Intelligent Strategy for Two-person Non-random Perfect Information Zero-sum Game

TONG Kwong-Bun

A Thesis Submitted in Partial Fulfilment

of the Requirements for the Degree of

Master of Philosophy

in

Information Engineering

Acknowledgement

First, I would like to express my deep gratitude to my supervisor, Professor Wing-shing Wong. It is fair to say that none of this work would have been possible without his encouragement, support and unlimited guidance throughout these years. I admire him for his broad knowledge and deep insight, as well as for his trust and patience. He is also so kind that he helped me with his full effort when I was in trouble. This manuscript has been improved considerably from its first draft thanks to comments from Prof. Wing-shing Wong. He read the thesis several times and made incisive suggestions which forced me to add some materials and made the whole thesis more substantial. He also kindly help improving the language of the thesis which hopefully has made it easier to read.

I would like to thank Prof. Will W. Ng who have tutored me much on minimax searching and introduced me the ACM Computer Chinese Checkers competition during my undergraduate studies. His previous input was invaluable in transforming my rough ideas into the research result that I have obtained. I would also like to thank Mr. Clarence C. Y. Chan and Mr. Kenny W. I. Lam for providing me their source and binary file of their Chinese Checkers program respectively. Without their donation, I would not have been able to participate the competition and complete my research.

I am pleased to express my gratitude to all my former and present lab-mates at Performance Evaluation Laboratory, who have created an excellent atmosphere there, especially Dr. Simon L. Y. Chan, Dr. Terence H. L. Chan, Mr. K. K. Leung, Mr. H. Y. Kwan, Mr. Eric K. Hu, Mr. Andy Y. F. Cheng, Mr. Calvin C. S. Chan, Mr. S. W. Ho, Mr. W. C. Chan and Mr. Michael W. Y. Ge. In these years, Mr. K. K. Leung shared his research experience with me and gave me a lot of good ideas, Mr. H. Y. Kwan had had a lot of stimulating discussion with me and Mr. Andy Y. F. Cheng provided a complete technical support for me. Without their help and encouragement, none of my work would be possible.

Finally, I would like to thank my family. I thank my mother and father for supporting me over the years, through love and friendship. I thank my sister for helping take care of me as I grew up. And I thank my brother-in-law for being my friend throughout these years. It is to my family that this thesis is dedicated.

Abstract

This thesis describes a new strategy to achieve a new plateau in computer chess performance. Experimental results show that the improvements can reduce search effort by 60%. The significant reduction of search effort gives room for deeper search in which better performance can be guaranteed.

Good human players conduct a highly selective look-ahead search in which they only rarely miss decisive variations. In the presence of a good evaluation function, selective Alpha-Beta searches based on ProbCut can approximate the focussed human search behavior. However, ProbCut still have to search many more path in order to come up with decisions of competitive quality. This thesis shows that there is still room for improvement.

In this work, we have developed a probabilistic forward pruning framework for two-person non-random perfect information zero-sum game. By reformulating Buro's ProbCut idea, we developed a generalised version of ProbCut called GPC such that other Alpha-Beta variants algorithms can benefit from it and result in a decreased search effort.

We also find that if the ordering of child moves is reasonably good, we can immediately stop the search and return the best minimax value found so far, once a shallow search yield a value outside the current search window. Experiments show that we can speedup the game of Chinese Checkers by 3 times.

In this thesis, we also developed a new strategy for two-person non-random perfect information zero-sum game. The strategy is the integration of forward pruning and node-cutting heuristic. By using the correlation and pattern information of the move ordering function, result of a short-depth search can be used to decide when to stop the search while keeping a reliable minimax value. Simulation results show that our strategy is superior in terms of the hit rate of minimax value as well as the speed of finding the minimax.

摘要

本論文旨在研究開發一新策略提昇計算機下棋的表現。實驗结果顯示搜尋工作量可以減少達百分之 60。搜尋工作量的大幅度減低給予計算機更多的空間作更深入的搜尋，從而保証有更優異的表現。

好棋手能夠高度地選擇性向前搜尋，他們很少錯過決定性變異的搜尋。在一個優良的評估函數協助下，ProbCut 選擇性阿爾貝他(selective Alpha-Beta)搜尋可以有接近好棋手的集中搜尋行為，然而，為了作出具競爭質素的決定，ProbCut 仍需要搜尋許多路徑，本論文顯示這仍存在改善空間。

在這論文裏，我們為二人非隨機完全資訊零總和(two-person non-random perfect information zero-sum)遊戲為本，開發一個機率向前修剪構架(probabilistic forward pruning framework)。我們重新演繹 ProbCut 並推廣出一個廣義版本，我們稱之為 GPC，使之其他阿爾貝他變異算法亦能降低其搜尋工夫從而受益。

我們并且發現如果走法排列(move ordering)不俗，一次淺深度搜尋出的 minimax 值在當前的搜尋視窗(search window)之外，我們能立刻停止整個搜尋並傳回目前覓得的最佳 minimax 值，實驗顯示我們能夠加快中國跳棋搜尋速度達 3 倍之多。

在這份論文中，我們還開發了一個新方案。這個策略是綜合了向前修剪技術和 node-cutting heuristic。基於走法排列的相互關係和式樣資訊，短深度搜尋的結果可以用作決定何時停止搜尋而仍然保留可靠的 minimax 值。實驗结果顯示我們的方案在正確 minimax 值的命中率及取得 minimax 值的速度都有優越的表現。

Contents

List of Figures

Chapter 1

Introduction

1.1 An Overview

To develope intelligent computer players for board games is a challenge which Artificial Intelligence research has been addressing since the field began. In 1949, Shannon started to surmise how computers could play chess. He proposed the idea that computers would need an evaluation function to successfully compete with human players [39]. In principle, one has to consider, for a given position, all possible moves, then all moves for the opponent and then all responses for opponent's moves, and so on until to the end of game. Each of the paths ends in a win, loss or draw. By working backward from the end one can determine whether there is a forced win, the position is a draw or is a loss. This is the general idea of Shannon's minimax search algorithm. However, it is not feasible to construct and compute such a tree. A typical chess game last about 40 moves and in each position there is on average $10^{1.5}$ legal moves, that is, in the order of 30. There will be 10^{120} variations to be calculated from the initial position [28]. For the case of Othello, another classic game that is in general less complex than chess, it last for 30 moves and on average 10 legal moves for each position [34]. The total number of moves counting from start game is 10^{60} which is still not feasible to modern computer. As a result, a heuristic evaluation function is needed. The general procedure of an intelligent program starts by looking ahead a few moves at a time and then evaluate the resulting board positions. In general, as no simple and exact evaluation is a priori known, evaluations must be based on game-specific knowledge to approximate the true function. Shannon suggested using a linear polynomial with variable coefficients to represent the evaluation function. Evaluation was made with respect to several selected parameters. In his chess, material advantage, pawn formation, positions of pieces, commitments, attacks and options and mobility was considered as the parameters. The construction of an intelligent program then falls into two problems, one is the method for fast tree searching and the other is the way to construct a good evaluation function. The former is related to the efficiency of the intelligent program while the latter is related to the quality of it. Researchers have paid attention to both problems. In this chapter we will discuss the previous works in these fields.

1.2 Tree Search

This section provides some background on minimax search algorithms. We briefly introduce the minimax function, and the concept of a cutoff. To find the value of the minimax function, one does not have to search the entire problem space. Some parts can be pruned; they are said to be cut off. The extension of basic minimax algorithm, Alpha-Beta, is discussed. Next we discuss common enhancements to Alpha-Beta.

1.2.1 Minimax Algorithm

Of central importance in most game-playing programs is the search algorithm. In trying to find the move to make, a human player would typically try to look ahead a few moves, predicting the replies of the opponent to each move, and the responses to these replies, and select the move that looks most promising. In other words, the space of possible moves is searched trying to find the best line of play. Game-playing programs mimic this behavior. They search each line of play to a certain depth and evaluate the position. Assuming that both players play perfectly, choose the move with the highest probability of winning for them, in each position the value of the best move is returned to the parent position. In a zero-sum game the loss of one player is the gain of the other. Player A tries to maximize the chance of winning the game; players B tries to maximize B's chance, which is equivalent to minimizing A's chances. Therefore, the process of backing up the value of the best move for alternating sides is called

```
function Minimax($n$) $\rightarrow f$
    if $n$ = LEAFNODE then return eval($n$);
    else if $n$ = MAXNODE then
        $g \leftarrow -\infty$;
        $c \leftarrow$ firstchild($n$);
        while $c \neq$ NOCHILD do
            $g \leftarrow$ max($g$, Minimax($c$));
            $c \leftarrow$ nextbrother($c$);
    else /* n is a min node */
        $g \leftarrow +\infty$;
        $c \leftarrow$ firstchild($n$);
        while $c \neq$ NOCHILD do
            $g \leftarrow$ min($g$, Minimax($c$));
            $c \leftarrow$ nextbrother($c$);
    return $g$;
```

Figure 1.1: The Minimax Function.

minimaxing; two-player search algorithms are said to perform a minimax search.

Figure 1.1 gives the recursive minimax function in pseudo code. The code takes a node n as input parameter and returns f_n, the minimax value for node n. Every node is either a leaf, a min, or a max node. An evaluation function, *eval*, exists that returns the minimax value for each board position at a leaf node. The functions $firstchild$ and *nextbrother* exists, returning the child node and brother nodes. If no child or brother exists, these functions will return $NOCHILD$. The minimax function traverses the tree in a depth-first order. The min and max operations implement the backing-up of the scores of nodes from a deeper level. The value of g represents an intermediate value of a node. When all children have been searched g becomes f, the final minimax value.

In many games it is not feasible to search all paths to the end of the game, because the complete minimax tree would be too huge. The evaluation function is changed to return a heuristic assessment.

An example of how a tree is traversed by the minimax algorithm is shown in appendix A.

```
function AlphaBeta($n$, $alpha$, $beta$) $\rightarrow g$
    if $n$ = LEAFNODE then return eval($n$);
    else if $n$ = MAXNODE then
        $g \leftarrow -\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g < beta$ and $c \neq$ NOCHILD do
            $g \leftarrow$ max($g$, AlphaBeta($c$, $alpha$, $beta$));
            $alpha \leftarrow$ max($alpha$, $g$);
            $c \leftarrow$ nextbrother($c$);
    else  /* n is a min node */
        $g \leftarrow +\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g > alpha$ and $c \neq$ NOCHILD do
            $g \leftarrow$ min($g$, AlphaBeta($c$, $alpha$, $beta$));
            $beta \leftarrow$ min($beta$, $g$);
            $c \leftarrow$ nextbrother($c$);
    return $g$;
```

Figure 1.2: The Alpha-Beta Function.

1.2.2 The Alpha-Beta Algorithm

In fact, to find the value of the minimax function, one does not have to search the entire problem space. Some parts can be pruned; they are said to be cut-off. This pruning idea builds up the Alpha-Beta algorithm. The enhancement to the minimax algorithm, in the best case, can search up to twice the search depth of full minimax.

Figure 1.2 gives the pseudo code for Alpha-Beta algorithm. It consists of the minimax function, plus two extra input parameters and cut-off tests. The *alpha* and *beta* parameters together are called the search window. At max nodes, current g is a lower bound on the return value. This lower bound is passed to the children as the *alpha* parameter. Whenever any of these children finds it can no longer return a value above that lower bound, further searching is useless and is stopped. At min nodes, current g is an upper bound. Parameter *beta* passes the bound on so that any max children with a lower bound greater than *beta* can stop searching. This is called cut-off. Together, *alpha* and *beta* form a search window which can be regarded as a task for a node to return a value that lies inside the window.

The g denotes the return value of an Alpha-Beta call. There are 3 cases for the return value:

1. $\alpha < g < \beta$ (success)
2. $g \leq \alpha$ (failing low)
3. $g \geq \beta$ (failing high)

For case 1, the search is success. The minimax value is equal to g. If it is case 2, it is failing low. It means that the correct minimax value is smaller than g, or from another point of view, g is the upper bound of it. Analogously, if it is case 3, it is failing high. The correct minimax value is greater than g. g is the lower bound of the true value. After each recursive Alpha-Beta calls, the lower and upper bounds become tighter until they converge. Usually Alpha-Beta is called with an initial window of $(-\infty, +\infty)$ to make sure the algorithm can find the minimax value.

The benefits of the algorithm come from the elimination of subtrees without search once it is proven their value must lie outside the alpha-beta search window. Subtrees eliminated in this manner are said to be cut-off. In the optimal case, for uniform trees of depth d and branching factor w, only $w^{\lceil d/2 \rceil} + w^{\lfloor d/2 \rfloor} - 1$ leaf nodes need be considered [19]. This can be said as the best case of Alpha-Beta.

Since its introduction in 1958 by Newell, Shaw and Simon [27], many additional heuristics and enhancements have been applied to achieve further speed-up. They are discussed in the following subsections.

A complete example of how a tree is traversed by the Alpha-Beta algorithm is shown in appendix A.

1.2.3 Alpha-Beta Enhancements

Move Ordering

The effectiveness of Alpha-Beta cut-offs is maximized if the best move is considered first at all interior nodes of the search tree. The size of the search tree built by the depth-first Alpha-Beta algorithm largely depends on the order in which branches are considered at interior nodes. The minimal game tree arises if the branch leading to the best minimax score is considered first at all interior nodes. Examining them in worst to best order results in the maximal tree.

As a result, a way to improve the effectiveness of Alpha-Beta pruning is to improve the order in which child positions are examined. On a perfectly ordered uniform tree Alpha-Beta

```
function AspWin($n$, $estimate$, $delta$) $\rightarrow f$
    $alpha \leftarrow estimate - delta$;
    $beta \leftarrow estimate + delta$;
    $g \leftarrow$ AlphaBeta($n$, $alpha$, $beta$);
    if $g \leq alpha$ then
        $g \leftarrow$ AlphaBeta($n$, $-\infty$, $g$);
    else if $g \geq beta$ then
        $g \leftarrow$ AlphaBeta($n$, $g$, $+\infty$);
    return $g$;
```

Figure 1.3: The Aspiration Window Searching Function.

will cut-off the maximum number of nodes. The approach is usually by using game-dependent knowledge to make a guess decision to order the moves. For example, in chess it is often wise to try moves that will capture an opponent's piece first, and in Othello certain moves near the corners are often better.

In real applications, like checkers, chess or Othello, however it is not easy to obtain perfect move ordering. An inexpensive estimation function is good enough if it is simple, can be compute fastly, and is close to perfect ordering, at least in the first few sibling nodes. Best-first move ordering becomes a reasonably good alternative as we will likely examine the best child first.

Techniques such as iterative deepening [15] and history heuristic [37] have shown that it is possible to achieve excellent move ordering. We can, therefore, obtain the function by these techniques.

Aspiration Window

Aspiration Search was first discussed and analyzed by Brudno [4], Marsland [23] and Marsland and Campbell [22]. In many games the values of parent and child nodes are correlated. Therefore we can obtain cheap estimates of the result that a search to a certain depth will return. We do a relatively cheap search to a shallow depth to obtain this estimate. This estimate can be used to create a small search window. This window is known as an aspiration window, since we aspire that the result will be within the bounds of the window. With this window an Alpha-Beta search is performed. If it succeeds, that is the returned value lies inside the small search window, then we have found the minimax value cheaply. If it fails (either failing low or failing high), then a re-search must be performed. Since failed search would return a bound, this re-search can also benefit from a window smaller than $(-\infty, +\infty)$. Aspiration window searching is commonly used at the root of the tree. One option for the estimate is to evaluate the current position. Aspiration window searching usually do more efficiently than Alpha-Beta(n, $-\infty$, $+\infty$). Figure 1.3 shows the pseudo code of aspiration window searching.

NegaScout

SCOUT [29] algorithm was motivated by the desire to reduce search effort by testing node values rather than by evaluating nodes. Full evaluation of a node is time consuming while node value testing is not. SCOUT offers little in the way of speedup over Alpha-Beta. Reinefeld has done some modifications to the algorithm to provide addtional cut-offs and named as NegaScout [32].

Pushing the idea of a smaller search window to the limit is the use of null-window. By assuming the minimax values are integer-valued, we can have a null-window if we choose *alpha* as $beta - 1$. This Alpha-Beta null-window search ensures the highest number of cutoffs as we can never find out a minimax value in between *alpha* and *beta*. We always obtain either failing low or failing high. However, the returned value serves as the bound of true minimax value. By using a null-window search, we can test the value of a node in a quick manner rather than evaluating it.

Figure 1.4 shows the pseudo code of NegaScout. It uses a wide search window for the first child but a null-window to the other children. At a max node the minimax value of the its first child node should be the greatest. If one of the null-window searches for other children returns a bound that is greater, then that child is indeed a better move and re-search with a wide window is needed to determine its minimax value. Analogously, at a min node the minimax value of its first child node should be the least. If the null-window searches show one of the brothers has a smaller minimax value, than that one becomes the most desired move and re-search is performed to determine its minimax value. NegaScout finds more cutoffs than Alpha-Beta and resulted in faster running time.

Using the technique of transposition table [41], which is commonly used in chess programs

```
function NegaScout($n$, $alpha$, $beta$) $\rightarrow g$
    if $n$ = LEAFNODE then return eval($n$);
    $c \leftarrow$ firstchild($n$);
    $g \leftarrow$ NegaScout($c$, $alpha$, $beta$);
    $c \leftarrow$ nextbrother($c$);
    if $n$ = MAXNODE then
        $b \leftarrow$ max($g$, $alpha$);
        while $g < beta$ and $c \neq$ NOCHILD do
            $t \leftarrow$ NegaScout($c$, $b$, $b + 1$);
            /* the last two ply of the tree return an accurate value */
            if $c$ = LEAFNODE or firstchild($c$) = LEAFNODE then $g \leftarrow t$;
            if $t >$ max($g$, $alpha$) and $t < beta$ then $t \leftarrow$ NegaScout($c$, $t$, $beta$);
            $g \leftarrow$ max($g$, $t$);
            $c \leftarrow$ nextbrother($c$);
            $b \leftarrow$ max($b$, $t$);
    else /* n is a min node */
        $b \leftarrow$ min($g$, $beta$);
        while $g > alpha$ and $c \neq$ NOCHILD do
            $t \leftarrow$ NegaScout($c$, $b - 1$, $b$);
            /* the last two ply of the tree return an accurate value */
            if $c$ = LEAFNODE or firstchild($c$) = LEAFNODE then $g \leftarrow t$;
            if $t <$ max($g$, $beta$) and $t < alpha$ then $t \leftarrow$ NegaScout($c$, $alpha$, $t$);
            $g \leftarrow$ min($g$, $beta$);
            $c \leftarrow$ nextbrother($c$);
            $b \leftarrow$ min($b$, $t$);
    return $g$;
```

Figure 1.4: The NegaScout Function.

to prevent re-searching of already evaluated positions, Negascout examines 20 to 30 percent fewer terminal nodes than alpha-beta algorithm [32]. Almost in the same time, Campbell and Marsland published a similar algorithm named as PVS (principal variation search) [21, 9]. But due to its complex implementation, it was not widely used.

Memory-enhanced Test

The previous subsection showed how null-widow Alpha-Beta searches can be used as an efficient method to compute the bounds. Plaat generalized the null-window searches to form MTD(f) algorithm as shown in Figure 1.5 and 1.6. MT is a null-window Alpha-Beta function with memory storage. The value and information of visited nodes are stored. They will be restored when re-visiting them. Since MT use the null-window search technique, it will always encounter the problem of re-visiting many visited nodes. With the help of memory storage, MT can save the overhead of re-search visited nodes. MTD is the driver function of MT. The driver function control the null-window test value for each MT function call. By choosing proper test value, the bounds of minimax value will converge and the desired minimax value is found.

At the root of a tree the return bounds from each MT function call are stored in upperbound (after Alpha-Beta failing low) and lowerbound (after Alpha-Beta failing high). The bounds delimit the range of possible values for the minimax value. Each time MTD(f) calls MT it gets a value back that narrows the range, and the algorithm is one step closer to hitting the minimax value. In order to get rid of the overhead inherent in multiple re-searches, storing nodes as well as their bounds in memory by transposition table is suggested. However, MTD(f) needs, and also makes use of a good first guess in order to find the minimax value efficiently. This can be done by feeding back the minimax value of last move, provided that the value is not oscillating. The improvement of MTD(f) over Alpha-Beta is even more than that of NegaScout over Alpha-Beta [31, 30].

1.2.4 Selective Search

Besides traditional full-width fixed-depth minimax search, we can have non full-width fixed-depth search. Unlike computer, human player would not examine all possible moves for a given

```
function MT($n$, $gamma$, $depth$) $\rightarrow g$
    if $depth = 0$ then /* leaf node */
        retrieve($n$)
        if $n.lowerbound = -\infty$ and $n.upperbound = +\infty$ then
            $g \leftarrow$ evaluate($n$);
        else if n.upperbound $= +\infty$ then
            $g = n.lowerbound$;
        else
            $g = n.upperbound$;
    else if $n$ = MAXNODE then
        $g \leftarrow -\infty$;
        $c \leftarrow$ firstchild($n$);
        /* $g \geq gamma$ causes a beta cutoff ($beta = gamma$) */
        while $g < gamma$ and $c \neq$ NOCHILD do
            retrieve($c$);
            if $c.upperbound \geq gamma$ then
                $g' \leftarrow$ MT($c$, $gamma$, $d$ - 1);
            else
                $g' \leftarrow c.upperbound$;
            $g \leftarrow$ max($g$, $g'$);
            $c \leftarrow$ nextbrother($c$);
    else /* $n$ is a MINNODE */
        $g \leftarrow +\infty$;
        $c \leftarrow$ firstchild($n$);
        /* $g < gamma$ causes an alpha cutoff ($alpha = gamma - 1$) */
        while $g \geq gamma$ and $c \neq$ NOCHILD do
            retrieve($c$);
            if $c.lowerbound < gamma$ then
                $g' \leftarrow$ MT($c$, $gamma$, $d$ - 1);
            else
                $g' \leftarrow c.lowerbound$;
            $g \leftarrow$ min($g$, $g'$);
            $c \leftarrow$ nextbrother($c$);
    /* Traditional transposition table storing of bounds */
    if $g \geq gamma$ then                /* Fail high result implies a lower bound */
        $n.lowerbound \leftarrow g$;
        store $n.lowerbound$;
    else                                  /* Fail low result implies an upper bound */
        $n.upperbound \leftarrow g$;
        store $n.upperbound$;
    return $g$;
```

Figure 1.5: The Memory-enhanced Test Function.

```
function MTDF(root, f, depth) → g
        g ← f;
        upperbound ← +∞;
        lowerbound ← -∞;
        repeat
                if g = lowerbound then gamma ← g + 1 else gamma ← g;
                g ← MT(root, gamma, depth);
                if g < gamma then upperbound ← g else lowerbound ← g;
        until lowerbound ≥ upperbound;
    return g;
```

Figure 1.6: The Memory-enhanced Test Driver Function.

board position but using their experience, narrow the game tree by pruning those unpromising variations in advance such that they can search to rather deep levels. This is how the idea of selective search comes from.

According to Shannon's description [39], tree search can be classified as two major types. Traditional full-width minimax searches are considered as Type-A strategies while selective searches are considered as Type-B strategies. Forward pruning is the major technique used by many selective search strategies. Unlike Alpha-Beta, due to its backtracking style of tree traversal, which only prunes nodes that will not be chosen, forward pruning techniques ignore all nodes that do not look very promising, thereby running the risk of missing the correct choice. We will discuss several well-known forward pruning techniques in the next part of this subsection.

Razoring

One heuristic that has been used to define a selective strategy is to expand only nodes that look at least as good as the current best. This heuristic defines a technique called razoring [3], a procedure that, at first glance, looks strikingly similar to Alpha-Beta. In fact, the only difference between them lies in the criteria used for determining the potential of a node. Alpha-Beta relies on backed up minimax value, razoring on a static evaluation. Thus, while razoring prunes nodes that do not look good, Alpha-Beta only eliminates nodes that are not good. Unlike Alpha-Beta, razoring cannot guarantee that it will find the minimax value. Razoring should be used in addition to Alpha-Beta, not instead of it. In the worst case, razoring will prune the same nodes as Alpha-Beta, with only the added cost of some extra evaluations. In the average case, however, razoring will prune nodes earlier than Alpha-Beta, narrow the branching factor more rapidly, and deepen the search, all in exchange for occasionally missing the best choice. The preliminary experiments described by [3] showed that in the exchange, razoring gained, on the average, an order of magnitude over Alpha-Beta in a four-ply tree, in terms of the number of nodes expanded.

B*

The B* algorithm [2] uses a simple heuristic of a very different nature. It "terminates the search when an intelligent move can be made". This algorithm was motivated by the desire to avoid the horizon effect by defining natural criteria for terminating search. The search proceeds in a best-first manner, and attempts to prove that one of the potential next moves is, in fact, the best. By concentrating only on the part of the tree that appears to be most promising, B* (and best-first searches in general) avoids wasting time searching the rest of the tree. Berliner's adaptation of best-first searches to game-trees included the first modification to Shannon's original model. Instead of associating a single value with each node, B* uses two evaluation functions, one to determine an optimistic value, or upper bound, and one for a pessimistic value, or lower bound. The search is conducted with two proof procedures, PROVEBEST, which attempts to raise the lower bound of the most promising node above the upper bounds of its siblings, and DISPROVEREST, which tries to lower the upper bounds of the siblings beneath its lower bound. The search terminates when the most promising choice has been proven best.

Although B* sounds particularly appealing from both the speedup and cognitive modeling viewpoints, it does have its drawback. Like all best-first searches, a good deal of storage space is needed to keep track of the promising nodes.

Conspiracy Search

Conspiracy search uses a heuristic to "attempt to stabilize the value of the root" [24]. The value of a node is stable if deeper searches are unlikely to have any major effect on it. In a conspiracy search, the root's stability is measured in terms of conspiracy numbers, the number of leaves whose values must change to affect its value. If the number of conspirators required to change the root value is above a certain threshold, the value is assumed to be accurate. At any given point during the search, the possible values of the root are restricted to the interval $[V_{min}, V_{max}]$, where V_{min} and V_{max} are the values of its minimum and maximum accessible descendants at the search frontier, respectively. To update the range, either prove that the minimizing player can avoid V_{max}, or that the maximizing player can avoid V_{min}. The decision of which to prove at each point can be made with the help of the conspiracy numbers.

ProbCut

The original minimax algorithm searches the entire game tree up to a certain depth and even with its efficient improvements such as Alpha-Beta pruning, null-window NegaScout search and MTD(f) is only allowed to prune backwards since they have to compute the correct minimax value. It was found that evaluations obtained from searches at different depths have strong correlation, provided that a reasonably good evaluation function exists [5]. The result of a shallow search can be used to decide with a prescribed likelihood whether a deep search would yield a value outside the current search window. Buro generalized the idea and came up with the probabilistic forward cuts heuristic [5].

The ProbCut selective search heuristics permits pruning of subtrees that are unlikely to affect the minimax value and uses the time saved for analysis of probably more relevant variations. This approach is based of the fact that values returned by minimax searches of different depths are highly correlated, provided that a reasonably good evaluation function and, if necessary, a quiescence search [1] [14] is used. A quiescence search extends the search at a leaf position until a quiet position is reached. In chess, "quiet" is usually defined as no captures present and not in check. In this case, a shallow search result v_s is a good predictor for the deep minimax value v_d.

A simple way to express the relationship between v_s and v_d is a linear model of the form $v_d = a \times v_s + b + e$ where a, b are real constants and e is a normally distributed error variable having mean 0 and variance σ^2 . These parameters are estimated by linear regression applied to a large number of training pairs $(v_d(p_i), v_s(p_i))$, where p_i is input of different board positions. After computing the shallow search result v_s, the search is terminated in the current position

```
function AlphaBetaPC($n$, $depth$, $alpha$, $beta$) $\rightarrow g$
    if $n$ = LEAFNODE then return eval($n$);

    //ProbCut heuristic:
    else if $depth = d$ then
        $bound \leftarrow$ round($beta$ - $b$ + $t$ * $sigma$) / $a$);
        if (AlphaBetaPC($n$, $s$, $bound$ - 1, $bound$) $\geq bound$ return $beta$;
        $bound \leftarrow$ round($alpha$ - $b$ - $t$ * $sigma$) / $a$);
        if (AlphaBetaPC($n$, $s$, $bound$, $bound$ + 1) $\leq bound$ return $alpha$;
    //

    else if $n$ = MAXNODE then
        $g \leftarrow -\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g < beta$ and $c \neq$ NOCHILD do
            $g \leftarrow$ max($g$, AlphaBetaPC($c$, $depth$ - 1, $alpha$, $beta$));
            $alpha \leftarrow$ max($alpha$, $g$);
            $c \leftarrow$ nextbrother($c$);
    else /* n is a min node */
        $g \leftarrow +\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g > alpha$ and $c \neq$ NOCHILD do
            $g \leftarrow$ min($g$, AlphaBetaPC($c$, $depth$ - 1, $alpha$, $beta$));
            $beta \leftarrow$ min($beta$, $g$);
            $c \leftarrow$ nextbrother($c$);
    return $g$;
```

Figure 1.7: The ProbCut enhanced AlphaBeta Algorithm.

if and only if $a \times v_s + b$, which is an unbiased estimator for v_d, lies outside of $[\alpha - t\sigma, \beta + t\sigma]$ where t is an adjustable confidence parameter. The pseudo code of ProbCut heuristic is shown in figure 1.7.

ProbCut was first implemented in its inventor's strong Othello program, LOGISTELLO which defeated the human world champion in 1997 [6]. With the help of ProbCut, the enhanced version beats original LOGISTELLO with a winning percentage of 74 percent. Later the inventor refined it to Multi-ProbCut (MPC) (figure 1.8) and EndCut. MPC allows pruning at different search depth together while EndCut allows a smooth transition from heuristic middle-game to exact endgame search that is able to find best moves in a limited time more often than the classic approach [7, 8]. The winning percentage of MPC against ProbCut

```
function AlphaBetaMPC($n$, $depth$, $alpha$, $beta$) $\rightarrow g$
    if $n$ = LEAFNODE then return eval($n$);

    //MultiProbCut heuristic:
    else if $depth \leq d$ then
        load $a$, $b$, $sigma$, $game\_stage$;
        for $i$ from 0 to NUMBER_OF_TRY do
            $bound \leftarrow$ round($beta - b + t * sigma$) / $a$);
            if (AlphaBetaMPC($n$, $s$, $bound - 1$, $bound$) $\geq bound$ return $beta$;
            $bound \leftarrow$ round($alpha - b - t * sigma$) / $a$);
            if (AlphaBetaMPC($n$, $s$, $bound$, $bound + 1$) $\leq bound$ return $alpha$;
    //

    else if $n$ = MAXNODE then
        $g \leftarrow -\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g < beta$ and $c \neq$ NOCHILD do
            $g \leftarrow$ max($g$, AlphaBetaMPC($c$, $depth - 1$, $alpha$, $beta$));
            $alpha \leftarrow$ max($alpha$, $g$);
            $c \leftarrow$ nextbrother($c$);
    else /* n is a min node */
        $g \leftarrow +\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g > alpha$ and $c \neq$ NOCHILD do
            $g \leftarrow$ min($g$, AlphaBetaMPC($c$, $depth - 1$, $alpha$, $beta$));
            $beta \leftarrow$ min($beta$, $g$);
            $c \leftarrow$ nextbrother($c$);
    return $g$;
```

Figure 1.8: The MultiProbCut enhanced AlphaBeta Algorithm.

enhanced version of LOGISTELLO is 72 percent. The result showed that MPC is even better.

However, the major deficiency of ProbCut heuristic is that it can only apply to Alpha-Beta algorithms but not its enhancement variants algorithms such as NegaScout or MTD(f).The reason is simply that these algorithms usually make use of null-window search technique. As null-window is already the minimal window, any boundary tests apply to it must fail. The minimax value of a node cannot lie inside the current search window (that is, the null-window). The tests must either return failing high or failing low. We cannot gain any additional cut-offs but have to pay for the cost of boundary tests.

Literatures [32][31][30] have already showed that MTD(f) and NegaScout out-perform Alpha-Beta. The search speed of Alpha-Beta is slow when compared with the state-of-the-art minimax search algorithm MTD(f). The reduced search effort by ProbCut to Alpha-Beta is not as significant as that of MTD(f) or NegaScout. That is, the search time of MTD(f) and NegaScout algorithms are even shorter than ProbCut enhanced Alpha-Beta algorithm. In this case, there is no gain to use ProbCut unless we can find way to apply ProbCut to MTD(f) or NegaScout. We will discuss this issue in the next chapter.

Others

Moriarty and Miikkulainen showed that evolutionary neural networks can be used to perform selective search as well. Their focus networks [25] were evolved using genetic algorithms to direct a minimax search away from poor information. At each state in the search, the focus networks determines which moves look the most promising and a subset of possible moves are explored. They tested their focus networks in the game of Othello and result showed that the focus searches are able to defeat full-width searches while examining vastly fewer positions. Aritificial evolution provides a promising paradgm for developing better game-playing programs.

1.3 Construction of Evaluation Function

Even there is a fast minimax search, without a reasonably good evaluation function, the program could not make any intelligent move. Therefore, the Board evaluation function is the most important component of an intelligent program.

The model described by Shannon is a linear polynomial of weighted board features [39]. This is widely used in many successful intelligent programs. Usually evaluated score are assumed to be integer-valued. However, one may choose real number ranged from 0 to 1 for evaluation to represent the probability of winning.

Devising a reasonably good board evaluation function is, in general, not an easy problem. Consequently, substantial work has been done on devising methods for automatically generating such functions. The earliest publication that actively employs machine learning was presented in 1959 by Samuel [35]. Samuel developed a checkers program that tried to find "the highest point in multidimensional scoring space" by using two players. The results from Samuel's experiment were impressive. In 1988, Sutton developed Samuel's ideas further and formulated methods for Temporal Difference Learning (TDL) [43]. Many researchers have since applied TDL to games. One of the most successful of these is Tesauro's backgammon program which achieved master-level status [44]. In 1993, Lorenz derived a framework for applying genetic algorithms to game evaluation function learning [20]. Besides, Ferrer and Martin [13], Sun and Wu [42] and Chisholm and Bradbeer [11] all successfully constructed good evaluation function for their Senet, Othello and Draughts programs by genetic algorithm respectively. Neural-Network was found to be useful in evolving intelligent game programs too. Chellapilla and Fogel made a great success in their checkers program [10]. Recently, evolutionary algorithms were found to be successful in optimizing the board evaluation function. One example is Kendall and Whitwell [18]. They developed an evolutionary approach to tune a Chess evaluation function.

1.4 Contribution of the Thesis

This thesis aims at developing various heuristic in tree searching for two-person non-random perfect information zero-sum game. According to Jackson's description, non-random means that the allocation and positioning of resources in the game is purely deterministic. The term perfect information indicates that both player have complete knowledge regarding the disposition of both player's resources while zero-sum means that any potential gain to one player will be reflected as a corresponding loss to the other player. Typical examples are Chess, Othello, Checkers and so on. In particular, the contributions of this research can be summerized as follows:

- *GPC-AB: A generalised version of ProbCut*

  The efficient ProbCut selective extension is reformulated, making the heuristic practical to Alpha-Beta enhancement algorithms. ProbCut can be expressed intuitively as a null-window call to Alpha-Beta from parent node, yielding a new formulation called GPC-AB.

- *GPC: A selective search framework based on ProbCut*

  Inspired by the GPC-AB reformulation, a new framework for selective minimax search is introduced. It is based on the null-window Alpha-Beta search. We present a simple framework of GPC that make calls to null-window Alpha-Beta search. Search results from previous passes are used to determine whether a deep search would be needed. The instances of this framework are readily incorporated into existing game-playing programs.

- *FGPC: A framework that can out-perform MTD(f) in time efficiency*

  In the GPC framework the essential part of the search is formed by a null-window search. Based on this boundary testing technique, we introduce a new framework called FGPC. Value returned from null-window call is used to determine when to stop the search of current node as well as its parent node. Using our new framework, we are able to compare the performance of FGPC to a number of well-known minimax search algorithms. A high performance game-playing program was used to ensure the generality and reliability of the outcome. The results of these experiments were quite surprising, FGPC is comparable with full-width minimax search in performance and is out-performing in time efficiency.

- *Node-cutting heuristic: An effective pruning technique that is superior than full-width minimax search*

  We formulate a heuristic, node-cutting. Node-cutting reduces the branching factor by the correlation of move ordering searched at different levels. Our experiments show that node-cutting heuristic can guide the search to a more aggressive approach and take advantage of possible mistakes by the opponent.

- *An integrated strategy: Efficiently use of allocated time*
  We introduce a technique to take better advantage of available time. It reduces the search tree size but maintaining a high rate of hitting the correct minimax value.

1.5 Structure of the Thesis

The organization of the thesis is as follows. In Chapter 2, a probabilistic forward pruning framework is described. This is a generalised selective search extension. In Chapter 3, a fast probabilistic forward pruning framework is proposed. The results of applying our heuristics to games are described. In Chapter 4, node-cutting heuristic is formulated. In Chapter 5, an integrated strategy is constructed. Finally, we will draw our conclusions and discuss the future works in the last chapter.

Chapter 2

The Probabilistic Forward Pruning Framework

2.1 Introduction

It has long been known that the Alpha-Beta algorithm is an inefficient searching method that it searches all nodes to the same depth. No matter how bad a move is, it gets searched as deep as the most promising move [39]. Alpha-Beta algorithm uses the backed-up return values propagated from leaf nodes for the cut-off decisions. This kind of pruning method is named as backward pruning. Backward pruning will probably make sure that most of the nodes in the subtree of the bad move get pruned, but a more selective search strategy could help to make sure that really bad moves are not considered at all. In contrast to backward pruning, these strategies are called forward pruning.

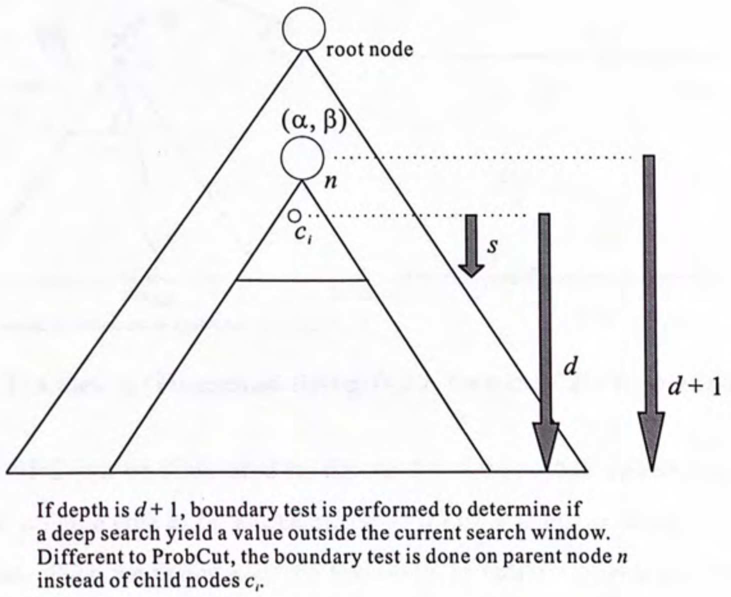
The previous chapter showed how the ProbCut heuristic is used in Alpha-Beta algorithm. This chapter generalizes the heuristic further. In the next section, we present a reformulation of ProbCut. The reformulation is based on the Alpha-Beta procedure. It examines the same leaf nodes in the same order as ProbCut. It is called Generalized Probabilistic Forward Cuts Heuristic (GPC). In section two, we will generalize the ideas behind GPC into a new framework that elegantly ties together a number of algorithms that are perceived to be dissimilar. The last section summarizes our idea.

2.2 The Generalized Probabilistic Forward Cuts Heuristic

Recall from previous chapter that the ProbCut selective search heuristics permits pruning of subtrees that are unlikely to affect the minimax value and uses the time saved for analysis of probably more relevant variations. This approach is based of the fact that values returned by minimax searches of different depths are highly correlated, provided that a reasonably good evaluation function and, if necessary, a quiescence search is used. In this case, a shallow search result is a good predictor for the deep minimax value. After computing the shallow search result, the search is terminated in the current position if and only if the unbiased estimator for the deep minimax value lies outside the search window. The core idea is the boundary tests.

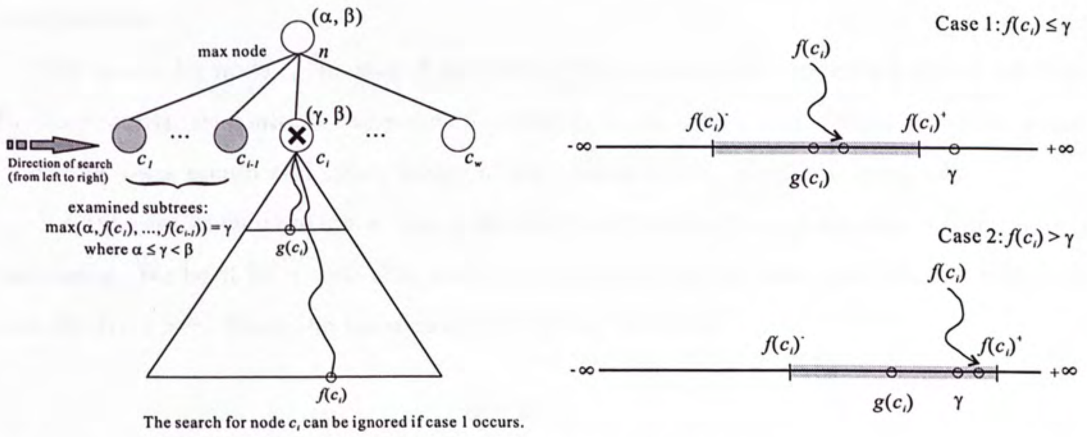
However, the major deficiency of ProbCut heuristic is that it can only apply to Alpha-Beta algorithms but not its enhancement variants algorithms such as NegaScout or MTD(f).The reason is simply that these algorithms usually make use of null-window search technique. As null-window is already the minimal window, any boundary tests apply to it must fail. The minimax value of a node cannot lie inside the current search window (that is, the null-window). The tests must either return failing high or failing low. We cannot gain any additional cut-offs but have to pay for the cost of boundary tests.

Literatures [32][31][30] have already showed that MTD(f) and NegaScout out-perform Alpha-Beta. The search speed of Alpha-Beta is slow when compared with the state-of-the-art minimax search algorithm MTD(f). The reduced search effort by ProbCut to Alpha-Beta is not as significant as that of MTD(f) or NegaScout. That is, the search time of MTD(f) and NegaScout algorithms are even shorter than ProbCut enhanced Alpha-Beta algorithm. In this case, there is no gain to use ProbCut unless we can find way to apply ProbCut to MTD(f) or NegaScout. Nevertheless, by reformulating ProbCut to perform boundary test for child nodes in parent level, selective search apply to Alpha-Beta enhancement variants algorithm that use null-window search technique is possible.



If depth is $d+1$, boundary test is performed to determine if a deep search yield a value outside the current search window. Different to ProbCut, the boundary test is done on parent node n instead of child nodes c_i.

Figure 2.1: The idea of Generalized Probabilistic Forward Cuts Heuristic.



The search for node c_i can be ignored if case 1 occurs.

Figure 2.2: The idea of Generalized Probabilistic Forward Cuts Heuristic (Max node).

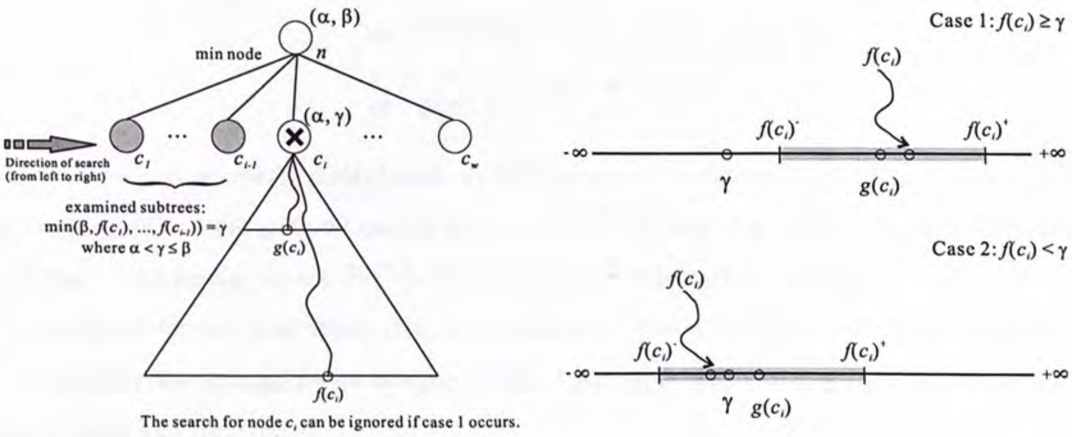

(α, β)
min node
n
(α, γ)
Direction of search (from left to right)
c_1
c_{i-1}
c_i
c_w
examined subtrees: min(β, f(c_1), ..., f(c_{i-1})) = γ where α < γ ≤ β
g(c_i)
f(c_i)
The search for node c_i can be ignored if case 1 occurs.
Case 1: f(c_i) ≥ γ
Case 2: f(c_i) < γ
f(c_i)^-
f(c_i)^+
-∞
+∞
γ
g(c_i)

Figure 2.3: The idea of Generalized Probabilistic Forward Cuts Heuristic (Min node).

The idea of GPC can be illustrated by figures 2.1, 2.2 and 2.3. Let us first define n as an arbitary node of a game tree to be searched. Node n has w children where w is the branching factor of the tree. Then we define $g(c_i)$ be the result of shallow search and $f(c_i)$ be the deep minimax value of node c_i.

A simple way to express the relationship between $g(c_i)$ and $f(c_i)$ is a linear model of the form $f(c_i) = a \times g(c_i) + b + e$ where a, b are real constants and e is a normally distributed error variable having mean 0 and variance σ^2 . These parameters are estimated by linear regression applied to a large number of training pairs $(g(p_i),\ f(p_i))$, where p_i is input of different board configurations.

The search for node c_i can stop if and only if $f(c_i)$ lies outside the current search window. In other words, we continue the search for node c_i if and only if $\alpha < f(c_i) < \beta$ where α and β are the lower bound and upper bound of the current search window respectively.

For the case when node n is a max node, the lower bound of search window is continuously increasing. We let it be γ such that $\alpha \leq \gamma < \beta$. In this case, we have to perform a boundary test for $f(c_i) > \gamma$. Using the same idea as ProbCut, we have:

$$\begin{aligned} & f(c_i) > \gamma \\ \Leftrightarrow\ & a \times g(c_i) + b + e > \gamma \\ \Leftrightarrow\ & \frac{a \times g(c_i) + b - \gamma}{\sigma} > -\frac{e}{\sigma} \end{aligned}$$

$$\begin{aligned}
\Leftrightarrow \quad & \frac{a \times g(c_i) + b - \gamma}{\sigma} > \Phi^{-1}(p) \\
\Leftrightarrow \quad & g(c_i) > \frac{\gamma - b + \Phi^{-1}(p)}{a} \qquad (2.1)
\end{aligned}$$

Since $-\frac{e}{\sigma}$ is normally distrubuted with mean 0 and variance 1, $f(c_i) > \gamma$ holds with probability of at least p if and only if $g(c_i) > \frac{\gamma-b+\Phi^{-1}(p)}{a}$ where Φ is the Normal distribution function. As a result, we use $\frac{\gamma-b+\Phi^{-1}(p)}{a}$ as the value of boundary test for $g(c_i)$.

Similarly, for the case when node n is a min node the upper bound of search window is continuously decreasing. We let the upper bound be γ such that $\alpha < \gamma \leq \beta$. Then we perform a boundary test for $f(c_i) < \gamma$.

$$\begin{aligned}
& f(c_i) < \gamma \\
\Leftrightarrow \quad & a \times g(c_i) + b + e < \gamma \\
\Leftrightarrow \quad & \frac{a \times g(c_i) + b - \gamma}{\sigma} < -\frac{e}{\sigma} \\
\Leftrightarrow \quad & \frac{a \times g(c_i) + b - \gamma}{\sigma} < -\Phi^{-1}(p) \\
\Leftrightarrow \quad & g(c_i) < \frac{\gamma - b - \Phi^{-1}(p)}{a} \qquad (2.2)
\end{aligned}$$

$f(c_i) < \gamma$ holds with probability of at least p if and only if $g(c_i) < \frac{\gamma-b-\Phi^{-1}(p)}{a}$. As a result, we use $\frac{\gamma-b-\Phi^{-1}(p)}{a}$ as the value of boundary test for $g(c_i)$.

Appendix A contains a detailed example of how a tree is searched by GPC, ProbCut and relevant algorithms.

2.3 The GPC Framework

GPC is a probabilistic forward pruning framework that generalized from the idea of ProbCut. GPC is easily applicable to other Alpha-Beta enhancement algorithms. In this section, we will illustrate how GPC is used together with Alpha-Beta and some well-known Alpha-Beta enhancement algorithms.

2.3.1 The Alpha-Beta Algorithm

Given an arbitary node, the search is terminated in the current position if and only if its minimax value already lies outside the current search window. The search window is passed from its parent. If the parent is a max node, it will continue increasing the lower bound of the search window in trasversing its subtrees, until cut-off occurs (that is the lower bound is greater than the upper bound). For the same reason, it will continus lowering the upper bound of the search window in trasversing its subtree until cut-off occurs if the parent is a min node. As we can see, we can perform the boundary tests for subtrees in the parent node instead of performing the boundary tests in child nodes. This can be done by placing the boundary test code before the recursive call of Alpha-Beta function. The pseudo code is shown in figure 2.4.

GPC-AB is equivalent to ProbCut. GPC-AB examines the same leaf nodes in the same order as ProbCut. They produce the same number of cut-offs. They not only make the same decision, but also obtain the same minimax value if given the same tree to them. Though the running time and complexity for both heuristic make no significant difference, GPC-AB give us a hints for us to apply selective search heuristic in Alpha-Beta enhancment algorithms which usually involve techniques of null-window search. Null-window search is not compatible with ProbCut since the search window is already the minimal window. We can never find a subtree that its minimax value does not lie outside the minimal search window. However, with the help of the reformulation of ProbCut, selective search heuristic to Alpha-Beta variants algorithms are applicable.

2.3.2 The NegaScout Algorithm

As described in previous chapter, NegaScout uses a wide search window for the first child, and a null-window for the other children. ProCut can be applied to the first child only, but cannot be applied to other children. As a max node the first node should be the highest, if one of the null-window searches returns a bound that is greater, then that child have to re-search with a wide window to determine its value. For a min node the first node should remain the lowest. If the null-window searches show one of the brothers to be lower, then re-search is needed. In both case, the search window for re-search use the result of null-window as the bound such that to reduce the search window for re-search. However, the previous null-window search is just a boundary test. The returned bound reduces the search window. It is not likely that the reduced search window would produce a failing low or failing high. As a result, the boundary test for ProbCut become useless. But if we put the tests before the recursive function call, we

```
function GPC-AlphaBeta($n$, $depth$, $alpha$, $beta$) $\rightarrow g$
    if $n$ = LEAFNODE then return eval($n$);
    else if $n$ = MAXNODE then
        $g \leftarrow -\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g < beta$ and $c \neq$ NOCHILD do

            //GPC heuristic:
            if $level = d+1$ then
                $ub \leftarrow g$;
                $gc \leftarrow$ round(($ub - b - t * sigma$) / $a$);
                $gc' \leftarrow$ GPC-AlphaBeta($c$, $s$, $gc$, $gc + 1$);
                if $gc' < gc$ then
                    $c \leftarrow$ nextbrother($c$);
                    continue;

            $g \leftarrow$ max($g$, GPC-AlphaBeta($c$, $depth$ - 1, $alpha$, $beta$));
            $alpha \leftarrow$ max($alpha$, $g$);
            $c \leftarrow$ nextbrother($c$);
    else /* n is a min node */
        $g \leftarrow +\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g > alpha$ and $c \neq$ NOCHILD do

            //GPC heuristic:
            if $level = d+1$ then
                $lb \leftarrow g$;
                $gc \leftarrow$ round(($lb - b + t * sigma$) / $a$);
                $gc' \leftarrow$ GPC-AlphaBeta($c$, $s$, $gc - 1$, $gc$);
                if $gc' > gc$ then
                    $c \leftarrow$ nextbrother($c$);
                    continue;

            $g \leftarrow$ min($g$, GPC-AlphaBeta($c$, $depth$ - 1, $alpha$, $beta$));
            $beta \leftarrow$ min($beta$, $g$);
            $c \leftarrow$ nextbrother($c$);
    return $g$;
```

Figure 2.4: The Alpha-Beta Algorithm enhanced with Generalized Probabilistic Forward Cuts Heuristic.

can obtain the same result as GPC-AB. The pseudo code is for GPC-NS is shown in figure 2.5.

2.3.3 The Memory-enhanced Test Algorithm

The previous chapter described how MTD(f) merely uses the techniques of null-window search to perform a minimax search. The transformation of ProbCut to MTD(f) is not trivial. The mechanism of GPC is to perform the boundary test for child nodes in parent level. Like Alpha-Beta algorithm, though the search window is the minimal, MTD(f) have a loop that continuously updating the bound, either the lower bound or the upper bound, until cut-off occurs. As shown in figure 2.6, the variable g and γ are the bounds. Using the same method, we can reformulate the function to GPC-MTD(f).

2.4 Summary

In this chapter, we reforumulated the idea of ProbCut, an effective and successful selective search heuristic, such that it is not only applicable to Alpha-Beta algorithm but also applicable to other enhanced Alpha-Beta minimax search algorithms. Generalised ProbCut has the same behaviour as ProbCut. They both make the same move and obtain the same minimax value. They also produce the same amount of cut-offs. The generalised version used the core idea of ProbCut, the boundary tests, but is more generic such that it can be easily applied to other Alpha-Beta variants search algorithms.

```
function GPC-NegaScout($n$, $depth$, $alpha$, $beta$) $\rightarrow g$
    if $n$ = LEAFNODE then return eval($n$);
    $c \leftarrow$ firstchild($n$);
    $gamma \leftarrow$ GPC-NegaScout($c$, $level$ - 1, $alpha$, $beta$);
    $c \leftarrow$ nextbrother($c$);
    if $c$ = MAXNODE
        while $gamma < beta$ and $c \neq$ NOCHILD do

            //GPC heuristic:
            if $depth = d+1$ then
                $ub \leftarrow gamma$;
                $gc \leftarrow$ round(($ub - b - t * sigma$) / $a$);
                $gc' \leftarrow$ GPC-NegaScout($c$, $s$, $gc$, $gc$ + 1);
                if $gc' < gc$ then
                    $c \leftarrow$ nextbrother($c$);
                    continue;

            $alpha \leftarrow$ max($gamma$, $alpha$);
            $g \leftarrow$ GPC-NegaScout($c$, $level$ - 1, $alpha$, $alpha$ + 1);
            if $g > alpha$ and $g < beta$ then
                $g \leftarrow$ GPC-NegaScout($c$, $level$ - 1, $g$, $beta$);
            $gamma \leftarrow$ max($g$, $gamma$);
            $c \leftarrow$ nextbrother($c$);
    else /* $c$ is min node */
        while $gamma > alpha$ and $c \neq$ NOCHILD do

            //GPC heuristic:
            if $depth = d+1$ then
                $lb \leftarrow gamma$;
                $gc \leftarrow$ round(($lb - b + t * sigma$) / $a$);
                $gc' \leftarrow$ GPC-NegaScout($c$, $s$, $gc$ - 1, $gc$);
                if $gc' > gc$ then
                    $c \leftarrow$ nextbrother($c$);
                    continue;

            $beta \leftarrow$ min($gamma$, $beta$);
            $g \leftarrow$ GPC-NegaScout($c$, $level$ - 1, $beta$ - 1, $beta$);
            if $g > alpha$ and $g < beta$ then
                $g \leftarrow$ GPC-NegaScout($c$, $level$ - 1, $alpha$, $g$);
            $gamma \leftarrow$ min($g$, $gamma$);
            $c \leftarrow$ nextbrother($c$);
    return $gamma$;
```

Figure 2.5: The NegaScout Algorithm enhanced with Generalized Probabilistic Forward Cuts Heuristic.

```
function GPC-MT($n$, $gamma$, $depth$) $\rightarrow g$
        if $depth = 0$ then /* leaf node */
                retrieve($n$)
                if $n.lowerbound = -\infty$ and $n.upperbound = +\infty$ then
                        $g \leftarrow$ evaluate($n$);
                else if n.upperbound $= +\infty$ then
                        $g = n.lowerbound$;
                else
                        $g = n.upperbound$;
        else if $n$ = MAXNODE then
                $g \leftarrow -\infty$;
                $c \leftarrow$ firstchild($n$);
                /* $g \geq gamma$ causes a beta cutoff ($beta = gamma$) */
                while $g < gamma$ and $c \neq$ NOCHILD do
                        retrieve($c$);
                        if $c.upperbound \geq gamma$ then
                                //GPC heuristic:
                                if $depth = d+1$ then
                                        $ub \leftarrow g$;
                                        $gc \leftarrow$ round(($ub - b - t * sigma$) / $a$);
                                        $gc' \leftarrow$ GPC-NegaScout($c, s, gc, gc + 1$);
                                        if $gc' < gc$ then
                                                $c \leftarrow$ nextbrother($c$);
                                                continue;
                                $g' \leftarrow$ MT($c$, $gamma$, $d - 1$);
                        else
                                $g' \leftarrow c.upperbound$;
                        $g \leftarrow \max(g, g')$;
                        $c \leftarrow$ nextbrother($c$);
        else /* $n$ is a MINNODE */
                $g \leftarrow +\infty$;
                $c \leftarrow$ firstchild($n$);
                /* $g < gamma$ causes an alpha cutoff ($alpha = gamma - 1$) */
                while $g \geq gamma$ and $c \neq$ NOCHILD do
                        retrieve($c$);
                        if $c.lowerbound < gamma$ then
                                //GPC heuristic:
                                if $depth = d+1$ then
                                        $lb \leftarrow g$;
                                        $gc \leftarrow$ round(($lb - b + t * sigma$) / $a$);
                                        $gc' \leftarrow$ GPC-NegaScout($c, s, gc - 1, gc$);
                                        if $gc' > gc$ then
                                                $c \leftarrow$ nextbrother($c$);
                                                continue;
                                $g' \leftarrow$ MT($c$, $gamma$, $d - 1$);
                        else
                                $g' \leftarrow c.lowerbound$;
                        $g \leftarrow \min(g, g')$;
                        $c \leftarrow$ nextbrother($c$);
        /* Traditional transposition table storing of bounds */
        if $g \geq gamma$ then          /* Fail high result implies a lower bound */
                $n.lowerbound \leftarrow g$;
                store $n.lowerbound$;
        else                            /* Fail low result implies an upper bound */
                $n.upperbound \leftarrow g$;
                store $n.upperbound$;
return $g$;
```

Figure 2.6: The MTD Algorithm enhanced with Generalized Probabilistic Forward Cuts Heuristic.

Chapter 3

The Fast Probabilistic Forward Pruning Framework

3.1 Introduction

The previous chapter showed how the probabilistic forward pruning framework, GPC, is used in Alpha-Beta and its enhancement algorithms. GPC allows selective search to be applicable to those algorithms. This chapter takes the idea further. In the next section, the enhanced forward pruning framework is described. This is a framework that can further reduce the search effort significantly. The third section will show the result of simulations. The last section summarizes our idea.

3.2 The Fast GPC Heuristic

GPC is a simple but effective framework. It makes use of null-window search to perform boundary tests. Null-window search is powerful and fast. Though it cannot find out the minimax value, null-window would lead to early alpha or beta cut-off, returning a bound in either case. This useful information give us hints that what the minimax value look likes.

GPC starts with a null-window shallow search for each child, then uses the returned bound to determine whether to have full-depth search for them or not. For the case if we have a good

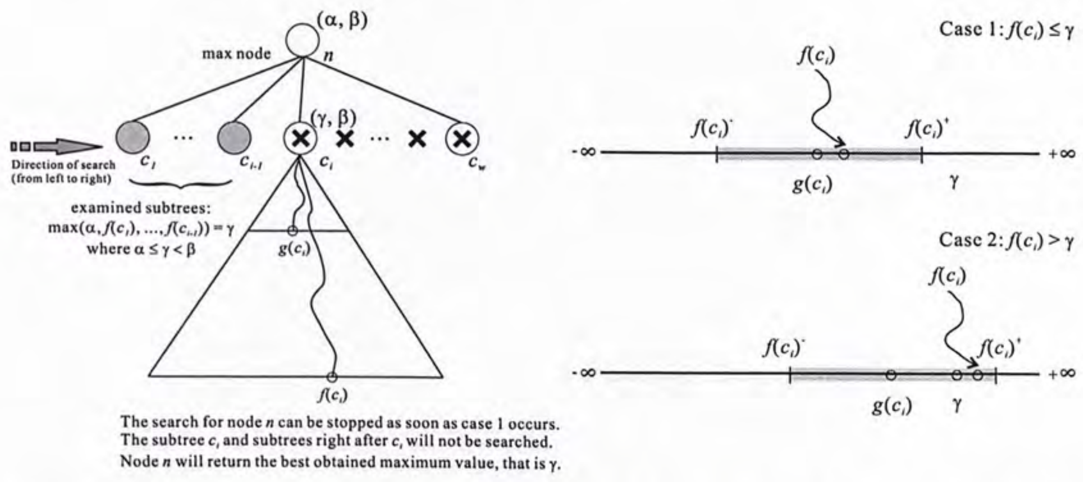

(α, β)
max node
n
(γ, β)
Direction of search (from left to right)
c1
ci-1
ci
cw
examined subtrees: max(α, f(c1), ..., f(ci-1)) = γ where α ≤ γ < β
g(ci)
f(ci)
The search for node n can be stopped as soon as case 1 occurs.
The subtree ci and subtrees right after ci will not be searched.
Node n will return the best obtained maximum value, that is γ.
Case 1: f(ci) ≤ γ
Case 2: f(ci) > γ
f(ci)⁻
f(ci)⁺
-∞
+∞
γ

Figure 3.1: The idea of Fast Generalized Probabilistic Forward Cuts Heuristic (Max node).

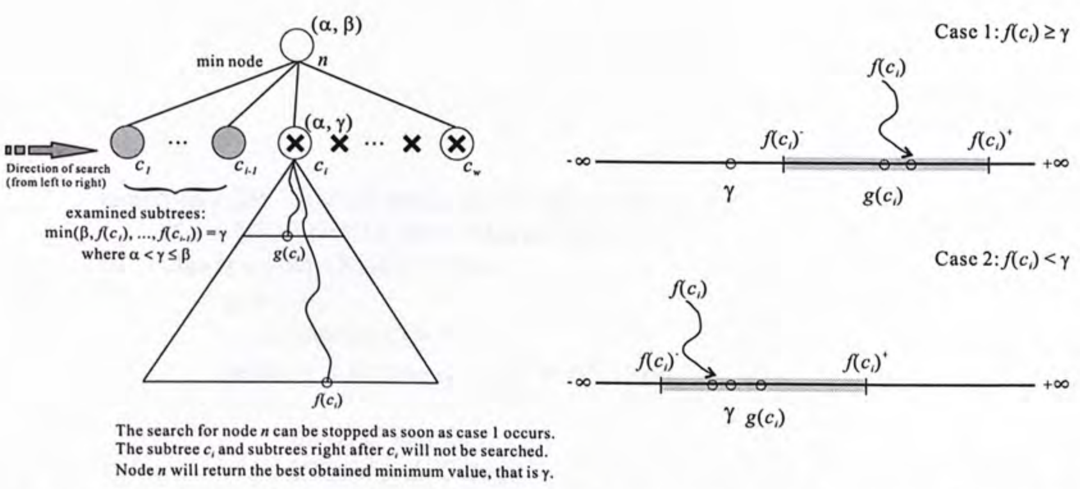
move ordering: the nodes are sorted in such a way that value of nodes are in monotonic order or best-first order, we can stop the search as soon as we found one child that its minimax value lies outside the search window. The current subtree and its sibling subtrees can be pruned in advance. This makes up the idea of Fast GPC.

The idea of FGPC is shown in figures 3.1 and 3.2. Let us first define n as an arbitary node of a game tree to be searched. Node n has w children where w is the branching factor of the tree. Then we define $g(c_i)$ be the result of shallow search and $f(c_i)$ be the result of the deep minimax value of node c_i.

A simple way to express the relationship between $g(c_i)$ and $f(c_i)$ is a linear model of the form $f(c_i) = a \times g(c_i) + b + e$ where a, b are real constants and e is a normally distributed error variable having mean 0 and variance σ^2 . These parameters are estimated by linear regression applied to a large number of training pairs $(g(p_i), f(p_i))$, where p_i is input of different board positions.

The search for node n can stop as soon as one of the child node, say c_i, its deep minimax value $f(c_i)$ lies outside the current search window. In other words, the search for node n will continue if and only if $\alpha < f(c_i) < \beta$ where α and β are the lower bound and upper bound of the current search window respectively and c_i is the current examining node.

FGPC obtains more cut-offs than GPC in general. As a result, FGPC reduces the search effort much more than that of GPC and thus turns out to be faster. The following subsections



(α, β)
min node
n
(α, γ)
Direction of search (from left to right)
c_1
c_{i-1}
c_i
c_w
examined subtrees: min(β, f(c1), ..., f(ci-1)) = γ where α < γ ≤ β
g(ci)
f(ci)
The search for node n can be stopped as soon as case 1 occurs. The subtree ci and subtrees right after ci will not be searched. Node n will return the best obtained minimum value, that is γ.
Case 1: f(ci) ≥ γ
f(ci)⁻
f(ci)⁺
-∞
+∞
γ
g(ci)
Case 2: f(ci) < γ

Figure 3.2: The idea of Fast Generalized Probabilistic Forward Cuts Heuristic (Min node).

will illustrate how FGPC is used together with Alpha-Beta and its enhanced algorithms.

3.2.1 The Alpha-Beta algorithm

This subsection describes the use of FGPC heuristic in the standard Alpha-Beta algorithm. The modified Alpha-Beta algorithm is shown in figure 3.3. When compare with the standard version, a null-window test is added inside the control loop of FGPC-AB. As describe in previous section, a null-window test consumes insignificant computation time. But the gain is, if the return value of the null-window test does imply that the testing node is beyond our desired score region, a series of nodes and subtrees will be pruned. The total time for running is reduced significantly. The time benefit we gain is much more than we pay. The set of experiments we conducted are described in the next section.

3.2.2 The NegaScout algorithm

This subsection describes the use of FGPC heuristic in NegaScout, a well-known Alpha-Beta Enhancement algorithm. The modified algorithm, FGPC-NS can be found in figure 3.4. The result of our conducted experiments are described in the next section.

```
function FGPC-AlphaBeta($n$, $depth$, $alpha$, $beta$) $\rightarrow g$
    if $n$ = LEAFNODE then return eval($n$);
    else if $n$ = MAXNODE then
        $g \leftarrow -\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g < beta$ and $c \neq$ NOCHILD do

            //FGPC heuristic:
            if $level = d+1$ then
                $ub \leftarrow g$;
                $gc \leftarrow$ round(($ub$ - $b$ - $t$ * $sigma$) / $a$);
                $gc' \leftarrow$ FGPC-AlphaBeta($c$, $s$, $gc$, $gc$ + 1);
                if $gc' < gc$ then return $g$;

            $g \leftarrow$ max($g$, FGPC-AlphaBeta($c$, $depth$ - 1, $alpha$, $beta$));
            $alpha \leftarrow$ max($alpha$, $g$);
            $c \leftarrow$ nextbrother($c$);
    else /* n is a min node */
        $g \leftarrow +\infty$;
        $c \leftarrow$ firstchild($n$);
        while $g > alpha$ and $c \neq$ NOCHILD do

            //FGPC heuristic:
            if $level = d+1$ then
                $lb \leftarrow g$;
                $gc \leftarrow$ round(($lb$ - $b$ + $t$ * $sigma$) / $a$);
                $gc' \leftarrow$ FGPC-AlphaBeta($c$, $s$, $gc$ - 1, $gc$);
                if $gc' > gc$ then return $g$;

            $g \leftarrow$ min($g$, FGPC-AlphaBeta($c$, $depth$ - 1, $alpha$, $beta$));
            $beta \leftarrow$ min($beta$, $g$);
            $c \leftarrow$ nextbrother($c$);
    return $g$;
```

Figure 3.3: The AlphaBeta Function enhanced with FGPC heuristic.

```
function FGPC-NegaScout(n, depth, alpha, beta) → g
    if n = LEAFNODE then return eval(n);
    c ← firstchild(n);
    gamma ← GPC-NegaScout(c, level - 1, alpha, beta);
    c ← nextbrother(c);
    if c = MAXNODE
        while gamma < beta and c ≠ NOCHILD do

            //FGPC heuristic:
            if depth = d+1 then
                ub ← gamma;
                gc ← round((ub - b - t * sigma) / a);
                gc' ← FGPC-NegaScout(c, s, gc, gc + 1);
                if gc' < gc then return gamma;

            alpha ← max(gamma, alpha);
            g ← FGPC-NegaScout(c, level - 1, alpha, alpha + 1);
            if g > alpha and g < beta then
                g ← FGPC-NegaScout(c, level - 1, g, beta);
            gamma ← max(g, gamma);
            c ← nextbrother(c);
    else  /* c is min node */
        while gamma > alpha and c ≠ NOCHILD do

            //FGPC heuristic:
            if depth = d+1 then
                lb ← gamma;
                gc ← round((lb - b + t * sigma) / a);
                gc' ← FGPC-NegaScout(c, s, gc - 1, gc);
                if gc' > gc then return gamma;

            beta ← min(gamma, beta);
            g ← FGPC-NegaScout(c, level - 1, beta - 1, beta);
            if g > alpha and g < beta then
                g ← FGPC-NegaScout(c, level - 1, alpha, g);
            gamma ← min(g, gamma);
            c ← nextbrother(c);
    return gamma;
```

Figure 3.4: The NegaScout Function enhanced with FGPC heuristic.

3.2.3 The Memory-enhanced Test algorithm

This subsection describes the use of FGPC heuristic in MTD(f), the state-of-the-art minimax search algorithm. Figure 3.5 shows the FGPC-MT function. The experimental results are shown in the next section.

3.3 Performance Evaluation

3.3.1 Determination of the Parameters

In this section, we present some experimental results in order to get a rough idea on the performance of the proposed framework. In our experiments, we mainly test the performance of standard Alpha-Beta, NegaScout and Memory-enhanced Test algorithm with and without FGPC. The comparison between GPC and FGPC is shown in later part.

The game of Chinese Checkers is used for testing purpose. Our test programs are based on Learner III, the Champion of ACM-HK Computer Chinese Checkers Competition 2002. Learner III use MTD(f) as the minimax search algorithm and look-ahead 3 levels for middle-game. No opening books are being used. We will have a detail description of Learner III on Appendix C.

In our experiments, our test programs only replace the search algorithm MTD(f) with Alpha-Beta and NegaScout (with and without GPC or FGPC) in the middle-game. The remaining parts of Learner III are remaining unchanged. We implement the algorithms in C language and our parameters are set as $s = 0, d = 2$ where s is the depth of shallow search while d is the depth of deep minimax search. All programs are run under linux platform on a P4 2.4GHz general purpose computer.

Figure 3.6 show 9669 evaluation pairs with linear approximation. The goodness of fit is visually obvious. The reason is that the evaluation function of Learner III is good enough and stable. By linear regression method, we find that the parameters needed for GPC and FGPC are $a = 0.9527, b = 24.1225, \sigma = 407.3506$. We choose the parameter $t = 1.3$ ($p = 90.32\%$) such that $\Phi^{-1}(0.9032) = 1.3$

NegaScout is the current algorithm of choice by most chess programmers and MTD(f)

```
function FGPC-MT($n$, $gamma$, $depth$) $\to g$
    if $depth = 0$ then /* leaf node */
        retrieve($n$)
        if $n.lowerbound = -\infty$ and $n.upperbound = +\infty$ then
            $g \leftarrow$ evaluate($n$);
        else if n.upperbound = $+\infty$ then
            $g = n.lowerbound$;
        else
            $g = n.upperbound$;
    else if $n$ = MAXNODE then
        $g \leftarrow -\infty$;
        $c \leftarrow$ firstchild($n$);
        /* $g \geq gamma$ causes a beta cutoff ($beta = gamma$) */
        while $g < gamma$ and $c \neq$ NOCHILD do
            retrieve($c$);
            if $c.upperbound \geq gamma$ then
                //FGPC heuristic:
                if $depth = d+1$ then
                    $ub \leftarrow g$;
                    $gc \leftarrow$ round(($ub - b - t * sigma) / a$);
                    $gc' \leftarrow$ GPC-NegaScout($c, s, gc, gc + 1$);
                    if $gc' < gc$ then return $g$;
                $g' \leftarrow$ MT($c, gamma, d - 1$);
            else
                $g' \leftarrow c.upperbound$;
            $g \leftarrow \max(g, g')$;
            $c \leftarrow$ nextbrother($c$);
    else /* $n$ is a MINNODE */
        $g \leftarrow +\infty$;
        $c \leftarrow$ firstchild($n$);
        /* $g < gamma$ causes an alpha cutoff ($alpha = gamma - 1$) */
        while $g \geq gamma$ and $c \neq$ NOCHILD do
            retrieve($c$);
            if $c.lowerbound < gamma$ then
                //FGPC heuristic:
                if $depth = d+1$ then
                    $lb \leftarrow g$;
                    $gc \leftarrow$ round(($lb - b + t * sigma) / a$);
                    $gc' \leftarrow$ GPC-NegaScout($c, s, gc - 1, gc$);
                    if $gc' > gc$ then return $g$;
                $g' \leftarrow$ MT($c, gamma, d - 1$);
            else
                $g' \leftarrow c.lowerbound$;
            $g \leftarrow \min(g, g')$;
            $c \leftarrow$ nextbrother($c$);
    /* Traditional transposition table storing of bounds */
    if $g \geq gamma$ then                /* Fail high result implies a lower bound */
        $n.lowerbound \leftarrow g$;
        store $n.lowerbound$;
    else                                  /* Fail low result implies an upper bound */
        $n.upperbound \leftarrow g$;
        store $n.upperbound$;
return $g$;
```

Figure 3.5: The MT Function enhanced with FGPC heuristic.

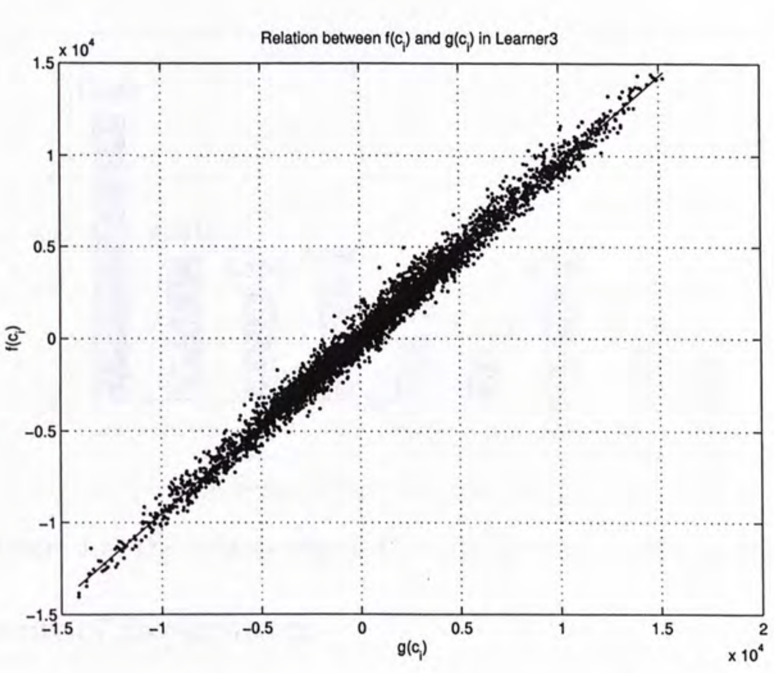


Figure 3.6: Relation between $f(c_i)$ and $g(c_i)$ in Learner III.

is the current the state-of-the-art minimax search algorithm, therefore we have chosen these algorithms as our baseline.

The algorithms we compared are denoted as:

- AB: *The standard AlphaBeta algorithm*
- GPC-AB: *The standard AlphaBeta algorithm enhanced with GPC heuristic*
- FGPC-AB: *The standard AlphaBeta algorithm enhanced with FGPC heuristic*
- NS: *The NegaScout algorithm*
- GPC-NS: *The NegaScout algorithm enhanced GPC heuristic*
- FGPC-NS: *The NegaScout algorithm enhanced FGPC heuristic*
- MTD: *The MTD(f) algorithm*
- GPC-MTD: *The MTD(f) algorithm enhanced GPC heuristic*
- FGPC-MTD: *The MTD(f) algorithm enhanced FGPC heuristic*

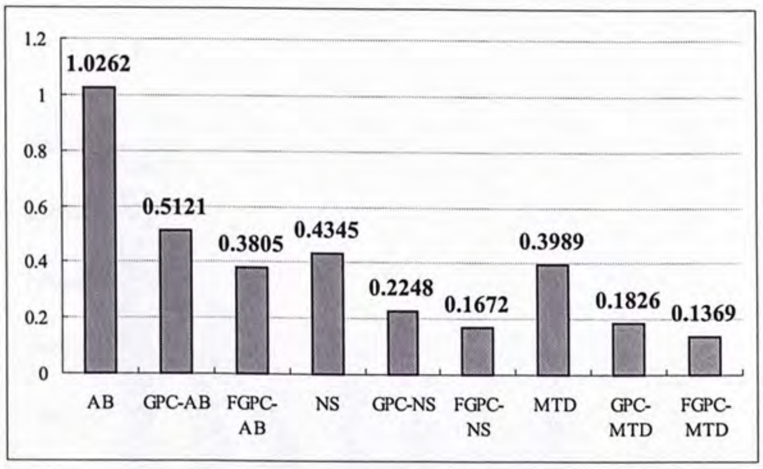


Figure 3.7: The Average Search Time Per Move for Each Algorithm.

3.3.2 Result of Experiments

One measure for comparing search algorithm performance is elapsed CPU time. However, any timing results are machine and implementation dependent. Another measure is the number of bottom positions (NBP), or sometimes called leaf nodes, examined. NBP has been used extensively in the literature [15], [9], [40] and [26]. Another measurement is the size of the tree. The node count (NC) measure counts all nodes in the tree where computaton occurs. This includes interior, leaf and any nodes in subtrees built as part of leaf node evaluation. This count has been shown to be strongly correlated with program running time by Schaeffer [36].

Since the execution time overhead of the enhancements is negligible, the computational cost being deminated by leaf node evaluation and the expansion of nodes. By comparing the measurement of NBP and NC as well as the elapsed CPU time, we can obtain a full picture for the performance of the proposed heuristics.

As shown from figure 3.7, the search time per move is reduced more than half after using GPC. The search effort is reduced even over 60% after FGPC. For example, FGPC-AB use 37% search effort of AB, FGPC-NS use 38% search effort of NS and FGPC-MTD use 34% search time of MTD only. FGPC-AB, FGPC-NS and FGPC-MTD have speed-up of 2.7 times, 2.6 times and 2.9 times respectively. The time reduced for FGPC is much more than that of

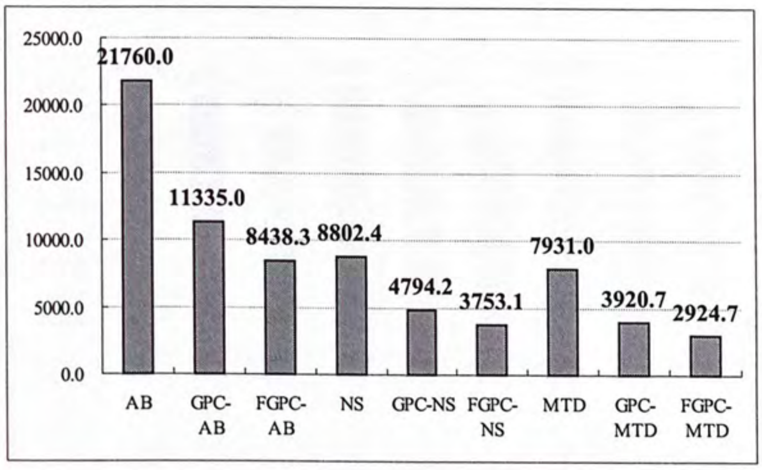


Figure 3.8: The Average Number of Node Visited Per Move for Each Algorithm.

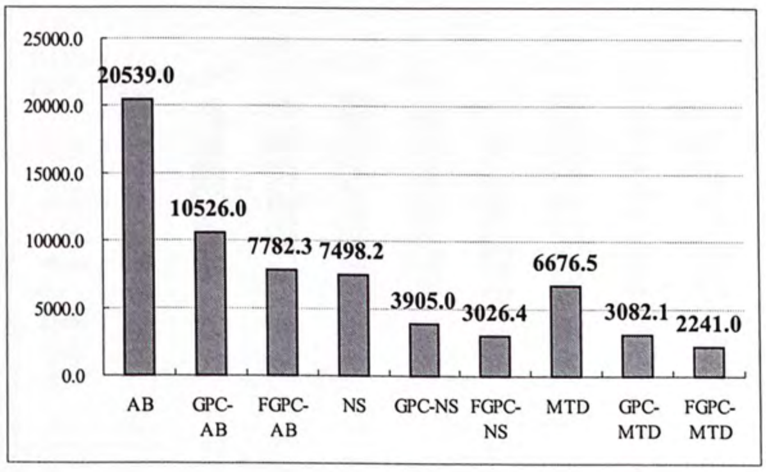


Figure 3.9: The Average Number of Bottom Position Visited Per Move for Each Algorithm.

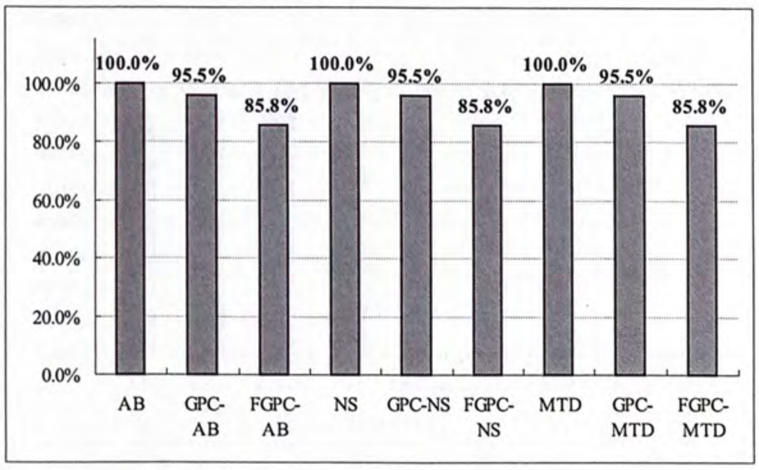


Figure 3.10: The Average Number of Same Move obtained Per Move for Each Algorithm.

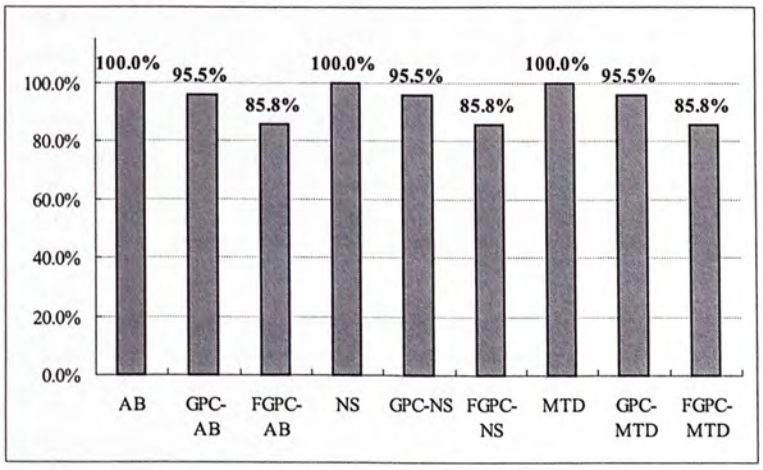


Figure 3.11: The Average Number of Same Value obtained Per Move for Each Algorithm.

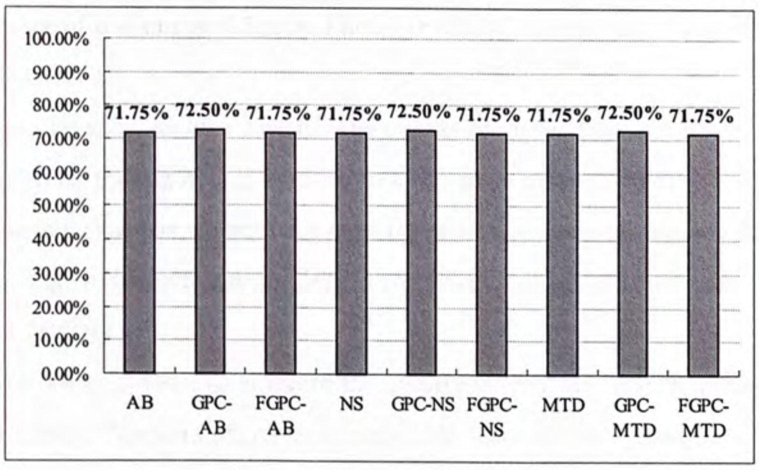


Figure 3.12: The Average Winning Percentage for Each Algorithm (Match against 100 different players).

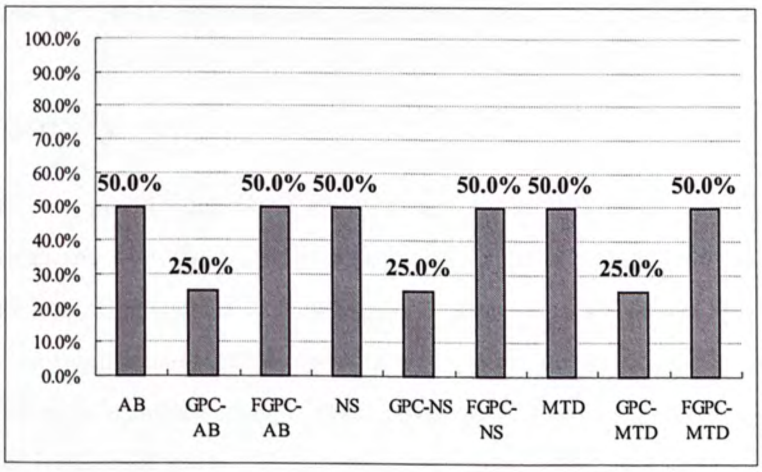


Figure 3.13: The Average Winning Percentage for Each Algorithm (Match against Learner III).

GPC.

The graph of NBP (figure 3.9) and NC (figure 3.8) consistent with the result of computation time. The number of bottom position and number of node visited are reduced with the same factor if FGPC is used.

Besides time efficiency, we also measure the quality of move. Figure 3.10 shows the percentage of cases in which the algorithm would select the same move as MTD(f) while figure 3.11 shows the frequency that the algorithm would reported the same minimax value as MTD(f). The high percentage indicated that both GPC and FGPC are good estimator and consist most search result of MTD(f).

Furthermore, we would like to measure the quality of play for each algorithm. We selected 100 different Chinese Checkers player programs that have similar strength as Learner III to have matches against our test programs. The result is shown in figure 3.12. The average winning percentage of algorithms using GPC and FGPC are closed to algorithms without using them. Apart from that, figure 3.13 shows the winning percentage of each algorithms match against Learner III. The winning percentage of MTD is 50%, which consistent with the fact that Learner III actually uses MTD(f) as the search algorithm. For algorithms enhanced with FGPC, their winnning percentage is higher than algorithms enhanced with GPC. As time is saved but performance can almost remain the same, FGPC is clearly a better choice.

3.4 Summary

In this chapter, we modify our GPC heuristic such that the search effort is significantly reduced. Experiments were conducted and showed that our idea outperformed previous successful heuristic and algorithms in time efficiency. Other results borne out by experiments are that the memory requirements of all algorithms are perfectly acceptable for typical tournament play, since only a small subset of the visited nodes has to be stored in memory. The instances of this framework are readily incorporated into existing game-playing programs. We believe that our selective search framework work efficiently and effectively.

Chapter 4

The Node-Cutting Heuristic

4.1 Introduction

The previous chapter showed how forward pruning techniques are used in tree searching. The idea of forward pruning strategies is, by reducing the number of moves to be considered, to reduce the computation time without lowering the quality of play. Apart from the heuristics discussed in previous chapters, cutting down the branching factor of the game tree is another workable idea. In the next section, we will discuss the effect of the quality of move ordering. Some properties are found. In section three, we generalize the idea to formulate our node-cutting heuristic. Some experiments are conducted and the results are shown in section four. The last section summarizes our proposed idea.

4.2 Move Ordering

Recalled from Alpha-Beta algorithm, it is essential that the best moves are to be searched first in order to maximize the number of cut-off occurs, resulting to minimize the search time. The other successful Alpha-Beta enhanced algorithms including NegaScout and MTD(f) also rely on the quality of move ordering. Though we obtain the same minimax value eventually, badly ordered game tree consumes more time for searching as the rate of having cut-off is very limited.

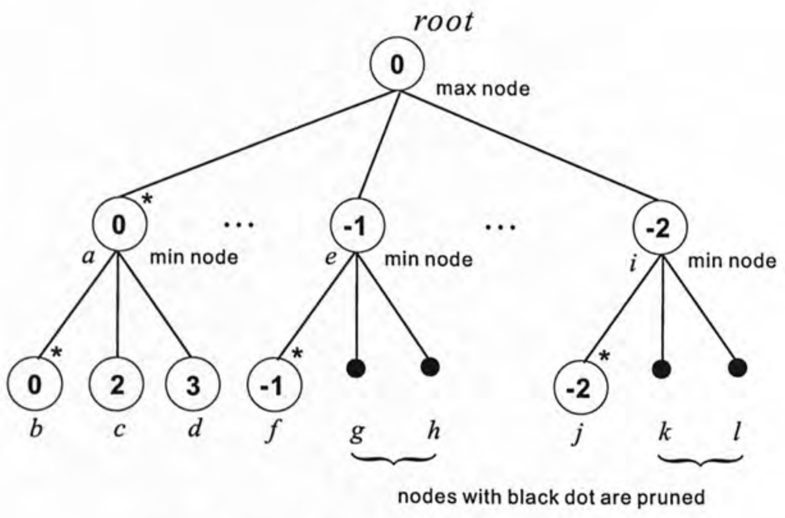


Figure 4.1: An Example of Early Pruning.

For a max position, the best move is the one whose minimax value is the highest. Analogously, the best move of a min position is the one with the lowest minimax value. In order for a cut-off to be occurred, perfect ordering is not necessary, provided that the best move appears at the front. The examples are shown in figures 4.1, 4.2 and 4.3. The best move of each node is marked by an asteria. It can be seen easily that the earlier the best move appears, the earlier the cut-off occurs. As a result, despite of constructing perfect move ordering, best-first ordering is an alternative simpler solution.

In practice, move ordering is done by a move ordering function which is an inexpensive estimatior of the board evaluation function. Similar to evaluation function, move ordering function is usually accomplished by game-depedent knowledge and heuristics. However, techniques such as iterative deepening [15] and history heuristic [37] have described how to achieve excellent move ordering. We can, therefore, obtain the function by these techniques.

4.2.1 Quality of Move Ordering

The quality of evaluation function can be fully reflected by the quality of play since it determines the game playing strength. The more accurate the evaluation function we have, the higher chance we would go into the win position. The measure of quality of move ordering

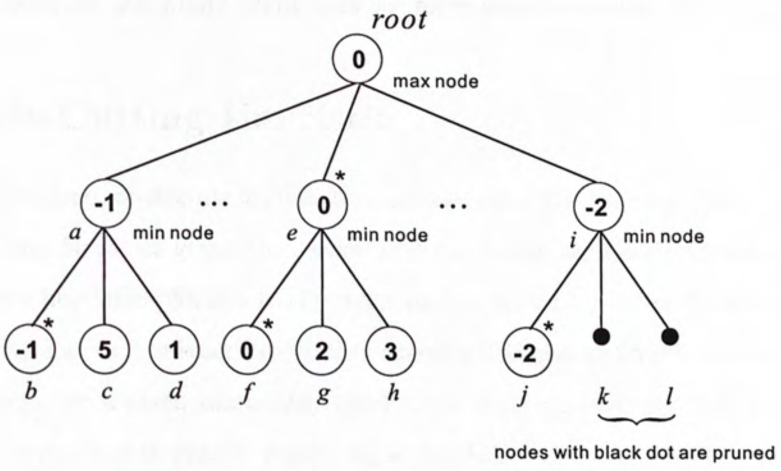


Figure 4.2: An Example of Late Pruning.

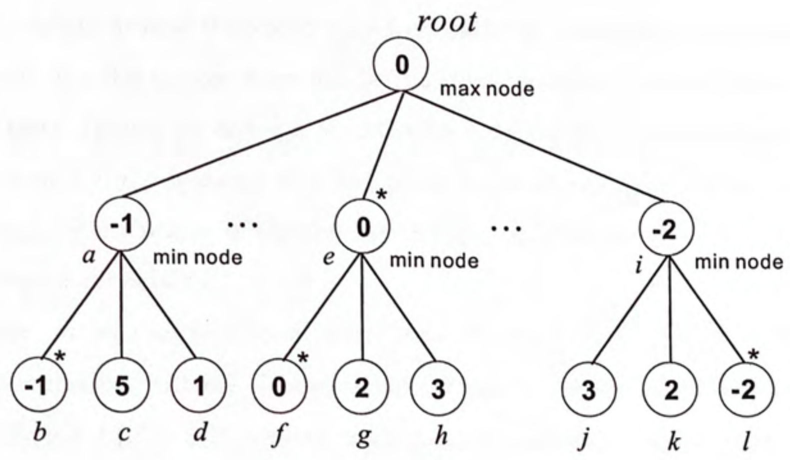


Figure 4.3: An Example of No Pruning.

becomes the measure of the position where the best move appears. The quality of move ordering directly affects the overall search speed. The more accurate the move ordering function we have, the more cut-offs would occur thus the more search speedup would gain.

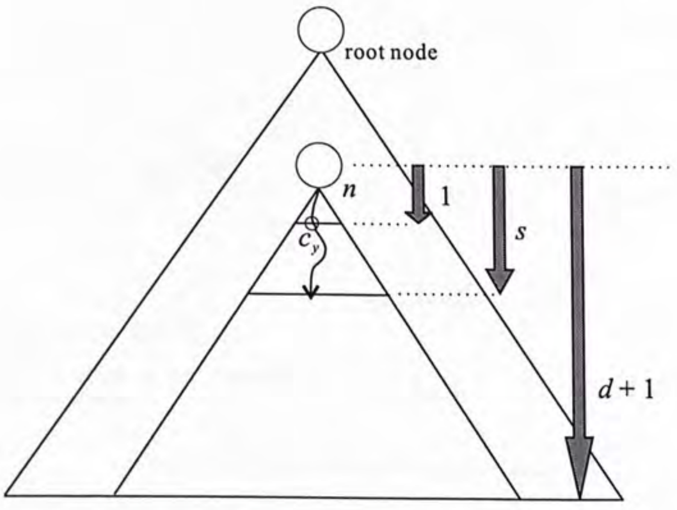
4.3 Node-Cutting Heuristic

One of the major time constraints in chess type games is that the running time is highly affected by the branching factor of game tree, even we have a fast search algorithm and successful forward pruning heuristic. We aim at, by reducing the branching factor by heuristic, ignoring the unpromising moves in advance to further speedup the overall search time.

We find that, for a given reasonably good move ordering function, the best move of an arbitrary board position is usually appearing at the front. The best child in this case is just the desired move. And, though the ordering positions of the desired move when searched at different look-ahead level are not always the same, they are highly correlated.

Let the distribution function $F_Y(x) = P(Y \leq y)$ characterize the random variable Y ranging from 0 to $w-1$ where w is the branching factor. The probability $p = P(Y = y)$ for node n is the probability of the event that the best move is the y-th branch. $F_Y(y)$ is estimated from practical data of Chinese Checkers program Learner III. The compiled statistics revealed that the static values around 0 occured very frequently. It means that the move ordering is done pretty well. For this reason, a normal distrubution could not fit these data well according to statistical tests. Hence, we define a distribution function $F_{Y1}(y)$ approximating our data as an estimate of $F_Y(y)$. It seems that the exact shape of the distribution function is not really important for the results of the simulation runs; only the values around 0 must have a significantly higher probability.

Analogously, we let the conditional distribution function $F_{X|Y=y}(x) = P(X \leq x|Y = y)$ characterize the random variable X ranging from 0 to $w-1$ where w is the branching factor. The probability $p = P(X = x|Y = y)$ for node n is the probability of the event that the best move is the x-th branch given that y-th branch is the best move obtained by a short depth minimax search. $F_{X|Y=y}(x)$ is, again estimated from practical data of Chinese Checkers program Learner III. The compiled statistics revealed that the static values around y occured



Y is the random variable ranging from 0 to w-1
where w is the branching factor.
c_y is the best move of node n computed by a short depth (s) search.

Figure 4.4: The idea of node-cutting.

very frequently. It means that the ordering position of the desired move when searched at different look-ahead level are not always the same, but are highly correlated provided that a reasonably good move ordering function is existed. For this reason, a normal distrubution could not fit these data well according to statistical tests. Hence, we define a conditional distribution function $F_{X1|Y1=y}(x)$ approximating our data as an estimate of $F_{X|Y=y}(x)$. It seems that the exact shape of the conditional distribution function is not really important for the results of the simulation runs; only the values around y must have a significantly higher probability.

The idea of node-cutting heuristic is illustrated in figures 4.4 and 4.5. The flow is described as follows:

Step 1: If current depth is $d+1$, then follow step 2, otherwise jump to step 6.

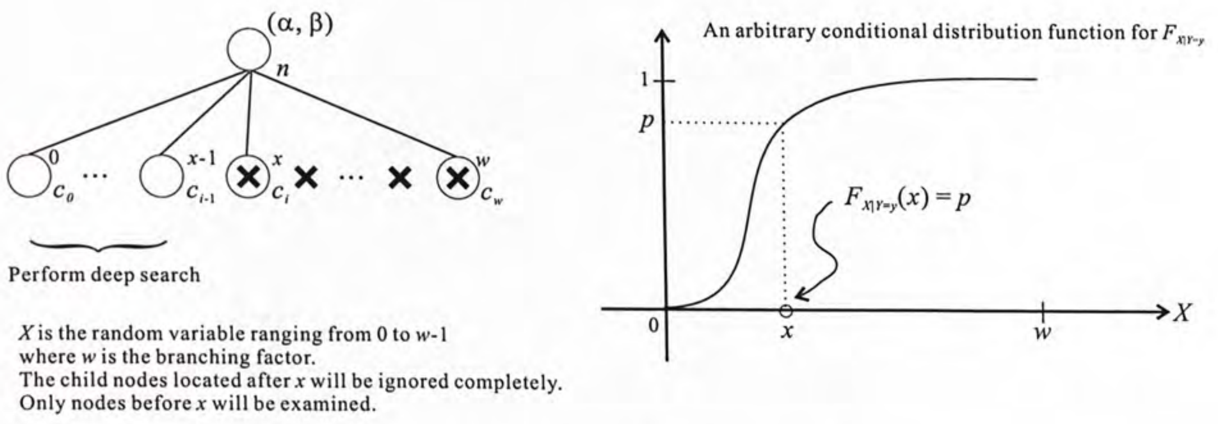
Step 2: Perform a short depth minimax search (depth $= s$).

Step 3: The desired move is found and it's position is label as y.

Step 4: Calculate x such that $F_{X|Y=y}(x) = p$ where p is an adjustable parameter.

Step 5: Ignore all move positioned after x.

Step 6: Perform the real minimax search with the reduced game tree.



(α, β)
n
0
c_0
x-1
c_{i-1}
x
c_i
w
c_w
Perform deep search
X is the random variable ranging from 0 to w-1
where w is the branching factor.
The child nodes located after x will be ignored completely.
Only nodes before x will be examined.
An arbitrary conditional distribution function for $F_{X|Y=y}$
1
p
$F_{X|Y=y}(x) = p$
0
x
w
X

Figure 4.5: The idea of node-cutting (2).

If we use aspiration NegaScout or MTD(f) algorithms we may experience with re-search some already visited nodes. In these cases, we can store the calculated x value so that we can directly jump to step 6 when revisiting those nodes.

4.4 Performance Evaluation

4.4.1 Determination of the Parameters

In this section, we present some experimental results in order to get a rough idea on the performance of the proposed idea. In our experiments, we use the game of Chinese Checkers to test the performance of node-cutting heuristic. In our 9669 board test positions, we find that the average branching factor of Chinese Checkers in middle-game is 67.8, which is reasonably large and is very suitable for testing our heuristic.

Figures 4.6, 4.7 and 4.8 show the conditional distribution function of Learner III. The result is obtained by measuring 9669 different board positions. We choose s to be 1 and d to be 2. It should be noticed that the meaning of the value s here is not the same as that of GPC. It can be seen that values around y have a significantly higher probability. This result is consistent with our expectation. The reason is that the ordering of move has strong correlation. The defined conditional distribution $F_{X1|Y1=y}(x)$ can be estimated from these graphs.

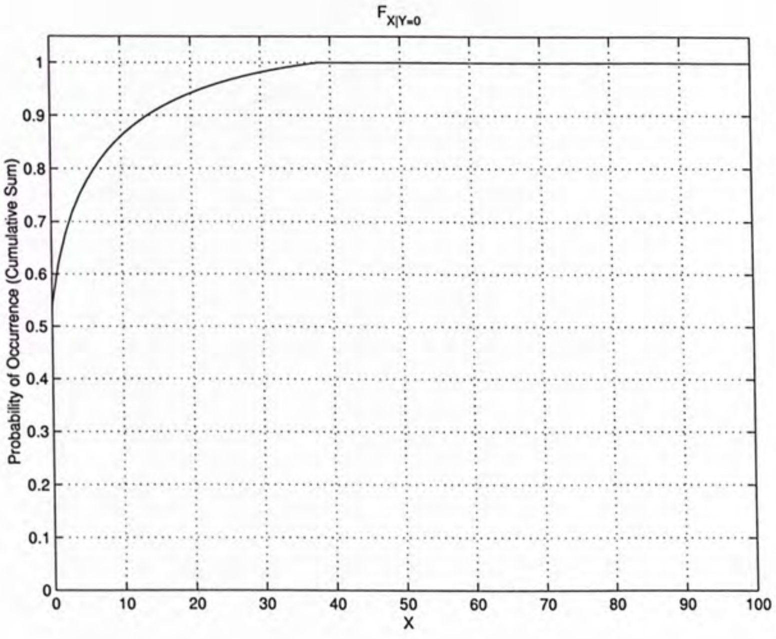


Figure 4.6: The conditional distribution $F_{X|Y=0}$ of Learner III.

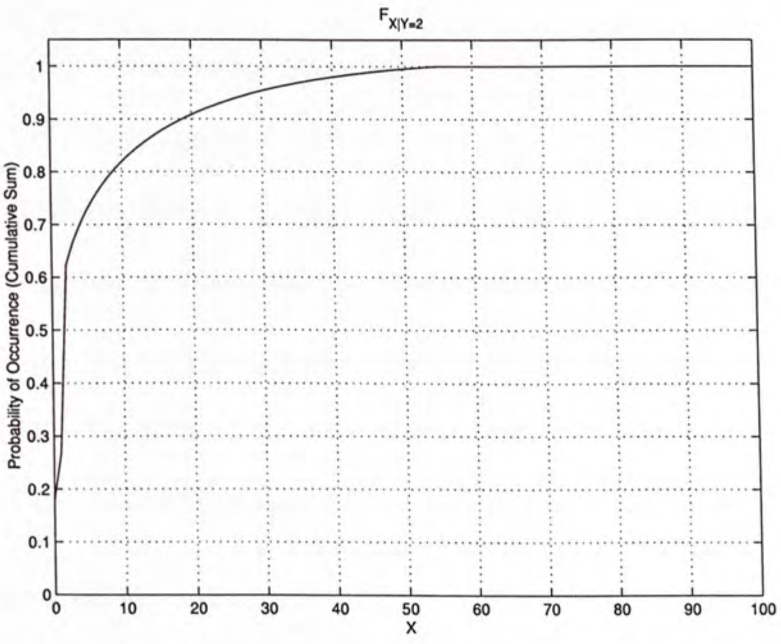


Figure 4.7: The conditional distribution $F_{X|Y=2}$ of Learner III.

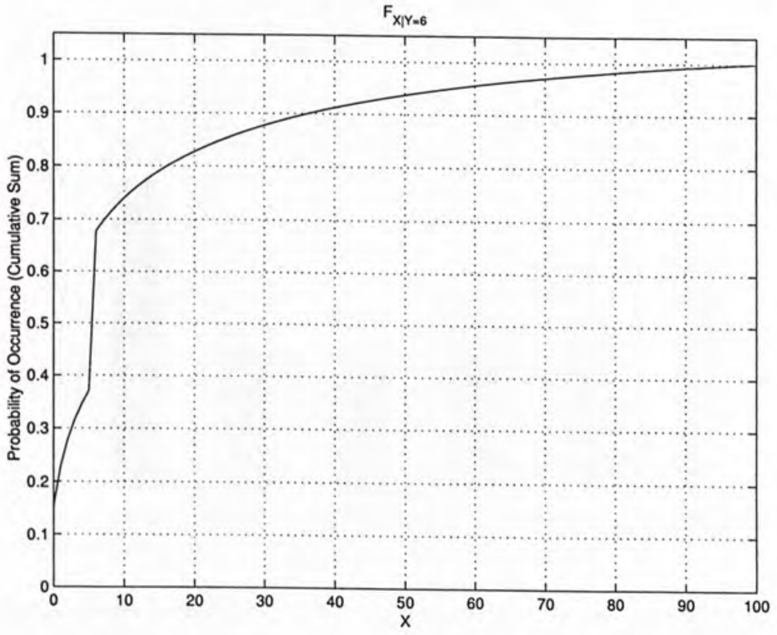


Figure 4.8: The conditional distribution $F_{X|Y=6}$ of Learner III.

Since MTD(f) is the current state-of-the-art minimax search algorithm, we choose this algorithm as our baseline.

The algorithms we compared are denoted as:

- MTD: *The MTD(f) algorithm*
- MTD-NC-95: *The MTD(f) algorithm enhanced with node-cutting heuristic (p = 0.95)*
- MTD-NC-96: *The MTD(f) algorithm enhanced with node-cutting heuristic (p = 0.96)*
- MTD-NC-97: *The MTD(f) algorithm enhanced with node-cutting heuristic (p = 0.97)*
- MTD-NC-98: *The MTD(f) algorithm enhanced with node-cutting heuristic (p = 0.98)*
- MTD-NC-99: *The MTD(f) algorithm enhanced with node-cutting heuristic (p = 0.99)*

4.4.2 Result of Experiments

Figures 4.9 to 4.15 show the comparison of 3 ply searches between Learner III with and without using node-cutting heuristic. It can be seen easily that node-cutting heuristic actually can

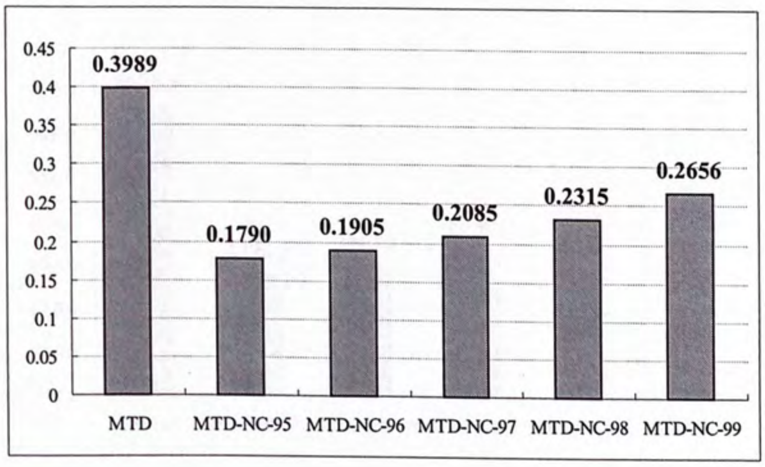


Figure 4.9: The Average Search Time Per Move for Learner III enhanced with Node-Cutting Heuristic.

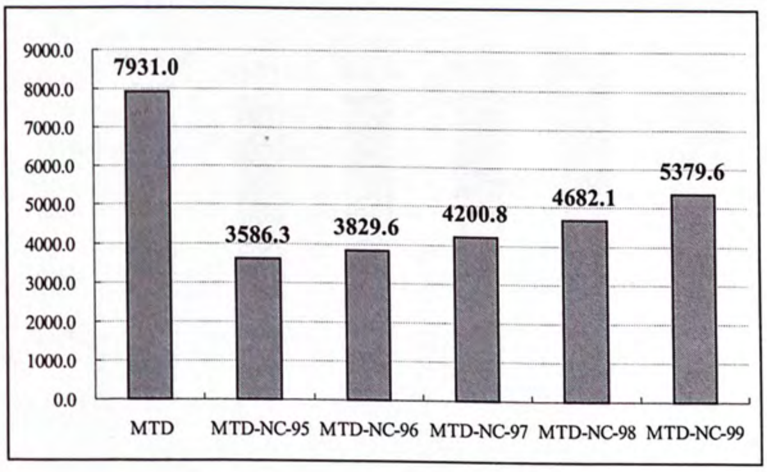


Figure 4.10: The Average Number of Node Visited Per Move for Learner III enhanced with Node-Cutting Heuristic.

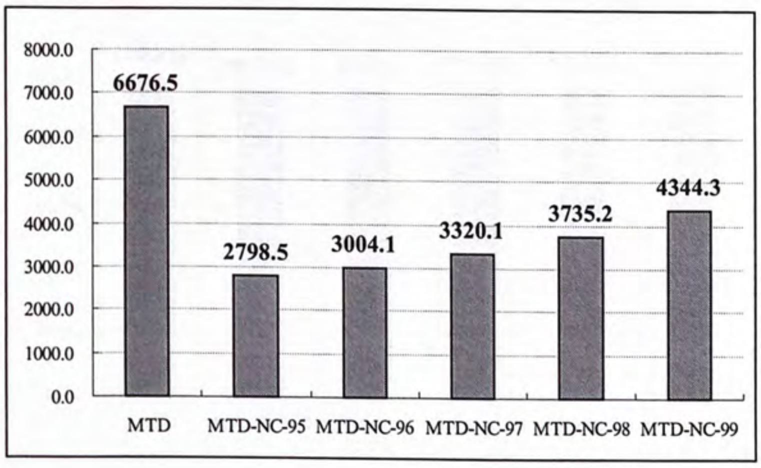


Figure 4.11: The Average Number of Bottom Position Visited Per Move for Learner III enhanced with Node-Cutting Heuristic.

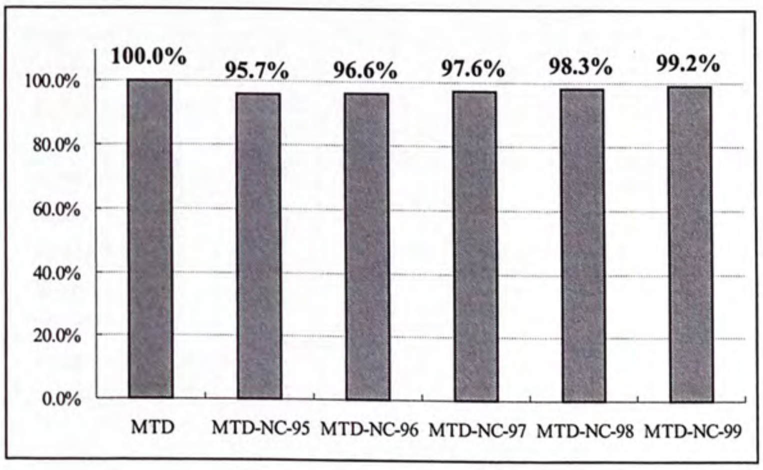


Figure 4.12: The Average Number of Same Move obtained Per Move for Learner III enhanced with Node-Cutting Heuristic.

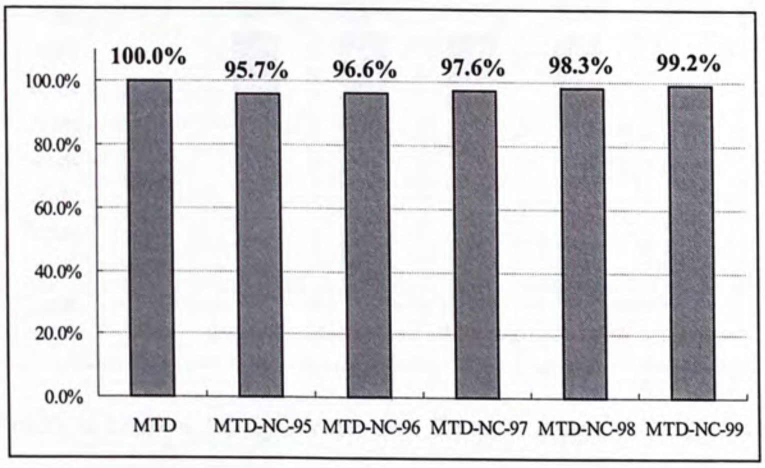


Figure 4.13: The Average Number of Same Value obtained Per Move for Learner III enhanced with Node-Cutting Heuristic.

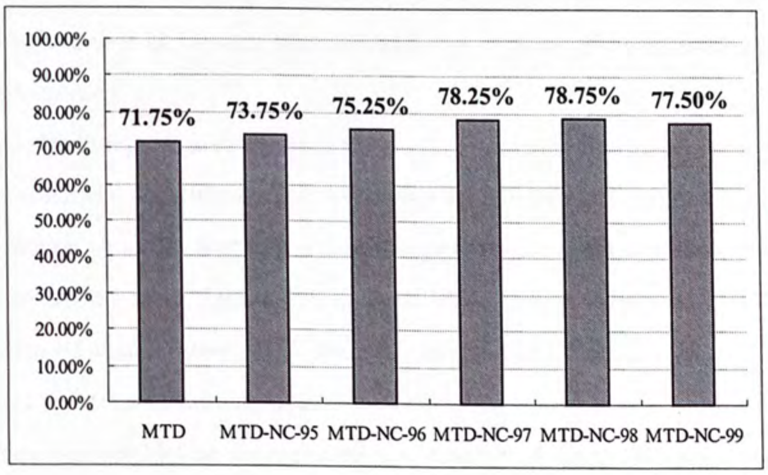


Figure 4.14: The Average Winning Percentage for Learner III enhanced with Node-Cutting Heuristic (Match against 100 different players).

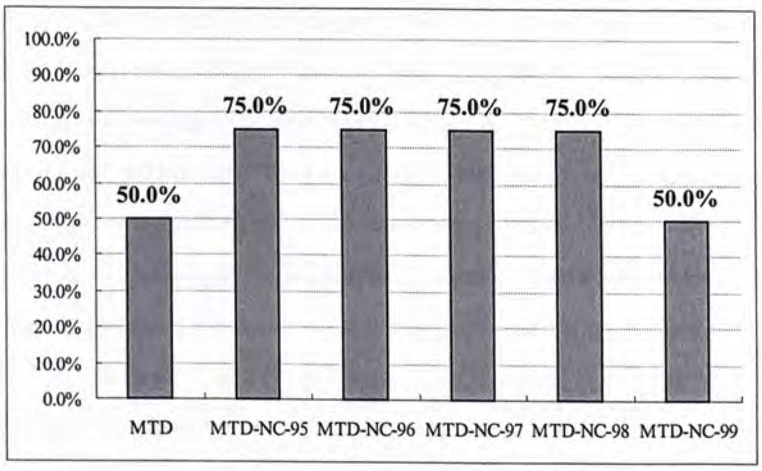


Figure 4.15: Result of Learner III against Learner III enhanced with Node-Cutting Heuristic.

speedup the search from 1.5 times to 2.2 times and maintain the same move for over 95 percent.

As shown from figure 4.9, the computation time per move can reduced at least 33% if enhanced with node-cutting heuristic. For example, MTD-NC-95 only consume 45% time of Learner III but obtain an even higher winning percentange.

The graph of NBP (figure 4.11) and NC (figure 4.10) have shown that node-cutting's execution time performance is proportional to the leaf node count and the number of static evaluations. The number of bottom position and the number of node visited decrease when the value of p decreases.

Besides time efficiency, we also measure the quality of move. Figure 4.12 shows the percentage of cases in which the algorithm would select the same move as Learner III while figure 4.13 shows the frequency that the algorithm would reported the same minimax value as Learner III. It can be seen that both figures are at least 95%, higher than that of GPC. This high percentage indicated that node-cutting heuristic can maintain most search result of Learner III and often hit the true minimax value.

Furthermore, we would like to measure the quality of play. We selected 100 different Chinese Checkers player programs that have similar strength as Learner III to have matches against our test programs. The result is shown in figure 4.14. The average winning percentage of algorithms enhanced with node-cutting heuristic are even higher than algorithms without using it. Apart from that, figure 4.15 shows the winning percentage of each algorithms match against Learner III. The winning percentage of MTD is 50%, which consistent with the fact that Learner III actually use MTD(f) as the search algorithm. It can be seen that Learner III enhanced with node-cutting heuristic have a stronger game playing strength.

The reason of node-cutting heuristic can obtain a high winning percentage is that minimax algorithm does not promote risk taking. Minimax algorithm always assume the opponent will make the best move even though its selected move resulting in a sure loss. Often in losing situations the best move may not be towards the highest min or max value, especially if it will still result in a loss. Node-cutting heuristic can guide a search to a more aggressive approach and take advantage of possible mistakes by the opponent. This can explain the reason of obtaining higher winning percentage.

The general performance of Node-Cutting heuristic is that it saves close to half of original search time but has a generally higher quality of play. By choosing a proper value of p, we can obtain the best winning percentage as well as minimizing the search time.

4.5 Summary

In this chapter, we proposed a node-cutting heuristic that is applicable to game tree of two-person non-random perfect-information zero-sum board game. Experiments were conducted and results showed that node-cutting lowered the number of total expanded nodes resulted in reducing the search time and also superior in terms of probability of hitting the mimimax value. We believe that our node-cutting heuristic is useful in the area of game, and especially large branching factor game.

Chapter 5

The Integrated Strategy

5.1 Introduction

The previous chapter showed how forward pruning techniques and Node-Cutting heuristic could speedup the search. We push this idea further in this chapter. In the next section, we will discuss the effect of combined strategy. In sections three and four, we integrate our strategy into a typical board game, Chinese Checkers. Experiments and results are shown in section three. The last section summarizes our proposed strategy.

5.2 Combination of GPC, FGPC and Node-Cutting Heuristic

Recalled from previous chapters, the mechanism of forward pruning heuristic is to eliminate unpromising move in advance by looking a few steps forward instead of a full-depth search. Node-cutting heuristic is to remove unpromising move at early stage by performing a short depth search. However, occasionaly there exists cases that node-cutting heuristic terminates the search prematurely. In order to avoid this undesirable effect, we try to combine GPC, FGPC and node-cutting heuristic. We observe that we can prevent the premature termination of search and simultaneously obtain a reliable result.

We can make efficient use of the allocated time. For the first few child nodes, we will

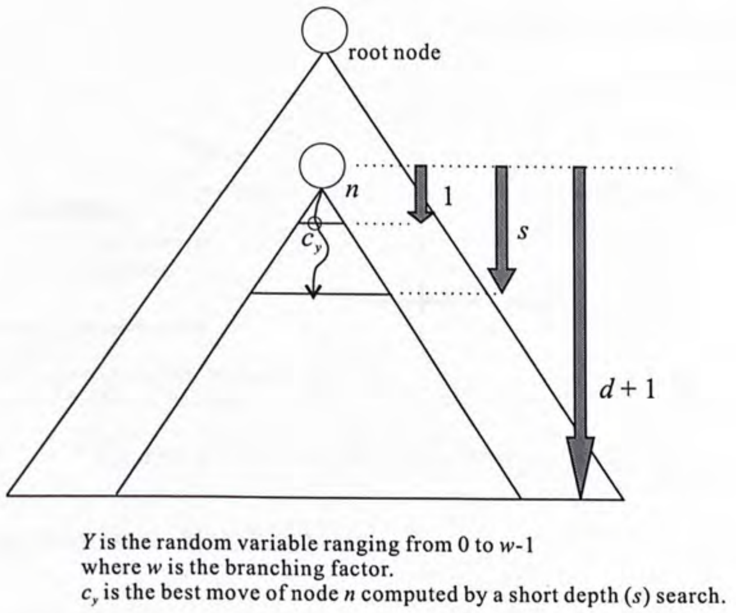

root node
n
c_y
1
s
$d+1$
Y is the random variable ranging from 0 to w-1
where w is the branching factor.
c_y is the best move of node n computed by a short depth (s) search.

Figure 5.1: The idea of integrated strategy.

examine them if they will not yield a value outside the current search window since child nodes order at the front usually contain good moves, if a reasonably good move ordering function exists. But for the rest of the child nodes, it is not likely to have good moves (by the practical result obtained from Chinese Checkers) such that its minimax value would be propagated to parent. We can perform FGPC such that once a child node's value lie outside the current search window, the search would be stopped and propagate the best obtained value to its parent. As a result, the child subtrees position after it will not be examined. It maintains a high number of cut-offs but the quality is still guranteed since fast cut-offs will not occur at the first few nodes.

Figures 5.1 and 5.2 describes the idea of the integrated strategy. The new strategy is easily incorporated into existing game-playing programs and the modification is quite trival that we will not describe much in here.

We take the advantage of GPC, FGPC and node-cutting. The new stretagy not only reduces the search tree size but also maintains a high rate of hitting the correct minimax value. The next section will show the experimental results to support it.

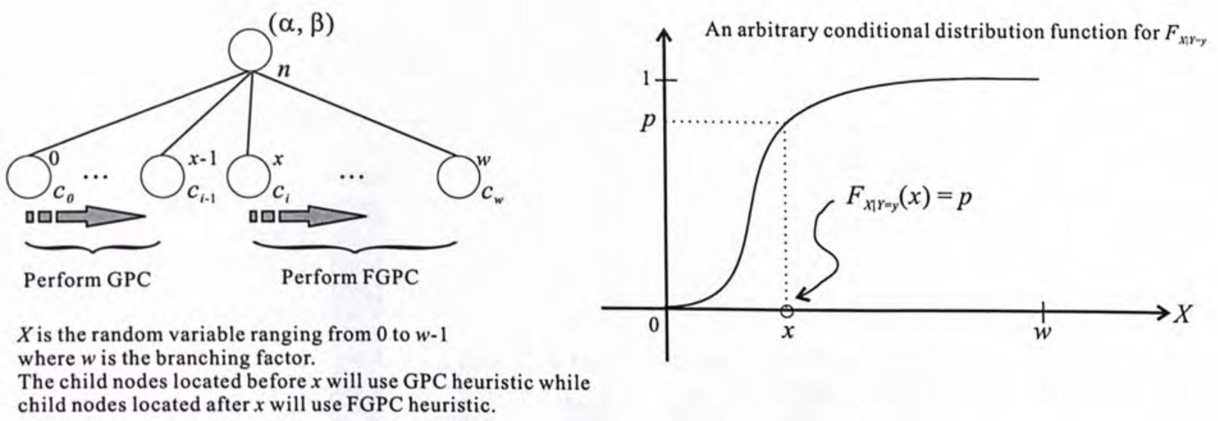

(α, β)
n
0
x-1
x
w
c_0
c_{i-1}
c_i
c_w
Perform GPC
Perform FGPC
X is the random variable ranging from 0 to w-1 where w is the branching factor.
The child nodes located before x will use GPC heuristic while child nodes located after x will use FGPC heuristic.
An arbitrary conditional distribution function for $F_{X|Y=y}$
1
p
$F_{X|Y=y}(x) = p$
0
x
w
X

Figure 5.2: The idea of integrated strategy (2).

5.3 Performance Evaluation

In order to demonstrate the potential of our work as a practical searching tool some experiments have been run. In our experiments, same as previous chapters, we use the same parameters to test the performance of our new strategy.

The algorithms we compared are denoted as:

- MTD: *The MTD(f) algorithm*
- MTD-IS-95: *The MTD(f) algorithm enhanced with integrated strategy (p = 0.95)*
- MTD-IS-96: *The MTD(f) algorithm enhanced with integrated strategy (p = 0.96)*
- MTD-IS-97: *The MTD(f) algorithm enhanced with integrated strategy (p = 0.97)*
- MTD-IS-98: *The MTD(f) algorithm enhanced with integrated strategy (p = 0.98)*
- MTD-IS-99: *The MTD(f) algorithm enhanced with integrated strategy (p = 0.99)*

Figures 5.3 to 5.9 show the comparison of 3 ply searches between Learner III with and without using integrated strategy. Speedup is the ratio of the number of nodes visited while the term same move states the percentage of cases in which both versions selected the same move. It can be seen easily that integrated strategy actually can speedup the search up to 2.5 times and maintain the same move with more than 94 percent.

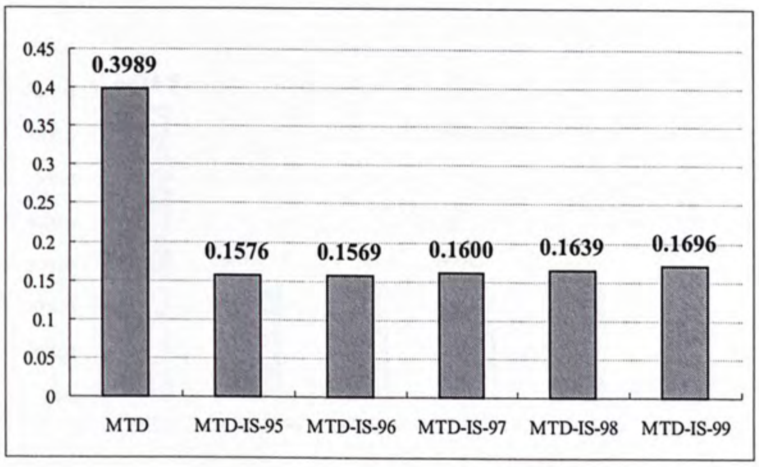


Figure 5.3: The Average Search Time Per Move for Learner III enhanced with integrated strategy.

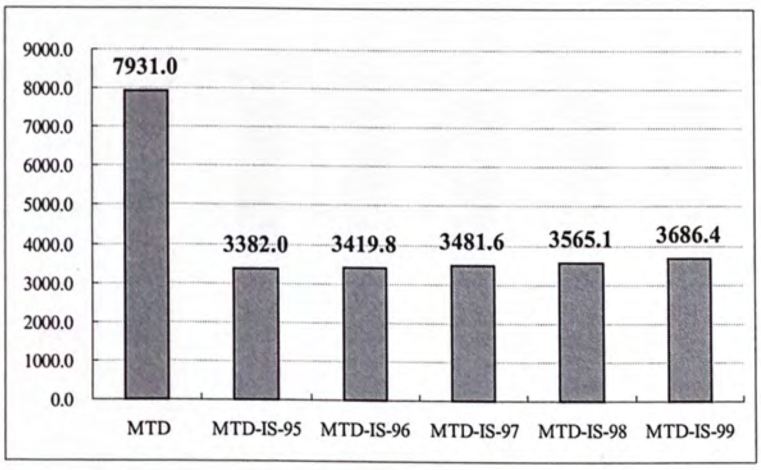


Figure 5.4: The Average Number of Node Visited Per Move for Learner III enhanced with integrated strategy.

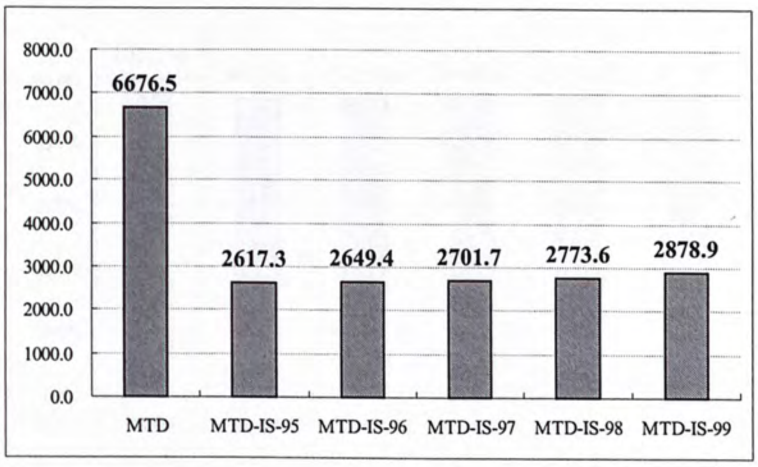


Figure 5.5: The Average Number of Bottom Position Visited Per Move for Learner III enhanced with integrated strategy.

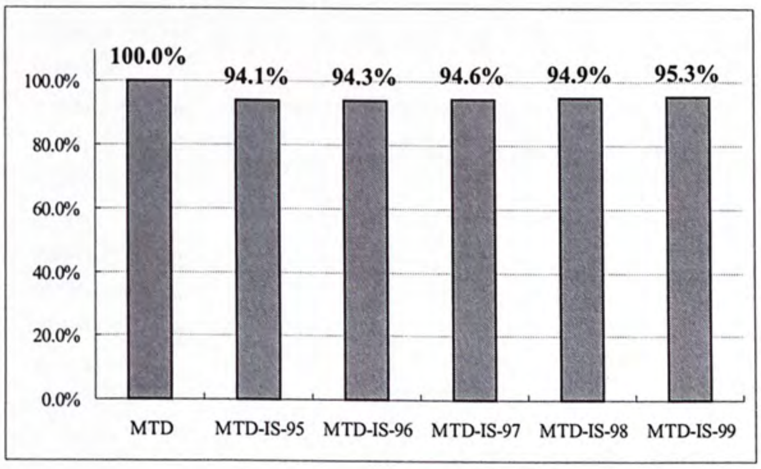


Figure 5.6: The Average Number of Same Move obtained Per Move for Learner III enhanced with integrated strategy.

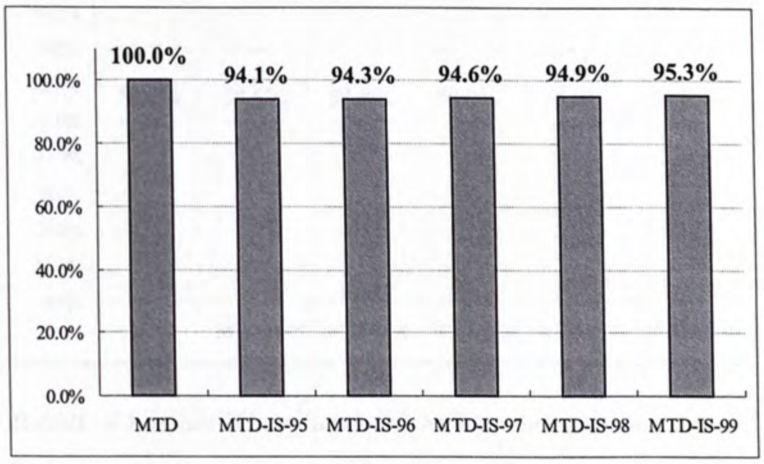


Figure 5.7: The Average Number of Same Value obtained Per Move for Learner III enhanced with integrated strategy.

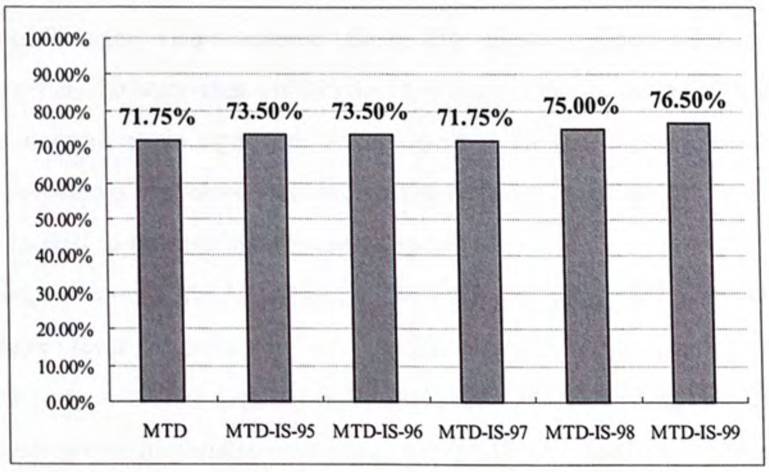


Figure 5.8: The Average Winning Percentage for Learner III enhanced with integrated strategy (Match against 100 different players).

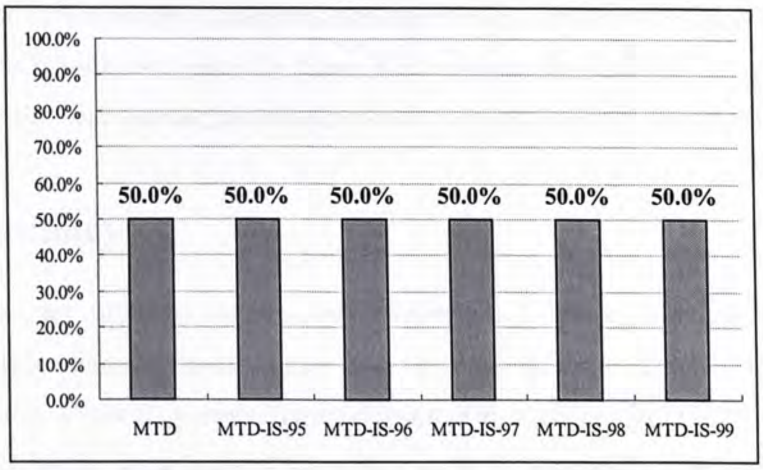


Figure 5.9: Result of Learner III against Learner III enhanced with integrated strategy.

As shown in figure 5.3, the computation time per move can reduced 60% after using Node-Cutting heuristic. For instance, MTD-IS-95 have a speedup of 2.5 times but obtain an even higher winning percentange than MTD times.

The graph of NBP (figure 5.5) and NC (figure 5.4) are consistent with the result of computation time. The number of bottom position and number of interior node visited decrease when the value of p decreases.

Besides time efficiency, we also measure the quality of move. Figure 5.6 shows the percentage of cases in which the algorithm would select the same move as Learner III while figure 5.7 shows the frequency that the algorithm would reported the same minimax value as Learner III. The high percentage indicated that integrated strategy can maintain most search result of Learner III as well as minimizing the search speed.

Furthermore, we would like to measure the quality of play. We selected 100 different Chinese Checkers player programs that have similar strength as Learner III to have matches against our test programs. The result is shown in figure 5.8. The average winning percentages of algorithms enhanced with integrated strategy are even higher than algorithms without using it. Apart from that, figure 5.9 shows the winning percentage of each algorithms match against Learner III. The winning percentage of MTD is 50%, which consistent with the fact that Learner III actually use MTD(f) as the search algorithm. It can be seen that Learner III and Learner III enhanced with integrated strategy having the similar game playing strength.

The general performance of integrated strategy is that it saves more than half of original search time but maintains a generally higher play strength. By choosing a proper value of p, we can obtain the best winning percentage as well as minimizing the search time.

5.4 Summary

In this chapter, we combined our proposed heuristics to formulated the integrated strategy framework. Experimental results shows that the combination of GPC, FGPC and node-cutting results in about 60 percent time reduction over conventional full-width search. The combination yields the same factor of fewer total nodes as compared to the original algorithm. These results showed that our strategy allocates time efficiently and successfully. We believe that using our forward pruning and node-cutting heuristics together, time is save significantly and saved time can allow us to search deeper for a better result.

Chapter 6

Conclusions and Future Works

6.1 Conclusions

Most game-playing programs choose their moves by searching a large tree of potential continuations. The problem with tree searching is that the search space grows exponentially with the depth of the search. The original minimax algorithms waste most of their time by analyzing irrelevant lines. On the other hand, efficient and intelligent pruning techniques are required to reduce the search space. We aim at making more efficient use of the allocated time.

The central idea of this thesis concerns about selective pruning for game tree of two-person non-random perfect information zero-sum game. Firstly, we have described a forward pruning framework. We reformulate ProbCut to GPC, a generalised selective search extension to alpha-beta algorithm. GPC examines the same leaf nodes in the same order as ProbCut. They produce the same number of cut-offs, obtain the same minimax value and make the same move for a given game tree.

We also propose a fast forward pruning framework, modified from GPC. The results of applying our heuristics to game are described. The modified framework reduces the search effort significantly. It outperformed previous effective and successful heuristic, ProbCut, in time efficiency. In order to demonstrate the potential of our work as a practical searching tool some experiments have been run. Experiment showed that the search speed can speedup by almost 3 times in the game of Chinese Checkers.

Apart from those discussed forward pruning strategies, cutting down the branching factor of a game tree is another workable way to reduce the computation time without lowering the quality of play significantly. In this work, we formulated a node-cutting heuristic. By using the pattern and correlation of move ordering function, node-cutting heuristic can speedup the search and obtain a higher winning percentage than full-width minimax search. Our node-cutting heuristic is useful in the area of game, and especially large branching factor game.

Finally, we constructed an integrated strategy that is the combination of GPC, FGPC and node-cutting heuristic. We can perform a fast selective search but still maintain a high rate of hitting the maximum value. Ken Thompson showed that search depth was strongly correlated with performance in chess [45]. Searching one move (or one ply) deeper made a huge difference in performance. We believe that using our forward pruning and node-cutting heuristic together, time is saved significantly and more room for deeper search is guaranteed.

6.2 Future Works

There are a number of possible future extensions of this work under consideration. The research described in the previous chapters has uncovered a number of interesting avenues for further research. We list the following:

- *GMPC: Generalised Multi-ProbCut*

  All experiments in this work were performed for fixed s and d values. Multiple s and d value pairs that used by MPC were not tested. We believe that since game tree of GPC and ProbCut are same in size, the behavior of GMPC will not be different from that of MPC. However, experiments are needed to show whether this is indeed the case.

- *Null-window Boundary Test*

  The main idea of ProbCut and GPC is that they make use of boundary test to determine whether a deep search is necessary. However, in some cases, one does not have to test both the upper and lower bounds of a search window but either one is enough. More analysis and experiments are needed to gain a better understanding.

- *Properties of Move Ordering*

  The node-cutting heuristic makes use of the correlation of the location of the best-move that searche in different depth. This property shows there are much room for further research in move ordering. Chapter 4 only scratches the surface of this.

- *GPC, FGPC and Node-cutting*

  In addition to more analysis, more experiments are needed to gain a deeper insight and gain a better understanding of the relation between GPC, FGPC and node-cutting heuristic.

- *Efficiently use of allocated time*

  Our experiments show the results of Alpha-Beta algorithm and its variants enhanced with our heuristics. Applying different heuristics to the same algorithm at different game stage may be a good model to understand the effectiveness and efficiency of this kind of time allocation.

Appendix A

Examples

To illustrate an idea of how the search algorithms described in previous chapters work, we give a simple example in this Appendix. For simplicity, we just show crucial steps of the algorithm (figure A.1) and it is easy to generalize to a complete one based on these steps. The figures beside the nodes are the minimax values of them. The example of the minimax algorithm is illustrated by graphs (figures A.2-A.5). Since the flow of tree traversed by other algorithms are similar, we just show the value changed and result in tables (figures A.6-A.8) for simplicity.

The parameters used in the examples are listed as follow: $d = 2, s = 1, a = 1, b = 0, \sigma = 10, \Phi^{-1}(p) = 1.5$. The values for α_n, β_n, and g_n of each node traversed by each alogrithm are shown in the following figures.

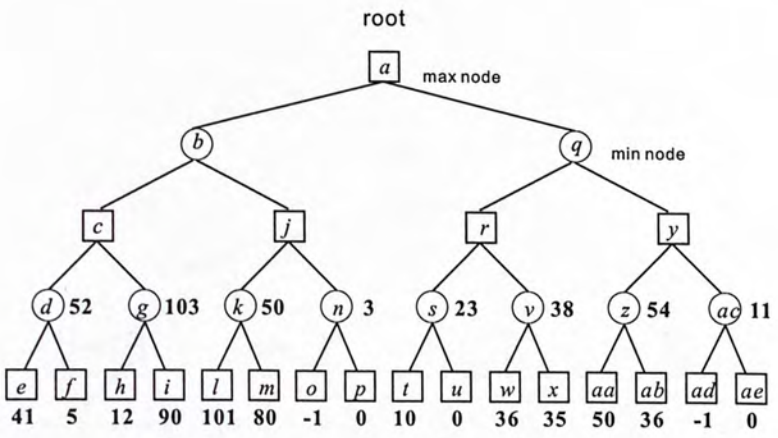


Figure A.1: The example tree.

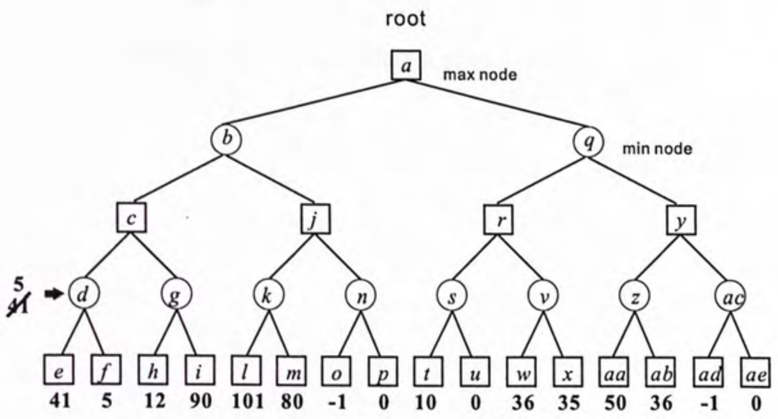


Figure A.2: The Minimax example (step 1).

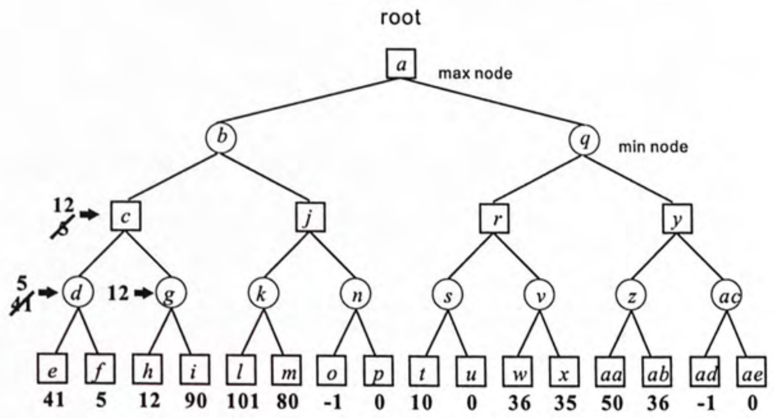


Figure A.3: The Minimax example (step 2).

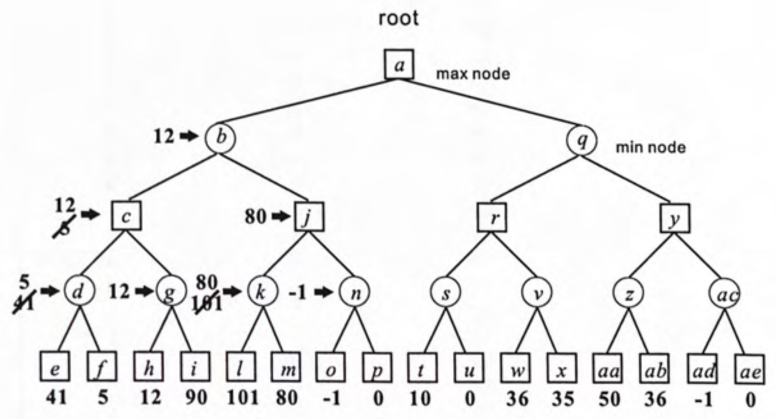


Figure A.4: The Minimax example (step 3).

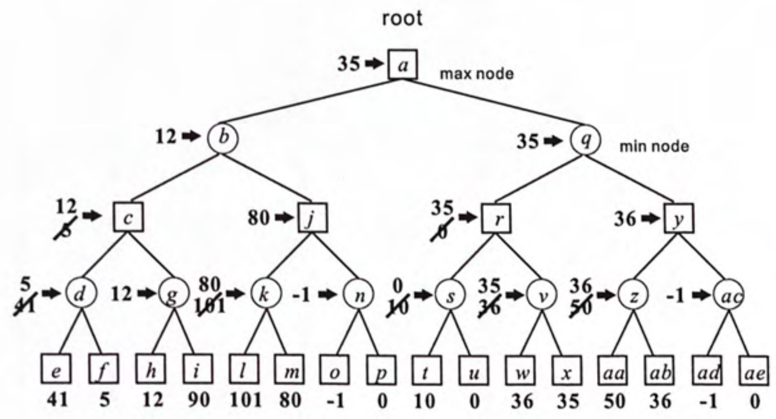


Figure A.5: The Minimax example (step 4).

#	*n*	α_n	β_n	g_n	*Cut-off?*
1	*a*	-∞	+∞	-∞	
2	*b*	-∞	+∞	+∞	
3	*c*	-∞	+∞	-∞	
4	*d*	-∞	+∞	+∞	
5	*e*	-∞	+∞	41	
6	*d*	-∞	41	41	
7	*f*	-∞	41	5	
8	*d*	-∞	5	5	
9	*c*	5	+∞	5	
10	*g*	5	+∞	+∞	
11	*h*	5	+∞	12	
12	*g*	5	12	12	
13	*i*	5	12	90	
14	*g*	5	12	12	
15	*c*	12	+∞	12	
16	*b*	-∞	12	12	
17	*j*	-∞	12	-∞	
18	*k*	-∞	12	+∞	
19	*l*	-∞	12	101	
20	*k*	-∞	12	101	
21	*m*	-∞	12	80	
22	*k*	-∞	12	80	
23	*j*	80	12	80	Yes
24	*b*	-∞	12	12	
25	*a*	12	+∞	12	
26	*q*	12	+∞	+∞	
27	*r*	12	+∞	-∞	
28	*s*	12	-∞	+∞	
29	*t*	12	-∞	10	
30	*s*	12	10	10	Yes
31	*r*	12	+∞	10	
32	*v*	12	+∞	+∞	
33	*w*	12	+∞	36	
34	*v*	12	36	36	
35	*x*	12	36	35	
36	*v*	12	35	35	
37	*r*	35	+∞	35	
38	*q*	12	35	35	
39	*y*	12	35	-∞	
40	*z*	12	35	+∞	
41	*aa*	12	35	50	
42	*z*	12	35	50	
43	*ab*	12	35	36	
44	*z*	12	35	36	
45	*y*	36	35	36	Yes
46	*q*	12	35	35	
47	*a*	35	+∞	35	Finished

Figure A.6: The Alpha-Beta example.

#	n	α_n	β_n	g_n	*Remark*
1	a	$-\infty$	$+\infty$	$-\infty$	
2	b	$-\infty$	$+\infty$	$+\infty$	
3	c	$-\infty$	$+\infty$	$-\infty$	
4	c	$+\infty - 1$	$+\infty$	$-\infty$	β-test
5	d	$+\infty - 1$	$+\infty$	52	
6	c	$+\infty - 1$	$+\infty$	52	
7	g	$+\infty - 1$	$+\infty$	103	
8	c	$+\infty - 1$	$+\infty$	103	
9	c	$-\infty$	$-\infty + 1$	$-\infty$	α-test
10	d	$-\infty$	$-\infty + 1$	52	
11	c	52	$-\infty + 1$	52	cut-off
12	d	$-\infty$	$+\infty$	$+\infty$	
13	e	$-\infty$	$+\infty$	41	
14	d	$-\infty$	41	41	
15	f	$-\infty$	41	5	
16	d	$-\infty$	5	5	
17	c	5	$+\infty$	5	
18	g	5	$+\infty$	$+\infty$	
19	h	5	$+\infty$	12	
20	g	5	12	12	
21	i	5	12	90	
22	g	5	12	12	
23	c	12	$+\infty$	12	
24	b	$-\infty$	12	12	
25	j	$-\infty$	12	$-\infty$	
26	j	26	27	$-\infty$	β-test
27	k	26	27	50	
28	j	50	27	50	cut-off
29	j	$-\infty$	12	12	ProbCut cut-off
30	b	$-\infty$	12	12	
31	a	12	$+\infty$	12	
32	q	12	$+\infty$	$+\infty$	
33	r	12	$+\infty$	$-\infty$	
34	r	$+\infty - 1$	$+\infty$	$-\infty$	β-test
35	s	$+\infty - 1$	$+\infty$	23	
36	r	$+\infty - 1$	$+\infty$	23	
37	v	$+\infty - 1$	$+\infty$	38	
38	r	$+\infty - 1$	$+\infty$	38	
39	r	-3	-2	$-\infty$	α-test
40	s	-3	-2	23	
41	r	23	-2	23	cut-off
42	s	12	$+\infty$	$+\infty$	
43	t	12	$+\infty$	10	
44	s	12	10	10	cut-off
45	r	12	$+\infty$	10	
46	v	12	$+\infty$	$+\infty$	
47	w	12	$+\infty$	36	
48	v	12	36	36	
49	x	12	36	35	
50	v	12	35	35	
51	r	35	$+\infty$	35	
52	q	12	35	35	
53	y	12	35	$-\infty$	
54	y	49	50	$-\infty$	β-test
55	z	49	50	54	
56	y	54	50	54	cut-off
57	y	49	50	50	ProbCut cut-off
58	q	12	35	35	
59	a	35	$+\infty$	35	Finished

Figure A.7: The ProbCut example.

#	n	α_n	β_n	g_n	*Remark*
1	*a*	$-\infty$	$+\infty$	$-\infty$	
2	*b*	$-\infty$	$+\infty$	$+\infty$	
3	*c*	$+\infty - 1$	$+\infty$	$-\infty$	β-test
4	*d*	$+\infty - 1$	$+\infty$	52	
5	*c*	$+\infty - 1$	$+\infty$	52	
6	*g*	$+\infty - 1$	$+\infty$	103	
7	*c*	$+\infty - 1$	$+\infty$	103	
8	*c*	$-\infty$	$+\infty$	$-\infty$	
9	*d*	$-\infty$	$+\infty$	$+\infty$	
10	*e*	$-\infty$	$+\infty$	41	
11	*d*	$-\infty$	41	41	
12	*f*	$-\infty$	41	5	
13	*d*	$-\infty$	5	5	
14	*c*	5	$+\infty$	5	
15	*g*	5	$+\infty$	$+\infty$	
16	*h*	5	$+\infty$	12	
17	*g*	5	12	12	
18	*i*	5	12	90	
19	*g*	5	12	12	
20	*c*	12	$+\infty$	12	
21	*b*	$-\infty$	12	12	
22	*j*	26	27	$-\infty$	β-test
23	*k*	26	27	50	
24	*j*	50	27	50	cut-off
25	*b*	$-\infty$	12	12	GPC cut-off
26	*a*	12	$+\infty$	12	
27	*q*	12	$+\infty$	$+\infty$	
28	*r*	$+\infty - 1$	$+\infty$	$-\infty$	β-test
29	*s*	$+\infty - 1$	$+\infty$	23	
30	*r*	$+\infty - 1$	$+\infty$	23	
31	*v*	$+\infty - 1$	$+\infty$	38	
32	*r*	$+\infty - 1$	$+\infty$	38	
33	*r*	12	$+\infty$	$-\infty$	
34	*s*	12	$+\infty$	$+\infty$	
35	*t*	12	$+\infty$	10	
36	*s*	12	10	10	cut-off
37	*r*	12	$+\infty$	10	
38	*v*	12	$+\infty$	$+\infty$	
39	*w*	12	$+\infty$	36	
40	*v*	12	36	36	
41	*x*	12	36	35	
42	*v*	12	35	35	
43	*r*	35	$+\infty$	35	
44	*q*	12	35	35	
45	*y*	49	50	$-\infty$	β-test
46	*z*	49	50	54	
47	*y*	54	50	54	cut-off
48	*q*	12	35	35	GPC cut-off
49	*a*	35	$+\infty$	35	Finished

Figure A.8: The GPC-AB example.

Appendix B

The Rules of Chinese Checkers

Chinese Checkers is played on a six-pointed star-shape board by two, three, four or six players. As shown in figure B.1, there are 121 positions on the board. At the beginning of a game, each player's ten marbles occupy a triangular area at an opposite side of the board. We call this the home area of a player. The other triangular areas are call neutral zones.

Since the board is embedded in a hexagonal grid, each position on it is generally connected to neighbors in six directions, except when located at the boundary or a corner, in which case the position has 5, 4, or 2 neighbors. At each turn, a player can move any one of his marbles into a neighboring position, provided that such a position exists and is not already occupied by another marble, either belonging to him or his opponent. A marble may also in one move, make a sequence of jump over other marbles, which either belong to the player or his opponent. Each jump must be made according to the follow rule. Suppose that a marble at A jumps over a marble at B. The former will land at position C, where B is equidistance from A and C, and A, B, and C are colinear. The jump is only allowed if every position on the line AC (inclusive) exists, and none of these are occupied before the jump except A and B. When a marble is moved to an adjacent position, or takes a sequence of jumps, it may not end up in a position in a neutral zone. The intermediate steps of a sequence of jumps, however, may use positions in the neutral zones.

The objective of the game is to move all of one's marbles into home area of one's opponent before one's opponent moves all his marbles into one's own home area. A game is considered

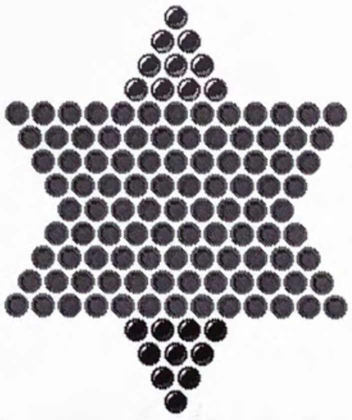


Figure B.1: The board of Chinese Checkers.

a draw if player 1 makes the first move of the game, and player 2 moves all his marbles into player 1's home area one move after player 1 moves all his marbles into player 2's home area.

Appendix C

Application to Chinese Checkers

In the game of Chinese Checkers, each player in turn moves a marble on the board to a desired destination. The board evaluation function is used to select the most desirable move by evaluating each resulting board configuration. Unlike previous efforts to implement computer game players, Chinese Checkers is not a well-understood game. There is no literature describing good Chinese Checkers strategy. The evaluation function of Learner III is used in conjunction with MTD(f) search mechanism to evaluate board position at a deep level. Learner III look-ahead 3 levels for middle-game and 4 levels for endgame. No opening books are used in the start-game.

In Learner III's evaluation function, it consists of 3 features. They are mobility, positioning and piece power. Mobility is the number of current legal moves. In Chinese Checkers, a move is either a simple move or a jump. A simple move is to move the marble to one of the six adjacent vacant positions. A jump is to move the marble over a marble with none or some vacant positions in between. A series of jump is a sequence of jump that the destination of a jump is the start of another jump. As we know, a jump is in general more benefit than a move as it can move much further by a single move. So mobility is further divided into two sub-features, they are move mobility and jump mobility. As we expected, jump mobility should weighted more. Positioning is the score of a particular position if a marble located in it. Positions that near the destination should scored higher than positions that near the home, as expected. Piece power is the power of each marble. In Chinese Checkers, the front-side marbles does not share the same amount of power as the back-side marbles. Marbles at the back-side is, in general, more important and need marbles at the front-side to take care them. The final score, counting from all the above features, is the score difference between the two players. The weighing coefficients of the above features are integers ranged from 0 to 8 inclusively. They are combined and self-tuned adaptively by genetic algorithm [42].

Bibliography

[1] Don F. Beal, "Experiments with the null-move," *Advances in Computer Chess 5*, pp. 65-79, Amsterdam, 1989. Elsevier Science Publishers.

[2] H. J. Berliner, "The b* tree search algorithm: A best-first proof procedure," *Artificial Intelligence*, no. 21, pp. 23-40, 1979

[3] J. A. Birmingham and P. Kent, "Tree searching and tree pruning techniques," *Advances in Computer Chess*, M. R. B. Clarke, Ed. Edinburgh Univ. Press, Edinburgh, Scotland, pp. 89-96, 1977

[4] A. L. Brudno, "Bounds and valuations for abridging the search of estimates," *Problems of Cybernetics*, Pergamon Press, Elmsford, N.Y., pp. 225-241, 1963.

[5] Michael Buro, "ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm," *Journal of the International Computer Chess Association*, 18(2), pp. 71-76, 1995.

[6] Michael Buro, "The Othello match of the year: Takeshi Murakami vs Logistello," *Journal of the International Computer Chess Association*, 20(3), pp. 189-193, 1997.

[7] Michael Buro, "Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. In H.J. van den Herik and H. Iida, editors," *Games in AI Research*, pp. 77-96, 2000.

[8] Michael Buro, "Improving Heuristic Mini-Max Search by Supervised Learning," *Artificial Intelligence*, vol. 134 (1-2), pp. 85-99, 2002.

[9] Murray S. Campbell and T. Anthony Marsland, "A comparision of minimax tree search algorithm," *Artificial Intelligence*, vol. 20, pp. 347-367, 1983.

[10] Kumar Chellapilla and David B. Fogel, "Anaconda Defeats Hoyle 6-0: A Case Study Competing an Evolved Checkers Program against Commercially Available Software," *Proceedings of the 2000 Congress on Evolutionary Computation*, IEEE Press, Piscataway, NJ, pp. 857-863, 2000.

[11] Kenneth J. Chisholm and Peter V. G. Bradbeer, "Machine Learning Using a Genetic Algorithm to Optimise a Draughts Program Board Evaluation Function," *IEEE International Conference on Evolutionary Computation*, pp. 715-720, 1997.

[12] Carl Ebeling, "All the Right Moves," *PhD thesis*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, MIT Press, Cambridge, Massachusetts, 1987.

[13] Gabriel J. Ferrer and Worthy N. Martin, "Using Genetic Programming to Evolve Board Evaluation Functions," *IEEE International Conference on Evolutionary Computation*, vol. 2, pp. 747-752, 1995.

[14] Peter W. Frey, editor, *Chess Skill in Man and Machine*, Springer-Verlag, New York, NY, 1977.

[15] J. Gillogly, "Performance analysis of the technology chess program," *PhD thesis*, Department of Computer Science, Carnegie-Mellon University, 1978.

[16] Rattikorn Hewett and Krishnamurthy Ganesan, "Consistent Linear Speedup in Parallel Alpha-Beta Search," *Proceedings of Fourth International Conference on Computing and Information*, pp. 237-240, 1992.

[17] Hermann Kaindl, Reza Shams and Helmut Horacek, "Minimax Search Algorithms with and without Aspiration Windows," *IEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 12, pp. 1225-1235, Dec. 1991.

[18] Graham Kendall and Glenn Whitwell, "An Evolutionary Approach for the Tuning of a Chess Evaluation Function using Population Dynamics," *Proceedings of the 2001 Congress on Evolutionary Computation*, vol. 2, pp. 995-1002, 2001

[19] Donald E. Knuth and Ronald W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293-326, 1975.

[20] David H. Lorenz and Shaul Markovitch, "Derivative Evaluation Function Learning Using Genetic Operators," *Games: Planning and Learning, Papers from the AAAI 1993 Fall Symposium*, Raleigh, North Carolina, October 22-24, AAAI Press, Menlo Park, California. Technical Report FS9302, 1993

[21] T. Anthony Marsland, "Relative performance of the alpha-beta algorithm," *Newsletter of the International Computer Chess Association*, vol. 5, no. 2, pp. 21-24, 1982

[22] T. Anthony Marsland and M. Campbell, "Parallel search of strongly ordered game trees," *Computing Surveys*, 14(4), pp. 533-551, December 1982.

[23] T. Anthony Marsland, "Relative efficiency of alpha-beta implementations," *Proceedings of the 8th International Conference on Artificial Intelligence*, Morgan Kaufmann, Los Altos, Calif., pp. 763-766, 1983

[24] David Allen McAllester, "A new procedure for growing min-max trees," Tech. rep., MIT Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., July 1985.

[25] David E. Moriarty and Risto Miikkulainen, "Improving Game-Tree Search with Evolutionary Neural Networks," *Proceedings of the First IEEE Conference on Evolutionary Computation*, vol. 1, pp. 496 -501, 1994

[26] Agata Muszycka and Rajjan Shinghal, "An empirical comparison of pruning strategies in game trees," *IEEE Transactions on Systems, Man and Cybernetics*, 15(3), pp. 389-399, May/June 1985.

[27] Allen Newell, Cliff Shaw and Herbert Simon"Chess playing programs and the problem of complexity," *IBM Journal of Research and Development*, vol. 4, no. 2, pp. 320-335, 1958.

Also in Computers and Thought, Feigenbaum and Feldman (eds.), pp. 39-70, McGraw-Hill, 1963.

[28] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publ., Palo Alto, Calif., 1980.

[29] J. Pearl, "Scout: A simple game-searching algorithm with proven optimal properties" *Proc. First Annu. Nat. Conf. Artificial Intelligence*, Stanford, CA., 1980.

[30] Aske Plaat, "Research Re: search & Re-search," *PhD Thesis*, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, Thesis Publishers, Amsterdam, The Netherlands, Jun. 20, 1996.

[31] Aske Plaat, Jonathan Schaeffer, Wim Pijls and Arie de Bruin, "Best-First Fixed-Depth Minimax Algorithms," *Artificial Intelligence*, vol. 87, no. 1-2, pp. 255-293, Nov. 1996.

[32] Alexander Reinefeld, "An Improvement to the Scout Tree Search Algorithm," *ICCA Journal*, vol. 6, no. 4, 1983.

[33] R.L. Rivest, "Game Tree Searching by MinMax Approximation," *Artificial Intelligence*, vol. 34, no. 1, pp. 77-96, 1988.

[34] P. Rosenbloom, "A World Championship-level Othello Program," *Artificial Intelligence*, vol. 34, pp. 77-96, 1982.

[35] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM J. Res. Dev.*, vol. 3, 1959.

[36] Jonathan Schaeffer, "Experiments in Search and Knowledge," *PhD thesis*, Department of Computing Science, University of Waterloo, Canada, 1986.

[37] Jonathan Schaeffer, "The History Heuristic and Alpha-Beta Search Enhancements in Practice," *IEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1), pp. 1203-1212, November 1989.

[38] Jonathan Schaeffer and Aske Plaat, "New Advances in Alpha-Beta Searching," *Proceedings of the 1996 ACM 24th annual conference on Computer science*, Feb. 1996.

[39] Claude E. Shannon, "Programming a Computer for Playing Chess," *Philosophical Magazine*, ser. 7, vol. 41, no. 314, , pp. 256-275, 1950.

[40] James H. Slagle and John K. Dixon, "Experiments with some programs that search game trees," *Journal of the ACM*, vol. 2, pp. 189-207, April 1969.

[41] David Slate and Larry Atkin, "Chess 4.5 - the Northwestern University chess program," *Chess Skill in Man and Machine*, P. W. Frey Ed., New York:Springer-Verlag, pp. 82-118, 1977.

[42] Chuen-Tsai Sun and Ming-Da Wu, "Self-Adaptive Genetic Algorithm Learning in Game Playing," *IEEE International Conference on Evolutionary Computation*, vol. 2, pp. 814-818, 1995.

[43] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, pp. 9-44, 1988.

[44] G. J. Tesauro, "TD-Gammon, A Self-teaching Backgammon Program, Achieves Master-level Play," *Neural Computation*, vol. 6, pp. 215-219, 1994.

[45] Ken Thompson, "Computer Chess Strength," *Advances in Computer Chess 3*, M. Clarke (ed.), pp. 55-56, 1982.