3 = old killer move 2 = counter move 0 = default	SEE (+offset)	history score

Figure 2.18: 16-bit pre-score representation

The lowest eight bits are for the history score. The next five bits hold the SEE value plus an offset to make sure it is positive in all cases. Five bits are needed to cover the extremes: moving a queen to an unsafe square (-9) and capturing a queen with a promoting pawn ($+9+9-1=17$). The most significant three bits are dynamic ordering again: it is zero for ordinary moves and contains a priority for suggested moves. Sorting the moves is done lexicographically on the 3-tuples, which is the same as sorting on the unsigned number that the 16 bits represent.

The move generator is responsible for generating the static, middle, part of the score. In reality, the 64×64 history table consists of 16-bit entries. Its high-bytes can then act as priority table, allowing the move generators to fold-in the two dynamic pre-score parts using a single read from memory.

2.6.4 Performance

It is normal practice to measure the effect when adding a new sorting feature to a chess program. It is also not surprising that the effect of individual contributions of each technique overlap. To measure the overall contribution of the various sorting techniques we have setup an experiment. We took a moderate set of realistic test positions and measured the average node counts for reaching a certain depth using the complete program[23]. Then we constructed several variations of the full program, each without one of the sorting techniques: iterative deepening, transposition table move, killer moves, counter moves and history heuristic. (Since the SEE-algorithm is so

[23] Also for these measurements we used the 94 middle game positions and 112 endgame positions from [Breuker96].

entangled with the quiescence search we have not included that aspect in the comparisons.) We repeated the node count measurements with these variations and plotted the results relative to the original program in figure 2.19[24].

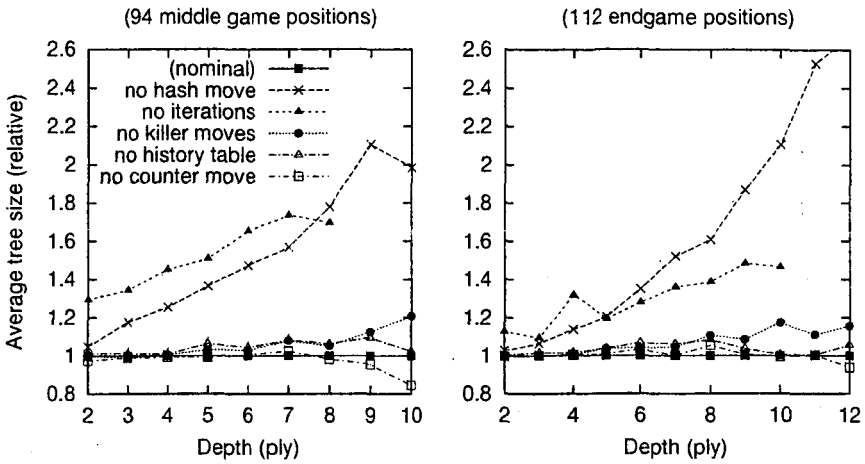

Figure 2.19: Relative tree sizes without each of the sorting techniques

From these measurements it is clear that the transposition table move ('hash move') and iterative deepening are the most important node savers. In middle games the table is responsible for saving half of the nodes, or a half to one extra ply of depth, and in endgames a bit more. When we compare the plots with the measurements of figure 2.12 we must conclude that during the middle game, transposition tables have their greatest effect for remembering earlier iterations, and a much smaller contribution for actually eliminating transpositions. Towards the endgame the role of the transpositions grows.

The next-best technique is the killer move heuristic, responsible for an approximate 20% node reduction. The history table contribution is small, but noticeable. The results of the counter move are most surprising: It has hardly any effect in the endgame, but removing the feature in the middle game speeds up the program by nearly 20%. So in the complete program is does more harm than good. The explanation that we can give for the latter is twofold: First, it is possible that the counter move has most effect when the static move ordering is not so good. Second, in the current Rookie,

[24]The runs for the iterative deepening test cases are 2 plies short for reasons that are beyond the scope of this report.

counter moves are given precedence over the SEE score. This may be the wrong order. We have not had the opportunity yet to test this hypothesis.

2.7 Implementation details

Now that we have described all of our search functionality, it is good to focus on a couple of implementation details.

Mate and stalemate In the pseudo-code of figure 2.2 we call the evaluator at the horizon or when the game is over. In chess, mate (win) and stalemate (draw) are end positions where the side to move has no more legal moves. So it is more effective to handle these situations in the search, which knows about moves. Therefore we do not have a game_over() function to test for mate or stalemate and the evaluator does not need to know about legal moves.

Draw by repetition The third way to end the game is when the players create a cycle and repeat positions. Technically, the position must be reached for at least the third time and one of the players must claim the draw, otherwise the game continues. Within the tree we can treat a repetition as a drawing move. There are two situations to distinguish in the search. If the created cycle is beyond the root position (the root position may be included), the first repetition can be treated as a draw. Otherwise, the cycle includes at least one move that was already made during the game. In that case we only return a draw score if the repetition is truly three-fold. This allows the program to find a better continuation by moving back to an earlier position and then selecting a different continuation, knowing that the opponent cannot claim draw yet.

Repetition detection It is tempting to use the transposition table to help detecting repeated positions. However, overloaded transposition tables can lose entries. So we choose something else[25]: a dedicated, small, repetition hash table. This table has 2^{14} one-byte entries that are initially zero (totaling

[25]Courtesy Ronald de Man for revealing this implementation trick.

16kB). When entering a new position, the low 14 bits of its hash-key are used to index the table and bump up its value by one. The value is restored after unmaking the move. When entering a node and its value is found to be non-zero already, we know there could be a cycle, which we verify by tracing back the actual variation. The repetition table is large enough to sufficiently reduce the number of false hits and this vaporizes the costs of futile back-traces.

Legal moves A move must not leave the king exposed to an enemy attack. It can be quite expensive to verify this in the move generator. Therefore we delay this test until the pseudo-move is actually made in the tree. Right after that we verify the attack status of the king's square to know if it was legal or not.

Check evasion generator There are three distinct move generators in Rookie. The capture generator was already described in section 2.2. The other two are the 'normal' move generator and more interestingly, the check evasion generator. When in check, the player is forced by the rules to escape check, so most pseudo-legal moves are illegal. It pays to design a special generator for that case. When the king is attacked by two pieces, the only escape is by moving the king. If the attack is by one piece, capturing it is also an escape. If this piece is a sliding piece, moving a piece between the attacker and the king is a third alternative. All other moves cannot be legal. Of course the moves generated in this way must still be verified for not exposing the king in some another way.

Piece lists For move generation we either cycle over the friendly pieces, or over the enemy pieces (captures). To avoid scanning the whole board we keep two piece lists, implemented as a small array with a size attribute. The king position is always at index 0: it is often needed. When a piece is captured, the list is compacted by copying the last entry into the gap.

Attack tables For the capture generator, the check detection and the check evasion (especially interposing pieces), we really need to have information about the attack status of each square. Such information is quite expensive to compute from scratch, so it should be updated on the fly. We

have two attack tables for each side. For each square it uses 15 bits to describe the minimum required information to make the move generators fast. It consists of 8 bits that flag sliding piece attacks, one for each direction. Two bits for pawn attacks (left or right), one bit for a king attack and 4 bits to count the number of knight attacks. This structure can be quickly updated with each move and provides just enough information when we need it.

Move makers The rules of chess make a generic move making function needlessly complex. When the move maker is also made responsible for updating hash keys, piece lists and attack tables it quickly becomes a big function with lots of conditional code. Such code not only executes slowly, the program is also throwing away information: most conditions are already known during move generation. We can avoid testing on many conditions by implementing dedicated move makers for dedicated move types. In Rookie we have special move makers for capturing each of the pieces. Also for non-capturing moves we factor out the piece type. On top of that we have special functions for en-passant capture, castling and pawn promotion and such. It is the move makers' responsibility to generate moves together with a pointer to the function that can make that move on the board.

Data moving With so many move makers and data structures around we are faced with the task of implementing their inverse functions as well. Not only is this error-prone, but it also increases the memory footprint again. So instead of that we decided to perform the move makers on *copies* of the data that we must keep invariant. If we do that, undo_move is a trivial function. Copying an attack table is, on average, about as expensive or faster than updating it. We keep the side-dependent data, such as the piece lists, within the same block that holds the attack table for that side. This brings us to the second advantage for copying: we now can *exchange* the friendly and enemy blocks during make_move(). Most operations in the move generators and evaluator do no operate on 'white' or 'black' data, but on 'friend' and 'enemy' data. Swapping these blocks with each move again eliminates a lot of conditional code in these modules.

Incremental move generation It is a waste of time to generate all moves just to find out that the first one causes a fail-high in the node. For that

reason, when not in check, the move generation is augmented in two parts. First, the capture generator is run. If it generates no favorable or equal captures, the remainder of the moves are generated. Otherwise, we insert a 'dummy move' in the move list with an artificial SEE value that is reserved between 0 and -1. Only until the search attempts to make this dummy move, the remainder of the moves are generated. This avoids unneeded move generation: most of the time, generating captures is good enough in cut-nodes. At the same time, it keeps the move lists short. In such short lists it is better not to sort the moves beforehand. Instead, we can simply make a fast linear search through the whole list to find the next move each time we need one.

Module testing With such complicated move generators and move makers, and with so many special cases regarding pawn promotions, en-passant and castling, it is too easy to make implementation mistakes. We therefore have built-in a second, simple, move generator. This one was designed to be simple instead of fast[26]. With such a dual implementation one can perform aggressive module testing based on automated long runs of generating full-width search trees with both generators. Under all circumstances should both systems generate exactly the same tree sizes. Otherwise we know we have an implementation problem. With this way of module testing we found many very subtle bugs in Rookie 2.0 that would otherwise probably never have been found.

[26] Actually, these are the chess functions that we use in the operator interfacing and for tracking the game as it is being played. When a search starts, its board representation is converted into the fast, but complex, representation of the engine.

Chapter 3

Position Evaluation

This chapter outlines the role of the *static position evaluator* (or just *evaluator* for short). We discuss the requirements that we set in Rookie 2.0 and how these are translated into a global decomposition. The speed considerations that drive the design tend to have a negative impact on the accuracy of the evaluation. We show how we address that issue with a triple-staged implementation. We also present a novel method for avoiding the type of consistency errors that similar lazy-evaluation solutions normally exhibit.

After that there is a detailed exploration of the heart of the design: the main data structures. Some examples demonstrate the capability of this design to scale and to absorb a wide spectrum of chess knowledge without losing much of its speed.

It turns out that our evaluator can also act as an efficient *detector* for some special types of tactical chess situations that are hard for the tree search to understand. We give some examples of the evaluator's side-effects that we can exploit for better tactical play.

Finally, the current status of Rookie's evaluation is summarized: how it performs and what kind of work remains to be done to fully exploit its framework.

3.1 Global analyses

As explained in section 1.4, the primary responsibility of the evaluator is to assess the search tree's leaf node positions. In games like chess, one cannot explore the tree exhaustively. So estimates are needed where we lack true game-theoretical values. These estimates must be computed both accurately and quickly, which are two competing requirements. This latter is driven by the observation that each ply of deeper search invariably leads to significantly stronger play. Therefore the evaluator must be lean and mean: the field has shown that it is hard to compensate depth with a slower but more knowledgeable evaluator.

The chess literature is rich of treatments on how to judge individual positions. Chess has been a popular game for many centuries and ever since the late 19^{th} century the world's strongest players have studied the game systematically and published their findings. Their nearly scientific approach has resulted in a vast corpus of texts about what makes good play and what does not [Philidor1749][Capablanca21][Euwe52][Kmoch56]. Being written to teach humans, such treatments usually focus on specific and common characteristics of positions and explain by example how these characteristics affect the player's winning opportunities. Hardly ever do they define with mathematical precision the conditions under which their treatment holds, neither do they accurately quantify the magnitude of their relevance with respect to each-other. It is left to the player's skill to decide when to apply such knowledge and how to balance it against the other possibilities.

For a computer player this is not good enough. If we are to use the existing chess knowledge in a program, the literature will give us a good start, but we have to specify the characteristics more accurately and quantify them numerically. We must also carefully balance the inclusion of such knowledge against its computational complexity and overall slowdown. It may sometimes be better to implement fast approximations instead.

From section 1.4 we recall that for each evaluation term t we have a pattern recognition function P_t and an associated weight W_t. In most cases, P_t will be predicate-like, yielding just 0 or 1. Implementing all P's as separate program functions will clearly slow down evaluation too much. Therefore these must be broken down into pieces sharing as many data and code as possible. This is the reason that the patterns are 'hard-wired' in Rookie's

C-code. For the W's there is no such reason to scatter them and in-line these in the C-code. Therefore, we prefer to keep them all in one place. In Rookie they are stored in a configuration file that is read during program startup. This allows tuning without recompiling the program.

3.2 Detailed requirements

In this section we first describe, in chess domain terminology, the knowledge that the evaluator must have. In this description we will attempt not to refer to the underlying implementation strategy, but in most cases we will mention the typical numerical values that we assign to these items. Such values are not something a chess player would normally talk or think about, but the numbers will help us later to guide the design. The values that we give as examples in this section are those that are used in the current version. The majority of these were determined by some superficial experimenting when implementing the corresponding feature. In fact, the issue of how to develop a well-founded calibration strategy is still open for Rookie.

Despite our attempt in this section to define some 'ideal' evaluator, we will see that the more we get to the function's details, the more it will be influenced by the pattern recognition capabilities that the implementation provides. So there is no strict waterfall development model here. Instead, global requirements lead to an implementation framework that in turn determines to what degree of refinement an 'ideal' function can be implemented. In some instances, the patterns may appear too simple, in the sense that there is an obvious generalization that gives more refined possibilities. The other side of that coin is that such generalization generates more values to tune that affect fewer positions. We do not consider this blurred area between pure evaluation ideas and implementation capabilities a deficiency of the design: one must keep in mind that in the end the function is intended as a heuristic approximation of some 'ideal' function that we can never hope to compute.

The position evaluator in Rookie 2.0 recognizes and scores chess patterns that are similar to those found in most other programs. In fact, they are similar to evaluation patterns as explained in chess learning books or game annotations in chess publications.

Our evaluator must take at least these categories of chess patterns into consideration:

- Material balance: traditional piece values, bishop pair, bishop versus knight, other piece combinations, basic endings.
- Pawn structure: doubled pawns, isolated pawns, passed pawns, weak pawns, pawn islands.
- King safety: pawn shelter, enemy attacking force, board control near the king, castling bonus, defending pieces.
- Piece placement: pieces on strong squares, rooks on open and half-open files, threats to enemy king, locked pieces, development
- Mobility and board control: controlling the center, and other important squares and regions, controlling enemy space.
- Basic endgames: special scoring for delivering mate, draws due to non-mating material, opposite colored bishop endgames.

The remaining subsections describe these terms in more detail.

3.2.1 Material balance

The material balance is the biggest term in the evaluation function. The queen is +8.800, rooks are +4.900, bishops +3.000 and knights +2.900. These values are close to the beginner's textbook values $9, 5, 3$, and 3. For pawns we have good experiences with a non-standard variable scoring: for each extra pawn the subsequent score is taken from the vector

$$\langle +1.500, +1.400, +1.300, \ldots, +0.800 \rangle.$$

So, for example, all eight pawns together are worth +9.200. In this scheme only the sixth pawn really has the unit value +1.000. The idea behind this scheme is to encourage the program to play less materialistic in the opening by not punishing the loss of a pawn too heavily. At the same time it makes the last couple of pawns very valuable.

Chess experts have found out that the material value of a piece can vary with the presence of other kinds of pieces, mostly irrespective of their precise location on the board [Timoshchenko93][Sturman96]. We recognize the following synergies between the pieces:

- a pair of bishops (+0.200), when they are of opposite colored squares
- a bonus for each knight if there still is a friendly queen (+0.120),
- a bishop and rook pair (+0.100),
- a knight bonus for each additional pawn (+0.048), to add the understanding that knights are relatively stronger than bishops in closed positions.

Some influential texts [Slate77][Condon83] mention a trade-off bonus that encourages trading pieces (but not pawns) when ahead. We have not added this in Rookie since we have not observed any problems to finish games from won positions.

3.2.2 Pawn structure

Understanding the pawn structure is the next most important aspect in chess evaluation [Philidor1749]. Pawns determine for a great part the 'shape' or 'space' on the board: where pieces can safely go and how hard it is to penetrate the enemy's position. Pawns move slowly and because the moves are irreversible, players must be careful advancing pawns without reason. As a result, structures change slowly during the course of the game and features created during the opening are often still present late in the middle game. All in all, pawn structures influence the game strategy for a long time. Its weaknesses become excellent focusing points for enemy forces and a sound pawn structure supports the own actions well.

The most important pawn features that we recognize are doubled, passed and isolated pawns. Figure 3.1 shows examples of these.

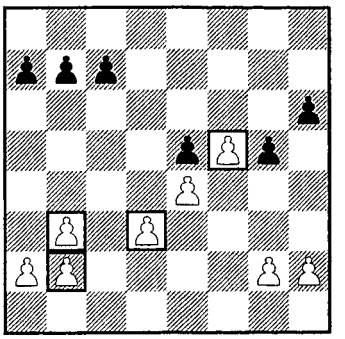

Figure 3.1: Pawn structure elements.

Doubled pawns

A player has a *doubled pawn* when two of his pawns are on the same file. This is a weakness in general. Our penalty depends on the file: for the a-file it is -0.200, b-file -0.160, c-file -0.160 and d-file -0.256. The penalties for the other files are given by observing file symmetry[1]. The penalty is really half of the above values and given for each pawn on a multiple-pawn file. So a tripled pawn is effectively treated 1.5 times as severe.

Passed pawns

A *passed pawn* is a pawn that could be pushed towards promotion without encountering enemy pawn opposition. We define it as a pawn for which no square on the file in front of it is either occupied by another pawn or under attack by an enemy pawn. Factors that make passed pawns more valuable include

1. the phase of the game (precisely: the amount of enemy material that could stop the pawn),
2. advancement: proximity to the promotion square
3. support from friendly pawns,
4. support from other pieces (especially the king),

[1]This symmetry rule is applied in all of Rookie's evaluation.

5. being blocked by other pieces or attacks, and

6. absence of enemy pieces within reach of the promotion path.

Not all these factors are currently accounted for in Rookie, but the most significant ones are. The bonus is $s_{hi}[r] - s_d[r] * a'$ (for $s_d = (s_{hi} - s_{lo})/7$), where r is the rank of the pawn minus 2 ($2 \leq r + 2 \leq 7$), s_{hi} is the vector of maximum possible bonuses by rank and s_{lo} is the vector of minimum bonuses. The enemy attacking strength is roughly quantified by a' and ranges from 0 (endgame with minor attacking material) to 7 (all pieces still on the board). The vectors s_{hi} and s_{lo} are different depending on whether or not the passed pawn is supported by a friendly pawn. We consider the pawn supported if a friendly pawn is present next to it or defending it.

$$s_{hi} = \begin{cases} \langle +0.148, +0.252, +0.500, +0.900, +1.400, +2.000 \rangle & \text{supported,} \\ \langle +0.100, +0.200, +0.300, +0.500, +0.900, +1.500 \rangle & \text{unsupported.} \end{cases}$$

For completeness we give s_{lo}:

$$s_{lo} = \begin{cases} \langle +0.148, +0.164, +0.236, +0.372, +0.872, +1.296 \rangle & \text{supported,} \\ \langle +0.136, +0.168, +0.148, +0.112, +0.100, +0.064 \rangle & \text{unsupported.} \end{cases}$$

So, in total, 24 values define the passed pawn bonus in Rookie and scores for $2 * 6 * 8 = 96$ cases are derived from these. The current evaluation does not yet factor in the vicinity of kings or pieces occupying or attacking squares in front of a passed pawn. Also, each file is evaluated equally. Later in this chapter we will understand why adding such extensions is relatively straightforward.

Backward pawns

A *backward* or *isolated* pawn is a pawn that is poorly supported by its neighboring pawns. They form a weakness because they need other pieces to defend them when under attack. Several aspects determine the type of backward pawn and its severity:

- The number of supporting pawns s. Those are the pawns next to it, or capable of becoming next to it, without being obstructed by another pawn, ($0 \leq s \leq 1$, since we do not consider a pawn backward when $s = 2$).

- Is the file half-open or not, and
- Mobility: is the square in front free of pawns and defended by at least as many friendly pawns as enemy ones?

For each of these $2^3 = 8$ combinations we have a penalty as shown in the table below:

	Pawn can advance		Pawn is stopped	
Supporters	Closed file	Half-open file	Closed file	Half-open file
0	-0.120	-0.260	-0.220	-0.360
1	-0.040	-0.120	-0.140	-0.220

As with passed pawns, a natural extension for this pattern would be to discriminate by file as well.

3.2.3 King safety

The king's safety is the most complex term to consider. It depends on factors such as the phase of the game, the presence of attacks on the king's position, the pawn structure protecting the king and the proximity of defending pieces. During the opening the *right* to castle into safety also has value. We define these aspects one by one:

Game phase

The protection of the king is most important during the opening and middle game. Towards the endgame, when most direct threats are gone, the king changes its role and can become a strong attacking piece.

Discriminating just three game phases is too coarse for a chess program. The lines between them will appear arbitrary and cause sudden evaluation changes when crossed. So we want to work with a more gradual measure of game phase. We also want to determine the game phase from both sides' perspective separately. After all, if one side is getting ahead, he needs to worry less about his own defense than the opponent.

Therefore, we relate the player's game phase p to the nominal material value

m' of his opponents pieces (without counting pawns). The high end of the scale represents the early opening and the low end is the far endgame. In between the value ranges from 0 to 7 and clipping is applied beyond these extremes. The phase-computation is given by formula 3.1, using integer arithmetic, and table 3.2 tabelizes some examples of this measure and gives a plot.

$$\begin{aligned} p &= ((m' - 10)/2 \ \mathbf{max}\ 0) \ \mathbf{min}\ 7 \\ m' &= 9 \cdot Q' + 5 \cdot R' + 3 \cdot B' + 3 \cdot N' \end{aligned} \tag{3.1}$$

Opponent pieces					phase
Q'	R'	B'	N'	m'	p
1	2	2	2	31	7
1	2	1	1	25	7
–	2	2	2	22	6
1	1	1	1	20	5
1	2	–	–	19	4
–	2	2	1	19	4
–	2	1	1	16	3
1	1	–	–	14	2
1	–	1	–	12	1
–	1	1	1	11	0

Figure 3.2: Examples for formula 3.1

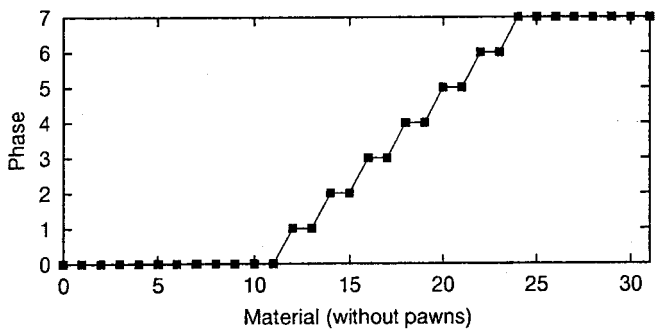

Figure 3.3: Plot for formula 3.1

Location and pawn shelter

In Rookie, vulnerability of the king is determined by its location and the protection that the pawn structure around it provides. The king is safest on the back rank and away from the center files. The pawn shelter is better when there are no open or half-open files near the king and when the protecting pawns are still on or near their original square.

The vulnerability is expressed as an integer that is 0 for a perfectly positioned and protected king. It grows for each deviation from that perfect situation. The location term is taken from the board table in figure 3.4. For the shelter we investigate the file that the king is on together with its left and right neighboring files. To the location term we add so-called *shelter values* of each of these files, counting the middle file twice. The shelter value for a file is given by the 4×4 matrix in figure 3.4. This matrix takes care of the above mentioned structural deficiencies; its axes range over possible pawn states for each side[2].

8	32	32	32	32	
7	28	28	28	28	
6	24	24	24	24	
5	18	18	18	18	...
4	10	12	14	14	
3	5	6	6	8	
2	2	2	3	4	
1	1	0	1	3	
	a	b	c	d	e...h

Black pawn (rank)	White pawn (rank)			
	none	2	3	4...7
none	8	1	3	6
7	6	0	2	3
6	7	0	2	3
5...2	8	1	2	4

Figure 3.4: White's king vulnerability per square (left) and structure (right).

King region attacks

The next aspect in king safety is the presence of direct enemy threats on its position. In Rookie we examine the (at most) eight squares around the king for pieces that attack these squares[3]. We separately sum the number

[2]When a side has more than one pawn on the file, we use the least advanced.

[3]Because we will not use the evaluator in positions where the king is in check, we ignore the case where there are attacks on the king's square itself.

of attacks from pawns, knights and sliding pieces[4]. We use these four sums as (clipped) indexes in a $4 \times 4 \times 4 \times 4$ table to find a suitable penalty. This penalty is progressive in the number of attacks: the penalty for two attacks is worse than twice the penalty for just one attack. In order to account for defending pieces, attacks are not counted for squares that are supposedly 'sufficiently defended'. To determine whether a square is 'sufficiently defended' we also examine the defenders of the square: each defense may cancel out at most one of the enemy attacks[5]. Only the remaining attacks, if any, are added to the sum. As a final remark it must be noted that we do not consider the kings themselves as defending or attacking pieces here.

Castling rights

In the opening before castling, the king is in an unsafe location. Without special measures a program would be tempted to move the king into safety too soon, without clearing the squares between king and rook first and then castling. Also the rook must be prevented from moving around by itself. A castling bonus remedies such anomalies. Whenever the king has not lost the *right* to perform a castling move, it is given a bonus. The bonus is +0.144 if the player could still castle in both directions, +0.100 if the right to castle to the queen-side was lost, and +0.072 if only the right to castle to the king-side is lost. This castling bonus is supposed to be large enough to enforce the required patience. At the same time it is small enough to leave a profit, evaluation-wise, by castling into a safer place.

It is worth noting that the hardware evaluation of Deep Blue [Hsu99] takes this term one step further by factoring in the pawn-shelter state and enemy attacks on the future king location. Doing this prevents the computer player from wrecking its safe haven and after that keeping the king in the center, sitting like a duck, holding onto its now worthless castling rights.

[4]Here we discriminate diagonal and orthogonal sliders, but not queens, rooks and bishops.

[5]We will not go into further detail of these cancellation rules here, because they are rather arbitrarily restricted to some implementation details. As a result they act more as a heuristic than as exact analyses.

3.2.4 Piece placement

In general, pieces are more valuable when they are developed, control the center, are mobile and hard to chase away by pawns and help attacking the king or enemy weaknesses. Mobility and board control go hand in hand and are treated in the next section (3.2.5). For each of the pieces we determine its positional score contribution separately.

Knights

Knights are more valuable when closer to the center of the board. Notably bad locations are the corners and edges. Keeping the empty boards' symmetry axes in mind, we distinguish just 10 different locations for a knight, equivalent to the the triangle a1-d1-d4.

	a	b	c	d	e...h
5...8					
4				+0.120	
3			+0.080	+0.120	
2		+0.120	+0.060	+0.000	
1	−0.200	−0.160	−0.080	−0.020	

Figure 3.5: Knight centralization scores

Each knight is also encouraged to move towards the *zone* where the enemy king is located, as defined by figure 3.6.

The distance to a zone is proportional to the the average distance to the squares that it is composed of. As metric for square-to-square distance we use a variation of 'Manhattan' or 'taxicab' distance here, one for which the difference between files is weighted slightly more than that between ranks. The result is scaled and clipped to the integer range $[0 \ldots 7]$. We will call this metric the *diamond distance*, because a plot of equidistant locations has a diamond shape[6]. Formula 3.2 defines this distance $d_{diamond}(s, Z)$ between a square s and a set of squares Z, using integer arithmetic:

[6]We give it a name to distinguish it from another metric, explained later, that we use for rooks and queens: *cross distance*.

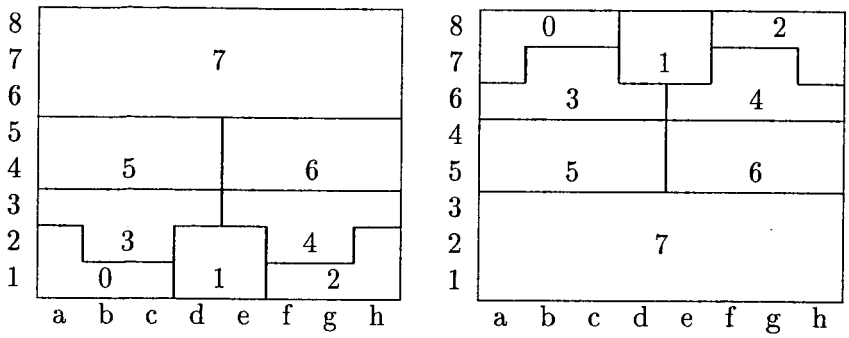

Figure 3.6: The king zones for white (left) and black (right)

$$\begin{aligned} d_{diamond}(s, Z) &= ((d'/(3 * |Z|) - 1) \textbf{ max } 0) \textbf{ min } 7, \\ \text{where } d' &= \sum_{z}^{z \in Z} (3 * |s_x - z_x| + 2 * |s_y - z_y|) \end{aligned} \tag{3.2}$$

As an example, the diamond distances to black's king zones 1 and 2 are given as the tables of figure 3.7.

	a	b	c	d	e	f	g	h
8	2	1	0	0	0	0	1	2
7	2	1	0	0	0	0	1	2
6	3	2	1	0	0	1	2	3
5	4	3	2	1	1	2	3	4
4	4	3	2	1	1	2	3	4
3	5	4	3	2	2	3	4	5
2	6	5	4	3	3	4	5	6
1	6	5	4	3	3	4	5	6

	a	b	c	d	e	f	g	h
8	5	4	3	2	1	0	0	0
7	5	4	3	2	1	0	0	0
6	6	5	4	3	2	1	0	0
5	7	6	5	4	3	2	1	1
4	7	6	5	4	3	2	2	2
3	7	7	6	5	4	3	2	2
2	7	7	7	6	5	4	3	3
1	7	7	7	6	5	4	4	4

Figure 3.7: 'Diamond' distances to black king zones 1 (left) and 2 (right)

The knight's king-proximity score is given by table 3.8. For $d = 4$ the score is defined to be nominally zero. In the current setting the bonus happens to be a linear function of distance. But since a table-lookup is used, the freedom to change that linear relation is there.

A *strong square* is a square that cannot be attacked by enemy pawns. They are excellent locations for pieces. A knight is given a +0.080 bonus for

	Knight $d_{diamond}$	Rook d_{cross}	Queen d_{cross}
$d = 0$	+0.064	+0.032	+0.048
1	+0.048	+0.024	+0.036
2	+0.032	+0.016	+0.024
3	+0.016	+0.008	+0.012
4	*+ 0.000*	*+0.000*	*+0.000*
5	−0.016	−0.008	−0.012
6	−0.032	−0.016	−0.024
7	−0.048	−0.024	−0.036

Figure 3.8: King proximity scores

occupying a strong square on the opponent's half, or +0.140 if this square is in the center[7].

Bishops

The bishop scoring does have a a table that encourages centralization, but the magnitude of its values is just a fraction of that of the knight. The bishop receives a −0.060 penalty for blocking undeveloped pawns on the two center files. This prevents moves like Bf1-d3 when there is still a pawn on d2. Furthermore, there is a −0.060 penalty for being on the same diagonal as a friendly blocked pawn. If the bishop is on the same diagonal as a enemy weak pawn, it gets a +0.080 bonus. Finally, the bishop receives the same bonus for occupying strong squares as that of a knight.

Rooks

Rooks are given bonuses for being on open files (+0.100), on half-open files (+0.060) and on files with weak enemy pawns (+0.044). A square table encourages rooks moving to the 7^{th} or 8^{th} rank (+0.200). King distance is awarded like that of knights, except that we use a different distance metric

[7]The strong square concept in Rookie 2.0 is rather primitive in its current form. A future refinement would use different scores depending on the distances to the center and the enemy king, and also factor in the presence of own pawn support.

(given in formula 3.3) and smaller values (see the table of figure 3.8) We will call this 'cross distance' because a plot resembles that of a cross, as shown in figure 3.9.

$$\begin{aligned} d_{cross}(s, Z) &= d'/(2 * |Z|) \textbf{ max } 0 \textbf{ min } 7, \\ \text{where } d' &= \sum_{z}^{z \in Z} \textbf{min}(4 * |s_x - z_x|, 3 * |s_y - z_y|) \end{aligned} \tag{3.3}$$

	a	b	c	d	e	f	g	h
8	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
6	2	2	2	0	0	2	2	2
5	3	3	2	1	1	2	3	3
4	5	4	3	1	1	3	4	5
3	6	5	3	1	1	3	5	6
2	6	5	3	1	1	3	5	6
1	7	5	3	1	1	3	5	7

	a	b	c	d	e	f	g	h
8	0	0	0	0	0	0	0	0
7	1	1	1	1	1	0	0	0
6	2	2	2	2	2	1	1	1
5	4	4	4	4	3	2	1	1
4	5	5	5	5	4	2	1	1
3	7	7	6	5	4	2	1	1
2	7	7	7	6	4	2	1	1
1	7	7	7	6	4	2	1	1

Figure 3.9: 'Cross' distances to black king zones 1 (left) and 2 (right)

Like the other pieces, rooks get the bonus for being on a strong square.

Queens

The queen scoring is the simplest. There are only the enemy-king proximity term and the strong square bonus. For king distance we use the d_{cross} metric and the values listed in figure 3.9.

King

In the endgame, where there are no more direct threats to the king, it is better for the king to move out of its shelter to become an active piece controlling the center and supporting pawn advancement.

For simplicity, this evaluation is always applied, even during the opening. Therefore the king safety evaluation that we discussed earlier must also compensate for the effects of this term, otherwise it would send the king into

action too early in the game. As the game progresses towards the endgame and the attacking forces are reduced, the king safety scoring becomes smaller and this evaluation term becomes dominant.

3.2.5 Mobility and board control

Mobility can be measured in a number of ways. It is tightly coupled to what data structures are already available. The direct approach is counting the moves that a player can make. The problem with this is that it requires generating moves in leaf nodes.

Board control is closely related to mobility. It determines which side controls the most squares. A square can be under control of the white pieces, the black pieces, or none. Center control and controlling the squares on the opponent's half of the board are more valuable. Therefore we divide the board in four regions that have different weights (see figure 3.10).

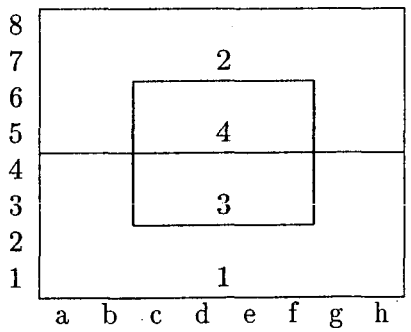

Figure 3.10: Relative square weights for board control

In Rookie 2.0 we base the mobility and board control terms on the attack tables. These tables are needed for move generation anyway (section 2.7). From it we collect information such as how many squares of each region are controlled by each side. We can also count the number of sliding attacks and compare that with the actual number of bishops, rooks and queens. If a piece is trapped somewhere on the board, we can detect that here in most cases because one of the counters will be low.

3.3 Design

Now that we have explored the spectrum of chess patterns that we must recognize, it should be clear that without some kind of up-front design the evaluator will become way too slow. Many of the patterns require scanning the board for features or depend on attacking patterns. We must avoid such scans whenever possible.

3.3.1 Dynamic piece-square tables

Piece-square tables are used in nearly every chess program as they are a fast structure to implement many evaluation terms. The idea is to pre-compute 768 values ($12 \cdot 64$), one for every kind of piece on every square of the board, and to use these in the actual evaluation. For each piece p on the board, the corresponding value

$$\text{piece\_square}[\text{type}(p)][\text{square}(p)]$$

is added to the score. Such a table is initialized before the search starts.

The big advantage of this is that the computation in the leaves is very fast. The piece-square sum can even be updated with each capture move in the search tree and thus be made available to the evaluator nearly for free. The second advantage is that it offers a place to add computationally expensive evaluation terms without affecting the tree search speed. One can make a more extensive examination of the positional structure because the tables are computed only once.

The disadvantage of these tables is that, because they are computed only at the root, they will become increasingly inaccurate with greater search depth: the positional features that were put into these tables may change in the tree, yet the scores are given as if they are still applied. For example, if the tables are set-up to encourage the queen to move close to a weak enemy pawn, and then the pawn is exchanged in the tree, the search will still try to hunt after this 'ghost' weakness in the deeper lines. This problem is real for more than a couple of moves down the line and gets bigger with increasing depth. In the end it makes no more sense to put anything in these tables but very basic evaluations like centralization of pieces, because these do not depend much on the changing positional features.

For Rookie we developed an entirely different strategy for piece-square tables. It was driven by an attempt to find a high-speed implementation of the evaluation ideas in [Levinson93]. (This article describes a way to build a mobility graph of the board and then to compute routes and distances of the piece sets to one another.) The idea is to compute the tables in the leaves, on demand, instead of in the root. At first sight this seems needlessly expensive: if the tables were recomputed in every leaf, 768 values will be computed of which just a couple are then used in the score, with the rest discarded. However, there is a refinement that we make: we *reuse* the tables among positions with similar structural properties. Positional structures change at a slower rate than the number of traversed moves. So if the tables only depend on such features, there are a lot positions that can share their tables.

The implementation of this idea involves a large enough pool to cache previously computed piece-square tables, a way to detect similar structural positions, and evaluation terms that only depend on such board structure and the location of a single piece.

We have chosen to let this structural component be a function of pawn structure plus king locations.

Tokens

In Rookie we call the 'characteristics' that define a class of related positions *tokens*. A token represents a feature that can be present in a position or not. We only have tokens for features that can be expressed in terms of pawn and king locations. Typical examples of simple tokens are

- open h-file,
- weak white pawn on e-file,
- white king in zone 2 (as in table 3.6),
- passed black pawn on d4, and
- blocked white pawn on f3.

The point is that tokens represent *just* the key features in a position from which we can generate meaningful piece-square tables. So we will not have tokens that just say 'white pawn on square x' or 'black king on y' because that is over-specific. Including these would make the mapping from pawn-king tables to evaluation tables one-to-one[8].

Since there is a finite number of tokens and they are typically parameterized by no more than one square, we can easily enumerate all of them using 16-bit integers.

We can detect the presence of simple tokens by inspecting three-file chunks: one center file's kings and pawns together with those on its two neighboring files, whose pawns attack the center file. All of the above mentioned examples are simple tokens.

Complex tokens are tokens that depend on the state of more than three connected files. Considering that tokens represent features of a position, and that features can be expressed as predicates, it should be no surprise that any complex token can be described as a logical combination of simpler tokens. Examples of a complex tokens are

- black king in 'the square' of passed pawn on a5, and
- square a1 is part of a rook prison.

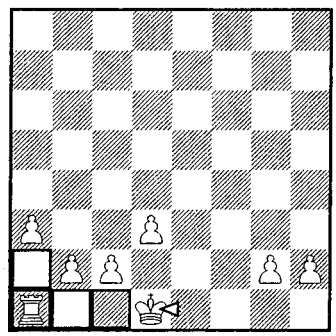

Figure 3.11: A rook prison.

[8]Having said that, we do have the four 'white/black e/d-pawn on starting square'-tokens to help the bishop piece-square table discourage bishops from blocking undeveloped center pawns (See also section 3.2.4).

The latter is visualized in figure 3.11. In this diagram, instead of castling, the white king has moved to d1 in search for a safer location. The result is a rook that is now confined to the squares a1, a2, b2 and c1. The white rook is now out of action unless white makes some more pawn moves to release it (and thereby wrecking its king protection). If we could recognize this pattern and assign penalty scores in the rook table for these prison squares, we can avoid the king move and encourage castling instead.

We will not go into further detail on how this complex token can be composed of simpler tokens. The key idea is that the partial tokens describe the chain a3-b2-c2-d1. There should be some kind of token-tree evaluator that takes partial tokens as input and emits the resulting complex tokens. In Rookie we have hard-coded such an evaluator in C for just a couple of such prisons. This demonstrated the extended possibilities of dynamic piece-square tables: the concept works, but the general case gets awkward this way. What is required to make this technique easy to use is an automated approach for decomposing complex tokens into simpler ones[9].

3.3.2 Multi-stage design

An interesting problem that lazy evaluation introduces is solved in Rookie 2.0: when to skip the remaining stages? If we skip too early the search may receive a value that is on the wrong end of its window. When a re-search visits this part again, with a modified alpha-beta window, it may suddenly fall in the full evaluation and get a value that is inconsistent with the earlier evaluation. Such search inconsistencies cause erratic searches, search order dependent results and likely weakened play. Although with the use of transposition tables and null moves we cannot completely eliminate search inconsistencies, we can eliminate those that are caused by lazy evaluation errors.

To break off correctly, we must know by how much the next stages can possibly affect the current partial score. Analytically examining these functions to find their extremes is hopeless: whenever we change some evaluation factors it must be done again. More important, the exact extremes may be overly large and unrealistic in practice. If we would use these mathematical

[9] One thing to consider in such a token-tree evaluator is that partial simple tokens that serve no other purpose than to construct a complex token, should not be included in the token list that identifies the `etb`.

bounds as thresholds in the lazy evaluation decision process, we would fall into the later stages far more often than really necessary. And this results in lots of wasted time and killing the idea of lazy evaluation in the first place.

Most programs are known to have artificially set limits as thresholds. The danger of this is that it leads to search inconsistencies: inconsistencies will occur where these bounds are to too tight for a given node, and when that node is revisited later with an alpha-beta window nearer to its score (not a rare situation at all). It is already hard enough to test and debug a chess engine without such errors.

So in Rookie we wanted to design an evaluator that suffered neither from speed loss nor from evaluation inconsistencies.

Our solution is that we use tight bounds between the stages. As an extra requirement, the later stages are forced to meet these bounds. To do this, at the exit of all later stages the partial evaluation is tested against the bound applied by the early stages. If it does not, the value is clipped to that bound. With this clipping technique we can easily tighten the bounds to be large enough to capture, for example, 95% of the scores, and the remaining extreme cases brought down to that maximum. This combines the best of two worlds.

In pseudo-code this clipping logic in the lazy evaluation is shown in figure 3.12.

3.4 Implementation

The evaluator operates in three consecutive stages (see figure 3.13). These stages each compute a part of the position value. The terms that contribute most to the value are computed first. Between consecutive stages the evaluator may exit early and skip the remaining stages. This happens when the partially computed score is so far out of the current alpha-beta window that the remaining stages could not possibly bring the score back into the window. This is known as *lazy evaluation.* The differences with known other programs are that

- in Rookie 2.0 lazy evaluation is applied between three stages instead

```
1   fun evaluate(pos, alpha, beta)
2   {
3       const stages = { stage_one, stage_two, stage_three }
4       const bounds = { 1.500, 1.000, 0.000 }
5       var score, lo, hi
6
7       score, lo, hi := 0.000, −∞, +∞
8       for (i in 0...2) {
9           score += stages[i](pos)
10          if (score − bounds[i] ≥ hi) return hi
11          if (score + bounds[i] ≤ lo) return lo
12          hi = min(hi, score + bounds[i])
13          lo = max(lo, score − bounds[i])
14          if (hi ≤ alpha) return hi
15          if (lo ≥ beta) return lo
16      }
17      return score
18  }
```

Figure 3.12: Multi-stage evaluation with clipping

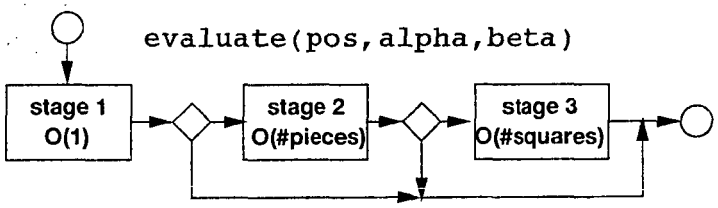

Figure 3.13: Triple-stage evaluation function

of just a fast one and a full one, and

- inconsistencies are avoided (see 3.3.2).

The motivation for lazy evaluation is speed: if we can avoid to compute the more expensive patterns we will search faster. Stage one in our evaluator computes the fast factors like material and pawn structure. Due to hashing, caching and incremental updates in the move makers, these are merely table lookups.

Stage two implements the dynamic piece-square tables. A loop is made over all pieces on the board and their piece-square value is added to the

score. Unlike other programs, we cannot incrementally update these partial evaluations because the board structure may change with a move. The board structure determines the contents of the piece-square tables. With the availability of incrementally updated piece lists this stage does not need to scan all squares to find the locations of the pieces. When during this scan a piece-square table is not found, it is computed. Typically this is needed in just 1% of all nodes, putting less stress on the efficiency of the piece-square table generator itself[10].

If the partial score sum after stage two is still close to the alpha-beta window, stage three is entered. Stage three inspects all 64 squares on the board. For each square it considers the attacks to the square. It determines who controls this square, how important it is (based on centralization), and keeps a couple of counters that accumulate this information. At the end of the board scan it examines these fields and applies evaluations for board control, mobility, king attacks and locked pieces.

Summarizing: stage one consists of table lookups, stage two is a piece scan, and stage three is a full board scan. Each stage takes an order of magnitude more time than its predecessor. Fortunately, the magnitude of the score contribution tends to decrease from stage to stage. The big terms are computed first. In the rare positions where this is not the case, the clipping as explained in section 3.3.2 enforces this.

Figure 3.14 shows the relation of the main main data structures in the three evaluation stages. The remainder of this section describes it all.

3.4.1 Stage one: fast evaluation

Stage one evaluates material balance, pawn structure and part of the king safety.

Material table

We do not look at the board, or scan the piece lists, to know which pieces are present. This information is kept in an unsigned 64-bit integer, as displayed

[10]... and thus allowing more knowledge in the tables.

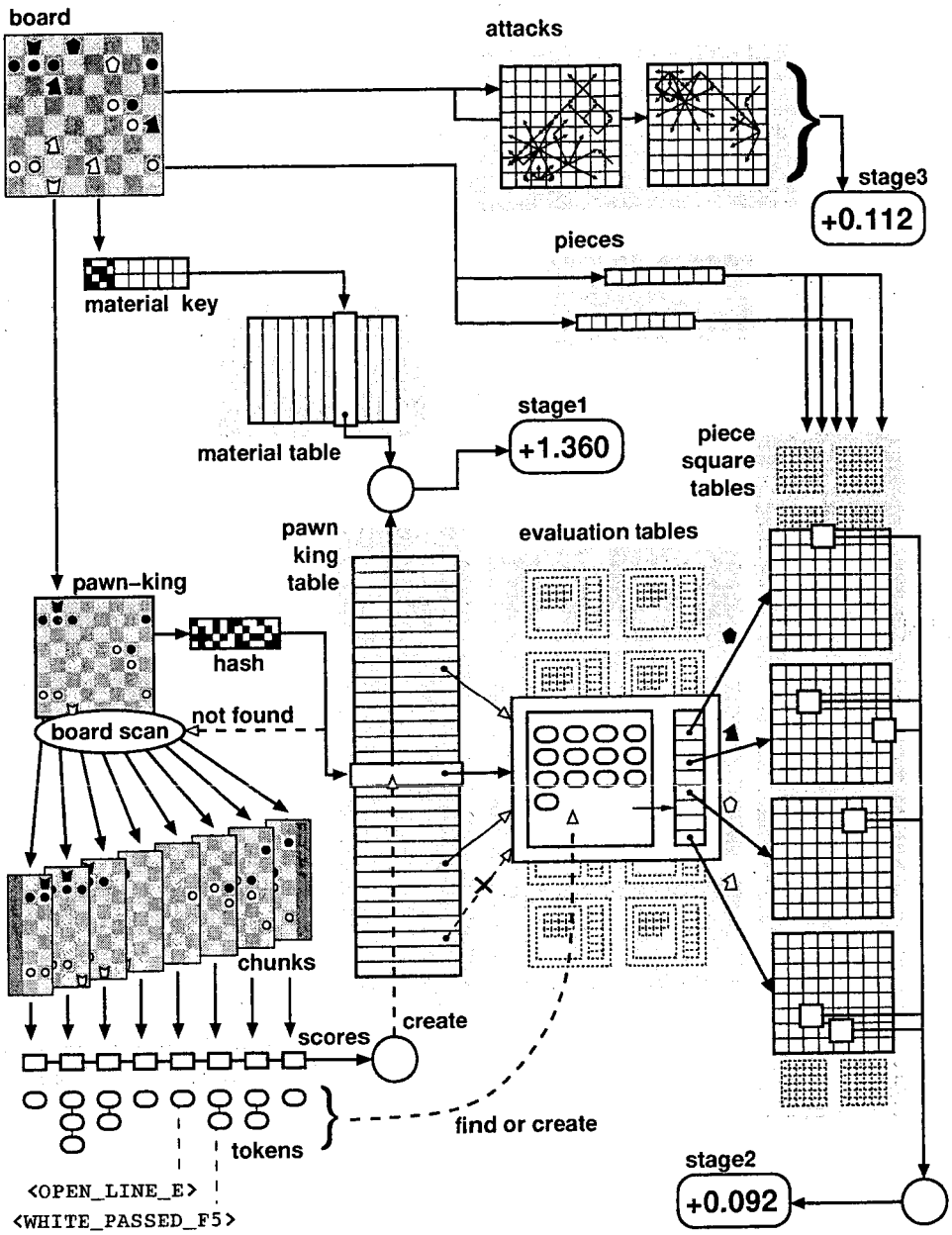

Figure 3.14: Overview of the Rookie 2.0 evaluator

in figure 3.15.

	White counters						Black counters					
hash	Q	R	B_l	B_d	N	P	Q'	R'	B'_l	B'_d	N'	P'
16	4	4	4	4	4	4	4	4	4	4	4	4

Figure 3.15: The 64-bit material key

This variable consists of twelve 4-bit counters, one for each piece type and side[11]. Four bits per field is sufficient to store any valid piece combination, even for artificial positions with, say, nine queens. The remaining 16 bits in the material key are reserved for a hash value. The hash is the sum (modulo 2^{16}) of a 'magic number' for each piece on the board. Each piece is assigned such a randomly chosen, but constant, 16-bit number. For each piece that is captured the corresponding 4-bit counter is decremented. At the same time the hash part is decremented with the magic number for that piece type. Both decrements are performed simultaneously by a single 64-bit substraction with a constant value that is determined by the piece type. This is possible because the counters are wide enough to prevent overflow and because the hash part is located at the high end of the variable, where underflow does not disturb the other fields. A similar update is made when a pawn promotes to another piece.

The evaluator uses the 16-bit hash to index a small array `material[]` to check if this configuration was computed and cached before. If found, it uses the stored data. Otherwise, it computes the material balance and writes it in the array.

A material table entry in this array is a 16 byte structure that contains:

- the 48 bits that identify this configuration (i.e., the material variable without the hash part)
- the material balance value (16 bits)
- game phase for each side (2 bytes), and
- a pointer that can point to a dedicated evaluator (usually `NULL`).

[11] Here we treat bishops that are on opposite colored squares as different pieces. So there are separate counters for light and dark square bishops (B_l and B_d).

The purpose of the last item is to link-in dedicated endgame evaluators or endgame databases. If this pointer is set, the function it points to is called instead of executing the remaining stages. This is only used for positions with just a couple of pieces. The function then either computes a special mating evaluation (one that drives the opponent king to the edge), or it performs a lookup in the disk-stored endgame tables with exact evaluations (draw or distance to mate). For this to work it is also needed that the light and dark squared bishops are counted separately.

Pawn-king table

The so-called pawn-king table in Rookie plays a central role in the whole evaluator, not only stage one, because it links to the piece-square tables.

Many of the pawn and king related terms described in section 3.2 can be computed without taking the presence, or locations, of any other pieces into account. A few of them can even be determined by just looking at the pawns of a three-file chunk. A bit more complex terms must factor in the enemy attacking force (like passed pawns and king safety). These are all dealt with in stage one. Any of the remaining terms that need more accurate information about other pieces are not evaluated by this stage.

For each position we have a 64-bit pawn-king hash. This hash is constructed from the the locations, piece types and colors of the pawns and kings, but ignores any other pieces. A number of high bits of this hash are used to identify a 16-byte entry in the pawn-king table[12]. Each entry in itself holds the 32 lower bits of the hash for verification. If these 32 bits match, the evaluator considers the contents of the found entry valid. Otherwise it discards and recomputes its contents.[13]

A pawn-king table entry occupies 16 bytes:

- **hash-key**: 32 bits of the pawn-king hash

[12]The exact number depends on the configured size. We typically use an 18 bits index. This results in a 4MB pawn-king table holding a quarter of a million entries.

[13]There is a room for error here, as the hashing scheme cannot be one-to-one. However, we consider fewer than one such error for every 4 billion evaluations rare enough to ignore this kind of mistake completely. As long as it will not crash the program the effect will be neglectable.

- `et`: a 16-bit reference to a stage-two *evaluation table block* (see 3.4.2)
- `color`: a 16 'color' of this evaluation table
- `score`: 16 bits score
- `light`, `dark`, `white`, `black`: four 8-bit values, described below
- `flags`: 16 bits to flag special conditions

The `score` field is simply added to the score for stage one. The use of `et` and `color` are explained in the next section.

The fields `white` and `black` indicate the vulnerability of the pawn and king score to the force of enemy material. We multiply these with the game phase (as found in the material table) of either side and add the product to the score. These fields are thus used to account for the game-phase dependent parts in terms like passed pawns and king safety.

The fields `light` and `dark` are used for bishop evaluation. Bishops are more valuable when there are more enemy pawns on its color of squares. Friendly pawns on this color reduce the bishop's value. The `light` field keeps the difference between black and white pawns on light squares. Blocked pawns count for three. The same measure for the pawns on dark squares is in `dark`. From these, the players' bishops get an additional score proportional to the either `light` or `dark`. For this we use, of course, the above-described material key that keeps track of each type of bishop separately.

Of the 16 bits in `flags` just two are currently in use. They help to detect pawn-race conditions that are further described in section 3.5.

3.4.2 Stage two: dynamic piece-square tables

Evaluation table

The central data structure in stage-two is a 256 byte *evaluation table block*, or `etb`. An `etb` holds, amongst others, 12 pointers to piece-square tables,

one for each piece type[14]. Positions with the same characteristics share the same `etb` and, consequently, use the same set of piece-square tables.

Evaluation table management

By default, memory for 2^{14} `etb`-s are allocated (4MB in total). For each invocation of stage two an `etb` is used. The pawn-king table holds references to these blocks. If a valid reference is already present in the king-pawn table, this is used. Otherwise the set of tokens that belong to the position is determined using a board scan for pawns and kings. Typically, more than multiple pawn-king entries share the same evaluation table, so the entire evaluation table is searched to see if the corresponding block already exists. To prevent scanning the whole pool, the blocks are spread over hash buckets.

If the block already exists, it is used and the reference in the pawn-king table is made to point to that table. Otherwise, a round-robin algorithm is run to select a currently used block and reuse it for the new position. When an `etb` is selected for reuse, all current references to it must become invalid. It is important to deal with that because the meaning of the block is about to change. Given the replacement frequency, it is too expensive to scan the entire pawn-king table in search for dangling references. It is also uneconomical to keep lists with back-references. To solve this problem, each reference to an `etb` is *colored*: every `etb` has a 16-bit value, its 'color', that is increased each time the block is reused. The pawn-king table also stores the actual color with its reference. When the pawn-king table color no longer matches that of the `etb`, the reference must have become invalid.

With this colored reference scheme the dangling pointers are taken care of at the expense of just a quick color check for each time an `etb` is accessed from the pawn-king table. The only time a scan over the entire pawn-king table is needed now is when a block's color field overflows. When that happens, we normalize all colors by resetting them to zero[15], and update the pawn-king table references correspondingly.

[14]Perhaps we could do away with the 4 pawn and king piece-square tables. In theory their contribution can be computed at the pawn-king table level, at the expense of some code complexity.

[15]Except for the color of the very first block in the pool: that one becomes 1 so that reference with index 0 and color 0 can be used as a null pointer in the pawn table.

With this is mind we can now summarize the fields in an `etb`:

- `color`: 16 bits holding the block's color
- `usage`: a 16 bit counter used in the round-robin algorithm
- `piecesquare`: 12 pointers to piece-square tables
- `tokens`: a sorted list of (at most 96) 16-bit tokens
- `hash`: a 32-bit hash code of the token list
- `next` and `prev`: two pointers in the doubly linked hash list

Piece-square table management

The piece-square tables that an `etb` points at belong to just one block. They are not shared with other `etb`'s, the mapping is injective. One may wonder why there is the extra indirection between `etb`s and piece-square tables. The answer is that embedding a full set of piece-square tables in each `etb` would consume far too much space.

Therefore, we maintain a separate pool of (typically 2^{16}, 8MB) piece-square tables that the `etb`'s have to compete for. Whenever a piece-square table lookup is needed and the `etb` only has a `NULL` pointer, a suitable 'donor `etb`' is selected and robbed from the piece-square tables it is holding. The donor selection follows the same round-robin strategy as the block selection we saw before: new and popular `etb`'s are more likely to keep hold of their entries.

3.4.3 Stage three: board scan

The final evaluation stage is the most expensive one, but the simplest to describe. Its purpose is to make the best use, evaluation wise, of the information that is available in the attacking tables.

We make a full scan over the board, inspect each square and determine which side controls it: how do the attacks on each square balance? We accumulate

the results over all squares and from that we should be able to extract some notion of mobility, board control score and king safety.

The attacking information is taken from the incrementally updated white and black attack tables. For each square and side these hold 15 bits of information: 8 bits for sliding piece attacks (1 bit per direction), 1 king-attack bit, 2 pawn-attack bits (left en right) and 4 bits to count the number of knight attacks.

For sliding pieces, Rookie's attack tables do not tell if the attacker is a queen, rook or bishop. Just the direction that the attack comes from is known. On top of that, the tables are blind for x-ray attacks, pinned pieces and batteries.

With these restrictions it may seem that the square investigations are not as precise as one may think is required for a good evaluation. However, the result is still worthwhile. Counting all 64 squares reduces the noise somewhat and the final outcome still serves as a reasonable heuristic to evaluate board control and mobility. When this stage was added to Rookie 2.0 we observed a significant increase in playing strength on the chess servers[16].

3.5 Exploiting side-effects

The evaluator can be used to help the search as well. We mention three areas: pawn races, endgame databases and special endgames.

Pawn races

Pawn races happen when a passed pawn can run for promotion without being stopped by a piece. When there is just one pawn running, the evaluator can anticipate the future value of the promoted pawn. When both sides have pawns running, this is too hard to resolve statically. The first pawn, after promotion, may or may not be able to stop the other pawn. Preferably, we would like to extend the search just enough to see the promotions happen.

[16] We have not measured the precise rating effect at the time, but our estimate would be about 150 points.

The evaluator can detect such situations and set a flag for the search. The search can then react and apply search extensions for the pawn moves.

Endgame databases

Endgame databases are disk-based tables of pre-computed game outcomes for basic endgames. When there are just a few pieces left, the endgame can be solved by enumerating the whole state-space and work out the game-theoretical outcome of each position.

The evaluator can detect, by its material table, that the true value of position is available on disk. Since disk access is slow, it is sometimes better not to probe the disk in all leaf nodes. Here, the evaluator can signal the search that it has detected a known endgame. The search can then decide, based on remaining depth, if a probe for disk should be made or not.

Special draws

As stated in the introduction, the purpose of the Rookie program was not just to play good competitive chess, but also to act as a good position analyzer program. This means that it should be able to understand puzzle-like positions that normally do not occur in regular games. Sometimes such positions are too hard for a regular search. One such case worth mentioning is illustrated by figure 3.16.

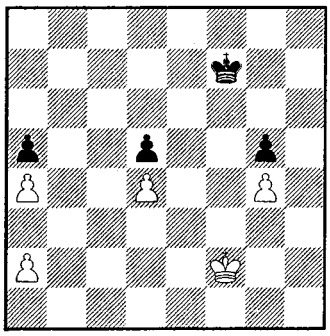

Figure 3.16: A dynamic draw.

This position is a draw because neither side can capture or even attack any of the opponent's pieces. This should be obvious to a human observer but not to an unprepared tree searcher: the lines that lead to the game-technical draw by repetition are very long. Even worse, a naive searcher will consider white's extra pawn as an advantage and will let white run away from repetition until finally forced to accept its fate. Transposition tables are of little help here: the number of possible positions is limited, yet the number of repetition evading paths is virtually endless. An unprepared program may need well over 20 plies of hard search before it starts to settle with a draw score.

Our evaluator is capable of absorbing such rare patterns without compromising the overall execution speed. Just like in the case of the rook prison in section 3.3.1, we could make use here of *complex* tokens to detect partial or complete pawn-king fortresses. In this case we can let a token tree detect the blocked pawn formation that splits the board. If that is the case we can run a dedicated check to see of the pawns are out of reach of the kings. If so we set a special flag in the pawn-king table. When a set flag is found by the evaluator, it checks if no other pieces are present[17]. If so, the position must be a draw and the evaluator can signal back to the search that it does not need to search beyond this position.

3.6 Evaluator performance

Summarizing, we can say that Rookie's evaluator is capable of recognizing the following pattern categories:

- anything material-count related,
- any pattern related to just pawns and kings,
- any pattern related to pawn-king structure versus material count,
- any pattern related to a single pawn in relation to the board's pawn and king structure,
- reasonable board control,

[17]If other pieces are present it is possibly still a draw, but harder to detect. We have not experimented with that.

- reasonable mobility,
- inter-piece relations whenever expressed in terms of attacks,
- far endgames and
- some special cases like the ones described in section 3.5.

The staged implementation ensures that little evaluation time is spent in nodes that are less critical. Table 3.17 below gives the relative number of nodes for which each stage is invoked:

Stage	Invokations (per node)
1 (lookups)	71%
2 (piece-square)	13%
3 (board scan)	7%

Figure 3.17: Stage profiles

Finally, the most important data structures for caching and re-using intermediate results show good efficiency, as revealed by table 3.18. A higher hit-ratio means that the cost of the cached function contributes less to the overall performance overhead, allowing it to contain more knowledge.

Table	Memory	Hit-ratio (per node)
Pawn-king table	4MB	97.1%
Pawn chunk table	4MB	99.1%
Evaluation table	4MB	99.5%
Piece-square table	16MB	98.7%

Figure 3.18: Table efficiencies

Chapter 4

Opening Book

4.1 Opening books

An opening book is a database containing chess opening theory. This theory is very specific. It states how good specific moves are in specific positions that often occur in games. As long as the game is still following established opening lines the program will immediately make one of the moves suggested by this database. Its presence in chess programs serves three important purposes:

1. To save time in the first phase of the game. Often the established theory lasts well over twenty moves. Without book one would be faced with a substantial time disadvantage when the middle-game is reached.

2. To avoid known and deep traps that otherwise would take too much time to be discovered during the game. There is no need to reproduce years of deep grandmaster analysis of lines on the board if these lines are instantly available.

3. A well-chosen opening repertoire helps the player to reach positions where his specific skills are most likely to be productive. For computers we could use the book to avoid the kind of positions that are too difficult for machines to understand.

An additional advantage is that opening books can be used to randomize a program's play to avoid repeating the same games over and over again.

The design of an opening book for a chess program can be a full-time job. Someone needs to analyze the program's games to carefully select the lines that go best with the program's playing style. This requires a strong chess player who is able to devote large amounts of time to develop such a book. As long as the program is still under development this work could be completely wasted whenever the program undergoes significant changes. Therefore we adopted the approach of automatically extracting a book from games played by strong players.

The rest of this chapter explains how our opening book is implemented.

4.2 Book move selection

Our opening database does not contain moves but only positions. The corresponding moves are inferred from the presence of the resulting 1-ply positions. The advantage is that moves can be recognized that lead to known positions without actually needing to have a game in the original game database where that move was played. So new transpositions can be easily recognized. The disadvantage is that in such situations these moves may actually not be the best one. In an attempt to deal with this problem we only play a move directly from the book if both the current and the new board position are present in the database. In order to recognize transpositions with reversed colors we normalize the positions by reflection and color flipping so that white is to move in the resulting positions.

Each position p carries a weight, $w_p (0 \leq w_p \leq 1)$, that reflects the goodness of the position. Also we know the number of games g_p where this position has occurred in before. Finally we have the average game result r_p from these games for the player to move. (For example, r_p is 0.75 if we have one won game and one drawn game for this position).

If n moves are available to be played then the probability for a move m (leading to p) to get chosen is proportional to $P(m) = (1 - w_p) \cdot g_p \cdot (1 - r_p)$. This way we favor moves with higher winning expectations and also we prefer to choose moves that will keep us in the book as long as possible.

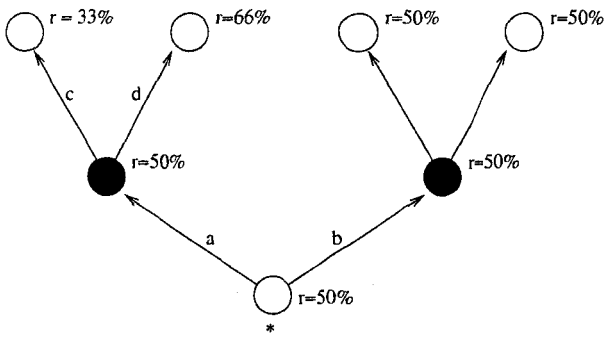

Figure 4.1: Four book lines with average game results

Before we select the move based on these values we first filter out those moves m for which $P(m)$ is below the 'noise' threshold of

$$\sqrt{\frac{1}{n}\sum_{i=0}^{n} P(m_i)}.$$

This is to prevent moves from being played that either occurred too infrequently or that have lost too often. The filter threshold also gives some protection against mistakes in the original raw game data. (For example it raises an extra barrier for simple blunders and typographical errors to affect our openings.) This also prevents moves from being blindly replayed for which we have only one game that was lost.

4.3 Tuning

The weights w_p are initially computed during book generation. A simple way to choose w_p would be to make it equal to r_p. This is our first approximation for w_p. However, this may give surprising results as is demonstrated in Figure 4.1. The Figure shows four possible opening lines, extracted from a number of games, together with the average game result for the player in each position. (For simplicity assume that we have just as many game for each of the opening lines.) At first sight the game results suggest that we could arbitrarily choose between playing move a or b. Both moves lead to positions where the the average score is 50%. However, move a is slightly

worse as it allows black to play move c, resulting in a position where white has scored no more than 33%.

To help overcome this problem we wish to decrease the probability to select move a. One way to do this is to assume that after move a black is more likely to play c rather than d. Given the game results and the move selection probabilities from the previous section we assume that black will actually prefer to play c 66% of the time. We then re-estimate the expected game results in this node as $(1-0.66)\cdot 0.66+(1-0.33)\cdot 0.33 = 55\%$. This becomes our new w_p for the position after a. If we do the same for the position after move b we still get 50%. So now the weights in the black nodes reflect the slight preference to play move b.

In the same way we can compute a new w_{root} from these newly computed weights. This back-propagation of weights is performed throughout our entire opening book at construction time. In our current implementation it is a time-consuming procedure taking over 6 hours of CPU-time on a 250MHz UltraSparc machine. The result is a slightly better tuned book that forces into better positions. Refuted opening lines that where once popular (and thus have a high g in our database) are now less likely to be chosen.

4.4 Learning

Using the tuning technique from the previous section we can already build a reasonable book from raw games without much effort. This book however isn't yet tailored for the specific strengths and weaknesses of the program. This last step is taken care of by a book learning function that adjusts the weights using the results of games played by the program.

The basic idea is to modify the weights w_p for the book positions p that

1. occurred during the game, and
2. where the opponent was to move.

After all, these are the weights that have influenced our opening selection. If the game was *won* then the selection could be favored next time by *decreasing*

the weights a bit. If the game was lost the weights apparently were too low and should be raised to encourage other moves to be played next time.

The question that immediately rises is in what way to affect the weights. After all, losing from a stronger player might not indicate a bad opening choice at all. And beating a *patzer* by using an unsound opening line does not mean the opening was good. These problems can be met by taking relative strength of the opponent into account. This is straightforward when games are played under rated conditions, like on the Internet chess servers. In Rookie we use a kind of exponential smoothing to adjust weights:

$$w_p := (1 - \alpha) \cdot w_p + \alpha \cdot (0.5 - \frac{R}{40})$$

Where R is the rating change in points (truncated at -20 or 20 when the change is outside these bounds). The speed of learning is determined by α. (We typically take α to be 0.20)

It is clear that our book learning only affects positions that are already present in the book. It cannot help learn 'new' openings. Even though this scheme is very simple and not so sophisticated it does seem to have a positive effect when many games are being played and when accurate strength estimates of both players are known. Positions that are more often won than lost slowly get played more often, and less successful openings are played less frequently.

4.5 Storing positions

Currently Rookie's position database contains over six million positions, taken from over 11,000 games. The database must be designed to

1. occupy as little disk space as possible, and
2. to allow fast lookups.

The second requirement is met by storing all positions in lexicographical order of representation. Positions can now be found by applying a binary

Huffman piece encoding	rank 2 to 7		rank 1 or 8	
	White	Black	White	Black
King	101111	111111	10111	11111
Queen	101110	111110	10110	11110
Rook	10110	11110	1010	1110
Bishop	10101	11101	1001	1101
Knight	10100	11100	1000	1100
Pawn	100	110	n/a	n/a
Empty square	0			

Figure 4.2: Prefix codes for compact position encoding

search on the disk file. Remember that to find a book move we must look up the current position as well as all positions that can be reached by making a move. Because all positions are normalized such that is white's move we can choose an ordering such that most 1-ply positions are contained in the same area of the database. (Most moves don't affect the high order part of the representation.) This way we can localize disk access, which is usually fastest in modern file systems.

To get a compact position representation we apply two successive compression steps.

The first step is to convert the 64 squares of the board into a bit string using a Huffman [Cormen95] representation as shown in Figure 4.2. Empty squares here only take one bit, Pawns three, and Kings 6. An exceptional coding can be used for the first and last rank on the board: As those parts cannot contain Pawns these codes can all be made one bit shorter. Invisible position information such as castling status and en-passant rights must be appended to this bit string when needed.

The strings generated by the first step are still too large to be stored completely. The second step uses the observation that the positions are going to be stored in lexicographical order. This means that successive positions in the disk file are likely to closely resemble each other. We prepend 8 bits to the code to indicate which of the 8 ranks contain changes in the position from the previous one. Likewise we prepend 8 bits that flag which files contain changes. Only the bit string codes of the squares that are both on a changed rank and on a changed file are stored in the file. The contents of

the other squares can be deduced if we know what the previous position, in storage order, looked like.

For example, if a position differs only in the squares c2, e2, e5, f2 and f5, we mark the files c, e, and f, and the ranks 3 and 5 as changed. After that we just list the contents of the 6 squares that are on the intersections of these files and ranks.

Because this second scheme requires a linear sweep to decode the stored positions we split the file into chunks of 4 kB each. The first position in each chunk is not compressed by the second step. This assures that we only have to decode a small fraction of the database to determine if a position is contained in a particular chunk.

A typical look up for the 1-ply positions now consists of:

1. sorting the positions we want to find,
2. applying a binary search to locate the 4 kB chunk that should contain the first position
3. a linear run through the chunk to find the positions we are interested in,
4. repeating this procedure from step 2 if there may still be positions left in a (later) chunk of the database.

The effectiveness of this two-level scheme is quite surprising. Finding book moves in the database is very fast. The time needed for disk access during a game does not notably slow down the program. Also the space needed to store the 6 million positions (plus weights, game counts and result statistics) is a modest 80 MB, or fewer than 13 bytes per position. Of those bytes, on average 75 bits are needed to actually encode a chess position. [1]

[1] Balkenhol [Balkenhol94] describes a complicated compression scheme that takes 79 bits on average for a position to be encoded. However, this result is not directly comparable to ours as we take advantage of redundancy between successive chess positions.

Chapter 5

Conclusions

With this report we give an extended overview of the design considerations and implementation decisions in Rookie 2.0. The development of this program represents about a year of implementing and testing. The program plays a decent game of chess and is robust: in its current life it is playing roughly 200 blitz games per day against human players on Internet chess servers.

The design ideas for this version were grown during some years of experimenting with earlier versions. Version 1.0, for example, pioneered the attack tables and dynamic piece-square tables. In this report we concentrate on the tree search, position evaluation and the opening book. That is the heart of the design, but not all of it. We have not described smaller topics in much detail, such as probing perfect-knowledge endgame databases, time management and game 'conduct' (ending the game by accepting draw offers or resigning).

5.1 Results

Rookie 2.0 is capable of playing chess under tournament conditions and it has been active on several Internet chess servers. Such servers are the Internet equivalent of chess clubs, where people from all over the world come to play chess. On these servers Rookie has played games all day long

against a variety of opponents, both humans and other computers. Over the years, is has played over 100,000 blitz games using a Pentium-II 360MHz computer[1]. On this machinery it has achieved the strength of a very strong club player, but not yet the level of a chess master. Figure 5.1 gives its average rating during 2001 plotted against the rating distribution of the entire server population of that year.

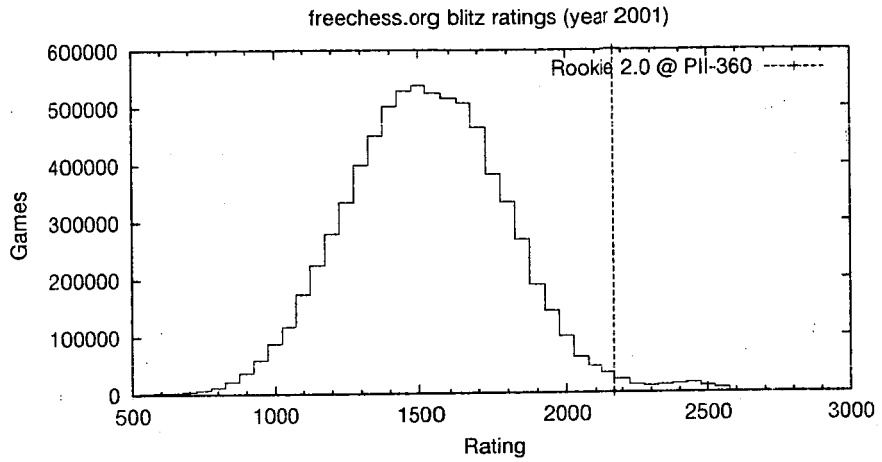

Figure 5.1: Population rating distribution and Rookie 2.0's average

The rating system is relative: the difference between two players predicts the expected score between them. A 100 point advantage corresponds to a 70% winning expectation, 200 points give 85% and 300 points 95%. On the other server (ICC) its absolute rating is consistently 300 points higher, but since those are separate rating pools the comparison has no meaning.

An early Rookie 2.0 participated in the Dutch Open Computer Chess Championships of 1997 and the full version in that of 1998. In that tournament it achieved a score of 50%.

The strong point is its speed in combination with its capability to absorb more evaluation knowledge. It is much faster than most other amateur programs. This can be attributed to

- the move generation and move making structure,

[1]All our played Internet games can be browsed at http://bitpit.net/blik/.

- the fast dynamic piece-square tables,
- and the SEE implementation that keeps the trees well-sorted and the quiescence searches short.

Its current weakest point is the tuning of the evaluator. We have not given tuning the attention it deserves. Most notably, its passed pawn evaluation is too optimistic. This is a matter of calibrating the evaluation weights in the configuration file, not re-programming the engine. While the program is stronger than most amateur programs on the same hardware, in comparison to the strong programs it is rather weak. This can fully be attributed to the badly tuned evaluation.

5.2 Future

For the future of Rookie the greatest challenges are getting the tuning right, adding parallel search, making smarter search extensions and making the quiescence search even more stable.

For the search we need to start another rewrite: currently the code has a single recursive function search() that performs PVS, regular search and quiescence search. Extending this is hardly possible.

For the tuning we have a more difficult challenge. We first need to find a good way to *measure* improvement. Playing real games on servers is too slow: the ratings are so volatile that it takes thousands of games before the rating noise is sufficiently reduced. And test sets with positions that cover every aspect of chess evaluation quickly grow too large. A good test set may very well be as large as 100,000 positions.

A final thought should be given to assessing the precise value of dynamic piece-square tables as compared to traditional, root-based, tables. It would be good to set up an experiment to measure their difference using, for example, self-play against a dumbed down version of Rookie 2.0: one modified to compute only piece-square tables in each new root position and to use these throughout the search.

Bibliography

[Allis94] *Searching for Solutions in Games and Artificial Intelligence*
V. Allis (1994)
Ph.D. Thesis, University of Limburg

[Balkenhol94] *Data Compression in Encoding Chess Positions*
B. Balkenhol (1994)
ICCA Journal, Vol 17, No. 3, pp. 132-140

[Breuker94] *Replacement Schemes for Transposition Tables*
D.M. Breuker, J.W.H.M. Uiterwijk and H.J. van den Herik (1994)
ICCA Journal, Vol 17, No. 4, pp. 183-193

[Breuker96] *Replacement Schemes and Two-Level Tables*
D.M. Breuker, J.W.H.M. Uiterwijk and H.J. van den Herik (1996)
ICCA Journal, Vol 19, No. 3, pp. 175-180

[Breuker98] *Memory versus Search in Games*
D.M. Breuker (1998)
Ph.D. Thesis, Universiteit Maastricht

[Capablanca21] *Chess Fundamentals*
J.R. Capablanca (1921)

[Condon83] *Belle Chess Hardware*
J. Condon and K. Thompson (1983)
Advances in Computer Chess 3 (ed. M.R.B. Clarke) pp. 45-54

[Cormen95] *Introduction to Algorithms*
T.H. Cormen, C.E. Leiserson and R.L. Rivest (1995)

[Donninger93] *Null Move and Deep Search*
Chr. Donninger (1993)
ICCA Journal, Vol. 16, No. 3, pp. 137-143

[Euwe52] *Oordeel en Plan*
M. Euwe (1952)

[FIDE97] *FIDE Laws of Chess*
Fédération Internationale des Échecs (1997)

[Goetsch90] *Experiments with the Null-Move Heuristic*
G. Goetsch and M.S. Campbell (1990)
Computers, Chess, and Cognition (ed. T.A. Marsland and J. Schaeffer) pp. 159-168

[Hartmann89] *Notions of Evaluation Functions tested against Grandmaster Games*
D. Hartmann (1989)
Advances in Computer Chess 5 (ed. D.F.Beal) pp. 91-141

[Hsu90] *A Grandmaster Chess Machine*
F.-H. Hsu, M. Anantharaman, M. Campbell and A. Nowatzyk (1990)
Scientific American, October 1990, pp. 18-24

[Hsu99] *IBM's Deep Blue Chess Grandmaster Chips*
F.-H. Hsu (1999)
IEEE Micro, March-April 1999, pp. 70-81

[Kaindl90] *Tree Searching Algorithms*
H. Kaindl (1990)
Computers, Chess, and Cognition (ed. T.A. Marsland and J. Schaeffer) pp. 133-158

[Ke93] *The Guard Heuristic: Legal Move Ordering with Forward Game-Tree Pruning*
Y.-F. Ke and T.-M. Parng (1993)
ICCA Journal, Vol. 16, No. 2, pp. 76-85

[Kernighan88] *The C Programming Language*
B.W. Kernighan and D.M. Ritchie (1988)
Second Edition

[Kmoch56] *Die Kunst der Bauernführung*
H. Kmoch (1956)

[Knuth75] *An Analysis of Alpha-Beta Pruning*
D.E. Knuth and R.W. Moore (1975)
Atrificial Intelligence, Vol. 6, No. 4, pp. 293-326

[Levinson93] *DISTANCE: Toward the Unification of Chess Knowledge*
R. Levinson and R. Snyder (1993)
ICCA Journal, Vol. 16, No. 3, pp. 123-136

[Nimzowitsch25] *Mein System*
A. Nimzowitsch (1925)

[Philidor1749] *L'analyze du Jeu des Échecs*
A.D. Philidor (1749)

[Plenkner95] *A Null-Move Technique Impervious to Zugzwang*
S. Plenkner (1995)
ICCA Journal, Vol. 18, No. 2, pp. 82-95

[Schaeffer83] *The History Heuristic*
J. Schaeffer (1983)
ICCA Journal, Vol. 6, No. 3, pp. 16-19

[Shannon50] *Programming a Computer for Playing Chess*
C.E. Shannon (1950)
Philosophical Magazine, Vol. 41, pp. 256-275

[Slate77] *Chess 4.5: The Northwestern University Chess Program*
D. Slate and L. Atkin (1977)
Chess Skill in Man and Machine (ed. P.W. Frey), pp. 82-118

[Sturman96] *Beware the Bishop Pair*
M. Sturman (1996)
ICCA Journal, Vol. 19, No. 2, pp. 83-93

[Timoshchenko93] *Bishop or Knight?*
G. Timoshchenko (1993)
ICCA Journal, Vol. 16, No. 4, pp. 209-216

[Uiterwijk92] *The Countermove Heuristic*
J.W.H.M. Uiterwijk (1992)
ICCA Journal, Vol. 15, No. 1, pp. 8-15