Chapter 2
Game Tree Searching and pruning:

In this chapter, we concentrate on game tree searching and pruning aspects. Section 2.1 presents background knowledge on game playing programs: how to build a game tree and how to decide the next move. In section 2.2, we further introduce the most successful refinement of minimax search—the alpha-beta algorithm. Section 2.3 is about some of the most important enhancements to alpha-beta algorithm based on following principles: move ordering, minimal window search, quiescence search and forward pruning. Then we conclude this chapter in Section 2.4. For more information about game tree searching and pruning, we refer to **[11]**.

2.1 Game trees and Minimax Search

Almost all game playing programs use a game tree to represent positions and moves. Nodes represent game positions, and the root node corresponds to the current position of the game. The branches of a node represent the legal moves from the position represented by the node. A node is called a leaf if it doesn't have a successor. Using the rules of the game, we can evaluate a leaf as a win, lose, draw, or a specific score.

But unfortunately the whole game tree size is tremendously huge for almost all interesting games. For example, checkers is 10^{20}, and chess is 10^{40}. The total number of nodes in game tree is roughly W^D, where W stands for the number of possible moves on average for each node, and D is the typical game length. For Amazons game, W is about 479 in randomized computer-game playing (Lida, 1999; Lida and Muller 2000) and D is around 80 [thesis S]. Therefore, no any practical algorithm can manage such a full tree due to lack of time.

One solution of such games is to stop generating the tree at a fixed depth, d, and use an evaluation function to estimate the positions d moves ahead of the root. In this thesis, we will use the term ply, which was first introduced by Samuel **[5]**, to represent the depth of a game tree. The nodes at the deepest layer will be leaves. Typically, the value of a leaf, estimated by the evaluation function, is represented the number in proportion to the chance of winning the game.

Game playing programs depend on game tree search to find the best move for the current position, assuming the best play of the opponent. In a two-person game, two players choose a legal move alternately, and both of them intuitively try to maximize their advantage. Because of this reason, finding the best move for a player must assume the opponent also plays his/her best moves. In other words, if the leaves are evaluated on the viewpoint of player A, player A will always play moves that maximize the value of the resulting position, while the opponent B plays moves that minimize the value of the resulting position. This gives us the MiniMax algorithm.

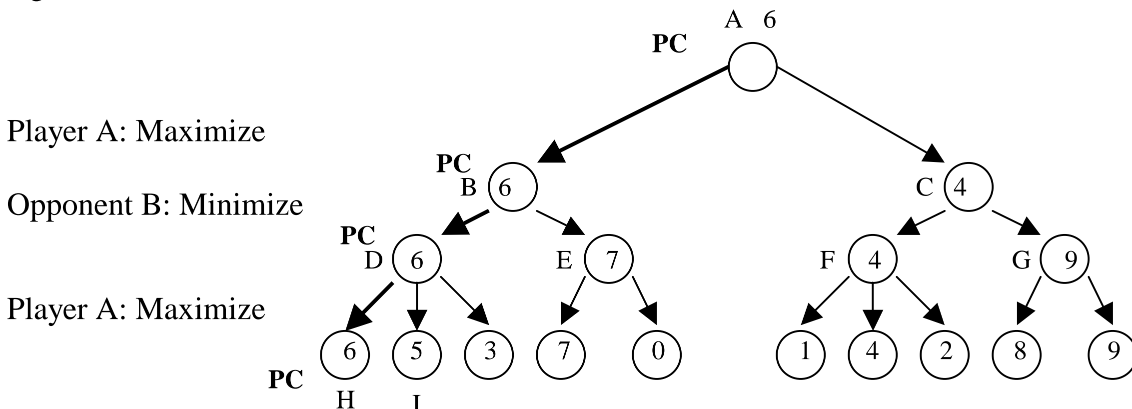

Figure 2.1 MiniMax Search Tree

Figure 2.1 simulates a MiniMax search in a game tree. Every leaf has a corresponding value, which is approximated from player A's viewpoint. When a path is chosen, the value of the child will be passed back to the parent. For example, the value for D is 6, which is the maximum value of its children, while the value for C is 4, which is the minimum value of F and G. In this example, the best sequence of moves found by the maximizing / minimizing procedure is the path through nodes A, B, D and H, which is called the principal continuation **[7]**. The nodes on the path are denoted as PC (principal continuation) nodes.

For simplicity, we can modify the game tree values slightly and use only maximization operations. The trick is to maximize the scores by negating the returned values from the children instead of searching for minimum scores, and estimate the values at leaves from the player's own viewpoint. This is called the NegaMax algorithm. Since most game-playing programs examine large trees, game tree search algorithms are commonly implemented as a depth-first search, which requires memory only linear with the search depth. Figure 2.2 is the pesudocode of NegaMax algorithm, implemented as a depth-first search, and Figure 2.3 illustrates the NegaMax procedure using the same game tree as Figure 2.1.

```
// pos : current board position
// d: search depth
// Search game tree to given depth, and return evaluation of root node.
int NegaMax(pos, d)
{
        if (d=0 || game is over)
                return Eval (pos);              // evaluate leaf position from current player's standpoint
        score = - INFINITY;                     // present return value
        moves = Generate(pos);                  // generate successor moves
        for i =1 to sizeof(moves) do            // look over all moves
        {
                Make(moves[i]);                 // execute current move
                cur = -NegaMax(pos, d-1);       // call other player, and switch sign of returned value
                if (cur > score) score = cur; // compare returned value and score value, update it if necessary
                Undo(moves[i]);                 // retract current move
        }
        return score;
}
```

Figure 2.2 Pseudo Code for NegaMax Algorithm

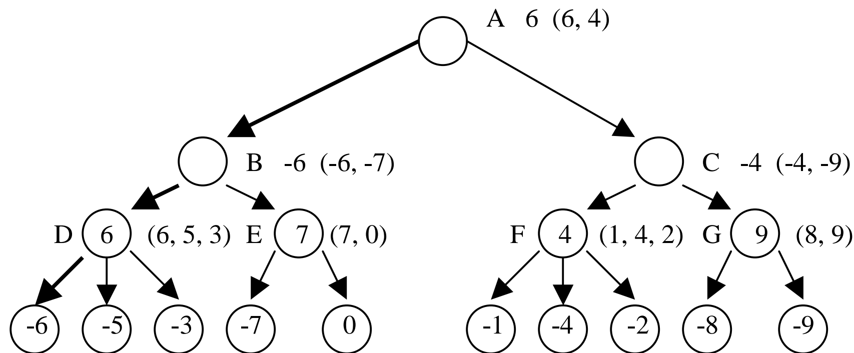

Figure 2.3 NegaMax Search Tree

A NegaMax search has to evaluate every leaf of the game tree. For a uniform tree with exactly W moves at each node, a d-ply NegaMax search will evaluate W^d leaves. This makes a deeper search of a "bushy" tree impossible. Fortunately, a refinement we will talk about in next section, Alpha-Beta pruning, can reduce the amount of work considerably: in the best case, to twice the depth we might reach using NegaMax search in same amount of time.

2.2 The Alpha-Beta Algorithm

As we mentioned in previous section, it is not necessary to explore all the nodes to determine the minimax value for the root. It can be proved that large chunks of the game tree can be pruned away. Knuth and Moore **[8]**

showed that the minimax value of the root can be obtained from a traversal of a subset of the game tree, which has at most $W^{\lceil d/2 \rceil} + W^{\lfloor d/2 \rfloor} + 1$ leaves, if the "best" move is examined first at every node.

McCarthy (1956) was the first to realize that pruning was possible in a minimax search, but the first thorough solution was provided by Brudno (1963) **[9]**. A few years later, Knuth and Moore (1975) **[10]** further refined it and proved its properties.

The success of alpha-beta search is achieved by cutting away uninteresting chunks of the game tree. For example, max(6, min(5, A)) and min(5, max(6, B)) are always equal to 6 and 5 respectively, no matter what values A and B are. Therefore, we can prune the subtrees corresponding to A and B during game tree search. To realize these cut-offs, the alpha-beta algorithm employs a search window (alpha, beta) on the expected value of the tree. Values outside the search window, i.e., smaller than alpha or larger than beta, cannot affect the outcome of the outcome of the search.

Figure 2.4 shows the pseudocode for the alpha-beta algorithm in NegaMax form. In order to return the correct minimax value, alpha-beta search should be invoked with an initial window of alpha = -∞ and beta = ∞.

```
// pos : current board position
// d: search depth
// alpha: lower bound of expected value of the tree
// beta: upper bound of expected value of the tree
// Search game tree to given depth, and return evaluation of root.
int AlphaBeta(pos, d, alpha, beta)
{
        if (d=0 || game is over)
                return Eval (pos);            // evaluate leaf position from current player's standpoint
        score = - INFINITY;                   // preset return value
        moves = Generate(pos);                // generate successor moves
        for i =1 to sizeof(moves) do          // look over all moves
        {
                Make(moves[i]);               // execute current move
                cur = - AlphaBeta(pos, d-1, -beta, -alpha); //call other player, and switch sign of returned value
                if (cur > score) score = cur; // compare returned value and score value, note new best score if necessary
                if (score > alpha) alpha = score;  //adjust the search window
                Undo(moves[i]);               // retract current move
                if (alpha >= beta) return alpha;  // cut off
        }
        return score;
}
```

Figure 2.4 Pseudocode for Alpha-Beta Algorithm

In order to illustrate the alpha-beta search process we discuss an example. Figure2.5 shows the same game tree as Figure 2.3, in which one node (L) and one subtree (the subtree of G) are pruned by alpha-beta search.

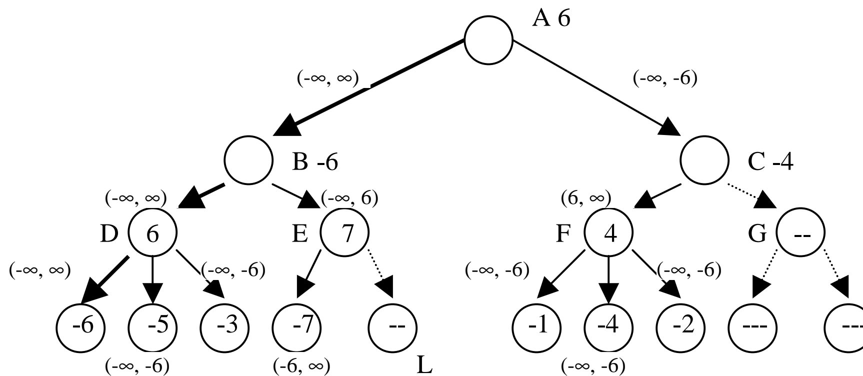

Figure 2.5 Alpha-Beta Search Tree

The leftmost branches are traversed with the initial window (-∞, ∞). After having evaluated the left child of D, the middle child of D is searched with the window (-∞, -6) since the value for D is at least 6. At node E, alpha is updated to -6 after completing the search of left child D. So the search window of the right sibling E is (-∞, 6). After E's left child is visited, the new alpha is adjusted to 7, which is larger than beta, so its right sibling L is cut off. We can also look this procedure as determining the value of min(6, max(7, -L)), or max(-6, - max(7, -L)) in NegaMax form. No matter what value L is, the result we get is always 6, or –6 in NegaMax form.

2.3 Enhancements to the Alpha-Beta Algorithm

In the last three decades, a large number of enhancements to the Alpha-Beta algorithm have been developed. Many of them are used in practice and can dramatically improve the search efficiency. In the section, we will briefly discuss some major Alpha-Beta enhancements, which are based on one or more of the four principles: move ordering, minimal window search, quiescence search and forward pruning.

2.3.1 Move ordering

The efficiency of the alpha-beta algorithm depends on the move search order. For example, if we swap the positions of D, E and F, G in Figure 2.5, then a full tree search will be necessary to determine the value of the root. To maximize the effectiveness of alpha-beta cut-offs, the "best" move should be examined first at every node. Hence many ordering schemes have been developed for ordering moves in a best-to-worst order. Some techniques such as iterative deepening, transposition tables, killer moves and the history heuristic have proved to be quite successful and reliable in many games.

2.3.1.1 Iterative Deepening

Iterative deepening was originally created as a time control mechanism for game tree search. It handles the problem that how we should choose the search depth depends on the amount of time the search will take. A simple fixed depth is inflexible because of the variation in the amount of time the program takes per move. So David Slate and Larry Atkin introduced the notion of iterative deepening **[12]**: start from 1-ply search, repeatedly extend the search by one ply until we run out of time, then report the best move from the previous completed iteration. It seems to waste time since only the result of last search is used. But fortunately, due to the exponential nature of game tree search, the overhead cost of the preliminary D-1 iterations is only a constant fraction of the D-ply search.

Besides providing good control of time, iterative deepening is usually more efficient than an equivalent direct search. The reason is that the results of previous iterations can improve the move ordering of new iteration, which is critical for efficient searching. So compared to the additional cut-offs for the D-ply search because of improved move order, the overhead of iterative deepening is relatively small.

Many techniques have proved to further improve the move order between iterations. In this thesis, we focus on three of them: transposition tables, killer moves and history heuristic.

2.3.1.2 Transposition tables

In practice, interior nodes of game trees are not always distinct. The same position may be re-visited multiple times. Therefore, we can record the information of each sub-tree searched in a transposition table **[12, 15, 16]**. The information saved typically includes the score, the best move, the search depth, and whether the value is an upper bound, a lower bound or an exact value. When an identical position occurs again, the previous result can be reused in two ways:

1) If the previous search is at least the desired depth, then the score corresponding to the position will be retrieved from the table. This score can be used to narrow the search window when it is an upper or lower bound, and returned as a result directly when it is an exact value.

2) Sometimes the previous search is not deep enough. In such a case the best move from the previous search can be retrieved and should be tried first. The new search can have a better move ordering, since the previous best move, with high probability, is also the best for the current depth. This is especially helpful for iterative deepening, where the interior nodes will be re-visited repeatedly.

To minimize access time, the transposition table is typically constructed as a hash table with a hash key generated by the well-known Zobrist method **[13]**.

For a detailed description about how to implement alpha-beta search with transposition tables, we refer to **[11]**.

2.3.1.3 Killer Move Heuristic

The transposition table can be used to suggest a likely candidate for best move when an identical position occurs again. But it can neither order the remaining moves of revisited positions, nor give any information on positions not in the table. So the "killer move" heuristic is frequently used to further improve the move ordering.

The philosophy of the killer move heuristic is that different positions encountered at the same search depth may have similar characters. So a good move in one branch of the game tree is a good bet for another branch at the same depth. The killer heuristic typically includes the following procedures:

1) Maintain killer moves that seem to be causing the most cutoffs at each depth. Every successful cutoff by a non-killer move may cause the replacement of the killer moves.
2) When the same depth in the tree is reached, examine moves at each node to see whether they match the killer moves of the same depth; if so, search these killer moves before other moves are searched.

A more detailed description and empirical analysis of the killer move heuristic can be found in **[17]**.

2.3.1.4 History Heuristic

The history heuristic, which is first introduced by Schaeffer **[18]**, extends the basic idea of the killer move heuristic. As in the killer move heuristic, the history heuristic also uses a move's previous effectiveness as the ordering criterion. But it maintains a history for every legal move instead of only for killer moves. In addition it accumulates and shares previous search information throughout the tree, rather than just among nodes at the same search depth.

Figure 2.6 illustrates how to implement the history heuristic in the alpha-beta algorithm. The bold lines are the part related to the history heuristic. Note that every time a move causes a cutoff or yields the best minimax value, the associated history score is increased. So the score of a move in the history table is in proportion to its history of success.

```
// pos : current board position
// d: search depth
// alpha: lower bound of expected value of the tree
// beta: upper bound of expected value of the tree
// Search game tree to given depth, and return evaluation of root.
int AlphaBeta(pos, d, alpha, beta)
{
        if (d=0 || game is over)
                return Eval (pos);              // evaluate leaf position from current player's standpoint
        score = - INFINITY;                     // preset return value
        moves = Generate(pos);                  // generate successor moves
        for i =1 to sizeof(moves) do  // rating all moves
                rating[i] = HistoryTable[ moves[i] ];
        Sort( moves, rating );                  // sorting moves according to their history scores
        for i =1 to sizeof(moves) do {          // look over all moves
                Make(moves[i]);                 // execute current move
                cur = - AlphaBeta(pos, d-1, -beta, -alpha);     //call other player, and switch sign of returned value
                if (cur > score) {      // compare returned value and score value, note new best score if necessary
```

```
                score = cur;
                bestMove = moves[i]; // update best move if necessary
            }
            if (score > alpha) alpha = score;  //adjust the search window
            Undo(moves[i]);                    // retract current move
            if (alpha >= beta) goto done;  // cut off
        }
    done:
            // update history score
            HistoryTable[bestMove] = HistoryTable[bestMove] + Weight(d);
            return score;
}
```

Figure 2.6 Alpha-Beta Search with History Heuristic

Two questions remain for the implementation of the history heuristic above. The first one is how to map moves to the index of history table. For Amazons, we use a method similar to **[thesis S]**. We divide a move into two separate parts, the queen move and the barricade move. So all queen moves can be fixed in a 10*10*10*10 array, whose index is generated by locations of from-square and to-square. Similarly, a 10*10 array is used for barricade moves.

The other question is how to weight results obtained from different search depths. Two reasons cause us to use 2^d, as suggested by Schaeffer **[18]**, as the weight in our program: first, we should increase a higher score for successful moves on deeper searches, and second, we should increase a higher score for successful moves near the root of the tree.

A well-known problem of the history heuristic is that a good move at the current stage may be overshadowed by its previous bad history. To overcome this problem, we divide all history scores by 2 before every new search.

2.3.2 Minimal Window Search

In the Alpha-Beta procedure, the narrower the search window, the higher the possibility that a cutoff occurs. A search window with alpha = beta –1 is called the minimal window. Since it is the narrowest window possible, many people believe that applying minimal window search can further improve search efficiency. Some alpha-beta refinements such as NegaScout and MTD(f) are derived from minimal window search. For some games with bushy trees, they provide a significant advantage. Since bushy trees are typical for Amazons, minimal window search has the potential of improving its search power.

2.3.2.1 NegaScout / PVS

NegaScout **[19]** and Principal Variation Search (PVS) **[20]** are two similar refinements of alpha-beta using minimal windows. The basic idea behind NegaScout is that most moves after the first will result in cutoffs, so evaluating them precisely is useless. Instead it tries to prove them inferior by searching a minimal alpha-beta window first. So for subtrees that cannot improve the previously computed value, NegaScout is superior to alpha-beta due to the smaller window. However sometimes the move in question is indeed a better choice. In such a case the corresponding subtree must be revisited to compute the precise minimax value.

Figure 2.7 demonstrates NegaScout search procedures. Note that for the leftmost child moves[1], line 9 represents a search with the interval (-beta, -alpha) whereas a minimal window search for the rest of children. If the minimal window search fails, i.e., (cur > score) at line 10, that means the corresponding subtree must be revisited with a more realistic window (-beta, -cur) (line 15) to determine its exact value. The conditions at line 11 show that this re-search can be exempted in only two cases: first, if the search performed at line 9 is identical to actual alpha-beta search, i.e., n=beta, and second, if the search depth is less than 2. In that case NegaScout's search always returns the precise minimax value.

```
// pos : current board position
```

```
        // d: search depth
        // alpha: lower bound of expected value of the tree
        // beta: upper bound of expected value of the tree
        // Search game tree to given depth, and return evaluation of root.
1       int NegaScout(pos, d, alpha, beta) {
2               if (d=0 || game is over)
3                       return Eval (pos);              // evaluate leaf position from current player's standpoint
4               score = - INFINITY;                     // preset return value
5               n = beta;
6               moves = Generate(pos);                  // generate successor moves
7               for i =1 to sizeof(moves) do {          // look over all moves
8                       Make(moves[i]);                 // execute current move
9                       cur = -NegaScout(pos, d-1, -n, -alpha);
10                      if (cur > score) {
11                              if (n = beta) OR (d <= 2)
12                                      score = cur; // compare returned value and score value, note new best score if necessary
13                              else
15                                      score = -NegaScout(pos, d-1, -beta, -cur);
                        }
16                      if (score > alpha) alpha = score;  //adjust the search window
17                      Undo(moves[i]);                 // retract current move
18                      if (alpha >= beta) return alpha;  // cut off
19                      n = alpha + 1;
                }
20              return score;
        }
```

Figure 2.7 Pseudocode for The NegaScout Algorithm

Figure 2.8 illustrates how NegaScout prunes nodes (node N and O) which alpha-beta must visit. After the left subtree has been visited, NegaScout gets the temporary minimax value cur = 6. So the right successor is visited with the minimal window (-7, -6). Then at node F, the leftmost child's value (-9) causes cutoffs of its right successors (at line 18 in Figure 2.7).

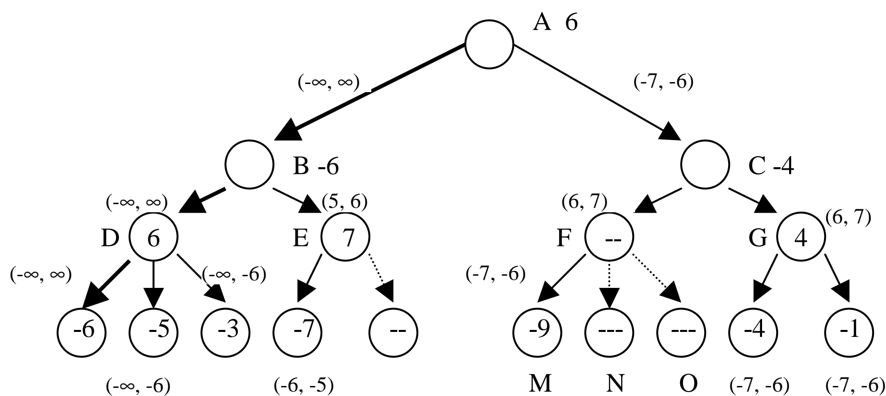

Figure 2.8 Game Tree Showing NegaScount's Superiority

A good move ordering is even more favorable to NegaScout than to alpha-beta. The reason is that the number of re-searches can be dramatically reduced if the moves are sorted in a best-first order. So other move ordering enhancements such as iterative deepening, the killer heuristic, etc, can be expected to give more improvement to NegaScout than to alpha-beta.

When performing a re-search, NegaScout has to traverse the same subtree again. This expensive overhead of extra searches can be prevented by caching previous results. Therefore a transposition table of sufficient size is always preferred in NegaScout.

2.3.2.2 MTD (*f*)

MTD(f) **[21]** is a new alpha-beta refinement which always searches with minimal windows. Minimal window search can cause more cutoffs, but it can only return a bound on the minimax value. To obtain the precise minimax value, MTD(f) may have to search more than once, and use returned bounds to converge toward it.

The general idea of MTD(f) is illustrated by figure 2.9. Note that the score for node "pos" is bound by two values: upper and lower bound. After each AlphaBeta search, the upper or lower bound is updated. When both the upper and lower bound collide at f, i.e. both the minimal window search (f-1, f) and (f, f+1) return f, the minimax score for node "pos" is assured to be f.

```
// pos : current board position
// d: search depth
// f: first guess of expected value of the tree
// Search game tree to given depth, and return evaluation of root.
int MTDF(node pos, int d, int f) {
        int score = f;                          // preset return value
        upperBound = + INFINITY;                // initialize lower and upper bounds of expected
        lowerBound =  - INFINITY;               // evaluation of root
        while (upperBound > lowerBound) do {
                if (score = = lowerBound) then  beta = score + 1;
                else    beta = score;
                score = AlphaBeta(pos, beta – 1, beta, d); // minimal window search
                if (score < beta) then upperBound = score; // re-set lower and upper bounds of expected
                else lowerBound = score;        // evaluation of root
        }
        return score;
}
```

Figure 2.9 Pseudocode for The MTD(f) Algorithm

In MTD(f), the argument f is our first guess of the expected minimal value. The better this first guess is, the fewer minimal searches are needed. In iterative deepening search, the new iteration typically uses the result of the previous iteration as its best guess. For some games, the values found for odd and even search depths vary considerably. In that case feeding MTD(f) its return value of two plies ago, not one, may be even better.

Similar to NegaScout / PVS, MTD(f) also depends on the usage of a transposition table to reduce the overhead of re-searching, so a good transposition table is essential to the performance of MTD(f).

2.3.3 Quiescence Search

A fixed-depth approximate algorithm searches all possible moves to the same depth. At this maximum search depth, the program depends on the evaluation of intermediate positions to estimate their final values. But actually all positions are not equal. Some "quiescent" positions can be assessed accurately. Other positions may have a threat just beyond the program's horizon (maximum search depth), and so cannot be evaluated correctly without further search.

The solution, which is called quiescence search, is increasing the search depth for positions that have potential and should be explored further. For example, in chess positions with potential moves such as capture, promotions or checks, are typically extended by one ply until no threats exist. Although the idea of quiescence search is attractive, it is difficult to find out a good way to provide automatic extensions of non-quiescence positions.

We did not implement quiescence search in our experiments. The first reason is because it is hard to apply the idea of quiescence search in Amazons. In relatively new games such as Amazons, humans are still in the learning stage, so not enough strategic knowledge is known about the game to decide what kind of positions should be extended. Another very important reason is it is very difficult to quantify the effect of the quiescent search. A fair way of comparing different quiescence searches is difficult to find. Therefore we decided not to implement quiescence search for the moment, but try to develop better evaluation functions to avoid threats beyond the horizon.

2.3.4 Forward Pruning

Forward pruning discards some seemingly unpromising branches to reduce the size of the game tree. The depth of the search tree explored strongly influences the strength of the game-playing program. So sometimes exploring the best moves more deeply is better than considering all moves. Many techniques have been developed to perform forward pruning. For example, N-best selective search **[16]** considers only the N-best moves at each node. Other methods, such as ProbCut and Multi-ProbCut, developed by Michael Buro **[22, 23]**, use the result of a shallow search to decide with a prescribed possibility whether the return value of a deep search will fall into the current search window. Although forward pruning can reduce the tree size dramatically, it is also error prone. The major problem is that the best move may be excluded because of its bad evaluation value at a low level in the game tree.

2.3.4.1 N-Best Selective search

To find out a good selection criterion, we need to consider the tradeoff between reducing the possibility of cutting off the best move and increasing the search depth. In our experiments, we implemented N-best selective search in two ways. Both of them use the board evaluation function as the selection criterion. One simply selects N promising moves for each node. The other divides a move into two separate operations: queen-move and arrow-location. For each node, N promising queen-moves are selected first, and then M favorable arrow-locations are determined for each queen-move. Obviously the second approach can further reduce the size of the game tree. On the other hand, the chance of cutting off decisive variations during moves selections may also be increased.

We can order moves based on board evaluation values obtained during move selection. This will give us a very good move ordering and prune more branches during alpha-beta search. For different evaluation functions, the best selective factor N may be different. For example, for faster but worse estimators, a bigger N should be used to increase the possibility that the best move is chosen. To compare the true performance of different evaluation functions, we tune N for every evaluator and chose the value with best performance.

2.3.4.2 ProbCut

The idea of ProbCut is based on the assumption that evaluations obtained from searches of different depths are strongly correlated. So the result V_D' of a shallow search at height d' can be used as a predictor for the result V_D of deeper search at height d. Before a deeper search is performed, the position is first searched to a shallow depth d'. From the return value, we predict the probability that the deeper search will lie outside the current (alpha, beta) window. If it is high, the deeper search is ignored since it is unlikely to affect the final result. Otherwise, the deeper search is performed to obtain the precise result. Note that the effort of shallow search is always negligible compared to a relatively expensive deeper search.

Michael Buro **[22]** suggested that V_D and V_D' are related by a linear expression V_D = a * V_D' + b + e, where the coefficients a and b are real numbers and e is a normally distributed error term with mean 0 and variance σ^2. For stable evaluation functions, we can expect that $a \approx 1$, $b \approx 0$, and σ^2 is small. So we can predict the probability that V_D>=beta from the following equivalences.

$$V_D >= beta \Leftrightarrow a * V_D' + b + e >= beta \Leftrightarrow V_D' >= (-e + beta - b) / a$$
$$\Leftrightarrow V_D' >= ((-e/\sigma)*\sigma + beta - b) / a$$

From the definition of e, we know that $(-e/\sigma)$ is normally distributed with mean 0 and variance 1. So the probability that V_D >= beta is larger than p if and only if V_D' is larger than $((1/\Phi(p)) *\sigma + beta - b) / a$.

Similarly we can deduce that the probability of V_D <= alpha is larger than p if and only if V_D' is larger than (-(1/Φ(p)) *σ + alpha –b) / a. The implementation of the ProbCut extension illustrated in Figure 2.10 is just based on these bounds. To further improve the efficiency of the new algorithm, we use a minimum window for shallow search at D'.

```
// pos : current board position
// d: search depth
// alpha: lower bound of expected value of the tree
// beta: upper bound of expected value of the tree
// Search game tree to given depth, and return evaluation of root.
int AlphaBeta(pos, d, alpha, beta)
{
        …
        // D is the depth of deeper search.
        // D' is the height of shallow search
        // t is the cut threshold which is equal to 1/Φ(p).
        if (d = = D) {
                Retrieve corresponding parameters σ, a and b according to current stage.

                // Is V_D' >= beta likely? If so, cut off and return beta.
                bound = round( (t * σ+ beta – b) / a);
                if (AlphaBeta(pos, D', bound-1, bound) >= bound) return beta;

                // Is V_D' <= alpha likely? If so, cut off and return alpha.
                bound = round( (-t * σ+ alpha – b) / a);
                if (AlphaBeta(pos, D', bound, bound+1) <= bound) return alpha;
        }
        …
}
```

Figure 2.10 Pesudocode of the ProbCut extension

It remains to choose D, D', and the cut threshold t=1/Φ(p) and estimate the corresponding parameters a, b and σ. The search depths D and D' can affect the performance of ProbCut a lot. The difference D-D' is in proportion to the depth of the cut subtree. On the other hand, if the difference is too large, the numbers of cuts will be reduced since the variance of the error will be also large. In the ProbCut implementation in the Othello program LOGISTELLO, the author chooses D'=4 and D=8 experimentally. Then parameters a, b and σ can be estimated for each game stage using evaluation pairs (V_D', V_D) generated by non-selective searches. After that the best t is determined using a tournament between versions of the selective program with different cut thresholds and non-selective version.

The ProbCut extension can increase some game programs' playing strengths considerably. In the Othello game LOGISTELLO, Buro **[22]** reports that the ProbCut-enhanced version defeats the brute-force version with a winning percentage of 74%. This tournament uses 35 balanced opening positions as starting positions and all program versions are with quiescence search and iterative deepening.

2.3.4.3 Multi-ProbCut

Multi-ProbCut **[23]** (or MPC for short) generalizes the ProbCut procedure to prune even more unpromising subtrees by using additional checks and cut thresholds. MPC refines the ProbCut procedure in three ways:

1. MPC allows cutting irrelevant subtrees recursively at several heights instead of only at one specific height.
2. By performing several check searches of increasing depth, MPC can detect extremely bad moves at very shallow check searches.

3. MPC optimizes the cut thresholds separately for different game stages instead of using a constant cut threshold for the whole game.

Figure 2.11 illustrates the implementation of MPC. Note that MPC uses a for loop to perform check searches at several depths. For every check search, a different cut threshold t is used. To avoid search depth degeneration, MPC does not call itself recursively in the check part.

```
// pos : current board position
// d: search depth
// alpha: lower bound of expected value of the tree
// beta: upper bound of expected value of the tree
// Search game tree to given depth, and return evaluation of root.
int MPC(pos, d, alpha, beta)
{
        …
        // MAX_DEPTH maximum height of shallow checks
        // NUM_TRY maximum number of shallow checks
        // t is the cut threshold which is equal to 1/Φ(p).
        if (d <= MAX_DEPTH) {
           for i =1 to NUM_TRY do {
                Retrieve corresponding parameters d’, t, σ, a and b according to current stage height and i.

                // Is V_d’ >= beta likely? If so, cut off and return beta.
                bound = round( (t * σ+ beta – b) / a);
                if (AlphaBeta(pos, d’, bound-1, bound) >= bound) return beta;

                // Is V_d’ <= alpha likely? If so, cut off and return alpha.
                bound = round( (- t * σ+ alpha – b) / a);
                if (AlphaBeta(pos, d’, bound, bound+1) <= bound) return alpha;
           }
        }
        …
}
```

Figure 2.11 Pesudocode of MPC extension

Michael Buro uses the following steps to determine MPC parameters for LOGISTELLO. First, the brute-force evaluations of thousands of example positions up to depth 13 are collected. Then he applies linear regression to this data to estimate the parameters a, b and σ for each disc number, search height and check depth. In the third step, the first check sequence is decided and additional check depth is added to minimize the total running time. Table 1 lists the check depths for different heights. After that two cut thresholds were determined for positions with <36 and >=36 discs respectively using two sets of tournaments.

h	3	4	5	6	7	8	9	10	11	12	13
d1	1	2	1	2	3	4	3	4	3	4	5
d2	--	--	--	--	--	--	5	6	5	--	--

Table 1: Check depths for different heights h

Buro’s experiments in Othello game programs show that MPC outperforms ProbCut. The winning percentage of the MPC version of LOGISTELLO playing against the ProbCut version was 72% in a tournament of 140 games of 30 minutes per move.

2.4 Conclusions

In this chapter we discussed some major algorithms for tree searching and pruning. The performance of these algorithms will vary for different games. One interesting question is how well they perform in the game of Amazon. We will investigate this question experimentally in Chapter 4.

Being able to search game-trees deeper and faster is essential for creating a high quality game-playing program. However, it is not possible to exhaustively search the whole bushy game tree. So we still need a good evaluation function to assess the merits of the game position at maximum depth. For some forward pruning enhancements, such as ProbCut and Multi-ProbCut, stable evaluate function is required. In the next chapter, we will discuss evaluation functions for Amazons.