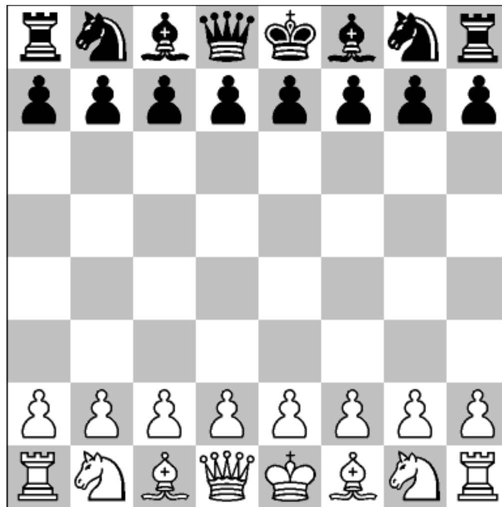
Alpha-Beta with Sibling Prediction Pruning in Chess

Jeroen W.T. Carolus



Masters thesis Computer Science

Supervisor: Prof. dr. Paul Klint

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
University of Amsterdam
Netherlands

2006

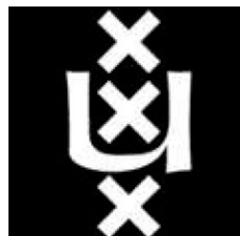


Table of Contents

Abstract....4
Organization....4
1 Introduction....5
1.1 Chess computer history....5
1.2 Basics....5
1.5 Complexity....7
2 Overview of search algorithms....8
2.1 Minimax....8
2.2 Alpha-Beta....10
2.3 Nega Max....11
2.4 Transposition tables (*)....13
2.5 Aspiration search (*)....15
2.6 Minimal (Null) Window Search / Principle Variation Search (*)....15
2.7 Iterative deepening (*)....16
2.8 Best first search (*)....17
2.8.1 SSS*....17
2.8.2 MTD(f)....18
2.9 Forward pruning....18
2.9.1 Null Move heuristic (*)....19
2.9.2 ProbCut 19
2.10 Move ordering (*)....20
2.10.1 Killer heuristic....20
2.10.2 Refutation tables....20
2.10.3 History heuristic....20
2.11 Quiescent search....20
2.12 Horizon effect....21
2.13 Diminishing returns....22
2.14 Search methods measured....22
3 Sibling Prediction Pruning Alpha-Beta....24
3.1 Evaluation function....24
3.2 Maximum difference....26
3.3 SPP Alpha-Beta....27
3.4 Expected behavior....30
3.4.1 Evaluation function and MPD....30
3.4.1 Move ordering....31
3.4.2 Stacking....32
3.4.3 Balance and material....32
4 Test setup....32
4.1 Search algorithms for comparison....33
4.2 Evaluation function dependencies....33
4.3 The test set....34
4.3.1 The positions....34
4.4 Measurements....35

4.5 Correctness.. 35
4.6 Testing environment & software used..36
5 Results..36
5.1 Exceptional results.. 37
5.2 Presentation & Interpretation of the results...37
5.2.1 AB vs SPP.. 38
5.2.2 ABT vs AB.. 38
5.2.3 SPPT vs SPP..38
5.2.4 SPPT vs ABT..39
5.3 Correctness... 39
6 Conclusions..39
7 Future investigation...40
Acknowledgments..41
Abbreviations.. 41
References & Bibliography..42
Appendix A, Test Results.. 44
Appendix B, Positions of the test set.. 51

Abstract

A lot of computation time of chess computers is spent in evaluating leafs of the game tree. Time is wasted on bad positions. In this research, a method that predicts the maximum evaluation result of sibling chess positions is defined. The idea is to prune brothers of a bad position without information loss.
The resulting algorithm is a forward pruning method on leafs of the game tree, which gives correct minimax results. A maximum positional difference of the evaluation function on siblings must be correctly measured or assessed for the algorithm to work properly.
The results of this thesis cannot be generalized because of the dependencies on the evaluation function, but are intended as a proof of concept and show that it is worthwhile to investigate Sibling Prediction Pruning Alpha-Beta in a broader context.

Organization

The organization of this thesis is as follows. First, a little history and theoretic background about the game of chess are provided. Then, an introduction to existing search algorithms is given. After this is done, the principles and conditions for the Sibling Prediction Pruning (SPP) Alpha-Beta search algorithm are covered. The system and test setup for the investigation of this thesis are described. Finally the results are presented and interpreted, followed by the conclusion and future investigation.

1 Introduction

1.1 Chess computer history

The first chess automaton was created in 1770. It had to contain a human inside to do the thinking... - that does however illustrate the point that people have been fascinated by the complexity of the game and have been trying to create automated chess.

The first electro-mechanical device dates back to 1890 and it could play chess end-games with just the kings and one rook, the so called KRK end games.

In terms of computer history, computer chess has a long history. Since the 1940's chess has been in the picture of automation and computer science. Much effort was put in developing theory and creation of chess programs on early computers. The paper "*Programming a Computer for Playing Chess"* by Claude Shannon [5] is recognized as a primary basis for todays computer chess theory.

In 1997 a chess computer called Deep Blue defeated chess grandmaster Gary Kasparov. Recently (June 2005), another super chess computer, Hydra, defeated Micheal Adams with big numbers – the era that computers dominate humans in chess has begun.

Many researchers in the field of computer chess argue that chess is seen as the Drosophila of Artificial Intelligence, but that in contradiction to the insights that the fruit fly has brought to biology, not much progress has been achieved in knowledge of chess for A.I. (Donald Michie [27], Stephen Coles [11]). This observation certainly reflects practice, where chess computers get their strength from brute force search strategies, rather than specific A.I. techniques.

Research on chess mainly focuses on two fields.

- The quest for search efficiency, which resulted in many optimizations to the search algorithms of chess computers. This field concentrates mainly on pruning (cutting) branches from the game tree for efficient search. Donskoy and Shaeffer argue that it is "unfortunate that computer chess was given such a powerful idea so early in its formative stages" [28]. They also note that new ideas, where the search is guided by chess knowledge are being investigated – that the research becomes more A.I. oriented - again.
- Representing chess knowledge in chess computers. This field concentrates on the evaluation of a position on the chess board, in order to differentiate which of given positions is better. Techniques used to represent knowledge in the evaluation function of chess computers range from tuning by hand, least squares fitting against grandmaster games (used for Deep Thought and later Deep Blue [24]), to Genetic algorithms [2, 3].

1.2 Basics

Chess is a zero-sum, deterministic, finite, complete information game.

- *Zero-sum* means that the goals of the competitors are opposite – a win is a loss for the opponent. In case of a draw , the sum of the game is zero. Suppose a position could be valued as +10 for one player, then the score for its opponent is and must be -10.
- *Deterministic* means that all possible games are either winning for white, winning for black, or a

draw.

- *Finite,* because games cannot not take forever (three repetitions of a position result in a draw, also the 25 move rule, where 25 moves occur without a hit or a pawn move is a draw).
- *Complete information* means that there is no hidden information or uncertainty like in a card game such as poker and the game is played in a sequential fashion: both players have the same information.

A two player game such as chess, can be represented by a tree. Each node is a position on the board, starting at the root with the beginning position of the game. The vertices then are the possible moves, leading to the next nodes (positions). This way, every possibility after one move can be generated and added to the tree. A game after it has been played can be viewed as a path in the "game tree".

A single move by black or white is called a ply. A (full) move contains the move from both white and black. Sometimes a ply or half move is also called a move; the terminology in the literature is inconsistent.

In the example below a simplified game tree.

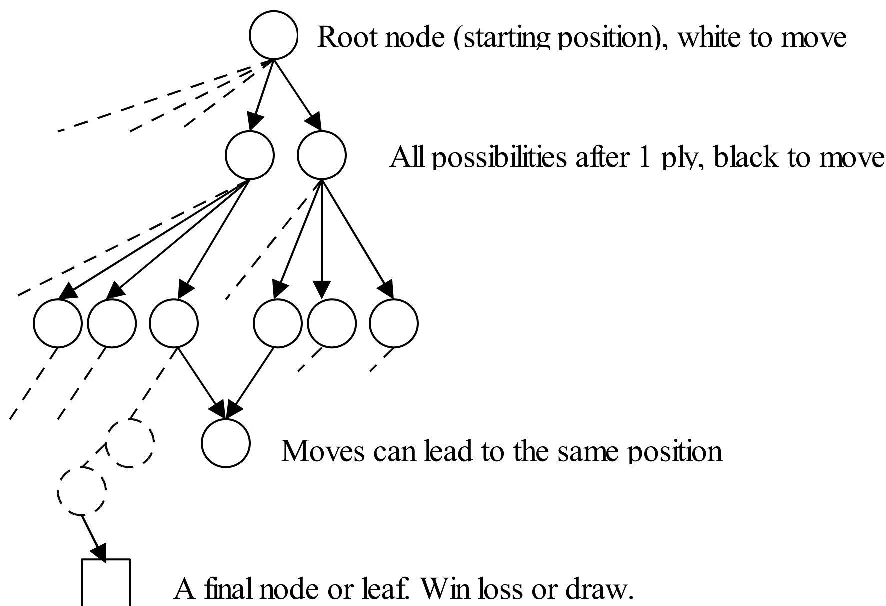


Figure 1: Game tree

It must be stressed that the game tree of chess is actually a *directed acyclic graph* in the case of chess. It is not cyclic because the rules of chess prohibit eternal loops and as a result, each sequential position is unique. Because different paths may lead to the same position it is a graph and not a tree. But since such a graph can be represented as a tree, it is often referred to as a tree and search methods treat it as such.

1.3 Computer Chess

The 1950 paper by Shannon (*"Programming a Computer for Playing Chess"* [5]) still describes the basis of today's chess computers. Since then nothing much has changed in the way computers are programmed to play chess. A chess program usually contains three modules that make playing chess possible. These are:

- move generator
- evaluation function
- search algorithm

The move generator module is used to recursively generate the (legal) moves and form a game tree. The evaluation function assigns a value to the leafs of the tree. The search method finds the path to the best position (value) in the tree. Logically, when a winning position is found, the evaluation function will return the highest possible number within the system, because it is the best position possible. The values for the opponent should be the exact opposite, negated value (zero sum property). All other non winning or non final leafs will get a numerical value between these maximum values, assessed by the evaluation function. How the search engine uses this information to find a move will be covered in the section about search algorithms.

1.5 Complexity

The term complexity used in this paper's context is to describe the computer time or space that is needed to solve a chess problem. Solving a chess problem is searching a forced sequence of moves that leads to the position that is looked for in the problem (usually a win for one of the players).

To get a numerical idea of the search space of a chess problem, a few numbers on chess are presented next. An average game situation on a chess board allows 30 legal moves [5]. With a complete information game, all possible games that can ever be played can be generated by recursively applying the rules of the game. This way, the game-tree can be generated. The root node is the starting position of the game. The 20 vertices from this root node, are the first possible moves of the white player (16 pawn moves plus 4 knight moves). All resulting nodes at the first level of this game tree can be seen as the search space when looking 1 ply ahead.

Looking one ply ahead requires looking at the resulting 30 configurations (average case). 2 plies requires another 30 moves per configuration, setting the counter at 30*30 ($30^2 = 900$). For 6 plies ahead, the search space will contain 30^6 = 729.000.000 leafs.

Taking an average of 30 moves and an average game duration of 40 moves (80 plies) [1], there are 30^{80} ($14.7*10^{117}$) possible chess games. This number is also called the Shannon number, to honor the father of computer chess, and usually rounded to 10^{120}. The previous statement that all possible games that can ever be played can be generated is practically impossible. Computer processing speed will keep growing, but generating 10^{120} games is practically impossible. This can be argued by taking the smallest theoretical measurable time and space, the Planck time and Planck space, which are respectively 10^{-43} s and $1.6*10^{-35}$ m. Suppose a computer could be build with a processors of that size, using that amount of time to generate one game. Taking a computer with the size of the earth ($\sim 3.5*10^{21}$ m^3), this still leaves10^{55} seconds (120-43-22 = 55) for calculation.

As a result, the statement that because the game is finite all chess problems can be solved in constant

time is true, but practically impossible: chess as a whole cannot be solved.

In “Languages and Machines” [12], Sudcamp describes that the complexity of an algorithm is ideally described as a function of the input size such that it can be compared to other algorithms in terms of “the big O”. In practice, it is hard to define an input size for a chess problem. The board has 8x8 squares and there are 32 or less pieces on the board. The size of the input as number of pieces on the board is not necessarily related to the solution space that has to be explored. There can be a situation which occurs in the beginning of the game (bigger input size) that can be solved by only looking one move ahead (early mate). In another situation an endgame (small input size) could be at stake, needing a lot of foresight to find the solution.

A better parameter for the input size would the number of plies that define the search space containing the solution to the problem. Note that the number of plies needed to find a solution must then be known beforehand.

2 Overview of search algorithms

In this chapter, an overview of existing search algorithms is provided before introducing the new method of this thesis. Not all methods are directly relevant for this thesis and these are marked with an asterisk (*). All methods are methods that get their efficiency from pruning techniques. SPP-Alpha-Beta finds its origin here as a pruning technique but is quite different.

As mentioned before, complete information games can be recursively generated in a game tree, as far as memory and time allow. Since most final leaves of the tree cannot be reached (some final leaves may be seen for short games, or when the game has reached its final stage), complete knowledge of the game is impossible. As a result, non final positions must be evaluated to be able to make a choice for a move. Positions that are visited by the search algorithm can be evaluated by the evaluation function, giving a numerical value to those leafs. The best value of the game tree is the so called minimax value of the tree and determines which move (branch) should be picked.

2.1 Minimax

The goals of the opponents are opposite. Each player will try to maximize its future score, while the other will try to minimize that. Players switch turn, so looking ahead means that the players must be aware of choices for branches in the tree that the opponents can make. The minimax principle is to minimize the maximum of the opponent, maximizing the minimum reachable.

Below a figure of a game tree that is expanded to depth 3, where every leaf has been assigned a value by the evaluation function.

Reasoning for the player who has the turn in this position, after 3 plies a value of 9 is the best position. However, after a choice for the left branch, his opponent is at turn and will choose the branch that leaves a remaining maximum obtainable result of 4. A choice for the right branch turns out to be the best, where 7 is the best score that can be obtained by the opponent. Therefore the minimax value of this tree is 7. Notice that at even levels (plies) in the tree, the nodes maximize the values of their children and at odd levels minimizing is needed.

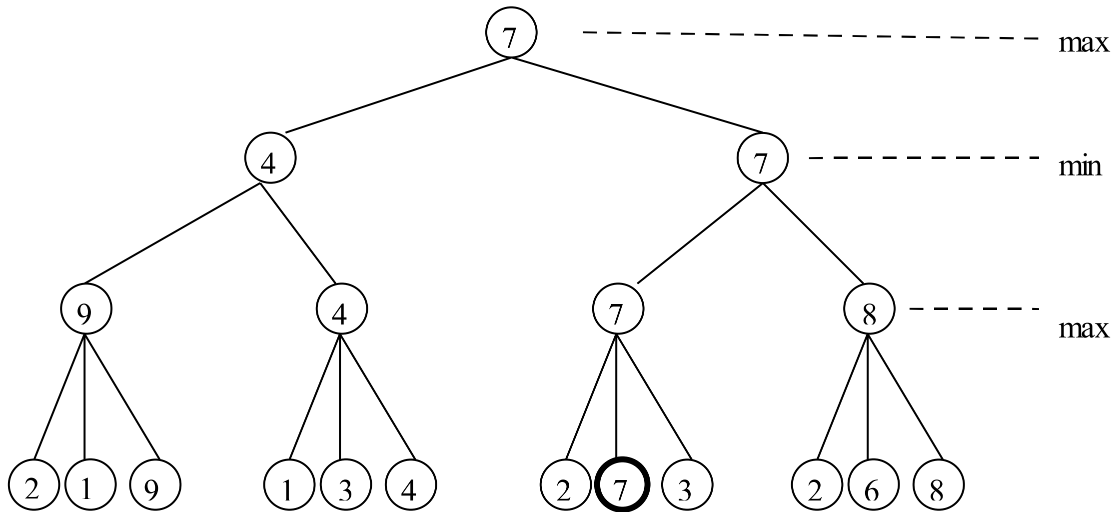


Figure 2: Minimax

A pseudo code example for an algorithm that finds the minimax value in a tree, starting at the board position contained in the board ChessBoard object:

```
// usage for a search to depth 5:
// minimaxValue = minimax(board, 5)

int minimax(ChessBoard board, int depth)
{
      return maxLevel(board, depth);
}

int maxLevel(ChessBoard board, int depth)
{
  int value;
  if(depth == 0 || board.isEnded())
    return evaluate(board);
  board.getMoves();
  int best = -MATE;
  int move; ChessBoard nextBoard;
  while (board.hasMoreMoves())
  {
    move = board.getNextMove();
    nextBoard = board.makeMove(move);
    value = minLevel(nextBoard, depth-1);
    if(value > best)
      best = value;
  }
  return best;
}

int minLevel(ChessBoard board, int depth)
```

```
{
  int value;
  if(depth == 0 || board.isEnded())
    return evaluate(board);
  board.getMoves();
  int best = MATE;
  int move; ChessBoard nextBoard;
  while (board.hasMoreMoves())
  {
    move = board.getNextMove();
    nextBoard = board.makeMove(move);
    value = maxLevel(nextBoard, depth-1);
    if(value < best)
      best = value;
  }
  return best;
}
```

Pseudo source code for the minimax algorithm

The complexity of minimax is directly proportional to the search space of size W^d, where W is the width of the tree and d the depth. The algorithm will visit every node – not only the leaves - so the number of nodes visited will be $W^{(d+1)}$. But the evaluation function will be the method consuming the most time and only works on the leaves, so visiting the nodes can be ignored. Using the average number of moves in chess, it can be formulated as:

$$tc(d)=30^d\, tc_{evaluation}=O(30^d)$$

Here, tc stands for time complexity. Note that $tc_{evaluation}$ may be dependent on the position it evaluates (a more complex a position may take longer to evaluate), so this number changes during a game.

A lot of time can be saved by using smarter search algorithms such as alpha-beta, that do not visit all leaves. Methods that will still find the correct minimax value of the tree follow the so called type A strategy. Type B strategy algorithms will filter the move alternatives and leave out moves that are considered bad moves. This can save enormous amounts of space in the game tree because whole branches are discarded (pruned) beforehand. In practice type B strategy solutions produce huge blunders because a "bad move" was actually an alternative that should have been considered. Most chess programs therefore incorporate a mix of type A and B where only after a "safe" depth a move is considered good or bad and will be followed or left out. Note that this strategy like the type B strategy does not return the absolute true minimax value of the original tree, but the minimax value of the reduced tree.

The following sections will cover search algorithms by their principle, algorithm (in pseudo code when not available in the case study) and complexity (when possible).

2.2 Alpha-Beta

The majority of search algorithms used by chess computers on game trees make use of or are derived from the alpha-beta algorithm. This algorithm is an improvement on minimax. Whereas minimax tests all leaves to find the minimax value, alpha-beta prunes leaves that cannot effect the outcome.

In the figure below an example of alpha-beta pruning is given.

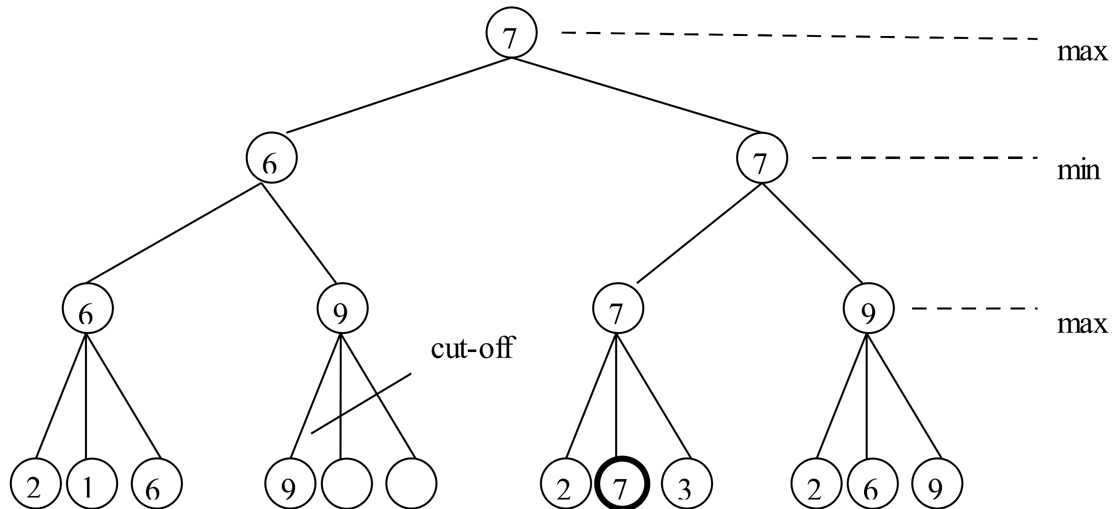


Figure 3: Alpha-Beta

Two leafs are cut-off from the search. In the leftmost leafs, 6 is the maximum value. This value is propagated to the next level in the direction of the root. In the branch where the 9 is found, the other leafs cannot influence the outcome of the minimum between 6 and 9 (Any value giving a bigger maximum than 9 will never be smaller than 6).

The algorithm uses a lower and upper bound (alpha and beta) to be able to prune branches that are outside these boundaries. The bounds are initialized at -MATE for the lower bound and +MATE for the upper bound – the minimum and maximum values within the algorithm. Every time a maximum value smaller than the current upper bound is found, the upper bound is updated with this new value. The lower bound is updated with a bigger minimum. At a minimizing level, nodes that have a child with a value bigger than the upper bound (the 9 in the example figure) can be pruned. At a maximizing level, branches with a value smaller than the lower bound can be pruned. This way, the pruning of leafs and branches using the alpha-beta algorithm leaves the minimax value intact.

In the next section about Nega Max a statement about the complexity will be given.

2.3 Nega Max

Nega max is used to make the implementation of minimax or alpha-beta easier. It does so by making use of the following equivalence:

max{ min {x1, x2, ...}, min {y1, y2...}} = max { - max {-x1, -x2, .. }, - max { -y1, -y2 }}

Here, the x's and y's are the values of leafs of a tree. Therefore his equivalence only holds when the evaluation results hold the zero sum property – which is the case with chess.

Instead of having to alternate minimum and maximum, the values only have to be negated from one level to the other (negation is the opposite value because of the zero sum property) and taking the maximum will do the trick.

Pseudo code for the alpha-beta algorithm using the nega max principle:

```
// method call with depth 5 and minimum and maximum boundaries
// minimaxValue = alphaBeta(board, 5, -MATE, +MATE)

int alphaBeta(ChessBoard board, int depth, int alpha, int beta)
{
  int value;
  if(depth == 0 || board.isEnded())
  {
    value = evaluate(board);
    return value;
  }
  board.getOrderedMoves();
  int best = -MATE-1;
  int move; ChessBoard nextBoard;

  while (board.hasMoreMoves())
  {
    move = board.getNextMove();
    nextBoard = board.makeMove(move);
    value = -alphaBeta(nextBoard, depth-1,-beta,-alpha);
    if(value > best)
      best = value;
    if(best > alpha)
      alpha = best;
    if(best >= beta)
      break;
  }
  return best;
}
```

Pseudo source code for the Alpha-Beta algorithm

It is easy to argue that this algorithm has the same complexity as minimax in the worst case scenario, where the minimax value is located in the rightmost branch (when iterating from the left to right) and all leaves must be visited. This directly shows the big effect in which order the branches are examined. The best case scenario is where only the so called minimal tree is traversed (always the best move first). Slagle and Dixon [14] showed that the number of leaves visited by the alpha-beta search (through following this minimal tree) must be at least:

$$W^{\frac{d}{2}} + W^{\frac{d}{2}} - 1$$

McIllroy then shows that the alpha-beta search on a randomly generated tree, will be 4/3 times faster than an exhaustive minimax search [15].

With a move ordering mechanism, a better performance is possible. In practice, the use of the alpha-beta algorithm will always be in combination with other improvements, such as move ordering and transposition table.

2.4 Transposition tables (*)

The mentioned search algorithms all work on trees, while the actual search space is an acyclic graph. This means that positions that have already been evaluated may be evaluated more than once. Especially in the end-game, a lot of redundancy occurs in the game tree, because of the many ways the few pieces around can create the same position. The prior algorithms have no notion of this, and will do a completely redundant search from those nodes. Storing previous results in memory makes it possible to avoid expensive re-searches by using this so called transposition. At first, this is just introducing another problem, because tree size is exponential and storing a position takes up space.

Zobrist [10] introduced a hashing method that makes it possible to store a chess position in a 64 bit number. The idea is to use a 3-dimensional array with indexes for piece, color and position, filled with random numbers (64 bit). The array could be accessed like:

zobristArray[king][black][e8]

where king, black and e8 are enumerated values in their ranges (0-5 for the six types of pieces, 0-1 for black and white, 0-63 for the squares respectively).

A hash key for a chess position can be created by iterating over all pieces on the board and binary exclusive or (xor) the hash key (initially 0) with the value from the 3 dimensional array like in the example above. Zobrist showed that with a good quality of random numbers, the chance that two chess positions would result in the same hash key can be ignored. Another good property of this method, is that a new hash key (after a change in a position as the result of a move), can be easily calculated by just xor-ing the old value with the old zobristArray value of the piece that is moved, with the new zobristArray value of the piece that is moved. This way, the move e2-e4 can be expressed in terms of the new hash key like this:

```
hashKey ^= zobristArray[pawn][white][e2];
hashKey ^= zobristArray[pawn][white][e4];
```

Because of the reversibility of the xor, this method is guaranteed to give the same result as recalculating the hash key completely by the first method. In order to incorporate all chess rules properties into the hash key, also extra random values must be used to represent the side to move and short and long castling values.

A typical entry in a transposition table would store the hash key together with the value that comes with the position. This can be an “exact” value – the value of a leaf in the search space, or the value that resulted in a cut-off: an upper bound or a lower bound. Also the depth of the node in the search space must be stored, because a transposition at a depth that is smaller than the current search depth is worthless.

Transposition table sizes are smaller than the size that is needed to store all 64 bit possible hash keys.

Therefore, the index to access a transposition table is found by taking the hash key modulo the size of the table or another way to select an index from a hash key to map to the size of the table.

In practice, using a table where positions are stored on the basis of an index, making the positions replaceable works out very well. For bigger search depths, a larger table is needed, otherwise too many entries will be re-used and a smaller benefit can be expected. For chess, an efficient size for transposition tables, relatively independent of the search depth, is 1024K entries [18].

A Pseudo code example of Alpha Beta enhanced with a transposition table:

```
// method call using depth 5:
// minimax = alphaBetaTT(board, 5, -MATE, +MATE);

// TT stands for Transposition Table

int alphaBetaTT(ChessBoard board, int depth, int alpha, int beta)
{
  int value;
  TTEntry tte = GetTTEntry(board.getHashKey());
  if(tte != null && tte.depth >= depth)
  {
    if(tte.type == EXACT_VALUE)     // stored value is exact
      return tte.value;
    if(tte.type == LOWERBOUND && tte.value > alpha)
      alpha = tte.value;            // update lowerbound alpha if needed
    else if(tte.type == UPPERBOUND && tte.value < beta)
      beta = tte.value;             // update upperbound beta if needed
    if(alpha >= beta)
      return tte.value;             // if lowerbound surpasses upperbound
  }
  if(depth == 0 || board.isEnded())
  {
    value = evaluate(board);
    if(value <= alpha)             // a lowerbound value
      StoreTTEntry(board.getHashKey(), value, LOWERBOUND, depth);
    else if(value >= beta)          // an upperbound value
      StoreTTEntry(board.getHashKey(), value, UPPERBOUND, depth);
    else                            // a true minimax value
      StoreTTEntry(board.getHashKey(), value, EXACT, depth);
    return value;
  }
  board.getOrderedMoves();
  int best = -MATE-1;
  int move; ChessBoard nextBoard;

  while (board.hasMoreMoves())
  {
    move = board.getNextMove();
    nextBoard = board.makeMove(move);
    value = -alphaBetaTT(nextBoard, depth-1,-beta,-alpha);
    if(value > best)
      best = value;
    if(best > alpha)
      alpha = best;
    if(best >= beta)
      break;
```

```
  }
  if(best <= alpha)                // a lowerbound value
    StoreTTEntry(board.getHashKey(), best, LOWERBOUND, depth);
  else if(best >= beta)         // an upperbound value
    StoreTTEntry(board.getHashKey(), best, UPPERBOUND, depth);
  else                            // a true minimax value
    StoreTTEntry(board.getHashKey(), best, EXACT, depth);
  return best;
}
```

Pseudo source code for the Alpha-Beta algorithm with Transposition Table

2.5 Aspiration search (*)

Aspiration search uses the alpha-beta principle in an optimistic way, by initializing the alpha and beta bounds to a small window around the previous value, typically plus and minus the value of a pawn. If the minimax value is within this window, the extra cut-offs pay off well. In the other case a re-search needs to be done. A re-search can be done by either using a sliding window by increasing one of the two bounds, or setting the bound that failed to + or – MATE. Using the sliding window approach can lead to large amounts of re-searches in the case of a mate or when a lot of material is lost or won. The cost of a re-search is not a big issue, since the expected value is the result of either material gain/loss or even a win/loss. The re-search might then be done with an extra ply. Moreover, this kind of search algorithm, where there is a chance that a re-search needs to be done, usually makes use of a memory table (transposition table) to cache previous results, making the re-search costs negligible.

Pseudo code with a single re-search in the case of a fail high or fail low:

```
// call method using depth = 5 and previous value = -3 and window size = 100
// minimax = aspiration(board, 5, -3, 100);

int aspiration(ChessBoard board, int depth, int prevValue, int window)
{
  int value = alphaBetaTT(board, depth, prevValue-window, prevValue+window);
  if(value >= beta)         // fail high
    value = alphaBetaTT(board, depth, value, +MATE);
  else if(value <= alpha)   // fail low
    value = alphaBetaTT(board, depth, -MATE, value);
  return value;
}
```

Pseudo source code for the Aspiration algorithm

2.6 Minimal (Null) Window Search / Principle Variation Search (*)

It has been shown that it is cheaper to prove that a subtree is inferior, than to determine its exact value [6]. The minimal window search makes use of this property by using the smallest alpha and beta window (beta = alpha+1).

Pseudo code example (with negamax principle, adapted from Marsland [6]):

```
int PVS(ChessBoard board, int depth, int alpha, int beta)
{
  if(depth == 0 || board.isEnded())
    return evaluate(board)();

  board.getOrderedMoves();
  int move; ChessBoard nextBoard;
  move = board.getNextMove();
  nextBoard = board.makeMove(move);
  int i, temp, value = -PVS(nextBoard, -beta, -alpha, depth-1);
  while (board.hasMoreMoves())
  {
    if(value >= beta)
      return value;
    if(value > alpha)
      alpha = value;
    move = board.getNextMove();
    nextBoard = board.makeMove(move);
    temp =  -PVS(nextBoard, -alpha-1, -alpha, depth-1);
    if(temp > value) // fail high
    {
       // check if temp value within bounds
      if(alpha < temp && temp < beta && depth > 2)
        value = -PVS(-beta, -temp, depth-1);
      }
      else
        value = temp;
  }
}
```

Pseudo source code for the Principle Variation Search algorithm

2.7 Iterative deepening (*)

Searching every branch of a tree until a certain depth using a depth first strategy is expensive and not flexible. E.g, the depth first methodology is not suitable for time-constraints. The algorithm can be stopped after a certain time has elapsed, but then possibly a complete branch of the tree is not evaluated and a bad result is very likely. Or the algorithm may be finished searching and have spare time that could give an advantage when it was used. In order to overcome these limitations another way of (selectively) extending the search can be used. This is called iterative deepening or progressive deepening, introduced by de Groot [17].

One of the advantages that can be gained by using iterative deepening is that at each level, moves can be re-ordered or parameters adjusted depending the current result. Also time limitation for a search can be easily implemented without the problems of depth first.

In the example code, adapted from Aske Plaat's website [32], the possibility to adjust a variable (firstguess) at each level. Stop the search when the time is up is also demonstrated.

```
int iterativeDeepening(ChessBoard board, int depth, int firstguess)
{
      for(d = 1; d <= depth; d++)
      {
            firstguess = searchMethod(board, firstguess, d);
            if timeUp()
                  break;
      }
      return firstguess;
}
```

Pseudo source code which uses iterative deepening

2.8 Best first search (*)

As the name of the algorithms suggests, this algorithms will follow the best alternative first. It does so by using the iterative deepening method, which gives the possibility to follow the best alternative found so far at each level of the tree.

At first, best first search algorithm complexity was different compared to the depth first (alpha-beta) algorithms, because they did not use transposition tables. This caused a lot of redundancy when for each level the previous search is repeated and continued one level deeper and slow move lists had to be used to overcome this problem. The authors of *"SSS* = α-β + TT"* [9], show that SSS (a best first search algorithm) can be implemented as an alpha-beta algorithm enhanced with a transposition table, giving roughly the same time complexity.

2.8.1 SSS*

As mentioned above, alpha-beta can be adapted to behave like a best first search using iterative deepening. SSS* uses a null window alpha beta search, with a window starting with upper bound of MATE. This window will cause a fail-high, giving a new upper bound. The continuous re-searches are fast, because of the transposition table.

The example code is adapted from [9] and should be called from an iterative deepening function (replace searchMethod by alphaBetaSSS in the iterative deepening algorithm of the previous section).

```
int alphaBetaSSS(ChessBoard board, int depth)
{
      int g = MATE;
      int w;
      do
      {
            w = g;
            g = alphaBetaTT(board, depth, w-1, 1);
      }while(g != w);
      return g;
}
```

Pseudo source code for the SSS algorithm*

2.8.2 MTD(f)

The SSS* algorithms can be extended not only an upper bound, but also a lower bound to change the boundaries of the window, as it is being done in aspiration search. Also g can be initialized by the previous value of g or another first guess [9]. The algorithm is named MTD(f) because it is **M**emory **T**able **D**riven using **f**irstguess.

Example code, again adapted from [9]:

```
int MTDf(ChessBoard board, int first, int depth)
{
  int g = first, beta;
  int upperbound = +MATE, lowerbound = -MATE;
  do
  {
    if(g == lowerbound)
      beta = g + 1;
    else
      beta = g;
    g = alphaBetaTT(board, beta - 1, beta, depth);
    if g < beta
      upperbound = g;
    else
      lowerbound = g;

  }while(lowerbound < upperbound);
  return g;

}
```

Pseudo source code for the MTDf algorithm

2.9 Forward pruning

Until now, only safe pruning mechanisms have been discussed. These methods prune leafs or complete branches of the search tree without information loss.

Forward pruning uses pruning mechanisms that prunes before anything about the leafs or branches has been actually evaluated. Most forward pruning methods do not return sound minimax values from the game tree.

In the standard forward pruning mechanism, moves are ordered and only the best N moves are followed. Usually N will be decreased in deeper levels of the tree. This type B strategy method and has proved not to be reliable. The major problem is the selectiveness of the search. When a key move at a low level in the tree is pruned, a blunder can be the result. Pruning at deeper levels of the tree is called razoring and was introduced by Birmingham and Kent in 1977 [22]. This is a widely used method, but the same basic problems will remain.

The authors of *Risk Management in Game-Tree Pruning* [16], discuss important aspects of forward pruning regarding risk management, applicability, cost-effectiveness and domain-independency.

2.9.1 Null Move heuristic (*)

Not making a move can be regarded as the worst thing that could be done in chess. Passing (not making a move, a null move) is not allowed in chess, but it is used to find if an alternative should be extended. At the level in the tree to decide to extend the search or to prune (usually at depth D-R-1, where D is the search depth and R the forward pruning height, usually 2), a so called null move is performed. Should the evaluation result of an evaluation of a board after the second move of the opponent be better than the upper bound of the search, the upper bound is returned and the the node at which the null move was done is pruned. The idea is that the opponent could find a good refutation to an actual move anyway, since the result is bigger than beta (the best minimax value possible).
The null move heuristic is not reliable at the final stages of the game, because here so called zugzwang positions could apply. A zugzwang position is a position where the player that moves first will loose (if they could, they would pass in such a situation).

2.9.2 ProbCut

ProbCut [20], is based on the idea that the result v' of a shallow search is a rough estimate of the result v of a deeper search. This is modeled with the following linear equation:

$$v = av' + b + e$$

Here, e is a normally distributed error variable with mean 0 and standard deviation σ. The variables a, b and σ can be computed with linear regression on the search results of thousands of positions [21]. Using this property in Alpha-Beta search, a branch can be pruned if v' gives v >= beta. Where the null move heuristic uses a pass to make the shallow search result safer, ProbCut uses the approximated value v' to guide a null window search for v.

Pseudo code adapted from [21]:

```
// using a probability check depth D, a cut-treshold of T and
// shallow search depth S

int probCutAlphaBeta(ChessBoard board, int alpha, int beta, int depth)
{
  if(depth == 0 || board.isEnded())
    return evaluate(board);
  if(depth == D)
  {
    int bound;
    bound = round((T * sigma + beta - b) / a);
    // is v >= beta likely?
    if(alphaBeta(board, bound-1, bound, S) >= bound)
      return beta;
    bound = round((-T * sigma + alpha - b) / a);
    // is v <= alpha likely?
    if(alphaBeta(board, bound+1, bound, S) <= bound)
```

```
        return alpha;
    }
    // rest of the alpha-beta algorithm
    ...
}
```

Pseudo source code for the PrbCut algorithm

2.10 Move ordering (*)

The alpha-beta algorithm complexity is dependent on the order of the moves. Cut-offs occur when good moves are followed first. Trying the move that produces the minimax value at each node, produces the minimal tree. The worst case results in the same complexity as a minimax search. The importance of a good move ordering mechanism is therefore essential.

2.10.1 Killer heuristic

Ordering moves can be done by using heuristics. Two methods commonly used are described by Gillogly, 1972 [23]. First he states that: “since the refutation of a bad move is often a capture, all captures are considered first in the tree, starting with the highest valued piece captured”. Secondly: ‘‘If a move is a refutation for one line, it may also refute another line, so it should be considered first if it appears in the legal move list”. This is done by keeping track of the minimax scores of moves while searching. In the next search, the moves with the highest scores will be explored first (when a legal move). This high scoring move is called a killer move. Algorithms that implement this idea usually store one or two moves per node in the tree and re-order the moves according to these moves.

2.10.2 Refutation tables

Transposition tables are large, so for systems with restricted memory, there is another way to re-use calculated results. A refutation table will hold the path of moves that lead to the minimax value of the previous search. For the next search, this “refutation path” will be followed first. This way, the property that the refutation of the move has a high probability to be the refutation of the next move is used (the second observation from Gillogly in the previous section about the killer heuristic).

2.10.3 History heuristic

Another more general method from Schaeffer [25] is to keep a history table of the moves, that keeps track of the number of times a move has been a successful refutation instead of a two move/score pair in the case of the killer heuristic. The most frequent moves should be tried first, regardless of the current position. This way, the information about the effectiveness of the moves is shared along the complete tree, instead of only between nodes at the same level in the tree.

2.11 Quiescent search

When a search stops at a certain depth, the position reached may contain a situation where a piece is about to be captured, a king about to be checked or a pawn about to be promoted. Evaluating such a situation gives unreliable results, because the next position may contain a big advantage to the other

player after that capture, check or promotion. On the other hand, "dead" positions – positions that have no active moves such as hit, check or promotion, are "relatively quiescent" and give a more reliable score. This reliability issue is a consequence of the correlation between the number and quality of the pieces and winning the game [30]. In a dead position, the chance that number and quality changes (after hit or promotion) is smaller – it might happen after the next move, but not in this position. Because the score at a quiescent position is not reliable by this definition, a further continuation of the search is needed.

A quiescent search intends to find the moves that complicate a position (there are situations thinkable that hit, check or promotion are not the only active moves). Different strategies exist to deal with further exploration of the tree because exploring all moves is expensive and search time is limited. A common strategy is to only explore all captures in the case of a capture, covering an exchange of pieces. Following all quiescent moves until "dead" positions are reached, only reveals a restricted future because not all moves are followed. This results in an incomplete (and therefore still incorrect) evaluation. But it will result in a more reliable score concerning the number and quality of the pieces on the board.

2.12 Horizon effect

The depth at which a search stops, is the horizon of the search. Any continuation that happens after the search has stopped is beyond this horizon. It can be seen as a more general case of a quiescent position, where danger may be hidden at deeper search depths. A perfectly acceptable result from a certain depth, may lead a player into a dangerous branch of the tree, where all nodes behind the horizon lead to losing the game.

An example of the horizon effect, from Kaindl [6]:

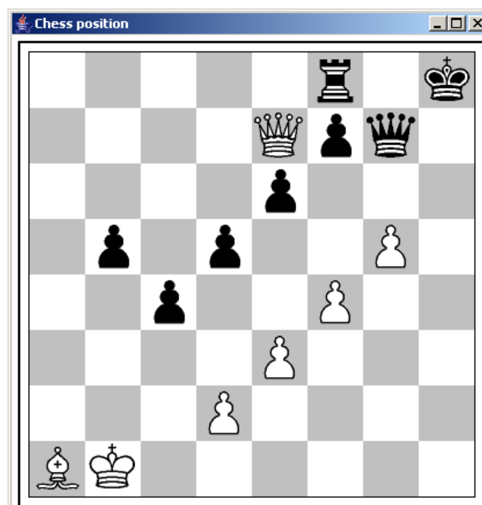


Figure 4: Horizon effect

In the example diagram, it is blacks turn – the search has ended, but a quiescent situation is discovered. In order to "save" the queen, even a 10 ply deep search would not rate the queen as lost because yet another pawn can be put in the line of the bishops attack. The situation for black is completely lost –

even though black has a rook in stead of the white bishop, and more pawns. This shows the horizon effect problem, which, in this case could not be solved by an extra quiescent search of 10 plies.

2.13 Diminishing returns

The strength of chess programs is strongly related to the search depth. Experiments show that a program playing at depth D+1 will win approximately 80% of games played against the same program playing at depth D [1].

The point where an additional ply does not yield better moves is called the diminishing return. In their conclusion, the authors of *"Diminishing returns for additional search in chess"* [1] state that diminishing returns for chess can be found, but in order to do so, the length of the games must be limited. For lengthy games the high error rate of moves has too much influence (every extra move another possibility for an error). They also found that the effect is hidden because it starts at high search depths – around 9 plies.

2.14 Search methods measured

The complexity of the above search algorithms is hard to asses mathematically. It has been done for minimax and alpha-beta, but these results are not practically usable. Minimax is inefficient and the results for alpha-beta have been generated using random numbers in trees, which turns out to be a bad comparison to chess trees in practice. As a result, usually Alpha-Beta is used as a reference method when the efficiency of a search algorithm is measured. The methods and results of other research will be used in this thesis to define the tests for SPP Alpha-Beta.

A research that is representable in this context is *"The History Heuristic and Alpha-Beta Search Enhancements in Practice"* by Schaeffer [13]. It gives an overview of measurements of experiments on a variety of search algorithms. In his research, single and dual enhancements to the alpha-beta algorithm are made. In his results, he uses both NBP (node bottom positions, leaves) and NC (node count) as a measurement of performance. He argues that these numbers give a good measurement of performance, because NC has a good correlation with CPU time, whereas CPU time alone depends too much on implementation and platform. Since NBP is a measurement used in other research as well, he also provides this number. A summary of his results is given, starting with a list of the enhancements used.

- Killer move
- History heuristic
- Aspiration search
- Null move
- Transposition table
- Refutation table

To get a number for the reduction percentage he uses the formula (no = no enhancement, enh = enhanced, best = best enhancement):

$$reduction = \frac{(N_{no} - N_{enh})}{(N_{no} - N_{best})} 100$$

As a single enhancement, *Null move* and *Aspiration* only contribute to a small improvement on Alpha-Beta. *History heuristic* shows a good enhancement until depth 7, where after the number of varieties flood the table and show a drop in performance. The improvement of a *Transposition table* grows with the depth, provided that enough memory is available. The *Killer move* shows a constant improvement but smaller than *History heuristic*.

In combination as a dual enhancement *History heuristic* and *Transposition table* shows by far to give the best performance, giving a reduction to 99% at high depths, while the others stay behind with insignificant numbers.

History heuristic and Killer move even have a negative influence when they are combined, compared to single use of any of the two. This is because both methods have influence on the move ordering and disagreeing in some cases, giving a reduction in expected performance.

3 Sibling Prediction Pruning Alpha-Beta

In *An Analysis of Forward Pruning* [26], Stephen Smith and Dana Nau conclude that forward pruning on minimax trees is most effective when there is a high correlation between sibling values in the tree. This thesis explores that idea further for chess, and tries to identify positions that have a high correlation in their evaluated values. In this chapter, properties of the evaluation result of sibling positions will be discussed. Where these properties can be used is the key to the Sibling Prediction Pruning Alpha-Beta algorithm.

In this chapter, the properties of evaluating chess board positions will be discussed. We will focus on board positions that originate from the same parent position (siblings).

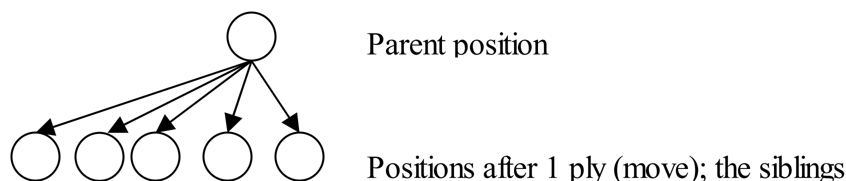


Figure 5: Siblings

Because all positions are the result of a move on the parent position, it is likely that the positions share properties because:

- Only one piece will be re-located (in the case of castling two).
- All other pieces keep their previous locations on the board, except when a piece has been hit.

In this chapter, the positions under investigation are the positions that are the result of one non hitting, non checking and non promoting move, and will be referred to as siblings or sibling positions with *spatial locality*. The reason that the "active" moves are left out is because they can have a big influence in the evaluation result of the resulting positions. This will become clear in the next section about the evaluation function.

The main question is: how does a move affect the evaluation result of sibling positions. What properties are reflected in the evaluation results of positions with *spatial locality* and when can they be used. In order to answer these questions, the evaluation function itself will be analyzed next.

3.1 Evaluation function

For any position under investigation, the evaluation function either returns a final status (win, loss or draw), or returns the summation of a weighted feature vector. If the feature exists in the position, it will be added with its specified weight to the summation.

In mathematical terms, the result of the evaluation function can be defined as:

$$value = \sum_{i=1}^{n} w_i f_i \mid w \in \mathbb{Q},\ f \in (0,1)$$

In this term, the evaluation function knows n features with their corresponding weights. In the case that the feature exists on the board, it is equal to 1 and the weight is added to the sum. In the other case it is equal to 0 and the weight has no influence.

In this thesis, a distinction between the material balance and positional balance of a position is made. This is because the weights of the material (piece) features are dominant in most evaluation functions, giving possible big differences in evaluations of positions with different material balances. By definition, the sibling positions with *spatial locality* all have the same material balance because hits and promotions (moves that have influence on the material balance) are specifically left out.

When comparing two positions, that is, values from an evaluation function, three cases can be identified:

1. The material balance is decisive. The material difference is bigger than the positional difference.
2. There is a zero material balance and positional features make the difference.
3. The difference in positional quality is bigger than the material difference. Some material is compensated by a positional advantage.

The three cases are depicted in the following figure.

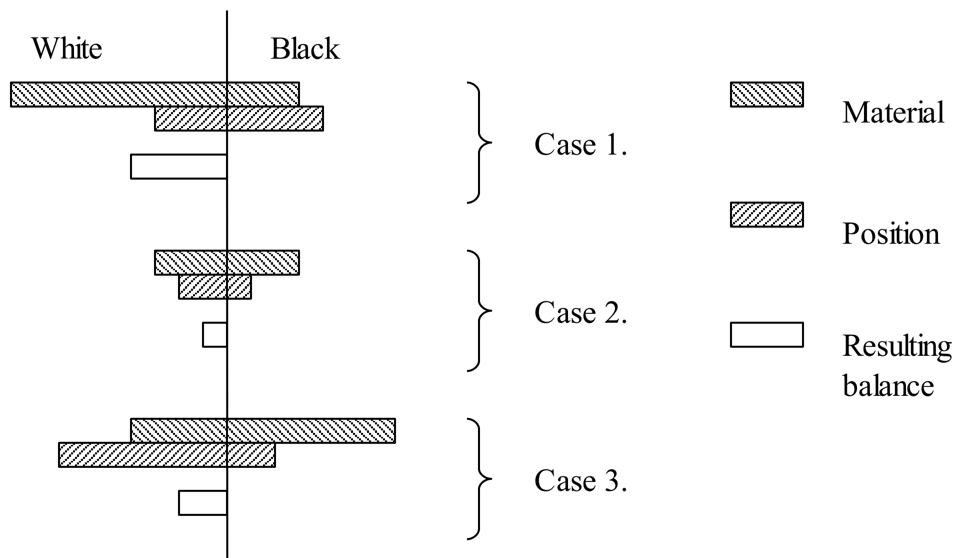


Figure 6: Three evaluation cases

With an equal material balance, such as in case 2, it is clear that the positional balance is decisive. In case 3, the positional balance must count for more than the value of a pawn in order to be different from case 1. When the positional balance is bigger than a pawn (the smallest piece in value), it is

possible that material will be compensated by position. The distinction of these three cases is made because it identifies the importance of the positional balance for siblings with *spatial locality*.

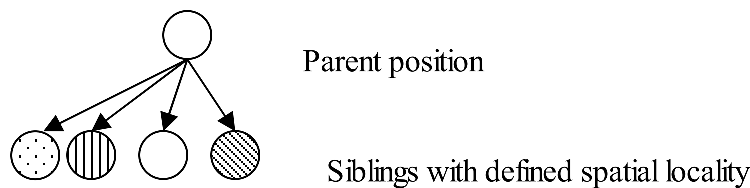
For sibling positions with *spatial locality*, the material balance is fixed. If a maximum difference of positional features can be found between these siblings, this can be used as a powerful prediction method to predict the maximum value of these siblings. If the positional balance plus this maximum positional difference is less than the best value found so far, all remaining brothers of this position can be pruned because they cannot reach a value bigger than the best value found so far.

3.2 Maximum difference

For both material and positional heuristics, a maximum difference between siblings can be identified. This is an important property that will be used in the search algorithm.

For the material part this maximum difference is easy: the highest capture possible. But captures (and promotions) are not taken into account.

Positionally, one (half) move also must have a maximum difference possible between the resulting positions. Important here is that captures are ignored because they can have big impact on positional properties (a piece with a big positional influence might be hit). The following figure illustrates possible values of sibling positions.



Two possible spreadings of evaluation results of siblings on a value axe:

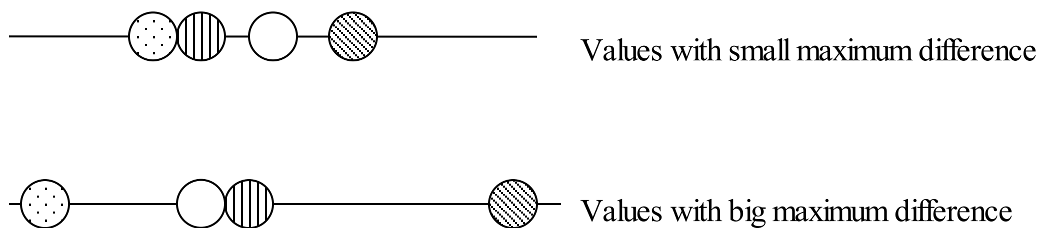


Figure 7: Possible spreadings of sibling evaluation results

The maximum difference must be the result of one piece relocation. As a result of a piece relocation the properties on the board change. New features can be created, others may disappear – a new permutation of the feature vector will emerge.

Only one piece is moved and the positional features of other pieces will ensure a certain correlation between the evaluation results of the siblings because most features (existing or non-existing) will remain unchanged in the feature vector.

With weights scaled such that they range from 0 to x, in stead of -y ... +y, with x = 2y, the maximum theoretical difference between two positions is the result of the two feature vectors, one with all f_i =0, the other with all 1's:

$$(0,0,0,....0) \quad and \quad (1,1,1,...1)$$

Giving a maximum difference of the sum of weights:

$$\sum_{i=1}^{n} w_i \ | \ 0<w_i<x$$

It is unlikely that the difference in the permutations of the feature vectors result in such a summation because of the features of pieces that remain on their positions. Reasoning from the parent position, it is possible for any of the features to determine if it will possibly occur in one of its siblings by simply evaluating every sibling. The maximum difference between siblings with given parent position can be determined: it is the the result of the summation of two siblings that share the least features. Keeping track of the features that change from parent to child is a complex and doing this runtime costs processing time. It might still be feasible because time is won by pruning bad positions, but for SPP another approach is used.

The approach used in SPP, is to define a maximum positional difference beforehand. This saves processing time, but will be less precise. A maximum difference can be found by analyzing the evaluation function by hand, checking the effect of a feature that changes due to a move. Another, more pragmatic method is to use a test set of positions to automatically search for a maximum difference. The positional balance difference of all sibling positions encountered in the search is administrated, leaving a maximum difference when the test is finished.

The maximum difference between two sibling positions depends on the kind of features and weights that are used within the evaluation function. It could be possible that for two given positions, one evaluation function will give a bigger difference than another evaluation function.

For this research, an automatic search is done to find a maximum difference beforehand and this method will be used further in this thesis. The reason that determining the maximum positional difference at runtime is not used is the following: It makes it hard to compare to another algorithm. The time spend for prediction cannot be ignored in the results because it depends on the evaluation function used, making it extra hard to proof that the algorithm can be more efficient in general. In the general case it then must be clear that the calculation time for prediction will be smaller than the time spent in evaluating the positions that were pruned. This is not obvious because of implementation dependencies. Also, the implementation of an efficient method that determines the maximum difference of siblings given a parent position is complex, and scalability in the case of expanding the evaluation function can therefore expected to be difficult. The “keep it safe and simple” principle is regarded more promising.

3.3 SPP Alpha-Beta

The maximum positional difference (MPD) between siblings is the property that will be used by SPP Alpha-Beta.

The algorithm is constructed as an ordinary alpha-beta with alpha and beta as lower and upper boundaries. Move ordering will ensure that any hits, promotions or checking moves will be evaluated first at the deepest level of the tree. This way, the siblings with defined properties remain. If the first of these remaining positions gives a score lower than alpha or lower than the best value found so far, a check is performed whether any of its brothers can get a value bigger than alpha or the best value. If not, all its brothers are pruned.

In the following figure, 3 leafs that are evaluated are drawn. With a MPD of 7, the remaining siblings with *spatial locality* can be pruned.

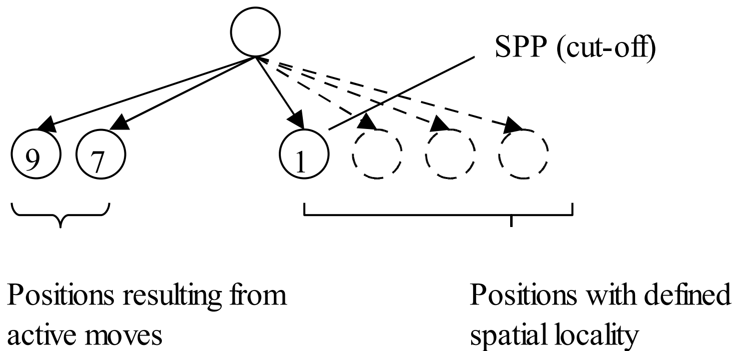


Figure 8: SPP cut-off

In the figure, the first two leafs are the result of a hit, promotion or checking move. The 3rd leaf is the first of the remaining siblings with *spatial locality* properties. The maximum value any of its brothers can obtain is lower than the the best value found so far ($1+7 < 9$). As a result, the leafs with dotted lines can be pruned.

A pseudo code example, adapted from the source code used in the experiments:

```
int SPPAlphaBeta(ChessBoard board, int depth, int alpha, int beta)
{
  int value = 0;
  if(board.isEnded())
    return evaluate(board);
  int best = -MATE-1;
  int move; ChessBoard nextBoard;

  board.getOrderedMoves();
  while (board.hasMoreMoves())
  {
    move = board.getNextMove();
    nextBoard = board.makeMove(move);
    if(depth == 1)
    {
      // negamax principle: value is negative of evaluation nextBoard
      value = - evaluate(nextBoard);
      if(Move.isHIT(move) || Move.isCheck(move) || Move.isPromotion(move))
      {
```

```
        // active move gets usual treatment
        if(value > best)
          best = value;
      }
      else
      {
        // passive move - sibling with spatial locality
        if(value > best)
          best = value;
        else if((best > value + MPD) || (alpha > value + MPD))
          break;  // prune (break from loop; done)
      }
    }
    else
    {
      value = -SPPAlphaBeta(nextBoard, depth-1,-beta,-alpha);
      if(value > best)
        best = value;
    }
    if(best > alpha)
      alpha = best;
    if(best >= beta)
      break;
  }
  return best;
}
```

Pseudo source code for the SPP Alpha-Beta algorithm

3.3.1 Correctness

For any method that uses pruning, correctness is at stake, because a position may be pruned that could give a different minimax value. With SPP, pruning of mates cannot occur, because checking moves are always followed and evaluated first. The other final game status, a draw, may be outside the maximum predicted values as it may not be the result of a maximum positional difference, but the result of repetition (e.g. 25 move rule).

A draw may be the result of a passive (non hitting non checking) move and can therefore be one of the siblings sharing *spatial locality*. As a result, it may happen that a draw is pruned, while this is possibly a good result. The draws that can be pruned by SPP are draws resulting from repetition or stalemates. A draw in material, where none of the sides can win, will be detected because when a certain material balance is reached, this is always the result of a hit.

- For a stalemate, the next search will reveal this final leaf that was pruned in the previous search and can probably be avoided when the search depth is not too shallow. The deeper a search, the more possibilities remain to avoid the stalemate in the next search.
- For draw due to repetition (3 identical positions due to move repetition or 25 move rule), the draw will also be detected in the next search and can be avoided just as it could have been in the previous search. In this case, the draw can also be detected by checking the parent position, where the repetition counter will yield a possible draw. Then the MPD can then be set such that it will catch the draw.

The stalemates need to be investigated further because positionally a stalemate is very close to a normal

mate. When the search depth is deep enough, the stalemates will not be a problem, unless a forced stalemate combination deeper than the search depth is at stake. These situations will be rare, but the user of SPP must be aware of this problem. A possible way to detect the stalemates, may be to keep track of the number of possible moves of the opponent in the parent of the parent position. This number must be low in the case of a stalemate because after the next move, no moves remain. This is not investigated any further for this thesis.

3.4 Expected behavior

In this section the expected behavior of the SPP Alpha-Beta algorithm is described: why it is expected to work. The issues will be taken into consideration for the test setup.

It is expected that SPP Alpha-Beta will never be less efficient than Alpha-Beta, because in the worst case scenario for SPP, no pruning can be done and the same algorithm as Alpha-Beta is followed.

The algorithm uses forward pruning, which means that correctness of the results is a concern. It is expected that when the MPD is correct, the pruning can be done safely without information loss because the maximum value of siblings is correctly predicted. Positions can only be pruned without information loss if their value does not affect the minimax value.

Pruning of siblings is expected when the MPD is not too big. Alpha and the best value found so far will then have values bigger than the maximum sibling value when examining certain branches in the tree. This is expected because chess has the property that only a few moves at the root are good alternatives [1], leaving a greater distance than the MPD between the best position found so far and a position in another branch.

The material balance is dominant in evaluation of chess. When comparing siblings of different branches of the tree with different material balances, pruning can be expected if the MPD is smaller than the value of a pawn (the smallest material imbalance). A MPD is the summation of positional features alone: among siblings it is believed that a difference of the value of a pawn is too much. This is however dependent on the evaluation function.

3.4.1 Evaluation function and MPD

A small MPD probably causes more pruning used in combination with the same evaluation function compared to a big MPD, simply because the possible maximum sibling value is smaller, leaving a greater change to be smaller than the best value or alpha. This is made explicit in the following case scenario:

```
best or alpha = 24
evaluation result of first sibling = 10
MPD <=13: prune brothers
```

Does another evaluation function with a bigger MPD (>=14) cause less pruning? This is not straightforward. Taking the positions of the previous case scenario, the other evaluation function results are likely to be different, e.g.:

```
best or alpha = 30
evaluation result of first sibling = 12
```

```
MPD <= 18: prune brothers
```

Even though the MPD is bigger, this position still causes pruning when it is smaller than 19.

The spreading of evaluation results of siblings is depends on the evaluation function. Also the overlapping of evaluation results of different positions influences pruning. This is depicted in the following figure.

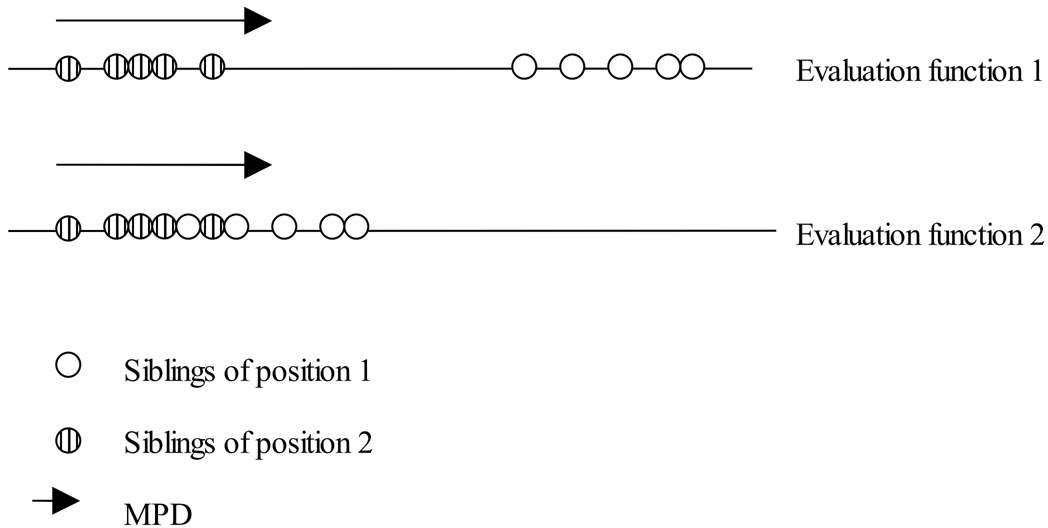


Figure 9: Evaluation functions and overlapping

Here, position 1 could be the parent of the siblings containing the best position found so far, and position 2 a position which siblings are investigated. For evaluation function 1, it is clear that the remaining siblings can be pruned after investigating any of the siblings of position 2. The MPD, or rather the overlapping of the results, of evaluation function 2 prevents pruning.

Not only the size of the MPD, but also the overlapping of evaluation results in different branches of the tree is of influence for SPP. This is hard to predict for a given evaluation function.

3.4.1 Move ordering

Another practical issue is the effect of the move ordering on the overall pruning. The move ordering for SPP is simple. Killer moves or other heuristics for move ordering can still be used, and they usually share hit, check and promotion preference. Apart from that, for non iterative deepening methods, only the specific move ordering of checks, hits and promotions first, need to be done at the deepest level of the tree.

Because move ordering plays a role in Alpha-Beta pruning, a fair comparison is hard when the move

ordering of the algorithms being compared are different. Since the move ordering is practical and overlapping with regular move ordering, this is not considered an issue.

3.4.2 Stacking

It is also expected that the combined (stacked) use of other enhancements such as transposition table and or aspiration window will decrease the effect of this method, because these are also pruning methods, leaving less positions to prune for SPP which is done at the deepest level. The enhancement with transposition table is hard to predict, because it is expected that SPP pruning causes less storage of positions and their corresponding values (a pruned leaf is not stored). This might behave contra productive because then they compete for pruning.

3.4.3 Balance and material

The best behavior is expected when the game is balanced – the evaluation result of a board position is near zero. When a position is materially balanced, a smaller positional imbalance can be expected because the opponents features will balance the score. The smaller the positional imbalance, the smaller the maximum sibling value can become as a result of the MPD, causing more cut-offs. Also the more pieces are on the board, the more pruning can be expected, in combination with the material balance property. With less pieces on the board, the positional influence per piece becomes bigger. The positional balance will then fluctuate more, causing less pruning of siblings with *spatial locality* because the maximum sibling value depends on the positional balance.

4 Test setup

The conceptual work has been done and empirical test results must proof if the principle works in practice. The chess search space is incomprehensibly big (~10^{120} leafs on the complete game tree). It will be clear that only a small (possibly representable) subset of this search space can be taken to use for measurements.

Somewhat the same kind of problem exists for the evaluation function because the number of features and their weights can be endlessly configured.

This thesis is intended as a proof of concept for SPP Alpha-Beta. Therefore, proving correctness of the results is the most important objective. Correctness alone is not enough, the efficiency of the SPP enhancement must be measured in some way.

As a result of the discussion on the expected behavior in the previous section and the objectives, the following considerations are taken for the test setup:

- Which search algorithms are used for comparison.
- Which evaluation function is used.
- Which positions are used in the test set

- Correctness of the results
- Which measurements on the algorithm should be taken
- Testing environment & software used

Every item on this list is handled in a separate section of this chapter, starting with the choice for a search algorithm.

4.1 Search algorithms for comparison

For this research, the standard Alpha-Beta algorithm is used as the algorithm to compare the other algorithms with. It is the base algorithm of many search algorithms including SPP and is therefore a good representative for comparing complexity. Also because the obvious reason that it needs to be shown that SPP Alpha-Beta is more efficient than Alpha-Beta.

One of the issues in the discussion about the expected behavior applies to the stacking of multiple enhancements. As a consequence, also measurements of Alpha-Beta enhanced with transposition table and SPP Alpha-Beta with transposition table will be taken and compared.

4.2 Evaluation function dependencies

The results of SPP enhancement to Alpha-Beta depends on the properties of the evaluation function because the evaluation function defines the MPD for sibling positions and the best value to beat.

Usually the only dependency of the evaluation function on performance of a search algorithm that needs to be considered is execution time (complexity of the evaluation function). The difference between SPP Alpha-Beta and Alpha-Beta in execution time for visiting a node is can be neglected because it only involves an if statement containing the summation of the evaluation value with the MPD to check if it is outside the range of alpha or the best value found so far.

The previous chapter about the expected behavior indicates the dependency on the evaluation function, but also that it is not straightforward what kind of relation can be expected between MPD and pruning. As a consequence, more evaluation functions could be tested and the differences interpreted. For this research, only one evaluation function will be used in the tests. The intention is to proof that SPP Alpha-Beta works.

It will be clear that using an evaluation function that only sums the material value on the board, will have a MPD of 0, because no positional features weigh in the sum. The evaluation function taken, must therefore have positional features, preferably represented in existing chess programs.

The following features and weights are selected for the evaluation function.

Feature	***Weight***
pawn	100
knight	300
bishop	300
rook	600

Feature	*Weight*
queen	1000
Center square attacked	2
Square next to opponent king attacked	2
Attacked square	1
Opponent piece attacked	1
Doubled pawn	-10
Passed pawn	10
Castled king	10
Early queen	-30
Central knight	5
Developed knight	2
Bishop range	2
Rook on 7th rank	10
Rook on open file	10

Most of these features are common and used in chess programs such as CRAFTY [30], gnu-chess [31] and Beowulf [4].

Features and weights used for end-game evaluation (after gnu-chess [31]).

- King+mating material against king. 150 – value of square opponent king can be mated on. Corner squares are 0, center squares up to 48. The evaluation function favors moves that push the opponent king to a corner square such that it can be mated.
- A KBNK (king, bishop and knight against king) routine. Weights for positions range from 0 to 70.

4.3 The test set

The test set contains 16 chess positions. Each position is used as a starting point for the search algorithm that is tested. The search algorithm will search the best move for that given position and a given search depth, and return the move score pair. Every position will be searched to different search depths, ranging from depth 4 to depth 8. Depths 1 to 3 are left out. For the algorithms with transposition tables, only depths beyond 4 plies are of interest, because transpositions only occur after 4 plies.

4.3.1 The positions

The first position is a chess problem that contains a mate in 6 plies. The idea behind this is that when picking positions, at least a clear winning position should be taken.

Positions 2 to 13 are positions taken from the Bratko Kopec test set [29]. This is a set of 24 positions,

used in much research on evaluation function and search algorithm efficiency [6, 7, 13]. The 12 selected positions are relatively balanced positions and are taken from just after the opening (openings itself are of no interest since these are covered by opening databases in chess computers), in the mid and mid to end game phase. The reason that 12 out of 16 positions are taken from the mid game, is because at this stage, the most exponentially growing game trees can be observed – where efficiency in pruning will pay off well.

Positions 14, 15 and 16 are end-game positions. Position 14 only has the two kings, two bishops and one side with more pawns. Position 15 is a KPK, 16 a KRK position.

The positions are not picked randomly. Intuitively this is not a good characteristic for testing, because of favoritism. On the other hand, randomly picking positions form a space of 10^{120} positions is another problem. The Bratko Kopec positions have been designed for testing and have been used as such. The added positions (1, 14, 15 and 16) are not expected to behave in favor of SPP.

An overview of the used positions can be found in Appendix B.

4.4 Measurements

In most research on search algorithms, the NBP (node bottom positions) count is used to indicate performance, because this measurement usually correlates with execution time [8, 13, 14]. In the benchmark for this research, NBP is measured because there is a direct linear relation to execution time. Execution time is not a good indication of what to expect on another system because of dependencies on implementation and environment. Execution time has only been measured on a single system to ensure relation to NBP count. The benchmarks were run as batches on workstations with an unknown workload, so execution time has been discarded from the test results.

Node count (NC) is used in other research because it also correlates with execution time. Because SPP only occurs at the deepest level of the tree, the algorithm has no influence on NC and is therefore omitted.

The efficiency of SPP Alpha-Beta is expected to come from extra pruning at the deepest level of the tree. This is why NBP count alone is sufficient for comparison with the other search algorithms. The pruning of Alpha-Beta is of influence on SPP, but since it always occurs before SPP, the percentage of pruning as a result of SPP can be safely determined. For the comparison between SPP Alpha-Beta with transposition table, this is not the case. SPP pruning is expected to result in fewer storage of values in the transposition table. In this case, only the pruning percentage can be determined, but not which enhancement contributed to it in what scale.

4.5 Correctness

In the case of a forward pruning method, correctness is at stake because pruning is done on non-investigated leafs. Correctness will be checked by comparing all move score pairs to the standard alpha-beta results. The same move ordering for both algorithms is used, so the path through the tree is the same. Therefore the results must also be the same. This is not true for all pruning mechanisms, where a branch with the same minimax value might be pruned, giving a different move. This is not the case with SPP, because pruning only occurs at the deepest level of the tree.

4.6 Testing environment & software used

The chess application is developed with development platform Eclipse [7], using programming language Java. The data structure for the board and pieces is adapted from the source code of gnu-chess [31]. The Java source of the application contains 38 classes with a total of 329 methods implemented using 5050 NCLOC (non commented lines of code). The code has been organized such that a search algorithm is an object, making it run-time configurable which search algorithm is used.

For the benchmark, the so called "ChessProblem" object is used, that contains one of the 16 positions. For every depth (from 4 to 9), a vector of 16 ChessProblem objects is iterated, starting the search on every ChessProblem and administrating the search result: the move score pair and the number of calls to the evaluation function to determine NBP count. The output is tab delimited text, which is imported into excel to create diagrams of NBP count and the resulting pruning percentages of the two algorithms. The list of move score pairs is compared to the list of the other algorithms to test correctness.

The MPD used in SPP has been determined by self play. The program uses a random opening database, and 10 different games were played. The MPD turned out to be 70, which is not a surprise when looking at the feature-weight table, where 70 is the biggest weight of a feature in certain end-games.

5 Results

In this chapter, the results of the tests with Alpha-Beta, SPP Alpha-Beta, SPP Alpha-Beta with Transposition table and Alpha-Beta with Transposition Table on the test set are presented.

The results for even and odd depths are presented separately in most cases. It is common practice in most publications. This is because a search on odd depths is cheaper than on even depths. The incremental cost of growing the tree an additional ply to an odd depth is greater than for an even depth for Alpha-Beta based algorithms [6, 13].

All diagrams containing measurements from the test are presented in Appendix A, for readability and a better overview.

Abbreviations used for the methods:

Abbreviation	***Method***
AB	Alpha-Beta
SPP	SPP Alpha-Beta
SPPT	SPP Alpha-Beta with Transposition Table
ABT	Alpha-Beta with Transposition Table

The following convention is used to indicate the percentage of extra pruning of algorithm A1 vs algorithm A2: **A1/A2**. This percentage is calculated as follows:

$$100 - \frac{NBP(A1)}{NBP(A2)} 100$$

Here, NBP(A1) is the NBP count of algorithm A1.

5.1 Exceptional results

This section is used to discuss exceptional results. This is done preliminary to the actual presentation and interpretation of the results, because the exceptional results are observed for all depths and all algorithms. Exceptional results are observed for positions 14, 15 and 16 of the test set, which are end game positions with only a few pieces on the board.

The NBP count of all algorithms for the selected depths is low for these positions. This is because the branching factor of the game tree is relatively low for positions with a few pieces.

Positions 15 and 16 are end game positions where one side only has its king. The other side has one extra piece. The consequence is that there are almost no capture possibilities. A king cannot be captured, and if the one remaining piece of the opponent is captured, an immediate draw follows. For SPP, this means that all siblings have *spatial locality* and no pruning can be done because the best value found so far only differs in positional quality, not in material quality.

Position 14 is an end game that needs a sacrifice to be solved. It turns out that all search depths are too shallow to solve the problem. Therefore, any continuation where a piece is lost, is seen as a bad continuation. As a result, again, only positional differences in quality can be taken to find the minimax value – loosing a lot of SPP. From position 14, also the big difference between searching at on odd or even depth can be observed. In this regard, the results of position 14 are not exceptional, but it must be noted that at even depths less to no SPP is observed in comparison to odd depth searches. This is because the refutation of a move at an odd depth in this particular case will almost always be a capture. The number of possibilities of the opponent are limited, but do contain captures.

With the use of a transposition table, positions 14, 15 and 16 also give exceptional results. This is due to the fact that when only a few pieces are on the board, the number of transpositions in comparison to the branching factor is high. This is normal behavior for transposition table based algorithms.

5.2 Presentation & Interpretation of the results

The results are presented pairwise, comparing the results of two algorithms per case. This is done to make it easier to observe the differences in efficiency due to the enhancements. Comparisons are made between:

- AB and SPP, in order to find the efficiency of sibling prediction pruning
- AB and ABT, in order to find the pruning effect of a transposition table
- SPPT and SPP, in order to find the effect of transposition table as stacked enhancement
- SPPT and ABT, to compare with SPPT and SPP to find negative effects

5.2.1 AB vs SPP

In the results, the NBP count of SPP is always less or equal compared to the NBP count of AB. This indicates that SPP is never less efficient than AB. This is an expected result because when the SPP algorithm cannot prune, the same instructions as for the AB algorithm are followed.

In diagram 7 and 8, it can be seen that for positions 15 and 16, SPP has the the same NBP count as AB for all depths. The reason that no extra pruning occurs, is explained in the previous section about exceptional results.

For even depths, the results give different pruning percentages (diagram 7), in comparison to odd depths (diagram 8). This shows the big difference in NBP count for searching on even and odd depths . Specially in combination with SPP, where expanding the tree with an active move is much cheaper on odd depths. Any hit will cause all its brothers to be pruned because the MPD is smaller than the value of a pawn. The resulting pruning percentages for odd depths range between 60% and 85%, except for positions 15 and 16. The pruning percentages on even depths are much more spread and lower than on odd depths. Chess computers usually use odd depths on Alpha-Beta based search algorithms, because this is more efficient. This is observed in other research [6, 13] and is certainly true for SPP. Except for the noted positions (15 and 16), the percentages do not fluctuate much on odd depths and are greater than on even depths.

The bad pruning percentages for positions 14, 15 and 16 show that there are positions that are not suitable for SPP pruning – some only at even depths (14). The effectiveness of SPP for the other positions is clearly visible and is between 60% and 80% with speedups between 2.5 and 5 respectively.

5.2.2 ABT vs AB

In order to interpret the difference in NBP for methods SPP and SPPT, AB and ABT are also compared. The transposition table enhancement improves over the search depth. This due to the caching of the results of transpositions and is also observed in other research [13].

In diagram 9, it can be seen that the extra pruning by the use of a transposition table, gives percentages that are spread in a range from 13% to 98%. The lower percentages can be observed at depth 4 and the high percentages at depth 9. The highest percentages can be seen on positions 14, 15 and 16. These positions have more transpositions in relation to their branching factor – as mentioned in the discussion of exceptional results.

5.2.3 SPPT vs SPP

Diagram 10 shows the extra pruning percentage that SPPT has over SPP. The extra pruning from the use of the transposition table gets better at higher depths. The diagram shows a similarity to diagram 9 from ABT vs AB, but percentages are shifted down. At higher depths, the similarity is the greatest, while at depth 4 almost no extra efficiency due to the transposition table is shown.

Especially at positions 14, 15 and 16 a lot of transpositions can be expected because of the few pieces in these end-game positions. The effectiveness of a transposition table however is much smaller than expected when comparing to AB vs ABT. This shows that the algorithms compete for pruning, giving a negative effect when used in combination. This effect will be further elaborated in the next section about SPPT vs ABT.

5.2.4 SPPT vs ABT

For this comparison, just as is done for SPP vs AB, the measurements are presented separately for even and odd depths. Diagrams 11 and 12 show the extra pruning percentage of SPPT over ABT for even and odd depths respectively.

The behavior is compared to that of SPP vs AB (diagrams 7 and 8). This is made visible in diagrams 13 and 14, where the stacked percentages of SPPT/ABT and SPP/AB are drawn. For even depths the biggest differences can be observed, whereas for odd depths the stacked percentages are more equal (around 50%) for most positions. For positions 14, 15 and 16 the negative effect of combining SPP and transposition table is clearly visible. The negative percentages of these positions show the inefficiency of SPPT compared to ABT.

All positive stacked percentages of SPP/AB in diagrams 13 and 14 (where SPP is efficient), are all above 50%, which indicates that for all these cases SPP and Transposition Table compete for pruning, because the efficiency of SPPT over ABT is smaller. The benefit of combining SPP and Transposition Table still pays off for these positions, but for even depths this is less stable.

5.3 Correctness

The move score pair lists of all used methods are identical. This is expected, because all methods use the same move ordering. It can be concluded that there is no information loss in the SPP algorithms for the test set used.

6 Conclusions

The proof of concept on SPP is a success. It is a new forward pruning method without information loss regarding to the minimax value of the game tree. This is an important property, because information loss is the down-side of other forward pruning mechanisms. The efficiency ranges from 1 to 5 times as efficient as Alpha-Beta on the test set.

The prediction mechanism depends on the MPD of the evaluation function. As such, it can be stated that the MPD contains some of the chess knowledge that is part of the evaluation function and that this knowledge is used to guide the search. In that sense, SPP is remotely related to ProbCut, which uses the heuristic that a shallow search predicts the evaluation result of a deeper search with a certain probability. SPP only predicts evaluation results of passive moves and does so without information loss: probability 1. ProbCut has no move selectiveness and cannot guarantee a search without information loss.

SPP enhanced algorithms should be used with odd search depths because the test results clearly show more efficiency at odd depths. SPP has been specially designed for the use in chess programs, and uses properties such as material dominance from the evaluation function for prediction pruning. SPP can be used on other games, but a certain spatial locality in evaluation results must be identified in order for the algorithm to work. The efficiency then strongly depends on the distinctiveness of the branches (for chess the number of good moves is low, giving excellent pruning perspectives). Efficiency and correctness must be shown by testing because for any complex game (such as Chinese chess or another

candidate), the behavior is hard to predict.

With the evaluation function chosen and the test set taken, SPP is a good enhancement. Tests with different weights must be done to learn more about the method; when to use it and when not to use it. Also the dependency on the starting position of the search plays a role. SPP does not work well on end-game positions because at these positions the evaluation function is tuned for position rather than material. It must be concluded that the use of SPP not efficient in all cases. This is not regarded as a problem because SPP is most efficient on positions with a high branching factor – the area that benefits most from pruning.

The extra enhancement of a transposition table competes for pruning with SPP. On positions where SPP does not work, this even gives a negative effect. It is possible however to turn SPP on and off, depending on its pruning efficiency. A counter can be used to see if any prediction pruning occurs. When this is not the case, SPP can be turned off completely. This may overcome the problem of inefficient competition between SPP and transposition table usage.

SPP can be used as a replacement for ABT on a system with not enough or slow memory, such as a mobile phone. As a single enhancement it is more efficient than ABT for positions with a high branching factor.

7 Future investigation

Some interesting questions that have raised during the research are recommendations for future exploration.

- Can the MPD be used as a quality measurement for an evaluation function. If it is too big, it might be the case that no distinction can be made between different branches of the tree. Apart from the fact if that is a good thing or not, the MPD might play a role in the investigation of this matter.
- Is there a linear relation between MPD and the material balance of the parent position? The bigger a material difference in a position, the bigger the positional influence?
  MPD = a * [material balance] + b?
  Also the number of pieces involved may play a role here, as this is the case in end-games and in end games the positional influence of pieces becomes bigger.
- Is it possible to determine an even more precise MPD whit a ProbCut like methodology? A shallow search would identify positional differences to determine the current MPD.
- How does SPP with transposition table perform when SPP can be turned on and off?

Acknowledgments

I would like to thank:

Prof. Dr. Paul Klint, my supervisor, for his clear and honest advise.

Drs. Daniel Vis for his proof reading and smart feedback.

Ir. Rolf Doets for his interest in my work and discussions on the topic.

Last but not least my wife drs. Yanming Tong for her advise and patience.

Abbreviations

AB	Alpha-Beta
ABT	Alpha-Beta with Transposition Table
KBNK	King, bishop and knight against king end game
KPK	King and pawn against king end game
KRK	King and rook against king end game
MPD	Maximum Positional Difference
MTDf	Memory Table Driven with first guess
NBP	Node Bottom Positions: leafs of a tree
NC	Node Count, number of visited nodes in a tree
SPP	Sibling Prediction Pruning
SPP	Alpha-Beta with Sibling Prediction Pruning
SPPT	Sibling Prediction Pruning (Alpha-Beta) with Transposition Table

References & Bibliography

[1] A. Junghanns, J Schaeffer, M. Brockington, Y. Bjornsson, T. Marsland, *Diminishing returns for additional search in chess*, Advances in Computer Chess 8, pages 53-67, 1997

[2] D. Gleich, *Machine Learning in Computer Chess: Genetic programming and KRK*, Technical Report, Harvey Mudd College, 2003

[3] J. Cogley, *Designing, implementing and optimising an object-oriented chess system using a genetic algorithm in Java and its critical evaluation*, dissertation, Open University, September 2001

[4] C. Frayn (project leader), *Beowulf,* http://www.frayn.net/beowulf/index.html*,* Free source code

[5] C.E. Shannon, *Programming a Computer for Playing Chess*, Philosophical Magazine, Ser. 7, Vol. 41, Bell Telephone Laboratories, March 1950

[6] T.A. Marsland, *A Review of Game-Tree Pruning*, Computing Science Department, University of Alberta, ICCA Journal 9, 1, pages 3-19, 1986

[7] Eclipse. Development platform, http://www.eclipse.org/

[8] A. Plaat, W. Pijls, A. de Bruin, J. Shaeffer, *An Algorithm Faster than NegaScout and SSS* in Practice*, Paper presented at Advances in Computer Chess 7, 1995

[9] A. Plaat, W. Pijls, A. de Bruin, J. Shaeffer, *SSS* = α-β + TT*, Erasmus University Rotterdam, Technical Report EUR-FEW-CS-95-02, December 1994

[10] A Zobrist, *A New Hashing Method with Application for Game Playing*, University of Wisconsin, Madison, Technical Report 88, April 1970

[11] L. S. Coles, *Computer Chess: The drosophila of AI*, AI Expert Magazine, April 1994

[12] T. A Sudkamp, *Languages and Machines*, Second edition, ISBN: 0-201-82136-2

[13] J. Schaeffer, *The History Heuristic and Alpha-Beta Search Enhancements in Practice*, Journal: IEEE Transactions on Pattern Analysis and Machine Intelligence, volume PAMI-11, pages 1203-1212, 1989

[14] J. R. Slagle, J. K. Dixon, *Experiments with some programs that search game trees*, Journal of the ACM (JACM), Volume 16 , Issue 2 (April 1969), pages 189 – 207

[15] D. McIllroy, *The solution for the branching factor of the alpha-beta pruning algorithm and its optimality*, Communications of the ACM, Volume 25, Issue 8, pages 559 – 564, August 1982

[16] Y. Björnsson, T.A. Marsland, *Risk Management in Game-Tree Pruning*, Information Sciences. 122(1), pages 23-41, 2000

[17] A. de Groot, *Thought and Choice in Chess*, ISBN: 9027979146, 1965

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

[19] FIDE Handbook, Chess Laws, http://www.fide.com/official/handbook.asp, World Chess Federation

[20] M. Buro, *ProbCut: An effective selective extension of the alpha–beta algorithm*, ICCA Journal, 18(2) pages 71–76, 2002

[21] A.X. Jiang, M. Buro, *First Experimental Results of ProbCut Applied to Chess*, Advances in Computer Games 10, 2003

[22] J.A. Birmingham, P. Kent, *Tree-searching and tree-pruning techniques*, Advances in Computer Chess, Edinburgh Univ. Press, pages 89-96, 1977

[23] J.J. Gillogly, *The Technology Chess Program*, Artificial Intelligence,Vol. 3, No. 1-3, pages 145-163, 1972

[24] F. Hsu, T.S. Anantharaman, M.S. Campbell, A. Nowatzyk, *Deep Thought*, Computers Chess and Cognition, pages 55-78, 1990

[25] J. Schaeffer, *The History Heuristic and Alpha-Beta Search Enhancements in Practice*, Journal: IEEE Transactions on Pattern Analysis and Machine Intelligence, volume PAMI-11, pages 1203-1212, 1989

[26] S. J.J. Smith, D. S. Nau, *Toward an Analysis of Forward Pruning*, Computer Science Technical Report Series; Vol. CS-TR-3096, University of Maryland, 1994

[27] D. Michie, *Chess with Computers*, Interdisciplinary Science Reviews 5, pages 215-227, 1980

[28] M. Donskoy, J. Shaeffer, *Perspectives on Falling from Grace*, Computers, Chess and Cognition. Springer-Verlag, 1990

[29] D. Kopec, I. Bratko, *The Bratko-Kopec experiment: a comparison of human and computer performance in chess*, Advances in Computer Chess 3 (ed. M.R.B. Clarke), 1982

[30] D. Gomboc, M. Buro, T.A. Marsland, *Tuning evaluation functions by maximizing concordance*, Theoretical Computer Science, Volume 349, Issue 2, pp. 202-229, 2005

[31] Chua Kong Sian (First author), *GNU Chess*, http://savannah.gnu.org/projects/chess/, GNU General Public License, Free Software Foundation Inc.

[32] A. Plaat, http://www.cs.vu.nl/~aske/mtdf.html, Implementations of MTDf and iterative_deepening

Appendix A, Test Results

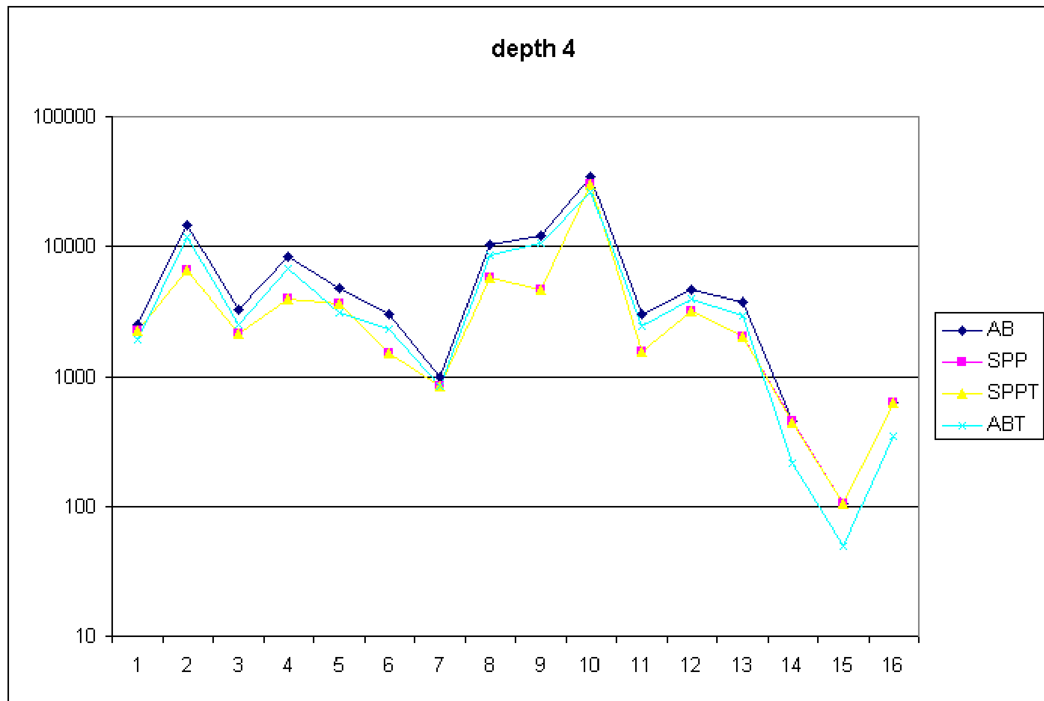


Diagram 1: NBP Counts of all methods (logarithmic scale) - depth 4

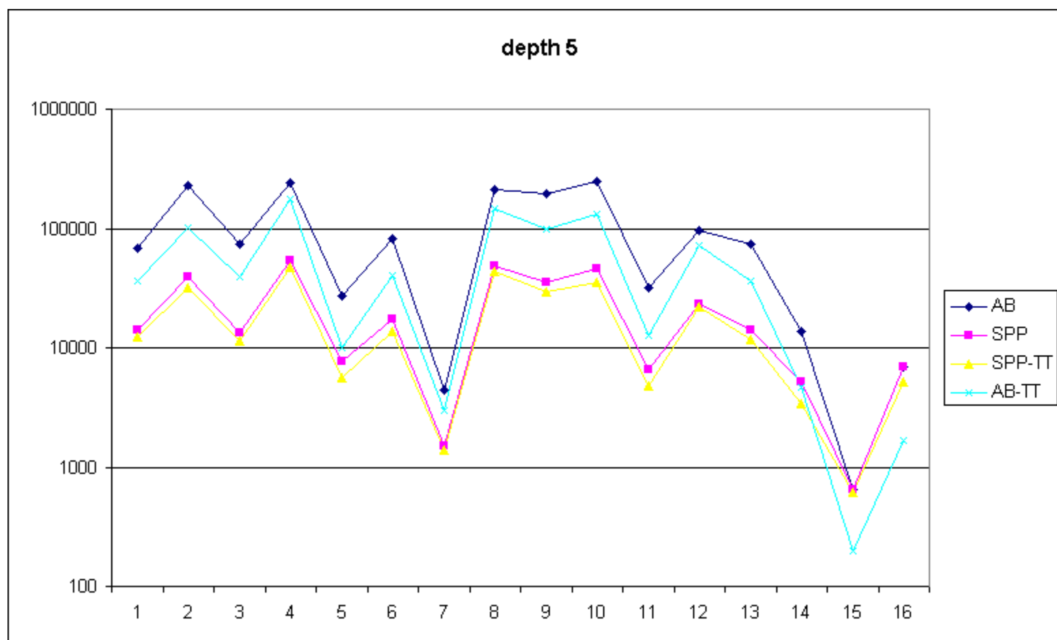


Diagram 2: NBP Counts of all methods (logarithmic scale) – depth 5

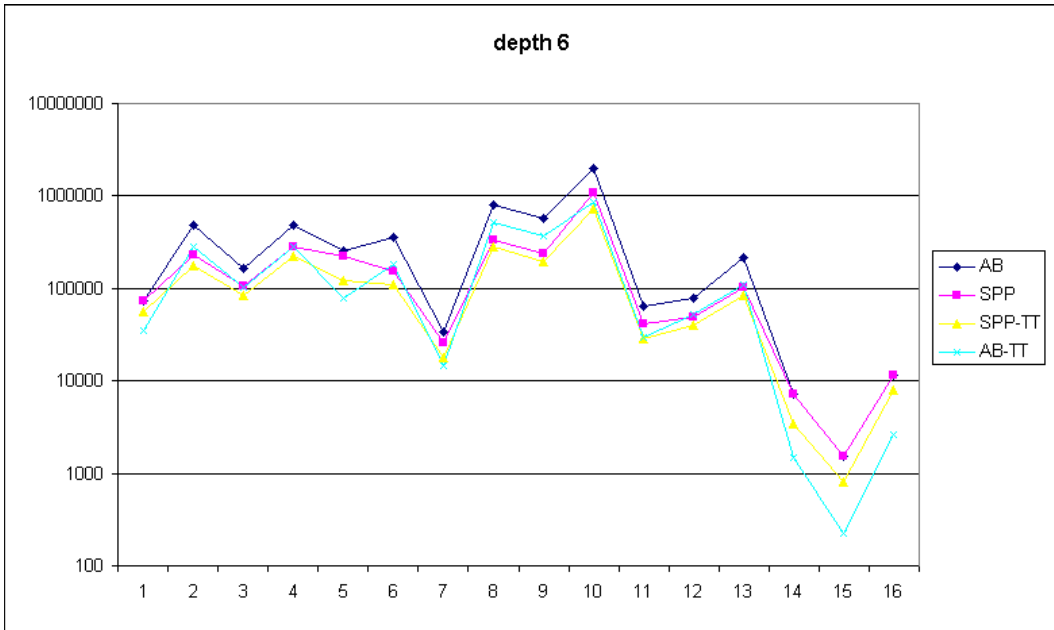


Diagram 3: NBP Counts of all methods (logarithmic scale) – depth 6

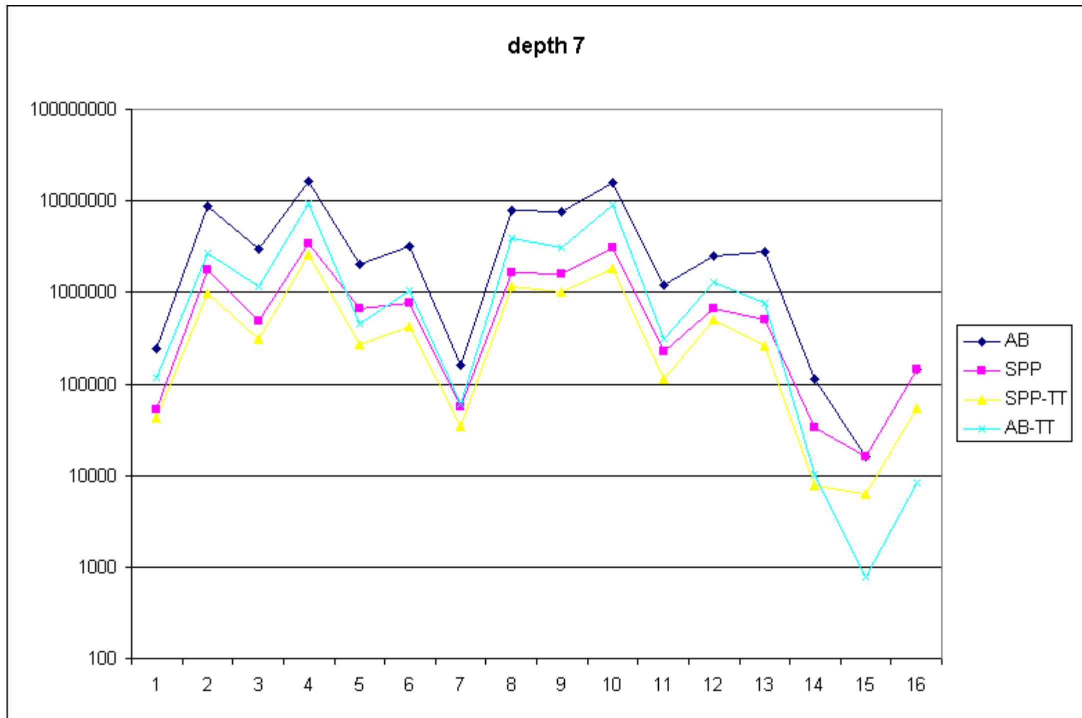


Diagram 4: NBP Counts of all methods (logarithmic scale) – depth 7

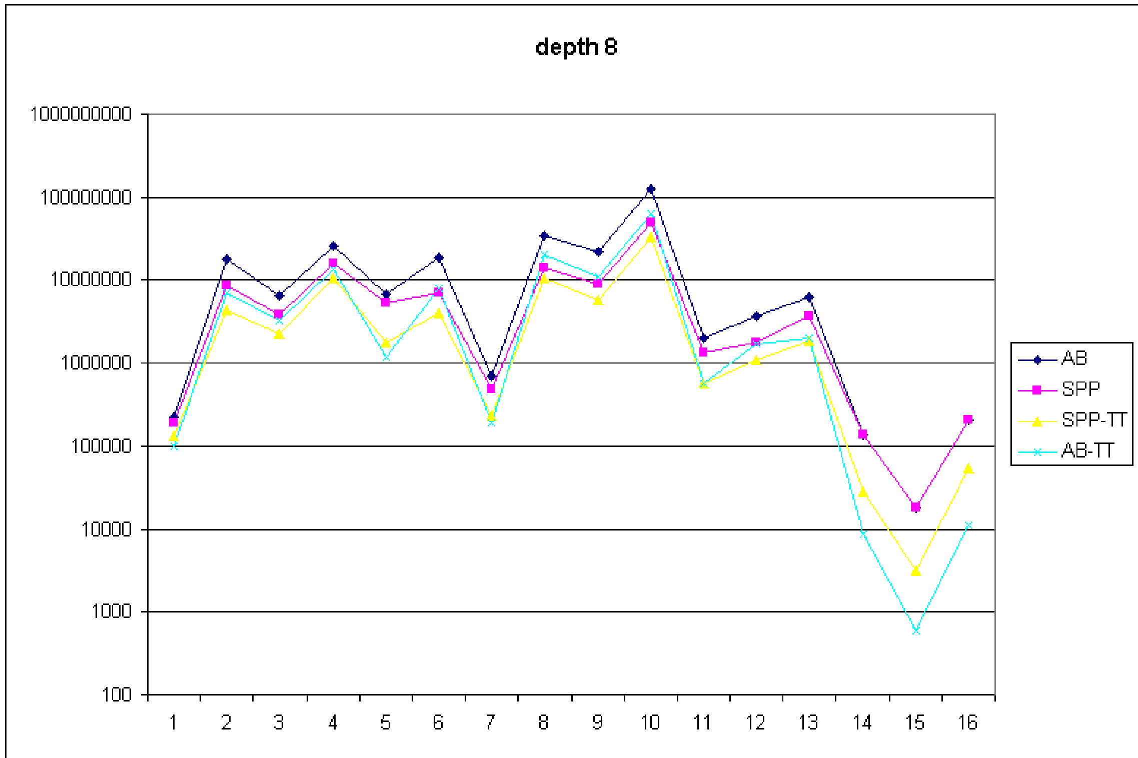


Diagram 5: NBP Counts of all methods (logarithmic scale) – depth 8

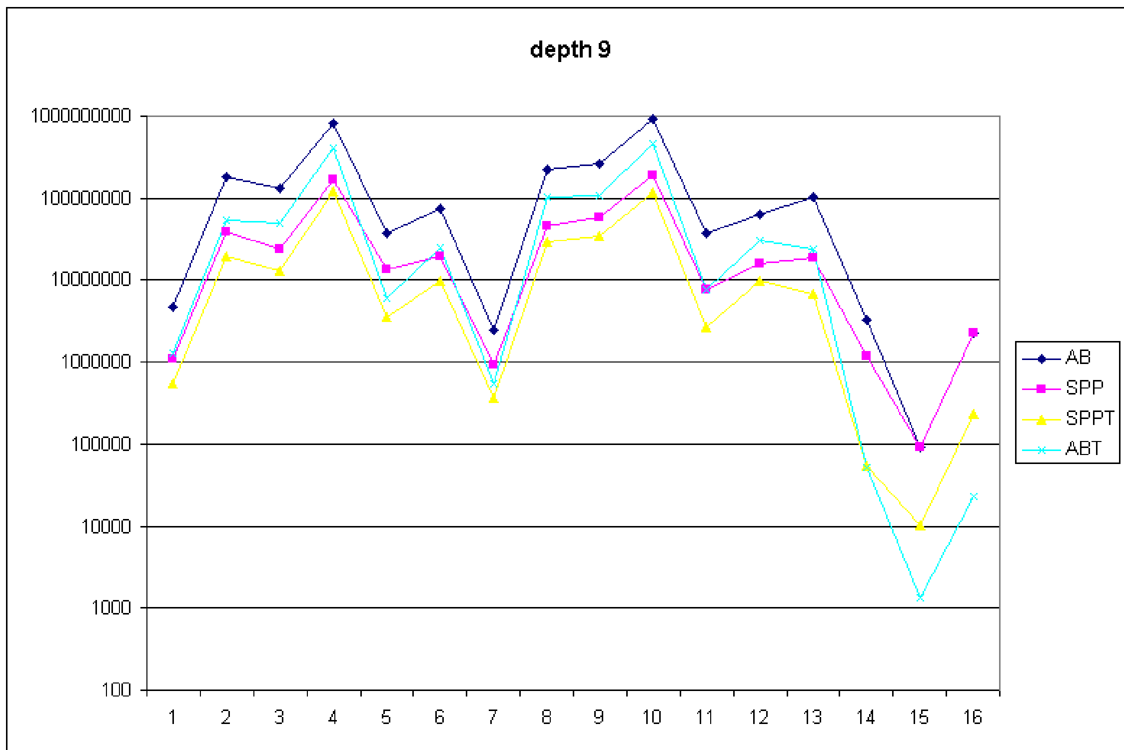


Diagram 6: NBP Counts of all methods (logarithmic scale) – depth 9

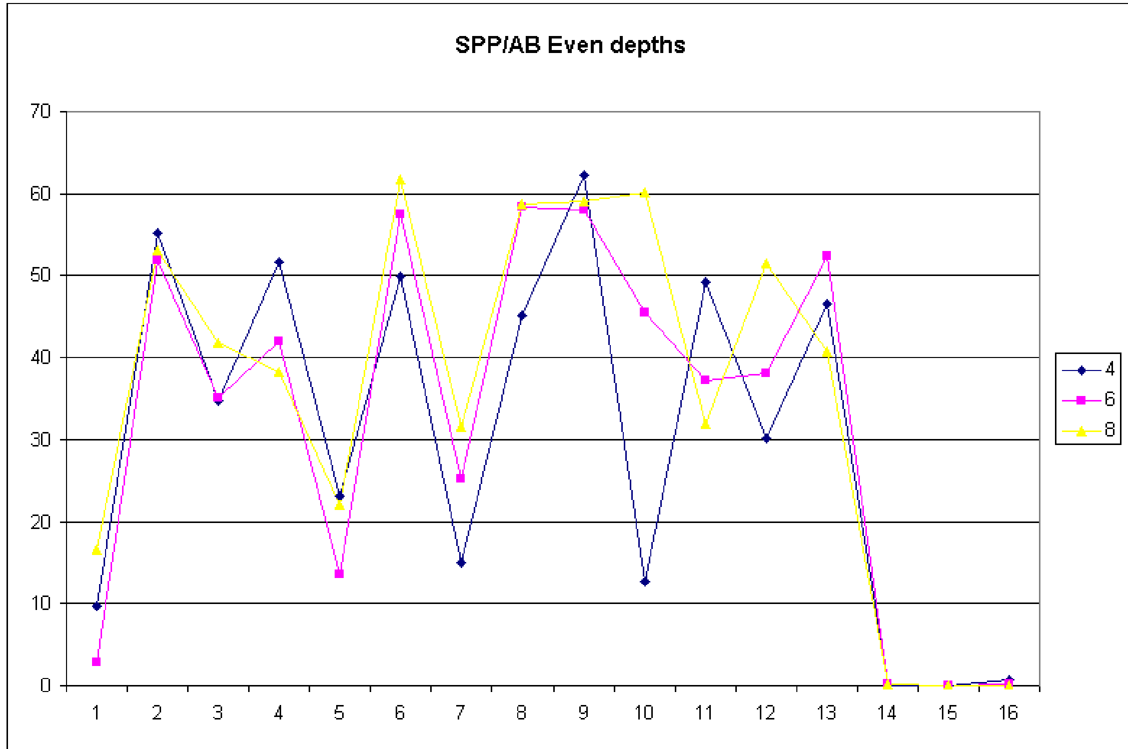


Diagram 7: Extra pruning percentage of SPP over AB on even depths

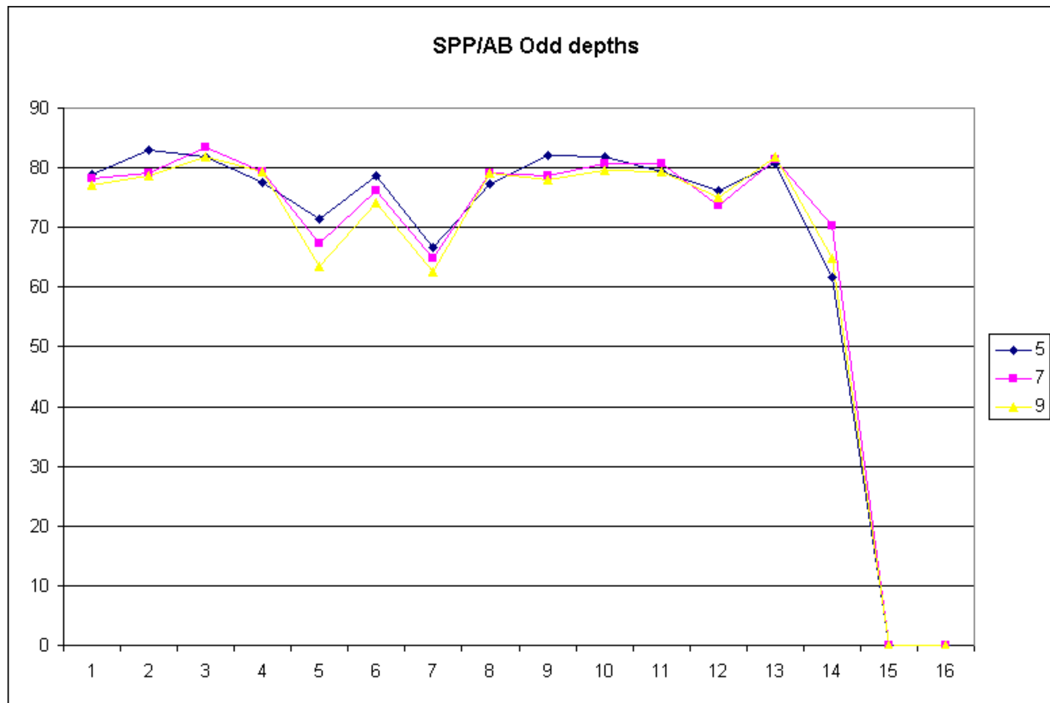


Diagram 8: Extra pruning percentage of SPP over AB on odd depths

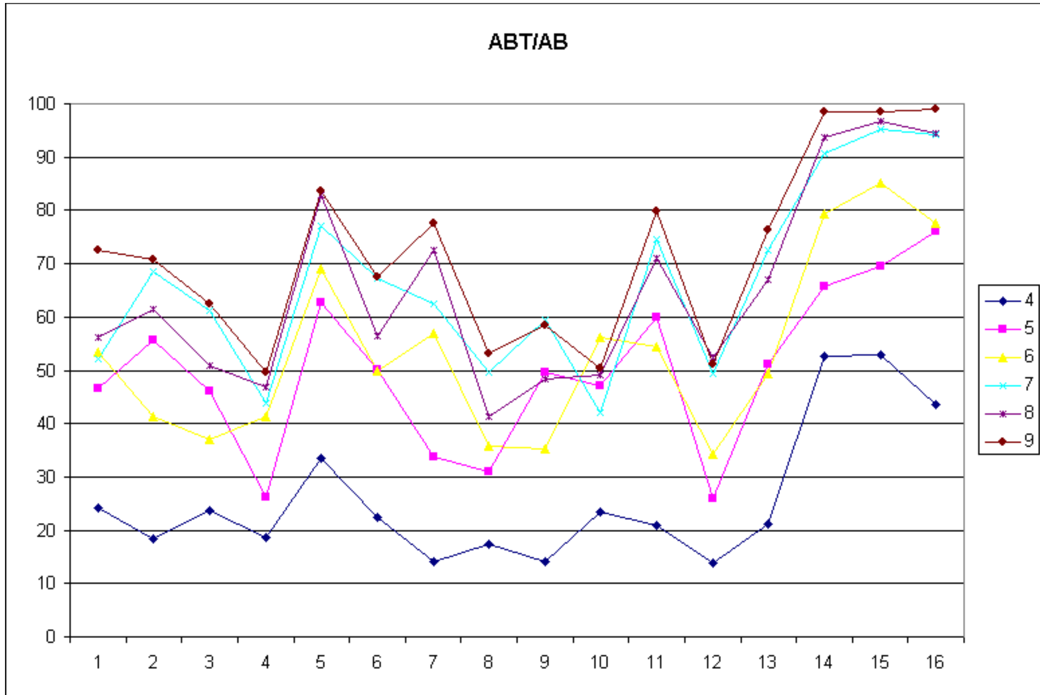


Diagram 9: Extra pruning percentage of ABT over AB

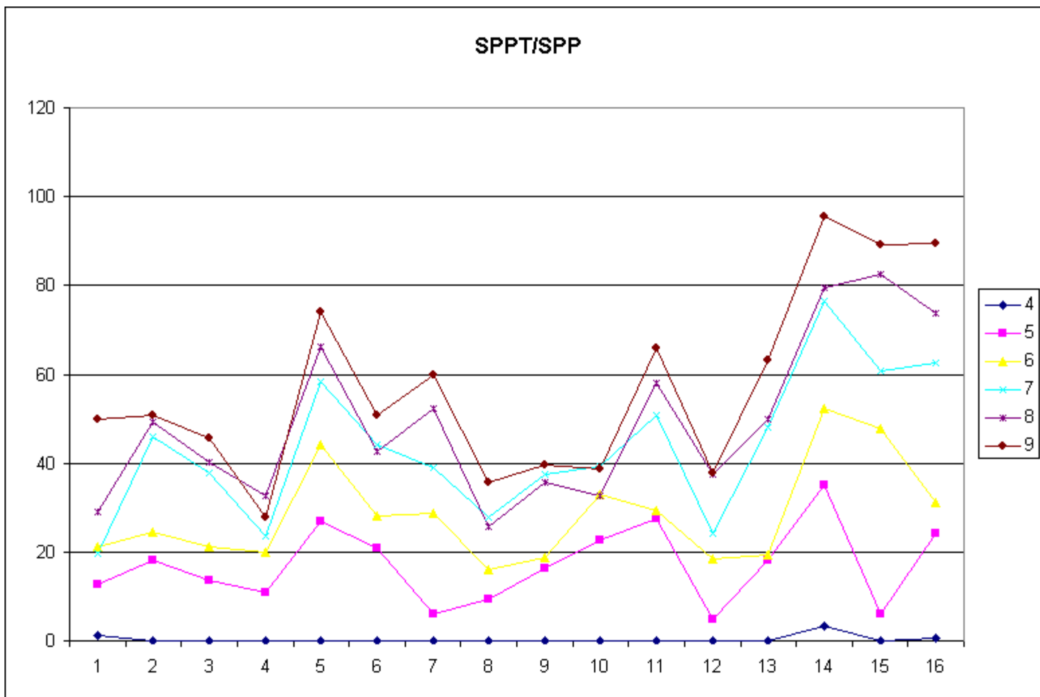


Diagram 10: Extra pruning percentage of SPPT over SPP

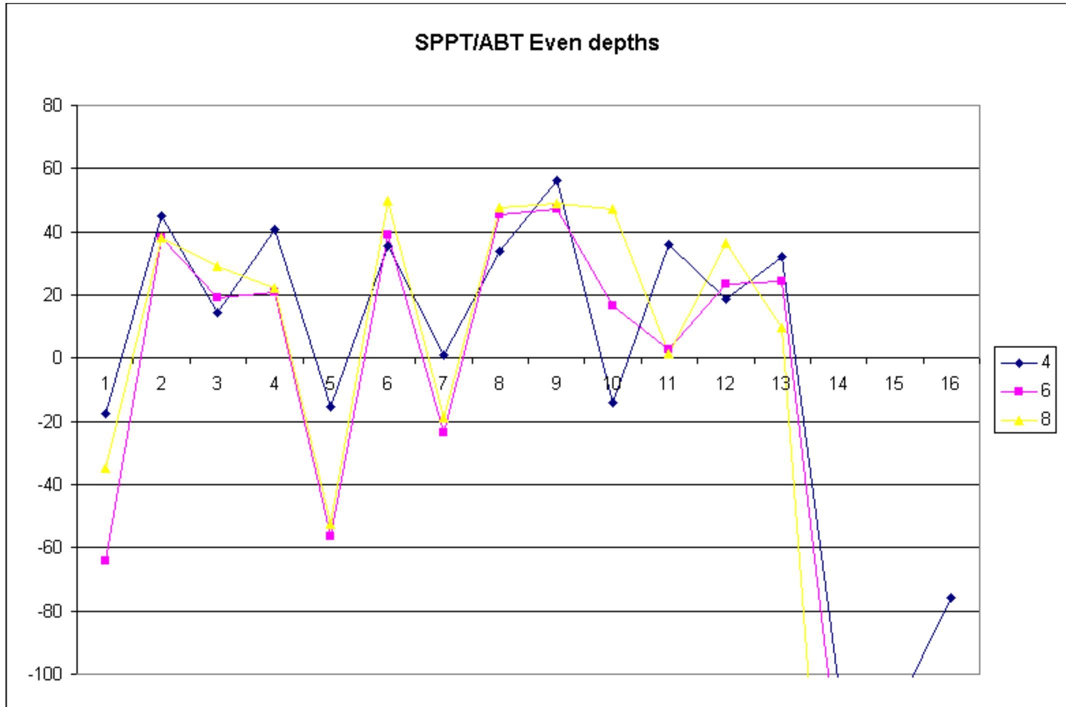


Diagram 11: Extra pruning percentage of SPPT over ABT on even depths

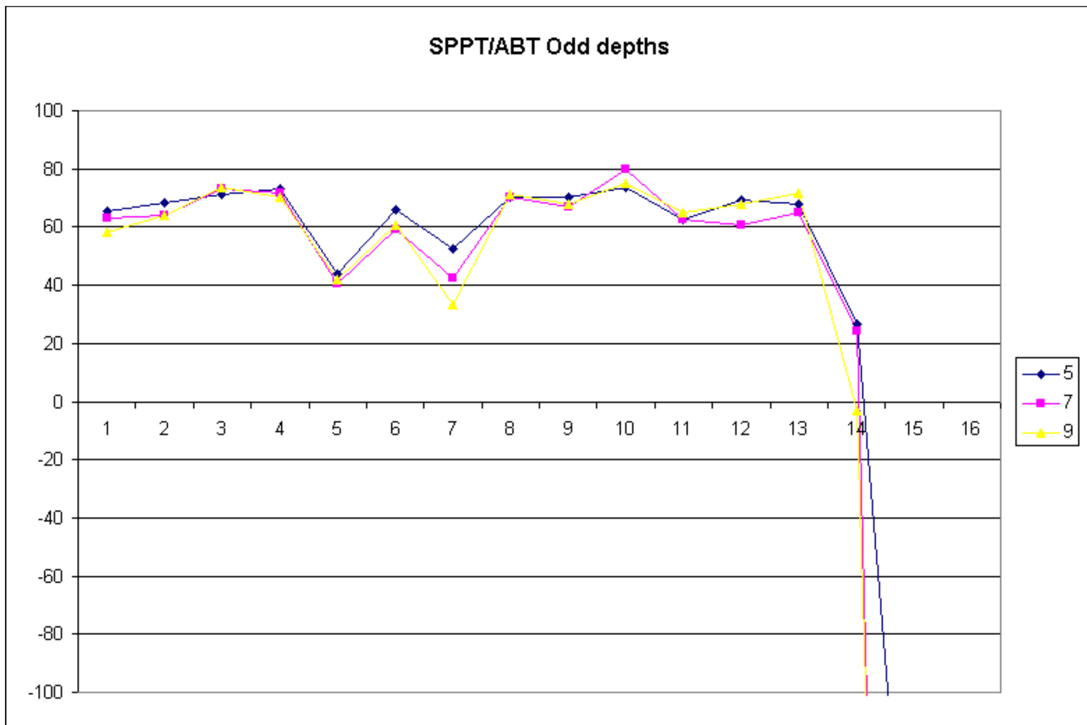


Diagram 12: Extra pruning percentage of SPPT over ABT on odd depths

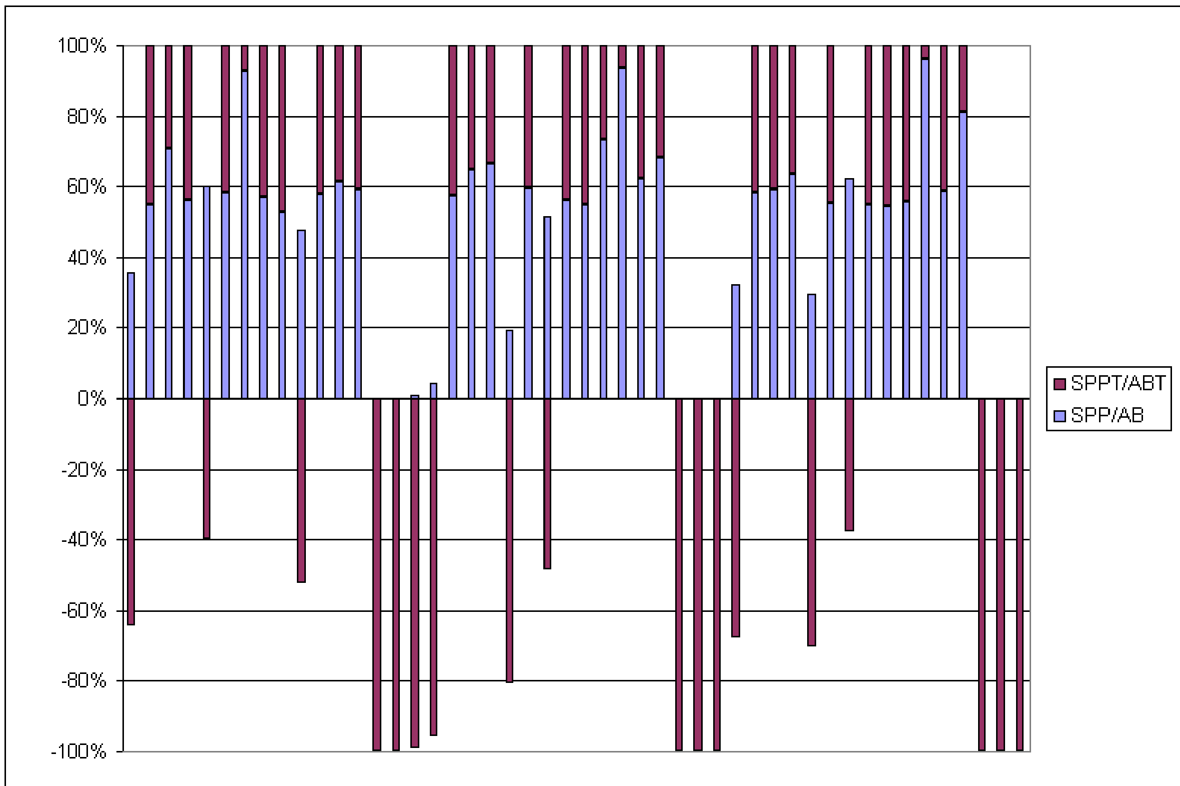


Diagram 13: Stacked percentages of SPPT/ABT and SPP/AB on even depths (4,6,8)

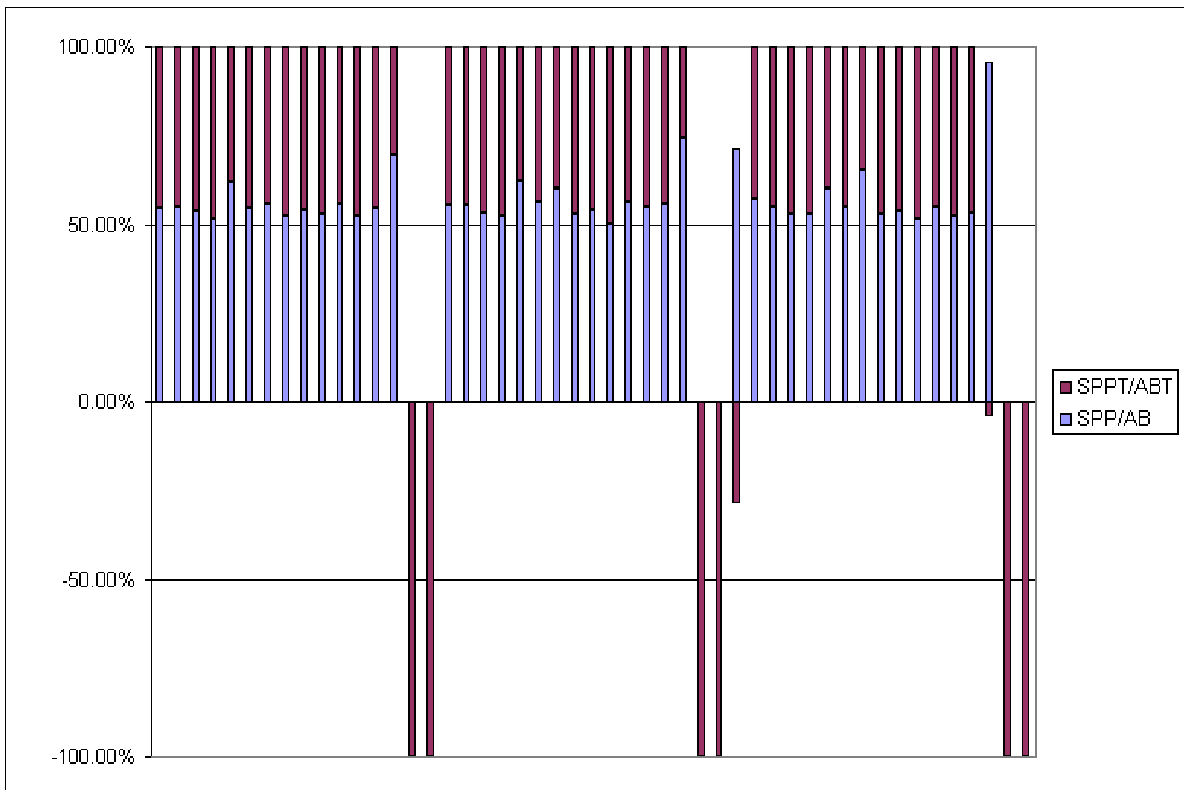
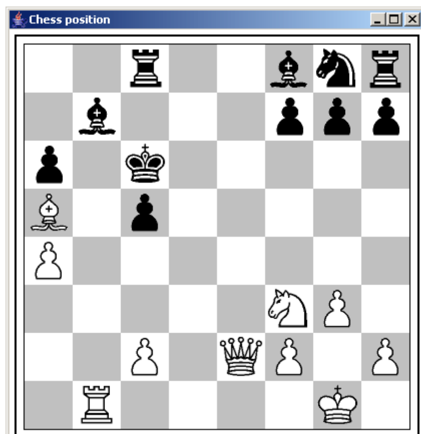
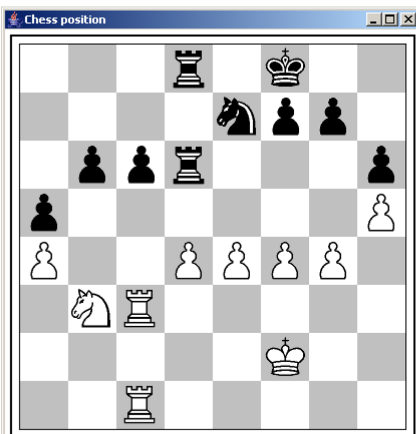


Diagram 14: Stacked percentages of SPPT/ABT and SPP/AB on odd depths (5,7,9)

Appendix B, Positions of the test set

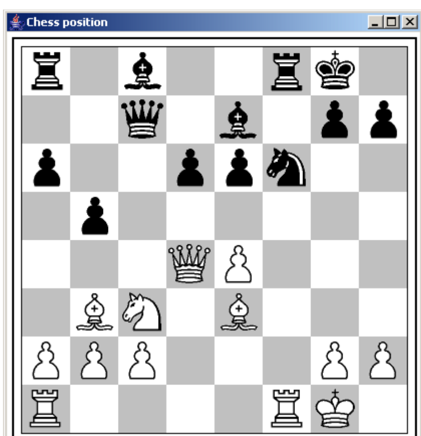


Position 1

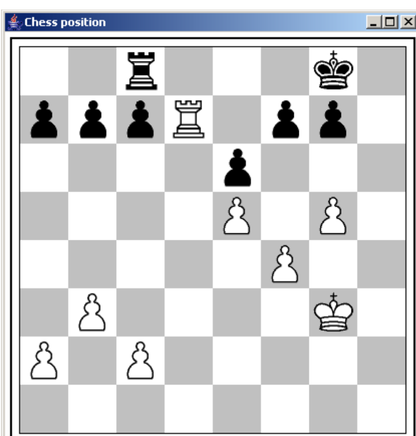


Position 2

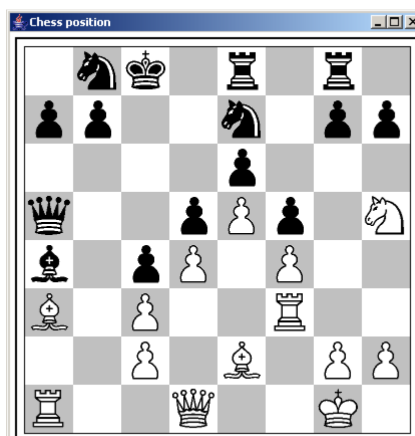
Position 3



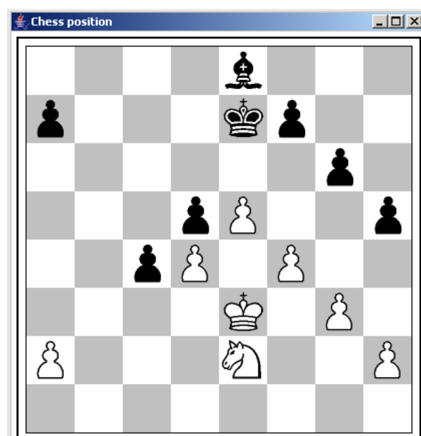
Position 4



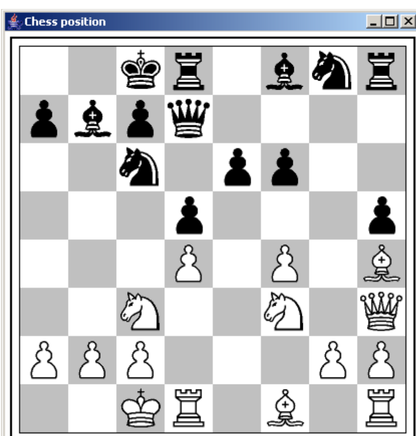
Position 5



Position 6

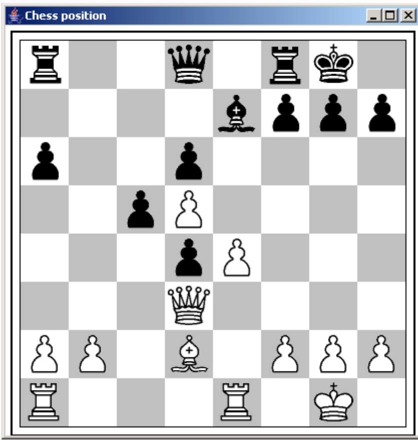


Position 7

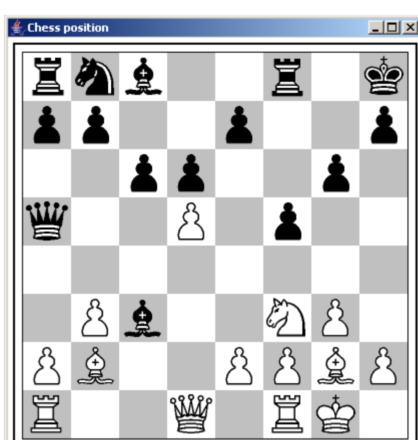


Position 8

Position 9

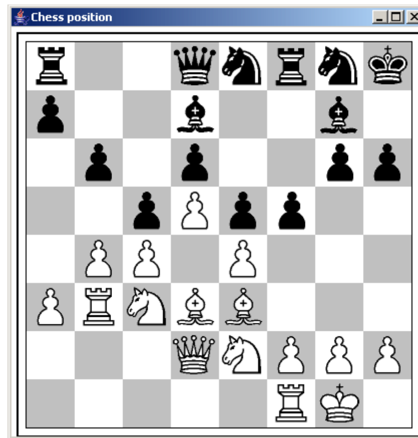


Position 10

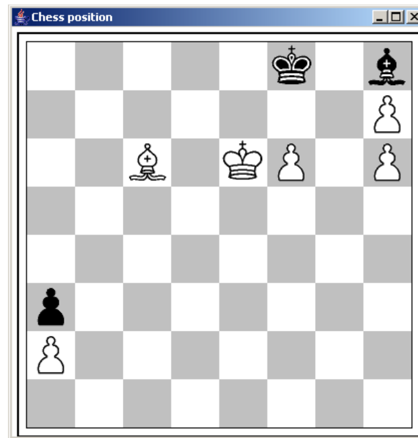


Position 11

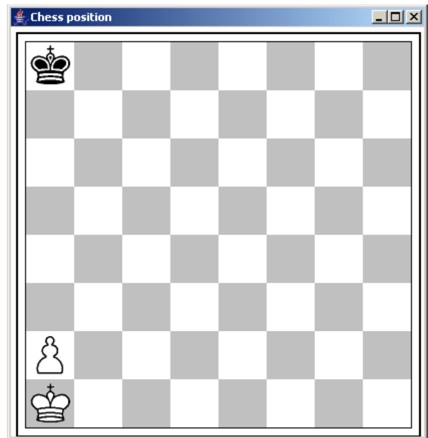
Position 12



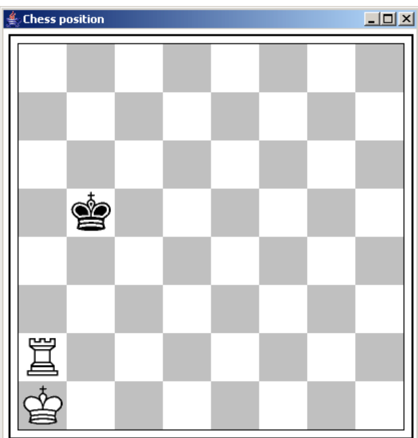
Position 13



Position 14



Position 15



Position 16