The Games Computers (and People) Play

Jonathan Schaeffer
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1
jonathan@cs.ualberta.ca

May 10, 2000

Abstract

In the 40 years since Arthur Samuel's 1960 *Advances in Computers* chapter, enormous progress has been made in developing programs to play games of skill at a level comparable to, and in some cases beyond, what the best humans can achieve. In Samuel's time, it would have seemed unlikely that only a scant 40 years would be needed to develop programs that play world-class backgammon, checkers, chess, Othello, and Scrabble. These remarkable achievements are the result of a better understanding of the problems being solved, major algorithmic insights, and tremendous advances in hardware technology. Computer games research is one of the major success stories of artificial intelligence.

This chapter can be viewed as a successor to Samuel's work. A review of the scientific advances made in developing computer games is given. These ideas are the ingredients required for a successful program. Case studies for the games of backgammon, bridge, checkers, chess, Othello, poker, and Scrabble are presented. They are the recipes for building high-performance game-playing programs.

1 Introduction

Arthur Samuel is one of the pioneers of artificial intelligence research. Together with Claude Shannon [1] and Alan Turing [2], he laid the foundation for building high-performance game-playing programs. Samuel is best known for developing his checkers program. Over his career, he consistently sold his work as research in machine learning. His papers describing the program and its learning capabilities are classics in the literature [3, 4]. These papers are still frequently cited today, almost four decades since the original research was completed. There are few computing papers around today whose lifespan is 10 years, let alone 40.

In the years since Samuel's 1960 chapter for the first volume of *Advances in Computers*, enormous progress has been made in constructing high-performance game-playing programs. In Samuel's time, it would have seemed unlikely that within a scant 40 years checkers (8×8 draughts), Othello[1], and Scrabble[2] programs would exist that exceed the abilities of the best human players, while backgammon and chess programs could play at a level comparable to the human world champion. These remarkable accomplishments are the result of a better understanding of the problems being solved, major algorithmic insights, and tremendous advances in hardware technology. The work on computer games has been one of the most successful and visible results of artificial intelligence research. For some games, one could argue that the Turing test has been passed [5].

When talking about computer games, it is important to draw the distinction between using games as a research tool for exploring new ideas in computing, versus using computing to do research into games. The former is the subject of this chapter; the latter is not. Nevertheless, it is important to recognize that building high-performance game-playing programs has also been of enormous benefit to the respective game-playing communities. The technology has expanded human understanding of games, allowing us to explore more of the rich tapestry and intellectual challenges that games have to offer. Computers offer the key to answering some of the puzzling, unknown questions that have tantalized game aficionados. For example, computers have shown that the chess endgame of king and two bishops versus king and knight is generally a win, contrary to expert opinion [6]. In checkers, the famous 100-year position took a century of human analysis to "prove" a win; the checkers program *Chinook* takes a few seconds to prove the position is actually a draw (it is now called the 197-year position)[7].

This chapter can be viewed as a successor to Samuel's 1960 chapter, discussing the progress made in developing programs for the classic board and card games over the last four decades. A review of the scientific advances made in developing computer games is presented (Section 2). It concentrates on search and knowledge for two-person perfect-information games, and simulation-based approaches for games of imperfect or non-deterministic information. These ideas are the ingredients needed for a successful program. Section 3 presents seven case studies to highlight progress in the games of checkers, Othello, Scrabble (superior to man), backgammon, chess (comparable to the human world champion), bridge, and poker (human supremacy may be threatened). These are successful recipes for building high-performance game-playing programs.

Although this chapter discusses the scientific advances, one should not underestimate the engineering required to build these programs. One need only look at the recent success of the *Deep Blue* chess machine to appreciate the effort required. That project spanned eight years, and included several full-time people, extensive computing resources, computer chip design, and grandmaster

[1]Othello is a registered trademark of Tsukuda Original, licensed by Anjar Co.

[2]Scrabble is a registered trademark of the Milton Bradley Company, a division of Hasbro, Inc.

consultation. Some of the case studies hint at the amount of work required to build these systems. In all cases, the successes reported in this chapter are the result of consistent progress over many years.

2 Advances

The biggest advances in computer game-playing over the last 40 years have come as a result of work done on the alpha-beta search algorithm. Although this algorithm is not suitable for some of the games discussed in this chapter, it received the most attention because of the research community's preoccupation with chess. With the *Deep Blue* victory over world chess champion Garry Kasparov, interest in methods suitable for chess has waned and been replaced by activity in other games. One could argue that the chess victory removed a ball and shackle that was stifling creativity doing research on game-playing programs.

Because of the historical emphasis on search, the material in this section is heavily biased towards it. In the last decade, new techniques have moved to the forefront of games research. Two in particular are given special emphasis since they are likely to play a more prominent role in the near future:

1. Monte Carlo simulation has been successfully applied to games with imperfect or non-deterministic information. In these games it is too expensive to search all possible outcomes. Instead only a representative sample is chosen to give a statistical profile of the outcome. This technique has been successful in bridge, poker and Scrabble.

2. Temporal-difference learning is the direct descendent of Samuel's machine learning research. Thus it is fitting that this method be included in this chapter. Here a database of games (possibly generated by computer self-play) can be used to bootstrap a program to find a good combination of knowledge features. The algorithm has been successfully applied to backgammon, and has recently shown promise in chess.

This section gives a representative sample of some of the major results and research thrusts over the past 40 years. Section 2.1 discusses the advances in search technology for two-player perfect information games. Advances in knowledge engineering have not kept pace, as discussed in Section 2.2. Section 2.3 discusses the emerging simulation framework for games of non-deterministic or imperfect information. The material is intended to give a flavor of the progress made in these areas, and it is not intended to be exhaustive.

2.1 Advances in Search

The minimax algorithm was at the heart of the checkers program described in Samuel's 1960 chapter. Minimax assumes that one player tries to maximize their result (often called Max), while the other tries to minimize what Max can achieve (the Min player). The program builds a search tree of alternating moves by Max and Min. A leaf node is assigned either the game-theoretic value if known (win, loss, draw) or a heuristic estimate of the likelihood of winning (using a so-called *evaluation function*). These values are maximized (by Max) and minimized (by Min) from the leaves back to the root of the search. Within given resources (typically time), it is usually not possible to search deep enough to reach leaf nodes for which the game-theoretic result is known. The evaluation function uses application-dependent knowledge and heuristics to come up with an estimate of the winning chances for the side to move.

Consider the example in Figure 1, where maximizing nodes are indicated by squares and minimizing nodes by ovals. The root, Max (node **A**), has to choose a move that leads to positions **B**, **C**, or **D**. It is Min to play at these three positions and, similarly, Min's choice of move will lead to a position with Max to move. At the leaves of the tree are the heuristic values. These values are maximized at the Max nodes (the nodes left-to-right beginning with **E**), minimized at the Min nodes (**B**, **C**, and **D**), and maximized at the root (**A**). In this example, the minimax value of the tree is 5. The bold lines indicate the best line of play: Max will play from **A** to **C** to maximize his score, while Min will play from **C** to **H** to minimize Max's score. Max then chooses the branch leading to a score of 5, the maximum of the possible moves. The best line of play is often called the *principal variation*.

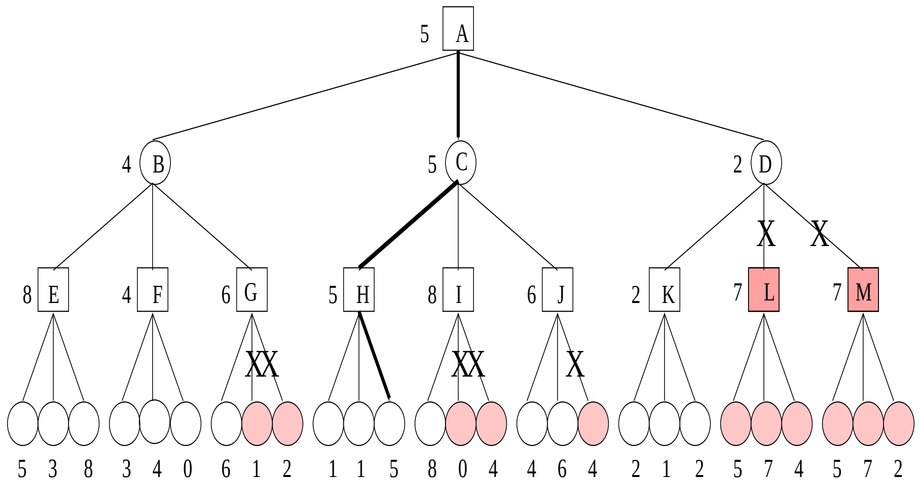


Figure 1: Searching a min-max tree.

The minimax algorithm is a depth-first, left-to-right traversal of the tree, evaluating all successors of Max and Min nodes. If one assumes that the tree has a uniform branching factor of w and a fixed depth of d moves (or *ply*), then the algorithm must examine $O(w^d)$ leaf nodes. Clearly, the exponential growth of the search tree limits the effectiveness of the algorithm.

Sometime in the late 1950s or early 1960s, the alpha-beta algorithm for searching minimax trees was invented (the algorithm may have been independently discovered, but the first publication appears in [8]). Alpha-beta simply and elegantly proves that many branches in the search tree need not be considered since they are irrelevant to the final search result.

Consider Figure 1 again. The search of nodes **E** and **F** show that the value of **B** is ≤ 4. Now, consider searching **G**. The first child of **G** has a value of 6. Thus, Max will guarantee that **G**'s value is ≥ 6. Searching **G**'s other children is pointless since they can only increase **G**'s value, which cannot affect **B**'s value. Hence further search at **G** has been proven to be unnecessary. The other children are said to be *cut-off* or *pruned*. Shaded nodes in the figure have been eliminated from the search by alpha-beta. In this example, the number of leaf nodes considered has been reduced from 27 using minimax to 16 with alpha-beta.

The alpha-beta algorithm searches a tree using two bounds: α and β. α is the minimum value that player Max has achieved. β is the maximum value to which player Min can limit Max to (conversely, the best that Max can achieve given Min's best play). Any node where a score results in the condition $\alpha \geq \beta$ causes a cut-off.

The alpha-beta algorithm is given in Figure 2.[3] For a d-ply search, it is called by:

$$\text{AlphaBeta(rootnode, } -\infty, +\infty, \text{ d, MAXNODE).}$$

Again assuming a tree of fixed branching factor w and search depth d, alpha-beta improves the best case of the search tree size to $O(w^{d/2})$ (or, to be more precise, $w^{\lceil d/2 \rceil} + w^{\lfloor d/2 \rfloor} - 1$) [9]. This best case occurs when the move leading to the best minimax score is searched first at every interior node.[4] If the worst move is search first at every node, then alpha-beta will build an $O(w^d)$ minimax tree.

Although the 20 lines of code in Figure 2 look simple, this is misleading. These are possibly the most deceptive lines of code in the artificial intelligence literature! Alpha-beta has the insidious property of hiding errors. For example, an evaluation function error may only be detected when the error happens to propagate to the root of the search tree. The deeper the search, the harder it is for the error to be minimized and maximized all the way back to the root. Consequently, many game-playing programs have bugs that survive for years before the right sequence of events occurs that allows the problem to manifest itself.[5]

In practice, a high-performance alpha-beta implementation is often 20 or more pages of code. The reason for this is the exponential difference in the search effort between the best and worst alpha-beta cases. Considerable effort has to be invested to ensure a nearly best-case result. The consequence is a

[3] The Negamax formulation is more concise [9].

[4] At nodes where a cut-off occurs, one only needs to search a move that is sufficient to immediately cause the cut-off.

[5] Empirical evidence suggests that this only happens in important tournament games!

```
int AlphaBeta( position p, int alpha, int beta, int depth, int type )
{
    /* Check for a leaf node */
    if( depth == 0 )
        return( Evaluate( p ) );

    /* Identify legal moves */
    numbmoves = GenerateMoves( p, movelist );
    if( numbmoves == 0 )
        return( Evaluate( p ) );

    if( type == MAXNODE )
         nexttype = MINNODE;
    else nexttype = MAXNODE;

    /* Call AlphaBeta recursively for each move */
    for( move = 1; move <= numbmoves; move++ )
    {
        p = MakeMove( p, movelist[ move ] );
        value = AlphaBeta( p, alpha, beta, depth-1, nexttype );
        p = UndoMove( p, movelist[ move ] );

        /* Update best value found so far */
        if( type == MAXNODE )
             alpha = MAX( value, alpha );
        else beta  = MIN( value, beta  );

        /* Check for a cut-off. Minimax without this line of code */
        if( alpha >= beta )
            break;
    }
    if( type == MAXNODE )
         return( alpha );
    else return( beta  );
}
```

Figure 2: The alpha-beta algorithm.

myriad of enhancements to alpha-beta, significantly increasing the complexity of the search process.

The main alpha-beta search enhancements can be characterized into four groups:

1. Caching information: avoiding repeated work.
2. Move ordering: increasing the likelihood of the best move being searched first at a node.
3. Search window: changing the $[\alpha, \beta]$ window to speculatively reduce search effort.
4. Search depth: dynamically adjusting the depth to redistribute search effort, attempting to maximize the value of the information gathered from the search.

Each of these enhancements is discussed in turn.

2.1.1 Caching Information

For most games, the search tree is really a misnomer; it is a search graph. Two different sequences of moves can transpose into each other. For example, in chess, the move sequence 1. d4 d5 2. ♘f3 gives rise to the same position as the sequence 1. ♘f3 d5 2. d4. Detecting these transpositions and eliminating redundant search effort can significantly reduce the search-tree size.

The *transposition table* is a cache of recently searched positions. When a subtree has been searched, the result is saved in the transposition table. Before searching a node in the tree, the table is consulted to see if it has been previously searched. If so, the table information may be sufficient to stop further search at this node. The table is usually implemented as a large hash table [10, 11].

The effectiveness of the transposition table is application dependent [12, 13]. For games like chess and checkers, where a single move changes a few squares on the board, the benefits can be massive (roughly a 75% reduction for chess and 89% for checkers for a typical search). For games like Othello, where a move can change many squares on the board, the likelihood of two move sequences transposing into each other is small (roughly a 33% reduction for a typical search).

2.1.2 Move Ordering

The exponential difference in the search-tree size between the best and worst case of alpha-beta hinges on the order in which moves are considered. At a node where a cut-off is to occur, it should be achieved with the first move searched. Hence, effort is applied at interior nodes to order the moves from the most to least likely to cause a cut-off.

The most important place for move ordering is at the root of the search. For example, if a 10-ply search is initiated without any prior preparation, the resulting search tree is likely to be large. The first move searched may provide a poor bound (α value), increasing the size of the search window used for the subsequent moves. Considering the best move first narrows the search window, increasing the chances for cut-offs in the tree.

Most alpha-beta-based programs use a technique called *iterative deepening* to maximize the chances of the best move being searched first [11]. The program starts by searching all moves 1-ply deep. The moves are then ordered based on the returned scores. The tree is then re-searched, this time 2-ply deep, and so on. The idea is that the best move for a $(d-1)$-ply search is likely to also be best for a d-ply search. By investing the overhead of repeating portions of the search, the chances are increased that the best move is considered first in the last (most expensive) iteration. Experience shows that the cost of the early iterations is a small price to pay for the large gains achieved by improved move ordering at the root of the tree. This is an important result that has been applied to many other search domains (for example, single-agent search [14]).

The idea of considering the best move first should be applied at all nodes in the tree. At interior nodes, cheaper methods are used to improve the quality of the move ordering. Three popular choices are:

- Transposition table. When recording a search result in the table, save the score and the move that leads to the best score. When the node is revisited (within the same or the next iteration), if the score information is insufficient to cause a cut-off, then the best move from the previous search can be considered first. Since the move was previously best, there is a good chance that it is still the best move.
- Application-dependent knowledge. Many games have application-dependent properties that can be exploited by a move ordering scheme. For example, in chess capture moves are more likely to cause a cut-off than non-capture moves. Hence, many programs consider all capture moves first at each node.
- History heuristic. There are numerous application-dependent move-ordering algorithms in the literature. One application-independent technique that has proved to be simple and effective is the *history heuristic* [15, 16]. A move that is best in one position is likely also best in similar positions. The heuristic maintains a global history score for each move that indicates how often that move has been best. Moves can then be ordered by their history heuristic score. A subset of this idea, the *killer heuristic*, is also popular [17].

The contrast between the transposition table and history heuristic is interesting. The transposition table stores the exact context under which a move is considered best (i.e., it saves the position with the move). The history heuristic records which moves are most often best, but has no knowledge about the context that makes the move strong. Other move ordering schemes fall somewhere in between these two extremes by, for example, adding more context to the history-heuristic moves.

Move ordering in game-playing programs is highly effective. For example, a recent study showed that the best move was searched first over 90% of the time in chess and checkers programs, and over 80% of the time in an Othello program [12]. The chess result is quite impressive when one considers that a typical position has 35 legal moves to choose from.

2.1.3 Search Window

The alpha-beta algorithm searches the tree with an initial search window of $[-\infty, +\infty]$. Usually, the extreme values do not occur. Hence, the search can be made more efficient by narrowing the range of values to consider, increasing the likelihood of cut-offs. *Aspiration search* centers the search window around the value expected from the search, plus or minus a reasonable range (δ) of uncertainty. If one expects the search to produce a value near, say, 40 then the search can be called with a search window of $[40-\delta,\ 40+\delta]$. The search will result in one of three cases:

- $40-\delta < result < 40+\delta$. The actual value is within the search window. This value has been determined with less effort than would have been required had the full search window been used.
- $result \leq 40-\delta$. The actual value is below the aspiration window (the search is said to *fail low*). To find the actual value, a second search is needed with the window $[-\infty,\ result]$.
- $result \geq 40+\delta$. The actual value is above the aspiration window (the search is said to *fail high*). To find the actual value, a second search is needed with the window $[result,\ +\infty]$.

Aspiration search is a gamble. If the result is within the search window, then the enhancement wins. Otherwise an additional search is needed. Aspiration search is usually combined with iterative deepening. The result of the $(d-1)$-ply iteration can be used as the center of the aspiration window used for the d-ply search. δ is application dependent and determined by empirical evidence.

The idea of speculatively changing the search-window size can be applied throughout the search. Consider an interior node with a search window of $[\alpha,\ \beta]$. The first move is searched and returns a score v in the search window. The next move will be searched with the window $[v,\ \beta]$. If the move ordering is effective, then there is a high probability that the best move at this node was searched first. Hence, the remaining moves are expected to return a score that is $\leq v$. Search effort can be saved by modifying the search window to prove that the remaining moves are inferior. This can be done using the window $[v,\ v+1]$. Occasionally, this assumption will be wrong, in which case a move that returns a value $v < v' < \beta$ will have to be re-searched with the new window $[v',\ \beta]$. This is the idea behind the NegaScout [18, 19] and Principal Variation Search [20] algorithms.

A search window of width one ($\beta-\alpha=1$) is called a minimal window. The idea of minimal windows can be taken to the extreme. Pearl's Scout algorithm [21] can be used to answer a Boolean question about the search value (e.g., is the root value ≥ 0?). More recently, the MTD(f) algorithm uses only minimal windows to determine the value of the root. This algorithm has been shown to be superior to alpha-beta in terms of number of nodes expanded in the search tree [12, 13].

2.1.4 Search Depth

Although alpha-beta is usually described as a fixed-depth search, better performance can be achieved using a variable search depth. The search can be compared to a stock portfolio; don't treat all stocks as being equal. You should invest in those that have the most promise, and reduce or eliminate your holdings in those that look like losers. The same philosophy holds true in search trees. If there is some hint in the search that a sequence of moves looks promising, then it may be a good idea to extend the search along that line to get more information. Similarly, moves that appear to be bad should have their search effort reduced. There are a number of ways that one can dynamically adjust the depth to maximize the amount of information gathered by the search.

Most alpha-beta-based programs have a number of application-dependent techniques for altering the search depth. For example, chess programs usually extend checking moves an additional ply since these moves indicate that something interesting is happening. Most programs have a "hopeless" metric for reducing the search depth. For example, in chess if one side has lost too much (e.g., a queen and a rook), it is very unlikely this subtree will eventually end up as part of the principal variation. Hence, the search depth may be reduced.

There are a number of techniques that may be useful for a variety of domains. In chess, *null-move searches* have been very effective at curtailing analysis of poor lines of play. The idea is that if one side is given two moves in a row and still can't achieve anything, then this line of play is likely bad. Hence, the search depth is reduced. This idea can be applied recursively throughout the search [22, 23].

Another important idea is *ProbCut* [24]. Here the result of a shallow search is used as a predictor of whether the deeper search would produce a value that is relevant to the search window. Statistical analysis of the program's searches is used to find a correlation between the values of a shallow and deep search. If the shallow search result indicates that the deeper search will not produce a value that is large enough to affect the node's value, then further effort is stopped.

Although both the null-move and ProbCut heuristics purport to be application independent, in fact they both rely on game-specific properties. Null-move cut-offs are only effective if the consequences of giving a side two moves in a row is serious. This causes problems, for example, in checkers where giving a player an extra move may allow them to escape from a position where having only one move loses (these are known as *zugzwang* positions). ProbCut depends on there being a strong correlation between the values of shallow and deep searches. For games with low variance in the leaf node values, this works well. If there is high variance, then the evaluation function must be improved to reduce the variance. In chess programs, for example, the variance is generally too high for ProbCut to be effective.

The most common form of search extension is the *quiescence search*. It is easier to get a reliable evaluation of a leaf position if that position is quiet or stable (*quiescent*). Hence, a small search is done to resolve immediate capture moves or threats [11]. Since these position features are discovered by search, this reduces the amount of explicit application-dependent knowledge required in the evaluation function.

A search-extension idea that has attracted a lot of attention is *singular extensions* [25]. The search attempts to identify forced (or *singular*) moves. This can be achieved by manipulating the search window to see if the best move is significantly better than the second-best move. When a singular move is found, then the search along that line of play is extended an additional ply (or more). The idea is that forcing moves indicate an interesting property of the position that needs to be explored further.

In addition, there are various other extensions commonly used, most based on extending the search to resolve the consequences of a threat [26, 27].

2.1.5 Close to Perfection?

Numerous studies have attempted to quantify the benefits of alpha-beta enhancements in fixed-depth searches (for example, [10, 16]). Move ordering and the transposition table usually make the biggest performance difference, with other enhancements generally being much smaller in their impact.

The size of trees built by game-playing programs appears to be close to that of the minimal alpha-beta tree. For example, in chess, *Belle* is reported to be within a factor of 2.2 [28], *Phoenix* within 1.4 [15], *Hitech* within 1.5 [28] and *Zugzwang* within 1.2 [29]. These results suggest that there is little room for improvement in fixed-depth alpha-beta searching.

The above comparisons have been done against the approximate minimal search tree. However, finding the *real* minimal tree is difficult, since the search tree is really a search graph. The real minimal search should exploit this property by:

1. selecting the move that builds the smallest tree to produce a cut-off, and
2. preferring moves that maximize the benefits of the transposition table (i.e. reuse results as much as possible).

Naturally, these objectives can conflict. In contrast to the above impressive numbers, results suggested that chess programs are off by a factor of three or more from the real minimal search graph [12, 13]. Thus, there is still room for improvements in alpha-beta search efficiency. Nevertheless, given the exponential nature of alpha-beta, that programs can search within a small constant of optimal is truly impressive.

Forty years of research into alpha-beta have resulted in a recipe for a finely tuned, highly efficient search algorithm. The program designer has a rich set of search enhancements at their disposal. The right combination is application dependent and a matter of taste. Although building an efficient searcher is well understood, deciding where to concentrate the search effort is not. It remains a challenge to identify ways to selectively extend or reduce the depth in such a way as to maximize the quality of the search result.

2.1.6 Alternative Approaches

Since its discovery. alpha-beta has been the mainstay of computer games development. Over the years, a number of interesting alternatives to alpha-beta-based searching have been proposed.

Berliner's B* algorithm attempts to prove the best move, without necessarily determining the best move's value [30, 31]. In its simplest form, B* assigns an optimistic (upper bound) and a pessimistic (lower bound) value to each leaf node. These values are recursively backed up the tree. The search continues until there is one move at the root whose pessimistic value is as good as all the alternative move's optimistic values. In effect, this is a proof that the best move (but not necessarily its value) has been found.

There are several drawbacks with B*, most notably the non-standard method for evaluating positions. It is difficult to devise reliable optimistic and pessimistic evaluation functions. B* has been refined so that the evaluations are now probability distributions. However, the resulting algorithm is complex and needs considerable application tuning. It has been used in the *Hitech* chess program, but even there the performance of alpha-beta is superior [31].

McAllester's conspiracy numbers algorithm tries to exploit properties of the search tree [32]. The algorithm records the minimal number of leaf nodes in a search tree that must change their value (or *conspire*) to change the value of the root of the tree. Consider a Max node having a value of 10. To raise this value to, say, 20, only one of the children has to have its value become 20. To lower the value to, say, 0, all children with a value greater than 0 must have their value lowered. Conspiracy numbers works by recursively backing up the tree the minimum numbers of nodes that must change their value to cause the search tree to become a particular value. The algorithm terminates when the effort required to change the value at the root of the search (i.e., conspiracy number) exceeds a predefined threshold.

Conspiracy numbers caused quite a stir in the research community because of its innovative aspect of measuring resistance to change in the search. Considerable effort has been devoted to understanding and improving the algorithm. Unfortunately it has a lot of overhead (for example: slow convergence, cost of updating the conspiracy numbers, maintaining the search tree in memory) which has been an impediment to its usage in high-performance programs. A variation on the original conspiracy numbers algorithm has been successfully used in the *Ulysses* chess program [33].

There are other innovative alternatives to alpha-beta, each of which is worthy of study. These include BPIP [34], min/max approximation [35], and meta-greedy algorithms [36].

Although all these alpha-beta alternatives have many desirable properties, none of them is a serious challenger to alpha-beta's dominance. The conceptual simplicity of the alpha-beta framework makes it relatively easy to code and highly efficient at execution time. The alpha-beta alternatives are much harder to code, the algorithms are not as well understood, and there is generally a large execution overhead. Perhaps if the research community devoted as much effort to understanding these algorithms as they did in understanding alpha-beta, we would see a new algorithm come to the fore. Until that happens, alpha-beta will continue to dominate as the search algorithm of choice for two-player perfect information games.

2.1.7 Conclusions

Research on understanding the alpha-beta algorithm has dominated games research since its discovery in the early 1960's. This process was accelerated by the discovery of the strong correlation of program performance with alpha-beta search depth [37]. This gave a simple formula for success: build a fast search engine. This led to the building of special-purpose chips for chess [38] and massively parallel alpha-beta searchers [29].

Search alone is not the answer. Additional search eventually leads to diminishing returns in the benefits achievable [39]. Eventually, there comes the point where the most significant performance gains are to be had by identifying and implementing missing pieces of application knowledge. This was evident, for example, in the 1999 world computer chess championship, where the deep-searching, large multi-processor programs finished behind the shallower-searching, PC-based programs that used more chess knowledge.

For many popular games, such as chess, checkers, and Othello, alpha-beta has been sufficient to achieve world-class play. Hence, there was no need to look for alternatives. For artificial-intelligence purists, this is an unsatisfactory result. By relying on so-called *brute-force searching*, these programs can minimize their dependence on knowledge. However, for other games, most notably Go, search-intensive solutions will not be effective. Radically different approaches are needed.

2.2 Advances in Knowledge

Ideally, no knowledge other than the rules of the game should be needed to build a strong game-playing program. Unfortunately, for interesting games it is usually too deep to search to find the game-theoretic value of a position. Hence knowledge for differentiating favorable from unfavorable positions has to be added to the program. Nevertheless, there are some cases where the program can learn position values without using heuristic knowledge.

The first example is the transposition table. This is a form of rote learning. By saving information and reusing it, the program is learning, allowing it to eliminate nodes from the search without searching. Although the table is usually thought of as something local to an individual search, "important" entries can be saved to disk and used for subsequent searches. For example, by saving some transposition table results from a game, they may be used in the next game to avoid repeating the same mistake [40, 41].

A second example is endgame databases. Some games can be solved from the end of the game backwards. One can enumerate all positions with one piece on the board, and record which positions are wins, losses, and draws. These results can be backed up to solve all positions with two pieces on the board, and so on. The result is an *endgame database* containing perfect information. For chess, most of the five-piece endgames have been solved, with some six-piece endgames also solved [6]. This is of limited value, since most games are over before such a simplified position is reached. In checkers, all eight-piece endgames have been solved [42]. The databases play a role in the search of the first move of a game! Endgame databases have been used to solve the game of Nine Men's Morris [43].

A third form of knowledge comes from the human literature. Most games have an extensive literature on the best opening moves of the game. This information can be collected in an *opening book* and made available to the program. The book can either be used to select the program's move, or as advice to bias the program's opening move selection process. Many programs modify the opening book to tailor the moves in it to the style of the program.

When pre-computed or human knowledge is not available, then the game-playing program must fall back on its evaluation function. The function assigns scores to positions that are a heuristic assessment of the likelihood of winning (or losing) from the given position. Application-dependent knowledge and heuristics are usually applied to a position to score features that are indicators of the true value of the position.

The program implementor (usually in consultation with a domain expert) will identify a set of *features* (f) that can be used to assess the position. Each feature is given a *weight* (w) that reflects how important that feature is in relation to the others in determining the overall assessment. Most programs use a linear combination of this information to arrive at a position value:

$$value \ = \ \sum_{i=1}^{n} w_i \times f_i \qquad (1)$$

where n is the number of features. For example, in chess two features that are correlated with success are the material balance and pawn structure (f_1 and f_2). Material balance is usually much more important than pawn structure, and hence has a much higher weighting ($w_1 \ >> \ w_2$).

Identifying which features might be correlated with the final result of the game is still largely done by hand. It is a complex process that is not well understood. Usually the features come from human experience. However, human concepts are often vague and hard to define algorithmically. Even well-defined concepts may be impractical because of the computational overhead. One could apply considerable knowledge in the assessment process, but this increases the cost of performing an evaluation. The more expensive the evaluation function is to compute, the smaller the search tree that can be explored in a fixed amount of time. Thus, each piece of knowledge has to be evaluated on what it contributes to the accuracy of the overall evaluation, and the cost (both programmer time and execution time) of having it.

Most evaluation functions are carefully tuned by hand. The knowledge has been judiciously added, taking into account the expected benefits and the cost of computing the knowledge. Hence, most of the knowledge that is used is of a general-purpose nature. Unfortunately, it is the exceptions to the knowledge that cause the most performance problems. As chess grandmaster Kevin Spraggett said [42]:

> I spent the first half of my career learning the principles for playing strong chess and the second half learning when to violate them.

Most game-playing program's evaluation functions attempt to capture the first half of Spraggett's experience. Implementing the second half is often too difficult and computationally time consuming, and generally has a small payoff (except perhaps at the highest levels of play).

Important progress has been made in setting the weights automatically. Although this seems like it should be much easier than building an evaluation function, in reality it is a laborious process when done by hand. Automating this process would result in a huge reduction in the effort required to build a high-performance game-playing program.

Temporal difference learning has come to the fore as a major advance in weighting evaluation function features. Samuel pioneered the idea [3, 4], but it only became recognized as a valuable learning algorithm after Sutton extended and formalized this work [44]. Temporal difference learning is at the heart of Tesauro's world-championship-caliber backgammon program (see Section 3.1), and has shown promising results in chess (discussed later in this section).

Temporal difference learning (TDL) is a reinforcement learning algorithm. The learner has an input state, produces an output action, and later receives feedback (commonly called the reward) on how well its action performed. For example, a chess game consists of a series of input states (positions) and actions (the move to play). At the end of the game, the reward is known: win, loss, or draw. In between the start and the end of the game, a program will use a function to map the inputs onto the outputs (decide on its next move). This function is a predictor of the future, since it is attempting to maximize its expected outcome (make a move that leads to a win). The goal in reinforcement learning is to propagate the reward information back along the game's move sequence to improve the quality of actions (moves) made. This is accomplished by attributing the credit (or blame) to the outputs that led to the final reward. By doing so, the learner's evaluation function will change, hopefully in such a way as to be a better predictor of the final reward.

To achieve the large-scale goal of matching inputs to the result of the game, TDL focuses on the smaller goal of modifying the learner so that the current prediction is a better approximation of the next prediction [44, 45]. Consider a series of predictions $P_1, P_2, ...P_N$ on the outcome of a game. These could be the program's assessment of the likelihood of winning from move to move. In chess, the initial position of a game, P_1, has a value that is likely close to 0. For a win $P_N = 1$ while a loss would have $P_N = -1$. For the moves in between, the assessments will vary.

If the likelihood of winning for position t (P_t) is less (more) than that of position $t + 1$ (P_{t+1}), then we would like to increase (decrease) the value of position t to be a better predictor of the value of $t + 1$. The idea behind temporal difference learning is to adjust the evaluation based on the incremental differences in the assessments. Thus,

$$\triangle \; = \; P_{t+1} - P_t$$

measures that difference between the prediction for move $t+1$ and that for move

$$\triangle w_t \ = \ \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \bigtriangledown_w P_k$$

where:

- w is the set of weights being tuned,
- t is the time step being altered, in a sequence of moves from $1, 2, ..., N-1$,
- $\triangle w_t$ is the change in the set of weights at step t as a result of applying temporal differences,
- P_t is the prediction at time step t (for the end of the game, P_N, the final outcome is used),
- λ $(0 \leq \lambda \leq 1)$ controls how much credit gets propagated back to the previous estimates ($\lambda = 0$ implies no feedback, while $\lambda = 1$ would have all previous moves sharing equally),
- $\bigtriangledown_w P_k$ is the set of partial derivatives for each component of w, and
- $\alpha > 0$ is the rate of learning (a small α causes small incremental changes; a large α makes larger steps).

λ and α are heuristic parameters that need to be tuned for each application domain.

Figure 3: The TD(λ) algorithm.

t. This adjustment can be done by modifying the weights of the evaluation function to reduce the $\triangle$ from move to move.

Temporal difference learning is usually described with a variable weighting of recency. Rather than considering only the previous move, one can consider all previous moves with non-uniform weights (usually exponential). These moves should not all be given the same importance in the decision-making process, since the evaluation of moves made many moves previously are less likely to be relevant to the current evaluation. Instead, previous moves are weighted by λ^p, where p reflects how far back the move is. The parameter λ controls how much credit is given to previous moves, giving exponentially decaying feedback of the prediction error over time. Hence, this algorithm is called TD(λ). Figure 3 gives the temporal difference relation used by TD(λ).

A typical application of TDL is for a program with an evaluation function, but unknown weights for the features. By playing a series of games, the program gets feedback on the relative importance of features. TDL propagates this information back along the move sequence played, causing incremental changes to the feature weights. The result is that the evaluation function values get tuned to be better predictors.

In addition to Tesauro's success in backgammon (Section 3.1), there are two recent TDL data points in chess. First, *Cilkchess*, currently one of the strongest chess programs, was tuned using temporal difference learning and the results are encouraging. Don Dailey, a co-author of *Cilkchess*, writes that [46]:

> Much to my surprise, TDL seems to be a success. But the weight set that comes out is SCARY; I'm still afraid to run with it even though it beats the hand-tuned weights. They are hard to understand too, because TDL expresses chess concepts any way that is convenient for it. So if you create a heuristic to describe a chess concept, TDL may use it to "fix" something it considers broken in your weight set.

An interesting data point was achieved in the *KnightCap* chess program [47]. Starting with a program that knew only about material and had all other evaluation function terms weighted with zero, the program was able to quickly tune its weights to achieve an impressive increase in its performance. The authors recognized that the predictions of a chess program were the result of an extensive search, and the score at the root of the tree was really the value of the leaf node on the principal variation. Consequently, the temporal difference learning should use the principal variation leaf positions, not the positions at the root of the search tree [48]. This algorithm has been called TDLeaf(λ) [47].

These successes are exciting, and offer the hope that a major component of building high-performance game-playing programs can be automated.[6]

2.3 Simulation-Based Approaches

In the 1990s, research into non-deterministic and imperfect information games emerged as an important application for artificial-intelligence investigations. In many ways, these domains are more interesting than two-player perfect-information games, and promise greater long-term research potential. Handling imperfect or probabilistic information significantly complicates the game, but is a better model of the vagaries of real-world problems.

For non-deterministic and imperfect information games, alpha-beta search does not work. The branches in the search tree represent probabilistic outcomes based on, for example, the roll of the dice or unknown cards. At best one can back up probabilities of expected outcomes. For these games it is usually impractical to build the entire game tree of all possibilities.

Simulations can be used to sample the space of possible outcomes, trying to gather statistical evidence to support the superiority of one action.[7] The program can instantiate the missing information (e.g. assign cards or determine dice rolls), play the game through to completion, and then record the result. This can be repeated with a different assignment of the missing information.

[6] An excellent survey of machine learning applied to games can be found in [49].

[7] Some of the material in this section has been taken from [50].

By repeating this process many times, a statistical ranking of the move choices can be obtained.

Consider the imperfect-information game of bridge. The declarer does not know in which hand each of the 26 hidden cards are. The simulator can instantiate one possible assignment of cards to each opponent, and then play the hand through to completion (a *trial*). Thus, a single data point has been obtained on the number of tricks that can be won. This can then be repeated by dealing a different set of cards to each opponent. When these simulated hands have been repeated a sufficient number of times, the statistics gathered from these runs can be used to decide on a course of action. For example, it may be that in 90% of the samples a particular card play led to the best result. Based on this evidence, the program can then decide with high confidence what the best card to play is.

For each trial in the simulation, one instance of the non-deterministic or unknown information is applied. Hence, a representative sample of the search space is looked at to gather statistical evidence on which move is best. Figure 4 shows the pseudo-code for this approach. Some of its characteristics include:

1. The program iterates on the number of samples taken.

2. The search for each sample usually goes to the end of the game.

3. Heuristic evaluation usually occurs at the interior nodes of the search to determine a subset of branches to consider, reducing the cost of a sample (and allowing more samples to be taken).

The simulation benefits from selective samples that use information from the game state (i.e. such as the bidding auction in bridge), rather than a uniform distribution or other fixed distribution sampling technique.

Statistical sampling has noise or variance. The sampling must be done in a way that captures the reality of the situation, ruling out impossible scenarios and properly reflecting the likelihood of improbable scenarios. The more representative the samples, the less the variance is likely to be. *Selective sampling* refers to carefully choosing the simulation data to be as representative as possible [50].

It is important to distinguish selective sampling from traditional Monte Carlo techniques. Selective sampling uses information about the game state to skew the underlying probability distribution, rather than assuming uniform or other fixed probability distributions. Monte Carlo techniques may eventually converge on the right answer, but selective sampling allows for faster convergence and less variance.

As an example, consider the imperfect-information and non-deterministic game of Scrabble. Brian Sheppard, author of *Maven*, writes that for his simulations he generates [51]:

> ... a distribution of racks that matches the distribution actually seen in games. In *Maven* we use a uniform distribution of the rack, and we take steps to ensure that every tile is represented as often as it

```
/* From a given state, simulate and return the best move */
move Simulator( known_state state )
{
  obvious_move = NO;
  trials = 0;
  while( ( trials <= MAX_TRIALS ) and ( obvious_move == NO ) )
  {
    trials = trials + 1;
    /* Generate the missing information */
    missing_info = selective_sampling_to_generate_missing_information;
    numbmoves = GeneratePlausibleMoves( state, missing_info, movelist );
    /* Consider all moves */
    for( m = 1; m <= numbmoves; m++ )
    {
      state = MakeAction( state, movelist[ m ], missing_info );
      value[m] = value[m] + Search( state );
      state = UndoAction( state, movelist[ m ], missing_info );
    }
    /* Test to see if one move is statistically better than all others */
    if( ∃ i such that value[ i ] >> value[ j ](∀ j, j ≠ i) )
    {
      obvious_move = YES;
    }
  }
  /* Return the move with the highest score */
  return( move i || value[ i ] >= value[ j ] (∀ j, j ≠ i) );
}
```

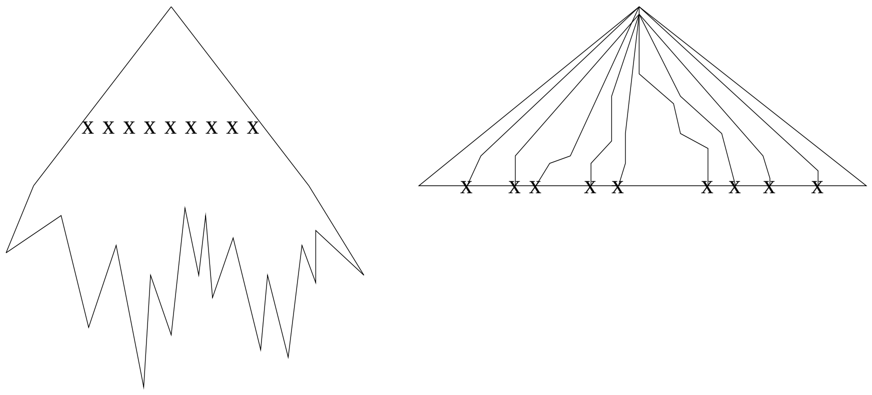
Figure 4: Simulation-based search.

> should be. We do this without introducing statistical bias by always including in the opponent's tiles for the next iteration the one tile that has been most underrepresented among all previous racks.

Selective sampling need not be perfect. In Scrabble, the opponent's tiles do not come from a uniform distribution: opponents tend to play away bad letters and keep good letters. Sheppard is convinced that this small refinement to the model of the opponent's hands would make little difference in the simulation results.

An important feature of the simulation-based framework is the notion of an obvious move. Although many alpha-beta-based programs incorporate an obvious move feature, the technique is usually *ad hoc* and the heuristic is the result of programmer experience rather than a sound analytic technique. In the simulation-based framework, an obvious move is statistically well-defined. As more samples are taken, if one choice exceeds the alternatives by a statistically significant margin, one can stop the simulation early and commit to the action, with full knowledge of the statistical validity of the decision.

It is interesting to compare alpha-beta and simulation-based search methods. Alpha-beta considers all possible moves at a node that cannot be logically eliminated; simulation-based search can only look at a representative sample. Whereas alpha-beta search typically has a depth limitation, most simulation-based programs follow a path from the root of the search to the end of the game. Thus, one can characterize alpha-beta search trees as having large width, but limited depth. Simulations, on the other hand, typically have limited breadth but large depth. Figure 5 illustrates the differences in these two approaches, where an "x" is used to indicate where evaluations occur in the search.



(a) Alpha-beta search (b) Simulation search

Figure 5: Contrasting search methods.

Simulations are used in many branches of science, but have only recently emerged as a powerful tool for constructing high-performance game-playing programs. They have proven to be effective in backgammon, bridge, poker, and Scrabble. This technique deserves to be recognized as an important framework for building game-playing programs, equal in stature to the alpha-beta model.

2.4 Perspectives

Enormous progress has been made in understanding the algorithms needed to build game-playing programs. Most of this work has been driven by the desire to satisfy one of the early goals of artificial intelligence research, building a program capable of defeating the human world chess champion. Hence, most games-related research has concentrated on alpha-beta search. With the chess success on the horizon, many researchers branched out into other games. As a result, research efforts on two-player perfect-information games has moved to the background. New vistas are being explored, with temporal-difference learning and simulations being samples of the current research thrusts.

The research in developing algorithms for game playing has applicability to other application domains, but the community of researchers involved have done a poor job selling the technology. For example, many of the search techniques pioneered with alpha-beta have become standard in other search domains (e.g., iterative deepening), with few realizing the lineage of the ideas.

3 Advances in Computer Games

This section summarizes the progress made in a number of popular games. These include games where computers are better than all humans (checkers, Othello, and Scrabble), are as good as the human world champion (backgammon and chess), and some where human supremacy may be challenged in the near future (bridge and poker). Each section contains a brief history of program development for that game, a case study on the best program in the area, and a representative sample of their play. The case study highlights interesting or unique aspects of the program.

The histories are necessarily brief. I apologize in advance to the many hard-working researchers and hobbyists whose work is not mentioned here.

3.1 Backgammon

The first concerted effort at building a strong backgammon program was undertaken by Hans Berliner of Carnegie Mellon University. In 1979 his program, *BKG9.8*, played an exhibition match against the the newly-crowned world champion Luigi Villa [52, 53]. The stakes were $5,000, winner take all. The final score was 7-1 in favor of the computer, with *BKG9.8* winning four of the five games played (the rest of the points came from the doubling cube).

Backgammon is a game of both skill and luck. In a short match, the dice can favor one player over another. Berliner writes that "In the short run, small percentage differences favoring one player are not too significant. However, in the long run a few percentage points are highly indicative of significant skill differences" [53]. Thus, assessing the results of a five-game match are difficult. Afterwards Berliner analyzed the program's play and concluded that [52]:

> There is no doubt that *BKG9.8* played well, but down the line Villa played better. He made the technically correct plays almost all the time, whereas the program did not make the best play in eight out of 73 non-forced situations.

BKG9.8 was an important first step, but major work was still needed to bring the level of play up to that of the world's best players.

In the late 1980s, IBM researcher Gerry Tesauro began work on a neural-net-based backgammon program. The net used encoded backgammon knowledge and, training on data sets of games played by expert players, learned the weights to assign to these pieces of knowledge. The program, *Neurogammon*, was good enough to win first place in the 1989 Computer Olympiad [54].

Tesauro's next program used a neural network that was trained using temporal difference learning. Instead of training the program with data sets of games played by humans, Tesauro was successful in having the program learn using the temporal differences from *self-play* games. The evolution in *TD-Gammon* from version 0.0 to 3.0 saw an increase in the knowledge used, a larger neural net, and the addition of small selective searches. The resulting program is acknowledged to be on par with the best players in the world, and possibly even better.

In 1998, an exhibition match was played between world champion Malcolm Davis and *TD-Gammon* 3.0. To reduce the luck factor, 100 games were played over three days. The final result was a narrow eight-point win for Davis. Both Davis and Tesauro have done extensive analysis of the games, coming up with similar conclusions [55]:

> While this analysis isn't definitive, it suggests that we may have witnessed a superhuman level of performance by *TD-Gammon*, marred only by one horrible blunder redoubling to 8 in game 16, costing a whopping 0.9 points in equity and probably the match!

3.1.1 *TD-Gammon*

Backgammon combines both skill and luck. The luck element comes from the rolls of the dice, making conventional search techniques impractical. A single roll of the dice results in one of 21 distinct combinations, each of which results in an average of 20 legal moves to consider. With a branching factor of over 400, many of which are equally likely and cannot be pruned, brute-force search won't be effective.

TD-Gammon is a neural network that takes as input the current board position and returns as output the score for the position (roughly, the probability of winning) [56]. The neural network acts as the evaluation function. Each of the connections in the neural net is parameterized with a weight. Each node is a function of the weighted sum of each of its inputs, producing an equity value as output.

The neural net has approximately 300 input values (see Figure 6) [45, 57]. For each of the 24 points on the board, there are four inputs for each player giving the number of pieces they have on that point. Additional inputs for each side are the number of pieces on the bar, the number of pieces taken off the board, and whose turn it is. The likelihood of achieving a gammon or a backgammon are also input. The remaining 100 inputs are from functions that compute positional features, taken from the *Neurogammon* program. The inputs to the net were chosen to simplify the system, and not to minimize the number of inputs.

TD-Gammon 2.0 used no backgammon knowledge and had a neural net containing 80 hidden units. This program was sufficient to play strong backgammon, but not at a world-class level. Tesauro was able to improve the program's performance to be world-class caliber by adding *Neurogammon*'s backgammon knowledge as input to the neural net. This version, *TD-Gammon* 3.0, contains 160 hidden units in the neural network. Each unit in the net takes a linear sum of the weighted values of its input, and then converts it to a value in the range

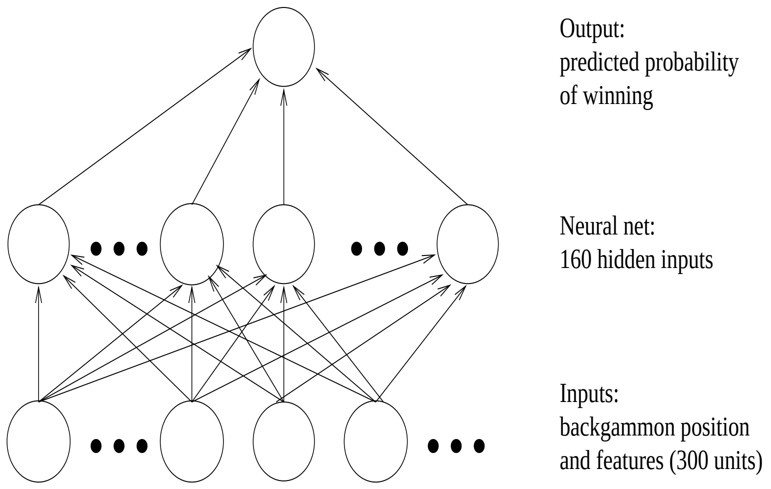


Figure 6: *TD-Gammon* 3.0's neural network.

-3 to 3. A backgammon is worth three points, a gammon two, and a win, one point. The conversion is done with a nonlinear sigmoid function, allowing the output to be a nonlinear function of the inputs. The resulting neural net has approximately 50,000 weights that need to be trained.

The weights in the hidden units were trained using temporal difference learning from self-play games. By playing the program against itself, there was an endless supply of data for the program to train itself against. In a given game position, the program uses the neural net to evaluate each of the roughly 20 different ways it can play its dice roll, and then chooses the move leading to the maximum evaluation. Each game is played to completion, and then temporal difference learning is applied to the sequence of moves. Roughly 1,500,000 self-play games were used for training *TD-Gammon* 3.0.

TD-Gammon has been augmented with a selective three-ply search. For each of its moves, *TD-Gammon* considers the most likely opponent responses, and its replies to those responses. Each state considered in the search has roughly 400 possibilities, so for each of the 21 dice rolls, *TD-Gammon* only considers a handful of likely best moves for the opponent (selectively paring down the search).

A critical component of strong backgammon is the handling of the doubling cube. The cube strategy was added after the program was trained. It uses a theoretical doubling formula developed by mathematicians in the 1970s [58]. During a game, *TD-Gammon*'s reward estimates are fed into this formula to come up with an approximation of the expected doubling payoff.

Post-mortem analysis of backgammon games use simulations (or *roll-outs* as they are called in the backgammon community). A roll-out consists of repeatedly simulating the play from a starting position through to the end of the game. Each trial consists of a different sequence of dice rolls. Each move decision is based on a one-ply search. A simulation is stopped after 10,000 trials or when a move becomes statistically better than all the alternatives.

3.1.2 The Best of Computer Backgammon

The following game was the 18th played in the exhibition match between *TD-Gammon* 3.0 and world champion Malcolm Davis, held at the 1998 conference of the American Association for Artificial Intelligence. The game comments are by Gerry Tesauro (GT), Malcolm Davis (MD) and *TD-Gammon* (TD). TD gives the top moves in a position, ordered by their score. These values were determined after the match by roll-outs. Tesauro explains how to interpret the scores [59]:

> Ignoring gammons and backgammons, if player's move decision is 0.1 worse than the best move, the player has reduced his winning chances by about 5%, and backgammon experts would regard that as a "blunder." On the other hand, if the error is 0.02 or less, it only costs about 1% in winning chances, and such errors are regarded as small.

Each of the 24 points is numbered and given relative to the side to move (White is counter clockwise from their home; Black is clockwise from their home). A move consists of 1 to 4 checkers being moved, each specified with their from- and to-points. The bar is labeled as point number 25. An × indicates a capture move. In the following text, for each turn the side to move (Black or White) is given, followed by the dice roll and the moves chosen.

Black: Malcolm Davis — White: *TD-Gammon* 3.0

B 5,1: 24-23 13-8; W 4,2: 8-4 6-4; B 6,2: 24-18 18-16; W 6,4: 24-18 13×9; *MD: A good play. Hitting twice is reasonably close. GT: Good play by TD. Aggressively blitzing with 13×9 8×2 or 8×2 6-2 is not bad, but committal. TD's play keeps more options open and seems to be a more solid all-around positional play, and in fact it comes out on top in the roll-out results. TD: 24-18 13×9 = 0.252; 13×9 8×2 = 0.216; 8×2 6-2 = 0.207.*

B 4,2: 25-23 13-9; *MD: A total toss-up versus 8-4. Going to the 9 point is more my style, ignoring the 3 duplication. GT: Whoops, the roll-outs say that 8-4 is slightly better. Not only is it fewer shots, but it's also a better point if missed and covered. Not sure what MD was thinking here. TD: 25-23 8-4 = -0.389; 25-23 13-9 = -0.407.*

W 6,6: 24-18 13-7 13-7 13-7; *MD: Not challenging. Going to the 3 point duplicates 1's and is about a 4% error. GT: I have to confess that this one is beyond me. I would have held on to the midpoint and slotted the 3 point with the last six. TD's play gives up a point, leaves two blots instead of one, in not very good locations, and yet it wins the roll-out. I guess what's going on is that TD's play leaves a bunch of builders to make the five point, which is perhaps the key point in this position. The midpoint is not so valuable when White already*

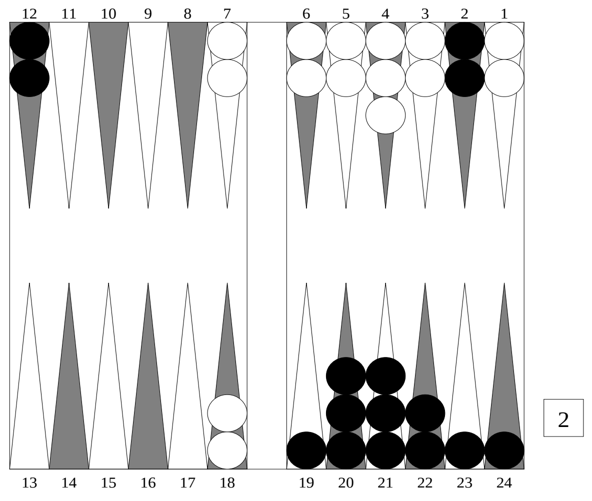


Figure 7: Malcolm Davis (Black) versus *TD-Gammon* 3.0 (White).
White to play a 2-1.

owns the 18 point, and playing 9-3 makes it very unlikely that White will be able to make the five point anytime soon. Well done, TD! TD: 24-18 13-7 13-7 13-7 = 0.516; 24-18 13-7 13-7 9-3 = 0.444.

B 4,3: 9-5 8-5; *MD: High marks to the human. Hitting is a slight error, perhaps more so if the cube is about to be turned, as it creates a more volatile position. TD: 9-5 8-5 = -0.479; 23-20 20×16 = -0.482.*

White doubles, Black takes; *MD: This double is apparently a little too aggressive, although not at all unreasonable. GT: Based on 10,000 full roll-outs with the cube, the double is barely correct. The no-double equity is 0.73, whereas the equity after double/take is 0.75.*

W 3,3: 8-5 8-5 6-3 6-3; *MD: TD's biggest piece movement error. Making the 15 and 3 points is about 2% better. GT: Many options here: one can attempt to disengage with 18-15 18-15, safety the blot, make the 5 point, or make the 3 point. TD makes the best play hanging back on the 18 point and keeping all full-contact options open. However, the position thematically points to disengaging since White is far ahead in the race and has gotten all the back checkers out. The roll-outs reveal that this is better than TD's choice. TD: 18-15 18-15 6-3 6-3 = 0.547; 8-5 8-5 6-3 6-3 = 0.522.*

B 1,1: 13×12 12-11 11-10 6-5; *MD: My biggest error – I was playing quickly and didn't consider the much better 6-4 6-4. GT: A tough choice for MD. He can hit the blot while he has the chance, or wait and build up his board with 6-4 6-4 and hope that TD has trouble clearing the 18 point. If he's going to hit, perhaps he should hit safely with 13×12 13-12 13-12 6-5. The choice is not clear to me, but the roll-outs say that MD's play is a big mistake. TD: 6-5 6-5 5-4 5-4 = -0.466; 13×12 13-12 13-12 6-5 = -0.489; (5th ranked move) 13×12 12-11 11-10 6-5 = -0.549.*

W 5,2: 25×23 23-18; B 5,1: 10-5 5-4; W 6,3: 18-15 15-9; B 4,2: 8-4 5-3; W 4,1: 9-5 7-6; B 5,1: 8-3 6-5; *MD: A photo compared to 6-5 6-1. GT: Bold play by MD, risking an immediate fatal hit in order to keep a nice inner board structure. I would have chickened out and played safe with 6-1 4-3. The roll-outs come out about equal. TD: 6-1 4-3 = -0.535; 8-3 6-5 = -0.544; 6-5 6-1 = -0.545.*

W 6,5: 6-1 9-3; B 5,2: 8-3 6-4; *MD: Perhaps a very good play. GT: If MD slots the ace point, it could be a liability in the event of immediate action after TD rolls a five. Instead, he chooses 6-4, avoiding the blot, but making his own fives awkward. The roll-outs give a statistically insignificant edge to 3-1. TD: 8-3 3-1 = -0.401; 8-3 6-4 = -0.414.*

W 2,1: 3-1 5-4; B 5,1: 6-1 3-2; *MD: Perhaps a very good play by the human. Leaving only one blot gives up about 2%. GT: Talk about fives being awkward! MD makes an outstanding play here, leaving three blots in the home board, but keeping a smooth distribution and hoping to straighten everything out next turn. I would have played the more craven 4-3 or 5-4, which turns out badly in the roll-outs. Being able to play the bad rolls well is the hallmark of a champion. TD: 6-1 3-2 = -0.657; 6-1 4-3 = -0.716; 6-1 5-4 = -0.728.*

W 2,1: 18-16 18-17; See Figure 7. *MD: A fine play. GT: Brian Sheppard, who was in the audience at the time, applauded TD for this spectacularly un-computer-like play. White could easily leave no blots with 7-6 7-5, and keep all the points with 4-1. However, TD realizes that it's ahead in the race and behind in timing, and that if it waits on the 18 point, it may well have to break it next turn, when MD's board will likely be cleaned up and much stronger. So this is an excellent time to run with 18-17 18-16. Black's board is such a mess that he probably won't hit even if he can. TD: 18-17 18-16 = 0.718; 18-16 16-15 = 0.668; 4-3 3-1 = 0.558; 7-6 7-5 = 0.550.*

B 4,2: 5-1 4-2; W 4,1: 16-15 15-11; *MD: Straightforward. GT: Another fine play by TD. It's tempting to button up and leave one blot and fewer shots with 17-16 7-3. However, this leaves a difficult point to clear next turn, plus it's not a good idea to allow Black to accomplish both hitting and escaping with fives next turn. TD: 16-15 15-11 = 0.424; 17-16 7-3 = 0.392.*

B 3,3: 13-10 13-10 10-7 10-7; W 3,2: 11-9 9-6; B 4,3: 7-4 6-2; *MD: Close, but the best play. GT: MD saves a six in the outfield so he won't have to break the 23 point next turn. An eminently reasonable idea, but curiously 7-3 7-4 comes out a tiny bit better in the roll-outs, quite possibly due to sampling noise. TD: 7-4 7-3 = -0.763; 7-4 6-2 = -0.777.*

W 2,1: 6-5 17-15; B 6,4: 23-17 7-3; *MD: Not clear and very close. GT: A tough choice. MD boldly breaks the anchor and leaves two blots, rather than wreck his board with 7-1 5-1. Comparing apples and oranges is often difficult for humans and, here, the roll-outs say that safety is better. TD: 7-1 5-1 = -0.822; 23-17 7-3 = -0.865.*

W 4,1: 7-3 3×2; *MD: TD is fearless. Hitting is right by a huge margin. GT: A scary play, but it's often been said that "Computers don't get scared." TD: 7-3 3×2 = 0.575; 7-6 7-3 = 0.514.*

B 2,1: 25-23 17-16; W 2,1: no move; Black redoubles, White passes; *MD: Against a human it would be right to double if there's any chance at all that the cube might be taken. However, there's no chance of that against a [TD-Gammon], so this is a small cube error. GT: Whoops! This position is actually too good to redouble! Black does slightly better by holding the cube and trying to win a gammon by picking up the second blot. 10,000 roll-outs with the cube indicate an equity advantage of 0.04 to playing on instead of cashing.*

Tesauro's postmortem analysis of the match strongly suggests that *TD-Gammon* was the better player [55]:

> I rolled out every position in the Davis–TD match where the doubling cube was turned (full roll-outs with the cube, no settlements). There were 130 such positions. In 72 positions, *TD-Gammon* doubled:
>
> 1. TD made 63 correct doubles and 9 incorrect doubles; total equity loss 1.25.
> 2. MD made 56 correct take/pass decisions and 16 incorrect; total equity loss 2.60.
>
> In 58 positions, Malcolm Davis doubled:
>
> - MD made 46 correct doubles and 12 incorrect; total equity loss 1.58.
> - TD made 54 correct take/pass decisions and 4 incorrect; total equity loss 0.19.
>
> Of course, to get the whole story, we also have to check all the positions where a player could have doubled but didn't. It's infeasible to roll out all these positions, but I did do roll-outs of each of the 130 "turn-before" positions, to see if a player missed a double the turn before the cube was actually offered. To summarize those results:
>
> - In 72 positions, TD correctly waited in 67 and missed doubles in 5; total equity loss 0.25.
> - In 58 positions, MD correctly waited in 45 and missed doubles in 13; total equity loss 1.24.
>
> However, 4 of MD's "errors" were at the end of the match when he was playing conservatively to protect his match lead. If we ignore these then he only missed doubles in 9 positions, for a total equity loss of 0.61.
>
> Malcolm has also done a preliminary analysis with *Jellyfish* [a commercial program] of the checker plays, which indicated that TD played better. (The fact that TD obtained more opportunities to double than MD also suggests it was moving the pieces better.)

3.2 Bridge

Work on computer bridge began in the early 1960s ([60], for example), but it wasn't until the 1980s that major efforts were made. The advent of the personal computer spurred on numerous commercial projects that resulted in programs with relatively poor capabilities. Perennial world champion Bob Hamman once remarked that the commercial programs "would have to improve to be hopeless" [61]. A similar opinion was shared by another frequent world champion, Zia Mahmood. In 1990, he offered a prize of £1,000,000 to the person who developed a program that could defeat him at bridge. At the time, this seemed like a safe bet for the foreseeable future.

In the 1990s, several academic efforts began using bridge for research in artificial intelligence [62, 63, 61, 64, 65]. The commercial *Bridge Baron* program teamed up with Dana Nau and Steve Smith from the University of Maryland. The result was a program that won the 1997 world computer bridge championship. The program used a hierarchical task network for the play of the hand. Rather than building a search tree where each branch was the play of a card, they would define each branch to be a strategy, using human-defined concepts such as finesse and squeeze [64, 65]. The result was an incremental improvement in the program's card play, but it was still far from being world-class caliber.

Beginning in 1998, Mathew Ginsberg's program *GIB* started dominating the computer bridge competition, handily winning the world computer bridge championship. The program started producing strong results in competitions against humans, including an impressive result in an exhibition match against world champions Zia Mahmood and Michael Rosenberg. The match lasted two hours, allowing 14 boards to be played. The result was in doubt until the last hand, before the humans prevailed by 6.31 IMPs (International Match Points). This was the first notable man-machine success for computer bridge-playing programs. Zia Mahmood, impressed by the rapid progress made by *GIB*, withdrew his million pound prize.

GIB was invited to compete in the Par Contest at the 1998 world bridge championships. This tournament tests the contestant's skills at playing out bridge hands. In a select field of 35 of the premier players in the world, the program finished strongly in 12th place. Michael Rosenberg won the event with a score of 16,850 out of 24,000; *GIB* scored 11,210. Of the points lost by *GIB*, 1,000 were due to time (there was a 10 point penalty per minute spent thinking), 6,000 were due to *GIB* not understanding the auction, and 6,000 were due to *GIB*'s inability to handle some hands where the correct strategy involves combining different possibilities [61]. The latter two issues are currently being addressed.

3.2.1 *GIB*

The name *GIB* originally stood for "Goren In a Box", a tribute to one of the pioneers of bridge. Another interpretation is "Ginsberg's Intelligent Bridge."

To play out a hand, a variation of alpha-beta search can be used. The average branching factor is roughly 4. Alpha-beta pruning and transposition tables reduces it to approximately 1.7. Ordering moves at interior nodes of the search to favor those moves that give the opponent the least number of possible responses (i.e. preferring small sub-trees over large ones), further reduces the branching factor to 1.3. Given the depth of the search (to the end of the hand; possibly a tree of depth 52), the trees are surprisingly small (on the order of 10^6 nodes).

Ginsberg's *partition search* algorithm is used to augment the search [63]. Partition search is a "smart" transposition table, where different hands that have inconsequential differences are treated as the same hand, significantly increasing the number of table hits. For example, from a transposition table's point of view, the hands "♠ K Q 8 4 2" and "♠ K Q 8 4 3" are different. However, by representing the entry as "♠ K Q 8 X X", where "X" denotes any small card, the analysis on the first hand can be applied to the second hand. The result of adding partition search reduces the average search tree size for a deal to a remarkably small 50,000 nodes.

To decide how to play a hand, *GIB* uses a simulation [61]. For each trial, cards are dealt to each opponent that are consistent with the play thus far. Typically 50 deals are used in the simulation; the card play that results in the highest expected number of tricks won is chosen to be played. Simulations are not without their disadvantages. An important component to the play of the hand are so-called information-gathering plays. A trick is played (and possibly lost) to reveal more information on the makeup of the opponent's hands. Unfortunately, since a simulation involves assigning cards to the opponents, the program has perfect knowledge of where all the cards lie and, within a given trial, information gathering plays are not needed! This demonstrates a limitation of perfect-information variants of imperfect-information reality.

Most previous attempts at bridge bidding have been based on an expert-defined set of rules. This is largely unavoidable, since bidding is an agreed-upon convention for communicating card information. *GIB* takes this one step further, building on the ability to quickly simulate a hand [61]. The program has access to a large database of bidding rules (7,400 rules from the commercial program *Meadowlark Bridge*). At each point in the bidding, *GIB* queries the database to find the set of plausible bids. For each bid, the rest of the auction is projected using the database, and then the play of the resulting contract is simulated. *GIB* chooses the bid that leads to the average best result for the program.

Although intuitively appealing, this approach does have some problems. Notably, as with opening books in other games, the database of rules may have gaps and errors in it. Consider a rule where the response to the bid 4♠ is incorrect in the database. *GIB* will direct its play towards this bid because it assumes the opponent's will make the (likely bad) database response. As Ginsberg writes, "it is difficult to distinguish a good choice that is successful because the opponent has no winning options from a bad choice that *appears* successful because the heuristic fails to identify such options" [61].

GIB uses three partial solutions to the problem of an erroneous or incomplete bidding system. First, the bidding database can be examined by doing extensive off-line computations to identify erroneous or missing bid information. This is effective, but can take a long time to complete. Second, during a game, simulation results can be used to identify when a database response to a bid leads to a poor result. This may be evidence of a database problem, but it could also be the result of effective disruptive bidding by *GIB*. Finally, *GIB* can be biased to make bids that are "close" to the suggested database bids, allowing the program the flexibility to deviate from the database.

To summarize, *GIB* is well on the way to becoming a world-class bridge player. The program's card play is already at a world-class level (as evidenced by the Par Contest result), and current efforts will only enhance this. The bidding needs improvement, and this is currently being addressed. Had Zia Mahmood not withdrawn his offer, he might have lost his money within a couple of years from now.

3.2.2 The Best of Computer Bridge

The following hand is board 11 of the 1998 exhibition match between *GIB* and world champions Zia Mahmood and Michael Rosenberg, held at the annual conference of the American Association for Artificial Intelligence in 1998. The humans won the match by 6.31 IMPs over 14 deals.

North: Zia
♠ A J 9
♡ 7 3
♢ K Q J 10 8 5 3
♣ K

West: *GIB*1
♠ 10 7 6 2
♡ J 6 4
♢ 7 2
♣ A 10 9 3

East: *GIB*2
♠ K Q 4
♡ A 10
♢ 9 6
♣ 8 7 6 5 4 2

South: Rosenberg
♠ 8 5 3
♡ K Q 9 8 5 2
♢ A 4
♣ Q J

South	West	North	East
		1♢	pass
1♡	pass	3♢	pass
3♡	pass	3♠	pass
4♢	pass	4♡	pass
pass	pass		

Opening lead: 2♠

Figure 8: Mahmood-Rosenberg versus two *GIB*s.

The hand shown in Figure 8 was analyzed by Mike Whittaker and reported in *Bridge Magazine* [66],[8] commenting on *GIB*'s defensive play.

GIB1, *West, led with a small ♠, won by the Queen.* GIB2 *switched to a ♢, won by dummy's Jack. Leading a ♡ to the King won, and Rosenberg then led a ♣, won by* GIB1. *A second ♢ lead was won by the Ace and Rosenberg tried a ♠ to the Jack, losing to the King.* GIB2 *cashed the Ace ♡ before leading a small ♠ to dummy's Ace. Rosenberg found himself locked in dummy, forced to lead a ♢. This had the effect of promoting the Jack ♡ for* GIB2 *and Rosenberg finished two down.*

Finally, we come to the FAQ (Frequently Asked Question): will the computers ever triumph against top quality human opposition? The idea has always been laughed at but I would not be too complacent. Before long the sheer computing power of the computer will give it a definite edge over even the best human declarer in contracts that require technical expertise. However, I think that the complexities of the bidding language, the use of deception in play and defense and some abstract qualities, such as table presence, will keep the humans ahead, at least for a while.

Figure 9 is used to illustrate *GIB*'s stellar play of the hand. The analysis was done by Onno Eskes and reported in *IMP* magazine [67][9].

West opens 2 ♣, showing a weak hand with both major suits. Unpleasant, but on the other hand, it becomes a lot easier for us to stay out of a ♡ contract. We confidently reach 7 ♢. West leads the Jack ♠. I greet dummy with approval. "Well bid," I remember thinking. I count five trumps, twice Ace-King-Queen, Ace ♠ and a ruff in dummy. What can go wrong?

Trumps four-zero. If left-hand opponent has them all, I will go down. If opponent on the right has them, I can finesse against his Jack. But then I cannot ruff a ♡ anymore. Well, then they'd better not be four-zero. I take Ace ♠ and lead a ♢ to the Ace. West discards a ♠. I curse under my breath and start thinking again. Are there any chances left? In ♣, maybe? If those are four-four, I can discard a loser on the fifth ♣.

[8]Reproduced with permission. Minor editing changes have been made to conform with the style of this chapter.

[9]Reproduced with permission. Minor editing changes have been made to conform with the style of this chapter.

North
♠ A 3
♡ 7 4
♢ Q 8 7 4
♣ A K Q 6 3

West East

South
♠ Q 2
♡ A K Q 10 8 3
♢ A K 10 3 2
♣ —

Contract: 7♢
Opening lead: J♠

Figure 9: Illustrating *GIB*'s play of the hand.

But I have only one entry left. I should have started by ruffing a ♣. "One down," I concede, "how are the clubs divided?" "Four-four," is the painful reply. Fortunately for us, exactly the same thing happened at the other table!

The next day I am still disgusted with the hand. It is a nice problem, however. I decide to present the hand to GIB*... I enter the hands and the auction. I also enter the explanation of the bids (West at least 4-4 in the majors, less than opening strength) and the opening lead. Then the computer starts to bubble. After 30 seconds it produces Ace ♠. I tell it which card East plays to the first trick. Again 30 seconds of thinking. Ace ♣! I let East and West follow small. The computer discards Queen ♠. Another 30 seconds. Small ♣, ruffed in hand! Forlorn, I watch the computer finish the rest of the play in immaculate fashion. Ace ♢ (discovering the bad trump split), ♢ to the Queen, King ♣, Queen ♣ and a good ♣. East ruffs, South over-ruffs and he can now ruff his last ♡ in dummy.*

Beaten by the computer! The humiliation is complete when the machine subtly announces that it just scored plus 2140.

3.3 Checkers

Arthur Samuel began thinking about a checkers program in 1948 but did not start coding until a few years later. He was not the first to write a checkers-playing program; Strachey pre-dated him by a few months [68]. Over the span of three decades, Samuel worked steadily on his program with performance taking a back seat to his higher goal of creating a program that learned. Samuel's program is best known for its single win against Robert Nealey in a 1963 exhibition match. From this single game, many people erroneously concluded that checkers was a "solved" game.

In the late 1970's, a team of researchers at Duke University built a strong checkers-playing program that defeated Samuel's program in a short match [69]. Early success convinced the authors that their program was possibly one of the 10 best players in the world. World champion Marion Tinsley effectively debunked that, writing that: “The programs may indeed consider a lot of moves and positions, but one thing is certain. They do not see much!” [70]. Efforts to arrange a match between the two went nowhere and the Duke program was quietly retired.

Interest in checkers was rekindled in 1989 with the advent of strong commercial programs and a research effort at the University of Alberta: *Chinook*. *Chinook* was authored principally by Jonathan Schaeffer, Norman Treloar, Robert Lake, Paul Lu, and Martin Bryant. In 1990, the program earned the right to challenge for the human world championship. The checkers federations refused to sanction the match, leading to the creation of a new title: the world man-machine championship. This title was contested for the first time in 1992, with Marion Tinsley defeating *Chinook* in a 40-game match by a score of 4 wins to 2. *Chinook*'s wins were the first against a reigning world champion in a non-exhibition event for any competitive game.

There was a rematch in 1994, but after six games (all draws), Tinsley resigned the match and the title to *Chinook*, citing health concerns. The following week he was diagnosed with cancer, and he died eight months later. *Chinook* has subsequently defended its title twice, and has not lost a game since 1994. The program was retired in 1997 after it became clear that there was no living person whose abilities came close to that of the program [42].

Chinook is the first program to win a human world championship. At the time of this writing, the gap between *Chinook* and the highest-rated human is 200 rating points (using the chess rating scale) [42], making it unlikely that humans will ever improve to *Chinook*'s level of play.

3.3.1 Chinook

In his 1960 *Advances in Computers* chapter, Samuel felt bad about using the article to describe his work instead of his predecessor, Strachey. The same comment that Samuel wrote in 1960 applies to this section, after substituting Schaeffer's name for Samuel's and replacing Strachey with Samuel: “While it is grossly unfair to dismiss Samuel's work in a single paragraph and to discuss the present writer's own efforts in some detail, in the interests of conciseness this will have to be done. Perhaps such high-handed behavior can be excused if the writer publicly apologizes for his action, as he does now, and publicly acknowledges the credit which Dr. Samuel is due.”

The structure of *Chinook* is similar to that of a typical chess program: search, knowledge, opening book, and endgame databases [42, 71]. *Chinook* uses alpha-beta search (NegaScout) with a myriad of enhancements including iterative deepening, transposition table, history heuristic, search extensions, and search reductions. Further performance is provided by a parallel search algorithm. With 1994 technology, an 18-processor Silicon Graphics Power Challenge, Chinook was able to average a minimum of 19-ply searches against Tinsley with search extensions occasionally reaching 45 ply into the tree. The median position evaluated was typically 25-ply deep into the search.

The search depths achieved are usually sufficient to uncover most tactical threats in a position, however they are inadequate to resolve positional subtleties. Hence, considerable computational effort is devoted to identifying promising lines of play to extend the search, and futile lines to reduce the search depth. Experiments show that a program searching 17 ply plus extensions will defeat a program going 23 ply deep without extensions (each program used the same amount of time for each search). By most standards, giving up 6-ply of search for the extensions is extraordinarily high. However, players like Tinsley have consistently demonstrated an ability to analyze 30-ply deep (or more), so *Chinook* has to be able to match this capability.

The evaluation function was manually tuned over a period of five years. For each of four game phases, it consists of 25 features combined by a linear function. Interestingly, most of the features in Samuel's program were not used in *Chinook* — many of them were there to overcome the limitations of the shallow search depths that Samuel could achieve using 1960's hardware. The evaluation function was carefully tuned by a checkers expert through extensive trial-and-error testing. Attempts at automatically tuning the evaluation function were unsuccessful.

There were two major improvements to the evaluation function that are of interest. First, in 1992 a major change enhanced the program's knowledge but resulted in a two-fold reduction in the number of positions that the program could analyze per second (effectively costing it one ply of search) [42]. Despite the reduced search proficiency, the new knowledge significantly improved the quality of the evaluations, resulting in a stronger program. This was strong evidence that *Chinook*'s search depths were hitting diminishing returns for additional search efforts [39]; more was to be gained by the addition of useful knowledge than additional search.

The second refinement was allowing the sum of positional scores to be able to exceed the value of a checker. In principle, this is dangerous since the program may prefer large positional scores over material ones. However, a critical component in human grandmaster play is the ability to recognize the exceptions; when material is a secondary consideration. Adding this capability eliminated a serious source of errors, and was a major reason for *Chinook*'s excellent result in the 1992 world man-machine championship.

Chinook uses an endgame database containing all checkers positions with eight or fewer pieces. This database has 444 billion (4×10^{11}) positions, compressed into six gigabytes for real-time decompression. Unlike chess programs which are compute-bound, *Chinook* becomes I/O-bound after a few moves in a game. The deep searches mean that the database is occasionally being hit on the *first move* of a game. The databases introduce accurate values (win/loss/draw) into the search, reducing the heuristic error. In many games, the program is able to backup a draw score to the root of a search within 10 moves by each side from the start of a game. This suggests that it may be possible to determine the game-theoretic value of the starting position of the game (one definition of "solving" the game).

Chinook has access to a large database of opening moves compiled from the checkers openings literature. This extensive opening book allows the program to play its first moves from the opening database, come out of the book, and then usually be able to search deeply enough to find its way into the endgame database. This implies that the window for program error is very small. In the 1994 *Chinook*–Tinsley match, five of the six games followed this pattern (in the other game, Tinsley made an error and *Chinook* had to try for a win). All the positions in the opening book were verified using at least 19-ply searches. This uncovered numerous errors in the published literature.

A database of all known games played by Marion Tinsley was compiled. When the program was out of its opening book, this database could be used to bias the search. For example, when playing the weaker side of an opening, the program would include a favorable bias towards any move that Tinsley had previously played in this position. The idea is that, since Tinsley rarely made a mistake, his move is likely to be the right choice. When playing the stronger side of the opening, the database was used for a different purpose. By biasing the search *against* moves suggested by the database, the program could increase the chances of playing a new move, thereby throwing the human opponent onto their own resources (increasing the chance of human error).

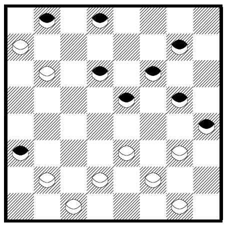
Arthur Samuels' program did not come close to reaching the pinnacle of checkers. In part, this was because of the limited hardware resources that he had available to him at the time. But it was also due to his insistence on developing a program that learned everything by itself. Samuel wrote in his 1960 chapter that "suggestions that [I] incorporate standard openings or other forms of man-devised checker lore have been consistently rejected. ... [I] refuse to pass judgment on whether the program makes good moves for the right reasons, demanding instead, that it develop its own reasons" [72]. Ironically, a major reason for the success of *Chinook* was the use of the "man-devised" lore that Samuel consistently rejected.

3.3.2 The Best of Computer Checkers

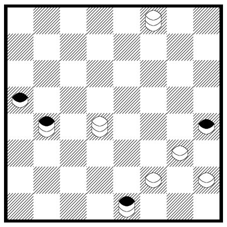
In the 1992 world man-machine championship, the match was tied at one win apiece at the start of game 14. The annotations are based on those appearing in [42]. Comments in italics are from Marion Tinsley. The game notation is identical to that used in chess: columns are labeled a to h from left to right, and the rows are labeled 1 to 8 from bottom to top. An $\times$ indicates a capture move.

Black: Marion Tinsley — White: Chinook

1. h6-g5 a3-b4 2. b6-c5 Since the standard starting position is drawish, tournament checkers uses the so-called "Three Move Ballot". The first three moves of the game (two by Black, one by White) are chosen randomly, resulting in some interesting, lop-sided starting positions. Hence two games are played for each opening, with the opponents switching colors after the first game. At the time of this match, 142 openings had been approved for tournament play. *Chinook* has to have opening knowledge for both sides of all the openings.



(a) **16. ... a5-b6**



(b) **34. ... e3-d4**

Figure 10: Tinsley (Black) versus *Chinook* (White).

2. ... b4-a5 3. g5-f4 g3×e5 4. d6×f4 e3×g5 5. f6×h4 d2-e3 6. g7-f6 c3-d4 7. e7-d6 d4×b6 8. a7×c5 b2-c3 9. h8-g7 c3-d4 *Chinook* is now out of its opening book. The program reports having a small advantage.

10. c5-b4 e1-d2 11. d6-e5 d4-c5 12. g7-h6 *I once laughed at* [grandmaster Willie] *Ryan for forgetting his own published play, but no more! Back in 1948, I gave the better f6-g5 to draw... Pat McCarthy* [a top British player] *later asked me why I didn't take this simple route. The answer? I had simply forgotten it!* Tinsley seems to think g7-h6 is a bad move, but *Chinook* sees nothing wrong with it.

12. ... h2-g3 *Chinook* expects c7-d6 in reply, with an even game.

13. h6-g5 Tinsley has no comment on this move, but *Chinook* thinks it is a major mistake. Analysis conducted months after the game indicates that after h6-g5 the game is probably lost. Tinsley attaches the blame to his previous move which apparently leads to a position that he feels uncomfortable with, even though it leads to a draw (assuming perfect play).

13. ... a1-b2 14. b4-a3 c5-b6 Suddenly *Chinook* believes it has a huge advantage (over one-half of a checker). This may not be obvious by looking at the position; it is the result of searching over 20 plies into the future.

15. f8-g7 b6-a7 This illustrates an important lesson about putting knowledge into a program: every piece of knowledge has its exceptions. Normally, putting a man into the *dog-hole* (a7 for White; h2 for Black) is bad, but here it turns out to be very strong. Unfortunately, *Chinook* always penalizes the dog-hole; it does not understand any of the exceptions. *After this b6-a7, I never saw a glimpse of a draw.*

16. c7-d6 a5-b6 See Figure 10a. Incredible! *Chinook* is sacrificing a checker against Tinsley. Prior to this match, *Chinook* had a history of misassessing these type of sacrifices, resulting in some bad losses. Tinsley himself identified this as a serious weakness in the program. Fortunately, a few days before the match began, a serious problem in *Chinook*'s evaluation function was uncovered (and fixed) that explained the program's poor play in these type of positions.

17. b8-c7 a7-b8=k Now the preceding moves make sense. This sacrifice has been in the works for a few moves now and Tinsley has been avoiding it. Now he has run out of safe moves and is forced to accept it. It is hard to believe that Black can survive. White has a mobile king and strong back rank (making it hard for Black to get a king). What is Black to do?

18. c7×a5 b8-a7 19. d8-e7 a7-b8 Is *Chinook* winning? The position looks very strong, but *Chinook* reports only a moderate advantage. The program intended to play b2-c3, but at the last minute switches to a7-b8 by a narrow margin. The alpha-beta search differentiates between the moves by an insignificant (and likely random) 3/100ths of the value of a checker. After the game, Tinsley revealed that he was praying for b2-c3, as he claimed that it led to a draw. a7-b8 may be the only winning move.

20. g7-h6 g1-h2 21. d6-c5 b8-c7 22. e5-d4 c7-d6 Winning back the checker with a dominating position. Unfortunately, *Chinook*'s score does not reflect this. It was searching so deep that it must have found a way for Black to extricate himself.

23. f6-e5 d6×f8 24. g5-f4 e3×g5 25. h6×f4 f8-e7 26. c5-b4 d2-c3 At this point, *Chinook*'s analysis revealed why the assessment had been low over the past few moves: it saw that it was winning a checker but thought Black could draw despite the checker deficit. Now the search is able to see far enough ahead to realize that the draw is unlikely.

27. b4×d2 c1×e3×g5 28. a3×c1=k e7-d6 29. d4-c3 d6×f4 30. c3-d2 g5-f6 *Chinook* announces it has found a forced win in its endgame database. For this match, the program had access to all the seven-piece positions, and a small portion of the eight-piece endgames.

31. d2-e1=k f6-e7 32. c1-b2 f4-e3 33. b2-c3 e7-f8=k 34. c3-b4 e3-d4 Tinsley resigns. See Figure 10b. The winning line goes as follows: b4-a3 f8-e7 a5-b4 d4-e3 b4-c3 e7-d6 c3-b2 d6-e5 b2-c1=k, and now g3-f4 frees White's checkers. After e1×g3, then f4-g5 surprisingly traps the king. The dominant White kings control the center of the board, keeping Black's pieces at bay.

After Tinsley extended his hand in resignation, the crowd rushed forward to congratulate Tinsley. Congratulate Tinsley? "That's a fine draw," exclaimed Grandmaster Con McCarrick, the match referee. Once the truth was revealed, the spectators were stunned. The audience thought that Tinsley had found a beautiful drawing line!

With *Chinook* leading 2-1, Tinsley looked like he was in trouble. However, *Chinook* forfeited game 18 due to technical problems and then was out-played in game 25. In the last game of the match, trailing 3-2 and needing a win, *Chinook* was pre-programmed to treat a draw as a loss. The program saw a draw, rejected it, and went on to lose the game and the match.

3.4 Chess

The progress of computer chess was strongly influenced by an article by Ken Thompson which equated search depth with chess-program performance [37]. Basically, the paper presented a formula for success: build faster chess search engines. The milestones in chess program development become a statement of the state-of-the-art in high-performance computing:

- 1980-1982: Thompson's *Belle*, the first program to earn a U.S. master title, was a machine built to play chess. It consisted of 10 large wire-wrapped boards using LSI chips [73].
- 1983-1984: *Cray Blitz* used a multi-processor Cray supercomputer [74].
- 1985-1986: The *Hitech* chess machine was based on 64 special-purpose VLSI chips (one per board square) [75, 28].
- 1985-1986: *Waycool* used a 256-processor hypercube [76].
- 1987-present: *ChipTest* (and its successors *Deep Thought* and *Deep Blue*) took VLSI technology even further to come up with a chess chip [38, 77, 78].

In 1987, *ChipTest* shocked the chess world by tieing for first place in a strong tournament, finishing ahead of a former world champion and defeating a grandmaster. The unexpected success aroused the interest of world champion Garry Kasparov, who played a two-game exhibition match against the program in 1989. Man easily defeated machine in both games.

The *Deep Blue* team worked for seven years on improving the program, including designing a single-chip chess search engine and making significant strides in the quality of their software. In 1996, the chess machine played a six-game exhibition match against Kasparov. The world champion was stunned by a defeat in the first game, but he recovered to win the match, scoring three wins and two draws to offset the single loss. The following year, another exhibition match was played. *Deep Blue* scored a brilliant win in game two, handing Kasparov a psychological blow that he never recovered from. In the final, decisive game of the match, Kasparov fell into a trap and the game ended quickly. This gave *Deep Blue* an unexpected match victory, scoring two wins, three draws and a loss.

It is important to keep this result in perspective. First, it was an exhibition match; *Deep Blue* did not *earn* the right to play Kasparov.[10] Second, the match was too short to accurately determine the better player; world-championship matches are typically at least 20 games long. Although it is not clear just how good *Deep Blue* is, there is no doubt that the program is a strong grandmaster.

What does the research community think of the *Deep Blue* result? Many are filled with admiration at this feat of engineering. Some are cautious about the significance. John McCarthy writes that [79]:

> In 1965, the Russian mathematician Alexander Kronrod said, "Chess is the Drosophila[11] of artificial intelligence." However, computer chess has developed much as genetics might have if the geneticists had concentrated their efforts starting in 1910 on breeding racing

[10] To be fair, it is unlikely that the international chess federation will ever allow computers to compete for the world championship.

[11] The drosophila is the fruit fly. The analogy is that the fruit fly is to genetics research as games are to artificial intelligence research.

Drosophila. We would have some science, but mainly we would have very fast fruit flies.

In retrospect, the chess "problem" turned out to be much harder than anyone thought in Samuel's time. The *Deep Blue* result is a tremendous achievement, and a milestone in the history of both artificial intelligence and computing science.

From the scientific point of view, it is to be regretted that *Deep Blue* has been retired, the hardware unused, and the programming team disbanded. The scientific community has a single data point that suggests machine might be better than man at chess. The data is insufficient and the sample size is not statistically significant. Moreover, given the current lack of interest in *Deep Blue* from IBM, it is doubtful that this experiment will ever be repeated. Of what value is a single, non-repeatable data point?

3.4.1 Deep Blue

Deep Blue and its predecessors represents a decade-long intensive effort by a team of people. The project was funded by IBM, and the principal scientists who developed the program were Feng-hsiung Hsu, Murray Campbell, and Joe Hoane.

Deep Blue's speed comes from a single-chip chess machine. The chip includes a search engine, a move generator, and an evaluation function [38]. The chip's search algorithm is alpha-beta, but it is restricted to always use a minimal window. Transposition tables are not implemented on the chip (it would take too much chip real estate). The search is capable of doing a limited set of search extensions. The evaluation function is implemented as small tables on the chip; the values for these tables can be downloaded to the chip before the search begins. These tables are indexed by board features and the results summed in parallel to provide the positional score.

A single chip is capable of analyzing over two million chess positions per second. It is important to note that this speed understates the chip's capabilities. Some operations that are too expensive to implement in software can be done with little or no cost in hardware. For example, one capability of the chip is to selectively generate subsets of legal moves, such as all moves that can put the opponent in check. These increased capabilities give rise to new opportunities for the search algorithm and the evaluation function. Hsu estimates that each chess chip position evaluation roughly equates to 40,000 instructions on a general-purpose computer. If so, then each chip translates to a 100 billion instruction per second chess supercomputer [38].

Access to the chip is controlled by an alpha-beta search algorithm that is implemented on the host computer (an IBM SP-2). *Deep Blue* uses alpha-beta with iterative deepening and transposition tables. Considerable effort was devoted to researching search extensions. The *Deep Blue* team pioneered the idea of singular extensions, using local search to identify forced moves [25]. Other extensions include those for threats and piece influence [27]. Extensive tuning was done to find the right combination of extensions that maximized the benefits while not causing an explosion in search effort. In *Deep Blue*, a search extension would increase the search depth by an amount up to two ply. The algorithm used fractional extensions (e.g. a threat might increase the search depth by 0.5 ply), allowing several features to combine to cause a search extension.

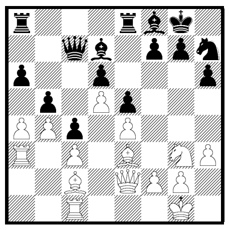
The search has been parallelized. For the 1997 Kasparov match, *Deep Blue* used a 30-processor IBM SP-2, with each processor connected to 16 chess chips. The parallel search algorithm uses a three-level hierarchy. The first four ply are done by a single master process. The leaves of the master's tree are searched an additional four or more ply deeper by the other SP-2 processors. The parallel search running on the SP-2 uses a variant on Hsu's Delayed Branch Tree Expansion algorithm [80]. The leaf nodes of these searches are passed off to the chess chips for additional search. In effect, one could view the chips as performing a sophisticated evaluation using at least a four-ply search, plus extensions. During the Kasparov match, *Deep Blue* "only" searched 200 million positions per second on average. The maximum hardware speed is roughly one billion positions per second (30 processors $\times$ 16 chips per processor $\times$ 2 million positions per second). The difference reflects both the difficulty of achieving a high degree of parallelism with alpha-beta, and the team decision that more efficient searching was unlikely to have an impact against Kasparov.

The biggest difference in *Deep Blue*'s performance in 1997 compared to 1996 was undoubtedly due to improved chess knowledge. Chess grandmaster Joel Benjamin worked with the team to identify weaknesses in the program's play. The evaluation function consists of over 8,000 tunable parameters. Most of the terms are combined linearly to arrive at a position value, but some terms are scaled to create a non-linear relationship. Although several attempts were made to tune the parameters automatically, in the end the tuning was primarily done by hand.

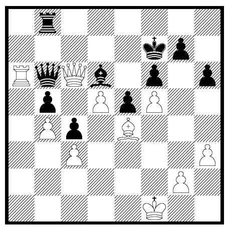
The program uses an endgame database of all positions with five or fewer pieces on the board, although this is rarely a factor in a game. The opening book is small, but *Deep Blue* has access to a large database of grandmaster games. When the program is out of its book, it will query the grandmaster database to find all games where the board position has arisen. All moves for this position retrieved from the database are assessed, based on who played it and how favorably the game ended. These moves receive a positive or negative bias that influences the evaluation of a line of play. Essentially, moves with a history of success are favored, and those with a bad track record are discouraged.

3.4.2 The Best of Computer Chess

In the 1997 match against world champion Garry Kasparov, *Deep Blue* lost the first game, causing many to predict an easy victory for man over machine. The second game astounded the chess world, convincingly demonstrating *Deep Blue*'s championship-caliber skills. In the game notation, columns are labeled *a* to *h* from left to right and the rows are labeled 1 to 8 from bottom to top. An



(a) 24. ♖a3



(b) 45. ♖a6

Figure 11: Deep Blue (White) versus Kasparov (Black).

× indicates a capture move.

White: *Deep Blue* — Black: Gary Kasparov

1. e4 e5 2. ♘f3 ♘c6 3. ♗b5 a6 4. ♗a4 ♘f6 5. O–O ♗e7 6. ♖e1 b5 7. ♗b3 d6 8. c3 O–O 9. h3 h6 10. d4 ♖e8 11. ♘bd2 ♗f8 12. ♘f1 ♗d7 13. ♘g3 ♘a5 14. ♗c2 c5 15. b3 ♘c6 16. d5 ♘e7 17. ♗e3 ♘g6 18. ♕d2 Until now, standard opening theory. Kasparov repeatedly claimed that he understood how to play against computers. Indeed this position is consistent with the common perception that computers are weak in closed (blocked) positions. Nevertheless, with his next move Kasparov begins to appear complacent, perhaps underestimating his dangerous foe.

18. ... ♘h7 19. a4 ♘h4 20. ♘×h4 ♕×h4 21. ♕e2 ♕d8 Black's last few moves have accomplished nothing except to exchange a pair of knights on the opposite side of the board from all the action.

22. b4 ♕c7 23. ♖ec1 With this move, the audience began to sense that something was different with *Deep Blue*'s play, compared to the quality of play seen in the 1996 match. From the human's point of view, this move shows extraordinary insight into the position. At first glance, it looks like the rook is being moved to a useless square. However, this is a "prophylactic move" that subtly restricts Black's options. The move becomes very strong if Black allows the *c*-file to become open.

23. ... c4 24. ♖a3 See Figure 11a. Another strong move, intending to double the rooks on the *a*-file. Most computer programs would immediately exchange *a*-pawns. Joel Benjamin revealed afterward that *Deep Blue* had a common computer tendency to release tension by exchanging pawns. Special knowledge was added to refrain from these exchanges, thereby maximizing the computer's options in the position.

24. ... ♖ec8 25. ♖ca1 ♕d8 26. f4 Another strong positional move that is "obvious" to humans, but usually difficult to find for computers. Having secured the advantage on the queen-side, the program now strives to dominate the king-side. Subsequent analysis showed that this move may not have been strongest. Although the idea is good, 26. a×b5 a×b5 27. ♗a7 would allow White to make inroads on the queen-side.

26. ... ♘f6 27. f×e5 d×e5 28. ♕f1 Another human-like move. 28. ♕f2 may have been even stronger.

28. ... ♘e8 29. ♕f2 ♘d6 30. ♗b6 ♕e8 31. ♖3a2 ♗e7 Kasparov has drifted into a horribly passive position. He can only wait for *Deep Blue* to attack.

32. ♗c5 ♗f8 33. ♘f5 ♗×f5 34. e×f5 f6 35. ♗×d6 ♗×d6 36. a×b5 At first glance, ♕b6 seems to win material. However e4 gives Black counter-play.

36. ... a×b5 37. ♗e4 Black's position is miserable, and everyone expected the seemingly crushing 37. ♕b6. However, there is a hidden trap: 37. ... ♖xa2 38. ♖xa2 ♖a8 38. ♖xa8 ♕xa8 39. ♕xd6 ♕a1+ 40. ♔h2 ♕c1 with a probable draw. In some lines, Black can play e4 and get (limited) counter-play. 37. ♗e4 upset Kasparov: the move eliminates all counter-chances. Kasparov couldn't believe that the program would pass up the chance to win material. This position gave rise to considerable controversy after the match. Kasparov's disbelief that a computer was capable of this level of sophistication resulted in his leveling unfounded accusations of cheating against the *Deep Blue* team.

37. ... ♖×a2 38. ♕×a2 ♕d7 39. ♕a7 ♖c7 40. ♕b6 ♖b7 41. ♖a8+ ♔f7 42. ♕a6 ♕c7 43. ♕c6 ♕b6+ 44. ♔f1 An error, but no one knew it at the time...

44. ... ♖b8 45. ♖a6 Kasparov resigns. See Figure 11b. The audience erupted in applause. History was made! But — incredible as it seems — the final position is a draw! The analysis is long and difficult, but the amazing ♕e3 secures a miraculous draw. Even the incredible search depths of *Deep Blue* were incapable of finding this within the time constraints of a game.

Much has been made of Kasparov's missed opportunity. However, this distracts the discussion from the real issue: *Deep Blue* played a magnificent game. Who cares if there is a minor imperfection in a masterpiece? Most classic games of chess contain many flaws. Perfect chess is still an elusive goal, even for Kasparov and *Deep Blue.*

Despite the defeat, even Kasparov had grudging respect for his electronic opponent [81]:

> In *Deep Blue*'s Game 2 we saw something that went well beyond our wildest expectations of how well a computer would be able to foresee the long-term positional consequences of its decisions. The machine refused to move to a position that had a decisive short-term advantage — showing a very human sense of danger. I think this moment could mark a revolution in computer science...

Kasparov pressed hard for a win in games 3, 4, and 5 of the match. In the end, he seemed to run out of energy. In game 6, he made an uncharacteristic mistake early in the game and *Deep Blue* quickly capitalized. The dream of a world-class chess-playing program, a 50-year quest of the computing science and artificial intelligence communities, was finally realized.

3.5 Othello

The first major Othello program was Paul Rosenbloom's *Iago* [82]. It achieved impressive results given its early-1980s hardware. It played only two games against world-class players, losing both. However, it dominated play against other Othello programs of the time. Based on the program's ability to predict 59% of the moves played by human experts, Rosenbloom concluded that the program's playing strength was of world-championship caliber.

By the end of the decade, *Iago* had been eclipsed. Kai-Fu Lee and Sanjoy Mahajan's program *Bill* represented a major improvement in the quality of computer Othello play [83]. The program combined deep search with extensive knowledge (in the form of precomputed tables) in its evaluation function. Bayesian learning was used to combine the evaluation function features in a weighted quadratic polynomial.

Statistical analysis of the program's play indicated that it was a strong Othello player. *Bill* won a single game against Brian Rose, the highest rated American Othello player at the time. In test games against *Iago*, *Bill* won every game. These results led Lee and Mahajan to conclude that "*Bill* is one of the best, if not the best, Othello player in the world." As usual, there is danger in extrapolating conclusions based on limited evidence.

With the advent of the Internet Othello Server (IOS), computer Othello tournaments became routine. In the 1990s they were dominated by Michael Buro's *Logistello*. The program participated in 25 tournaments, finished first 18 times, second six times, and fourth once. The program combined deep search with an extensive evaluation function that was automatically tuned. This was combined with an extensive opening book and a perfect endgame player.

Although it was suspected that in the mid-1990s, computers surpassed humans in their playing abilities at Othello, this was not properly demonstrated until 1997, when *Logistello* played an exhibition match against world champion Takeshi Murakami. In preparation for the match, Buro writes that [84]:

> *Bill* played a series of games against different versions of *Logistello*. The results showed that *Bill*, when playing 5-minute games running on a PentiumPro/200 PC, is about as strong as a 3-ply *Logistello*, even though *Bill* searches 8 to 9 plies. Obviously, the additional search is compensated for by knowledge. However, the 3-ply *Logistello* can only be called mediocre by today's human standards.
>
> Two explanations for the overestimation of playing strength in the past come to mind: (1) during the last decade human players have improved their playing skills considerably, and (2) the playing strength of the early programs was largely overestimated by using ... non-reliable scientific methods.

Logistello won all six games against Murakami by a total disc count of 264 to 120 [84]. This confirmed what everyone had expected about the relative playing strengths of man and machine. The gap between the best human players and the best computer programs is believed to be large and effectively unsurmountable.

3.5.1 *Logistello*

Outwardly, *Logistello* looks like a typical alpha-beta-based searcher. The program has a highly-tuned search algorithm, sophisticated evaluation function, and a large opening book.[12] The architecture of the program is illustrated in Figure 12.

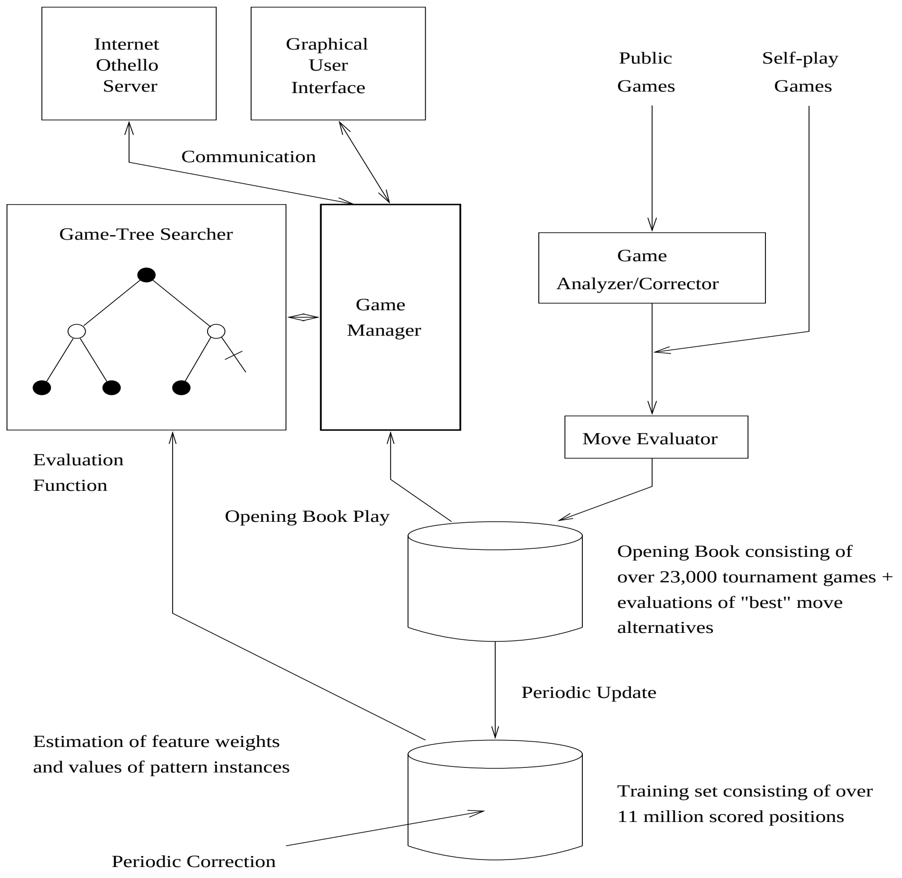


Figure 12: Logistello's architecture [85].

The search algorithm is standard alpha-beta (NegaScout) with iterative deepening, a large transposition table, and the killer heuristic. Corners are a critical region of the board. The program does a small quiescense search if there is some ambiguity about who controls a corner.

Buro introduced his ProbCut algorithm in *Logistello* [24]. This search enhancement takes advantage of the Othello property that the results of a shallow search (s ply) are correlated with the results of a deep search (t ply). An s-ply

[12]Note that endgame databases are not possible in Othello because, unlike chess and checkers, the number of pieces on the board increases as the end of game approaches.

search produces a value v_s. This value is extrapolated to give the value of a t-ply search v_t. The deeper search's estimated value is then compared to the alpha-beta search window, and the likelihood that v_t will be relevant to this window is computed. If v_t is likely to be irrelevant (e.g., is unlikely to reach alpha), then the search is pruned on the basis of the shallow search.

The deeper search value can be viewed as being:

$$v_t = a \times v_s + b + e$$

where a and b are constants and e is the error (normally distributed with a mean of 0 and variance σ^2). Given s and t, the parameters a, b and e are determined using linear regression on a large number of samples. For each sample position, searches to depth s and t are performed.

In a game, this information can be used to probabilistically eliminate trees using the relationship $v_t \leq \alpha$ with probability p if $v_s \leq (-\Phi^{-1}(p) \times \sigma + \alpha - b)/a$. The value of $\Phi^{-1}(p)$, the cut threshold, is set according to how much confidence one has in these cut-offs. Buro uses $\Phi^{-1}(1.5) = 0.93$. With this value, *Logistello* enhanced with ProbCut defeated the program without ProbCut by a score of 52–18 [24]. Since then, Buro has refined the algorithm to make it more applicable and powerful [86].

A special search case occurs as the end of game approaches. Here it is possible to search to the end and find the exact search result given perfect play by both players. Typically, *Logistello* can solve a position when there are 22–26 moves left in the game. The search is a highly tuned, blindingly fast routine with no knowledge to slow it down. When the program thinks it can solve the position, it will allocate a large portion of its remaining time to determine whether the position is a win, loss, or draw. Once the result is determined, searches on subsequent moves can be used (if necessary) to refine the score to maximize the result. Once the position is solved, all the remaining moves of the game can be played instantly.

Logistello uses an evaluation function that has been automatically tuned. The program treats the game as having 13 phases: 13–16 discs on the board, 17–20 discs, ..., and 61–64 discs.[13] Each phase has a different set of weights in the evaluation function. Thus, Equation 1 can be viewed as being:

$$value = \sum_{i=1}^{n_p} w_{p,i} \times f_{p,i} \qquad (2)$$

where p is the phase of the game.

The evaluation-function features are patterns of squares comprising combinations of corners, diagonals, and rows. These patterns capture important Othello concepts, such as mobility, stability and parity. *Logistello* has 11 such patterns, which with rotations and reflections yields 46. Some of the patterns are a 3×3 and a 5×2 configuration of stones anchored in a corner, and all diagonals of length greater than 3.

[13]Note that there is no need for a phase for less than 13 discs on the board, since the search from the first move easily reaches 13 or more discs.

The weights for each entry in each pattern (46) for each phase of the game (11) are determined by linear regression. There are over 1.5 million table entries that need to be determined. The data was trained using 11 million scored positions obtained from self-play games and practice games against another program [87]. The evaluation function is completely table-driven. Given a position, all 46 patterns are matched against the position, with a successful match returning the associated weight. These weights are summed to get the overall evaluation which approximates the final disc differential.

Opening books are a critical component to Othello programs. Ignoring passed moves (occasionally one side has no legal moves), each side has only 30 moves to play in a game. Given that programs can solve games with, say, 24 discs left on the board, that means that only the first 18 moves by each side are relevant. A strong opening book can guarantee that most of these moves are correct. Hence, considerable computational effort is spent on building an opening book.

The book is essentially a large tree of move sequences starting from the initial position of the game. It is built off-line using the results from tournament and self-play games [88]. The move sequence of a game can be analyzed backwards, allowing each position to get a more accurate score than had the moves been analyzed in a forward manner (the backwards searches benefit from the results obtained from the later searches in the game). The final result (win, loss, draw) is propagated as far back in the game as possible. These positions are added to the book as accurate values. Other positions are assigned their heuristic value (obtained from a deeper search than occurred in the game) and added to the book. At all positions added to the book, all the move alternatives not played are evaluated with a deep search. The scores are then minimaxed back up the opening book tree. For example, consider the scenario of *Logistello* playing a game and losing. Assume that the program identifies the loss on its 20th move. The book program will analyze this position and score all the alternative moves in this position. If the first 19 moves of a future game repeat the above move sequence, then the book will select for its 20th move the one that leads to the highest score (which is likely not the loss score). In this way, *Logistello* is guaranteed to deviate its play from the earlier game.

Deep searches, good evaluation, and a strong opening book are a winning recipe for Othello. Michael Buro comments on the reasons why *Logistello* easily won the Murakami match[84]:

> When looking at the games of the match the main reasons for the clear outcome are as follows:
>
> 1. Lookahead search is very hard for humans in Othello. The disadvantage becomes very clear in the endgame phase, where the board changes are more substantial than in the opening and middlegame stage. Computers are playing perfectly in the endgame while humans often lose discs.
>
> 2. Due to the automated tuning of the evaluation functions and deep selective searches, the best programs estimate their winning

chance in the opening and middlegame phase very accurately. This leaves little room for human innovations in the opening, especially because the best Othello programs are extending their opening books automatically to explore new variations.

3.5.2 The Best of Computer Othello

In August 1997, the World Champion Takeshi Murakami played a six-game exhibition match with *Logistello*. Having lost the first five games, Murakami fought hard for a win in the last game. Michael Buro, author of *Logistello*, annotates this game (comments are in italics) [89]. *Logistello*'s analysis is enclosed in []s giving the main lines of play and the final predicted result from *Logistello*'s point of view. Moves are given by specifying the column from left-to-right, "A" to "H", and the row from top-to-bottom, 1 to 8.

White: *Logistello* — Black: Takeshi Murakami

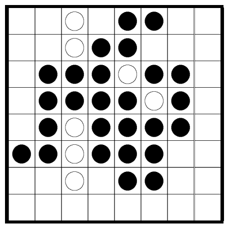
1. F5 D6 2. C4 D3 3. C3 F4 4. C5 B3 5. C2 E6 6. C6 B4 7. B5 D2 8. E3 A6 9. C1 B6 10. F3 F6 11. F7 E1 12. E2 F1 13. E7 G3 14. C7 G4 Logistello *prefers D7 over Mr. Murakami's G4. After D7 the position seems to be quite close. Mr. Murakami's opening and early midgame were flawless in* Logistello*'s view.*

15. G5 D1 See Figure 13a. *According to* Logistello*'s 26 ply selective search, Mr. Murakami's D1 is probably two discs worse than playing F2.*

16. G1 F2 17. H3 H4 *One move earlier, Mr. Murakami missed the last opportunity to deprive* Logistello *of its free move to B1. While H5 flips F5 and thereby denies* Logistello *access to B1, it also leads to a risky edge configuration after the moves H6 and H7. This may be the reason why Mr. Murakami prefered H4, which, however, loses two discs. Today's best programs would start selective win-loss-draw searches in this position after conducting a ≥24 ply "midgame" Multi-ProbCut search. This leaves human players with very little room for the slightest errors. [H5 H6 H7 = 4; H4 H5 D8 = 6]*

18. H5 C8 *Here, Mr. Murakami does not want to break* Logistello*'s wall which would create additional moves for* Logistello. *One plan is to move into the south-west region (C8 or D8) or to exploit regional parity in the north-east by playing G2. Although Mr. Murakami's G2 gives up corner H1 it leaves him with the last move in this region. Mr. Murakami chose C8, losing two discs. [G2 B1 D8 = 6; D8 C8 D7 = 6; C8 B1 D7 = 8]*

19. B1 D7 20. H2 E8 21. D8 G6 22. F8 G8 23. H6 G7 24. A2 A3 25. A4 B7 26. A8 B8 27. B2 See Figure 13b. *In this game Mr. Murakami never thought he was losing until the late endgame where he was faced with a swindle threat. At first glance, Mr. Murakami seems to have the advantage because the eastern and northern edge configurations look weak for* Logistello. *However, the only losing move in this position is G2 which allows Mr. Murakami to grab H1 and to secure enough edge and interior discs later on. The optimal move is B2 which Mr. Murakami had not anticipated in his earlier calculations.*


(a) before 15. ... D1

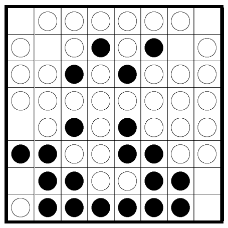

(b) before 27. B2

Figure 13: White: *Logistello* — Black: Takeshi Murakami

27. ... A1 *This move creates a so-called swindle in the south-east corner region, meaning that* Logistello *gets both remaining moves (H8 and H7) there—winning by 10. A little better is A5 which loses by 8. [A5 A7 A1 H8 = 8; A1 H8 H1 H7 = 10]*

28. H8 H1 29. H7 G2 30. A5 A7 Logistello *wins by 10 discs: 37–27.*

3.6 Poker

There are many popular poker variants. Texas Hold'em is generally acknowledged to be the most strategically complex variant of poker that is widely played. It is the premier event at the annual world series of poker.

Until recently, poker has been largely ignored by the computing academic community. However, poker has a number of attributes that make it an interesting domain for mainstream artificial-intelligence research. These include imperfect knowledge (the opponent's hands are hidden), multiple competing agents (more than two players), risk management (betting strategies and their consequences), agent modeling (identifying patterns and weaknesses in the opponent's strategy and exploiting them), deception (bluffing and varying your style of play), and dealing with unreliable information (taking into account your opponent's deceptive plays). All of these are challenging dimensions to a difficult problem.

There are two main approaches to poker research [90]. One approach is to use simplified variants that are easier to analyze. However, one must be careful that the simplification does not remove challenging components of the problem. For example, Findler worked on and off for 20 years on a poker-playing program for 5-card draw poker [91]. His approach was to model human cognitive processes and build a program that could learn, ignoring many of the interesting complexities of the game.

The other approach is to pick a real variant, and investigate it using mathematical analysis, simulation, and/or ad-hoc expert experience. Expert players with a penchant for mathematics are usually involved in this approach. None of this work has led to the development of strong poker-playing programs.

There is one event in the meager history of computer poker that stands out.

In 1984 Mike Caro, a professional poker player, wrote a program that he called *Orac* (Caro spelled backwards). It played one-on-one, no-limit Texas Hold'em. Few technical details are known about *Orac* other than it was programmed on an Apple II computer in Pascal. However, Caro arranged a few exhibitions of the program against strong players [92]:

> It lost the TV match to casino owner Bob Stupak, but arguably played the superior game. The machine froze on one game of the two-out-of-three set when it had moved all-in and been called with its three of a kind against Stupak's top two pair. Under the rules, the hand had to be replayed. In the [world series of poker] matches, it won one (from twice world champion Doyle Brunson — or at least it had a two-to-one chip lead after an hour and a quarter when the match was cancelled for a press conference) and lost two (one each to Brunson and then-reigning world champion Tom McEvoy), but — again — was fairly unlucky. In private, preparatory exhibition matches against top players, it won many more times than it lost. It had even beaten me most of the time.

Unfortunately, *Orac* was never properly documented and the results never reproduced. It is highly unlikely that *Orac* was as good as this small sample suggests. No scientific analysis was done to see whether the results were due to skill or luck (as was done, for example, in the *BKG9.8*–Villa match; see Section 3.1). As further evidence, none of the commercial efforts can claim to be anything but intermediate-level players.

In the 1990s, the creation of an Internet Relay Chat poker server gave the opportunity for humans (and computers) to play interactive games over the Internet. A number of hobbyists developed programs to play on IRC. Foremost among them is *R00lbot*, developed by Greg Wohletz. The program's strength comes from using expert knowledge at the beginning of the game, and doing simulations for subsequent betting decisions.

The University of Alberta program *Loki*, authored by Darse Billings, Denis Papp, Lourdes Peña, Jonathan Schaeffer and Duane Szafron, is the first serious academic effort to build a strong poker-playing program. *Loki* plays on the IRC poker server and, like *R00lbot*, is a consistent big winner. Unfortunately, since these games are played with fictitious money, it is hard to extrapolate these results to casino poker.

At best, *Loki* and *R00lbot* are strong intermediate-level poker players. A considerable gap remains to be overcome before computers will be as good as the best human players.

3.6.1 *Loki*

Most readers will be familiar with one or more variants of poker. To avoid confusion, the following gives a brief summary of Texas Hold'em. A hand begins with the "pre-flop", where each player is dealt two cards face down (the "hole" cards), followed by the first round of betting. Three community cards are then dealt face up on the table, called the "flop", and a second round of betting occurs. On the "turn", a fourth community card is dealt face up and another round of betting ensues. Finally, on the "river", a fifth community card is dealt face up and there is a final round of betting. All players still in the game reveal their two hole cards for the showdown. The best five-card poker hand formed from the two hole cards and the five community cards wins the pot. If a tie occurs, the pot is split. Texas Hold'em is typically played with 8 to 10 players.

Loki is named after the Norse God of mischief [93].[14] Figure 14 shows the program's architecture and how the major components interact [94]. In the diagram, rectangles are major components, rounded rectangles are major data structures, and ovals are actions. The data follows the arrows between components. An annotated arrow indicates how many times data moves between the components for each of the program's betting actions.

The architecture revolves around generating and using probability triples [50]. A probability triple is an ordered set of values, PT = [f,c,r], such that f + c + r = 1.0, representing the probability distribution that the next betting action in a given context should be a fold, call, or raise, respectively. The Triple Generator contains the poker knowledge, and is analogous to an evaluation function in two-player games. The Triple Generator calls the Hand Evaluator to evaluate any two-card hand in the current context. It uses the resulting hand value, the current game state, and expert-defined betting rules to compute the triple. To evaluate a hand, the Hand Evaluator enumerates over all possible opponent hands and counts how many of them would win, lose, or tie the given hand.

Each time it is *Loki*'s turn to bet, the Action Selector uses a single probability triple to decide what action to take. For example, if the triple [0.0,0.8,0.2] were generated, then the Action Selector would never fold, call 80% of the time and raise 20% of the time. A random number is generated to select one of these actions so that the program varies its play, even in identical situations.

After the flop, the probability for each possible opponent hand is different. For example, the probability that Ace-Ace hole cards are held is much higher than the cards 7-2, since most players will fold 7-2 before the flop. There is a Weight Table for each opponent. Each Weight Table contains a value for each possible two-card hand that the opponent could hold (47 choose 2 = 1,081 possibilities). The value is the probability that the hand would be played exactly as that opponent has played so far. After an opponent action, the Opponent Modeler updates the Weight Table for that opponent in a process called re-weighting. The value for each hand is increased or decreased to be consistent with the opponent's action. The Hand Evaluator uses the Weight Table in assessing the strength of each possible hand, and these values are in turn used to update the Weight Table after each opponent action.

For example, suppose the weight for Ace–Ace is 0.7. That is, if these cards have been dealt to an opponent, there is a 70% chance that they would have played it in exactly the manner observed so far. What happens if the opponent

[14]This section is largely based on previously-published descriptions of *Loki* [50].

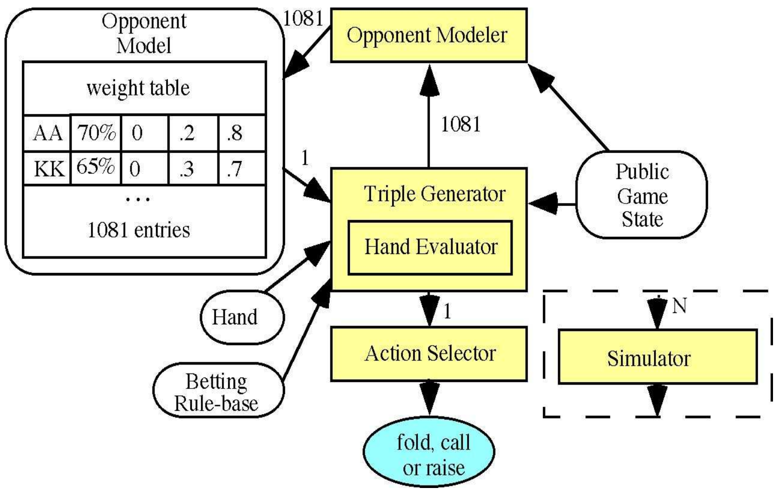


Figure 14: *Loki*'s architecture.

now calls? *Loki* calculates the probability triple for these cards in the current context (as it does for all possible two-card holdings). Assume that the resulting triple is [0.0, 0.2, 0.8]. The updated weight for this case would be $0.7 \times 0.2 = 0.14$. The relative likelihood of the opponent holding Ace-Ace has decreased to 14% because they did not raise. The call value of 0.2 reflects the possibility that this particular opponent might deliberately try to mislead us by calling instead of raising. Using a probability distribution allows us to account for uncertainty in our beliefs.

The Triple Generator provides good betting decisions. However, better results can be achieved by augmenting the evaluation with simulation. *Loki* can play out many likely scenarios to determine how much money each decision will win or lose. Every time it faces a decision, *Loki* invokes the Simulator to get an estimate of the expected value (EV) of each betting action (see the dashed box in Figure 14 with the Simulator replacing the Action Selector). A simulation consists of playing out the hand a specified number of times, from the current state of the game through to the end. Folding is considered to have a zero EV, because there is no future profit or loss. Each trial is played out twice — once to consider the consequences of a check/call and once to consider a bet/raise as *Loki*'s first action. In each trial, cards are dealt to each opponent (based on the probabilities maintained in the Weight Table), the resulting game is simulated to the end, and the amount of money won or lost is determined. Probability triples are used to approximate the actions of the opponents and *Loki*'s subsequent actions based on their two cards assigned for that trial. The average amount won or lost over all of the trials is taken as the EV of each action. In the current implementation, the action with the greatest expectation is selected, folding if both expectations are negative. To increase the program's unpredictability, the selection of betting actions whose EVs are close in value can be randomized.

It should be obvious that the simulation approach must be better than the simple evaluation approach, since simulation essentially uses a selective search to augment and refine a static evaluation function. Barring a serious misconception (or bad luck on a limited sample size), playing out relevant scenarios will improve the default values obtained by heuristics, resulting in a more accurate estimate. For example, a simulation contains implicit knowledge such as:

1. hand strength (fraction of trials where *Loki*'s hand is better than the one assigned to the opponent),
2. hand potential (fraction of trials where *Loki*'s hand improves to the best, or is overtaken), and
3. subtle implications not addressed in the simplistic betting strategy (e.g. "implied odds", extra bets won after a successful draw).

It also allows complex strategies to be uncovered without providing additional expert knowledge. For example, simulations can result in the emergence of advanced betting tactics like a check-raise, even if the basic strategy without simulation is incapable of this play.

In strategic games like chess, the performance loss by ignoring opponent modeling is small, and hence it is usually ignored. In contrast, not only does opponent modeling have tremendous value in poker, it can be the distinguishing feature between players at different skill levels. If a set of players all have a comparable knowledge of poker fundamentals, the ability to alter decisions based on an accurate model of the opponent may have a greater impact on success than any other strategic principle.

The Weight Table is the first step toward opponent modeling since the weights for opponent cards are changed based on the dynamics of the games. The simplest approach to determining these weights is to treat all opponents the same, calculating a single set of weights to reflect reasonable behavior, and use them for all opponents. An initial set of weights was determined by ranking the starting hands (as determined by off-line simulations) and assigning a probability commensurate with the average return on investment of each hand. These weights closely approximate the ranking of hands by strong players. In *Loki*, the Opponent Modeler uses probability triples to update the Weight Table after each opponent action (re-weighting). To accomplish this, the Triple Generator is called for each possible two-card hand. It then multiplies each weight in the Weight Table by the entry in the probability triple that corresponds to the opponent's action.

The above scheme is called Generic Opponent Modeling (GOM) [95]. Each hand is viewed in isolation and all opponents are treated as the same player. Each player's Weight Table is initially identical, and gets modified based on their betting action. Although rather simplistic, this model is quite powerful in that it does a good job of skewing the hand evaluations to take into account the most likely opponent holdings.

Obviously, treating all opponents the same is clearly wrong. Each player has a different style. Specific Opponent Modeling (SOM) customizes the probability triple function to represent the playing style of each opponent. For a given game, the re-weighting factor applied to the entries of the Weight table is adjusted by betting frequency statistics gathered on that opponent from previous hands. This results in a shift of the assumed call and raise thresholds for each player. During each round of a game, the history of previous actions by the opponent is used to influence the probability triple generated for that opponent.

In competitive poker, opponent modeling is much more complex than portrayed here. For example, players can act to mislead their opponents into constructing an erroneous model. Early in a session a strong poker player may try to create the impression of being very conservative, only to exploit that image later in that session when the opponents are using an incorrect opponent model. A strong player has to have a model of each opponent that can quickly adapt to changing playing styles.

An important part of strong poker is bluffing. Although mastering this is difficult for humans, it is not an obstacle for a poker program. The computer can extend the range of hands it will play to include a few that have small negative expectations.

3.6.2 The Best of Computer Poker

The hand shown in Figure 15 was played on IRC against six opponents. The following abbreviations are used to show the betting in each round: "sb": small blind, "bb": big blind", "c": call, "k": check, "b": bet, "f": fold, and "r": raise. Instead of an ante, Texas Hold'em uses blinds to initially seed the pot. The sample game is \$10/\$20 Hold'em. Here the first player puts in \$5 (the small blind), while the second player puts in \$10 (the big blind). The first two betting rounds use \$10 bets; the last two use \$20 bets. There are a maximum of three raises per betting round.

The following annotations in italics are by Darse Billings, co-author of *Loki* and a former professional poker player.

Loki *makes a "loose call" with a fairly weak hand before the flop, because the conditions are otherwise ideal (last position, no raises, and a suited hand with 5 or 6 opponents). In slightly less favorable circumstances,* Loki *would fold this hand before the flop.*

The flop yields a good flush draw with an overcard to the board. After the bet and two calls, a raise is a viable option, since it would have positive expectation against three opponents (> 25% of winning), and might also earn a "free card" (no bet on the turn). Loki *opts for the quieter alternative, which gains an additional caller in the small blind (which is favorable in this situation). A higher spinner value for the mixed strategy would have resulted in a raise.*

The turn card adds a straight possibility to the draw, and after everyone shows weakness by checking, Loki *decides to "semi-bluff". Unfortunately, the big blind was playing possum and check-raises with the best possible hand (a straight). In hindsight, this was a very risky play on his part — if* Loki *had*

```
Events                 opp1 opp2 opp3 opp4 opp5 opp6 Loki

Hole cards 4♢ Q♢
Preflop betting         sb   bb    c    c    c    c    c
                         c    k
Flop cards 5♢ J♠ 7♢
Flop betting             k    b    f    c    f    c    c
                         c
Turn card 3♣
Turn betting             k    k         k         k    b
                         c    r         f         c    c
                         c
River card J♢
River betting            k    b                   f    r
                         f    r                        c

Showdown               opp2 shows 6♠ 4♠
                       Loki shows Q♢ 4♢

                       Loki wins $400
```

Figure 15: *Loki* in Action

checked, he would have failed to earn anything from the other players with his very strong hand, and would have given away a free chance to make a better hand. After Loki*'s bet, he is happily able to build a large pot.*

Loki *is lucky enough to make the flush, and raises on the river. After the re-raise, the opponent's betting pattern suggests a full house (at least as likely as a straight) and* Loki *calls.*

Loki's flush wins against the opponent's straight. *Loki* wins $400. Were this only real money...

3.7 Scrabble

The first documented Scrabble program appears to have been written by Stuart Shapiro and Howard Smith and was published in 1977 [96]. In the 1980s a number of Scrabble programming efforts emerged and by the end of the decade, it was apparent that these programs were strong players. With access to the entire Scrabble dictionary (now over 100,000 words), the programs held an important advantage in any games against humans.

At the First Computer Olympiad in 1989 the Scrabble winner was *Crab* written by Andrew Appel, Guy Jacobson, and Graeme Thomas [97]. Second was *Tyler* written by Alan Frank. Subsequent Olympiads saw the emergence of *TSP* (Jim Homan), which edged out *Tyler* in the second and third Olympiads. *TSP* later became the commercial program *Crosswise*. All of these programs were very good, and quite possibly strong enough to be a serious test for the best players in the world.

Part of their success was due to the fast, compact Scrabble move generator developed by Andrew Appel [98]. Steven Gordon subsequently developed a move generator that was twice as fast, but used five times as much storage [99].

Brian Sheppard began working on a Scrabble program in 1983, and started developing *Maven* in 1986. In a tournament in December 1986, *Maven* scored eight wins and two losses over an elite field, finishing in second place on tie-break. Sheppard describes the games against humans at this tournament [51]:

> *Maven* reels off JOUNCES, JAUNTIER, and OVERTOIL on successive plays, each for exactly 86 points, to come from behind against future national champion Bob Felt. Maven crushed humans repeatedly in offhand games. The human race begins to contemplate the potential of computers.

In the following years, *Maven* continued to demonstrate its dominating play against human opposition. Unfortunately, since it did not compete in the Computer Olympiads, it was difficult to know how strong it was compared to other programs at the time.

In the 1990s, Sheppard developed a pre-endgame analyzer (for when there were a few tiles left in the bag) and improved the program's ability to simulate likely sequences of moves. These represented important advances in the program's ability. It was not until 1997, however, that the opportunity arose to properly assess the program's abilities against world-class players. In 1997, a two-game match between *Maven* and Adam Logan, one of the best players in North America, ended in two wins for the human. Unfortunately, the match was not long enough to get a sense of who was really the best player.

In March 1998, the New York Times sponsored an exhibition match between *Maven* and a team consisting of world champion Joel Sherman and the runner-up Matt Graham. It is not clear whether the collaboration helped or hindered the human side, but the computer won convincingly by a score of six wins to three. The result was not an anomaly. In July of that year, *Maven* played another exhibition match against Adam Logan, scoring nine wins to five.

Shortly after the Logan match, Brian Sheppard wrote:

> The evidence right now is that *Maven* is far stronger than human players. ... I have outright claimed in communication with the cream of humanity that *Maven* should be moved from the "championship caliber" class to the "abandon hope" class, and challenged anyone who disagrees with me to come out and play. No takers so far, but maybe one brave human will yet venture forth.

No one has.

3.7.1 *Maven*

The following description of *Maven* is based on information provided by *Maven*'s author, Brian Sheppard [100].

Maven divides the game into three phases: early game, pre-endgame, and endgame. The early game starts at move one and continues until there are nine or fewer tiles left in the bag (*i.e.*, with the opponent's seven tiles, this implies that there are 16 or fewer unknown tiles). From there, the pre-endgame continues until there are no tiles in the bag. In the endgame, all the tiles in the opponent's rack are known.

Maven uses the following techniques in regular play, before the pre-endgame is reached. The program uses the simulation framework described in Section 2.3, with some important Scrabble-specific refinements. Whereas for other games, such as bridge and poker, the number of candidate moves is small, for Scrabble there can be many moves to consider. On average there are over 700 legal moves per position, and the presence of two blanks in the rack can increase this figure to over 5,000! Thus, *Maven* needs to pare the list of possible moves (using the move generator algorithm described in [98]) down to a small list of likely moves. Omitting an important move from this list will have serious consequences; it will never be played. Consequently, *Maven* employs multiple move generators, each identifying moves that have important features that merit consideration. These move generators are:

- Score and Rack. This generator finds moves that result in a high score and a good rack (tiles remaining in your possession). Strong players evaluate their rack based on the likeliness of the letters being used to aid upcoming words. For example, playing a word that leaves a rack of QXI would be less preferable than leaving QUI; the latter offers more potential for playing the Q effectively.

- Bingo Blocking. Playing all seven letters in a single turn leads to a bonus of 50 points (often called a *bingo*). This move generator finds moves that reduce the chances of the opponent scoring a bingo on their next turn. Sometimes it is worth sacrificing points to reduce the opponent's chances of scoring big.

- Immediate Scoring. This generates the moves with the maximum number of points (this becomes more important as the end of the game nears).

Each routine provides up to 10 candidate moves. Merging these lists results in typically 20-30 unique candidate moves to consider. In the early game only the Score and Rack generator is used. In the pre-endgame there are four: the three listed above plus a pre-endgame evaluator that "took years to tune to the point where it didn't blunder nearly always" [101]. In the endgame, all possible moves are considered.

The move generation routines are highly effective at filtering the hundreds or thousands of possible moves [101]:

> It is important to note that simply selecting the one move preferred by the Score and Rack evaluator plays championship caliber Scrabble. My practice of combining 10 moves from multiple generators is

> evidence of developing paranoia on my part. "Massive overkill" is the centerpiece of *Maven*'s design philosophy.

Sheppard points out that his program is missing a *fishing* move generator. Sometimes it is better to pass a move or play a small word (one or two letters), so that you can exchange some of your tiles. For example, with the opening rack of AEINQST, you can play QAT for 24 points. Instead, you can *fish* by not playing a word and exchanging the Q. Of the 93 remaining tiles, 90 will make a bingo.

For the simulations, *Maven* does a two-ply search to evaluate each candidate move (in effect, this is a three-ply search). It could use a four-ply search for the evaluation, but this results in fewer simulation data points. Sheppard discusses the trade-offs:

> If you compare a four-ply horizon and a two-ply horizon, you find that each iteration of the four-ply horizon takes twice as long, and the variance is twice as large, so you need $2 \times \sqrt{2}$ times as much time to simulate to equal levels of statistical accuracy. Since Scrabble has only limited long-term issues, it makes sense to do shallow lookaheads.

The limited long-term issues mentioned are a consequence of the rapid turnover in the rack. *Maven* averages playing 4.5 tiles per turn. After a two-ply lookahead, there are few (if any) tiles left from the original rack. Consequently, positions being evaluated at the leaves of a two-ply search are very different than the root node.

Typically, 1,000 two-ply simulations are done when making a move decision. The move leading to the highest average point differential is selected. After a few simulations, it may become statistically obvious that some of the candidate moves have little or no chance of being selected because their expected values are too low. If a move's score is at least two standard deviations below that of the best move, and at least 17 simulation iterations have been performed then the low-scoring move is eliminated from consideration. The assignment of tiles to opponent hands is done in a way that guarantees a uniform distribution. A minimum of 14 iterations are needed to place all tiles in an opponent's rack at least once. The 17 iterations comes from 14 being rounded up to a power of two (16) and then an inadvertent off-by-one error giving 17.

Other pruning schemes are used to refine the move list. First, nearly identical plays usually lead to almost identical scores. For example, an opening move of "PLAY" versus "PALY" makes no difference in the simulation results. After 101 simulations, the lower rated of almost-identical moves is pruned. Secondly, if it becomes impossible for a low-scoring move to catch up to the best-scoring move given the number of trials remaining, then that move is pruned without any risk.

In the pre-endgame, the program's emphasis changes from scoring points to scoring wins. With fewer moves to consider, the simulations are extended to reach the end of the game to determine which side wins. The simulations contain additional pruning. If a candidate move is generating significantly fewer points than the best move and its frequency of wins is less, then that move is eliminated.

Using the simulations to count the frequency of wins and points can cause a dilemma. It may be ambiguous as to what the best move to play is [101]:

> Sometimes one move is the winner both on points and wins, so the choice is clear. But sometimes it is not clear, because wins and points do not agree. In that case *Maven* "mixes" wins and points on a linear basis. There are two important practical reason for this. First, the simulation might not be representative of the actual play of the game, either because the opponent is incapable of playing as well as *Maven* (the good case), or because *Maven*'s simulations are mishandling the situation (the bad case). In either of these cases extra points may come in handy. Second, in tournaments it is important to have a high point differential, since that is used to break ties. My calculation shows that a 1% higher chance of winning a game is worth roughly a three to four point sacrifice of point spread. We don't want to go overboard on defensive gestures at the end of a game. It is better to lose occasionally to keep a high differential.

A special case occurs when there are only eight unknown tiles. In this case, the opponent can have only one of eight possible tile holdings, so *Maven* searches each case to the end of the game to determine the final result. Sheppard has recently extended his program to handle up to 12 unknown tiles (924 combinations).

When there are no tiles left to be drawn, Scrabble reverts to a game of perfect information (all missing tiles are in the opponent's rack). Alpha-beta would take too long to exhaustively search this, since the branching factor is large, and the program (move generation) is slow. Instead, *Maven* uses the B* algorithm (see Section 2.1.6). The success of B* hinges on assigning good upper and lower bounds to the moves. Considerable heuristic code is devoted to determining these bounds. Although *Maven* is capable of making an error in the search (e.g. poor bounds, or limits on space), in practice this is rarely seen. This may be the only example of a real system where B* is to be preferred to alpha-beta.

The Scrabble community has extensively analysed *Maven*'s play and found a few minor errors in the program's play. Postmortem analysis of the Logan match showed that *Maven* made mistakes that averaged nine points per game. Logan's average was 40 points per game. *Maven* missed seven fishing moves (69 points lost), some programming errors (48 points lost), and several smaller mistakes (6 points lost). The programming errors have have been corrected. If a future version of *Maven* included fishing, the points per game error rate would drop to less than one per game. *Maven* would be playing nearly perfect Scrabble.

Of the points lost due to programming errors, Brian Sheppard writes:

> It just drives me crazy that I can think up inventive ways to get computers to act intelligently, but I am not smart enough to implement them correctly.

And that is the soliloquy of every games programmer.

3.7.2 The Best of Computer Scrabble

In July 1998, at the annual conference of the American Association for Artificial Intelligence, *Maven* played an exhibition match against Adam Logan, one of the top Scrabble players in North America. Logan won three of the first four games of the match, but *Maven* won six of the next seven games. Going into the critical 12th game, *Maven* led by a score of seven wins to four. The following annotations are by Brian Sheppard and originally appeared in the *Scrabble Players News*.[15] The columns of a Scrabble board are specified from left-to-right by the letters *a* to *o*. Rows are specified from top-to-bottom using the numbers 1 to 15. Moves are specified by giving the square of the first letter of the word. If the coordinate begins with a number, then the word is placed horizontally. If the coordinate begins with a letter, then the word is placed vertically. The blank is referred to by "?".

Maven — Adam Logan

Maven(ACNTVYZ) plays CAVY at 8f, 24 pts, *Maven*=24 Logan=0. *The alternative is ZANY for 32, but the CVT rack is poor. Much better is 24 points with an NTZ rack. As to placement, a better choice than* Maven*'s is probably CAVY 8G. This version of* Maven *was not ideal at first-turn placement, for inexcusable internal reasons. Fortunately this is not a significant skill factor compared to scoring and keeping good tiles.* Maven *is almost ideal at those skill factors.*

Logan(EGLNORY) plays YEARLONG at g6, 66 pts, *Maven*=24 Logan=66. *Adam finds the only bingo.*

Maven(ADNNOTZ) plays DOZY at 6d, 37 pts, *Maven*=61 Logan=66. *It is DOZY(6d,37,ANNT) versus AZLON(10e,34,NTD) or ZOON(11e,26,ADNT). DOZY's extra points and retention of a vowel win despite duplicate Ns.*

Logan(ADEFOTV) plays OFT at h13, 21 pts, *Maven*=61 Logan=87. *Adam's choice is best. He also has VOTED(5A,27,AF), OVA(H13,21,DEFT), FOVEAL(10b,22,DT), and ADVENT(12c,22,FO). Adam didn't think long, and since the choices are so close it doesn't pay to think long!*

Maven(AENNNOT) plays NEON at 5b, 15 pts, *Maven*=76 Logan=87. *NEON(5b,15,ANT) edges ANON(5b,15,ENT). I am not sure why, but clearly ENT and ANT are both good rack leaves, and there must be some benefit to avoiding a FANON hook in the "a" column. It may also be that ANON's vowel-consonant-vowel-consonant pattern is easier to overlap than NEON.*

[15] Reproduced with permission. Minor editing changes have been made to conform with the style of this chapter.

Logan(ACDEEIV) plays DEVIANCE at 12b, 96 pts, *Maven*=76 Logan=183. *Adam finds the only bingo.*

Maven(AHINRTU) plays HURT at 4a, 34 pts, *Maven*=110 Logan=183. *HUNT would usually surpass HURT, because R is better than N, but here there are three N's already on the board versus one R. It is important to note that* Maven *did not choose HUNT for the reason I gave;* Maven *chose HUNT because in 1,000 iterations of simulation it found that HUNT scored more points than HURT. The reason I am giving (that three N's have been played versus one R) is my interpretation of that same body of data.*

Logan(DDEEMMN) plays EMENDED at c7, 26 pts, *Maven*=110 Logan=209. *EMENDED is a good play, following sound principles: score points, undouble letters. Simulations give a two-point edge to MEM(13a,25,EDDN), however. Possibly the 8a-8d spot weighs against EMENDED, plus keeping an E is a valuable benefit for MEM. These advantages outweigh the extra point and duplicated "D"s.*

Maven(ABEINNP) plays IAMB at 8a, 33 pts, *Maven*=143 Logan=209. *IAMB is really the only play, doubled N's notwithstanding.*

Logan(AILMTTU) plays MATH at a1, 27 pts, *Maven*=143 Logan=236. *MATH(a1,27,ILTU) is best, with UTA(3a,20,ILMT) second. The advantage of MATH over UTA is its seven extra points, but the disadvantage is keeping a U. These almost wash, with an edge to MATH.*

Maven(EFGNNPS) plays FEIGN at e10, 18 pts, *Maven*=161 Logan=236. *FEIGN is the only good move. FENS(j9,24,GNP) is higher scoring, but FEIGN keeps better tiles; NPS easily makes up the scoring deficit plus a lot more on top.*

Logan(AILORTU) plays TUTORIAL at 15h, 77 pts, *Maven*=161 Logan=313. *Adam finds the only bingo. (Actually, TUTORIAL also plays at 15f, but scores only 59 there.)*

Maven(?ABNOPS) plays BOS at j10, 26 pts, *Maven*=187 Logan=313. See Figure 16. Maven *made a great draw from the bag, and then made one of the most difficult plays of the game.* Maven *has no bingos, and has to choose how to make one. Playing off the B and P is indicated, so plays like BAP or BOP (7i,20) come to mind. But* Maven *finds two stronger, and surprising, alternatives: BOS(j10,26,?ANP) and BOPS(j9,25,?AN). These plays score a few extra points as compensation for playing the S, and they open the "k" column for bingo-making. I would have thought that BOPS would win out, but BOS is better. BOS does show a higher point differential, but that is not why it is better. It is better because the chance of getting a big bingo is higher due to the creation of a spot where a bingo can hit two double word squares. I believe that the great majority of human masters would have rejected BOS without a second thought, probably choosing BOP. BOS is a fantastic play, and yet, there are two plays still to come in this game that are more difficult still.*

Logan(IILPRSU) plays PILIS at 15a, 34 pts, *Maven*=187 Logan=347. *PILIS, PULIS, PILUS, and PURIS are all good. Adam's choice is best because there are only two U's left, and Adam doesn't want to risk getting a bad Q. When you lead the game you have to guard against extreme outcomes.*

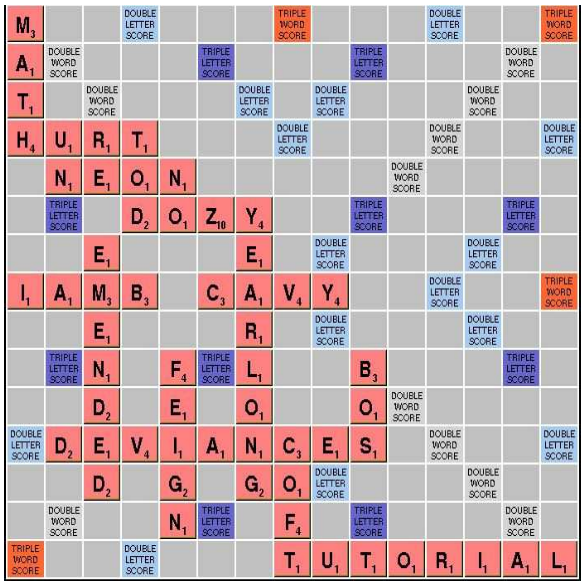


Figure 16: *Maven* plays BOS (j10) scoring 26 points.

Maven(?AKNPRS) plays SPANKER at k5, 105 pts, *Maven*=292 Logan=347. *This is the only bingo, and a big boost to* Maven*'s chances. I saw SPANKER but I wasn't sure it was legal, so I was sitting on the edge of my seat. Being down 160 points is depressing. Worse than depressing: it is nearly impossible to come back from that far behind. The national championship tournament gives a prize to the greatest comeback, and in this 31-round, 400-player event there is often only one game that features such a comeback.*

Logan(EEEORUS) plays OE at b1, 12 pts, *Maven*=292 Logan=359. *Adam plays the best move again. This play scores well, as his highest-scoring play is just 13 points (ERE L6). OE dumps vowels while keeping all his consonants (an edge over ERE). It also keeps the U as "Q-insurance," an edge over MOUE(1a,7,EERS). And it blocks a bingo line. Not bad value, and a good example of how to make something positive happen on every rack.*

Maven(?HJTTWW) plays JAW at 7j, 13 pts, *Maven*=305 Logan=359. Maven*'s draw is bad overall, but at least there is hope if* Maven *can clear drek. Any play that dumps two of the big tiles is worth considering, with JAW, WORTH(11i,16,?JW), and WAW(b7,19,?JHTT) as leading contenders. JAW wins because the WH and TH are bearable combinations, and the TT isn't too bad either. Many players would exchange this rack, but* Maven *didn't consider doing so. I don't know how exchanging (keeping ?HT, presumably) would fare, but I suspect it wouldn't do well; there are few good tiles remaining, and drawing a Q is a real risk.*

Logan(AEEGRSU) plays GREASE at m3, 31 pts, *Maven*=305 Logan=390. *Simulations show AGER(L9,24,ESU) as three points superior to GREASE, but*

I suspect that GREASE does at least as good a job of winning the game, since it takes away _ _ _ S bingos off of JAW. It also pays to score extra points, which provide a cushion if Maven *bingos. And it pays to turn over tiles, which gives* Maven *fewer turns to come back.*

Maven(?HRTTWX) plays AX at 6m, 25 pts, *Maven*=330 Logan=390. Maven*'s move is brilliant. Who would pick AX over GOX(13G,36)? Would you sacrifice 11 points, while at the same time creating a huge hook on the "o" column for an AX _ _ _ E play? And do so when there are two E's unseen among only 13 tiles and you don't have an E and you are only turning over one tile to draw one? It seems crazy, but here's the point: among the unseen tiles (AAEEIIIILOQUU) are only two consonants, and one of them is the Q, which severely restricts the moves that can be made on the "o" column. If Adam has EQUAL then* Maven *is dead, of course, but otherwise it is hard to get a decent score on the "o" column. In effect,* Maven *is getting a free shot at a big "o" column play. AX is at least 10 points better than any other move, and gives* Maven *about a 20% chance of winning the game. The best alternative is HAW(b7,19). GOX is well back.*

Logan(EIIILQU) plays LEI at o5, 13 pts, *Maven*=330 Logan=403. *Adam sensibly blocks, and this is the best play. The unseen tiles from Adam's perspective are ?AAEHIORTTUW, so Adam's vowelitis stands a good chance of being cured by the draw.*

Maven(?AHRTTW) plays WE at 9b, 10 pts, *Maven*=340 Logan=390. *Again a problem move, and again* Maven *finds the best play. In fact, it is the only play that offers real winning chances.* Maven *calculates that it will win if it draws a U, with the unseen tiles AEIIIOQUU. There may also be occasional wins when Adam is stuck with the Q. This move requires fantastic depth of calculation. What will* Maven *do if it draws a U?*

Logan(AIIIOQU) plays QUAI at j2, 35 pts, *Maven*=340 Logan=438. *Adam's natural play wins unless there is an E in the bag. AQUA(N12,26), QUAIL(O11,15), QUAI(M12,26), and QUA(N13,24) also win unless there is an E in the bag, but with much, much lower point differential than QUAI because these plays do not block bingos through the G in GREASE. There is no better play. If an E is in the bag then Adam is lost.*

Maven(?AHRTTU) plays MOUTHPART at 1a, 92+8 pts, *Maven*=440 Logan=438. See Figure 17. *Maven* scores 92 points for MOUTHPART, and eight points for the tiles remaining in Logan's rack. Maven *was fishing for this bingo when it played WE last turn. With this play* Maven *steals the game on the last move. Adam, of course, was stunned, as it seemed that there were no places for bingos left on this board. If I hadn't felt so bad for Adam, who played magnificently, I would have jumped and cheered. This game put* Maven *up by eight games to four, so winning the match was no longer in doubt.*

How often do you score 438 points in a game of Scrabble... and lose?

3.8 Other Games

Conspicuously absent from this chapter is the Oriental game of Go. It has been resistant to the techniques that have been successfully applied to the games

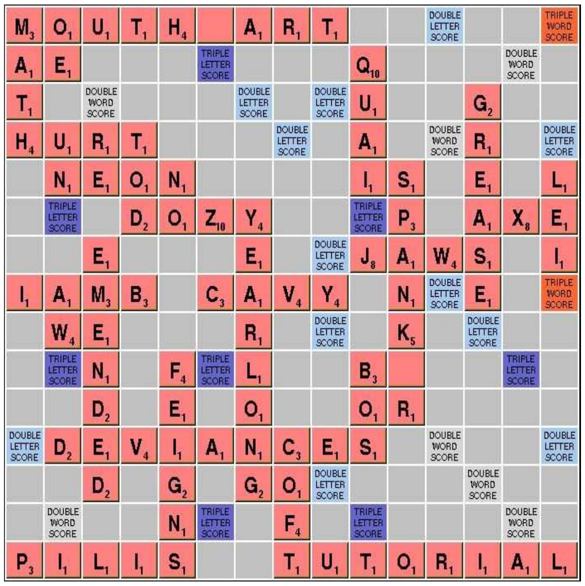


Figure 17: *Maven* — Logan, final position

discussed in this chapter. For example, because of the 19 × 19 board and the resulting large branching factor, alpha-beta search alone has no hope of producing strong play. Instead, the programs perform small, local searches that use extensive application-dependent knowledge. David Fotland, the author of the *Many Faces of Go* program, identifies over 50 major components needed by a strong Go-playing program. The components are substantially different from each other, few are easy to implement, and all are critical to achieving strong play. In effect, you have a linked chain, where the weakest link determines the overall strength.

Martin Müller (author of *Explorer*) gives a stark assessment of the reality of the current situation in developing Go programs [102]:

> Given the complexity of the task, the supporting infrastructure for writing Go programs should offer more than is offered for other games such as chess. However, the available material (publications and source code) is far inferior. The playing level of publicly available source code ..., though improved recently, lags behind that of the state-of-the-art programs. Quality publications are scarce and hard to track down. Few of the top programmers have an interest in publishing their methods. Whereas articles on computer chess or general game-tree search methods regularly appear in mainstream AI journals, technical publications on computer Go remain confined to hard to find proceedings of specialized conferences. The most interesting developments can be learned only by direct communication

with the programmers and never get published.

Although progress has been steady, it will take many decades of research and development before world-championship-caliber Go programs exist.

At the other end of the spectrum to Go are solved games. For some games, computers have been able to determine the result of perfect play and a sequence of moves to achieve this play.[16] In these games the computer can play perfectly, in the sense that the program will never make a move that fails to achieve the best-possible result. Solved games include Nine Men's Morris [43], Connect-4 [103], Qubic [104], and Go Moku [104].

This chapter has not addressed one-player games (or puzzles). Single-agent search has been successfully used to optimally solve the 15-puzzle [14] and Rubik's Cube [105], and progress is being made on solving Sokoban problems [106]. Recently, major advances have occurred in building programs that can solve crossword puzzles [107].

The last few years have seen research on team games become popular. The annual RoboCup competition encourages hardware builders and software designers to test their skills on the soccer field (www.robocup.com).

Finally, other areas of games-related interest include commercial computer games, such as sports and role-playing games. The artificial-intelligence work on these games is still in its infancy.

4 Conclusions

Samuel was writing as a pioneer, one of the first to realize that computer games could be a rich domain for exploring the boundaries of computer science and artificial intelligence. Since his 1960 paper, software and hardware advances have led to significant success and milestones in the history of computing. With it has come a change in people's attitudes. Whereas in Samuel's time, understanding how to build strong game-playing program was at the forefront of artificial-intelligence research, today, 40 years later, it has been demoted to lesser status. In part this is an acknowledgment of the success achieved in this field — no other area of artificial intelligence research can claim such an impressive track record of producing high-quality working systems. But it is also a reflection on the nature of artificial intelligence itself. It seems that as the solution to problems become understood, the techniques become less "AIish".

The work on computer games has resulted in advances in numerous areas of computing. One could argue that the series of computer-chess tournaments that began in 1970 and continue to this day represents the longest running experiment in computing science. The games research has demonstrated the benefits of brute-force search, something that has become a widely accepted tool for a number of search-based applications. Many of the ideas that saw the

[16]This is in contrast to the game of Hex where it is easy to prove the game to be a first player win, but computers are not yet able to demonstrate that win.

light of day in game-tree search have been applied to other algorithms. Building world-championship-caliber games programs has demonstrated the cost of constructing high-performance artificial-intelligence systems. Games have been used as experimental test beds for many areas of artificial intelligence. And so on.

Samuel's concluding remarks from his 1960 chapter are as relevant today as they were when he wrote the paper [72]:

> Just as it was impossible to begin the discussion of game-playing machines without referring to the hoaxes of the past, it is equally unthinkable to close the discussion without a prognosis. Programming computers to play games is but one stage in the development of an understanding of the methods which must be employed for the machine simulation of intellectual behavior. As we progress in this understanding it seems reasonable to assume that these newer techniques will be applied to real-life situations with increasing frequency, and the effort devoted to games ... will decrease. Perhaps we have not yet reached this turning point, and we may still have much to learn from the study of games.

5 Acknowledgments

I would like to extend my deepest admiration to the brave human champions who accepted the challenge of a computer opponent. In most cases, the champion had little to gain, but everything to lose. Malcolm Davis, Garry Kasparov, Adam Logan, Zia Mahmood, Marion Tinsley, Michael Rosenberg, and Takeshi Murakami made it possible to scientifically measure the progress of game-playing programs.

Many people made significant contributions to this chapter. I would like to extend my sincere appreciation to: Backgammon: Gerry Tesauro (author of *TD-Gammon*) and Malcolm Davis (world backgammon champion); Bridge: Matt Ginsberg (author of *GIB*), Mike Whittaker (*Bridge Magazine*), and Onno Eskes (*IMP* magazine); Chess: Murray Campbell (co-author of *Deep Blue*); Othello: Michael Buro (author of *Logistello*); Poker: Darse Billings (co-author of *Loki*) and Mike Caro (*Card Player Magazine*); Scrabble: Brian Sheppard (author of *Maven*) and Joe Edley (*Scrabble Players News*).

I am appreciative of the feedback from Darse Billings, Michael Buro, Andreas Junghanns, Lourdes Peña, Jack van Ryswyck, and Roel van der Goot. Technical help was provided by Mark Brockington and Alice Nodelman.

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Finally, I would like to thank Marvin Zelkowitz for the opportunity to contribute to this volume. It was an honor to follow in Arthur Samuel's footsteps.

References

[1] C. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.

[2] A. Turing. Digital computers applied to games. In B. Bowden, editor, *Faster than Thought*, pages 286–295. Pitman, 1953.

[3] A. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.

[4] A. Samuel. Some studies in machine learning using the game of checkers: Recent progress. *IBM Journal of Research and Development*, 11:601–617, 1967.

[5] M. Krol. Have we witnessed a real-life Turing test. *Computer*, 32(3):27–30, 1999.

[6] K. Thompson. Retrograde analysis of certain endgames. *Journal of the International Computer Chess Association*, 9(3):131–139, 1986.

[7] D. Lafferty. Chinook draws the hundred years problem. *The American Checker Federation Bulletin*, (269):6, 1997.

[8] A. Brudno. Bounds and valuations for abridging the search for estimates. *Problems of Cybernetics*, 10:225–241, 1963. Translation of the original that appeared in *Problemy Kibernetiki*, 10:141–150, 1963.

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

[10] T. Marsland. A review of game-tree pruning. *Journal of the International Computer Chess Association*, 9(1):3–19, 1986.

[11] D. Slate and L. Atkin. Chess 4.5 – The Northwestern University chess program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer Verlag, 1977.

[12] A. Plaat. *Research Re: Search and Re-Search.* PhD thesis, Erasmus University, The Netherlands, 1996.

[13] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Exploiting graph properties of game trees. In *AAAI National Conference*, pages 234–239, 1996.

[14] R. Korf. Iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[15] J. Schaeffer. *Experiments in Search and Knowledge.* PhD thesis, University of Waterloo, Canada, 1986.

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

[17] S. Akl and M. Newborn. The principle continuation and the killer heuristic. In *ACM Annual Conference*, pages 466–473, 1977.

[18] A. Reinefeld. An improvement of the Scout tree-search algorithm. *Journal of the International Computer Chess Association*, 6(4):4–14, 1983.

[19] A. Reinefeld. *Spielbaum Suchverfahren.* Informatik-Fachberichte 200, Springer Verlag, 1989.

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

[21] J. Pearl. Scout: A simple game-searching algorithm with proven optimal properties. In *AAAI National Conference*, pages 143–145, 1980.

[22] D. Beal. A generalised quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.

[23] C. Donninger. Null move and deep search: Selective-search heuristics for obtuse chess programs. *Journal of the International Computer Chess Association*, 16(3):137–143, 1993.

[24] M. Buro. Probcut: An effective selective extension of the α-β algorithm. *Journal of the International Computer Chess Association*, 18(2):71–76, 1995.

[25] T. Anantharaman, M. Campbell, and F. Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, 1990.

[26] Thomas Anantharaman. *A Statistical Study of Selective Min-Max Search.* PhD thesis, Carnegie Mellon University, United States, 1991.

[27] M. Campbell, J. Hoane, and F. Hsu. Search control methods in Deep Blue. In *AAAI Spring Symposium on Search Techniques for Problem Solving Under Uncertainty and Incomplete Information*, pages 19–23. AAAI Press, 1999.

[28] C. Ebeling. *All the Right Moves.* MIT Press, 1987.

[29] R. Feldmann. *Spielbaumsuche mit massiv parallelen Systemen.* PhD thesis, Universität-Gesamthochschule Paderborn, Germany, 1993.

[30] H. Berliner. The B* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12:23–40, 1979.

[31] H. Berliner and C. McConnell. B* probability based search. *Artificial Intelligence*, 86(1):97–156, 1996.

[32] D. McAllester. Conspiracy numbers for min-max searching. *Artificial Intelligence*, 35:287–310, 1988.

[33] U. Lorenz, V. Rottmann, R. Feldmann, and P. Mysliwietz. Controlled conspiracy-number search. *Journal of the International Computer Chess Association*, 18(3):135–147, 1995.

[34] E. Baum and W. Smith. A Bayesian approach to relevance in game playing. *Artificial Intelligence*, 97(1–2):195–242, 1997.

[35] R. Rivest. Game tree searching by min/max approximation. *Artificial Intelligence*, 34(1):77–96, 1987.

[36] S. Russell and E. Wefald. *Do the Right Thing.* MIT Press, 1991.

[37] K. Thompson. Computer chess strength. In M. Clarke, editor, *Advances in Computer Chess 3*, pages 55–56. Pergamon Press, 1982.

[38] F. Hsu. IBM's Deep Blue chess grandmaster chips. *IEEE Micro*, (March-April):70–81, 1999.

[39] A. Junghanns and J. Schaeffer. Search versus knowledge in game-playing programs revisited. In *International Joint Conference on Artificial Intelligence*, pages 692–697, 1997.

[40] T. Scherzer, L. Scherzer, and D. Tjaden. Learning in Bebe. In T. Marsland and J. Schaeffer, editors, *Computers, Chess and Cognition*, pages 197–216. Springer Verlag, 1990.

[41] D. Slate. A chess program that uses its transposition table to learn from experience. *Journal of the International Computer Chess Association*, 10(2):59–71, 1987.

[42] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers.* Springer Verlag, 1997.

[43] R. Gasser. *Efficiently Harnessing Computational Resources for Exhaustive Search.* PhD thesis, ETH Zürich, Switzerland, 1995.

[44] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[45] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[46] D. Dailey. Email message, May 27, 1999.

[47] J. Baxter, A. Tridgell, and L. Weaver. Experiments in parameter learning using temporal differences. *Journal of the International Computer Chess Association*, 21(2):84–99, 1998.

[48] D. Beal. Learning piece values using temporal differences. *Journal of the International Computer Chess Association*, 20(3):147–151, 1997.

[49] J. Fürnkranz. Machine learning in computer chess: The next generation. *Journal of the International Computer Chess Association*, 19(3):147–161, 1996.

[50] D. Billings, L. Pena, J. Schaeffer, and D. Szafron. Using probabilistic knowledge and simulation to play poker. In *AAAI National Conference*, pages 697–703, 1999.

[51] B. Sheppard. Email message, March 9, 1999.

[52] H. Berliner. Backgammon computer program beats world champion. *Artificial Intelligence*, 14:205–220, 1980.

[53] H. Berliner. Computer backgammon. *Scientific American*, 242(6):64–72, 1980.

[54] G. Tesauro. Neurogammon wins computer olympiad. *Neural Computation*, 1:321–323, 1989.

[55] G. Tesauro. Email message, August 14, 1998.

[56] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[57] G. Tesauro. Email message, May 28, 1999.

[58] N. Zadeh and G. Kobliska. On optimal doubling in backgammon. *Management Science*, 23:853–858, 1977.

[59] G. Tesauro. Email message, April 6, 1999.

[60] E. Berlekamp. A program for playing double-dummy bridge problems. *Journal of the ACM*, 10(4):357–364, 1963.

[61] M. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *International Joint Conference on Artificial Intelligence*, pages 584–589, 1999.

[62] I. Frank. *Search and Planning under Incomplete Information: A Study Using Bridge Card Play.* Springer Verlag, 1998.

[63] M. Ginsberg. Partition search. In *AAAI National Conference*, pages 228–233, 1996.

[64] S. Smith, D. Nau, and T. Throop. Computer bridge: A big win for AI planning. *AI Magazine*, 19(2):93–105, 1998.

[65] S. Smith, D. Nau, and T. Throop. Success in spades: Using AI planning techniques to win the world championship of computer bridge. *AAAI National Conference*, pages 1079–1086, 1998.

[66] Mike Whittaker. The 1998 world computer bridge championships. *Bridge Magazine*, 96(10):22–24, 1998.

[67] O. Eskes. GIB: Sensational breakthrough in bridge software. *IMP*, 8(2), 1997. http://www.imp-bridge.nl/articles/GIB1Sens.html.

[68] C. Strachey. Logical or non-mathematical programmes. *Proceedings of the Association for Computing Machinery Meeting*, pages 46–49, 1952.

[69] T. Truscott. The Duke checkers program. *Journal of Recreational Mathematics*, 12(4):241–247, 1979-1980.

[70] M. Tinsley. Letter to the editor, 1980.

[71] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2–3):273–290, 1992.

[72] A. Samuel. Programming computers to play games. In F. Alt, editor, *Advances in Computers*, volume 1, pages 165–192, 1960.

[73] J. Condon and K. Thompson. Belle chess hardware. In M. Clarke, editor, *Advances in Computer Chess 3*, pages 45–54. Pergamon Press, 1982.

[74] R. Hyatt, A. Gower, and H. Nelson. Cray Blitz. In T. Marsland and J. Schaeffer, editors, *Computers, Chess and Cognition*, pages 111–130. Springer Verlag, 1990.

[75] H. Berliner and C. Ebeling. Pattern knowledge and search: The SUPREME architecture. *Artificial Intelligence*, 38(2):161–198, 1989.

[76] E. Felten and S. Otto. A highly parallel chess program. In *Conference on Fifth Generation Computer Systems*, 1988.

[77] F. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk. Deep Thought. In T. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 55–78. Springer Verlag, 1990.

[78] F. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk. A grandmaster chess machine. *Scientific American*, 263(4):44–50, 1990.

[79] J. McCarthy. AI as sport. *Science*, 276(June 6):1518–1519, 1997.

[80] F. Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess.* PhD thesis, Carnegie Mellon University, United States, 1990.

[81] G. Kasparov. IBM owes me a rematch. *Time Magazine*, 1997. May 26, American edition.

[82] P. Rosenbloom. A world-championship-level Othello program. *Artificial Intelligence*, 19(3):279–320, 1982.

[83] K-F. Lee and S. Mahajan. The development of a world class Othello program. *Artificial Intelligence*, 43(1):21–36, 1990.

[84] M. Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *Journal of the International Computer Chess Association*, 20(3):189–193, 1997.

[85] M. Buro. Logistello—A strong learning Othello program, 1997. www.neci.nj.nec.com/homepages/mic/ps/log-overview.ps.gz.

[86] M. Buro. Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. In J. van den Herik and H. Iida, editors, *Games in AI Research.* University of Maastricht, 1999. To appear.

[87] M. Buro. Statistical feature combination for the evaluation of game positions. *Journal of Artificial Intelligence Research*, 3:373–382, 1995.

[88] M. Buro. Toward opening book learning. *Journal of the International Computer Chess Association*, 22(2):98–103, 1999.

[89] M. Buro. Email message, May 17, 1999.

[90] D. Billings. *Computer Poker.* MSc thesis, University of Alberta, Canada, 1995.

[91] N. Findler. Studies in machine cognition using the game of poker. *Communications of the ACM*, 20(4):230–245, 1977.

[92] M. Caro. Email message, March 13, 1999.

[93] D. Papp. *Dealing with Imperfect Information in Poker.* MSc thesis, University of Alberta, Canada, 1998.

[94] J. Schaeffer, D. Billings, L. Pena, and D. Szafron. Learning to play strong poker. In *ICML Workshop on Machine Learning in Game Playing*, 1999.

[95] D. Billings, D. Papp, J. Schaeffer, and D. Szafron. Opponent modeling in poker. In *AAAI National Conference*, pages 493–499, 1998.

[96] S. Shapiro and H. Smith. A Scrabble crossword game-playing program. Technical Report 119, State University of New York at Buffalo, Department of Computer Science, 1977.

[97] D. Levy and D. Beal, editors. *Heuristic Programming in Artificial Intelligence.* Ellis Horwood, 1987.

[98] A. Appel and G. Jacobson. The world's fastest Scrabble program. *Communications of the ACM*, 31(5):572–578, 585, 1988.

[99] S. Gordon. A faster Scrabble move generation algorithm. *Software Practice and Experience*, 24(2):219–232, 1994.

[100] B. Sheppard. Email message, October 23, 1998.

[101] B. Sheppard. Email message, June 1, 1999.

[102] M. Müller. Computer go: A research agenda. *Journal of the International Computer Chess Association*, 22(2):104–112, 1999.

[103] V. Allis. *A Knowledge-Based Approach to Connect-Four. The Game is Solved: White Wins.* MSc thesis, Vrije Universiteit, The Netherlands, 1988.

[104] V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, University of Limburg, The Netherlands, 1994.

[105] R. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI National Conference*, pages 700–705, 1997.

[106] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In *International Joint Conference on Artificial Intelligence*, pages 570–575, 1999.

[107] G. Keim, N. Shazeer, M. Littman, S. Agarwal, C. Cheves, J. Fitzgerald, J. Grosland, F. Jiang, S. Pollard, and K. Weinmeister. Proverb: The probabilistic cruciverbalist. In *AAAI National Conference*, pages 710–717, 1999.