# From MiniMax to Manhattan

**Tony Marsland and Yngvi Björnsson**
University of Alberta
Department of Computing Science
Edmonton, Alberta
CANADA T6G 2H1
E-mail: {tony,yngvi}@cs.ualberta.ca

### Abstract

The thinking-process for playing chess by computer is significantly different from that used by humans. Also, computer hardware/software has evolved considerably in the half century since minimax was first proposed as a method for computers to play chess. In this paper we look at the technology behind today's chess programs, how it has developed, its current status, and explore some directions for the future.

## Introduction

One of the most spectacular shows ever to be hosted on Broadway takes place this May. Human supremacy in chess is being challenged by a computer. Garry Kasparov, the current world-champion and one of the greatest chess players in history, will defend the honor of mankind. The challenger, Deep Blue, is a computer specially designed for playing chess, a masterpiece of science and engineering. Whatever the outcome of the match, the significance of this event is the same. A combined effort of research in artificial intelligence (AI) and computer hardware development, has made it possible for computers to match humans in a game that has for centuries been considered a measure of human intelligence.

Problem solving in AI often reduces to a process of tackling three main issues: representing knowledge, searching for solutions, and using knowledge to direct the search. A hallmark of intelligence is the use of knowledge to make search problems more tractable. The most studied search problems are those that arise in game playing. For decades chess has been viewed as an important testbed for machine intelligence, with some researchers regarding computer-chess as its *Drosophila*. Early chess programs were inspired by concepts from human thinking. However, with steadily increasing computer power—making brute-force methods more attractive—and with the difficulties of implementing human-like reasoning, computer-chess research took a different direction. Most of the effort went into search techniques, while less attention was paid to the acquisition and use of knowledge.

The paper opens with a brief history of computer-chess, followed by an overview of current methodology. Finally we consider the future of computer-chess and its technology, and offer some direction for future work on the software aspects.

## The Past: a Brief History

Development of computer-chess techniques has been strongly influenced by advances in search algorithms and computer hardware. The evolution of these two has often gone together, more powerful hardware allowing more complex search strategies.

### Search Algorithms

Half a century ago von Neumann and Morgenstern proposed *minimax* as a method to decide which move to make in chess. Using the negamax framework, the minimax algorithm is described recursively as

$$MinMax(n) = MAX_i(-MinMax(n_i))$$

where $\{n_i\}$ are the successors of node $n$. If $n$ does not have successors, then it is a leaf node and $MinMax(n)$ is defined as the merit of node $n$ from the perspective of the side that has the move. In principle, minimax can be used to determine an optimal action in every chess position, simply by expanding all continuations to the end and backing up the precise value of the outcome; in this case win, loss, or a draw. In practice, this is of course infeasible because of the astronomical number of possible continuations.

In the 1950's early pioneers of computer chess, like Claude Shannon and Alan Turing, proposed various search strategies based on the minimax principle, and laid the foundation of an active research area. Faced with the large search space, early chess programs used highly selective search methods, expanding just a few plausible moves in each position. This human-like

approach proves difficult, the programs often neglect good moves—throwing the baby out with the bathwater. The quality of play by the early chess programs was therefore poor.

Over the years, numerous algorithms were proposed to backup minimax values in game-trees, most notably *Alpha-Beta*. Knuth and Moore (Knuth & Moore 1975) worked the algorithm into the negamax framework and described its full pruning capabilities. However, the underlying idea was known in 1958, and weaker forms of it were implemented in such early chess programs as NSS, by Newell, Shaw, and Simon. The method uses an effective cutoff technique based on the observation that parts of the tree expanded by the minimax algorithm are irrelevant for proving the value of the tree, and thus need not be visited. The basic idea behind the pruning is that having already found one way of refuting the opponent's play, there is no need to look for others. The Alpha-Beta algorithm is a major improvement over exhaustive minimax. In the best case, the number of nodes visited is about the square root of the number seen by minimax, and it never searches more nodes.

As more and more powerful computers became available, and the full pruning capabilities of Alpha-Beta became better known, brute-force search programs based on the method replaced other approaches. During the past two decades, much effort has been invested in enhancing the efficiency of the search. By expanding the expected best line of play first (the principal variation) the alternatives only need to be shown inferior to that best line. This observation led to more efficient variants, such as *Principal Variation Search* (Marsland & Campbell 1982) and *NegaScout* (Reinefeld 1983). These variants rely on having a good move ordering scheme that expands promising moves early. Various techniques to improve the move ordering and to make the search more efficient were developed, for example, iterative-deepening, use of transposition-tables, and forward pruning. From now on we will refer to these techniques as search enhancements, and call Alpha-Beta—particularly its variants that employ such enhancements—by the generic name *Null Window Search* (NWS), thus reflecting the essence of the more effective implementations.

Minimaxing is not the only way to backup values in game trees, nor do the enhanced methods necessarily traverse the smallest trees. Various other algorithms have been proposed and implemented, but none have found wide use in practice, because they are not time-efficient.

### Hardware

As computers evolved it became clear that increasing search speed dramatically improves the playing strength of even the simplest chess programs. Using only elementary chess knowledge, the programs could play reasonably well by applying extensive search. The quest for more speed took two directions; on the one-hand developing special-purpose hardware for playing chess, and on the other adapting the search to run on many general-purpose processors in parallel.

The first special-purpose high-speed circuitry for playing chess was developed by Ken Thompson in the late 1970's, and was used in his chess program Belle. The chess-specific hardware provided fast move generation. In the 1980's more high-speed chess circuitry saw the light of day, like the developments at CMU for Hitech and Deep Thought. Not only were the new custom chips capable of move generation, but they also provide position evaluation and search.

In the early 1980's chess programs running on multiprocessor systems started to compete in computer-chess tournaments. While the first multi-processor based chess programs ran on only few processors, today's systems use hundreds. Searching in parallel poses many new programming problems, and many techniques have been developed to allow NWS algorithms to traverse game-trees in parallel.

The Deep Blue program combines both of the above approaches; it runs on many general-purpose processors, where each processor has access to several high-speed special-purpose chess circuits.

### Other Improvements

Although improvements in search algorithms and computer hardware have dramatically increased chess programs' playing strength, other sources have contributed as well. For instance, use of openings books and endgame databases have proved very useful. Last, but not least, decades of experience have shown programmers what kind of chess knowledge most benefits the program, and how it can efficiently be represented. For chess programs, just as for humans, the knowledge acquisition period needed to achieve mastery is long and tedious.

## The Present: Current Technology

It is safe to state that today's best chess programs are playing at the GM level. This success comes from gradual improvements over the last half century; a direct result of active research in computer-chess. Advances in search algorithms, parallel computing, special purpose chess hardware, knowledge representation, and

opening/endgame databases all contribute to increased playing strength.

## Search Algorithms

Most contemporary programs use some form of NWS, see Figure 1. It keeps track of a lower-bound, $\alpha$, and an upper-bound, $\beta$, of the value a player can hope to achieve. During the search, whenever the value returned from a subtree exceeds the upper-bound $\beta$, a cutoff occurs. Intuitively this means that we have found a refutation of the current line of play, and therefore our opponent will not select this variation, but will instead play the line that achieves the value indicated by the upper-bound $\beta$. The algorithm first expands the so called principal variation with a full $\alpha$-$\beta$ window, and then the remaining moves with a minimal window around the score the principal variation returned. This results in efficient searches and only occasionally, when an alternative move really turns out to be better, is a re-search with a wider window necessary.

When equipped with enhancements such as a transposition table (accessed by functions *LookTT*() and *AddTT*() in Figure 1), move-ordering (*AddCredit*(), *NextMove*()), search extension/pruning schemes (*Extend*(), *NullMove*), and quiescence search (*QS*()), the algorithm is extremely efficient. Thus the fastest chess programs look ahead about 5-7 moves for each side in complicated middle-game positions, and search lines of special interest even more deeply. In the endgame and in other less complicated positions a much deeper search is possible.

## Search Enhancements

Various search enhancements are essential for the NWS algorithm to achieve exceptional performance. Many of them focus on allowing the algorithm to be more selective in how the search effort is spent. The selectivity is introduced by varying the search horizon, some lines are searched to a depth that is greater than the nominal depth, while others are terminated at a shallower depth. The real task is to identify which alternatives are worth considering further and which can be pruned off. In Figure 1 three common methods to vary the search horizon are shown: quiescence search, extending forcing lines, and a forward pruning heuristic called null-move.

Having searched some initial position to a designated maximum depth, some of the positions that arise are volatile and hard to evaluate. For example, if we evaluate a position where a capture has just occurred, without giving the opponent opportunity to re-capture, a huge error may be introduced. Therefore, all captures for both sides are played out before a position is evaluated. Sometimes other moves that can dramatically

```
int NWS( int height, int alpha, int beta ) {
 MOVE move, tt_move=NullMove, best_move=NullMove;
 int  score, best_score, bound, ttbound;
 TT   tt;

 //Check for draw by repetiton or 50 move rule
 if ( IsDraw() ) return DRAW;
 if ( Extend() ) height++;  //Extend forced moves

 //Call quiescence search if horizon is reached
 if ( height == 0 ) return QS( alpha, beta );

 //Look the position up in the transposition table
 if ( LookTT( &tt ) ) {
   if ( tt.height >= height ) {
     switch ( tt.bound ) {
      case TValue: return tt.score;          break;
      case UBound: beta = MIN(tt.score,beta);break;
      case LBound: alpha = MAX(tt.score,alpha);
     }
     if ( alpha >= beta ) return tt.score;
   }
   best_move = tt_move = tt.move;
 }

 //Do a null-move search with search reduction R>1
 //if not in check and last move not a null-move
 if ( (height > R) && NullmoveIsOK() ) {
   Make( NullMove );
   score = -NWS( height-1-R, -beta-1, -beta );
   Retract( NullMove );
   if ( score >= beta ) return beta;
 }

 //Get the first move if it wasn't found in the TT.
 //If no legal moves, either a mate or a stalemate.
 if (best_move==NullMove && !NextMove(&best_move))
   return IsCheckmate() ? LOSS : DRAW;

 //Search the principal move before the others
 Make( best_move );
 best_score = -NWS( height-1, -beta, -alpha );
 Retract( best_move );
 while ( (best_score < beta) && NextMove(&move) ) {
   if ( move == tt_move ) continue;
   bound = MAX( alpha, best_score );
   Make( move );
   score = -NWS( height-1, -bound-1, -bound );
   if ( (score > bound) && (score < beta) )
     score = -NWS( height-1, -beta, -score );
   Retract( move );
   if ( score > best_score ) {
     best_move = move; //A new best move is found
     best_score = score;
   }
 }
 //Update the TT and move ordering information
 ttbound = (best_score <= alpha) ? UBound :
           (best_score >= beta)  ? LBound : TValue;
 AddTT( height, best_move, best_score, ttbound );
 AddCredit( ttbound, height, best_move );
 return best_score;
}
```

Figure 1: NWS enhanced with TT and Null-move

change the evaluation are also made, for example promotions. This resolution of dynamic factors is called the *quiescence search* phase.

A common practice is to extend the search along "forced" lines, such as certain re-captures, or when one side is moving out of check. In Figure 1 we assume the existence of a function *Extend*() that checks for forced moves. If a decision is made to extend, then the maximum depth of search for this line is extended by one.

As opposed to search extensions, pruning schemes select branches which are to be searched to less than nominal depth. This is referred to as forward pruning, and can involve erroneous decisions by prematurely truncating good lines. The null-move heuristic is the commonest forward pruning method used in contemporary chess programs. The underlying idea is that in chess it is almost always beneficial to make a move rather than to pass. Therefore, if the score received by giving up a move is still good enough to cause a cutoff, it is very likely that at least one legal move will too. When the position is assessed using a shallower search than we would otherwise, considerable search effort is saved. The constant $R$ shown in Figure 1 decides the search reduction for the null-move searches, and is typically set to 2. Some precautions are necessary when using a null-move, for instance, we can not allow null-move to be made when the side to move is in check, nor can two consecutive null-moves be allowed. In chess it is almost always safe to make the assumption that making a move will improve the position, but for a special case, called *zugzwang* positions, this is not true. Most chess programs turn off the null-move heuristic when entering the endgame, because zugzwang positions are more likely to arise there.

The *transposition table* is used to store information about positions that have been visited before, either during an earlier iteration or when different move sequences transpose into the same position. Typical items to store in a transposition table are: the merit value of a position, type of the value (i.e. true value, upper-bound, or lower-bound), the height of the subtree the position was searched to establish the value, the best move in the position, and a hash key for the position. The hash key is necessary because hashing is used to index the table. Whenever a node is visited, the transposition table is checked to see if it was seen before, and if so, if it was searched sufficiently deep so that we can reuse the merit value from the table. The merit can be used to determine a true value for the position, thus eliminating the need to search the position further, or to adjust the current $\alpha$-$\beta$ bounds, which can also lead to a cutoff. The use of a transposition table can reduce the search space significantly, especially in endgames where transpositions occur frequently. The table can only hold a small portion of actual positions searched, therefore various replacement schemes are used to decide which positions to keep; one uses two-level transposition tables. A two-level table stores for each entry, not only the most recent position hashed into that entry but also the position that was searched the deepest. In Figure 1 the functions *LookTT*() and *AddTT*() retrieve and insert entries into the transposition table, respectively.

NWS-like search algorithms perform best when good moves are expanded first. Therefore, various *move-ordering heuristics* have been developed. One such is to store the best move in the transposition table. When a node is revisited this move is always tried first; the rationale being that a move previously found to be good in one position is also likely to be good when the position is searched to a greater depth. Brute-force methods explore all possible moves in each position, many of which can easily be refuted by an obvious capture. Chess programs therefore often order capture moves early in the move-list. The killer move (Slate & Atkin 1977) and the history heuristic (Schaeffer 1989) are more sophisticated move ordering schemes that are widely used, both are based on the idea that a move that is good in one position is often good elsewhere. The former keeps track of a few best moves found at each height level in the tree, while the second keeps a global table for all possible moves. The best move found in each position gets a credit, shown in Figure 1 as a function call *AddCredit*(), and when moves are generated by *NextMove*(), captures are done first and then the remainder in order by the credit they have.

When using a depth-first search it is necessary to decide beforehand how deep to search. This makes it difficult to estimate how long the search will take. However, by gradually increasing the search depth one can better decide how long the search will take and when to stop searching. This is called *iterative-deepening*. Because the time for each iteration grows exponentially with increased search depth, the effort spent in earlier iterations is relatively small compared to the last iteration. The search overhead introduced by iterating on the search depth is therefore small. Furthermore, when used in combination with a transposition table, the principal variation is kept between iterations. This leads to a better move ordering, often resulting in iterative-deepening searching fewer nodes in total than the non-iterative approach. The technique of iterative-deepening search later found its way into other AI-domains, such as theorem-proving and single-agent search.

## Hardware

Today's state of the art PC is extremely powerful. The commercial chess programs running on them are also quite sophisticated, and the playing strength approaches the GM-level. The state of the art computer-chess hardware is undoubtedly Deep Blue. It uses 256-512 chess-specific processors, and under tournament time controls is capable of searching 50-100 billion positions for each move.

## Knowledge Representation

The chess knowledge used in contemporary chess programs is primitive by human standards. The programs have a notion of basic chess concepts such as material, mobility, pawn-structures, king-safety, weak/strong squares, space, center-control, and development. A good description of a typical knowledge encoded in chess programs can be found in Cray Blitz (Hyatt, Gower, & Nelson 1990), whose evaluation function is described in some detail.

More elaborate chess concepts, such as long-term planning, piece co-ordination, theme consistency, imbalances, long-term weakness, are totally missing from their vocabulary. It would be impossible for human players to achieve master strength without this knowledge. The chess programs partially compensate for this lack by using deep exhaustive searches.

## Openings Books and Endgame Databases

Just as for humans, good opening play is an important part of a strong chess program. The best programs invariable have opening books prepared by chess masters. Not only do the masters manually prepare new openings lines, but computers are also used to verify existing analysis in the hope of finding flaws or further improvements. The opening book must guide the program to positions that are not only favorable, but also well suited to the computer's play. This often implies avoiding closed positions in favor of more open and tactical ones. The importance of having a good opening book is evident from the fact that the top commercial programs specifically prepare opening lines against other top programs. Therefore, to prevent both chess programs and human players from repeatedly exploiting the same opening mistakes, some programs automatically update their opening book during a game; deleting lines that lead to an inferior positions soon after deviating from the book, while adding others that showed promise. Similar concepts were tried in earlier chess programs such as BEBE (Scherzer, Scherzer, & Tjaden 1990). It stored positions from its previous games in a special long term transposition table, thus preventing the program from making the same mistake twice.

Pre-calculated endgame databases are also available, for endgames with 5 or fewer pieces. When arriving at a position in the database, the programs play the best moves from there without any search.

# The Future: What is the Right Way

The benefit of each additional ply diminishes the deeper we search. This implies that the utility of searching deeper will become less and less. Therefore, we believe that there will be a shift in emphasis; less effort will be spent on search algorithms and more on the knowledge part.

## Search

Search is, and will remain, fundamental to chess playing programs—after all, tactical combinations are a big part of the game. This is one reason why chess programs have an advantage over human players. By exhaustively looking at all alternatives the programs often discover unexpected tactics or brilliant defending resources that are hidden to the human eye. Further search improvements can come from one of two sources:

- *Searching faster.*
  The search may benefit from increasing computing speed, but this will not be as dominating a factor as before. It can be more beneficial to invest the extra computing speed in incorporating additional knowledge. Research in parallel search is, and will, stay important, although not of ultimate importance to the AI community.

- *Searching smarter.*
  Alternatively, more selective search algorithms can be developed, that search "smarter" by more aggressively extending branches of interest while pruning others of lesser potential. Selective search methods are likely to attract increased attention from the research community. One of the fundamental questions of AI—how to use knowledge to direct search—has to be revisited.

## Knowledge

Further major improvements must come from additional chess knowledge. Questions like, what additional knowledge is important, how can it be acquired and effectively implemented, need to be addressed. These issues have been largely neglected in the past.

As mentioned before, knowledge acquisition is a long and tedious task. Methods to make this task easier, like tools for automatically extracting relevant chess

knowledge from big databases, will become increasingly useful.

The long-range planning component of chess programs should also be worked on. An old adage says that it is better to play with a bad plan in mind than no plan at all. Deep searches certainly give the programs some notion of a plan, but often it is too superficial and short sighted. It is sometimes embarrassing to watch a well prepared human opponent make even the strongest chess programs look like absolute beginners; simply by applying an uncomputer-like style of play.

Ideas are the tools of the chess player. Players do not reinvent the wheel every time a game is played. The same themes occur over and over again, and during a game players recall similar situations from their own and other people's games, and reuse the ideas from there. Often these ideas are based on long-term plans that are outside the scope of any search, they are simply memorized. Humans have an extra-ordinary talent for adapting existing knowledge and applying it to similar situations, reasoning by analogy. This is, for example, reflected in the way they play chess. Chess might become an active testbed for Case-Based Reasoning methods that experiment with such memory based reasoning. In the long run, chess programs could greatly benefit from such methods, allowing them to reuse plans and adapt them to different over the board situations.

## Chess

An interesting question to ask is whether chess programs will influence the way chess is played in the future. Chess, like most other things is receptive to changes, both from fashion and from new discoveries. Computers have already been used to discover new facts about various chess endgames, and to improve opening theory. The question here is more on the lines of whether they will change the style humans play. Humans often deliberately play slightly inferior moves, to surprise the opponent, complicate things, or to seek some counter chances.

The psychological effect of a surprise move is of little value against a computer program. One can argue that the programs have certain psychological advantages over the humans. For example, in the first game of the 1996 Kasparov vs. Deep Blue match, Deep Blue showed a total disregard for Kasparov's attack, calmly grabbing a pawn on the queenside while its king fortress was under seemingly disastrous threats. A human player might have shown the world champion a little more respect by taking his attack seriously. In the last game of the same match, Kasparov had a winning piece sacrifice on h7, but after thinking for a long time he played differently, apparently deferring to the judgment of the computer.

## Conclusions

It is inevitable that chess programs will one day outsmart even the strongest humans. The future programs will continue to rely on extensive search, but whether that approach alone is sufficient to reach the top is moot. More sophisticated methods are needed, that allow the programs to understand the more elaborate chess concepts. However, even though a pure brute-force approach will suffice to make a world-class chess program, the program will to be of limited interest unless it can share its knowledge. For a program to be an effective tutor, it must be able to explain various chess related concepts in a manner that is understandable and natural to us humans, instead of simply returning a raw numeric score indicating a merit of a chess position. Therefore, independent of the outcome of the Garry Kasparov vs. Deep Blue match, we believe that computer-chess will remain an important testbed for AI applications, not only in search but also in fields such as reasoning, knowledge representation, and knowledge acquisition.

## References


Hyatt, R. M.; Gower, A. E.; and Nelson, H. L. 1990. *Computers, Chess, and Cognition.* New York, NY: Springer-Verlag. chapter 7 — Cray Blitz, 111–130.

Knuth, D. E., and Moore, R. W. 1975. An analysis of alpha-beta pruning. *Artificial Intelligence* 6(4):293–326.

Marsland, T. A., and Campbell, M. S. 1982. Parallel search of strongly ordered game trees. *ACM Computing Surveys* 14(4):533–551.

Reinefeld, A. 1983. An improvement to the Scout tree search algorithm. *ICCA Journal* 6(4):4–14.

Schaeffer, J. 1989. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-11(1):1203–1212.

Scherzer, T.; Scherzer, L.; and Tjaden, D. 1990. *Computers, Chess, and Cognition.* New York, NY: Springer-Verlag. chapter 12 — Learning in Bebe, 197–216.

Slate, D. J., and Atkin, L. R. 1977. *Chess Skill in Man and Machine.* New York, NY: Springer-Verlag. chapter 4. CHESS 4.5 – Northwestern University Chess Program, 82–118.