CS 242 Term Project: Advanced Game Playing Algorithms

Maksim Orlovich

7th May 2002

Abstract

Game playing represents a rather peculiar corner of AI: on one hand, the discrete, well-defined, strictly-governed game boards are a lot easier to deal with than reality, as the actions always have direct effect, and the goals are well known. On the other hand, game playing involves a competition between different agents, one of whom maybe often be a human being, so a game-playing program must be able to deal with creativity human players typically exhibit. Because branching factors are generally reasonable, most commonly-used approaches are derived from the brute-worse MiniMax search, with the addition of pruning. This paper examines the development of traditional search algorithms, the advancements that have been made recently, and provides basic case studies of the application of these principles in real game-playing systems. It also attempts to highlight some of the newer, more radical approaches, which, although far from dominance, often exhibit some significant advantages over the more traditional search-based algorithms, and which might eventually come to dominate the field.

Contents

1 Minimax Searches **3**

1.1 Basic Minimax and Alpha-Beta searches 3

1.2 Negamax. 5

1.3 Transposition tables. 6

1.4 Narrow window searches. 7

1.5 Scout and NegaScout 8

1.6 MT-SSS* and MTD(f) 9

1.7 Tournament applications 11

1.7.1 Deep Blue II 11

1.7.2 Logisthello 12

1.7.3 Crafty 13

2 Beyond minimax **15**

2.1 Opponent modeling. 16

2.2 Adversarial planning. 17

2.3 Heuristic pruning 18

2.4 Learning evaluation functions. 20

3 Conclusion **21**

1 Minimax Searches

1.1 Basic Minimax and Alpha-Beta searches

Minimax search is the foundation for most of the common game-playing programs right now, although the basic algorithm itself is no longer useful.

The idea behind minimax is simple: when playing a two-person game, one player attempts to improve the board situation for himself, while the opponent is trying to prevent that. Thus, one can model such competition by having the opponent always chose the best defending move – the one that makes things worse for the game-playing program, and have the program chose the option that forces the opponent to give up as much as possible. Of course, since searching the entire game tree would be impossible, one would probably try to search to some fixed limit in the simplest implementation:

```
ChooseSelf(boardState, depth, depthLimit)
    if (depth == depthLimit):
        return (boardValue(boardState), -1)
    bestValThusFar = -∞

    for move in legalMoves(boardState):
        curVal = ChooseOpponent
                       (simulateMove(boardState,move),
                       depth+1, depthLimit)[0]
        if (curVal > bestValThusFar):
            bestValThusFar = curVal
            bestMove       = move

    return (bestValThusFar, move)


ChooseOpponent(boardState, depth, depthLimit)
    if (depth == depthLimit):
        return (boardValue(boardState), -1)
    bestValThusFar = ∞
```

```
for move in legalMoves(boardState):
    curVal = ChooseSelft
                   (simulateMove(boardState,move),
                   depth+1, depthLimit)[0]
    if (curVal < bestValThusFar):
        bestValThusFar = curVal
        bestMove       = move

return (bestValThusFar, move)
```

There are a number of problems with this. The biggest one is that this is entirely brute force – it searches absolutely everything, regardless of whether the moves are horrible ideas which would never matter in practice. Another is that stopping at some limit will likely force one to miss something; and even if one tries to search deeper in interesting situations (doing "quiescence search"), there will still be a stopping point soon, as the branching factor is simply too high. Even with more advanced algorithms, on very high-end hardware, even more-or-less forced sequences often can not be searched in entirety – the Deep Blue team has reported that difficulty. A more subtle problem is that the assumption that the opponent always makes the best defensive move is not realistic against human opponents , who do not search everything, and may miss a really odd winning combination.

Alpha-beta pruning partially addresses the search width performance, by having the code for each node cut off the search when it is guaranteed to produce something that is worse than what it already has; this is done by maintaining bounds (α, β) on the values of the best move thus far for the two players, and cutting off branches of search that permit opponents to beat them. When combined with a heuristic ordering function, that puts apparently stronger moves first (which can be often done by using an iterative-deepening framework), it can often reduce the branching factor to nearly the square root of the minimax one, thus doubling the search depth ([Pe]). However, what this does not address (and what further improvements,

that also often focus on branching factor don't address), are the other issues: with limited search depth and cutoffs based on the assumptions of a perfect opponent searching to the same depth, some of the pruned regions may be blind spots that hide a winning strategy for self or opponent (This can, in some cases, be exploited by cleverly constructed computer opponents, as will be discuss in the second part of this paper). It is also still brute force to a large extent, as the pruning is to a large degree mechanical, and selectivity is quite limited.

1.2 Negamax.

One of the simplest improvements to MiniMax did not accomplish much algorithmically, but it provided an elegant simplification of the basic algorithm which was used in many further versions, which is why it is mentioned here. The observation is simple: in most cases, the scores for two players of a game state are exactly the additive inverses of each other; so if there is a simple way of doing separate evaluations for two players (perhaps "flipping" the game board efficiently), the code for minimax can be simplified considerably, to have simply a single routine, instead of two co-routines:

```
ChooseNega(boardState, depth, depthLimit, side)
    if (depth == depthLimit):
        return (boardValue(boardState,side), -1)
    bestValThusFar = -∞

    for move in legalMoves(boardState):
        curVal = -ChooseOpponent
                       (simulateMove(boardState,move),
                       depth+1, depthLimit, 1-side)[0]
        if (curVal > bestValThusFar):
            bestValThusFar = curVal
            bestMove       = move

    return (bestValThusFar, move)
```

There is a similar version of alpha-beta search, of course; only the values of alpha and beta need to be swapped and have their signs changed for recursive calls.

1.3 Transposition tables.

One obvious problem with the above searches is that they may search various transposition of move sequences repeatedly, while those sequences are actually identical in most cases. An obvious idea is to tradeoff some memory for speed, and to store a dictionary (typically, a hash-table) of the positions and their values; yet there are some complexities when this is used within an $\alpha - \beta$ framework – one needs not only to remember positions, but also to adjust values of α and β accordingly; and if values of α or β are stored; it is not immediately clear what relation they have with the present ones.

The important realization for solving this dilemma is that alpha-beta searches of portions of the tree do more than provide best known moves – they also provides bounds on possible values, through narrowing of the α and β window; so if these are stored, as well as the classification of the bound as upper or lower, one can use this information to update the bounds properly. Unfortunately, there are still some hard-to-resolve difficulties – what happens if the search which produced the values was extended for some reason, beyond the typical bound, producing the values that are somewhat difficult to compare directly with those of lower levels? This is an instance of the search instability problem – which may causes apparently inconsistent results for searches done slightly differently, and there are no really good answers – the two most commonly used ones are to ignore the problem, or to store the search depth within transposition table entries, and to only use those of matching depth – which solves this instance of the instability problem, but reduces the effectiveness of the tables.

A common application of transposition tables is for move ordering, when combined with iterative deepening: simply do alpha-beta for depth 1, then use the results stored in the transposition table to order the moves for depth 2, then results from there to order depth 3, and so on. This is typically a pretty effective heuristic, and the effect of pruning far outweighs extra work done on lower levels. One important considerations for its application is, however, that the deepening might have to be done 2-ply at a time if the game score can change significantly after a single ply and be restored close to original after the next one ([Pl]). Of course, iterative deepening itself it useful in tournament conditions, to be able to interrupt the search somewhat cleanly – if the time runs out in the middle of level 7, at least level 6s's results are available.

1.4 Narrow window searches.

The observation that failed $\alpha - \beta$ searches provide bounds on the results can be used to do pruning more efficiently. One important property of many games is that state of the board doesn't change wildly due to a single move – it's not usually the case that one move someone is with severe advantage, and the next that same player is about to move. Therefore, after each move there is a window of possible evaluation scores that are extremely likely to occur. One can easily modify alpha-beta to take advantage of this observation, by simply giving it the narrow window for the initial α and β values, instead of the canonical $-\infty$ and ∞. Most of the time, the search will return with the expected value, and it will do it quicker than alpha-beta would, since the pruning gets rid of many of "silly" moves. Of course, the search can also fail, and at this point the program has to re-search. Fortunately, since the failed search returns a lower or an upper bound on the result, it can be used to restrict the window on the re-searches – except for search instability issues. When search instability arises, it

is possible that the bounds contradict themselves; and thus programs implementing this technique, known as "Aspiration window" search may need to have additional complications to handle inconsistent results sanely.

1.5 Scout and NegaScout

The Scout algorithm, proposed by Judea Pearl, takes the idea of narrowing the window to the extreme, by making use of search with windows that are too small to ever produce an exact value. Such searches always produce only bounds. Of course, since at the root node an exact value is needed, doing searches so restricted is in general counterproductive, but it turns out that at many of the inner nodes a bound is sufficient. The Scout algorithm takes advantage of this by searching the move considered likely best by the ordering heuristic with a wide window, and further nodes with a null window. Scout considers a proper subset of the nodes alpha-beta would, but is a comparably easy to implement algorithm. A somewhat improved version is NegaScout, which is NegaMax-based and provides some performance improvements at the leaves ([Pl]):

```
NegaScout(boardState, alpha, beta)
    if isLeaf(boardState):
       return boardValue(boardState)

    children = orderedLegalMoves(boardState)
    candidate = NegaScout(children[0], α, β)
    if maxNode(boardState):
        b = max(alpha, candidate)

        for child in children[1:]:
            if (candidate < beta):
                break

            test = NegaScout (child, b, b+1)
```

```
            if isLeaf(child) or isLeaf(children(child)[0]):
               candidate = test //Always works near leaves.

            if test > max(alpha, candidate) and test < beta:
               test = NegaScout (candidate, test, beta) //Re-search

            candidate = max(candidate, test)
            b = max (b, test)

    else: //Min node
        b = min(beta, candidate)

        for child in children[1:]:
            if (candidate > alpha):
                break

            test = NegaScout (child, b-1, b)
            if isLeaf(child) or isLeaf(children(child)[0]):
               candidate = test //Always works near leaves.

            if test < min(beta, candidate) and test > alpha:
               test = NegaScout (candidate, alpha, test) //Re-search

            candidate = min(candidate, test)
            b = min (b, test)
    return candidate
```

NegaScout combined with aspiration window search at the root is in common use.

1.6 MT-SSS* and MTD(f)

Further improvements to search performance have been achieved by researches ([Pl, PlScPiBr]) by eliminating wide window searches entirely. In the MT framework, developed in late 90s, the central operation is the "Memory-enhanced Test", or MT, which acts much like the null-window testing searches in Scout, but with addition of use of transposition tables to store partial results. This makes it possible for the algorithms built on this framework to do re-

searches a lot more efficiently, as the results from previous partial searches of the tree can be re-used.

A somewhat surprising algorithm based on this framework it MT-SSS*. This algorithm essentially it visits the same leaf nodes, in the same order) implements the SSS* algorithm - an "alternative" algorithm, which has long been considered to be better at pruning than alpha-beta, but too difficult to understand and implement to use in practice - as an instance of alpha-beta derived search. Classic SSS* is an iterative best-first algorithm that tries to find a good move by tightening an upper bound on its value. MT-SSS* uses memory-tests to examine the nodes SSS* would, but it is a tiny program driving the MT routine:

```
MT-SSS*(boardState)
   g = +∞
   do:
      gamma = g
      g = MT(boardState, gamma)
      //Equivalent to calling Alpha-Beta(boardState, gamma-1, gamma)
   while g ≠ gamma
```

A similar algorithm can implement DUAL* by searching from $-\infty$ instead of $+\infty$. Empirical studies of these algorithms, however, suggest that they, despite expanding fewer inner nodes than NegaScout, expand more total nodes, and are only slightly faster than alpha-beta in practice. The difference from earlier predictions is largely due to the fact that unlike them, the research into MT-based algorithm made comparisons against NegaScout and a realistic - i.e. with transposition tables ordering heuristics, variable depth - version of alpha-beta, using actual game playing programs as testbeds.

More interesting results can be achieve by using somewhat more complicated drivers. One approach is MTD(bi), which tries to bisect the search window using MT calls. A more effective one is MTD(f), which tries to use values from iterative deepening to guess a

good starting point. MTD(f) expanded about 5-10% less leaf nodes than Aspiration-window NegaScout in experiments, as well as fewer inner nodes. It was also able to perform 5-15% faster than native implementations of NegaScout in programs as advanced as the checkers champion Chinook, and more for less "advanced" game playing programs. Although this is not earth-shattering, it certainly means that an MTD(f)-based game playing program might be able to do 10% more work evaluating nodes, or perhaps be able to reach greater search depths somewhat. This is particularly significant since MTD(f) is extremely simple in its base form, and thus has a lot of potential for improvements and add-ons.

1.7 Tournament applications

1.7.1 Deep Blue II

The designers ([CaHoSh]) of the well known Deep Blue chess computer has solved the problem of beating the human world champion in a rather direct fashion: by building a fast machine, with lots of general-purpose processors, and specialized hardware designed exclusively for chess playing. As such, Deep Blue is probably more of a marvel of computer engineering than one of computer science. The machine consists of a 30-processor cluster of IBM RS/6000 computers, with 480 dedicated chess-playing co-processors. It is capable of searching 100 million moves per second in difficult positions, and more in quiescent ones – it was able to achieve the rate of 330 million positions per second at one point when playing against Garry Kasparov. The search algorithm is NegaScout with transposition tables, iterative deepening and quiescence search. The evaluation function is implemented in hardware, and it incorporates 8000 different features. The weights are programmable, and the function is able to either perform quick (and relatively rough) evaluations, which take only a single clock cycle, or more complex ones, as needed. The main innovations where in the search

control software, which was able to try to focus on the most promising forced move sequence; and could combine the more sophisticated software algorithms with fast null-window searches performed by the hardware. Another interesting feature was the extended gamebook; which summarized 700,000 various chess games, and allow deep blue to lookup similar moves, and if they were proven in practice, to bump up the score returned by the search for them.

1.7.2 Logisthello

Logisthello is an Othello (a somewhat modified version of Reversi) playing program that was able to beat the games human champion 6-0 in tournament game, while running on an ordinary Pentium Pro 233 computer ([Bu3]). Othello, due to its relatively low branching factors and simple playing rules has been a relatively "easy" target for searching programs; as they can achieve significant search depths on ordinary hardware, and can play simply perfectly near the endgame. However, there is a significant difficulty involved: othello lacks the depth of analytic literature present in chess and checkers, so creation of a good evaluation functions is a difficult task. Therefore, quite a bit of research has gone into learning the evaluation functions for the game. In fact, while the first programs used hand-crafted evaluators, next generations of programs shifted to learning component weights, and Logisthello is in fact capable of learning values of various (small rectangles, lines, diagonals of various lengths, etc.) patterns of disks on the the board, separated out by game stages, and assigning them various shared of the total evaluation value. The approach also has the advantage of creating very fast table-based evaluators. The first versions of the learning algorithm took final scores, tried to use differentiation to approximate the contribution of various parts to the partial positions, and then used a logistic regression to fit an evaluation function. The approach, however, didn't account for correlations between various patterns properly, and the second

version of the program addressed that, by learning over combinations of patterns as well, considering presence or absence of a pattern to be a binary value. According to the designer, the improvements in playing strength produced by the stronger evaluators were equivalent to a 10-fold speed up, which is far beyond anything recent improvements in searching produced. The basic search strategy employed by the program is familiar - NegaScout , with transposition tables, iterative deepening, and quiescence search. Yet there is a significant modification – Logisthello employs a probabilistic cutoff technique, called ProbCut, to avoid searching portions of the game graph that are extremely unlikely to contain anything interesting, and to focus on the more relevant portions of the search space ([Bu2]). An improved version of this technique, called MultiProbCut, is discussed in the second section of this paper.

1.7.3 Crafty

Crafty([Hy]) is a free chess-playing engine, with freely available source code. Although it did not beat any human world champions, it is a very strong chess program, and the availability of the source code makes it possible to study it in more detail than other programs. It is also quite fast, being able to search 420,000 positions a second on a desktop Celeron 950 computer.

The engine stores game state as bit boards, packing information about positions in tightly; this also permits the use of assembly routines in the most performance-critical portions of the source code (such as attack tests, statistic gathering functions), as well as the streamlining of operation with bit and/or operators.

Iterative deepening is used to drive the search algorithm, in combination with using transposition tables to store results from lower levels. An aspiration window search is performed

at top level, first trying a window about 1/3 of a pawn wide of previous depth, and then collapsing it to a null window, re-expanding it if needed. The algorithm used is Principal Variation Search, which is quite similar to Scout. Crafty is also able to search on the opponents time, using the principal variation or the playbook to try to anticipate the opponent's moves. Quiescence search is implemented as well – searching only captures;

A number of move ordering heuristics are employed. First, the moves bounds for which are in the hashtable are considered when they are likely to be a part of the principal variation (i.e. the likely line of play). Then, the captures, and promotions are tried. Beyond that, the history heuristic is used, keeping track of how successful each move was in the past, using it as an indication of its usability in the future. The engine also checks whether any pieces are under attack, and demotes the moves that leave them undefended to the end of the search order. Also, penalties/bonuses for various piece positions found by the evaluator are used to guide the search. At the root, the engine uses captures-only search values for guidance.

The evaluation function combines a number of factors. Of course, the relative weight of material is included, as its significance is rather dramatic. Passed pawns are checked for explicitly, and it is analyzed whether the promotion can be stopped. It is checked whether there are pawns trapping bishops. Before castling, development is evaluated. So is the protection of the king. At this point, it is checked whether the score is quite extreme (to the extent that additional evaluations can't avoid a cutoff), and if it is, either an alpha or beta cutoff is performed, avoiding more subtle (and costly) evaluations. Here, things like trapped rooks, are evaluated, as well as positional strengths of knights, bishops, queens, etc. Overall, the evaluator is extremely complicated, involving 1800 lines of code, consisting of mostly very dense conditionals, as well as the underlying data tables.

Crafty employs an opening book implemented as a hash table. The usage of actual moves

varies a lot on what role the engine is performing. For instance, in tournaments the engine may attempt to avoid well-known lines of play, since they will often result in a quick response from the opponent. The book is also used when thinking during the opponent's time, to pick out the move the opponent will likely make. The book can be learned by maintaining statistics on move usefulness in early play. The learning algorithm mostly looks at extreme swings in score for the move, and doesn't care much for minor variations, when updating the values.

The source code underscores how crucial a simple underlying algorithm must be – the complexities of chess, the subtleties of evaluations of positions (and of interaction with users) are quite overwhelming, and every single optimizations takes more code, adds more complexity, etc. The flexibility and code simplicity of alpha-beta based search algorithms makes it possible to mold them and tune them into well-performing game programs. The process is quite a bit of an incremental one – at the time of writing, Crafty was at version 18.14; yet it underscores the reasons for popularity of alpha-beta and its descendants – they are capable of being part of a true game-playing system – they are simple, and work well, which is as good as it gets for most things when it comes to software engineering.

2 Beyond minimax

Although minimax techniques have proved themselves in many games, they simply perform inadequately in others. Further, the basic assumption – that of a perfect play by the opponent – isn't generally true when the opponent is human (and is thus not doing full-width search). Thus there have been quite a bit of research into alternative approaches. And even when the basic minimax-style framework is used, the question of the evaluation function is not an easy one, as even in case where there is enough expert knowledge to create a very good

one (i.e. in chess), the effort involved in implementation is enormous; and thus automatic learning approaches are quite attractive. There is also a very difficult question of trying to model the opponent, which is crucial in many games with more than two players, as well some probabilistic games like Poker.

2.1 Opponent modeling.

One of the techniques commonly used by human in playing various games is attempting to understand the opponent's intentions in the play, and to try to anticipate a flaw in their plan. The basic idea underlies Opponent Model search - OM - which, under the unrealistic assumptions of knowing the opponent's evaluation function and search depth can perform this type of calculation. In the simplest terms, OM-search involves using the opponent's evaluation function for min nodes. A more advanced version of the algorithm, $(D,d)-$OM, has similar assumptions but is able to work at a different search depth (with the assumption that the a search depth is great than or equal to that of the opponent). The basic search algorithm (rather than the efficient one – the direct analogue of plain minimax) is actually quite simple. Before the opponent's depth has been reached, the code calculates 2 values at the node – for its model and the opponent's model. At the max node, the maximum's for both the players and the opponents nodes are returned; while at the min node, the algorithm returns the minimum for the opponent, and what it thinks the right value for that nodes it. After the opponent's search space ends, a basic minimax-style search is used. An important property of this search is that it can never do worse than minimax if the model of the opponent is correct. The algorithm as such, of course, would perform horribly, as it does no pruning; but pruning techniques are available. By using $\beta-$pruning (i.e. pruning on β values at max nodes) at upper levels, and standard $\alpha-\beta$ at lower ones (a combination

called $\alpha - \beta^2$), the branching factor can be reduced significantly, but the overall algorithm is still outperformed by $\alpha - \beta$ on node evaluation counts. If the evaluation function of the opponent is known, this algorithm has been shown experimentally to outperform basic minimax in practice, winning as 2/3 of matches at the same search depth ([GaIiUiHe]).

There are a number of difficulties with this approach other than the ones already described, however. Even to implement the algorithm for the same search depth as the opponent, one has to do a lot more work per node – which requires significantly faster hardware or software, something that by itself could produce a significant advantage. This limitations, however do not make the technique useless – although not applicable to tournament play, it is suggested for tutoring applications. A somewhat more sophisticated approach is probabilistic opponent modeling (PrOM). , which uses many different evaluations functions for opponents, weighted with different probabilities, and uses expected value calculations instead of direct evaluation. The approach performs somewhat better than alpha-beta on the same search depths, but scales poorly to great depths, and still requires knowing quite a bit about the opponent ([DoUiHe]).

2.2 Adversarial planning.

Go is one of the games that have been essentially insurmountable by alpha-beta. The main reason is the extraordinarily high branching factor (typically around 235), the long games (around 300 ply); yet there are some not related to the actual science involved – there was simply little academic research into Go, and developers of commercial programs have been very secretive about their approaches.

One approach to dealing with the complexities of the game is the use of adversarial planning. Adversarial planning expands traditional single-agent planning to multiple agents by

including the refutation of the opponent's goals into each agents goal set, and letting the players alternate, trying to refute each others advances and to force backtracking. The interaction creates a tree-like plan structure, which includes plans for dealing with contingencies (i.e. opponent's actions).

This, however, requires complicated knowledge engineering; and it is not at all clear how it could be adopted to online analysis. Therefore, first research examinations of the technique focused on solving simple, beginner-level problems. The planner was able to solve 74% percent of such problems, providing accurate explanations for its actions and details on how the defenses of the opponent could be countered. Further, in the case where it failed, the failure was attributed by researches to the limited knowledge base, and not the deficiencies of the basic technique. The results also show, that the technique has a higher cost per nodes expanded than search – about 3.5 times more than alpha-beta in the test implementation, but with node counts growing considerably slower – the program was able to discover 33-ply winning sequences, something alpha-beta can't do in even simpler games. Thus, the approach has a lot of potential, but whether it can be used to create an on-line playing program remains to be answered.

2.3 Heuristic pruning

A big difference between between human and computer players in the search space – humans take a much more focused approach, pruning many moves, while $\alpha-\beta$ only prunes moves that wouldn't affect the minimax value. Some pruning heuristics are in wide use – particularly the null move heuristic. It is based on the fact that in many games, in most situations, giving someone two moves in a row is going to be helpful. Thus, one way to use this is, at max nodes, is, as a preliminary test to go straight to evaluating min's options, using reduced

search depth, and to see whether the score is severe enough to cause a cut off, and to take it if that happens. There are some situations in chess, and other games, however, when such a heuristic can produce outright disastrous estimations, and therefore it is typically used with care; yet it is beneficial enough to be used in most of today's game playing problems.

Another approach is ProbCut, used by Logisthello ([Bu1]). It uses the observation of correlation between values of shallow searches and their deeper versions, to try to find out which nodes have an extremely low probability of avoiding a cutoff if the search was continued, and prunes the tree when that occurs. Surprisingly, the values at bigger depths can often be adequately approximated by a linear-regression-fit line. Then, at game time, when the desired shallow search depth has been reached, the linear formula is evaluated, and it is checked whether the approximated value is more than a few times more than the variance of error of such an approximation outside of error range. When this heuristic was added to Logisthello, is was able to out play its basic version, capturing 74% of all disks in games played.

MultiProbCut is a significant incremental enhancement of ProbCut, which performs the probabilistic checks at various depths (and not just one), is configurable dependent on game stage, and does additional checks to improve detections of extreme conditions. The version of Logisthello using this technique dominates the simple ProbCut one, scoring 72% of points in a long tournament. The experiments by Logisthello's authors showed that with MinProbCut the lookahead could be increased by as much as 5 to 7 ply compared to $\alpha - \beta$, and that the strength of the algorithm would be roughly equal to that of $\alpha - \beta$ if it was given only 4% of the time $\alpha - \beta$ is permitted. MultiProbCut is a standard technique in today's Othello programs, and has been applied successfully to Amazons, a game with an extremely high branching factor (about 600), but has not yet been used with success in chess.

2.4 Learning evaluation functions.

Logisthello certainly showed the power of computer-learned evaluations functions in a young game which is not yet well understood intellectually. It was also very successful in Backgammon, which is a challenge for game playing due to the presence of randomness which results in extremely high branching factors. TD-Gammon[Te], a dominant computer player, uses a neutral network taught using temporal difference credit assignment as its evaluation function. The networks itself is a standard multilayer one, with one layer of hidden nodes, that is typically used with the classical back propagation algorithms. Teaching it, however, is more complex since one needs to not only assign credit to various parts of the network, but also various moves in the game. To handle the game part of the issue, a practice game is played, and then positions at various sequential moves are all given as the input to the neural network. The actual learning algorithm is quite similar to back-propagation, but an additional lambda factor is used to diminish the significance of the input patterns corresponding to earlier moves.

The performance results were quite significant – even given no special features, on an entirely unpatterned input, the network was able to develop recognition for the game's basic strategic properties. With addition of hand-designed input patterns, the algorithm was able to out play all of its computer opponents – while doing only 2-ply search. The program is estimated to be at a level comparable to that of advanced human players – some analysts have placed is at a level of a human master; and many top players have commended it for exhibiting positional analysis superior to any human being. Its games have also influenced the game strategies in use by human masters, as the program could produce original ideas that proved more effective than traditional openings. An important factor of the success is the random element of the game, which permits self-play to results in a wide variety of

situations, and which gets rid of the potential for the program to develop some pathological function simply to out play itself.

3 Conclusion

Numerous advances have been made by research in game playing. The basic alpha-beta algorithm has been displaced by faster descendants such as the NegaScout and MTD(f), and a lot of expertise has been developed in creating fast, well-tuned implementations.

There are also significant accomplishments when it comes to machine-learned evaluation functions – the evaluators in TD-Gammon and Logisthello are capable of evaluating properties of board states that are beyond the scope of human understanding. Ideas like ProbCut and the Null Move Heuristic also helped to transform search algorithms somewhat, permitting them to ignore many irrelevant nodes, and to search deeper.

A number of difficult problems still remain – computer Go and opponent-awareness are at very early stages; yet overall, and impressive array of accomplishments have been produced by the creators of game-playing algorithms and programs – in checkers and Othello computers easily dominate best human players; in backgammon TD-Gammon is able to come close to the best, and to shape the development of the game with fresh ideas. What's also quite remarkable is that in many of these games, the advent of strong computer players has only strengthened the interest in the respective game, by providing new insight, and by making strong practice opponents available to players worldwide.

References

[Bu1] Buro, Michael. "Improving heuristic mini-max search by supervised learning". *Artificial Intelligence*. 134(2002), 85-99.

[Bu2] Buro, Michael. "The Evolution of Strong Othello Programs". *NEC Research Institute Technical Report.*

[Bu3] Buro, Michael. "Takeshi Murakami vs. Logisthello". *NEC Research Institute Technical Report.*

[CaHoSh] Campbell, Murray. Hona, Joseph A. Jr. Shu, Feng-hsiung. "Deep Blue". *Artificial Intelligence.* 134(2002), 57-83.

[DoUiHe] Donkers, H.H.L.M. Uiterwijk, J.W.H.M. Van den Herik, H.J. "Probabilistic opponent-model search". *Information Sciences* 135(2001) 123-149.

[GaIiUiHe] Gao, Iida, Uiterwijk, and Van Den Herik. "Strategies anticipating a difference in search depth using opponent-model search". *Theoretical Computer Science.* 252(2001) 83-104.

[Hy] Hyatt, Robert. "Crafty" Chess Engine. `http://www.limunltd.com/crafty/`

[Pe] Pearl, Judea. "The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and its Optimality". *Programming Techniques and Data Structures.*

[PlScPiBr] Plaat, Aske. Schaffer, Jonathan. Pijls, Wim. De Bruin, Arie. "Best-first fixed-depth minimax algorithms". *Artificial Intelligence* 87(1996), 255-293

[Pl] Plaat, Aske. *Research Re:Search &Re-Search.* Doctoral Thesis, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam.

[Sc] Schaeffer, Jonathan. "The Games Computers (and People) Play". Univeristy of Alberta Technical Report.

[Te] Tesauro, Gerald. "Temporal Difference Learning and TD-Gammon". *Communications of the ACM.* March 1995/Vol 38, No 3.

[WiRiBuLe] Willmott, Richardson, Bundy, and Levine. "Applying adversarial planning technies to Go". *Theoretical Computer Science.* 252(2001), 45-82