The Morphy Project

- Distributing a chess game

DAT3 / F7S project, Fall 2004.

By projectgroup E1-207.

Aalborg University
Department of Computer Science

Aalborg University
Department of Computer Science

TITLE:
The Morphy Project
- Distributing a chess game

THEME:
Modelling

PROJECT PERIOD:
9/9-17/12, 2004

PROJECT GROUP:
E1-207

GROUP MEMBERS:
Jonas Bang Christensen
Jacob Eskildsen
Lars Hornbæk Jensen
Kristian Thorsted Madsen
Niels Birkekjær Platz
Stinus Silassen
Bjørn Mølgård Vester

SUPERVISOR:
Alexandre David

SYNOPSIS:

This report covers the development of a distributed chess engine called Morphy Chess. The goal of the project is to take an existing chess program and modify it to be distributed. The more computing power and memory available, the better the chess program should perform. At the beginning of the report we analyse a number of algorithms for playing chess, including a distribution algorithm named APHID. This is used as the basis idea for the design of a distribution algorithm, and is implemented using the MPI framework. The implementation is in the end evaluated through performance testing.

NUMBER OF COPIES: 9

NUMBER OF PAGES: 109

CONCLUDED: 17/12-2004

First we would like to give thanks our supervisor, Alexandre David, for sound advice and inspiration throughout the project, and to the university for access to use the cluster to test our system. Finally we would like to thank Gerd Behrman for giving us a tour of the cluster room.

The name for this project was inspired by the 19th century chess player "Paul Charles Morphy" (June 22, 1837 — July 10, 1884). Morphy is one of the first recorded chess prodigies in history and considered the best chess player of his time [Dom].

Information about and the latest version of Morphy Chess, including sources, can be found online at:

`http://www.larshj.dk/morphy/`

Jonas Bang Christensen

Jacob Eskildsen

Lars Hornbæk Jensen

Kristian Thorsted Madsen

Niels Birkekjær Platz

Stinus Silassen

Bjørn Mølgård Vester

Contents

Chapter 1

Introduction

Making computers play the game of chess in a skillful manner has gotten a lot of attention over the years and many different solutions to the problem have been presented. Yet the best human chess players can still match the computer.

Through time, two different approaches have been tried to conquer the game of chess: simulation and brute force. The first approach is to simulate how human plays the game of chess and replicate it in the computer. This is done by examining the board, recognising tactical advantage in positions and moves, and by making the computer learn the concepts of sacrificing, pawn positioning, opening moves, forks, discovered attacks etc. The other approach is to use the huge amount of computational power, a computer has, to examine all possible moves, and then choose the one that would lead to a victory for the computer. Even though this has never fully succeeded due to the sheer amount of moves to examine, the approach has been the most successful. Instead of searching all posible moves from a given position, the algorithms search as many moves possible in a given time to give an estimate of what might be the best.

1.1 Problem Formulation

> The goal of this project is to take an existing chess program and modify it to be distributed. The more computing power and memory available, the better the chess program should perform. Load balancing issues will need to be addressed.

Through the analysis we will examine the heuristics used for finding the best move in a given board position in chess, find a method of distribution and a chess program that we can use. These will serve as basis for designing a new distributed system that focuses on finding better moves through the speed and efficiency of distribution. Finally the system will be implemented and tested.

1.2 System Definition

To further examine the task presented in the problem formulation, we give a system definition using *FACTOR* from OOAD [MMMNS01].

1. **Functionality:** Play a game of chess with the computations distributed on more than one computer.

2. **Application domain:** Chess.

3. **Conditions:** A larger number of computers to run the distributed program on.

4. **Technology:** Part of NorduGrid owned by Aalborg University [Nor].

5. **Objects** Those present in a normal game of chess such as the board, pieces etc.

6. **Responsibility:** Present a game of chess to the user and use distribution to compute moves.

Part I

Analysis

Chapter 2

Chess and Game Theory

2.1 Game Theory

“Game theory is a universal language for the unification of the behavioural sciences” [Gin00]. Chess is one of the hot spots within behavioural science and makes game theory a good place to start. We give a short overview of different game theory concepts and important definitions [Ros]. A chess game is a finite game with perfect information and sequential move game. Perfect information means that there is no hidden information (we know all there is to know) and sequential moves means that the players take turns in moving.

Finite games that have perfect information are the simplest of games. This is because players can use a straightforward procedure for predicting outcomes. This procedure or algorithm can be predicted but not necessarily in linear time. This is possible in chess where we can make a tree where nodes are board positions and edges represent moves. Then we need to find a path from the root position to a position where we have won the game. This structure is also called a game tree. A game tree for chess is finite, but it grows exponential for each level, or ply[1], in the tree. A complete game tree would show which moves to take to secure a victory.

A human player will in a rational manner consider each possible series of moves and counter-moves, and choose the one that brings him in the most beneficial position. By making the first move in this series, the player aims to get as close as possible to the optimal path. This method is known as backward induction, as the reasoning works backwards from a wanted outcome to the current problem at hand.

To analyse the search algorithms, we will make use of the following definitions from game theory [Gin00].

Node: A point at which a player takes an action.

[1] In chess, a ply is a turn of one of the players. For instance, after 10 moves in a chess game, 20 ply has been completed.

Root node: The point at which the first action in the game occurs.

Terminal node: Any node that, if reached, ends the game.

Sub game: Any set of nodes and branches descending uniquely from one node.

Payoff: an ordinal utility number assigned to a player at an outcome.

Outcome: an assignment of a set of payoffs, one to each player in the game.

Strategy: an instruction that tells a player which action to take at every node in the tree where a choice is demanded.

Chess is a game that belongs to the category of zero-sum games. These have the property that the total score is preserved as a constant, usually 0, such that if one player obtains x points, the opponent will loose an equivalent amount of points.

Here is first an example of the zero-sum game, Coin Flip, to illustrate the properties of zero-sum games. Secondly we will illustrate how chess is a more complex zero-sum game.

Coin Flip

- Every time a coin is flipped, one player obtains a score of 1 point and at the same time, the opponent obtains a score of -1.
- A win is equal to half the decided flips plus 1.
- A loss is equal the negative of the win situation.
- A draw is the value of half the coin flips on each player.

The Coin Flip game is a good illustration of a simple zero-sum game. When you toss a coin, you get either a head or a tail. Thereby every round has a winner and a looser. If winning gives 1 point and loosing withdraws a point, then the total amount of points in the game will always sum up to zero.

Chess

- For every position, a score is calculated. The score depends on the value of the pieces in this position and the position itself. The reason we can not just use the pieces as value is that if you take a piece with a high value, for example a queen, and afterwards is put in a mate position, then the value of the taken piece will not help us much.
- A win is equal to ∞.

- A loss is equal to $-\infty$
- A draw is 0.

2.2 Chess Theory

Since it is impossible to make a complete game tree for chess in a fair amount of time, game trees for chess are made to look ahead some number of moves. Still, chess has very large game-trees, so some sort of reduction is needed. If we play with the assumption that the opponent will play at his best, there are moves that are too good and some that are too bad to even consider. The fact that very bad moves can be removed does not come as a surprise since we of course do not want to play badly. The fact that we can also eliminate very good moves comes from the assumption that the opponent would never play so badly as to allow us to play those very powerful moves. This means that we want moves of such a quality that we can force the opponent to choose among only bad moves. Should it happen that our assumption of opponent skill does not hold, we can always take advantage of this, even if we had eliminated the powerful move from the game tree. This elimination of "unworthy" moves is called pruning. Elimination of an entire branch of the game tree is also called a "cut-off".

When all "unworthy" nodes/branches have been pruned from the tree, an evaluation of each leaf node is made. Then backward induction is made from the leaf node with the best evaluation and by that, a move is found.

The evaluation of nodes can take several things into account, and is done differently from chess program to chess program. The evaluation could for example look at what pieces are left on the board, where are they are positioned etc.

2.3 Chess Algorithms

In this chapter we have looked at some general theory for games and chess theory. In the following chapters we will look at a selection of the many different algorithms used for computer chess. These can be placed in the following categories, which will also structure the next few chapters in the analysis.

Search Algorithms that create and traverse game trees.

Pruning Algorithms that remove parts of the game tree and thereby reduce its size.

Move ordering Pruning is done when we find "unworthy" nodes as explained earlier. Algorithms in this category attempts to put such nodes, and other interesting nodes, ahead in the search to get pruning and cut-offs done sooner.

Speedup This category can be thought of as a miscellaneous category. Any other kind of algorithm that will help increase the speed of a search or make better searches will be put in here.

Chapter 3

Search Algorithms

In 1928, John von Neumann began his work on "Theory of Games and Economic Behavior" [Bri] in which the cornerstone is the theory of the MiniMax algorithm. The algorithm is for creating and searching a game tree of a game played by two people. In this section we will describe both MiniMax and an algorithm that is only a slight alteration of it, NegaMax. The principles behind these two algorithms are the backbone of many chess engines. These are often enhanced through variations that we will see in the following sections. First, we will describe the principles behind MiniMax and NegaMax and present pseudo code for both.

3.1 MiniMax

The MiniMax algorithm is illustrated in Figure 3.1 on the following page. We show how a game tree where three plies are evaluated. From the beginning we have three possible moves. No matter which we choose, the opponent will have three ways to respond and so on down the depth of the tree.

MiniMax uses the evaluation function to decide which path to take in the game tree. This is done by two functions: one that maximises the value and one that makes sure that if a player has chosen a move, then the opponent will make the move that minimises the value of the move; in effect, a counter move. By shifting between a maximiser and a minimiser, the algorithm finds the path that is the best if the opponent plays his best.

As we explained earlier, it is intractable to make and traverse a complete game tree for a chess game unless it is in the endgame with few pieces. This leaves the problem of when to consider a node as leaf node. That is, when to stop traversing and make a decision. This could be done in several ways, for example with a depth parameter or the occurrence of a final position (a draw or a mate). We implement all these possibilities in the algorithm.

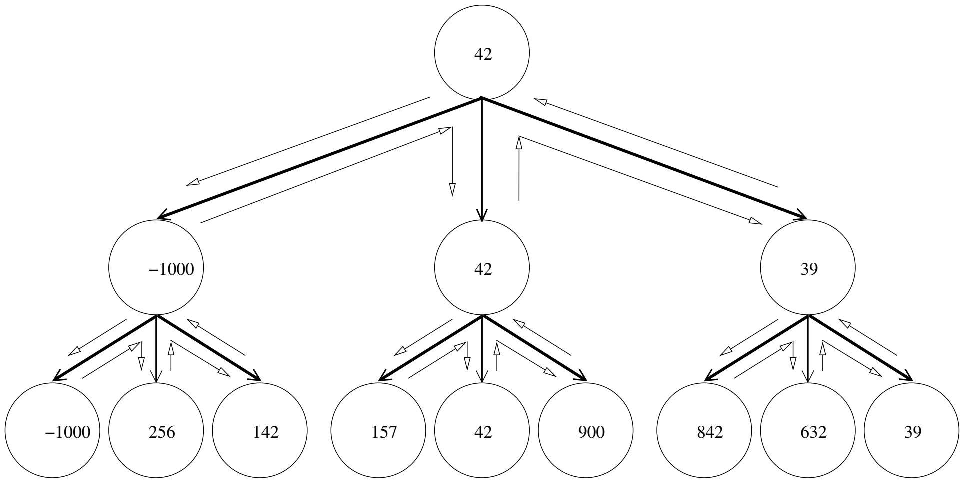


Figure 3.1: Traversal with the MiniMax algorithm of a board game with three possible moves each round.

Pseudo Code for the MINI-MAX Algorithm

Input: Node n is the root of the tree and *depth* is the number of plies to search.

Output: The MiniMax value of the tree.

Other functions:

GET-NEXT-SUCCESSOR(n): Returns the next successor of node n, or NIL if none is left.

HAS-REMAINING-SUCCESSORS(n): Returns TRUE if node n has remaining successors, otherwise FALSE.

EVALUATE(n): Evaluates the node n in accordance to whose turn it is to move.

The pseudo code for the MiniMax algorithm is presented as two functions called MAXIMISE and MINIMISE. The two functions will alternate on calling each other. It is the MAXIMISE function we need to invoke on a tree to find the MiniMax value. For example, if we want to find the MiniMax value on a tree with root node A, and we want to search no further than to depth 5, we invoke MAXIMISE$(\mathrm{A}, 5)$.

```
MAXIMISE(n, depth)
 1  if depth = 0 or HAS-REMAINING-SUCCESSORS(n) = FALSE
 2     then return EVALUATE(n)
 3
 4  best ← −∞
 5
 6  while HAS-REMAINING-SUCCESSORS(n)
 7        do next ← GET-NEXT-SUCCESSOR(n)
 8           value ← MINIMISE(next, depth −1)
 9
10           if value > best
11              then best ← value      ▷ Better move found
12
13  return best
```

```
MINIMISE(n, depth)
 1  if depth = 0 or HAS-REMAINING-SUCCESSORS(n) = FALSE
 2     then return EVALUATE(n)
 3
 4  best ← ∞
 5
 6  while HAS-REMAINING-SUCCESSORS(n)
 7        do next ← GET-NEXT-SUCCESSOR(n)
 8           value ← MAXIMISE(next, depth −1)
 9
10           if value < best
11              then best ← value      ▷ Better move found
12
13  return best
```

Note that the only difference between the two lies in lines 4, 8 and 10.

3.2 NegaMax

The difference between MiniMax and NegaMax is that the latter does not minimise the result at any time, and so the code does not have to check for this within the algorithm. It makes the exact same choices, but negates the result when it is the opponent's turn. This removes the part of the code that distinguishes between the players, which makes it simpler to use.

Pseudo Code for the NegaMax Algorithm

Input: Node n is the root of the tree and *depth* is the number of plies to search.

Output: The MiniMax value of the tree.

Other functions:

GET-NEXT-SUCCESSOR(n): Returns the next successor of node n, or NIL if none is left.

HAS-REMAINING-SUCCESSORS(n): Returns TRUE if node n has remaining successors, otherwise FALSE.

EVALUATE(n): Evaluates the node n in accordance to whose turn it is to move.

The pseudo code for the NegaMax algorithm is presented as a function called NEGA-MAX.

```
NEGA-MAX(n, depth)
if depth = 0 or HAS-REMAINING-SUCCESSORS(n) = FALSE
   then return EVALUATE(n)

best ← −∞

while HAS-REMAINING-SUCCESSORS(n)
      do next ← GET-NEXT-SUCCESSOR(n)
         value ← −NEGA-MAX(next, depth −1)

         if value > best
            then best ← value       ▷ Better move found

return best
```

Chapter 4

Pruning Algorithms

Before we continue with the more advanced search algorithms, we have to look at pruning extensions since most of these algorithms use pruning as a central part in some form or another.

4.1 Alpha-Beta Pruning

In 1958, Allen Newell, John Shaw and Herbert Simon developed Alpha-Beta Pruning [Wal]. The difference in the tree traversal from the original MiniMax, which is described in Section 3.1 on page 9, is that a search window is created. This is done by the values of α and β initialised to $-\infty$ and ∞. Throughout the algorithm, α will always be less than or equal to β.

α is the value of the best known move found so far that the opponent cannot prevent. If the algorithm encounters a value of a branch that is lower than α, the branch is cut off since this subtree is a worse choice of play than the path behind the current α.

β is the value of the worst-case scenario for the opponent. That is, if we find a move with a value larger than β, it is too good for the opponent to allow it. When the algorithm encounters a value of a subtree greater than β, it returns β to indicate that the subtree has been cut off.

This means that only values of x where $\alpha < x < \beta$ are of interest. This interval is called the *search window*.

4.1.1 Pseudo Code for the Alpha-Beta Pruning Algorithm

Input: Node n is the root of the tree, α and β constitutes the search window and *depth* is the number of plies to search to.

Output: The MiniMax value of the tree.

Other functions:

Get-Next-Successor(n): Returns the next successor of the node n, or nil if none is left.

Has-Remaining-Successors(n): Returns true if node n has remaining successors, otherwise false.

Evaluate(n): Evaluates the node n in accordance to whose turn it is to move.

The pseudo code for Alpha-Beta Pruning is presented as a function called Alpha-Beta. We base it on Nega-Max since, as it was explained earlier in Section 3.2 on page 11, it is less complex than the Maximise/Minimise version.

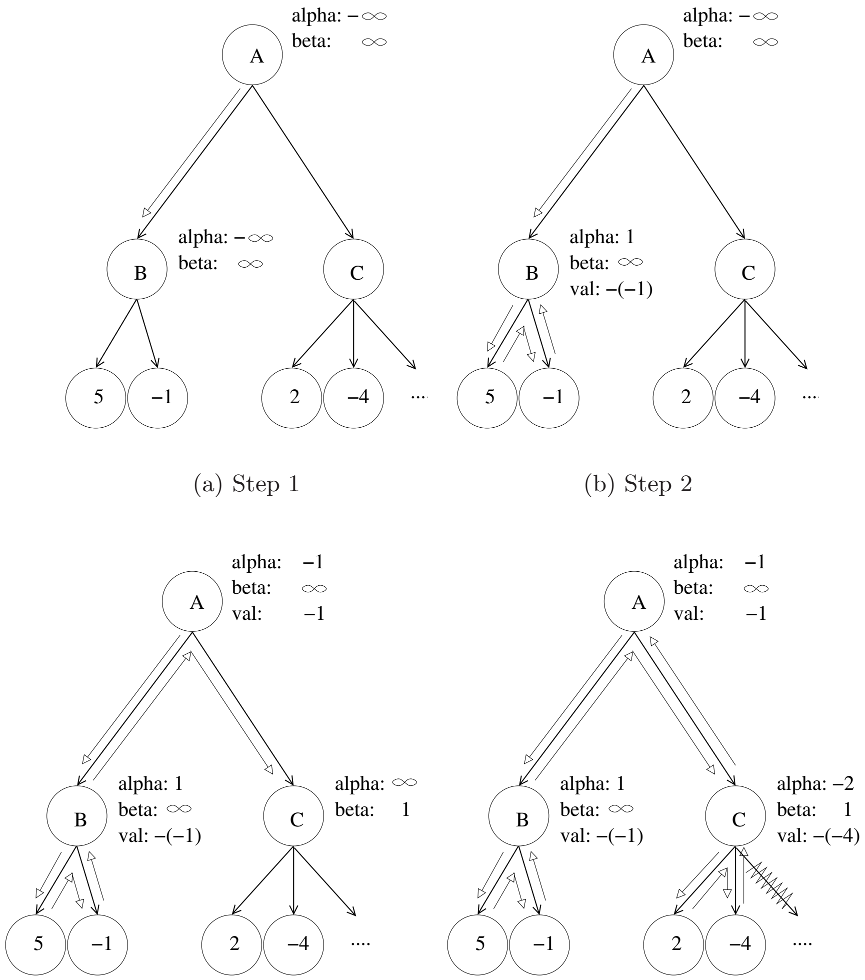
It should be invoked with the parameters α and β set to $-\infty$ and ∞, respectively. For example, if we want to find the MiniMax value of a tree where A is the root, and we want to search to depth 5, we will invoke: Alpha-Beta$(\mathrm{A}, -\infty, \infty, 5)$.

```
Alpha-Beta(n, α, β, depth)
if depth = 0 or Has-Remaining-Successors(n) = false
   then return Evaluate(n)

while Has-Remaining-Successors(n)
      do next ← Get-Next-Successor(n)
         value ← −Alpha-Beta(next, −β, −α, depth −1)

         if value >= β
            then return β          ▷ Cut-off

         if value > α
            then α ← value         ▷ Better move found

return α
```

4.1.2 Example of Use

To explain how the algorithm works, we use the example illustrated in Figure 4.1 on the facing page. It shows a series of game trees, each with the new information gathered by the algorithm as it progresses recursively through the branches. We have chosen to take a fully known game tree to make it more clear how the evaluation part of the algorithm advances.

The first tree (Figure 4.1(a)) shows the game tree we want to use the algorithm on. As mentioned earlier, the two variables α and β have been initialised to $-\infty$ and ∞, hereby ensuring that the first move we check will



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

Figure 4.1: Example of an alpha-beta pruning.

change these values. That is, the function Alpha-Beta$(\mathrm{A}, -\infty, \infty, 2)$ is invoked. Since the node A has successors, we call the algorithm recursively on each of them, but with depth decreased by one and with α and β switched and with opposite signs. This can be seen by viewing α and β for the node B.

The second tree (Figure 4.1(b)) shows the evaluation after the second recursive call where we have reached the wanted depth. Here we see that α has been set to the value 1 since this value represents the best move the opponent can choose if the path to B is chosen.

The third tree (Figure 4.1(c)) shows how the value of the best move is negated at each level in the game tree to represent the changing of the sides.

We now return to node A with the α value updated in line 12 and find another legal move to node C. Therefore we call the algorithm on C with the new α and β values switched.

The fourth tree (Figure 4.1(d)) shows how we examine the first successor of C. This is a leaf node so we just return the negated value. Because this value lies within the search window, we update our α value since this represents a better move for C. This means that we will narrow the search window for all subsequent calls to the remaining children. We now evaluate the second child of C. Since this returns a value greater than β, this means C has found a move stronger than earlier within the subtree of A and therefore the opponent will never allow it. Hereby we can ignore any remaining subtrees or leaves of C and return β as seen in line 9. So by searching to depth two we find that the best move attainable had a MiniMax value of -1.

4.2 Futility Pruning

Futility Pruning [Frab] was developed by Heinz et al. in their development of the program Deep Thought, and is an extension to Alpha-Beta Pruning.

4.2.1 How Futility Pruning Work

This algorithm examins the move one ply before reaching the estimated search depth and if it does not seem that the move will improve the current situation, it prunes away this branch of the tree. This is done by evaluating the board position. Normally, a "quiet" move (one that does not involve captures or checks) in chess does not change the evaluated score of the game by much. We can use this fact to prune away branches which scores within a margin of the current α value. This margin, which is called the *futility margin*, is usually the value of a pawn. This might seem unwise since it could overlook strong hidden tactics, but the idea is that futility pruning will give us enough time for an additional ply in the search. This extra ply should make up for the lost information and give us additional insight.

4.2.2 Extended Futility Pruning and Razoring

Extended Futility Pruning is initiated a ply earlier than the full futility pruning and uses a larger futility margin – usually the value of a rook or more. If the value of the move plus the margin does not exceed the current α value, the ply is decremented by one. An even more aggressive version of this method is called Razoring. This is identical to Extended Futility Pruning with the exception that it starts one ply before Extended Futility Pruning and uses the value of a queen or more as futility margin.

4.2.3 Pseudo Code for Futility Pruning

Input: Node n is the root of the tree, α and β constitutes the search window and *depth* is the number of plies to search to.

Output: The MiniMax value of the tree.

Other functions:

GET-FUTILITY-MARGIN(): For Extended Futility Pruning, the value of a rook is returned. For Razoring, the value of a queen is returned.

GET-FUTILITY-DEPTH(): For Extended Futility Pruning, the target depth minus 2 is returned; for Razoring, minus 3.

EVALUATE(n): Evaluates the node n in accordance to whose turn it is to move.

IS-IN-CHECK(n): Returns true if a king is in check.

HAS-REMAINING-CAPTURES(n): Returns true if there are remaining capture moves.

GET-NEXT-SUCCESSOR(n): Returns the next successor of node n, or NIL if none is left.

HAS-REMAINING-SUCCESSORS(n): Returns TRUE if node n has remaining successors, otherwise FALSE.

A general observation that can be derived from the progression through a full Futility Pruning down through the Extended Futility Pruning and Razoring, is that the longer you get from the last ply, the higher is the chance of doing erroneous pruning due to overlooked tactics.

```
FUTILITY-RAZORING(n, α, β, depth)
if (depth = GET-FUTILITY-DEPTH())
and (IS-IN-CHECK(n) = FALSE)
and (HAS-REMAINING-CAPTURES(n) = FALSE)
and (α > EVALUATE(n) + GET-FUTILITY-MARGIN)
   then depth ← depth −1 ▷ Decrease search depth by one

if depth = 0 or HAS-REMAINING-SUCCESSORS(n) = FALSE
   then return EVALUATE(n)

while HAS-REMAINING-SUCCESSORS(n)
      do next ← GET-NEXT-SUCCESSOR(n)
         value ← −ALPHA-BETA(next, −β, −α, depth −1)

         if value >= β
            then return β          ▷ Cut-off

         if value > α
            then α ← value         ▷ Better move found

return α
```

Chapter 5

Search Algorithms Revisited

Now we have looked more closely at different pruning methods, we are equipped to look at more advanced search algorithms. The ones that we will focus on are NegaScout and MTD(f). We chose those, since they are both widely used. NegaScout is an improvement to the NegaMax algorithm and MTD(f) shows promise for distribution [DKC03, Page 35].

5.1 NegaScout

The NegaScout algorithm [Rei83] strives to minimise the number of nodes that need to be examined by narrowing the Alpha-Beta Pruning search window. It will never examine a node which can be pruned by the normal Alpha-Beta Pruning algorithm. Like the regular Alpha-Beta Pruning, the search window is bounded by two values, α and β. If the successors of a node have been arranged using move ordering algorithms, which we will explore later on in Chapter 6 on page 27, we will most likely find the best choice early on, and all we have to do is to prove that the rest of the moves in a particular branch are inferior. We do this by making a test on the successors using what is called a *null* (or *zero-width*) window. This test quickly determines if a move is worse than the best move so far.

The regular Alpha-Beta Pruning would use the window $(-\beta, -\alpha)$. We will use this window on the first branch to be examined, but the subsequent ones will be called using $(-\alpha - 1, -\alpha)$. This window does not contain any elements and so will always return a value outside the interval (assuming the evaluation function is based on integers). If the negated returned value is less than the current α, we say that the call has *failed low* and actually *is* worse than the current best move. If it is larger than the current β, we say it has *failed high* and we will have to re-search the node with a wider window. We will not repeat the search, though, if the node to be examined is a terminal node or a node with only terminal nodes under it. This is because the algorithm always returns a correct evaluation score in these situations.

If the node is a terminal node, we will just evaluate the node and get the exact value. If the node only has terminal nodes as children, all the values returned were the evaluations of the child nodes and we need not re-search them.

The algorithm will be faster than the normal Alpha-Beta Pruning if the tests succeed often enough to outweigh the overhead of any nodes which must be searched twice.

5.1.1 Pseudo Code for the NegaScout Algorithm

Input: Node n is the root of the tree, α and β constitutes the search window and *depth* is the number of plies to search to.

Output: The MiniMax value of the tree.

Other functions:

HAS-REMAINING-SUCCESSORS(n): Returns TRUE if node n has remaining successors, otherwise FALSE.

EVALUATE(n): Evaluates the node n in accordance to whose turn it is to move.

GET-FIRST-SUCCESSOR(n): Returns the first successor of node n, or NIL if n is a leaf.

GET-NEXT-SUCCESSOR(n): Returns the next successor of node n, or NIL if none is left.

The pseudo code for NegaScout is presented as a function called NEGA-SCOUT. It should be invoked with the parameters α and β set to $-\infty$ and ∞, respectively. For example, if we want to find the MiniMax value of a tree where A is the root, and we only want to search to depth 5, we will invoke: NEGA-SCOUT$(A, -\infty, \infty, 5)$.

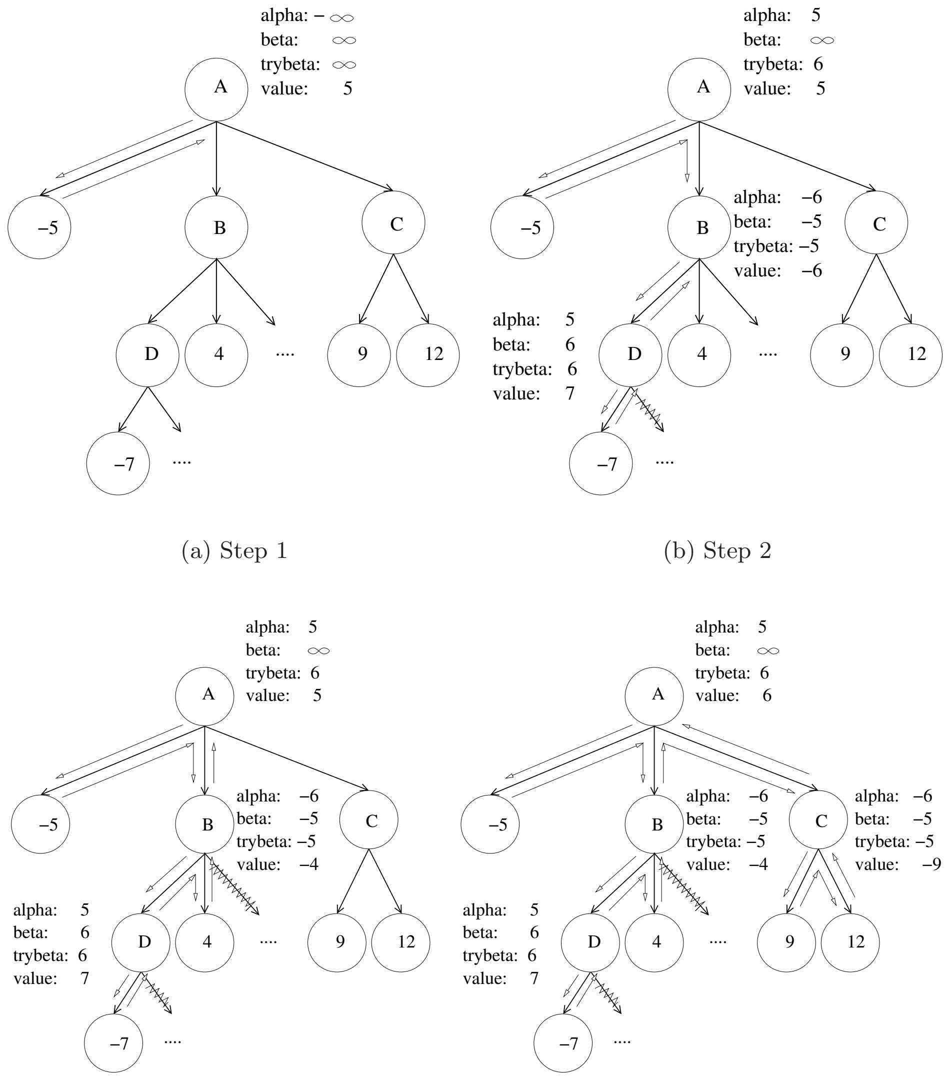
```
NEGA-SCOUT(n, α, β, depth)
if depth = 0 or HAS-REMAINING-SUCCESSORS(n) = FALSE
   then return EVALUATE(n)

trybeta ← β

while HAS-REMAINING-SUCCESSORS(n)
      do next ← GET-NEXT-SUCCESSOR(n)
         value ← −NEGA-SCOUT(next, − trybeta, −α, depth −1)

         if (value > α)
         and (value < β)
         and (next ≠ GET-FIRST-SUCCESSOR(n))
         and (depth > 0)
            then value ← −NEGA-SCOUT(next, −β, − value, depth −1)
                                                        ▷ Re-search

         if value >= β
            then return β          ▷ Cut-off

         if value > α
            then α ← value         ▷ Better move found

         trybeta = α + 1           ▷ Set new null-window

return α
```

5.1.2 Example of Use

To show how the algorithm can prune branches, which would otherwise be searched with the regular ALPHA-BETA algorithm, we will examine the game tree seen in Figure 5.1 on the following page.

The first tree, Figure 5.1(a), shows the game tree with the variables updated from the search of the first (leftmost) successor. This first search creates a narrowed window for the search of B from the *alpha* value 5 found in node A, namely $(5, 6)$.

In the second tree, Figure 5.1(b), the algorithm reaches node D in its search. With its first successor with value -7, we fail high, but do not perform a re-search since we are at a terminal node. But the rest of the branch is cut-off for now and will only be searched if there are no siblings that will force the parent to do a re-search.



(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

Figure 5.1: Example of the NegaScout algorithm.

In the third tree, Figure 5.1(c), the algorithm now carries on searching the other branches of B and makes a normal alpha-beta pruning of B's second branch. This makes sure that B fails low when returning to A, so the cut-off in D proved to be a sound investment in this situation, even though we risked a re-search.

In the fourth tree, Figure 5.1(d), we show that a re-search can be necessary. In the first search of the subtree where C is the root, we have only found better moves. Since no value in C's subtrees changed the α value of C, an unchanged $\alpha = -6$ is returned. The unchanged α will always be greater than the negated α of its parent. In this case we have reached the top level where our return value is also less than β, thereby triggering a re-search. Note that the figure does not show the re-search, since this is done almost the same way as a normal ALPHA-BETA search and will return the exact MiniMax value of the subtree. The re-search is done with a search window from *value* to β. The reason we can search with a window *value* to β is that the former search failed high. This gives us a new lower bound on the MiniMax value. In the NEGA-SCOUT algorithm, each re-search will return the MiniMax value of the searched subtree. The value found in the re-search is then compared to the former best move and if better, α is updated.

5.2 MTD(f)

In the NegaScout algorithm, we saw how a null window could improve the search. A group of algorithms, the so-called *driver* algorithms, take this concept further by using null windows on the root of the tree. One of these algorithms is called MTD(f), which stands for "Memory-enhanced Test Driver". It works by calling Alpha-Beta Pruning a number of times with a null window. Each call returns either an upper or a lower bound on the MiniMax value of the tree. Initially, these bounds are set to $-\infty$ and ∞. When both the upper and the lower bound collide, the MiniMax value is found.

MTD(f) needs a heuristic guess on the MiniMax value to start with (this is the value f). The better the guess is, the faster the bounds will converge.

In order to work efficiently, MTD(f) must be used with some form of memory, which we will look at in Section 6.3 on page 28. This is because of all the re-searches necessary. Even if the initial guess is actually the real MiniMax value, the algorithm still needs to search twice through the tree; first to find the lower bound, second to find the upper bound. Typically, MTD(f) will perform between 5 and 15 passes on a tree before finding the MiniMax value [Pla].

Overall, MTD(f) has been shown by comparisons from empirical experiments to be better than NegaScout [PSPdB].

5.2.1 Pseudo Code for the MTD(f) Algorithm

Input: Node n is the root of the tree, f is a heuristic guess on the MiniMax value, and *depth* is the number of plies to search to.

Output: The MiniMax value of the tree.

The pseudo code for MTD(f) is presented as a function called MTD. For example, if we want to find the MiniMax value of a tree where A is the root, we should search no more than to depth 5, and we guess that our move will improve the new position to the value 12, we will invoke: $\mathrm{MTD}(\mathrm{A}, 12, 5)$.

```
MTD(n, f, depth)
lowerBound ← −∞
upperBound ← ∞
g ← f ▷ Set the initial score to the guess

while lowerBound ≠ upperBound
      do if g = lowerBound
            then β ← g + 1
            else  β ← g

         g ← Alpha-Beta(root, β − 1, β, depth)
         if g < β
            then upperBound ← g
            else  lowerBound ← g

return g
```

5.2.2 Example of Use

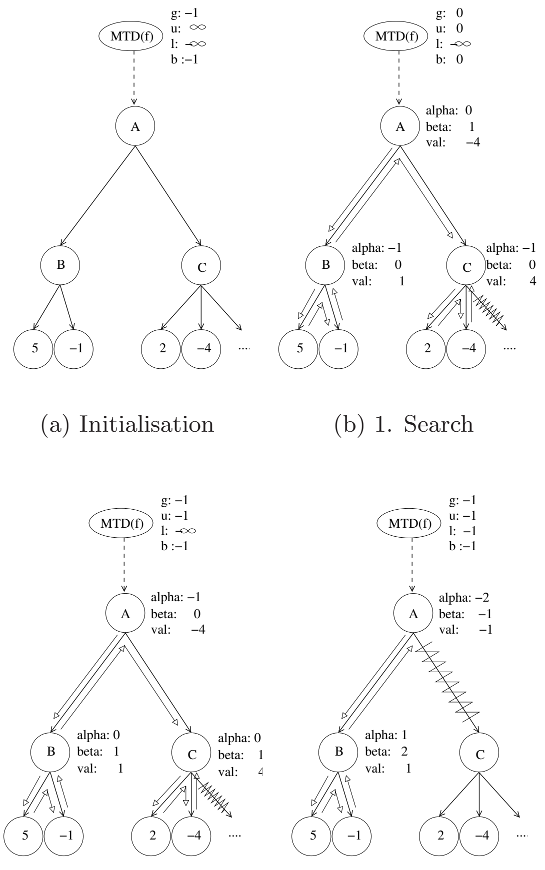
Because the MTD(f) algorithm often performs many searches on the same tree and we do not want our example to be too large or complex, we will illustrate how it works by running it on the same tree we used in Section 4.1.2 on page 14. Note that this will not show why MTD(f) is generally faster than, for example, NegaScout since the speedup comes from the use of transposition tables[1] (not shown) and in much larger trees. The example can be seen in Figure 5.2 on the facing page.

From the example of the Alpha-Beta algorithm, we found the exact MiniMax value to be -1. Let us try using a guess of 1 on this value and see how the bounds converge to the exact value.

On the initial tree illustrated in Figure 5.2(a), we set the values according to a guess of 1. The first search will be made with a null window on this value. The two bounds are initialised to $-\infty$ and ∞ respectively.

The first search can be seen in Figure 5.2(b). We will not go into details about it, as it is a normal Alpha-Beta search. What is interesting is how the window affects the nodes. From the B node, for example, we see that the value -1 (which is the MiniMax value we are looking for) actually causes a cut-off on B. Since the C node is also cut off, we return an unmodified α value of 0 from A.

[1] A table of previously searched nodes; will be explained in Section 6.3 on page 28



(a) Initialisation (b) 1. Search

(c) 2. Search (d) 3. Search

Figure 5.2: Example of the MTD algorithm.

From the first search, using a null window centered around the value 1, the value 0 was returned. This means that the search *failed low* and this gives us an upper bound on the exact MiniMax value. In Figure 5.2(c), we try a second search with a null window around this new bound. Again we see that the B and C nodes are still cut-off after a while, resulting in another *failing low*, now with a value of -1. From the second search, the algorithm reduced the upper bound by 1, thereby closing in on the exact MiniMax value. The lower bound is still $-\infty$.

The third search is shown in Figure 5.2(d). This time the B node is not cut off, resulting in an updated α value in A. Now, we cut node C off and return the updated α value -1. This time the value returned from the search results in a *failing high*, which gives us for the first time a new lower bound. Since the upper bound and the lower bound have now converged to the same value, it means we have found the exact MiniMax value.

Chapter 6

Move-ordering Algorithms

As explained earlier in Section 2.2 on page 7, move ordering is about placing the best moves at the top of the list of moves to be considered at any given depth. The quicker we examine the best moves, the quicker we will be able to make cut-offs which will have a dramatic effect on the size of our search tree. Therefore it will be worth considering using some amount of resources to make a good move ordering. In this section, we will examine three ways of doing this. We will look at Killer Heuristics, History Heuristics and Transposition Tables.

6.1 Killer Heuristics

In Killer Heuristics, we keep track of the moves that at a given ply seem to be causing the most cut-offs, and place these moves closer to the top of the move list [Sch]. We could for example be in a position where the opponent has a good move that we simply must prevent. This could be an example of a move that would become a consistent killer move for a period of time in a game until we have hopefully prevented its play.

6.2 History Heuristics

The concept History Heuristics comes from the idea that the difference in a board position before and after a move is quite small, so a move that was good before might still be good in a new board position later on in the game. This gives the idea of keeping a table of all previously recorded good moves and putting them ahead in the list of moves to consider in a new board position. This increases the possibility of a cut-off. This is not that dissimilar from Killer Heuristics, but now we are move ordering independently of ply.

6.3 Transposition Tables

A transposition table is a "memory" of what we have already done. That is, it will store previously calculated board positions (whether they are good or bad) for later retrieval. Before calculating moves for a new position, we make a look-up in the transposition table.

If the position already exists in the table and it has been searched to the wanted depth, its value is returned. If not, we store the value in the table along with the depth that the value was found at, an indication of whose move it is, and the best move from the position. This gives us two advantages: we can do move ordering from the "best move" entry, and we can avoid duplicate work by using the stored value for a position.

For every position that has to be saved in the transposition table, we need to generate a hash key for indexing into the table. This could be done using so-calles *zobrist keys* [Morc]. Each key is a signature of a board position, typically of 64 bits. Since 64 bits are not enough to enumerate every possible board position, some board positions may evaluate to the same key. In practise, however, this happens so rarely that it can be ignored.

Chapter 7

Speed-up Algorithms

In this chapter, we will look at some of the different steps that can be taken in order to improve the performance of the previously described search algorithms. We present two algorithms that will work as minor add-ons to the search algorithms.

7.1 Iterative Deepening

Search algorithms like MiniMax and NegaScout traverse the tree in a depth-first manner. If the algorithm has a time frame and does not succeed in examining all branches to the specified depth within the time frame, a number of good moves could be missed. Likewise, if all branches have been searched within the time frame, valuable time could be left unused. In this unused time, we could have examined to an even greater depth, foreseeing traps, etc. The problem with only being able to examine a certain number of plies is known as the *horizon effect* [Frab].

Iterative Deepening [Mora] is given a time frame. Within this time frame, it makes sure that all branches are searched and no unused time is left.

The theory behind iterative deepening is to search for the best moves at depth 1. Then increase the search to depth two, then three, etc. This process might seem to actually take much longer than, for example, starting a search for the best move at a depth of three. But if we store information from previous iterations in a transposition table, the overhead should be smaller than the gain. This is mainly because we use this information in move ordering, so that the best moves are searched first and then provide cut-offs before they would occur when searching directly to depth three.

Pseudo Code for Iterative Deepening

Input: Node n is the root of the tree.

Output: The MiniMax value of the tree.

Other functions:

> TIME-LEFT(): Returns FALSE if we have used all the time given to us for searching. Returns TRUE if there is time remaining.

The pseudo code for iterative deepening is presented as a function called ITERATIVE-DEEPENING.

```
ITERATIVE-DEEPENING(n)
value ← 0
depth ← 1

do
    value ← MTD(n, value, depth)
    depth ← depth +1
while TIME-LEFT()

return value
```

The idea behind the algorithm is to continually perform a search on the same tree. Here, we have used the MTD function. As we explained previously in Section 5.2 on page 23, this function needs a heuristic guess on the MiniMax value. Here, a good candidate for the MiniMax value of the current iteration is to use the MiniMax value of the previous iteration.

7.2 Quiescence Search

Quiescence Search [Morb] is a way of dealing with the issue of the horizon effect, described in Section 7.1 on the preceding page. The idea is that when a normal tree search is done, an additional search may begin at interesting leaves. In the game of chess, quiescence search can for example be used at every capture move. This will incorporate the ability to discover advantages of sacrificing pieces that would not normally present themselves until a much higher search depth. Basically, as the name allude to, the idea is to continue an interesting search path until everything is quiescent. It can be extended to take into account a number of other scenarios. One extension could be to search when the king is under attack.

Using Quiescence Search can present problems, though, if not used carefully. If for example we extend the search at some point in the game tree, we will get additional subgames added to the tree. This additional part could potentially also activate Quiescence Searches, thereby prohibiting the original algorithm from finishing its search on the wanted depth in the original game tree.

Chapter 8

Distribution

In the previous chapters, we have looked at different algorithms. In this chapter, we will look at how we can use those in a multiple processor environment. We will present a method of distribution and examine ways of sharing information and technologies for communication. Finally, we will look at possible overheads from distribution.

8.1 Distribution Methods

In a chess engine, there are two main components: the evaluation function and the search algorithm. When wanting to distribute a chess engine, the main focus must be the search algorithm. Traditionally, when distributing a search algorithm, the synchronised parallel algorithms have been the ones which have recieved the most attention, while the asynchronous ones have not been experimented with much.

When comparing synchronised algorithms with asynchronous ones, the main difference is that while synchronous algorithms have a number of synchronisation points throughout their execution, e.g. at the end of each iteration in iterative deepening, their asynchronous counterparts have only the standard time bound deadline as point of synchronisation; all other communication goes on dynamically during execution.

Performance- and code-wise, the main differences are that the synchronous algorithms have a more simplistic and ordered approach to solving the problem, while asynchronous ones exceed them in performance since they do not have to waste as many server cycles in synchronisation overhead.

In our project, we will base the system on the asynchronous algorithm APHID (Asynchronous Parallel Hierarchical Iterative Deepening), [BS99, Chapter 3 and 5] and [Bro97], which is described in the following section, as inspiration for our distribution.

8.2 Asynchronous Parallel Hierarchical Iterative Deepening

The APHID algorithm is asynchronous in nature. The algorithm is built upon alpha-beta pruning and is open for additional modifications at the sequential level of the algorithm.

It is built with a master/slaves architecture. The master searches with iterative deepening by dividing the tree into many distinct pieces on which the slave processes can search independently of each other. The slaves work independently of the master, and if they do not receive any other work to do, they continue to search to deeper levels than required in order to anticipate the next demand from the master – until the master stops them.

The master is responsible for covering the first d' levels of the game tree (given a search depth of d where $d > d'$). This area is continuously iterated through by the master until the slaves has computed the whole structure. While iterating, the master takes care of workload by exempting big jobs and moving nodes from highly loaded processes to processes with free resources.

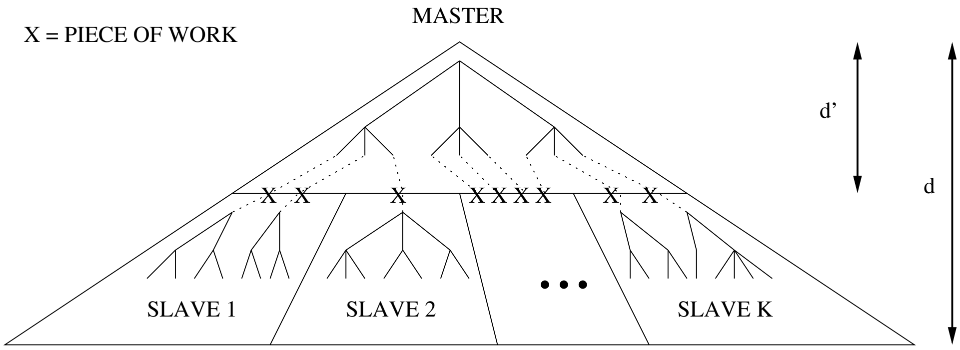


Figure 8.1: Structure of the APHID algorithm.

The author of the APHID thesis [Bro97] has made an implementation library written in C [Bro] freely available for use[1] when distributing a sequential algorithm. In the thesis, he also notes that the implementation gives the possibility to use mid-level processes that can act as both a master and a slave. This is used through a static hierarchy given by the user at start up of the program. He does not go into further detail about this in the thesis.

[1]The library was unavailable during the main part of this project, and therefore we have not examined it.

8.2.1 Operation of the Master in APHID

The master algorithm has the responsibility of distributing work and taking care of workloads. The operation of the master algorithm can be described through the following steps [Bro97, Page 98-99]:

1. Execute a quick search (or pass) of the d' ply tree using an alpha-beta search. The search uses exact or guessed evaluations of the remaining $(d - d')$ ply of the tree at the leaves of the d' ply tree.

2. If we have determined an accurate d ply MiniMax value during the last pass, exit the loop.

3. Compute any required changes to the work lists.

4. Inform slaves of changes to their work lists.

5. Wait for new information from a slave process before going back to the first step.

The d' levels are not kept in memory, as the information from the parallel searches is constantly changing. This is not a problem, as the d' levels is searched through rather quickly. When the master reaches level d', it checks whether $(d - d') < g$ plies (the minimum granularity). That is, if there is less than g plies down from the current depth to the target depth, the master finishes the search itself; otherwise a slave is given the task.

In case the slave processing the node has not yet completed the $(d - d')$ ply search of the node, the algorithm has to guess a value to be able to return something [BS99, Appendix A]. Every time the value of a node is based on a guess, the node is marked as an *uncertain* node. APHID uses the guessed scores together with the true ones to predict where cut-offs are likely to occur, and to decide whether or not to allocate it as work.

8.2.2 Operation of the Slaves in APHID

The content of a slave is usually nothing more than a sequential search algorithm. The APHID specific points of the slave are described by the four steps below:

1. Look in the local copy of APHID table and find the best node to search.

2. Execute the search.

3. Report the result back through the APHID table and update it regarding relevant information from master.

4. Repeat process from step 1.

Finding the best node to search is done from two criteria on the nodes: how deep a search has already been made and how good a position the node has. The first criterion is introduced to ensure an equal level for all the nodes given to a single slave, while the positional criteria is taken in to give the leftmost nodes an advantage as they are less likely to be pruned away due to move ordering.

As it can be rather hard to make a total ordering of all d' levels of master-nodes, an algorithm that calculates an estimated positional rating is used which assignes a priority level to the nodes. Such an algorithm can seen in [Bro97, Page 109].

If a node has a priority of zero, a slave will never choose to search it, because it is not currently a part of the master's tree. When a slave is to select a node, it almost always goes for the node with highest priority until it reaches the limit stated by the master. When the master's required depth is reached, the nodes with the lowest search-depths is extended, with less focus on priority values.

8.2.3 The APHID Table

The APHID table is used for communication between the master and the slave processes. If a leaf node at the d' ply in the search tree is visited by the master for the first time, it is allocated to a slave process. This information is recorded into the APHID table, which is shared between all the servers in the system.

The table contains two sections: one that is writable by the master-process and one that is writable by the slave-processes. The master can read from the whole table, while a slave can only read the part of the table relevant to it.

When delegating a leaf to a slave-process, the master gives an initial MiniMax value. This value will be used for the initial iterative deepening search, and for each ply the slave finishes, the new MiniMax value will be added to the APHID table together with its level, which again will function as input for the master process. The best result of the search is always stored at the top of the entry together with the depth to which it is calculated. Each time the table is written to, the slave and the master will be notified to update the table, thus sharing the information.

8.2.4 Load Balancing

Even though the master algorithm works on sharing the work equally among the slaves, it is next to impossible to determine in advance the amount of work it takes to complete a $(d - d')$ ply search. Because of this, it becomes relevant to move some load from an overburdened slave to a slave with a lighter burden. This is called load balancing.

When detecting the load of a node, the master looks at the number of uncertain nodes on a slave; if it is overworked (many uncertain nodes), one or more of its nodes will be moved to an underworked slave (no uncertain nodes). When selecting nodes, the algorithm chooses the nodes which has been search to the smallest depth, as they create less overhead when moved. When moving a node, we sacrifice the calculations which is already completed at that node and in exchange for the sacrifice we gain the advantage that another processor can cope with entire problem instead. There are two exceptions when selecting: we never select the first uncertain node as the process is likely to be working on it, and we never move a node more than once, which solves the pitfall where a small node is pushed back and forth between processes.

Another load balancing problem that can occur is the case where a single node is significantly larger than the other nodes and therefore is trapped by the sequentiality of a single slave. In this case, the master exempts the node in question. This is done by expanding the d' depth of the master tree with two plies at the current node. By doing that, the large amount of work will be split into smaller pieces, which in turn are delegated to the slaves. The reason for going two plies down is to avoid eventual cut-offs and thereby get a better chance of splitting out the work.

8.2.5 Shadow Table

To avoid duplicate work from different slaves in the distribution, APHID uses so-called shadow tables. These are special localised transposition tables that are used to transmit relevant information about finished work among slaves. One way to prioritise the information for the shadow tables is to prefer nodes that are close to the root node, since they are more likely to be of use for other slaves. To transmit this shadow table, the slaves need to have a communication link with each other. In APHID each slave has a link to the subsequent slave and the last slave has a link to the first. Each slave can initiate a transmission of a shadow table itself, but may never resend information until previously send information has reached itself again, although several slaves may be transmitting shadow tables at the same time. One problem with shadow tables is how to find a good size for it. If we make it too big, it will be to slow to access and send the information. If we make it too small, it will be fast, but the information will not be very useful. The correct size can only be determined through testing.

Chapter 9

Choice of Chess Engine

9.1 Existing Chess Engines

The following table lists the chess engines we have found most interesting, along with a short overview.

Engine	Source code	License	Language	Algorithms	Readability
AI Chess [Fraa]	980KB	Unknown	Java	Moderate	Good
Crafty [Hya]	2043KB	Non-Commercial	C	Many	Moderate
GNU-Chess [KSCea]	937KB	GPL	C	Many	Good
JavaChess [FR]	947KB	GPL	Java	Few	Good
The Genius [Kum]	2160KB	LGPL	C++	Few	Lacking

Table 9.1: Chess Engines.

From the above table we have a quite concise overview of what each algorithm is capable of. Our criteria for choosing an algorithm are the following (in prioritised order):

1. The engine *must* have a license that allows modification and distribution.

2. The code should be comprehensible, readable and well commented.

When examining chess engines, we contacted the author of AI Chess to inquire about a license or perhaps special permission to use his program. We were unable to get a reply from him, so this engine fails the license criteria. While looking at readability we found that The Genius and Crafty where not sufficiently comprehensible to be used as basis for our system. This was concluded both by looking at the code and comments, but also the overall structure of the code.

Lastly, we have GNU Chess and JavaChess left. They both have comprehensible and readable code, both have a limited amount of code and both use the same license. The deciding factor in this situation became the complexity of the implementation. We chose JavaChess because it had a more limited amount of implemented algorithms, making it a more simple basis for distribution.

In the following section we will examine JavaChess more thoroughly.

9.2 The JavaChess Engine

JavaChess comes complete with both an engine and a GUI and as we want to distribute the system to many computers, it would make sense to split the GUI and engine as only one user interface is needed. During the rest of the chapter, the JavaChess program will be described in greater detail and finally we will list the optimisations that we deem necessary before we start up distributing the system.

9.2.1 GUI

The GUI is build out of four main components which can be seen in Figure 9.1 on the facing page.

A: Chessboard.

B: Menus.

C: History of moves.

D: Engine debug output.

The chessboard is a usable graphical representation of an ordinary chessboard in which the player is always white. The game can be configured through the menus that contain the following options:

- **File:** Reset, save and load a game.
- **Edit:** Change player names and edit the board.
- **Engine:** Set the search time, size of transposition table, toggle the permanent brain and loading of an opening book.
- **Help:** Information about JavaChess and help with its use.

The possibility to make a custom board setup exists only in the GUI. The underlying code has not been implemented and is as such an unsupported feature.

The code for the GUI of JavaChess is centered around the two packages: `renderer` and `renderer2d`. When supplied with game information, the class

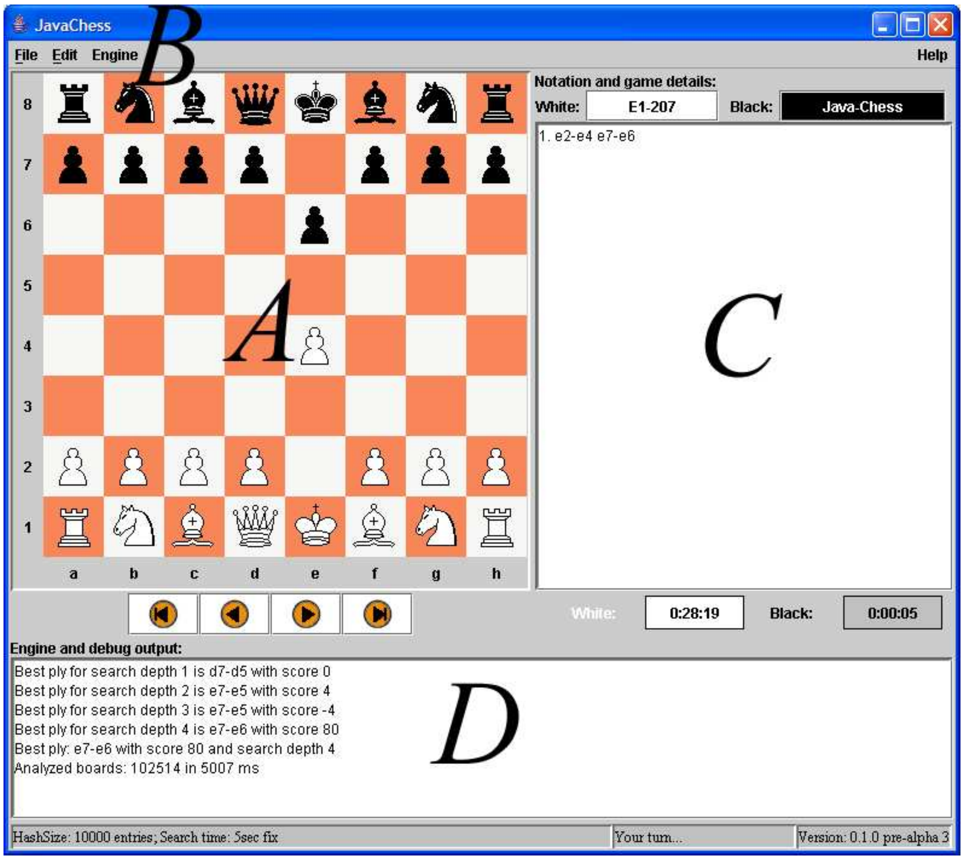


Figure 9.1: Screen shot of JavaChess.

`ChessBoardRenderer2d` generates an interface, which is then displayed to the user. The GUI has three main areas: graphic representation, control of engine and communication with engine.

All communication between GUI and engine happens over the class `GameController`. This is the main component, as it both instantiates the engine and the GUI, and thereby uses interfaces from both.

The GUI starts up the computation of a single move by calling `computerPly`(...) at an instance of the class `GameController`. During the computation, an instance of `GameController` acts as listener to the engine and every time a notification from the engine occurs, the `GameController` uses the GUI interface to update it with the new information.

Overall the GUI is rather independent with the `GameController` lying between it and the engine, thus making it easy to isolate the engine.

9.2.2 The Engine

Basically, the engine part does not consist of many classes since much of the needed information is stored elsewhere. The key information class being that of BitBoards, which is a data structure used for representation of a board position (see for example [Frab]). The core of the engine consists of three parts:

1. PlyGenerator
2. BitBoardAnalyzerImpl
3. ChessEngineImpl

PlyGenerator Used by the tree search algorithm to generate all the legal moves for a specific player on a specified bitboard. This is done by adding the legal moves for each piece type one at a time resulting in a sorted array of objects of type `Ply`.

BitBoardAnalyzerImpl This handles the analysis of a game position stored as a bitboard. First, the evaluation function checks if any player is in check and adds or subtracts point accordingly, depending on which player is to move. Also, early checks are punished in the evaluation by cutting the check points by a quarter:

```
if (getGame().getNumberOfPlies() < 12) {
    checkValue = (short) (checkValue / 4);
}
```

Next, the evaluation function examines the material value of the pieces on the board, their positions, and returns a weighted score:

```
return (short) ((short) 2 * positionalValue +
          (short) 7 * materialValue + checkValue);
```

ChessEngineImpl

The `ChessEngineImpl` is where the tree search is handled. That is, the central role of the class is in figuring out which move to play.

If we are in an opening book situation, the engine knows the best ply immediately; it is found simply by following the instructions from the opening book, which is written in *PGN* format (Portable Game Notation). Otherwise, we need to perform the actual search. Before this is started, we check to see if the *Permanent Brain* was used. This is simply a feature that allows the computer to perform a preliminary search while the opponent's clock is running. If this feature was used, we ask it for the branch that matches the move the user has made. This, of course, is only a fraction of the actual computations made by the permanent brain, but this is something we essentially get for free. Then the algorithm starts the actual search either from scratch or the branch we got from the permanent brain.

This is a MiniMax search (see Section 3.1 on page 9) with Alpha-Beta Pruning (see Section 4.1 on page 13) and with the use of a transposition table (see Section 6.3 on page 28). The algorithm terminates with the best ply found so far when its allocated time has been used. The allocated time is either a fixed or average amount of time.

It should be noted that the implementation of zobrist keys in JavaChess, contrary to the one mentioned in Section 6.3 on page 28, does not take into account the legality of *en passant*[1] or *castling*[2].

9.2.3 Optimisations

It is clear that the actual algorithms used in the tree search are limited in comparison to those we have considered. Some being enhancements that could be applied without distribution and some being especially suitable for distribution. This applies both to the tree search, pruning and alterations on the move ordering. The evaluation function is also something which should be considered for possible tuning. Tactics, such as piece threats and protection, are not considered and would enhance the detail level of the board analysis. Also the evaluation of a board uses a lot of hard coded values; this naturally leads to the question of whether or not these are the optimal values.

[1] A special capture made by a pawn. It occurs when a pawn on the fourth or fifth rank captures an enemy pawn on an adjacent file which moved 2 squares on the previous move.

[2] Interchanging the positions of the king and a rook.

Part II

Design

Chapter 10

Requirements

10.1 Purpose

As stated in the problem formulation in Section 1.1 on page 1, the purpose of this project is to create a distributed chess engine with foundation in an already existing non-distributed implementation. In Section 8.2 on page 32 we noted that there existed an implementation library for APHID, although it will not be used[1]. Instead, we will implement our own distributed algorithm based on APHID but with modifications that will be presented through the design.

10.2 Quality Goals

In this section, we will set up a list of criteria that we will use as guidelines throughout the design. These criteria will also serve as quality goals which will be evaluated in later sections as to whether or not they have been fulfilled. The list of criteria is taken from the OOAD method [LMS00].

In Table 10.1 on the following page, we have outlined the design criteria along with a classification of their importance. The classifications are as follows:

- *Very important:* A criterion essential for our fulfilment of the project.
- *Important:* A criterion that covers essential features or has issues which need to be resolved in the development process.
- *Less important:* A criterion that covers non-essential features.
- *Irrelevant* A criterion which will not recieve much focus, if any.

[1]The reason we will not use the APHID library is partly because it was unavailable at the time of the design and implementation, and partly because we wanted to address certain issues concerning distribution ourselves rather than relying on an already implemented system.

- *Trivially fulfilled* Most issues concerning this criterion will be resolved beforehand, and should not need much additional focus.

Criteria	Very important	Important	Less important	Irrelevant	Trivially fulfilled
1. Usability					X
2. Security				X	
3. Efficiency	X				
4. Correctness		X			
5. Reliability	X				
6. Maintainability		X			
7. Testability		X			
8. Flexibility			X		
9. Comprehensibility		X			
10. Re-usability				X	
11. Portability					X
12. Interoperability			X		

Table 10.1: Priority of the design criteria.

We will now give an assessment for each of these criteria and explain how they help fulfil our main goal:

1. **Usability:** As we have chosen an existing chess program, a graphical user interface already exists. Since our focus is on distribution and not user interfaces and since the existing user interface is acceptable, we set this criterion to trivially fulfilled.

2. **Security:** There is no information in the system that needs to be kept confidential from other people. The data generated when the system is running may be very interesting but need not be kept secret. The entire system will have the security of the systems and network that the distribution will be executed on. Furthermore, the server has no privileged access to any parts outside the system. Therefore, this issue is irrelevant.

3. **Efficiency:**

   The search algorithm is a major part of a chess engine and its strength relies heavily on which algorithm is used. Therefore one way to increase the strength of the engine is to choose a stronger search algorithm. This, however, is not an issue we want to explore, since our main purpose is not to create a strong chess engine but to create an efficient distribution model.

   Also, in a distributed system, a part of the strength lies in the amount of resources available. Therefore, a major concern in such a system is to make sure each resource is used to its fullest extent. This can be achieved by eliminating bottlenecks and reducing overheads in for example communication or redundant data. This is a central issue for the overall efficiency of a distributed system, and thus very important to our distribution model.

4. **Correctness:**

   Our only clear demand for the project is to achieve distribution of a chess program, and we do not have a customer who has put further demands for the functionality. However, we do have certain overall goals with this project that needs to be fulfilled. Because of this, the criterion will be set to important.

5. **Reliability:**

   As a consequence of the distribution of the chess program, we introduce several new possibilities for crashes internally in the system since several computers are involved. Like the original program, we cannot do anything about a crash on the computer where the user is situated. It is, however, very important to be able to handle and recover from crashes at any other point in the distributed system.

6. **Maintainability:**

   The original system was chosen because, among other reasons, it had source code that was both easily readable and maintainable. It is important to us to keep the code maintainable and possibly improve upon the code in that area, so that if we decide to put more work in this engine later on it will be more manageable.

7. **Testability:**

   Since the main goal is to improve the system, we need to have a base to evaluate the improvements. Furthermore, to ensure correct code and the possibility of testing of it continuously throughout the implementation, it is important to us to have good testability.

8. **Flexibility:**

   The system should be designed with independent modules to provide the possibility of replacing certain aspects of the system without the need of major recoding. This gives us the possibility to change different parts of the system, such as search algorithms and user interface.

9. **Comprehensibility:**

   Generally, since a lot of people are involved in this project, it is important that the overall design of the system is comprehensible. This goes for both the people involved in the development and evaluation and those that might be interested in further development after the conclusion of the project.

10. **Reusability:**

    We consider reusability to be irrelevant as the purpose of our project is to examine the strength of distribution; not to make a general purpose game engine.

11. **Portability:**

    As our platform of choice is Java, portability is trivially achieved for our main engine. We have no intention of moving the system to other technical platforms in which Java is not supported. We do, however, also use MPI (see Section 11.2), which is a framework written in C. Nevertheless, the system will be able to run on several platforms and operating systems, and is thus quite portable as well.

12. **Interoperability:**

    It would be a nice feature to include the option for the system to play against other systems. But as it is a feature that is not necessary to fulfil our main goal, we rate it as less important.

Evaluation of the Criteria In this section, we have set some criteria that we will strive to obtain. The next task is then to find a way to evaluate these criteria as to whether or not they have been fulfilled.

We will use the above mentioned criteria as guidelines throughout the design process. The different design decisions will be commented on how they adhere to the these criteria. In the end of the design part, we will evaluate how we fulfil the criteria on a more overall basis.

Chapter 11

Technical Platform

11.1 Equipment

A distributed program can in theory exist perfectly well on just two computers. However, to test issues such as scalability and performance, we need more processors. For solving our task, we have gotten permission to use part of the NORDU-grid [Nor]. A separate machine is needed for the user interface that accesses the others.

11.2 Basic Software

Since the chosen engine is written in Java, this will be the platform used for the development of the system. The chosen version will be Java 2 Platform, Standard Edition version 1.4.2 [Sunb]. Documentation of the implementation will be done with the use of JavaDoc [Sunc] comments. The communication between server and client will be handled by Java RMI [RMI] and the internal communication on the grid by the Message Passing Interface (MPI) standard [MPI]. To see a comparison of MPI versus other communication technologies (see [EE99]). For testing purposes, we will use JUnit [Jun]; this will help ensure correctness of our code.

Chapter 12

Data Flow

Throughout this section, we will give an overview of the data flow in a normal game. The description of the flow is divided into two parts: a part for describing the flow under the initialisation of the game and a part for describing the flow during a game.

12.1 Design of Initialising Data Flow

As described in Section 8.2 on page 32, the distribution algorithm only involves one master that has contact to a number of slaves. This may cause problems: If we have many resources in the form of servers, we may take too many slaves so that we get problems with the communication or not use all servers that we have available. Like the APHID game library [Bro] had support for mid-level processes, we have introduced a similar solution through the use of what will be called submasters. Contrary to the game library, which can only initialise a static hierarchy from user input, we have developed an algorithm for the initialisation of the server tree. We present this algorithm in the following.

The Distribution Structure In Figure 12.1 on the following page, two sets of arrows are presented: the black, dotted arrows symbolise the flow of the game, and the white arrows represent communication. When the chess program is started, we need to initialise the distribution structure before a game can begin. This initialisation is only run once and the structure will only need to be modified if a server crashes and a recovery algorithm takes over. The figure shows the initialising steps.

First, the program starts a client and sends a request for a master to the server pool, which then initiates an election. In our case the election will be started by using MPI to connect to the server with the lowest ranking in the server pool. The election could be modified to contact all servers and find the one with the greatest communication threshold. This, however, would

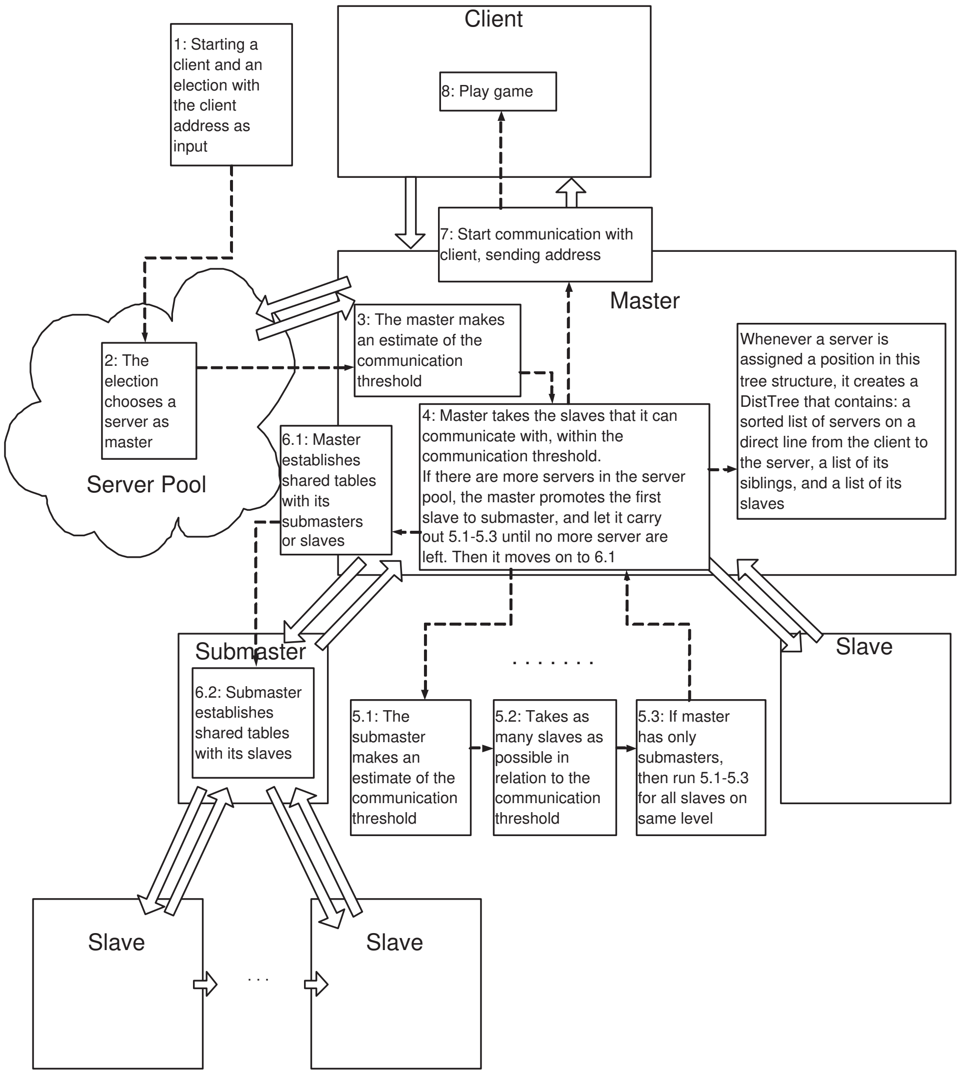


Figure 12.1: Initial setup of the distributed game structure.

result in parsing messages from all to all. The best approach would be to find the concrete distance between all servers on the grid, their connection type, their bandwidth and so on. This is not realistic for us to do; therefore a simple election is used instead. Since we are not going to use the entire grid for testing, the algorithm will give us servers located in the same node on the grid. This means that they are situated near each other. This election of a master server is done in the second step in the figure.

Thirdly, this master server estimates how many slaves it can handle efficiently. This is the former described method, where we send messages to all other servers. As mentioned before, this is a bad idea if everyone sends messages to everyone, so here it is just the master server that examines its communication threshold. The problem still remains that this is a temporary picture of how the communication threshold looks. Since it is on a grid, the fluctuations will be minor. Therefore, we obtain a quick election that is relatively simple, but still quite precise.

Fourthly, the master server runs an algorithm that creates a tree of servers for the distribution of work. This process is described in steps 5.1 to 5.3. During this creation of the structure of a distribution tree, each server needs to perform certain tasks. In step 6.1 and 6.2, the servers establish shared tables with their children. These tables are used to distribute the results throughout the tree structure. The shared tables are used as described in Section 8.2.3 on page 34 and help the system obtain robustness through redundancy of data by keeping them on each server. The tree of servers is incapsulated in a structure called the *DistTree* that holds a list of servers on the path from the client to the specific server, a list of the specific children of the server and a list of its siblings. Furthermore, each sibling has a pointer to the next one in its list, forming a circular message parsing structure. This is used to share data between the siblings, as is described in Section 8.2.5 on page 35 about shadow tables.

Shadow Tables To avoid memory shortage, we must make sure the shadow tables have a fixed size so they do not grow too large to become a bottlenect in the system. As can be seen in Figure 12.2, a shadow table is divided into two parts: a shadow part and a transposition part. The transposition part acts as a normal transposition table and is also used for move ordering. The transposition part has a greater fixed size since it is situated on the server all of the time. The shadow part is smaller so that we save bandwidth and avoid too much communication overhead.

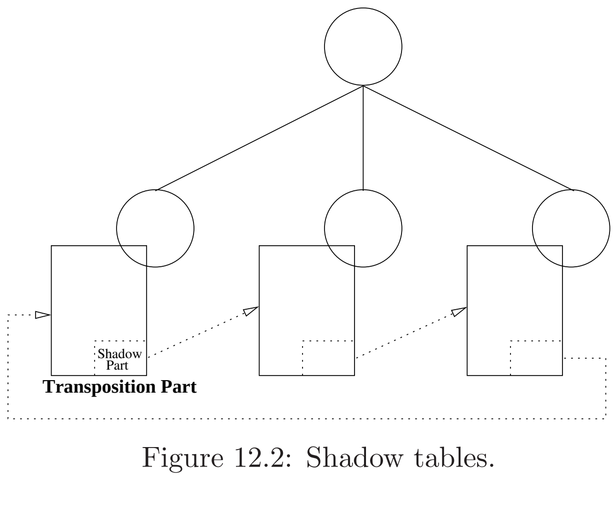


Figure 12.2: Shadow tables.

A sibling knows the existance all of its siblings, but will only send data to the one which is specified in the DistTree. If a sibling crashes, the list of siblings will be recreated without the one which creashed. This is made possible by a recovery feature.

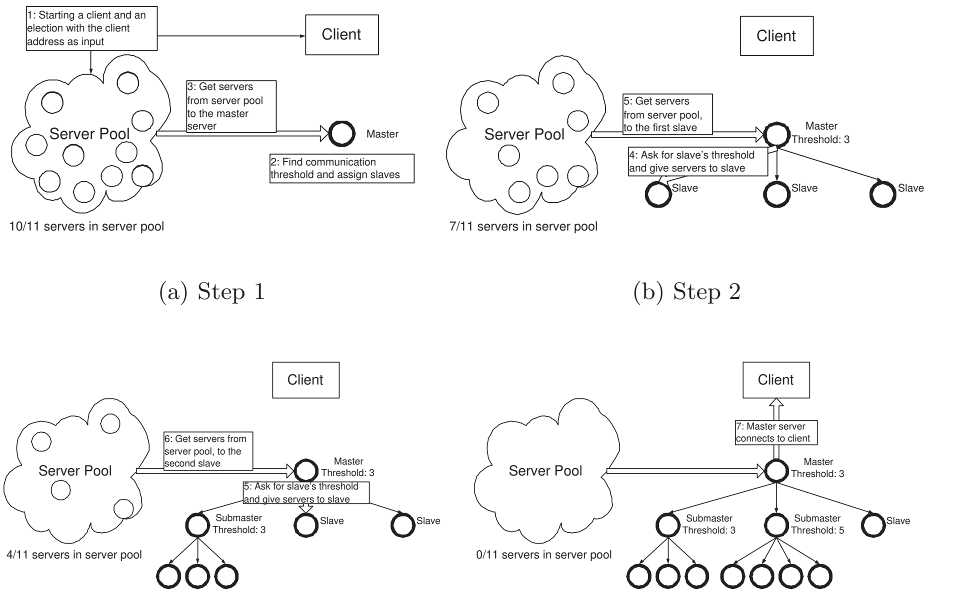
The flow of communication is modified so that when a server sends its shadow part, it only sends a fraction of the shadow table. What is sent is decided by only viewing what has been changed or added since last shadow part has been sent. If this exceeds the size of the shadow part, we evaluate what is important. This is done in relation to how close the node is to the top of the assigned root node. These are the nodes that have been searched to the greatest depth and therefore are the positions that could cause the greatest cut-offs. Another reason to take these nodes is that the closer the node is to the root node, the greater is the chance that it will also be found in a sub-tree of positions of one of the siblings. The shadow table also has a table of what nodes have been received from its siblings. If we know that other siblings are working on the same node, we update the shadow table no matter how close it is to the root node.

When the shadow part has been created, it is sent to the first sibling. The sibling then merges the received shadow part with its transposition part. If the node already exists in the transposition part, it makes a comparison of the new and the old node. The one that is searched to the greatest depth will be inserted into the table. If the node is not in the table, it is inserted.

Building the Server Tree In Figure 12.3 on the facing page, we see an example of how we create the structure of the distributed server tree. This figure is explained below.

(a) As mentioned earlier, the first step in the initialisation is to start a client and to start an election within the server pool that contains all the servers available to the game (this was what we described in Figure 12.1). In this example, we only have 10 servers in the server pool when the election has taken place and a master server has been elected. After this, the master server uses a new algorithm to estimate its communication threshold, so it does not take on more slaves than it can keep communication alive at all times without falling behind.

(b) After this, the master server requests servers from the server pool. These servers are setup as slaves and are assigned to the master. To use any remaining resources, we transform some of the slaves to submasters, which themselves control new slaves. We try to obtain this by letting the master server examine whether more servers exist in the server pool. If this is the case, the master server asks its first slave for its threshold so that it can upgrade this slave to submaster by giving the slave its own slaves to handle. Still it must be noted that only the master server can communicate with the server pool. This communi-



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

Figure 12.3: Creation of distributed server tree structure.

cation is also done through MPI. To avoid using mutual exclusions, and in doing so, bypassing the need to address the issues they pose, only the master server can assign slaves to any other server. By first establishing contact to the client when the structure of the distribution pattern has been created, we achieve that the system does not use computation power for this when playing chess, on the cost of longer start-up time.

(c) If the master server could give its first slave enough slaves to satiate its communication threshold, it moves on to the next slave. In this example, only four servers are left in the server pool.

(d) The second slave has a threshold, which makes it capable of keeping communication with five slaves, but since only four are left in the server pool, only four can be given to the slave. After assigning the slaves, the slave is upgraded to a submaster. In this case, not all slaves or submasters directly under the master server have been assigned slaves. If they had, and if there were more servers left in the server pool, the master would ask its first submaster for the threshold of its first slave. The master server would then give the first submaster the wished number of slaves from the server pool.

This process will be run over and over until all servers in the server pool are used. By keeping up this process, we can use all servers and thereby keep them busy. By using this pattern of server distribution, we obtain the largest amount of slaves while still making sure that we avoid communication bottlenecks. The master server will then close the connection to the server pool, create a new connection to the client with its own address and place a notice that it is ready to take on new work.

12.2 Data Flow of the Game

The flow of data during a game can be seen in Figure 12.4 on page 58. This flow is divided into eleven steps.

The first step in the game flow is for the client to play the game until the board position is no longer in the opening book. This is done to obtain a fast response in the beginning so that we save time for later calculations when the moves are found by the distributed part of the system. This will make the system more efficient.

When we get out of the opening book, the new board position will be sent to the master server with a time frame so that when time is up, a result will be returned. To obtain a more efficient system, the return of the result will not stop the slaves from searching. This part follows the algorithm described in Section 8.2 on page 32.

When the master receives a work assignment, it lets the engine search from the given position and to a certain depth. The search tree created from this search will have leaves that become the root nodes in new work assignments, which are distributed to the servers under the master. If these are submasters, this process is repeated in each submaster. Otherwise, a full search is commenced. By doing this, we lose some efficiency because a submaster is only doing a very shallow searching of new moves. This is the price we pay to avoid communication bottlenecks. However, since slaves are given to the submaster with respect to their communication threshold, we make sure they are used to their fullest.

In the figure, steps 6.1 to 6.4 and 7.1 to 7.3 all show the usual steps in a search: first search to a certain depth; then update the shadow table so that as few board positions as possible are examined more than once.

After this, the parent of the server gets its shared table updated. The last part in this process is for the parent to update any changed search windows to its other children, or to report that certain work assignments no longer are in the master's search tree because they have been pruned away.

The recovery part is not described in the figure, but it works as follows: when a slave does not get a response from its parent, it calls a function in the `Recovery` class, which uses MPI and an election algorithm to pick the slave to take its parent's place. The shared table then helps reconstruct the lost parent, since data is duplicated.

When the time is up, a result is returned to the client. Every master and submaster will keep searching their given assignments to the specific depth to look for updates and to update any work assignments in the rest of the tree structure. The structure is also used to survey the workload for the children of the server so that it can redistribute the work and use their resources to a maximum.

Another way to control workload is by using exemptions. If a work assignment takes a long time to search through, it might not do much good to redistribute it to another slave. This is because other slave will be overworked immediately upon recieving the new work which would require new load balancing. This is avoided by exempting slaves from a big work assignment, and searching it a ply deeper before distribution. This way, a large work assignment will be split into smaller work assignments.

When a user move is entered into the client, it is examined if it is within the endgame. If not, the process is repeated until it is the case. Then the client takes over again and plays the game to the end. Again we do so to make sure that we can use more time in the middle-game.

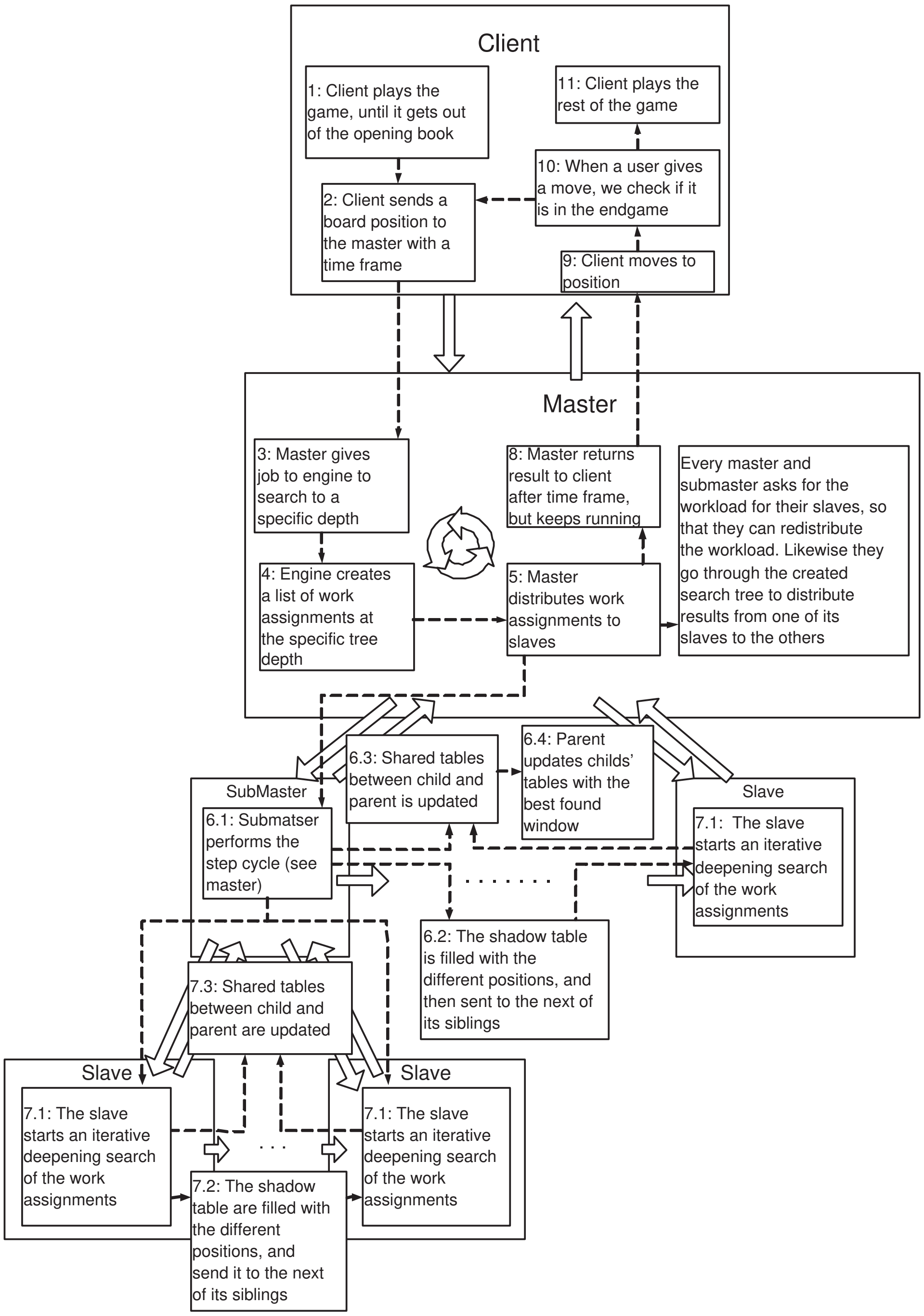


Figure 12.4: Flow of the game.

Chapter 13

Architecture

This chapter will outline the architecture of our system. The overall components will be presented and their connections will be described.

13.1 Component Architecture

An illustration of the component architecture can be seen in Figure 13.1 on the following page. The architecture is divided into a client part and a server part. While the client represents the GUI and the users control of the game, the server component represent the distributed engine that ensures distribution of computations on a set of servers. Each major component is divided into a number of sub-components. These will be described in the following chapter.

13.1.1 Standards

The system uses the SUN's Java code conventions [Suna]. If any changes or additions are to be made in the existing GUI code, they will follow the existing standard established by JavaChess.

13.1.2 Process Architecture

As mentioned, the system will consist of a client and a number of servers. The communication in the system is limited to communication between the client and a master server, and internally between the servers. To keep this communication structured, all communication between the components have been placed in dedicated communication modules. This way all communication will be uniform, and the code will be more modular and comprehensible.

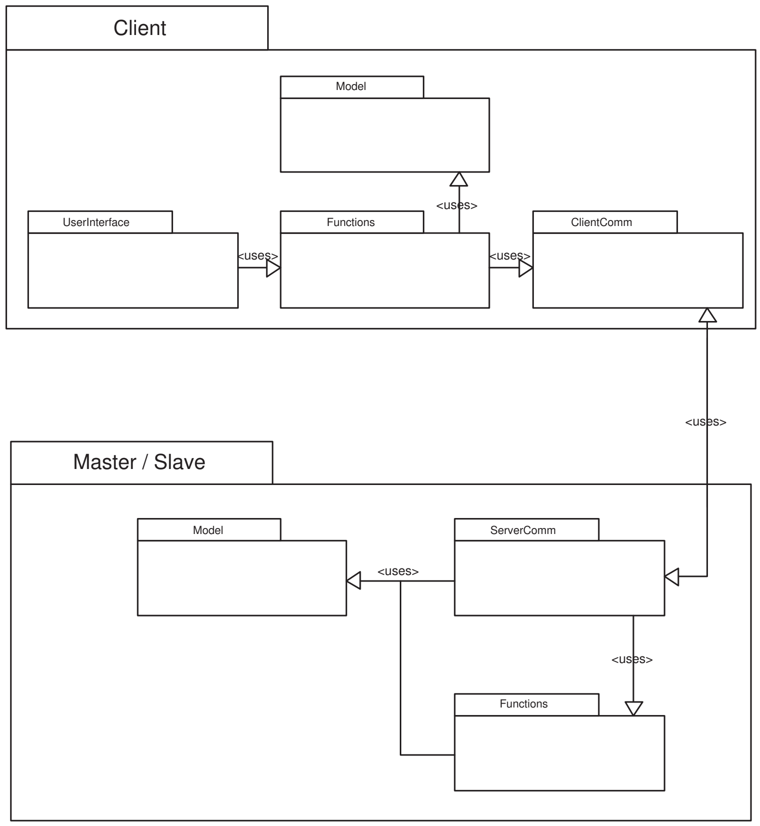


Figure 13.1: The component architecture.

Chapter 14

The Components

In the following, the structure of the client and server components will be described through two class diagrams. In addition, each of the classes in the components have a description and a figure giving an overview of its structure. In these figures we see the names for variables and methods, the types for variables and input, and finally the return types of the methods. We use the symbols +, − and # to indicate visibility as seen below:

+ public

− private

package

In the diagrams, we will not use the *protected* access modifier. In addition to this, set and get methods are implied and will not be described in the text or shown on the figures.

14.1 The Client Component

The client component shown in Figure 14.1 on the following page consists of the five sub-components: `UserInterface`, `ChessModel`, `DistChess`, `ClientComm` and `GameController`. `DistChess` is the main component, and handles the initialization of the program. `ClientComm` handles all communication with the server cluster and `GameController` handles the overall flow of the game. As part of the original system, the `UserInterface` and `ChessModel` have already been described in Section 16.2 on page 76. The three remaining components will be described in the following sections.

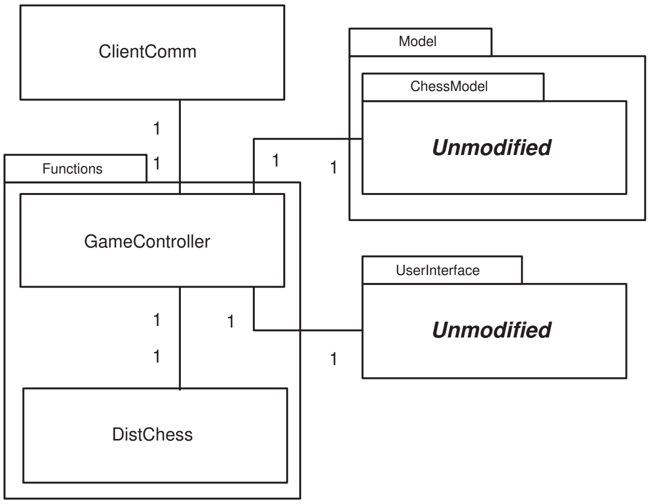


Figure 14.1: Class diagram for the client component.

14.1.1 Classes

ClientComm	
Variables:	
- Address	serverName
- RMI	masterServer
Methods:	
# Move	requestMove(Board board, boolean white, long time)
+ void	setOption(Option option)

Figure 14.2: Class description for ClientComm

ClientComm

Purpose Handles communication between client and master together with `ServerComm`, and acts as communication interface for the client application.

Attributes **masterServer** is the RMI connection to `ServerComm`, **serverName** is a host containing the interface for the server.

Methods When this object is created, it will initiate a RMI connection to the master. The method `requestMove`(...) is called from the class `gameController` and is used for requesting a computer move from the server, given a board position and a time frame. The last method `setOption`(...) changes options for the search algorithm.

DistChess

DistChess	
Variables:	
- GameController	gameController
- Bitboard	chessBoard
- Address	masterAddress
- int	masterPort
Methods:	
+ void	main(String[ ] args)

Figure 14.3: Class description for DistChess

Purpose Instantiates the game by creating instances of `GameController` and `ClientComm`. It also creates the GUI through the classes in UserInterface.

Attributes The most important ones are **gameController** that holds the state of the current game and **chessBoard** that contains the state of the current chessboard. It also contains **masterAddress** and **masterPort** that are the setup for the RMI connection to the master.

Methods The method `main`(...) starts up the game by creating and assigning objects to the attributes and starting up the GUI.

GameController

Purpose This class handles the game of chess. This includes the timer, whose turn it is to move etc. It is the internal controller class in the client that, besides controlling the game, also handles initialisation. This is also a class from the existing code and needs to be modified as such (see Section 16.2 on page 76). Since this is from the existing code, any modifications will be handled in the implementation. Therefore attributes and methods for this class will not be further described.

14.2 The Server Component

The server component shown in Figure 14.4 on the following page is structured into `Communication`, `Function` and `Model` components. Like the client component, the communication of the server component goes through one class, `ServerComm`. This is shown in Figure 14.4 on the following page. As in the client component, we also have components that we do not describe, since they are from existing code. In this case, these are `ChessModel` and `Engine`. The additional components and classes will be presented below.

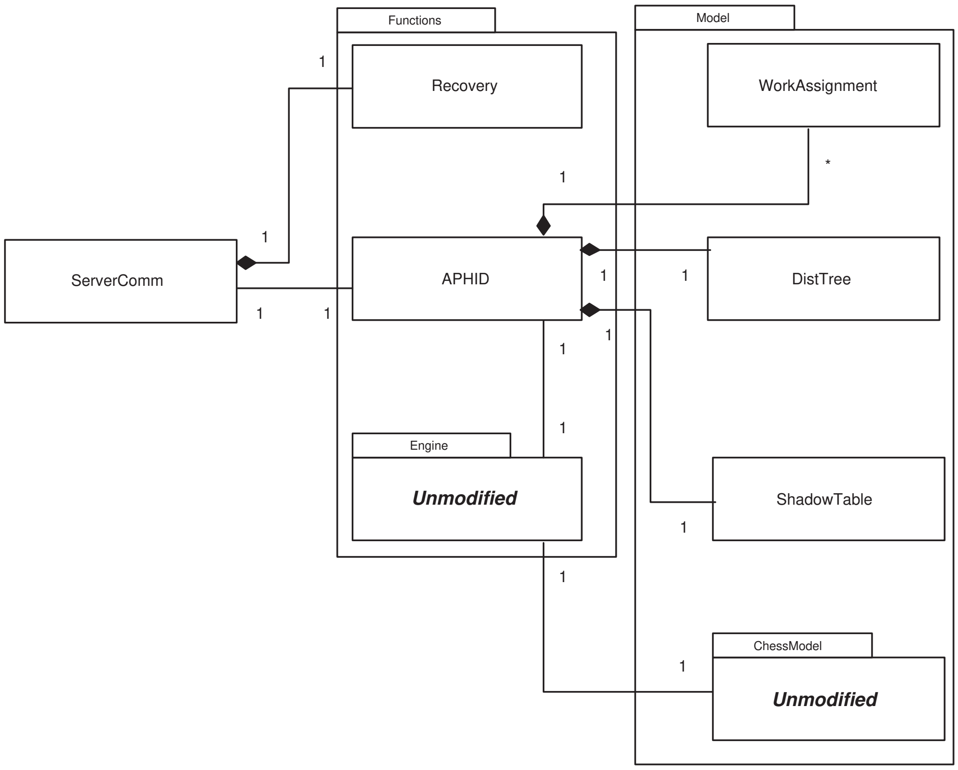


Figure 14.4: Class diagram for the server component.

The `Communication` component handles the communication between servers as well as the communication to and from the client. This is done in the class `ServerComm`.

The `Function` component consists of the functional classes used to distribute and perform the computation. The main class in this component is the `APHID` class. This handles the discussed distribution algorithm described in Chapter 12 on page 51, and uses classes from the `Model` component. In addition to this, we also have a `Recovery` class to handle the situations when a server crashes.

The `Model` component contains the classes needed to represent the data used locally to represent game trees, computations and the distribution tree. The component consists mainly of the classes `DistTree`, `SharedTable`, `ShadowTable`, `WorkAssignment` and the original model component from JavaChess.

The entire server component can exist in one of three different modes: master, submaster and slave. For each mode, the server has different responsibilities. The functionality for all these modes exists in all servers, so they are ready to take on new responsibilities if needed.

14.2.1 Classes

ServerComm

ServerComm	
Variables:	
- RMI	rmiConnect
- MPI	mpiConnect
- Address	hostName
- int	rank
- int	threshold
Methods:	
- void	messageHandler()
# void	sendMessage(Message message)
+ int	commThresholdCalc()
+ void	connectRMI()
+ void	initTree()

Figure 14.5: Class description for ServerComm

Purpose The server communication class is the communication interface from servers to other servers on the network and to the client in case of the master server. The class encapsulates communication internally among the different server processes. While the client/server communication is done by RMI, the communication between the servers is done by passing messages from the model across the network using MPI. Because the class is the only link from the server to the rest of the network, it also keeps track of information about the current network status like the current communication threshold and workload on a specific node. Finally, it also handles the creation of the distribution tree, as described in Section 12.1 on page 51.

Attributes In this class we have both RMI and MPI connections that links to the client and other servers respectively. The **hostName** attribute is the address for the host on which the RMI server is situated. Likewise we need a unique attribute to identify the different servers. This is handled by the MPI-system, which returns a **rank** for each server. The **rank** is also used in recovery to find a new master or submaster. The attribute **threshold** indicates how many servers a given master or submaster can have as slaves.

Methods This class contains four different main functionalities. First, it handles sending and reception of messages between servers through the methods `messageHandler`(...) and `sendMessage`(...). The method `commThresHoldCalc`(...) is used to find the communication threshold. The creation of the distribution tree is handled by `initTree`(...). Finally, the class contains functionality to connect to a remote client by the method `connectRMI`(...).

APHID

APHID	
Variables:	
- List	workAssignments
- List	sharedTable
- ShadowTable	shadowTable
Methods:	
+ Move	requestMove()
# void	distributeWork()
# void	checkRedistribution()
# void	checkExemptions()
# void	exempt()
# void	markZeroPriority(WorkAssignment wa)
# void	startSendingShadowMessage()

Figure 14.6: Class description for APHID

Purpose Contains the implementation of our distributed search-algorithm. Besides administering the distribution of work assignments, creating exemptions and doing actual computations, the class also administers shadow tables. The class can work in two modes; either as a slave or a submaster.

Attributes In the initialisation the different variables **workAssignment**, **sharedTable** and **shadowTable** are created.

Methods The method `requestMove`(...) is used to initialise the game and start a search on the master that searches to a certain depth. Once this depth is reached, the method `distributeWork`(...) is invoked to distribute the leafs as work to slaves. Afterwards, the method `checkRedistribution`(...) is used to redistribute work assignments from overworked servers to other servers. Likewise `checkExemptions`(...) and `exempt`(...) are used to check if a **workAssignment** is too large and split it into smaller **workAssignment**s if it is the case. The `markZeroPriority`(...) is used to mark **workAssignments** as pruned and therefore not worth searching on this iteration. The last method `startSendingShadowMessage`(...) is used to send the **shadowTable** to the nearest sibling.

DistTree

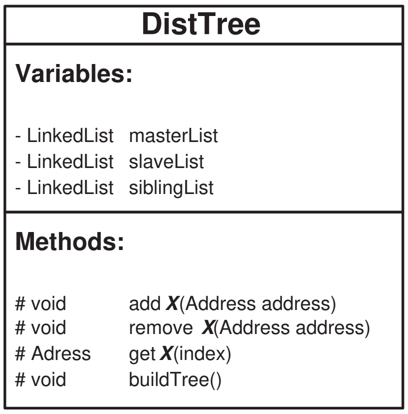


Figure 14.7: Class description for DistTree

Purpose The DistTree class is a container for the relevant structure of the distribution tree. This means that the DistTree holds information needed for recovery, sharing of shadow tables and assignment of work to slaves.

Attributes The class has three linked lists: **masterList**, **slaveList** and **siblingList**. The **masterList** contains an id for all submasters on a direct path between itself and the client. This is used when recovery is needed and servers need to be moved to new positions in the DistTree. The **siblingList** is used for sharing of the **ShadowTables** and **slaveList** for the distribution of work. The **masterList** need to be sorted, since we need to know in which order they appear towards the master; the other lists need not be sorted.

Methods As can be seen in Figure 14.7, we have the methods `add`(...), `remove`(...) and `get`(...). These exist for all three lists. We also have a method `buildTree`(...) that creates the distribution tree from the threshold value.

WorkAssignment

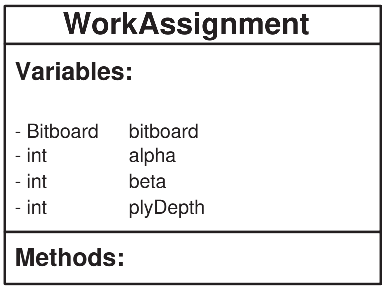


Figure 14.8: Class description for WorkAssignment

Purpose The `WorkAssignment` class is a container for values regarding a single piece of work given to a server. It holds all the information about what has been searched from a specific node in the game tree.

Attributes In `WorkAssignment` we have four attributes. A bitboard, a search window and a search depth.

Methods There are only set and get methods in this class and as such, none are listed in the figure.

SharedTable

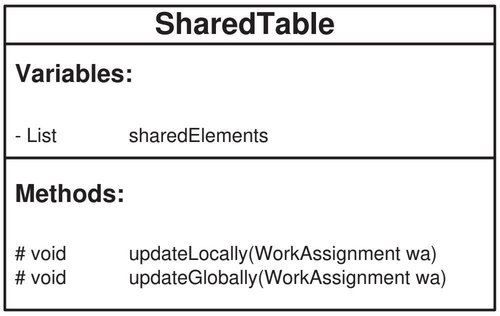


Figure 14.9: Class description for SharedTable

Purpose The `SharedTable` class is the table that is shared between two servers as described in Section 8.2.3 on page 34.

Attributes The attribute **sharedElements** is a list of `WorkAssignment`s.

Methods The method `updateLocally`(...) is called to update an element in the local table, while the method `updateGlobally`(...) sends the update to the server which the information is shared with and uses `updateLocally`(...) to update the local table.

ShadowTable

ShadowTable

Variables:

- HashMap transpositionPart
- HashMap shadowPart

Methods:

void sendShadowPart()
void merge(HashMap recivedShadow)

Figure 14.10: Class description for ShadowTable

Purpose The `ShadowTable` class is an extended hashmap. It is used like the shadow tables described in Section 8.2.5 on page 35.

Attributes The class has two hashmaps that constitutes the previously mentioned shadow and transposition parts; see Section 8.2.5.

Methods We have two methods. One to merge a received shadow part and one to select what to send to the next sibling and uses the class `ServerComm` to send it to the next sibling.

Recovery

Recovery

Variables:

Methods:

void promoteNode()
void denoteNode()
void moveNode()
void startElection()

Figure 14.11: Class description for Recovery

Purpose The Recovery class handles the cases where a failure has occurred. The main purpose of this class is to handle the case where the master of a server fails and one of its slaves or submasters has to take the role of their previous master or submaster.

Attributes The class recovery exclusively uses information stored elsewhere, so there are no attributes.

Methods The class holds methods for manipulating the nodes in the DistTree when one node has failed. In addition to this the class holds the method `startElection`(...) that initiates an election among it self and its siblings to elect a new master/submaster.

Chapter 15

Design Recommendations

15.1 Usefulness of the System

In this section, we will look at the design compared with the design criteria. We will evaluate each point as how the overall design adheres to the individual criterion, except those listed as *trivially fulfilled* or *irrelevant*. In some cases further investigation will be needed to examine a criterion.

Efficiency To obtain efficiency, we have designed a form of distribution that uses all resources available and where the searches do not need to be synchronized. This is done to avoid bottlenecks within communications. This had the cost of not using every server as a slave. To see if these methods succeed in obtaining our goal for efficiency we need to examine this further in the test, Part IV on page 89. Calculating the communication threshold also help to both avoid too many slaves and to secure a minimal number of submasters. The shadow tables are created, so that the most important elements in the transposition tables becomes shared. This also increases efficiency by creating more cut-offs. This again can not be confirmed before the tests. By making the search algorithm exchangeable, we get opportunity to have it replaced and the tests can show if a new search algorithm will improve the capabilities of the system.

Correctness Since we have created a model for distribution of the original program, addressed load balancing issues through redistribution and exemptions, the design fulfils the correctness criteria in accordance to the problem formulation.

Reliability Reliability is obtained through the shared tables and the methods for recovery. The shared tables gives us redundancy of data needed for recovery from crashes. Also, by keeping a record of siblings, ancestors and children, we make it possible for a child to take the place of a parent with a minimal loss of work.

Maintainability We have kept a modular design which helps increase maintainability, but it is very dependent on the actual implementation, for example comments. This needs to be kept in mind throughout the implementation to make sure it is achieved.

Testability We will examine this in the Test part.

Flexibility By having a modular design we have achieved some flexibility.

Comprehensibility We achieve comprehensibility by having a modular design. Each part have a clearly defined purpose and this gives us the required comprehensibility.

Interoperability The entire distributed engine is designed separately and it would be possible to replace the client with an interface that would make it possible to play against other engines.

Further details on how we plan to implements the different features of the program can be found in Appendix A.

Part III

Implementation

Chapter 16

Implementation

In this chapter, the implementation of Morphy Chess is described through five parts. First, it is described what changes has been done to the design. Secondly a description is given of what has been changed in the original JavaChess code. Thirdly, we will describe the parts of the system that were not yet implemented at the time of the project conclusion. Then, parts of the implementation will be described, and a summary of the implementation is given. Lastly, we will look at what have been done in comparison to the design criteria.

16.1 Changes to the Design

Some classes from the design have been split into multiple classes, but from an architectural point of view, not much has been changed from our ideas in the design to the final implementation. The client and server component are implemented as described. Though one change has been made to this. We have added a model component which consists of code shared by the server and client. The elements in the model component are the different objects for representing board, pieces, ply, etc.

A configuration file has been added to ease the activation or deactivation of different features in the system. This can for example be what search algorithm is to be used, whether or not the permanent brain should be used etc.

Also, in the code there is no difference between submasters and masters anymore, apart from the fact that the RMI part of the master server is active. The jobs that they perform are the same. For the master server we have added the client as a parent. Then if any servers need to send a message to their parents, it is checked whether or not the parent is the client. If so, the message is send through RMI, otherwise through MPI.

The initialisation of distribution tree has been changed to only use a single calculated threshold value. This will be elaborated in Section 16.3.3.

Finally, the use of endgame and opening books have been removed. This has been done because we do not concern ourselves with a chess clock and are more interested in looking at what moves will be generated by our algorithm and not what is in endgame and opening books.

16.2 Changes to the Original Code

In this Section, we will review the existing code, to determine what modifications are necessary to implement in the code, before it will be usable.

The first modification we will examine is the separation of the original code into the 3 modules, that we presented in Section 16.1 on the preceding page:

Client Packages: actions, dialogs, engine, menu, pgn, renderer, renderer2d, test and timer. **Classes:** DistChess, GameController and GameState.

Model Packages: bitboard, board, game, notation, piece, ply, position and util.

Server Packages: engine.

In the `Client` module, the most important part was to modify it such that the local engine would not be used for requesting moves, but only for testing the validity of legal moves. All move requests were redirected to go through the client communication part instead. The refactoring of the model and server code was fairly easy and required little more than reference updates.

For the server part, we also made a modification that made it possible to change the used search algorithm quickly. The actual algorithm of the original engine was saved unmodified for testing purposes, while new search algorithms were created, to make them more easily distributed.

Lastly, the original game did not know the concept of draw[1] and was modified to recognise those situations.

16.3 Implemented Parts

In the following, we will describe a few selected topics from the implementation. These are the "Message Passing System", "communication threshold calculation", "DistTree creation" and "move calculation".

[1]A draw situation occurs in one of 3 situations: If a situation has occurred in which there are no combinations of legal moves that can produce a checkmate, also called lack of material, if however, there exists no moves for one player, without moving the king into a check, it is called stalemate. If the same board position (including legality of castling and en passant) has occurred 3 times during a game. If 50 moves have occurred, without a pawn move or a capture.

16.3.1 Message Passing System

Part of the main criteria, for a successful implementation of a distributed system, is a powerful communication protocol. In this section, we will explain how communication between servers in our system is carried out using message passing. We will not go into details about the communication between client and master.

As mentioned earlier, the communication between the servers in our system is done through MPI. This system uses message objects to communicate.

All message objects in the system inherit from the class `Message`. Below we will go through the purpose of some of the message objects used in our system.

AssignRoleMessage: Used in the creation of the distribution tree to assign a role to a node.

ConfigMessage: Used to propagate configurations throughout the system.

GameStatusMessage: Informs about end of turn etc.

ShadowTableUpdateMessage: Contains information for updating shadow tables.

SharedTableUpdateMessage: Update message for shared tables.

ErrorMessage: Messages for error handling.

When a server receives a message, the method `dispatchMessage`(...) in class `ServerMPIImpl` is called. This method looks at the type of the received message and calls specific handlers for each message type. In case of the `ErrorMessage` type, it is simply passed on to the parent using the `sendMessage`(...) method in `MPIComm`. When the message reaches the master server, it is once again passed on to the parent, which in this case is the client. This time, a construct in the `sendMessage`(...) method ensures it is given to the RMI handler, `ServerRMIImpl`. This is done by adding the latter as an observer to the `MPIComm` object using Javas's built-in `Observer` pattern. The `dispatchMessage`(...) method in the `ServerRMIImpl` class and the `ServerMPIImpl` class are similar. The method handles each kind of message received. In case of an `ErrorMessage`, an entry is added to the error log file and an error description is sent to the client using RMI.

16.3.2 Communication Threshold Calculation

The calculation of the communication threshold for a server is based on ping times to all other servers in the system. This is shown in Equation (16.1). Here, Server_Count is the number of servers in the system, Ping_Size is the size of a ping message in bytes, and Ping_Time is the number of milliseconds from when the server transmits a ping message to a server to when it receives a ping message back from that server. The resulting value gives the total amount of bytes per second.

$$\mathrm{Bps} = \sum_{n=1}^{\mathrm{Server\_Count}-1} \frac{\mathrm{Ping\_Size} \cdot 2 \cdot 1000}{\mathrm{Ping\_Time}_n} \tag{16.1}$$

In Equation (16.2), the average bitrate is calculated in bits per second.

$$\mathrm{Avg\_Bitrate} = \frac{\mathrm{Bps} \cdot 8}{\mathrm{Server\_Count}} \tag{16.2}$$

Lastly, Equation (16.3) calculates the amount of slaves the server can communicate with. This is done by dividing the average bitrate by the assumed bitrate, Bits_Per_Child, in bits per second that a single slave will require. This assumed value is hardcoded in the program.

$$\mathrm{Threshold} = \frac{\mathrm{Avg\_Bitrate}}{\mathrm{Bits\_Per\_Child}} \tag{16.3}$$

These calculations are done by the method `computeCommunication-Treshold(...)` in class `MPIComm`.

16.3.3 DistTree Initialisation

When the server program is started, a `AssignRoleMessage` class message is sent to the server with rank zero. This informs the server that it is the master and gives it a list of all the servers available for distribution. In the design, we only calculate one threshold value. We assume that this value holds true for all the servers in the system. This makes sense in our case since we are working on a cluster which can be assumed to have a homogeneous internal communication network.

The next thing we do is to calculate a model for the distribution tree. That is, the server uses the threshold value to make a *sketch* of the distribution tree. Firstly, the sketch shows if any slaves should be added directly to the master server. Secondly, it shows how many servers are needed to maximise the work load for a given submaster. It also shows if a submaster is to create its own submasters from the servers given to it. When the master sends a series of servers to a submaster, the submaster repeats the creation of a sketch, but now from the already calculated threshold, and the servers given to it. This way, we get a tree structure that is symmetric on every level, except possibly on the last. If a child needs to be given any slaves, an `AssignRoleMessage` object will be sent to it with the information about its available servers. If no slaves are to be passed on to a child, it is given the role of a slave.

16.3.4 Move Calculation

In this section, we will look at what happens when the client requests a move. The parts of pseudo code we present here are the simplified versions of the corresponding Java code in Morphy Chess.

Client-Server Control The server needs a method responsible for starting a search when the client requests a move, cleaning up after the search and returning the best move found. This method is called RequestMove and is invoked from the client through RMI. The parameters are *position* that specifies the board position to search on, and *time*, which is the time given to search before a resulting move must be returned to the client.

```
RequestMove(position : Node, time : Integer)
 1  if IsBrainOn() ▷ End possible search on human move
 2     then serverMPI .DispatchMessage(EndOfTurn)
 3
 4  α : Integer ← −∞          ▷ Use full window for root node
 5  β : Integer ← ∞           ▷ Use full window for root node
 6  work : SharedTableElement(position, α, β) ▷ Create new work
 7  serverMPI .DispatchMessage(work)
 8  Sleep(time)               ▷ wait until time is up
 9  result ← serverMPI .GetBestMove()
10  serverMPI .DispatchMessage(EndOfTurn) ▷ End search
11
12  if IsBrainOn()
13     then work.DoMove(result);
14          serverMPI .DispatchMessage(work)
15
16  return result
```

In this method, we start in line 1 and 2 by sending an EndOfTurn message to all servers in case the *permanent brain* is enabled (see 9.2.2 on page 40). The EndOfTurn message will be explained in the end of this paragraph. We can not use the best move found from the search made in the opponent's turn, but in the process of finding it, the transposition tables get filled with information, useful for further searches.

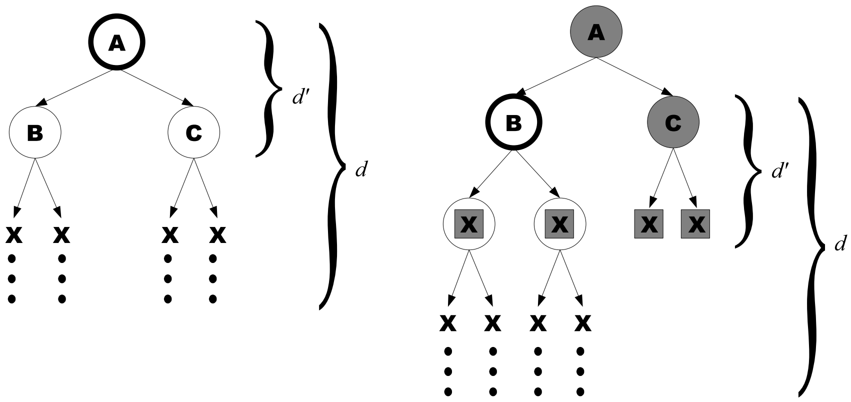
Once this is done, a new shared table element, *work*, is created (line 6). This encapsulates the position we want to search, and states that it should be searched with a full window. A message with the new element is then given to the MPI part of the master server (line 7).

After the message has been dispatched, it waits the specified amount of time after which the result is read from a variable shared with the search part (line 8 and 9).

We then send an EndOfTurn to clean up after the search (line 10) and if the *permanent brain* is enabled, the newly found move is performed on the board position and the updated work message is dispatched (line 12 to 14). Lastly, the move is returned to the client (line 16).

The EndOfTurn message will reach all servers and has the effect of cleaning up the transposition and shadow tables and removing all entries in the shared tables. Here, cleaning a table means removing all entries which has not been touched since the last clean-up.

To see why it is necessary to remove all work from the slaves between turns, we will examine Figure 16.1 that shows a game tree, which we want to search. The fact that the tree is binary is only to simplify the figure. The subtrees below the bottom nodes, all shown as **x**'s, are represented by dots, since they are of no interest in this context. The d variable is the depth we want to search to and d' is the search horizon of the master (the depth to which the master should search before assigning work to the slaves).



(a) Node A is the current root of the tree to search.

(b) Node B is the current root of the tree to search.

Figure 16.1: How a game tree changes after a move.

The first figure (Figure 16.1(a)) shows the game tree at the end of a search. The search horizon has been set to two ($d' = 2$) and the bottom nodes at depth d' have been assigned to a number of slaves. The time frame has been reached and the best move is found (from node A to B).

The second figure (Figure 16.1(b)) shows what happens when the next move has been made. The grey **x**'s need to be removed. First of all, we need not search the old alternative moves, so the slaves working on any nodes in the branch from C need to have them removed. Secondly, since the search horizon of the master (d') is fixed, we need to move this horizon one step forward. It now includes nodes that were previously assigned to the slaves. Therefore we need to remove these nodes from the slaves as well, even though they are working on potential strong moves. Not all information will be lost though, since we save all information given to the master from the slaves about these nodes. Only the local information in the slaves will not be useful.

Server Control Each server has a thread that runs the ServerControl method. Its main task is to continuously pick the most important element in the shared table, search until the search depth of the task is increased by 1 and update it in the shared table.

```
ServerControl()
 1  result : Move
 2  while not IsEndOfGame()
 3          do work : SharedTableElement ← GetWork()
 4             if not work = null ▷ Otherwise the shared table is empty
 5                then position : Node ← work.GetPosition()
 6                     α : Integer ← work .GetAlpha()
 7                     β : Integer ← work .GetBeta()
 8                     depth : Integer ← work .GetDepth() + 1
 9                     ClearUncertainList()
10                     ▷ Make passes until all leaf nodes are exact
11                     repeat ▷ Get the best move from the engine
12                            result ← GetBestPly(position, depth, α, β)
13                        until IsUncertainListEmpty()
                                              Or IsSearchAborted()
14                     if not IsSearchAborted()
15                        then ▷ The iteration is complete for the depth
16                             work .SetScore(result .GetScore())
17                             result .SetDepth(depth)
18                             SetBestMove(result)
19                             Update(work)
20
21             if IsSearchAborted() ▷ Ready for next iteration
22                then SetAbortSearch(false)
23
24             if IsEndOfTurn() ▷ Ready for next turn
25                then SetEndOfTurn(false)
```

While the game is running, we try to get the most important work assignments from the shared table through the GetWork method (line 3). In line 5 to 8, the variables are initialised, and will be passed on to the search method.

In line 9, the uncertain list is cleared to remove any information from previous searches. After this, we continuously make passes over the position until it is certain or the search will be aborted. We also need to abort the search if a bad bounds notification is received or the priority of the position we are working on is changed to zero. Furthermore, if we do not distribute any work throughout the search (i.e. this is a slave or the depth is smaller than the search horizon or minimum granularity), only one pass will be made. Otherwise there is distributed work which has not yet been searched to the needed depth.

Once the search is completed, the entry in the shared table is updated (line 16 to 19). If the search was aborted, the aborted flag is reset, since we are now ready for a new search (line 21 to 21).

Search Algorithms The search algorithms presented in Chapter 5 on page 19 need a new structure for handling the distribution. Besides this, only few modifications have been made. However, we have also implemented transposition and shadow tables which in themselves require a lot of changes. These could be implemented in a sequential algorithm in the same way, and therefore we will only examine the new structure which handles the distribution. In the alpha-beta search, it needs to be inserted just after the loop through all possible moves. The structure can be seen on the next page in SEARCH.

We start by checking if the current server is a master (i.e. there are servers at our disposal) and if the distribution depth has been reached (line 2 to 4). Otherwise the search is continued as the rest of the search algorithm specifies. We will then try to get a corresponding element from the shared table. If a such does not exist, we need to assign the board position resulting from the move currently examined, to a slave. This is done through the DISTRIBUTEWORK method, which distributes work in a round robin manner. The board also needs to be analysed, so the current score can be used now instead of having to wait for the slave to pass one back. If an element is found in the shared table, its score is read and a *touched* flag is set. If the element has not yet been searched to the required depth, it is added to the uncertain list. If not, it is removed from it.

Lastly, it is checked if the shared element needs to be updated, with respect to the bounds or the priority (line 19 to 26).

```
Search(position : Node, α, β, plyToGo, fullDepth, priority : Integer)
  ⋮
  if IsMaster()
   & fullDepth − plyToGo = GetSearchHorizon()
   & plyToGo > GetMinimumGranularity()
     then ▷ We have reached the search horizon
          sharedElement : SharedTableElement ←
           GetSlavesSharedTable().GetElement(successorBoard)
          if sharedElement = null
            then ▷ Assign work
                 successorScore ← −Analyze(successorBoard)
                 GetUncertainList().Add(sharedElement)
                 DistributeWork(successorBoard, −β, −α,
                                            − successorScore, priority)
            else ▷ Get score from slave
                 successorScore ← −sharedElement.getScore()
                 sharedElement.SetTouched(true)
                 if sharedElement.GetDepth() < plyToGo −1
                   then GetUncertainList().Add(sharedElement)
                   else GetUncertainList().Remove(sharedElement)

                 if sharedElement.HasBadBounds(−β, −α)
                   then sharedElement.SetAlpha(−β)
                        sharedElement.SetBeta(−α)
                        SendBadBounds(sharedElement)

                 if not sharedElement.GetPriority() = priority)
                   then sharedElement.SetPriority(priority)
                        SendChangedPriority(sharedElement)
  ⋮
```

16.4 Unimplemented Parts

This section presents the overview of different parts of the system that at the conclusion of the project were not implemented.

16.4.1 Recovery

The recovery feature in our program has already been described briefly in Section 12.2 on page 56. The intention is to be able to replace a master, that for some reason has become unavailable, with a slave. In the case where a slave discovers that it can not get in contact with its parent, the slave invokes a method in the `Recovery` class that selects a replacement for the lost server.

This is done by an election algorithm that picks the most suitable substitute in order to keep the system as efficient as possible. Any shared information from the lost master can be recovered by the replacement by contacting the slaves. The same is true for information recovery from the parent to the lost nodes, where the lost information can be recovered by contacting the next node in the parent list.

This feature was not implemented since we wanted to focus on the actual distribution and it was also not a part of our problem formulation, Section 1.1 on page 1.

16.4.2 Redistribution

If a slave is overloaded, we need to take some of the work from it and distribute the work to the slaves that are not burdened. A slave is overloaded if another found slave is less burdened (the slave is then underloaded). We define the work load of a slave as the negated lowest search depth of all the slave's work assignments. For instance, if slave A has searched four work assignments to depths (2, 2, 5, 6) respectively, it has a workload of -2. If a slave B has searched two work assignments to depth (4, 6) respectively, slave B has a work load of -4. Slave A is now overloaded compared to slave B, and we would move work assignment from A to B.

The reason for not implementing control of redistribution is that it requires a more thorough design and implementation than first expected in order to get a beneficial effect from this concept. First, overload control generates an overhead in sorting the slaves from the least to most overloaded. Secondly, it does not redistribute a board which is currently being searched. If redistribution of currently searched board, all the calculations made by the slave in the current iteration are lost, and the information stored about this board in the shadow- and transposition table would be useless for the slave.

One problem is a master does not know what board a slave is currently working on, and it would generate a lot of otherwise unnecessary messages, if a slave notifies their master each time they start working on a new board. An implementation of this is possible in a later version of the system.

16.4.3 Exemptions

If the size of a work assignment is too big to be redistributed, it could be beneficial to break it into several smaller assignments. We give an estimate of the size of the work through the amount of analysed bottom nodes. This includes the bottom nodes found in the sub-trees given to the slaves (if any). This means that the slaves maintains a count in each entry in the shared table so the master can give a useful estimate. We only want to exempt a work assignment, if the estimation of the size is x times greater than the average work size, where x needs to be found through empirical tests. This leads to the problem of finding the average size of a work assignment, which will change throughout the game. Therefore we also maintain a table of average work sizes for each depth (for instance, a work assignment searched to depth 3 will be smaller than if it was searched to depth 4), which will be updated throughout the game.

A master in the system makes a call to an EXEMPTION-CHECK algorithm after each iteration of a work assignment. It should run through each entry in the table shared with its slaves, and find if any of these have a work size x times greater than the average for the depth that it was searched to. If this is the case, it is removed from the slave that is working on it and we set an exemption flag for that board position.

If the master reaches a board position in one of its passes, and it has found that it has reached the depth where it should distribute a work assignment to a slave, it checks if the exempt flag has been set. If so, the master will not distribute the node, but instead search the work assignment one depth further itself and then distribute the work assignment to the child nodes instead. This has the effect of splitting up a piece of work in the amount of legal moves from the board position.

It is only the EXEMPTION-CHECK algorithm that is unimplemented in Morphy Chess. The design of the rest of the system should be able to handle an exempted board position. Like redistribution, this can be implemented in a later version, since the rest of the system takes it into account.

16.5 Evaluation of Implementation

In the design criteria section, a list of quality goals were presented. Some of these goals could not be evaluated through the design of Morphy Chess, and were passed on to the implementation to be evaluated. In this section, we will revisit the design goals with focus on the implementation:

Correctness The load balancing features of our program have not been implemented and on that account, we do not fulfil the demand set forth in the problem formulation (see Section 1.1) to address load balancing issues in the implementation. Since we are using a cluster though, this should not prove to be of much concern, since a cluster should be fairly homogeneous.

Reliability The reliability from recovery has not been obtained. Again, this will not prove to be much of a problem since we are working on a closed cluster.

Maintainability As in the design, we have kept the different components separated. We have also separated the GUI from the rest of the classes in the system and done the same to the search algorithm. The added model described in Section 16.1 also helps to a more maintainable code.

Flexibility By having a modular design and by implementing with this in mind, we have achieved some flexibility. The change of the search algorithm to a more modular structure also makes the system more flexible, since we can now easily change the search algorithms.

Comprehensibility We achieve comprehensibility by having a modular design. Each part has a clearly defined purpose. By implementing with this in mind and by commenting in the same manner, the system will become more comprehensible. Also by giving classes and methods expressive names, a more comprehensible code is obtained.

Portability Even though the Morphy Chess program in itself should be highly portable, it relies on other systems which we found much less portable than first expected. It uses a specific package of wrapper classes for the MPI communication, which again relies on an actual MPI library. It should still be possible to make Morphy Chess run on other platforms than the one it was developed for, but it will not be a triviel task as stated in the design goals.

Part IV

Test

Chapter 17

Test

For Morphy Chess, we have performed tests in the following areas: unit, acceptance, optimisation and evaluation testing. Unit testing focuses on the low level components while acceptance testing focuses on the high level design. Optimisation testing is an attempt to find the optimal configuration of the system, and evaluation testing is about how well the system performs.

In this chapter, we will go through each of the tests that we have performed for Morphy Chess, describe how we performed the individual tests and finally present the results.

17.1 Unit Testing

Unit testing is applied to ensure the functionality of the individual units when isolated from the rest of the system. By ensuring the function of each low-level component, it becomes easier to do debugging and functionality testing later on in the development process.

During the development of our program, we have continiously tested the code by means of both structured test cases and integration testing when combining units.

To be able to test the distributed system, we have made a special version of the server which bypasses the RMI connection and runs a number of test cases on the server-system using the unit testing framework JUnit [Jun]. This is done by placing listeners throughout the system which catches events, and passes these to the messaging system on the master server.

17.2 Acceptance Testing

Since the original chess engine has been modified, the testing of functionality in our program must include testing of all possible moves in the game. Since this is practically impossible, a set of test cases are created to handle the standard operations. The JUnit framework is also used here to get a structured testing of the changes made, and to evaluate the original engine. Within this framework a number of tests have been made to assert the legality of moves for the individual pieces. Each test contains a test case of how the piece can move and make captures. Additionally, the individual piece may have some special circumstances concerning certain moves. In the following is given an overview of how these tests were performed.

Pawn

- Moves and captures.
- En passant.
- Pawn promotion.

Rook

- Only moves and captures have been tested.

Bishop

- Only moves and captures have been tested.

Knight

- Only moves and captures have been tested.

Queen

- Only moves and captures have been tested.

King

- Moves and captures.
- Castling left and right.
- In check.
- Checkmate.

Miscellaneous

- Draw situations (see Footnote 1 on page 76).

The source code and documentation of these tests can be found on the Morphy Chess website at [CEJ+]. Here, each of the acceptance tests can be found. By doing these tests we found minor flaws in the code. This included errors in complex moves like en passant and castling.

17.3 Optimisation Testing

As explained in Section 16.1 on page 75, we have a configuration class in our system. This controls most of the hardcoded values and settings of the system and optimisation testing is an attempt to find the best combination of settings. We have performed tests with different configurations of the following:

- Search horizon
- Size of shadow table
- Search algorithms
- Minimum granularity

All the tests in both this section and the following on performance testing (see Section 17.4 on page 93) has been performed on the cluster with nodes of the following specifications:

- Pentium 4 HT 2.6 GHz
- 2*512 MB 400 MHz ECC RAM
- 80 GB HDD
- Intel 875PE motherboard
- Intel Pro/1000 network card on CSA connected to an HP Procurve switch

17.3.1 Shadow Table Size Test

The following test has been performed with the purpose of finding optimal values for the size of the shadow table. As shadow tables are passed between the nodes, it is important to balance the message overhead with the performance gain. To do this, we have executed a series of tests to examine how long time it takes to reach different depths with different sizes of the shadow tables. The test was run with the following configuration:

- Algorithm: Alpha-Beta Pruning.
- Search horizon: 1.
- Minimum granularity: 2.
- Use shadow tables: true.
- Number of machines used: 18.

Entries	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6
64	59 ms.	82 ms.	186 ms.	1,357 ms.	1,554 ms.	13,824 ms.
128	35 ms.	58 ms.	161 ms.	1,423 ms.	1,476 ms.	13,433 ms.
256	34 ms.	56 ms.	159 ms.	1,502 ms.	1,549 ms.	13,492 ms.
1024	35 ms.	57 ms.	160 ms.	1,391 ms.	1,445 ms.	13,644 ms.
102400	41 ms.	64 ms.	166 ms.	1,025 ms.	1,390 ms.	13,813 ms.

Table 17.1: Shadow table size test results.

The values in Table 17.1 show that there is no difference between the five tests. This indicates a bug in the transmission of shadow tables, which was later confirmed. There were no sharing of shadow tables between any servers, which means that the five tests are infact identical. This also means that the configurations for anything regarding shadow tables have been disregarded in the following tests.

17.3.2 Search Horizon Test

When testing the search horizon setting, it was found that zero and one were the only working values. On the cluster any values above these resulted in the program crashing. Outside the cluster this occured with all values above zero. Due to instability of the cluster, it was not possible to reproduce a large enough quantity of consistent test data to get any conclusive results. However, preliminary results indicate that a search horizon of one give better results. As noted above, a test with search horizon of two has never succesfully been completed and so it is unknown whether or not this would provide further improvement. Due to the fact that the cluster was able to handle more than the development servers, it would seem that internal communication hardware could be the bottleneck.

17.3.3 Minimum Granularity Test

In the last optimisation test we focus on the value for minimum granularity. The values from 0 to 10 have been tested with the following configurations.

- Search horizon: 1.
- Algorithm: Alpha-Beta Pruning.
- Number of machines used: 19.

The symbol '-' in Table 17.2 on the facing page indicates that the specified depth could not be reached within 2 minutes.

As seen in Table 17.2 on the facing page, the performance steadily increases until granularity of two, after which it decreases until it reaches higher granularities again. This shows that a granularity value of two is

Size	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6	Depth 7
0	47 ms.	-	-	-	-	-	-
1	34 ms.	56 ms.	712 ms.	1,070 ms.	1,671 ms.	15,388 ms.	-
2	34 ms.	56 ms.	160 ms.	1,513 ms.	1,603 ms.	16,496 ms.	88,579 ms.
3	47 ms.	69 ms.	175 ms.	729 ms.	41,837 ms.	42,034 ms.	-
4	34 ms.	56 ms.	160 ms.	715 ms.	8,492 ms.	-	-
5	38 ms.	59 ms.	157 ms.	676 ms.	7,777 ms.	86,086 ms.	-
6	75 ms.	95 ms.	199 ms.	759 ms.	8,644 ms.	94,237 ms.	-
7	49 ms.	72 ms.	175 ms.	727 ms.	8,469 ms.	93,145 ms.	-
8	34 ms.	56 ms.	160 ms.	707 ms.	8,353 ms.	93,081 ms.	-
9	64 ms.	88 ms.	192 ms.	747 ms.	8,517 ms.	93,774 ms.	-
10	32 ms.	56 ms.	160 ms.	724 ms.	8,489 ms.	92,917 ms.	-

Table 17.2: Minumum granularity size test results.

the best choice, as it is the only one to reach a depth of seven plies. The gain in performance with higher granularities can be explained from the fact that the master server never reaches the point of distribution and therefore makes a non-distributed search without that overhead.

17.4 Performance Testing

In our system, the primary concern regarding performance is the non-functional parts of the system in respect to the non-distributed versus the distributed version of the program. That is, a test that shows that the distribution process contributed something to the overall performance of the program.

17.4.1 Engine Comparison

This test shows how Morphy Chess performs compared to the original JavaChess engine. The test is run with the original engine running on a single machine on the cluster.

- Search horizon: 1.
- Minimum granularity: 2.
- Number of machines used: 17.

Morphy Chess is run with the Alpha-Beta Pruning algorithm ($\alpha\beta$).

Note that “original” engine does not refer to the original JavaChess program but to the engine classes used together with the Morphy Chess server part. A fresh download of the original JavaChess program performs as seen in the first row of Table 17.3[1]. Note that the time of the original engine is shown in seconds as a result of the limited debug options available in JavaChess.

	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6	Depth 7
JavaChess	<3 s.	<3 s.	<3 s.	3-5 s.	30-45 s.	Not reached	Not reached
Morphy original	128 ms.	149 ms.	296 ms.	1,147 ms.	9,806 ms.	190,818 ms.	954,731 ms.
Morphy $\alpha\beta$	34 ms.	57 ms.	161 ms.	1,499 ms.	1,604 ms.	28,686 ms.	111,776 ms.

Table 17.3: Engines test results.

As can be seen in Table 17.3, the original single-processor engine and the unmodified JavaChess are outperformed consistenly. Morphy Chess reaches 5 plies about six times as fast as the original engine. A depth of 7 plies is reached more than 8 times as fast as with the original engine.

Also, a comparison of Table 17.3 and the performance measurements from the original JavaChess game given above shows that we have actually succeeded in making our server part perform much better than the original game.

17.4.2 Scalability Test

This test is run to find out how well Morphy Chess scales on several machines.

The test uses the following setup:

- Algorithm: Alpha-Beta Pruning.
- Search horizon: 1.
- Minimum granularity: 2.
- Number of machines used: {2, 4, 6, 8, 9, 11, 19}.

The results of the test are shown Figure 17.1. It shows the time it takes to complete depth 4, 5 and 6 with different numbers of servers. As can be seen, more machines result in lower search times. When the number of servers increase, the efficiency per server should decrease as an effect of work distribution. This can be observed on the figure, where the search time decreases in a non-liniear manner, flattening as the number of servers increase.

[1]Tested on an AMD Athlon 1.11 GHz with 256 MB of RAM running Windows XP.

# nodes	Depth 4	Depth 5	Depth
2	1,242 ms.	5,942 ms.	88,896 ms.
4	973 ms.	3,450 ms.	58,436 ms.
6	869 ms.	2,180 ms.	30,180 ms.
8	588 ms.	1,675 ms.	27,808 ms.
9	1,248 ms.	2,583 ms.	23,095 ms.
11	873 ms.	1948 ms.	19,278 ms.
19	1,484 ms.	1,610 ms.	15,408 ms.

Table 17.4: Scalability test results.

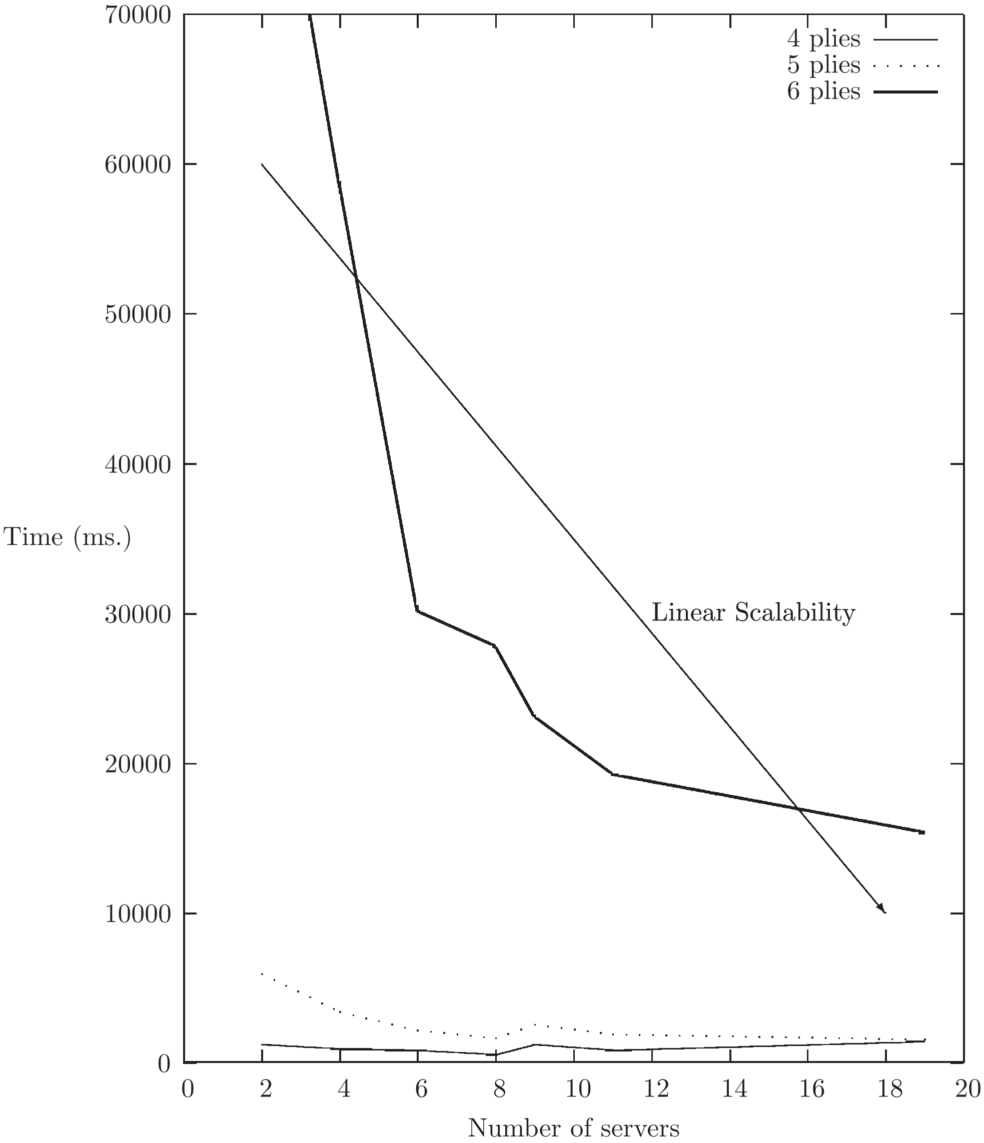


Figure 17.1: Overview of scalability test.

Part V

Project Conclusions

Chapter 18

Conclusion

The problem formulation was: " The goal of this project is to take an existing chess program and modify it to be distributed. The more computing power and memory available, the better the chess program should perform. Load balancing issues will need to be addressed."

We have chosen the chess program JavaChess [FR] as basis for our distribution. We have developed a distribution algorithm based on APHID [Bro97] and modified JavaChess accordingly. Load balancing issues have been addressed in the design of the algorithm but are not implemented in the version available as of project conclusion because of time constraints. Tests showed that the distributed Morphy Chess clearly outperformed JavaChess.

The above is elobarated in the following:

18.1 Analysis

During the analysis phase, we have examined different algorithms from gametheory which gave us an insight into computer chess. We have looked at the distribution method APHID, from which we gained inspiration for later designing a method of distribution. In addition to this, we have examined a number of different chess programs and have chosen JavaChess as basis for the distribution. The choice of JavaChess was based on a number of factors with the most important ones being the license, complexity of used algorithms and source code readability.

The information gathered in the analysis phase gave us the insight into computer chess and the algorithms used to continue our work in the design with the development of our own distribution algorithm.

18.2 Design

In the design phase, we have developed the distribution algorithm for Morphy Chess. A special feature of this algorithm is its possibility to set up a communication structure according to available bandwidth. Two forms of loadbalancing were designed for this distribution. The first redistributes work assigments from overloaded to underloadet servers, while the other splits large pieces of work into minor ones. Another feature was that once the system was setup, it could recover from crashes by using redundant data to remake parent servers in the distribution.

By creating a distribution structure on the basis of bandwith, we obtain a system where we use all resources available with as little resources as possible used for distribution control. In the same way, the loadbalancing algorithms help us to ensure that work assignments are finished according to their importance.

18.3 Implementation

Before we were able to start implementing our design, we had to clean up, correct and modify the JavaChess sourcecode. With minor modifications to the design, we successfully implemented Morphy Chess with a working distribution using the MPI communication framework. Three subjects that we covered in the design were not implemented. First, recovery was dropped, since we were developing for a cluster and have chosen to focus mainly on the distribution. In addition to this, the two loadbalancing algorithms were not implemented at the time of project conclusion due to time constraints. As a consequence from the fact that we developed for a cluster, a simplification to the distribution tree setup was also made. Instead of calculating threshold values for each parent server, we have made the assumption that one calculated value will be true for the entire system due to the homogenous nature of clusters.

18.4 Tests

In the testing phase, we performed a number of tests in the areas of unit, acceptance, optimisation and performance testing. Unit testing helped us reveal and fix minor flaws in the code, while acceptance testing revealed bugs in the chess game functionality. During the optimisation tests we were unable to complete all the proposed tests but managed to observe a set of optimised values. The performance test showed that we could greatly outperform the original engine. A scalability test showed that the performance of the system increased as the number of nodes went up.

Chapter 19

Future Work

In this chapter, we will review different areas that would be interesting to explore if we got the opportunity to work more on this project.

For Morphy Chess, we used the evaluation function unmodified as it existed in the original JavaChess program. This evaluation function does not take into account many tactical elements of a chess game and uses a lot of hard-coded values that seem untested. It would be interesting to make a genetic algorithm to approximate some better values, to implement a new evaluation function from scratch or implement a copy of the evaluation function from, for example, the open source chess program GNU Chess, which is known to be a fairly strong chess program.

It would also be interesting to implement the discussed load balancing algorithms from the design and see how much they would help to increase the performance of Morphy Chess.

It could also be nice to implement the recovery algorithm and perform tests on it by purposefully crashing nodes in the cluster.

A different approach to hardware for the system could also be to implement a SETI@home-like [Uni] version of Morphy Chess that should allow Internet users to lend us some computing power. This would require a dynamic distribution tree that allows for the adding and removal of nodes without much warning. In addition to this, the threshold value could also be made dynamic. This would be beneficial in a situation where a computer does not have a stable bandwidth and the amount of slaves it can handle changes drastically during the course of a game.

Continuing with the threshold value, it would also be interesting to see if there is a minimal threshold where it is beneficial to make a server a master/submaster.

The possibility to play against other chess engines would also be interesting. This could be implemented by using for example XBoard / WinBoard [Man].

For now, we have chosen not to focus on the graphical part of the system.

However, It would be nice, as stated in the problem formulation, to be able to control more of the game directly from the GUI. This control could include adding and removing servers and retrieving information about eg. server topology during a game.

Part VI

Appendixes

Appendix A

Implementation Plan

The following gives an overview of the order in which the different parts of the system will be modified or created.

JavaChess Program Improvement First, some changes has to be made to the original JavaChess program. The changes include adding support for draws, and a general clean-up.

Division into Modules In order to distribute some of the functionality of the JavaChess program, it is important that the code is kept modular in order to easier predict the consequences of any changes. This requires some initial work on the original code.

Client Clean-up As any computation will be performed on servers, there is no need for a chess engine as such on the client side. To ensure that this is the case, all parts referring to the engine on the client part has to be either modified to use results from the server or removed from the system.

Communication Between Client and Server The first step to create a distributed system is to make a simple client/server communication model. We do this by using parts of the old engine to create a server component capable of calculating a move in the same way as the original JavaChess program. The client component is linked with this part, by implementing Java RMI communication into the system. This will enable us to separate the client completely from the engine part and make it possible to finish the client part without much work done on the server part.

Preparation of Classes in the Model Component To support the implementation of the different functions in APHID like shadow tables and a shared table, some additional classes are needed to encapsulate data structures and various minor functions like prioritizing entries in a shared table.

Communication Between Servers An essential part of our system is the communication between the different servers. This communication is built on MPI, which we have already examined earlier. Now, we have to make MPI fit seamlessly into the rest of the system in a efficient and modular way.

Implementation of the Distribution algorithm When the above steps are completed, we are ready to implement the core functionality of the algorithm described in the design. This includes replacing the original search algorithm with our own implementation that has support for sharing information via shadow- and shared tables.

System Test The final part of the implementation phase will be to test the system. First to reveal any bugs and logical errors that might exist in the system. Later to test the system with different setups to determine the optimal values for any constant variables, and finally to test the performance of our system. The tests are described in more detail in Chapter 17 on page 89.

References

[Bri] Encyclopædia Britannica. John von Neumann. `http://www.dam.brown.edu/people/yiannis/neuman.html`. (Last visit 26.09.04).

[Bro] Mark G. Brockington. APHID Game Library. `ftp://ftp.cs.ualberta.ca/pub/games/aphid`. (Last visited 13.12.04).

[Bro97] Mark G. Brockington. *Asynchronous Parallel Game-Tree Search*. PhD thesis, University of Alberta, 1997.

[BS99] Mark G. Brockington and Jonathan Schaeffer. APHID: Asynchronous Parallel Game-Tree Search. `http://utenti.lycos.it/gekochess/link/parallelgame.pdf`, 1999.

[CEJ+] Jonas Bang Christensen, Jacob Eskildsen, Lars Hornbæk Jensen, Kristian Thorsted Madsen, Niels Birkekjær Platz, Stinus Silassen, and Bjørn Mølgård Vester. Morphy. `http://www.larshj.dk/morphy`. (Created 15.12.04).

[DKC03] T. A. Marsland D. Kopec and J. L. Cox. Search. `http://www.cs.ualberta.ca/~jonathan/Papers/Papers/icpp02kishi.ps`, September 2003. (Last visited 06.12.04).

[Dom] Public Domain. Wikipedia - The free Encyclopedia. `http://en.wikipedia.org/wiki/Main_Page`. (Last visit 29.09.04).

[EE99] Maurice Eggen and Roger Eggen. Java Versus MPI in a Distributed Environment. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 390–395. CSREA Press, July 1999. (Last visit 06.12.04).

[FR] Harald Faber and Andreas Rueckert. JavaChess. `http://java-chess.de`. (Last visit 10.10.04).

[Fraa] Mikolaj Franaszczuk. Ai chess. http://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/chessmain.html. (Last visit 05.12.04).

[Frab] Colin Frayn. Chess Programming Theory. http://www.chessbrain.net/beowulf/theory.html. (Last visit 28.09.04).

[Gin00] Herbert Gintis. *Game Theory Evolving*. Princeton: Princeton University Press, 2000.

[Hya] Bob Hyatt. Crafty. http://www.crafty-chess.com/. (Last visit 05.12.04).

[Jun] JUnit, Testing Resources for Extreme Programming. http://www.junit.org. (Last visit 16.11.04).

[KSCea] Chua Kong-Sian, Stuart Cracraft, and et al. Gnu-chess. http://www.gnu.org/software/chess/. (Last visit 05.12.04).

[Kum] Senthil Kumar. The genius. http://thegenius.sourceforge.net/. (Last visit 05.12.04).

[LMS00] Peter Axel Nielsen Lars Mathiassen, Andreas Munk-Madsen and Jan Stage. *Object-oriented analysis & design*. Aalborg, Marko, 2000.

[Man] Tim Mann. Tim Mann's Chess Pages. http://www.tim-mann.org/xboard.html. (Last visited 14.12.04).

[MMMNS01] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Objekt Orienteret Analyse & Design*. Marko, 2001.

[Mora] Bruce Moreland. Iterative Deepening. http://www.seanet.com/~brucemo/topics/iterative.htm. (Last visit 11.10.04).

[Morb] Bruce Moreland. Quiescent Search. http://www.seanet.com/~brucemo/topics/quiescent.htm. (Last visit 30.09.04).

[Morc] Bruce Moreland. Zobrist Keys. http://www.seanet.com/~brucemo/topics/zobrist.htm. (Last visit 14.11.04).

[MPI] The Message Passing Interface (MPI) standard. www-unix.mcs.anl.gov/mpi. (Last visit 16.11.04).

[Nor] Nordugrid. http://www.nordugrid.org. (Last visit 16.11.04).

[Pla] Aske Plaat. MTD(f). http://www.cs.vu.nl/~aske/mtdf.html. (Last visit 18.10.04).

[PSPdB] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-First and Depth-First Minimax Search in Practice. http://www.cs.vu.nl/~aske/Papers/csn.pdf. (Last visit 18.10.04).

[Rei83] Alexander Reinefeld. An Improvement to the Scout Tree Search Algorithm. *ICCA Journal*, 6(4), December 1983.

[RMI] Java Remote Method Invocation (Java RMI). http://java.sun.com/products/jdk/rmi. (Last visit 16.11.04).

[Ros] Don Ross. game-theory. http://plato.stanford.edu/entries/game-theory/. (Last visit 01.12.04).

[Sch] Jonathan Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. http://www.brillianet.com/programming/artificial_intelligence/tutorials%/tut1.pdf. (Last visit 29.09.04).

[Suna] Sun Microsystems, Inc. Code conventions for the Java™ programming language. http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html. (Last visit 28.11.04).

[Sunb] Sun Microsystems, Inc. J2SE 1.4.2. http://java.sun.com/j2se/1.4.2/. (Last visit 29.11.04).

[Sunc] Sun Microsystems, Inc. JavaDoc Tool. http://java.sun.com/j2se/javadoc. (Last visit 16.11.04).

[Uni] University of California. SETI@home . http://setiweb.ssl.berkeley.edu/. (Last visited 14.12.04).

[Wal] Bill Wall. Computer Chess History. http://www.geocities.com/SiliconValley/Lab/7378/comphis.htm. (Last visit 28.09.04).