THE EVOLUTION OF MULAN:

SOME STUDIES IN GAME-TREE PRUNING AND

EVALUATION FUNCTIONS IN THE GAME OF AMAZONS

BY

QIAN LIANG

B.S., Electronic Engineering,
South China University of Technology, 1994

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

December, 2003

ACKNOWLEDGEMENTS

I heartily acknowledge Dr. Cris Moore, my advisor and thesis chair, for his guidance through the long time of working on this study. Without his continuous encouragement, the completion of this thesis would have been simply impossible.

I also thank Dr. Robert Veroff and Dr. Lance Williams, who shared their extensive knowledge with me and give me expert opinions and ideas on the issues involved in this thesis.

My greatest gratitude also goes to Terry Van Belle, who provided a prototype of the game of Amazons and gave me a lot of help with programming and AI game techniques.

To my family, thanks for all the support over the years. And finally to my wife, Ying Ning, you make my life bright.

THE EVOLUTION OF MULAN:

SOME STUDIES IN GAME-TREE PRUNING AND

EVALUATION FUNCTIONS IN THE GAME OF AMAZONS

BY

QIAN LIANG

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

December, 2003

THE EVOLUTION OF MULAN:
SOME STUDIES IN GAME-TREE PRUNING AND EVALUATION FUNCTIONS IN THE GAME OF AMAZONS

by

Qian Liang

B.S., Electronic Engineering, South China University of Technology, 1994

M.S., Computer Science, University of New Mexico, 2003

ABSTRACT

Amazons, as a relatively young game, has caught the attention of AI game programmers. Though Amazons is a simple game for humans, it is difficult for computers. It has 2176 possible moves at the initial position and about five hundred possible moves on average during the game. This makes brute-force search highly impractical. Hence, it is an excellent subject for AI techniques such as selective search and evaluation functions.

In this project, we create a computer Amazons program—Mulan. Using this program, we implemented and tested some popular Alpha-Beta enhancements and forward pruning algorithms. For Chess, a variety of studies have been performed investigating the relative performance of these techniques. But for Amazons, no one seems to know how effective they are. We experimentally answered this question. For studying evaluation functions in Amazons, we implemented and compared three existing

features used for Amazons and four new features we developed. Furthermore, starting with the three features which gained the best results in our experiments, we combined them to form an evaluation function using two methods: 1) linear combinations using tournaments; 2) a pattern classification approach based on Bayesian learning. Though Mulan is created as a test bed for Artificial Intelligence, it has evolved to a strong Amazons playing program. Our test results show that Mulan can substantially beat some strong Amazons playing programs such as Yamazon and Arrow.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

In the past decades, researchers have put a lot of effort into improving the strength of computer program for games such as Othello, Checkers, and Chess -- not just because it is fun, but also because games are useful test beds for Artificial Intelligence. Many search algorithms and pattern recognition methods can be easily tested on game playing programs. After years of hard work, many amazing results have been obtained. One of the most impressive examples is Deep Blue's defeat of Garry Kasparov, the World Chess Champion, in 1997 **[4]**. However, for some games such as Shogi(Japanese Chess) and Go, computers still have considerable room for improvement and humans are still far superior.

Amazons, as a relative young game, has caught the attention of AI game programmers. There are two reasons for this. First, though Amazons is a simple game for humans, it is difficult for computers. Amazons has 2176 possible moves at the initial positions and about five hundred possible moves on average during the game. Since this large branching number makes brute-force search impractical, Amazons is an excellent subject for the study of search algorithms and evaluation functions. Secondly, in Amazons, humans and computers are still in a stage of learning from each other. Since Amazons is a newly invented game, there are not many mature strategies known. Therefore, programmers have to develop implicit strategic knowledge gained from their own playing experience and convert them into rules which can be used by computers. This is a far more challenging and exciting experience than creating game playing programs for more mature games such as Chess.

1.1 What is Amazons?

The game of Amazons is a simple board game invented by Argentinian Walter Zamkauskas in 1988. Amazons shares many characteristics with other games, such as Chess and Go. On one hand, it has a Go-like subgoal of controlling and enlarging territory. On the other hand, players have to keep the mobility of their own pieces as large as possible. This is just the idea which has been used in some other board games like Othello, Checkers, and Chess.

1.1.1 Rules of Amazons

Amazons is a two-person, perfect-information and zero-sum game. Two players compete on a board of size10*10 and each side has four queens (called amazons). Figure 1.1 shows the initial position of the game. The rules can be summarized as the following:

(1) Two players alternate choosing an amazon of their color to move. In this thesis we assume the player with white pieces moves first.

(2) The amazons move like chess queens except they cannot capture. After each move, an amazon shoots an arrow to burn off a square reachable by second queen move from its position.

(3) Any amazon or arrow cannot move to or pass over a square that is burnt off or occupied by another queen.

(4) The last player who is able to complete a move is the winner.

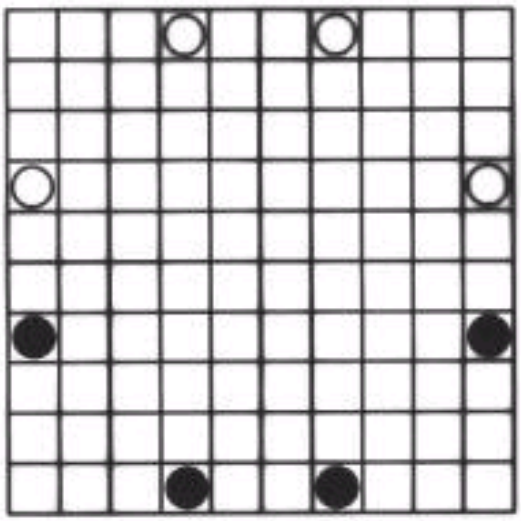

Figure 1.1 The initial position of Amazons

1.1.2 Notation

In this thesis, the columns on the board are labeled by letters (a to j) and rows by number (1 to 10). A move is denoted as initial_position – end_position(shoot). For example, d1-d6 (i6) means “amazon at d1 moves to d6 and then shoots an arrow to i6”.

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 and D is around 80 **[33]**. 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.

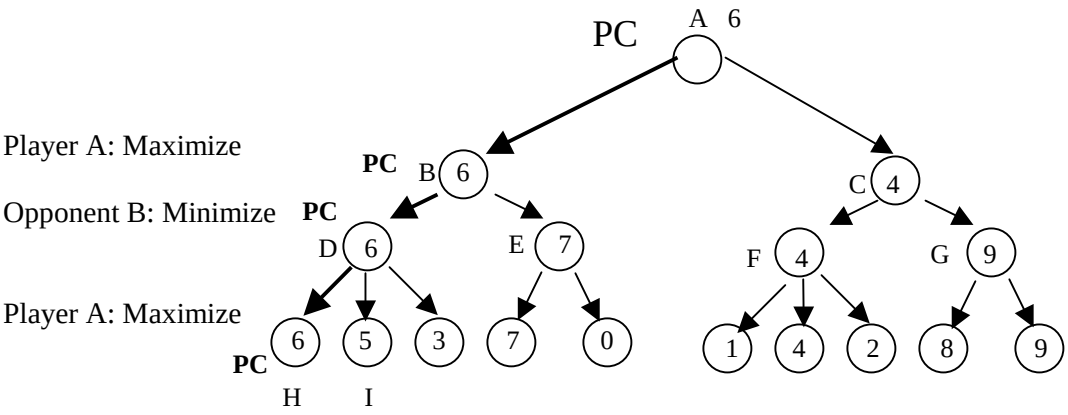

Figure 2.1 MiniMax Search Tree

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
        // call other player, and switch sign of returned value
        cur = -NegaMax(pos, d-1);
        // compare returned value and score value, update it if necessary
        if (cur > score) score = cur;
        Undo(moves[i]);                    // retract current move
    }
        return score;
}
```

Figure 2.2 Pseudo Code for NegaMax Algorithm

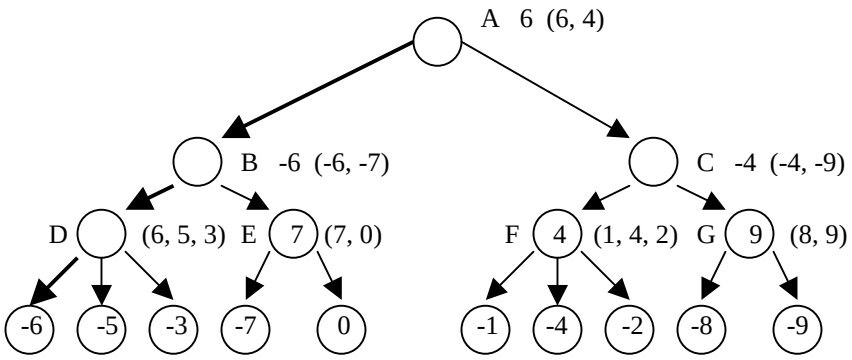

Figure 2.3 NegaMax Search Tree

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
       //call other player, and switch sign of returned value
       cur = - AlphaBeta(pos, d-1, -beta, -alpha);
       //compare returned value and score value, note new best score if necessary
       if (cur > score) score = cur;
       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.

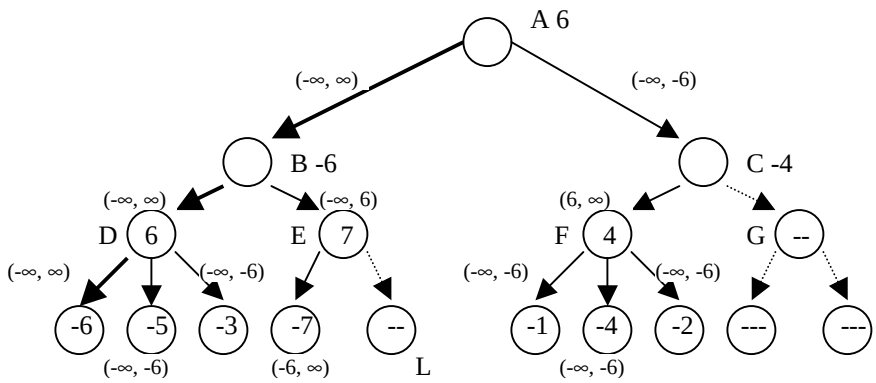

Figure 2.5 Alpha-Beta Search Tree

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);
    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
        //call other player, and switch sign of returned value
        cur = - AlphaBeta(pos, d-1, -beta, -alpha);
```

```
        //compare returned value and score value, note new best score if necessary
        if (cur > score) {
            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 **[33]**. 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);
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;
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).

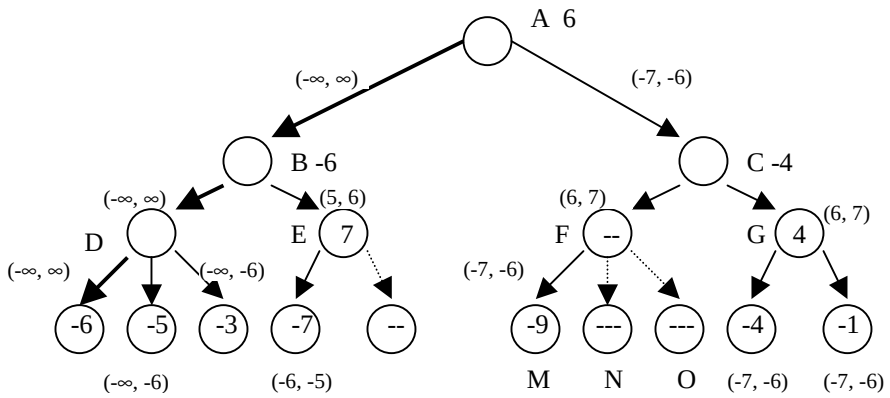

Figure 2.8 Game Tree Showing NegaScount's Superiority

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
    // initialize lower and upper bounds of expected
    upperBound = + INFINITY;
```

```
    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
        // re-set lower and upper bounds of expected
        if (score < beta) then upperBound = score;
        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 ≈ 1, b ≈ 0, and σ^2 is small. So we can predict the probability that V_D>=beta from the following equivalences.

V_D >= beta ⇔ a * V_D' + b + e >= beta ⇔ V_D' >= (-e + beta –b) / a

⇔ V_D' >= ((-e/σ)*σ + beta –b) / a

From the definition of e, we know that (-e/σ) 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/Φ(p)) *σ + 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 $t=1/\Phi(p)$ 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 2.1 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 2.1 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.

CHAPTER 3 EVALUATION FUNCTIONS

3.1 Introduction

Most successful game-playing programs apply heuristic evaluation functions at terminal nodes to estimate the probability that the player to move will win. Typically a successful evaluation function is the combination, e.g., a weighted sum, of a number of distinct features. Each feature measures a property of the board position. Thus constructing evaluation functions has two phases:

1) selecting good features.
2) combining them appropriately to obtain a single numerical value.

Selecting features is important and difficult. We have to avoid too few features as well as redundant ones. It also requires both expert game knowledge and programming skill because of the well-known tradeoff between the complexity of the evaluation function and the number of positions we can evaluate in a given time: a more accurate evaluation function might actually result in inferior play if it takes too long to calculate. Feature combination is also critical and very unintuitive. We need to not only establish a balance among diversified strategies but also be aware of the interactions between related features.

In this chapter, we will introduce the existing features used for Amazons and some automatic evaluation function construction methods.

3.2 Evaluation Features of Amazons

Recently three different evaluation features have been proposed for Amazons. These are Mobility **[24]**, Territory **[25]** and Territory-and-Mobility (TM) **[26]**. As

explained in Chapter 1, Amazons has characteristics of both Chess and Go. The idea of the Mobility feature is from Chess. It focuses on the number of possible moves since the more possible moves a player has, the less likely it is that the player will run out of legal moves. Similar to Go, the territory feature is based on the concept of controlling more squares, since controlling more squares can provide more space for the pieces to move. The feature Territory-and-Mobility, suggested by Hashimoto **[26]**, combines the merits of Mobility and Territory in a way which we describe below.

3.2.1 Mobility

In the game of Amazons, the last player who is able to complete a move wins the game, so having more possible moves than the opponent is a key factor for winning. The mobility of a player is defined as the sum of the possible moves of all his/her queens, and the mobility feature **[24]** is calculated by subtracting the opponent's mobility from the player's.

The mobility feature may be useful for the opening stage where the territory classification is not clear enough. But in the mid-game and endgame, the board is divided into several battlefields. So controlling your own territories and invading the opponent's territories are more important than enlarging your mobility.

Another problem is that if we maximize mobility, all the player's pieces tend to stay in the middle of the largest territory **[26]**. So a human player can block the computer's pieces in the center of the board and occupy all four corners easily.

The main advantage of this feature is its evaluation speed. Typically the evaluator can evaluate the mobility feature faster than the territory feature.

3.2.2 Territory

Amazons is a game with Go-like characteristics. The number of possible moves is huge in the opening phase and decreases gradually. From the mid-game phase on, the board is separated into several battlefields. Therefore, keeping your own territory as large as possible is critical for winning the game.

Kotani **[25]** introduces the Minimum Stone Ply (MSP) function to evaluate the territory feature of Amazons. In the MSP function, a square belongs to the player who can reach it faster with one of his/her pieces without counting arrows. If both players can reach a square in the same minimum number of moves, this square is neutral, i.e., it doesn't belong to either player. Figure 3.1 shows the procedure of the MSP function. In our program, we used bitmap operations, which are designed and coded by Terry, to implement this algorithm efficiently.

```
// pos : current board position
// evaluate the territory feature for the board position "pos" and return the result.
int MSP(pos)
{
    blackSquares = 0;
    whiteSquares = 0;
    for all empty squares x do
    {
        blackStonePly = the minimum number of moves that a black piece needs to arrive at x;
        whiteStonePly = the minimum number of moves that a white piece needs to arrive at x;
        if (blackStonePly < whiteStonePly)
            blackSquares = blackSquares + 1;
        else
            whiteSquares = whiteSquares + 1;
```

```
        }
        if (white's turn to play)
            return (whiteSquares – blackSquares);
        else
            return (blackSquares – whiteSquares);
    }
```

Figure 3.1 The procedure of the MSP function

The territory feature can correctly evaluate enclosed and almost-enclosed areas. Its performance is reasonably good in the endgame and in well-balanced situations. However, the territory feature assumes queens can go all directions and defend against attackers approaching from different sides at the same time. Therefore, in unbalanced situations such as where one queen is facing several, this feature evaluates the board too optimistically.

3.2.3 Territory-and-Mobility

Hashimoto et al. **[26]** combined the concepts of Mobility and Territory to build a new evaluation function, called Territory-and-Mobility, or TM.

Typically the TM feature is evaluated via three steps: (1) Use the MSP function to evaluate the each player's territory; (2) Count mobility in each player's territory; (3) Sum the results of (1) and (2) using a specific weight. Figure 3.2 demonstrates the procedure of the TM function.

```
// pos : current board position
// evaluate the TM feature for the board position "pos" and return the result.
```

```
int TM(pos)
{
    blackPoints = 0;
    whitePoints = 0;
    for all empty squares x do
    {
       blackStonePly = the minimum number of moves that a black piece needs to arrive at x;
       whiteStonePly = the minimum number of moves that a white piece needs to arrive at x;
        blackMob = the number of black queens which can arrive at square x in one move.
        whiteMob = the number of white queens which can arrive at square x in one move.
        if (blackStonePly < whiteStonePly)
            blackPoints = blackPoints + weight + blackMob;
        else
            whitePoints = whitePoints + weight + whiteMob;
    }
    if (white's turn to play)
        return (whitePoints – blackPoints);
    else
        return (blackPoints – whitePoints);
}
```

Figure 3.2 The procedure of the TM function

Hashimoto et al. **[26]** believes that adding mobility to the territory feature allows the program to place all four queens in a coordinated way. In addition, the calculation of territory becomes more precise by adding the mobility scores in Step 3. After testing various values, they suggest that setting the weight to 4 gives the best performance.

3.3 Automatic Evaluation Function Construction

We now describe how we combine features to create an evaluation function. Traditionally, an evaluation function is a linear combination of a number of features (F1, F2, …, Fn), i.e., a weighted sum:

Eval = C1 * F1 + C2 * F2 + … + Cn * Fn

where the coefficients (C1, C2, … , Cn) are either guessed by the implementer or found by some optimization process.

But there are two problems with this method. First, it is difficult for humans to estimate these coefficients correctly, since they don't use game tree search and evaluation functions. That was also the initial motivation for Samuel to propose ways to tune weights automatically in **[5, 29]**. Furthermore, this method assumes that no correlations or redundancies between features exist. This assumption is clearly false since almost every pair of features is correlated to some degree. To solve this problem, Lee et al **[27]** present a pattern classification approach.

3.3.1 Samuel's work on automatic feature combination

Arthur Samuel, a novice Checkers player, is one of the earliest and most important researchers on Checkers learning programs. From 1947 to 1967, he proposed and experimented on many different methods of machine learning. In the next two sections, we introduce the two most important ones: (1) linear evaluation learning through self-play, and (2) nonlinear evaluation learning through book moves.

3.3.1.1 Linear evaluation learning through self-play

In linear evaluation learning **[5]**, Samuel tuned the coefficients by arranging two copies of the Checkers programs Alpha and Beta to play against each other. At the beginning, Alpha and Beta are identical. The only difference is that Beta keeps its weights fixed while Alpha continuously tunes its weights during the experiment. If Alpha beats Beta, Beta adopts Alpha's evaluation on the next round of experiments. Otherwise, Alpha tries other ways to tune its weight. Sometimes manual intervention is necessary if the learning process gets stuck. When Alpha consistently defeats Beta, its evaluation function is considered as the stabilized final version. After this learning procedure, the final program can play a reasonably good game of checkers.

As one of the first machine learning examples, Samuel's procedure is a milestone in automatic evaluation function construction. But as Lee et al pointed out in **[27]**, it is based on several incorrect assumptions. First, it incorrectly assumes that all features in the evaluation function are independent, so it cannot capture the relationships between features. Second, it assumes that all the inaccurate evaluations are caused by the evaluation function, while sometimes the real reason is the limited horizon of the search. Third, it assumes that when the evaluation function is overly optimistic, the problem must come from positive features. This is clearly incorrect because it may be due to negative features are not negative enough. Finally, it assumes that Alpha's evaluation must be better than Beta's if player Alpha beats Beta. But when both two programs are naive, a win may be the result of luck or the opponent's errors.

3.3.1.2 Nonlinear evaluation learning through book moves

In order to cope with these incorrect assumptions, Samuel introduces a new procedure to construct nonlinear evaluation functions through book moves in **[29]**.

To handle nonlinear interactions among features, Samuel devised signature tables. These are multi-dimensional tables where each dimension is indexed by the value of some feature. For each game position, the table cell indexed by its feature values contains the corresponding evaluation value. Samuel collected 24 features for the game of Checkers. Obviously applying this scheme directly could result in an impractically large table. Samuel dealt with this problem using two methods. First, he organized the tables using a three-level hierarchical organization. At the first level, each table combines four features, and only interactions between those four features are considered. Each table in level one or two produces a value to index into tables in the higher level. Furthermore, Samuel restricted the feature values to (-1,0,1) or (-2,-1,0,1,2). This results in a final configuration with a reasonable number of cells.

To avoid the incorrect assumptions in self-play, Samuel used book moves to train these cells. He collected a “book” of board positions and the corresponding moves played by human masters. For each cell, he counted how many times the corresponding feature combination was chosen in book moves, A, and how many times the corresponding combination was a legal move but was not chosen in book moves, D. The cell value was then evaluated as (A-D)/(A+D).

According to Samuel’s experiments, signature table learning through book moves substantially outperformed the self-play learning procedure. But there are a number of new problems with this approach. First, this approach is based on a problematic

assumption that no other moves are as good as or better than book moves. Second, restricting feature values causes some smoothness problems. Finally, the higher-level signature tables cannot handle inter-table correlations. So the correlated features must be arranged into the same group at the first level.

As a result, Samuel's procedure needs excessive human tuning. The implementer has to put a lot of effort in determining how to restrict the feature values and arranging the structure of signature tables. This is undesirable since the learning procedure may be affected by human errors. In the next section, we will introduce another approach to evaluation function learning, which is exempt from weary and dangerous human intervention.

3.3.2 A pattern classification approach based on Bayesian learning

In this section, we will introduce a pattern classification approach to evaluation function learning, which was first introduced by Lee et al in [27]. Unlike Samuel's approaches, it is based on Bayesian learning, and can be easily applied to different domains. First, we give a brief introduction for Bayesian learning. After that, its applications to evaluation function learning will be described in detail.

3.3.2.1 Bayesian learning

Bayesian reasoning provides a way to encode probability relationships among variables of interests. Over the last decade, Bayesian learning has become a popular representation for learning uncertain knowledge in expert systems **[29].** A Bayesian model can be used to learn causal relationships, and hence can be used to gain

understanding about a problem domain and to predict the consequences of intervention; it is ideal for combining prior knowledge (which often comes in causal form) with data.

Bayes' theorem is the cornerstone of Bayesian learning methods. Before defining Bayes' theorem, let's first introduce some basic notation. h is a hypothesis, for instance, the hypothesis that a board is a wining or losing position. The *Prior probability* $p(h)$ is the initial probability that h holds. $p(h)$ may give us information about the chance that h is a correct hypothesis. x is a new data set. $p(x)$ is the probability that x will be observed. $p(x \mid h)$ denotes the probability of observing data x given h. From a large number of training data sets, p(x) and p(x|h) can be estimated correctly. In Bayesian learning, our object is to classify the new instance x, i.e., calculate the *posteriori probability* $p(h \mid x)$. Bayes' theorem provides a direct way to do this.

$$p(h \mid x) = \frac{p(x \mid h)\, p(h)}{p(x)} \tag{1}$$

Sometimes we are only interested in finding the maximum a posteriori (MAP) hypothesis h_{MAP} from the various candidates in a set of hypotheses, H. The MAP hypothesis can be determined by using Bayes' theorem to calculate the posterior probability of each candidate hypothesis. To simply the calculation, we usually ignore the term p(x) because it is independent of h. So we have

$$h_{MAP} = \arg\max_{h \in H} p(x \mid h)\, p(h) \tag{2}$$

Applying log to both sides of equation (2), we get the discriminant function $g_h(x)$:

$$g_h(x) = \log h_{MAP} = \log p(x \mid h) + \log p(h) \tag{3}$$

During the training stage, a number of training samples are evaluated. Each training sample is evaluated to a feature vector and classified to a hypothesis. For a hypothesis *h*, the corresponding mean vector μ_h and variance-covariance matrix V_h can be estimated as the following:

$$\mu_h = \frac{1}{N_h}\sum_{i=1}^{N_h} x_i \tag{4}$$

$$V_h = \frac{1}{N_h}\sum_{i=1}^{N_h} (x_i - \mu_h)(x_i - \mu_h)^T \tag{5}$$

where N_h denotes the total number of training samples for hypothesis h, and x_i denotes the feature vector of the i^{th} sample in *h* hypothesis.

Let us assume the distribution of the features is multivariate normal. Then the density function p(x|h) can be written as a function of μ_h and V_h as:

$$p(x \mid h) = \frac{1}{2\pi^{-N/2} \mid V_h \mid^{1/2}} \exp\{-\frac{1}{2}(x-\mu_h)^T V_h^{-1}(x-\mu_h)\} \tag{6}$$

Substituting (6) into (3), we have:

$$g_h(x) = -\frac{1}{2}(x-\mu_h)^T V_h^{-1}(x-\mu_h) - \frac{1}{2}N\log 2\pi - \frac{1}{2}\log \mid V_h \mid + \log p(h) \tag{7}$$

Furthermore, the posterior probability p(h|x) can be derived by normalizing $g_h(x)$:

$$\log p(h \mid x) = \frac{\exp g_h(x)}{\sum \exp g(x)} \tag{8}$$

3.3.2.2 Evaluation function learning

Based on Bayesian Learning, Lee et al introduce a new learning algorithm for evaluation function construction in **[27]**. They use the game of Othello as their test

domain and dramatically improve an Othello program BILL2.0 that has performed at the world championship level. Like Bayesian learning, this approach also includes two stages: training and recognition.

In the training stage, a database of labeled training positions is required. Lee et al took these positions from actual games generated from BILL 2.0's self-play. All positions of the winning player were labeled as winning positions, and all positions of the losing player as losing positions. Of course, positions might be mislabeled, e.g., that Bill lost from a position in which an optimal player would win. First, although BILL2.0 was still using a linear evaluation function, it had been carefully tuned and it was a world-championship-level player. Furthermore, the initial position of each game was generated by 20 random moves, after which the player that is ahead usually goes on to win the game.

To obtain the training positions, two copies of BILL2.0 are used to play with each other from initial positions. The initial positions were generated by 20 random moves. After that, each side played the remaining 40 half-moves in 15 minutes and the last 15 moves were played using perfect endgame search. A total of 3000 games were played and their positions were recorded as training data.

For each training board, the four features were calculated and represented as a feature vector. Then, the mean vectors and covariance matrices for both categories, winning and losing, could be estimated. In the game of Othello, different strategies should be used for different stages of the game. Lee et al defined a stage as the number of discs on the board. For a stage with N discs, they used training positions with N-2, N-1, N, N+1, and N+2 discs to generate a corresponding discriminant function. Using a series

of slowly varying discriminant functions, the evaluation function provides a fine measure of game positions for different stages.

To recognize a new position, we first compute the features and combine them to form the feature vector, x. Then we can evaluate the position by substituting x into the final evaluation function. From (7), we know that the discriminant functions for winning and losing position are:

$$g_{win}(x) = -\frac{1}{2}(x-\mu_{win})^T V_{win}^{-1}(x-\mu_{win}) - \frac{1}{2}N\log 2\pi - \frac{1}{2}\log|V_{win}| + \log p(win) \quad (9)$$

$$g_{loss}(x) = -\frac{1}{2}(x-\mu_{loss})^T V_{loss}^{-1}(x-\mu_{loss}) - \frac{1}{2}N\log 2\pi - \frac{1}{2}\log|V_{loss}| + \log p(loss) \quad (10)$$

Since the final evaluation function should determine the possibility of winning, it is defined as:

$$g(x) = \frac{P_{win}}{P_{loss}} = g_{win} - g_{loss} \quad (11)$$

Substituting (9) and (10) into (11), Lee et al. give the final evaluation function:

$$g(x) = -\frac{1}{2}(x-\mu_{win})^T V_{win}^{-1}(x-\mu_{win}) - \frac{1}{2}\log|V_{win}| + \frac{1}{2}(x-\mu_{loss})^T V_{loss}^{-1}(x-\mu_{loss}) + \frac{1}{2}\log|V_{loss}| \quad (12)$$

Note $\log p(win)$ and $\log p(loss)$ are cancelled here. The reason is that the possibilities of winning and losing are equal since we use the same program play against each other.

Sometimes we might want to print out some useful search information for humans. In that case, the probability of winning is a more meaningful. It can be derived from g(x) as the following:

$$\frac{p(win \mid x)}{p(win \mid x) + p(loss \mid x)} = \frac{\exp g_{win}(x)}{\exp g_{win}(x) + \exp g_{loss}(x)}$$

$$= \frac{\exp\{g_{win}(x) - g_{loss}(x)\}}{\exp\{g_{win}(x) - g_{loss}(x)\} + 1}$$

$$= \frac{\exp g(x)}{\exp g(x) + 1} \qquad (13)$$

Using the final evaluation function (12) and the same four features as BILL2.0, a new version, BILL 3.0, was created. As we mentioned above, BILL2.0, which is using a linear evaluation function, already played at world-championship level. But surprisingly, dramatic improvements could still be observed in BILL3.0. Lee et al used three ways to comparing BILL2.0 and BILL3.0: (1) having the two versions of the program play against each other, (2) comparing their solutions for endgame problems with the known solution, (3) having the two versions of program play against human experts. In all three experiments, the results favored BILL3.0, which use the new evaluation function constructed by Bayesian learning.

Although Lee et al focus on applying Bayesian learning to the game of Othello, their approach may also be applicable to other games and search-based applications. In next chapter, we will apply Bayesian learning to our Amazons program—Mulan.

3.4 Conclusion

In this chapter, we first described three popular evaluation features of the game of Amazons. Furthermore, some automatic evaluation function construction methods and their applications in Checkers and Othello were introduced.

There are two interesting questions left in this chapter. The first one is how important these features are in position evaluation. The second one is whether we can apply Bayesian learning to the game of Amazons effectively. In the next chapter, we will use a series of experiments to answer these two questions.

CHAPTER 4 EXPERIMENTAL RESULTS

In this section, we discuss how we implemented and compared some popular Alpha-Beta enhancements and forward pruning algorithms using Mulan. Furthermore, we implemented and compared three existing features used for Amazons and four new features we developed. Starting with three features which gained the best results in our experiments, we combined them to form an evaluation function using two methods: 1) linear combination using tournaments; 2) a pattern classification approach based on Bayesian learning.

4.1 Performances of Alpha-Beta Enhancements

In chapter 2, we discussed several important Alpha-Beta enhancements. All of them have been proved to be useful Alpha-Beta refinements. However, their performance is game- and implementation-dependent. For Chess, a variety of studies have been performed investigating their relative performance. But for Amazons, no one seems to know how effective they are. In this section, we will try different combinations of enhancements to find out which combination performs best in Amazons.

4.1.1 Implementation

The implementations of these enhancements vary for different games. For each enhancement, many strategies have been developed over the years. Finding the best strategy for Amazons is a challenge. Here we briefly describe our implementations in Mulan.

a) Transposition Tables

There are some interesting thoughts about implementing a Transposition Table for Amazons. As we mentioned in Chapter 2, a previous search and evaluation can be reused in two ways: 1) If the previous search on the position is at least the desired depth, then the corresponding score can be used to narrow the search window. 2) When the previous search is not deep enough, the corresponding best move can be retrieved and tried first.

In Amazons, the number of "burnt" squares increases by one every move, so a given position can only occur at a fixed depth in the game. This dramatically reduces the occurrence of Transpositions, so most of the performance gained by using the Transposition Table is a result of reordering the moves. This is especially helpful for iterative deepening, where the interior nodes will be re-visited repeatedly. But as the previous searches of re-visited positions are always not deep enough, keeping the previous upper and lower bound in the Transposition Table is useless. So when no minimal window search enhancement is applied, we can simplify the Transposition Table and only save the best move from the previous search for the position.

On the other hand, when a minimal window search enhancement is used, re-visits of positions at the same depth is common. To reduce the overhead of re-searching, we need to use the previous search score to narrow the search window. So in this case, storing the previous upper and lower bound makes sense. Table 4.1 shows the useful information in each entry of the hash table. Figure 4.1 contains pseudo-code showing how we implement the Transposition Table in Alpha-Beta search.

pos	the information of the position, used to distinct different positions with the same hash key.
depth	the effective subtree height of the previous search, depth < 0 if the entry is empty.
upper	upper bound of the subtree
lower	lower bound of the subtree
move	the best move determined

Table 4.1 Fields for the Transposition Table Entry

```
// 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) {
    retrieve(pos, depth, upper, lower, move);
    orgAlpha = alpha;
    if (depth >= d) {
        if (upper <= alpha || upper == lower)
            return upper;
        if (lower >= beta)
            return lower;
        if (lower > alpha)
            orgAlpha = alpha = lower;
        if (upper < beta)
            beta = upper;
    }
    if (d=0 || game is over)
```

```
    return Eval (pos);    // evaluate leaf position from current player's standpoint
score = - INFINITY;                    // preset return value

Make(move);                            // try the previous best move first.
//call other player, and switch sign of returned value
cur = - AlphaBeta(pos, d-1, -beta, -alpha);
// compare returned value and score value, note new best score if necessary
if (cur > score) score = cur;
if (score > alpha) alpha = score;      //adjust the search window
Undo(moves[i]);                        // retract current move
if (alpha >= beta) goto End;           // cut off

moves = Generate(pos);                 // generate successor moves
for i =1 to sizeof(moves) do  {        // look over all moves
    Make(moves[i]);                    // execute current move
    //call other player, and switch sign of returned value
    cur = - AlphaBeta(pos, d-1, -beta, -alpha);
    //compare returned value and score value, note new best score if necessary
    if (cur > score) score = cur;
    if (score > alpha) alpha = score;  //adjust the search window
    Undo(moves[i]);                    // retract current move
    if (alpha >= beta) goto End;       // cut off
}
End:
    upper = +INFINITY;
    lower = -INFINITY;
    if (max <= orgAlpha)
        upper = max;
    if (max > orgAlpha && max < beta)
        upper = lower = max;
    if (max >= beta)
```

```
            lower = max;
        store(pos, depth, upper, lower, move);
    return score;
}
```

Figure 4.1 Alpha-Beta Search with Transposition Tables

The most popular hash key generation function for Transposition Tables is the well-known Zobrist method **[13]**. However, this method didn't consider the increasing burnt squares in Amazons. In Mulan, we use the method proposed by Terry Van Belle to represent the board and generate the hash key. We represent the Amazons game board as two 10*10 bit matrices called "piece" and "type". The representation is as follows:

piece	type	square value
0	0	empty
0	1	burnt
1	0	white
1	1	black

For simplicity and speed, we use an unsigned integer array whose size is 10 to represent each 10*10 bit matrix and use bit operations to access the specific bit. The index of the position is produced by the following function:

```
for (t = piece, p = type; t < piece+10; t++, p++) {
        hash <<= 1;
        hash ^= (*p | (*t << 10));
}
```

Now that hash key generation function enables rapid access to the Transposition Table, we need to choose a good replacement scheme to handle another problem - collisions.

Unlike a normal hash table, saving all visited positions in the Transposition Table is impossible, so our Transposition Table always replaces the old position in the table when a collision occurs. This is because most re-visited positions are met locally, the new positions is more useful than the old one. This simple replacement scheme also simplifies the table access to a single probe. Many elaborate schemes have been proposed and tested **[14].** Some of them do improve the table usage, but the cost of the increased complexity undermines the improvement of total performance. So in our experiment, we didn't repeat these schemes.

As we mentioned before, trying the best move determined by the previous iteration first can reduce the game search tree significantly. One interesting question is that if we try both the best and the second best move first, can we further improve the efficiency of the algorithm? To answer this question, we tried another type of Transposition Table which keeps both the best and the second best moves determined by the previous iteration.

b) Killer Heuristic

We use the moves saved while determining the *principal continuation* to serve as a killer list. Here the term *principal continuation* denotes the best sequence of moves found by the game tree search procedure. During the development of the *principal continuation*, we can gather the PRINC array. A move enters the PRINC array only if it can improve the provisional score of a node. Hence, the PRINC array maintains moves

that seem to cause the most cutoffs at each depth. The detailed description about how to generate and update the PRINC array can be found in **[17]**.

Many strategies have been developed to find killer moves in the PRINC array. The strategy used in Mulan is to compare all the relevant moves in PRINC. For example, suppose the PRINC array is as follows:

mv1	*mv2*	*mv3*	*mv4*	*mv5*	*mv6*
	mv7	*mv8*	*mv9*	*mv10*	*mv11*
		mv12	*mv13*	*mv14*	*mv15*
			mv16	*mv17*	*mv18*
				mv19	*mv20*
					mv21

and the search has arrived at a node *n* in ply 2. Then all moves generated at *n* will be compared with elements mv3, mv5, mv8, mv10, mv12, mv14, mv17 and mv19. When matches are found, the moves will be reordered and the matching moves will be moved to the top of the move list.

To use the information obtained from iterative deepening, we keep two copies of the PRINC array: PrincCur and PrincPre. Until the current search iteration is as deep as the previous one, we use the information gotten from the previous iteration to sort the moves.

c) History Heuristic

We illustrated how to implement the history heuristic in the alpha-beta algorithm in Chapter 2. But two questions remain for the implementation of the history heuristic in Amzons. The first one is how to map moves to the index in the history table. For

Amazons, we use a method similar to **[33]**. We divide a move into two separate parts, the queen move and the arrow 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 arrow moves.

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

d) Minimal Window Search

The "grain" of the evaluation value affects the effect of minimal window search significantly. This is even more obvious for MTD(*f*). The coarser the "grain", the less passes MTD(*f*) has to make. Unfortunately we use a very fine-grained of the evaluation function in Mulan, where the value of a position ranges from –1000 to +1000. With this evaluation function, MTD(*f*) has to use hundreds of passes to converge to the minimax value. This is impractical even with a Transposition Table to reduce the overhead. To reduce the passes to a reasonable number, we increase the step size of every pass to 50 in our implementation.

In MTD(*f*), the first guess of the expected minimal value is related to the size of the search tree. The better this first guess is, the fewer minimal searches are needed. In

iterative deepening, the new iteration typically uses the result of the previous iteration as its best guess. But in Amazons, the values found at odd and even search depths vary considerably. So if the return value of two plies before is available, we use it as the first guess of the current iteration. Otherwise we use 0 instead.

4.1.2 Experimental design

The popular way to assess various enhancements are trying all possible combinations of the enhancements on a set of positions and comparing their relative performance. For example, the standard Bratko-Kopec positions **[30]**, which have an average branch factor of 34, have been used extensively as test positions for chess programs. But for Amazons, people still don't know what the best test positions are yet.

In our experiment, we use Mulan to generate the test positions through self-play. First, we selected 30 best positions from the opening book. Then for each selected initial position, we set two copies of Mulan to play with one another. The search depth of each stage is set to the most "practical" number, i.e. the deepest depth Mulan can search in the tournament. For each intermediate game position met during self-play, we try all possible combinations of the enhancements and compare their relative performance.

There are two obvious advantages of our approach. First, the test positions used in the experiments are practical positions for Amazons that actually arise in play. Second, we compare the performance of different enhancements at the most useful search depth. Other researchers tend to use the same set of test positions to test the performance of different depths of search. In Amazons, a deep search in the early stage is impossible because of the huge branching factor. On the other hand, comparing the performance of a

shallow search in the middle and endgame stages is of no use because we can search them in a deeper depth in practical. Our approach avoided these problems by comparing the performance at the deepest depth the search can finish in the tournament. Table 4.2 lists the corresponding search depth used for different stages of the game (given as the number of move).

step	1-14	15-44	45-49	50-54	55-64	65-70
depth	2	3	4	5	6	7

Table 4.2 Search depths for different stages

Two measures are used to compare search algorithm performance. One is the amount of CPU time required for the search. Practically, this seems to be the most useful measure. However, the execution time of the various enhancements is machine- and implementation-dependent. The other measure used is the number of leaf nodes (also called bottom positions) visited (LN). It assumes evaluating these nodes is usually more expensive than evaluating the interior nodes. In Amazons, this assumption is true since the relatively expensive evaluation function is applied only to the bottom positions. We use this measurement to assess various enhancements in theory.

4.1.3 Experiment results

a) Notation

In this chapter, we use one or two capitals to denote each enhancement. Typically it is the first letter of the name of the enhancement. So MTD(f), NegaScout, AlphaBeta,

Iterative Deepening, Transposition Table, Transposition Table with two best moves, Killer Heuristic and History Heuristic are denoted as M, N, A, I, T, TT, K and H respectively. Similarly, the different combinations of enhancements can be denoted as a series of capitals. For example, MTKH means the MTD(f) algorithm with Transposition Table, Killer Heuristic and History Heuristic.

b) Results and Interpretations

Our experiments compared the performance of MTD(f), NegaScout, and AlphaBeta with different combinations of Iterative Deepening, Transposition Table, Killer heuristic and History Heuristic.

Iterative deepening is a very good time control mechanism for game tree search, but it wastes some amount of time on previous iterations. Figure 4.2 and Table 4.3 compare the relative time and leaf nodes visited between iterative deepening and direct search if we don't use previous iteration to order moves. In fact, we found that even without using the results of previous searches to order moves, the amount of time/leaf nodes in previous searches is only a small fraction of the total time/leaf nodes. So we used Iterative Deepening in all our experiments.

Figure 4.3 and Table 4.4 show that the average width (branching factor) of the game tree in Amazons decreases quickly as the game proceeds. At step 5, A and AITKH generate search trees whose nodes have an average width of 386.97 and 322.70 respectively. But at step 70, the average widths drop to 7.95 and 7.41 respectively, since the number of possible moves decreases when more and more squares are burnt. This also explains why we can't use the same set of test positions to test the performance of different search depths.

Tables 4.5 - 4.10 and Figure 4.4 - 4.9 illustrate the effect of different enhancement combinations in AlphaBeta, MTD(*f*) and NegaScout. These graphs show that:

a) The percentage improvements contributed by all enhancement combinations seem to be highest in the middle game. There are two reasons for this. The first is that at the beginning of the game, the search depth that can be reached is too shallow because of the huge branching factor. According to the result in Chess **[11]**, the improvement of enhancements is typically not very obvious for a shallow search. A second reason is that the branching factor is too small in the endgame of Amazons and the enhancements typically work better for a bushy game tree.

b) The relative performances of CPU time and leaf nodes visited are very similar. This indicates the practical enhancement's performance is identical with the theory one in Amazons.

c) Figure 4.5 and Table 4.6 indicate that the number of leaf nodes visited decreases slightly when we use a Transposition Table that keeps the two best moves instead of only the best move. This implies that keeping two best moves in the Transposition Table can further improve the move order. But the cost of the increased complexity undermines the improvement of whole performance. From Figure 4.5 and Table 4.5, we can observe that the original Transposition Table is superior to the one with two best moves in total CPU time.

d) The killer moves have a good performance in Amazons. Actually, it seems to be the best enhancement we tested. The killer heuristic is a popular AlphaBeta

enhancement, but its effectiveness has been questioned. In Chess, Hyatt observed as much as 80% reductions **[32]** in search time, whereas Gillogly claimed no benefits **[31]**. In our experiment, we observed up to 90% reductions in the middle game of Amazons.

e) Sometimes using two enhancements can only provide a small improvement compared with a single enhancement. For example, Figures 4.4 and 4.5 show that ATK does not significantly improve the performance of AK. There are two reasons for this. First, different enhancements may improve the move order in a similar way. For example, part of the improvement of the Killer heuristic comes from trying the previous iteration's best moves first, which is also the major reason to use the Transposition Table. Another reason is that when applying several enhancements at the same time, the enhancement applied later may reorder the move sequence suggested by previous enhancements. For example, the History Heuristic may indicate some moves we should try first while Killer Heuristic says other moves are more important. So we have to decide which enhancement should have higher priority. Our priority sequence is Transposition Table, then the Killer heuristic, and finally the History heuristic.

f) In all three algorithms, AlphaBeta, MTD(f) and NegaScout, the combination of TKH provides the biggest overall improvement.

As we discussed in Chapter 2, MTD(f) and NegaScout are actually two kinds of Alpha-Beta variants derived from minimal window search. Figure 4.10 and 4.11 compare

the performance of these two variants with the standard AlphaBeta algorithm. These two graphs show:

a) The performance of MTD(f) is unstable. One possible reason is that MTD(f) is sensitive to the "grain" of the evaluation value. We adjusted the step size of MTD(f) to reduce this problem. But in Amazons, the range of the evaluation function can differ dramatically in different positions. Although we adjust the step size of MTD(f) for best overall performance, it is impossible to find a step size which works well in all positions, especially when the positions are from different stages of the game.

b) The performance of NegaScout is very stable and slightly superior to the performance of the standard AlphaBeta search.

Finally, we find out the combination of NegaScout, Iterative Deepening, Transposition Table, Killer Heuristic and History Heuristic is the combination of enhancements with the best overall performance in Amazons.

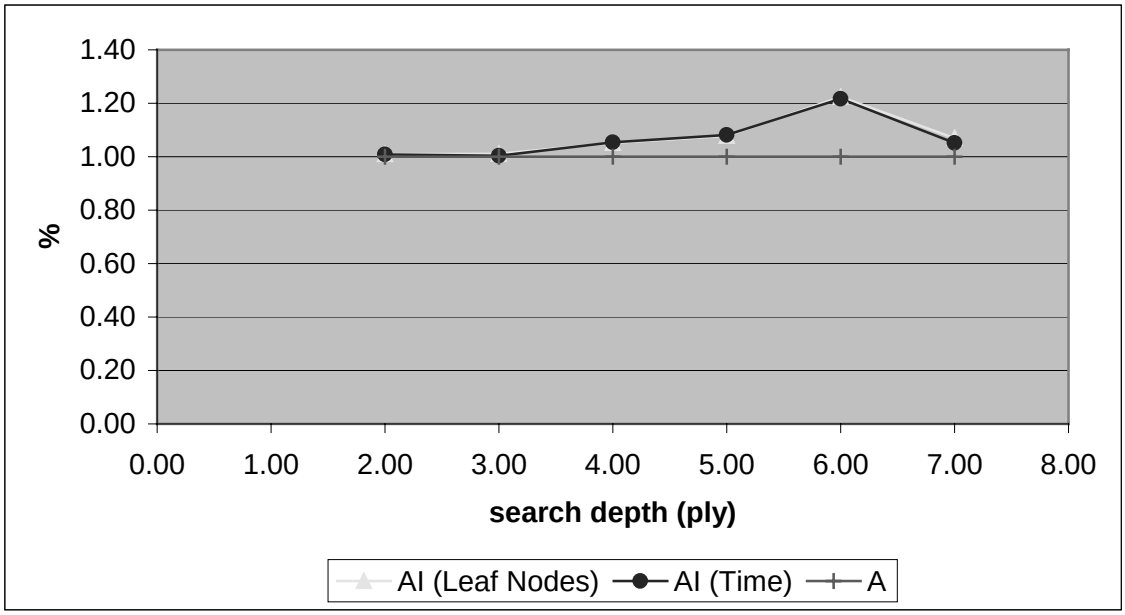

Figure 4.2 Comparison of Iterative Deepening and direct search

	A (Leaf Nodes)	AI (Leaf Nodes)	AI/A (Leaf Nodes)	A (Time)	AI (Time)	AI/A (Time)
Step5 (ply 2)	149743	151260	1.01	2.16	2.18	1.01
Step25 (ply 3)	1675468	1693933	1.01	33.90	34.03	1.00
Step45 (ply 4)	1127314	1186022	1.05	25.65	27.02	1.05
Step54 (ply 5)	1076793	1160188	1.08	23.88	25.81	1.08
Step56 (ply 6)	1541367	1886262	1.22	33.24	40.45	1.22
Step70 (ply 7)	2009315	2149421	1.07	33.59	35.33	1.05

Table 4.3 Comparison of Iterative Deepening and direct search

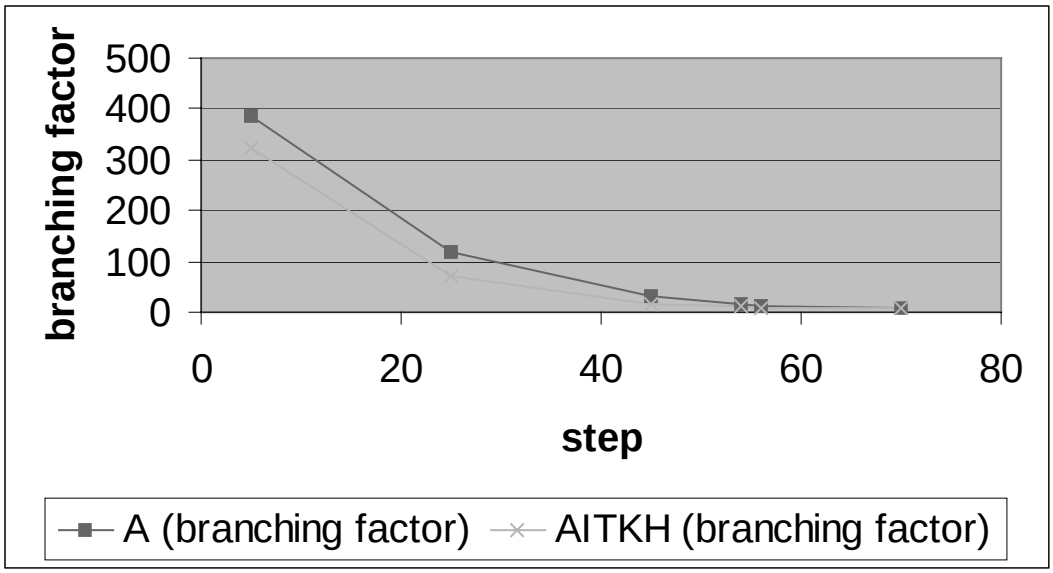

Figure 4.3 branching factor of different stages of Amazons

	A (Leaf Nodes Visited)	A (branching factor)	AITKH (Leaf Nodes Visited)	AITKH (branching factor)
Step5 (ply 2)	149743	386.97	104137	322.70
Step25 (ply 3)	1675468	118.77	347858	70.33
Step45 (ply 4)	1127314	32.58	64231	15.92
Step54 (ply 5)	1076793	16.09	155348	10.92
Step56 (ply 6)	1541367	10.75	436634	8.71
Step70 (ply 7)	2009315	7.95	1228785	7.41

Table 4.4 branching factor of different stages of Amazons

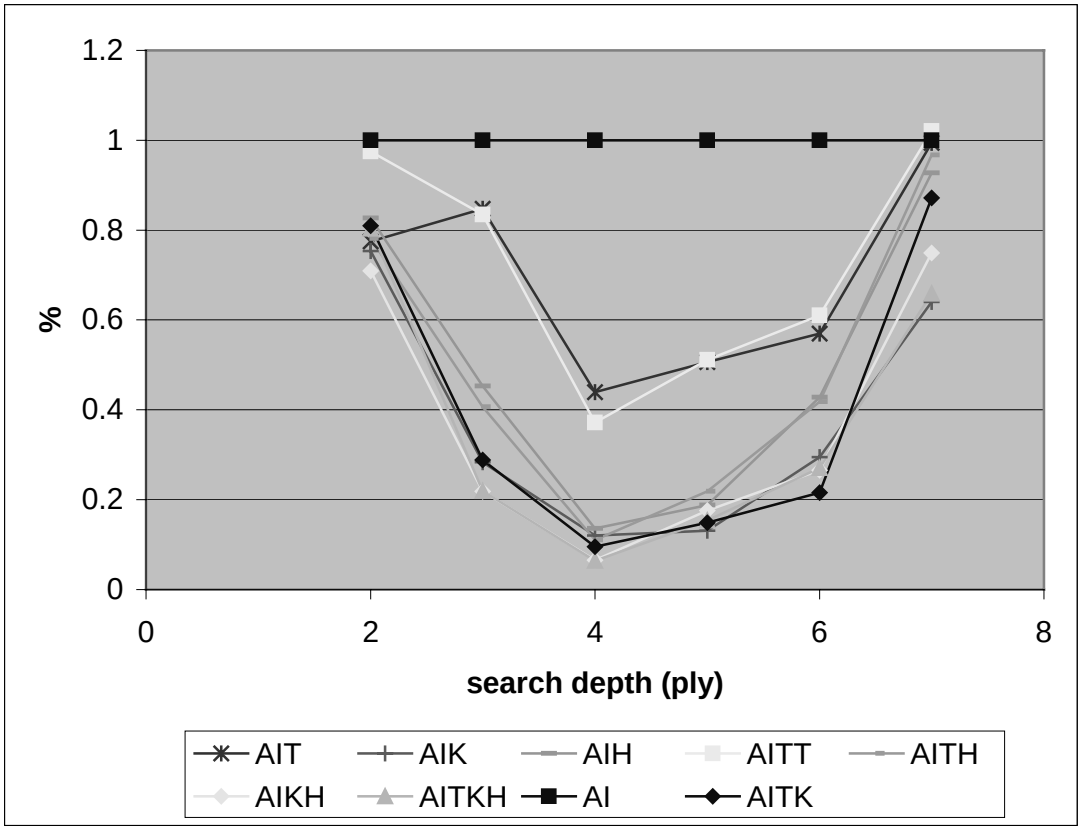

Figure 4.4 Time Comparison of Enhancements in Alpha-Beta

	A	**AI**	**AIT**	**AIK**	**AIH**	**AITT**
Step5 (ply 2)	149743	151260	112117	112074	122657	98638
Step25 (ply 3)	1675468	1693933	1400047	482021	758272	1378788
Step45 (ply 4)	1127314	1186022	476202	107867	146210	402873
Step54 (ply 5)	1076793	1160188	563325	172109	232256	569329
Step56 (ply 6)	1541367	1886262	1127316	613480	910796	1190450
Step70 (ply 7)	2009315	2149421	2086921	1232130	1991488	2142105

Table 4.5 Time Comparison of Enhancements in Alpha-Beta

	AITK	AITH	AIKH	AITKH
Step5 (ply 2)	112074	104351	104137	104137
Step25 (ply 3)	479419	647056	352565	347858
Step45 (ply 4)	99375	116292	67262	64231
Step54 (ply 5)	161854	214561	157291	155348
Step56 (ply 6)	437070	719316	454454	436634
Step70 (ply 7)	1230427	1897285	1247210	1228785

Table 4.5 (cont.)

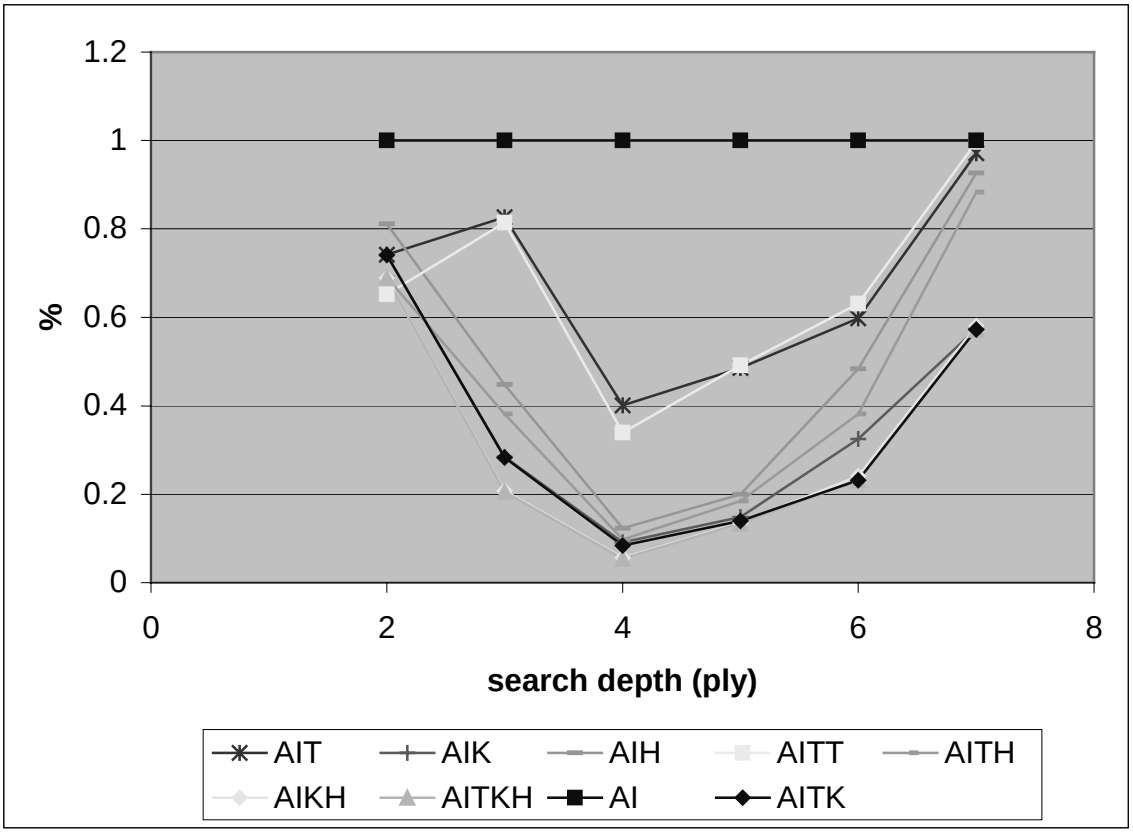

Figure 4.5 Leaf Nodes Visited Comparison of Enhancements in Alpha-Beta

	A	AI	AIT	AIK	AIH	AITT
Step5 (ply 2)	2.16	2.18	1.69	1.64	1.80	2.13
Step25 (ply 3)	33.90	34.03	28.83	9.65	15.42	28.40
Step45 (ply 4)	25.65	27.02	11.87	3.25	3.67	10.04
Step54 (ply 5)	23.88	25.81	13.07	3.39	4.85	13.21
Step56 (ply 6)	33.24	40.45	23.03	11.93	17.32	24.69
Step70 (ply 7)	33.59	35.33	35.12	22.63	32.75	36.06

Table 4.6 Leaf Nodes Visited Comparison of Enhancements in Alpha-Beta

	AITK	AITH	AIKH	AITKH
Step5 (ply 2)	1.77	1.70	1.55	1.75
Step25 (ply 3)	9.82	13.84	7.44	7.47
Step45 (ply 4)	2.57	2.99	1.77	1.73
Step54 (ply 5)	3.85	5.64	4.54	3.98
Step56 (ply 6)	8.73	16.92	10.74	10.95
Step70 (ply 7)	30.79	34.18	26.48	23.27

Table 4.1.6 (cont.)

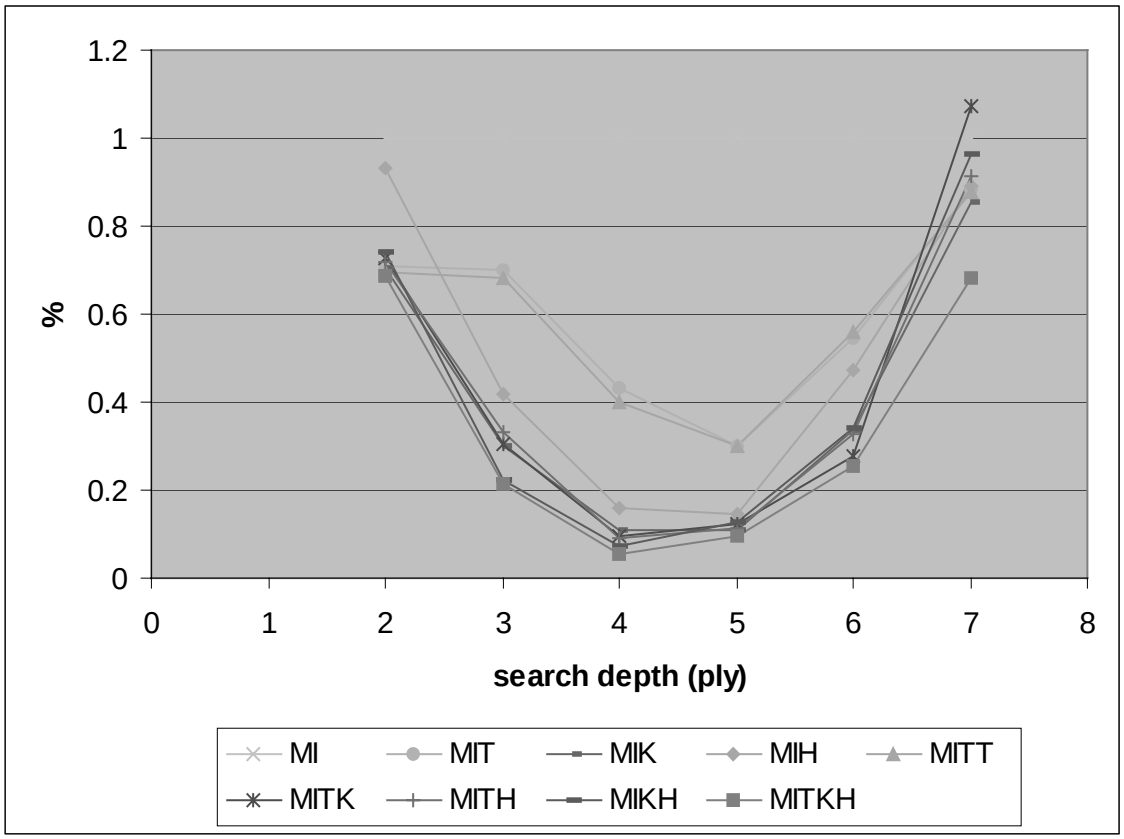

Figure 4.6 Time Comparison of Enhancements in MTD(*f*)

	AI	MI	MIT	MIK	MIH
Step5 (ply 2)	149743	179280	179280	183513	243824
Step25 (ply 3)	1675468	1869969	1246754	558288	772309
Step45 (ply 4)	1127314	1266889	500658	129719	179579
Step54 (ply 5)	1076793	1313729	390552	175417	222649
Step56 (ply 6)	1541367	1627051	852528	578470	854116
Step70 (ply 7)	2009315	1452229	1241227	1242644	1244619

Table 4.7 Time Comparison of Enhancements in MTD(*f*)

	MITT	MITK	MITH	MIKH
Step5 (ply 2)	174241	179255	171411	175418
Step25 (ply 3)	1245520	516197	581063	397394
Step45 (ply 4)	467254	107886	98395	81700
Step54 (ply 5)	386729	150801	167412	158936
Step56 (ply 6)	891421	394445	531011	587735
Step70 (ply 7)	1241250	1065398	953918	1258596

Table 4.7 (cont.)

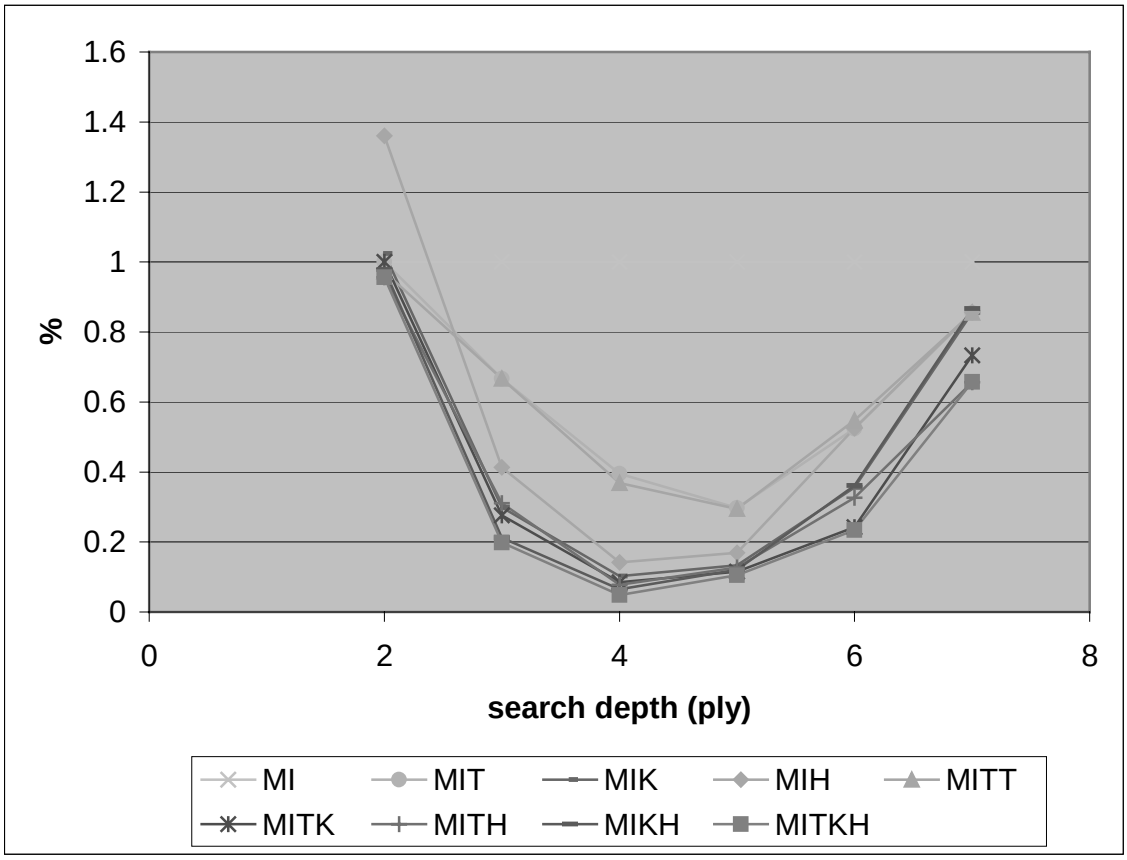

Figure 4.7 Leaf Nodes Visited Comparison of Enhancements in MTD(*f*)

	AI	MI	MIT	MIK	MIH	MITT
Step5 (ply 2)	2.18	3.97	2.82	2.80	3.71	2.77
Step25 (ply 3)	34.03	37.62	26.28	11.22	15.67	25.67
Step45 (ply 4)	27.02	28.79	12.39	3.18	4.53	11.49
Step54 (ply 5)	25.81	31.24	9.41	3.47	4.59	9.34
Step56 (ply 6)	40.45	34.59	18.88	11.62	16.31	19.26
Step70 (ply 7)	35.33	24.47	21.56	20.86	21.75	21.49

Table 4.8 Leaf Nodes Visited Comparison of Enhancements in MTD(*f*)

	MITK	MITH	MIKH	MITKH
Step5 (ply 2)	2.89	2.85	2.95	2.73
Step25 (ply 3)	11.38	12.51	8.41	8.01
Step45 (ply 4)	2.73	2.61	2.12	1.61
Step54 (ply 5)	3.83	3.53	3.96	3.01
Step56 (ply 6)	9.57	11.26	11.87	8.78
Step70 (ply 7)	26.29	22.39	23.62	16.72

Table 4.8 (cont.)

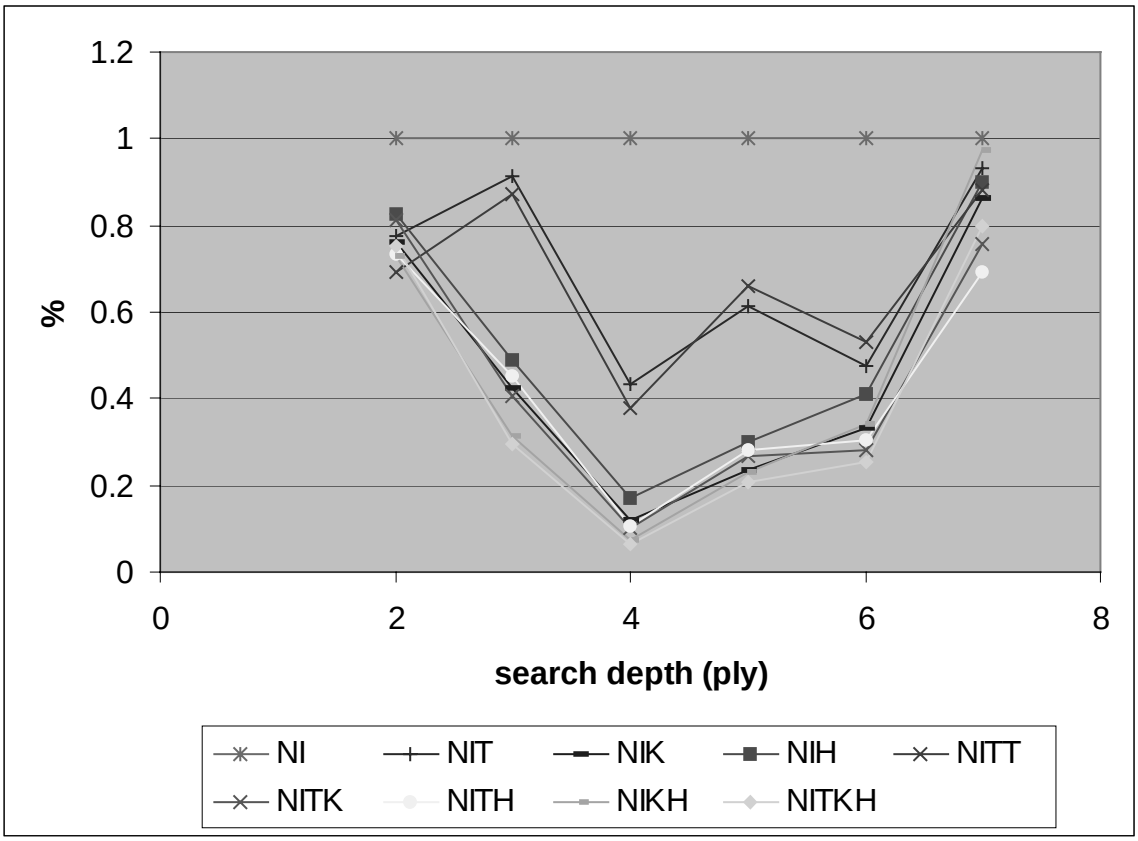

Figure 4.8 Time Comparison of Enhancements in NegaScout

	AI	NI	NIT	NIK	NIH
Step5 (ply 2)	149743	151260	112117	112074	122657
Step25 (ply 3)	1675468	1035430	893798	412064	503388
Step45 (ply 4)	1127314	1026527	405539	114937	158432
Step54 (ply 5)	1076793	565390	331563	162472	200586
Step56 (ply 6)	1541367	1602767	753652	556758	705649
Step70 (ply 7)	2009315	1419900	1215287	1232113	1261702

Table 4.9 Time Comparison of Enhancements in NegaScout

	NITT	NITK	NITH	NIKH	NITKH
Step5 (ply 2)	98638	112074	104351	104137	104137
Step25 (ply 3)	883401	406703	436909	298870	293637
Step45 (ply 4)	329653	94275	94438	67032	54032
Step54 (ply 5)	331422	146135	149789	151816	134591
Step56 (ply 6)	796063	389368	462122	465399	370562
Step70 (ply 7)	1214670	1050597	955325	1242209	947140

Table 4.9 (cont.)

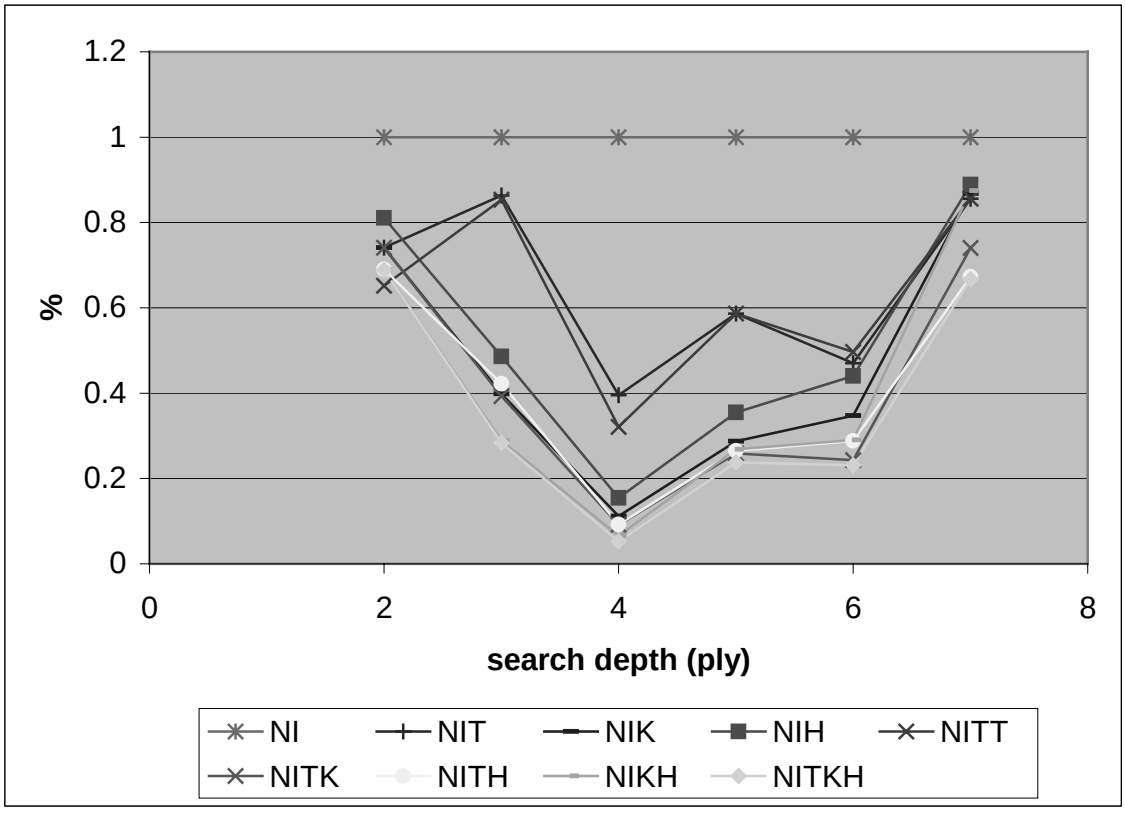

Figure 4.9 Leaf Nodes Visited Comparison of Enhancements in NegaScout

	AI	NI	NIT	NIK	NIH
Step5 (ply 2)	2.18	2.18	1.69	1.66	1.81
Step25 (ply 3)	34.03	20.87	19.05	8.90	10.24
Step45 (ply 4)	27.02	23.29	10.11	2.83	3.98
Step54 (ply 5)	25.81	13.78	8.43	3.21	4.15
Step56 (ply 6)	40.45	32.49	15.50	10.78	13.34
Step70 (ply 7)	35.33	23.97	22.32	20.69	21.53

Table 4.10 Leaf Nodes Visited Comparison of Enhancements in NegaScout

	NITT	NITK	NITH	NIKH	NITKH
Step5 (ply 2)	1.51	1.78	1.60	1.60	1.64
Step25 (ply 3)	18.25	8.50	9.42	6.55	6.13
Step45 (ply 4)	8.82	2.40	2.52	1.73	1.45
Step54 (ply 5)	9.10	3.71	3.86	3.18	2.88
Step56 (ply 6)	17.32	9.08	9.92	11.04	8.19
Step70 (ply 7)	21.11	18.13	16.58	23.32	19.19

Table 4.10 (cont.)

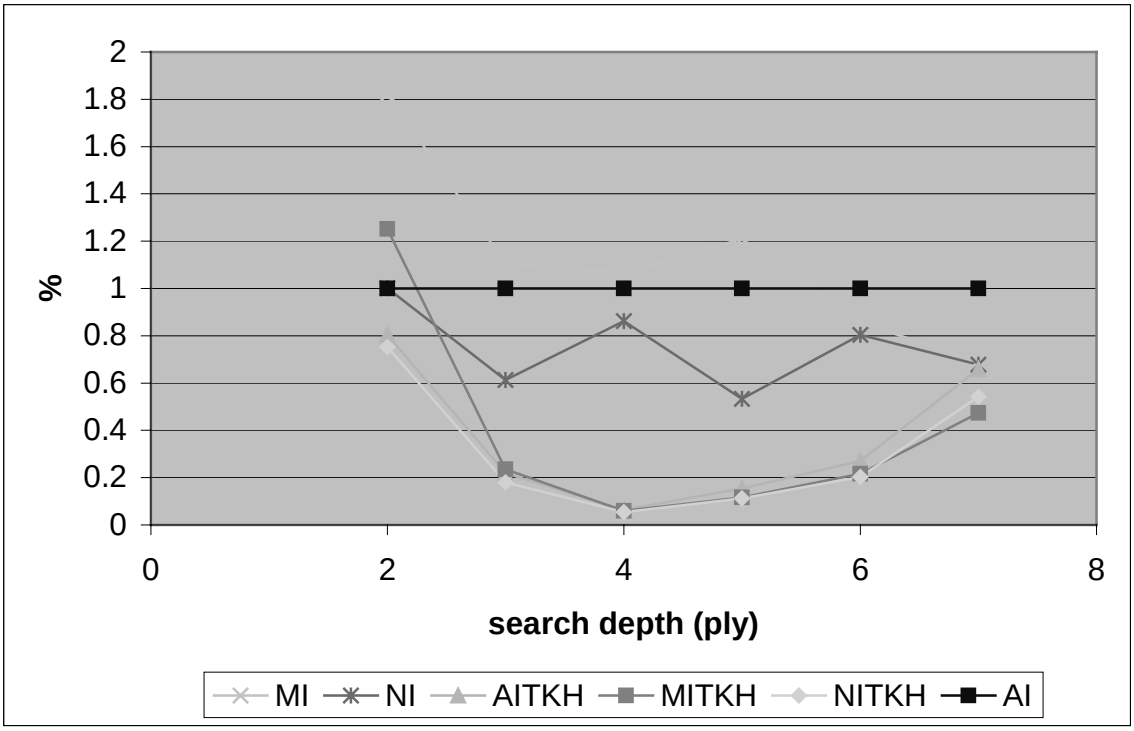

Figure 4.10 Time Comparison of Minimal Window Search

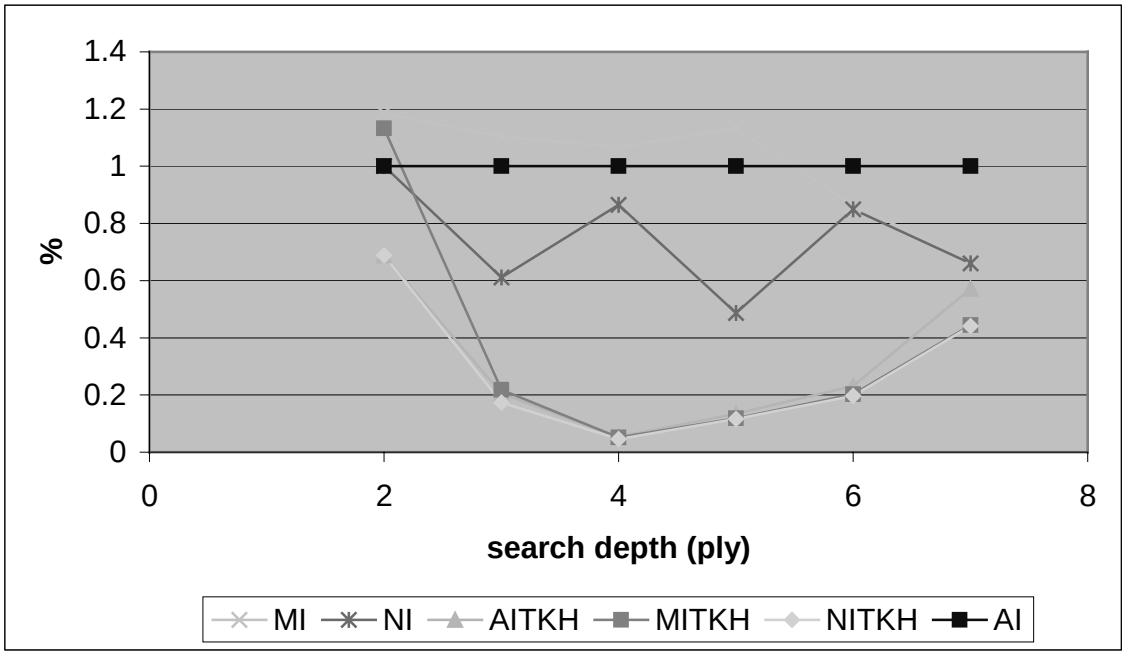

Figure 4.11 Leaf Nodes Visited Comparison of Minimal Window Search

4.2 N-Best Selective Search versus Alpha-Beta with Enhancements

As we discuss in Chapter 2, N-Best selective search **[16]** considers only the N-Best moves at each node, so it can explore the bushy game tree more deeply. This heuristic is error prone, since sometimes the best move is excluded because of the horizon effect. Despite this disadvantage, N-Best is still a popular search technique, since it can reduce the tree size dramatically, making it even smaller than the minimal Alpha-Beta game tree.

4.2.1 Implementation

To reduce the possibility of cutting off the best move, we use the board evaluation values as the criterion to select promising positions. Two kinds of N-best selective search are implemented. One divides a move into two separated operations: queen-move and arrow-move. For each node, N promising queen-moves are selected first, and then M favorable arrow-locations are determined for each queen-move. The other one simply selects F promising moves (combinations of queen-move and arrow-move). for each node. In both of them, we sort the selected moves in descending order of evaluation values.

4.2.2 Experiment design

We use two experiments to find out which kind of N-best selective search is better and what is the best selective factor.

The first one is based on comparing the performance of different selective searches on the same test positions. We use a method similar to the one we used in

section 4.1 to generate the test positions; the only difference is that this time we use 150 initial positions from the opening book. Table 4.11 lists the corresponding search depth used for different stages. For each intermediate game position, we compare different selective searches with the best combination of enhancements, NITKH. The comparison statistics include speed-up and how close the value gotten from selective search is to the correct value. This experiment helps us to understand the performance of different selective searches in different stage of the game.

step	1-14	15-44	45-49	50-54	55-64	65-70
depth	2	3	4	5	6	7

Table 4.11 Search Depths for Different Stages

Another experiment is playing a tournament to determine the optimal selective factor. Starting with 75 initial positions from the opening book, 150 games are played between the non-selective program and each version of the selective program under normal tournament conditions-30 seconds per move. The optimal selective factor is the one used in the version with the highest winning score. This experiment can determine the most practical selective factor.

4.1.3 Experimental results

Table 4.12 lists some statistics about different selective searches on the test positions. These statistics show that:

a) In Amazons, selective searches have good speed-up on the early stage. But during the middle and the endgame, the overhead of selective search becomes worse. After step 55, N-selective search is not practical in Amazons anymore.

b) Dividing a move into two separate operations (queen-move and arrow-move) is not a good idea for selective search in Amazons. Though this approach can further reduce the size of the game tree, the chance of cutting off decisive variations during move selection increases. Comparing statistics of F=20 with N=10 and M=6, we find that the former gets greater speed-up and accuracy.

c) The selective search F=40 seems to be the optimal selective factor, since it balances speed-up with accuracy.

Since we proved that N-Best search is not practical during the endgame of Amazons, all versions of the selective program here will change to use NITKH after move 55 during a tournament. Table 4.13 shows the tournament results from the point of view of the selective programs. The best winning percentage is 69%, which proved that F=40 seems to be the optimal selective factor.

Step (Ply)	**NITKH**	**Selective Search (F=20)**					
	CPU Time	**Speed-up**	**Same Move**	**Same Value**	**+/- 2**	**+/- 5**	**+/- 10**
Step5 (Ply 2)	6.93	8.45	75%	81%	81%	83%	88%
Step25 (Ply 3)	25.07	12.78	56%	55%	59%	59%	65%
Step45 (Ply 4)	5.01	3.10	73%	78%	79%	79%	81%
Step50 (Ply 5)	17.26	4.01	66%	89%	89%	90%	91%
Step54 (Ply 5)	9.09	2.13	66%	97%	97%	97%	97%
Step55 (Ply 6)	17.09	0.44	69%	99%	99%	99%	99%

Table 4.12 Comparison of N-best Searches and Brute-Force (NITKH)

Step (Ply)	NITKH	Selective Search (F=30)					
	CPU Time	Speed-up	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step5 (Ply 2)	6.93	6.83	83%	89%	89%	91%	96%
Step25 (Ply 3)	25.07	8.78	64%	65%	65%	65%	71%
Step45 (Ply 4)	5.01	2.07	80%	88%	89%	89%	89%
Step50 (Ply 5)	17.26	2.02	67%	93%	93%	95%	95%
Step54 (Ply 5)	9.09	1.46	67%	97%	97%	97%	97%
Step55 (Ply 6)	17.09	0.36	69%	99%	99%	99%	99%

Table 4.12 (cont.)

Step (Ply)	NITKH	Selective Search (F=40)					
	CPU Time	Speed-up	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step5 (Ply 2)	6.93	5.84	85%	91%	91%	93%	96%
Step25 (Ply 3)	25.07	6.61	70%	75%	76%	77%	80%
Step45 (Ply 4)	5.01	1.56	81%	91%	91%	91%	92%
Step50 (Ply 5)	17.26	1.38	70%	97%	97%	97%	97%
Step54 (Ply 5)	9.09	1.20	68%	99%	99%	99%	99%
Step55 (Ply 6)	17.09	0.33	70%	100%	100%	100%	100%

Table 4.12 (cont.)

Step (Ply)	NITKH	Selective Search (F=50)					
	CPU Time	Speed-up	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step5 (Ply 2)	6.93	5.19	90%	96%	96%	97%	100%
Step25 (Ply 3)	25.07	5.40	75%	81%	82%	83%	85%
Step45 (Ply 4)	5.01	1.29	83%	94%	94%	94%	94%
Step50 (Ply 5)	17.26	1.06	69%	98%	98%	98%	98%
Step54 (Ply 5)	9.09	1.03	68%	99%	99%	99%	99%
Step55 (Ply 6)	17.09	0.31	70%	100%	100%	100%	100%

Table 4.12 (cont.)

Step (Ply)	NITKH	Selective Search (N=5, M=4)					
	CPU Time	Speed-up	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step5 (Ply 2)	6.93	31.00	61%	56%	58%	72%	77%
Step25 (Ply 3)	25.07	28.99	37%	29%	33%	35%	46%
Step45 (Ply 4)	5.01	1.98	43%	45%	48%	49%	51%
Step50 (Ply 5)	17.26	3.97	29%	75%	75%	75%	77%
Step54 (Ply 5)	9.09	2.83	18%	95%	95%	95%	95%
Step55 (Ply 6)	17.09	0.46	18%	98%	98%	98%	98%

Table 4.12 (cont.)

Step (Ply)	NITKH	Selective Search (N=6, M=5)					
	CPU Time	Speed-up	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step5 (Ply 2)	6.93	19.00	67%	66%	68%	77%	81%
Step25 (Ply 3)	25.07	16.62	43%	33%	35%	41%	51%
Step45 (Ply 4)	5.01	1.01	49%	55%	57%	57%	59%
Step50 (Ply 5)	17.26	1.92	35%	79%	79%	80%	81%
Step54 (Ply 5)	9.09	2.04	22%	96%	96%	96%	96%
Step55 (Ply 6)	17.09	0.40	21%	99%	99%	99%	99%

Table 4.12 (cont.)

Step (Ply)	NITKH	Selective Search (N=8, M=6)					
	CPU Time	Speed-up	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step5 (Ply 2)	6.93	10.00	76%	80%	83%	88%	93%
Step25 (Ply 3)	25.07	8.39	51%	44%	46%	51%	59%
Step45 (Ply 4)	5.01	0.45	60%	65%	65%	67%	70%
Step50 (Ply 5)	17.26	0.85	42%	89%	89%	89%	91%
Step54 (Ply 5)	9.09	1.31	32%	97%	97%	97%	97%
Step55 (Ply 6)	17.09	0.34	32%	99%	99%	99%	100%

Table 4.12 (cont.)

Step (Ply)	NITKH	Selective Search (N=10, M=6)					
	CPU Time	Speed-up	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step5 (Ply 2)	6.93	6.88	79%	85%	85%	90%	95%
Step25 (Ply 3)	25.07	5.71	55%	53%	55%	60%	67%
Step45 (Ply 4)	5.01	0.27	71%	76%	76%	77%	79%
Step50 (Ply 5)	17.26	0.53	65%	92%	92%	93%	93%
Step54 (Ply 5)	9.09	1.11	48%	97%	97%	97%	97%
Step55 (Ply 6)	17.09	0.32	43%	99%	99%	99%	100%

Table 4.12 (cont.)

Selective Factor *F*	Result (Win-Loss)	Winning Percentage
20	65: 45	59%
30	75: 35	68%
40	76: 34	69%
50	73: 37	66%

Table 4.13 Tournament Results between N-best search and Brute-Force (NITKH)

4.3 ProbCut and Multi-ProbCut

Both ProbCut and Multi-ProbCut are 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. ProbCut uses fixed values of D and D' in different stages, while Multi-ProbCut performs several check searches of increasing depth. In Amazons, the search branches in different stages vary a lot. If we use fixed D and D', the search strength would be similar to that of the non-selective search program in most stages. We choose to implement Multi-ProbCut in Amazons.

The parameters a, b and sigma can be estimated by three steps. First, we used a method similar to the one we used in section 4.2 to generate test positions from 150 games. Thereafter, for each test position, we do the iterative search up to a specific maximum search depth. Table 4.14 lists the corresponding search depth used for different stages. In the third step, for each stage and each pair of (D', D), the parameters a, b and sigma can estimated by a linear regression. Table 4.14 lists the heights and check depths that we chose to test in Mulan.

Table 4.15 lists the parameters gotten from the experiments. Figures 4.12 - 4.15 show 150 evaluation pairs in different stages in Amazons. From these figures and the regression coefficient of determination -- r^2 -- listed in Table 4.15, it is obvious that the linear model is only suitable for the endgame stages of the Amazons. For example, the linear relationship of evaluation pairs at move 40 is visually obvious, and its r^2 is larger than 0.95 indicating that less than 5% of the variance in the data is caused by the random error.

h	3	4	5	6	7	8
d	1	2	1	2	3	4

Table 4.14 Check depths for different heights h used in Mulan

step	1-4	5-34	35-45	46-55	56-61	62-65	66-70
depth	2	3	4	5	6	7	8

Table 4.15 Search depths for different stages

Step	5	6	7	8	9	10	11	12	13	14	15
v	1	1	1	1	1	1	1	1	1	1	1
v'	3	3	3	3	3	3	3	3	3	3	3
a	0.78	0.64	0.64	0.45	0.61	0.66	0.70	0.80	0.75	0.63	0.75
b	38.70	30.14	28.84	43.19	30.52	16.29	13.93	-6.36	-3.96	25.12	2.14
σ	21.13	25.43	31.28	33.27	35.18	35.49	36.47	41.01	43.18	44.84	48.52
r^2	0.71	0.64	0.52	0.38	0.50	0.54	0.58	0.61	0.56	0.59	0.63

Table 4.16 Parameters a, b and σ estimated by linear regression

Step	16	17	18	19	20	21	22	23	24	25	26
v	1	1	1	1	1	1	1	1	1	1	1
v'	3	3	3	3	3	3	3	3	3	3	3
a	0.72	0.72	0.74	0.77	0.75	0.76	0.74	0.78	0.82	0.81	0.83
b	-2.91	1.04	-3.72	-3.92	-8.43	-13.75	-6.9	-10.4	-24.0	-20.2	-28.5
σ	54.84	57.14	55.66	48.85	60.96	60.94	65.75	67.58	65.26	65.30	61.29
r^2	0.57	0.60	0.61	0.72	0.60	0.67	0.61	0.64	0.69	0.72	0.76

Table 4.16 (cont.)

Step	27	28	29	30	31	32	33	34	35	35	36
v	1	1	1	1	1	1	1	1	1	2	1
v'	3	3	3	3	3	3	3	3	3	4	3
a	0.79	0.86	0.79	0.84	0.81	0.90	0.91	0.89	0.88	1.02	0.88
b	-28.1	-33.7	-24.5	-30.4	-30.4	-39.4	-55.2	-40.4	-41.3	45.4	-69.9
σ	72.57	70.05	80.98	75.64	92.52	72.65	90.24	79.14	91.20	52.73	89.60
r^2	0.71	0.76	0.68	0.75	0.63	0.81	0.71	0.82	0.74	0.93	0.79

Table 4.16 (cont.)

Step	**36**	**37**	**37**	**38**	**38**	**39**	**39**	**40**	**40**	**41**	**41**
v	2	1	2	1	2	1	2	1	2	1	2
v'	4	3	4	3	4	3	4	3	4	3	4
a	0.99	0.93	1.00	0.94	1.02	0.99	1.03	1.00	1.02	0.98	1.03
b	0.5	-41.7	11.2	-35.5	-2.7	-19.6	5.7	-26.3	0.2	-21.8	-0.3
σ	39.04	71.52	36.72	54.29	34.81	50.24	28.76	47.49	32.13	43.11	27.63
r^2	0.96	0.86	0.97	0.92	0.97	0.94	0.98	0.95	0.98	0.96	0.98

Table 4.16 (cont.)

Step	**42**	**42**	**43**	**43**	**44**	**44**	**45**	**45**	**46**	**46**	**46**
v	1	2	1	2	1	2	1	2	1	2	1
v'	3	4	3	4	3	4	3	4	3	4	5
a	0.98	1.04	1.01	1.03	1.03	1.04	1.01	1.03	0.97	1.03	1.01
b	-22.5	5.7	-19.4	1.1	-14.4	2.4	-16.2	5.3	-20.7	2.62	-20.7
σ	48.42	29.54	40.04	23.56	37.18	20.50	35.87	23.20	57.85	20.12	63.90
r^2	0.95	0.98	0.97	0.99	0.97	0.99	0.97	0.99	0.94	0.99	0.93

Table 4.16 (cont.)

Step	**47**	**47**	**47**	**48**	**48**	**48**	**49**	**49**	**49**	**50**	**50**
v	1	2	1	1	2	1	1	2	1	1	2
v'	3	4	5	3	4	5	3	4	5	3	4
a	1.01	1.04	1.03	1.01	1.03	1.05	1.01	1.04	1.06	1.01	1.05
b	-20.7	-1.53	-20.7	-20.7	1.30	-20.7	-6.83	1.39	-20.7	-20.7	4.28
σ	30.67	19.52	44.40	34.90	20.19	47.67	29.09	21.38	42.20	36.19	17.37
r^2	0.98	0.99	0.97	0.98	0.99	0.96	0.99	0.99	0.97	0.98	1.00

Table 4.16 (cont.)

Step	**50**	**51**	**51**	**51**	**52**	**52**	**52**	**53**	**53**	**53**	**54**
v	1	1	2	1	1	2	1	1	2	1	1
v'	5	3	4	5	3	4	5	3	4	5	3
a	1.05	1.04	1.05	1.09	1.03	1.05	1.08	1.05	1.06	1.11	1.06
b	-20.7	-8.73	2.77	-20.7	-6.83	2.31	-11.0	-6.01	1.73	-9.00	-2.25
σ	43.96	25.31	18.01	32.50	24.11	14.95	34.62	20.15	11.44	25.31	16.46
r^2	0.97	0.99	1.00	0.99	0.99	1.00	0.98	0.99	1.00	0.99	1.00

Table 4.16 (cont.)

Step	**54**	**54**	**55**	**55**	**55**	**56**	**56**	**56**	**56**	**57**	**57**
v	2	1	1	2	1	1	2	1	2	1	2
v'	4	5	3	4	5	3	4	5	6	3	4
a	1.06	1.12	1.06	1.06	1.14	1.06	1.07	1.13	1.14	1.07	1.07
b	1.36	-3.09	-2.01	0.02	-6.50	-1.64	2.74	-2.42	5.09	-4.16	0.11
σ	11.64	21.82	11.05	11.62	21.36	14.99	16.39	22.01	26.01	22.48	8.08
r^2	1.00	0.99	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.99	1.00

Table 4.16 (cont.)

Step	**57**	**57**	**58**	**58**	**58**	**58**	**59**	**59**	**59**	**59**	**60**
v	1	2	1	2	1	2	1	2	1	2	1
v'	5	6	3	4	5	6	3	4	5	6	3
a	1.14	1.15	1.06	1.07	1.14	1.16	1.07	1.07	1.16	1.16	1.07
b	-6.65	1.25	-1.99	1.87	-3.73	2.97	-2.14	1.13	-4.21	-0.75	-2.48
σ	30.92	19.98	14.23	14.04	26.78	29.52	14.08	15.39	29.02	32.43	14.80
r^2	0.99	1.00	1.00	1.00	0.99	0.99	1.00	1.00	0.99	0.99	1.00

Table 4.16 (cont.)

Step	**60**	**60**	**60**	**61**	**61**	**61**	**61**	**62**	**62**	**62**	**62**
v	2	1	2	1	2	1	2	1	2	1	2
v'	4	5	6	3	4	5	6	3	4	5	6
a	1.08	1.15	1.17	1.08	1.08	1.17	1.17	1.07	1.08	1.17	1.18
b	0.71	-2.95	1.74	-0.49	0.25	-1.78	-0.21	-1.04	1.26	-0.64	2.69
σ	10.60	28.45	17.79	10.27	16.01	17.77	26.12	17.49	10.43	27.20	19.03
r^2	1.00	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00

Table 4.16 (cont.)

Step	**62**	**63**	**63**	**63**	**63**	**63**	**64**	**64**	**64**	**64**	**64**
v	3	1	2	1	2	3	1	2	1	2	3
v'	7	3	4	5	6	7	3	4	5	6	7
a	1.75	1.07	1.08	1.17	1.18	1.19	1.08	1.09	1.18	1.21	1.21
b	1.19	-1.04	1.26	-0.64	2.69	1.75	-0.56	1.30	0.32	4.07	3.82
σ	21.29	17.49	10.43	27.20	19.03	21.29	12.88	10.75	20.61	36.84	40.95
r^2	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	0.99	0.99

Table 4.16 (cont.)

Step	**65**	**65**	**65**	**65**	**65**	**66**	**66**	**66**	**66**	**66**	**66**
v	1	2	1	2	3	1	2	1	2	3	4
v'	3	4	5	6	7	3	4	5	6	7	8
a	1.09	1.09	1.20	1.21	1.22	1.09	1.10	1.20	1.22	1.23	1.25
b	-0.86	-0.70	-1.94	-1.12	-2.10	-0.96	0.08	-1.78	-0.63	-1.85	-0.83
σ	10.01	11.73	21.74	25.05	34.57	13.42	11.28	23.07	24.32	29.77	37.68
r^2	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	0.99

Table 4.16 (cont.)

Step	**67**	**67**	**67**	**67**	**67**	**67**	**68**	**68**	**68**	**68**	**68**	**68**
v	1	2	1	2	3	4	1	2	1	2	3	4
v'	3	4	5	6	7	8	3	4	5	6	7	8
a	1.10	1.10	1.22	1.23	1.25	1.27	1.10	1.11	1.22	1.24	1.26	1.29
b	0.19	0.68	0.57	1.22	-0.34	-0.07	-2.15	-0.71	-3.38	-0.72	-1.29	-0.56
σ	10.68	13.48	24.12	27.14	35.72	41.50	15.28	14.76	26.75	32.30	37.73	50.59
r^2	1.00	1.00	1.00	1.00	0.99	0.99	1.00	1.00	1.00	0.99	0.99	0.99

Table 4.16 (cont.)

Step	**69**	**69**	**69**	**69**	**69**	**69**	**70**	**70**	**70**	**70**	**70**	**70**
v	1	2	1	2	3	4	1	2	1	2	3	4
v'	3	4	5	6	7	8	3	4	5	6	7	8
a	1.10	1.11	1.23	1.24	1.27	1.29	1.10	1.11	1.23	1.25	1.27	1.31
b	-0.69	-0.44	-3.08	-3.03	-5.76	-3.11	-1.81	0.80	-1.27	0.94	-3.09	2.92
σ	8.32	9.89	24.53	26.20	41.49	57.22	10.71	14.13	20.08	21.82	42.16	53.82
r^2	1.00	1.00	1.00	1.00	0.99	0.99	1.00	1.00	1.00	1.00	0.99	0.99

Table 4.16 (cont.)

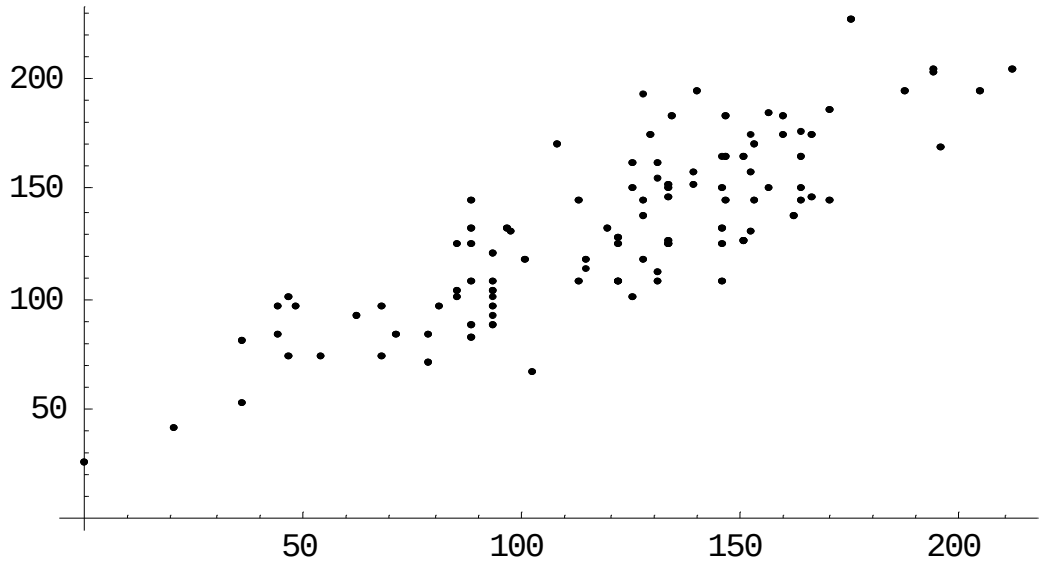

Figure 4.12 Relation between v=1 and v'=3 at move 5

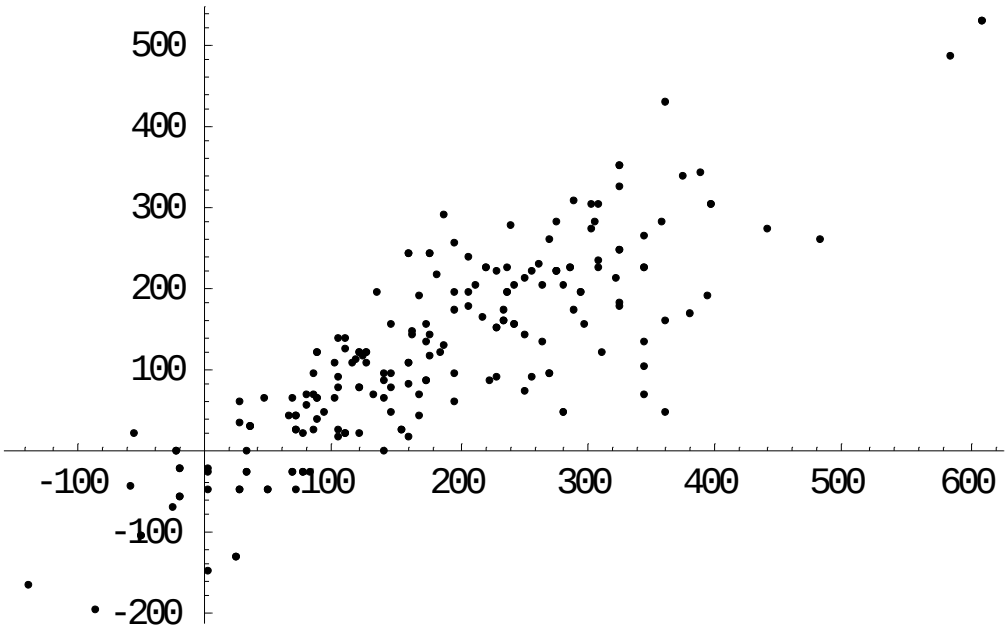

Figure 4.13 Relation between v=1 and v'=3 at move 25

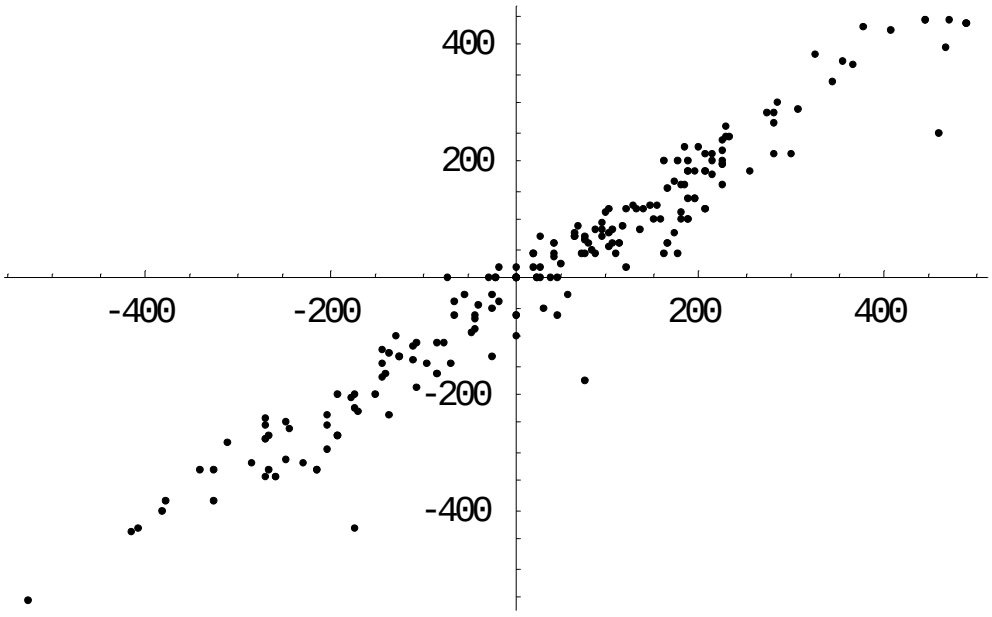

Figure 4.14 Relation between v=1 and v'=3 at move 40

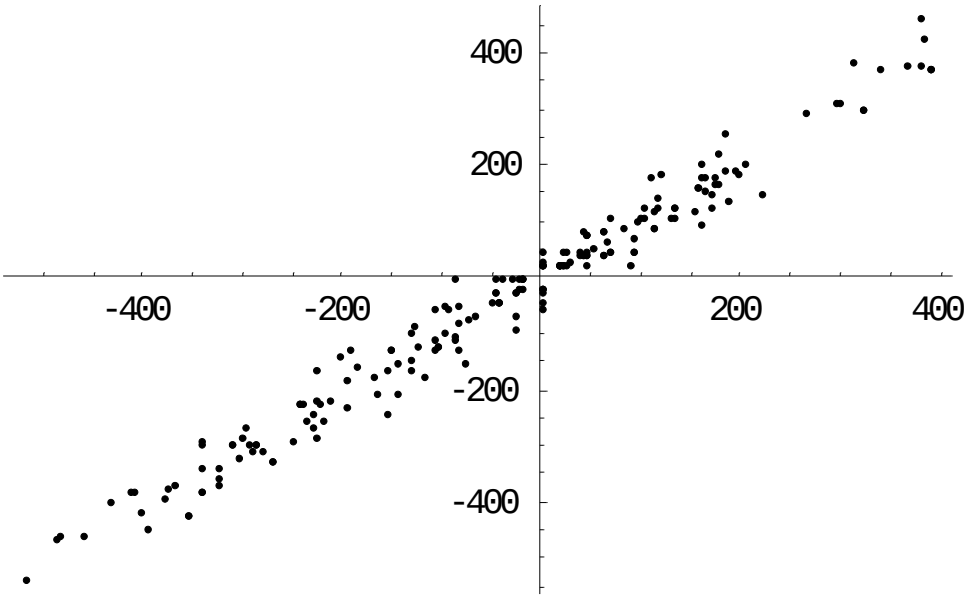

Figure 4.15 Relation between v=2 and v'=4 at move 40

As we discuss in section 2.4.2, we can predict the probability that V_D>=beta from V_D'>=((1/Φ(p)) *σ + beta –b) / a. So t = 1/Φ(p) determines the cutoff threshold of the search. Choosing a correct *t* is essential for the performance of Multi-ProbCut. Choosing a larger *t* can cause more cuts, but also increases the probability of cutting off the best move. We use two methods, which are identical to those used in the last section, to compare the effect of different values of t. The first one is collecting statistics include speed-up and accuracy; the other one is using tournaments.

Table 4.17 shows statistics including speed-up and accuracy about Multi-ProbCut with different thresholds on the set of test positions. These statistics show that:

a) In Amazons, Multi-ProbCut is not practical at the beginning of the game. For example, in the first 25 moves, typically the computer can only search to ply 2-3 in 10 seconds. Since ply 4 is the minimum depth at which we can use

Multi-ProbCut, we cannot take any advantage by applying Multi-ProbCut at this stage.

b) In our experiment, the performance of Multi-ProbCut is bad during the midgame of Amazons. Typically Multi-ProbCut can only get at speed-up of a factor of 2-4 compared to non-selective search in this stage, and 40-60% of the best moves are cut off. For Multi-ProbCut, a stable evaluation function is necessary, but unfortunately building a stable evaluation function for Amazons is very difficult. Figures 4.12 - 4.15 shows that the evaluation function we use in Mulan cannot fit the linear prediction model used by Multi-ProbCut during the beginning and midgame. So the large difference between the true and the predicted value makes Multi-ProbCut error-prone.

c) The performance of Multi-ProbCut becomes better and better as we approach the endgame. This is because almost all territories on the board have been determined by this time, so evaluations obtained from searches of different depths are strongly correlated.

To further investigate the performance of Multi-ProbCut in Amazons, we played a tournament to compare Multi-ProbCut with a non-selective search algorithm. Table 4.18 shows the results. They show that t=1.5 is the optimal cutting cutoff for our implementation. The tournament results also suggest that Multi-ProbCut cannot improve the playing strength of Mulan.

NITKH	Step25 (Ply 3)	Step35 (Ply 4)	Step40 (Ply 4)	Step50 (Ply5)	Step55 (Ply5)	Step60 (Ply6)	Step63 (Ply7)	Step68 (Ply8)
CPU Time	6.80	11.53	3.62	4.48	1.69	3.16	16.24	22.68
Leaf Nodes	320596	452930	129924	208787	79682	160714	902219	1291226

Step (Ply)	Selective Search (t=1.0)						
	Speed-up (time)	Speed-up (LN)	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step25 (Ply 3)	1.74	1.00	100%	100%	100%	100%	100%
Step35 (Ply 4)	3.37	3.45	34%	34%	37%	42%	44%
Step40 (Ply 4)	2.99	3.02	56%	59%	60%	63%	64%
Step50 (Ply5)	9.89	10.44	83%	87%	88%	88%	88%
Step55 (Ply5)	8.16	8.12	93%	93%	93%	94%	94%
Step60 (Ply6)	18.11	18.37	92%	97%	97%	97%	97%
Step63 (Ply7)	80.39	87.31	96%	98%	98%	98%	98%
Step68 (Ply8)	76.98	76.46	90%	97%	97%	97%	97%

Table 4.17 Comparison of Multi-ProbCut and Brute-Force (NITKH)

Step (Ply)	Selective Search (t=1.2)						
	Speed-up (time)	Speed-up (LN)	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step25 (Ply 3)	1.01	1.00	100%	100%	100%	100%	100%
Step35 (Ply 4)	3.00	3.05	42%	43%	44%	45%	47%
Step40 (Ply 4)	2.59	2.58	63%	65%	66%	68%	70%
Step50 (Ply5)	8.10	8.13	88%	90%	89%	92%	93%
Step55 (Ply5)	8.02	8.00	95%	95%	97%	97%	98%
Step60 (Ply6)	15.90	16.31	93%	96%	96%	96%	96%
Step63 (Ply7)	70.45	70.27	95%	98%	98%	98%	98%
Step68 (Ply8)	70.16	70.49	93%	98%	98%	98%	98%

Table 4.17 (cont.)

Step (Ply)	Selective Search (t=1.5)						
	Speed-up (time)	Speed-up (LN)	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step25 (Ply 3)	1.01	1.00	100%	100%	100%	100%	100%
Step35 (Ply 4)	2.67	2.75	48%	48%	49%	51%	54%
Step40 (Ply 4)	2.39	2.27	68%	69%	69%	70%	72%
Step50 (Ply5)	7.59	7.75	92%	93%	93%	93%	94%
Step55 (Ply5)	7.36	7.31	93%	93%	93%	94%	94%
Step60 (Ply6)	13.11	13.31	96%	97%	97%	97%	97%
Step63 (Ply7)	70.39	77.03	95%	97%	97%	98%	98%
Step68 (Ply8)	76.98	75.01	96%	97%	98%	98%	98%

Table 4.17 (cont.)

Step (Ply)	Selective Search (t=2.0)						
	Speed-up (time)	Speed-up (LN)	Same Move	Same Value	+/- 2	+/- 5	+/- 10
Step25 (Ply 3)	1.01	1.00	100%	100%	100%	100%	100%
Step35 (Ply 4)	2.24	2.09	54%	54%	55%	57%	59%
Step40 (Ply 4)	2.01	2.07	70%	72%	73%	73%	74%
Step50 (Ply5)	6.44	6.14	90%	93%	93%	94%	95%
Step55 (Ply5)	6.16	6.10	94%	95%	95%	95%	96%
Step60 (Ply6)	10.13	10.18	91%	94%	95%	96%	99%
Step63 (Ply7)	69.88	66.71	96%	98%	98%	98%	98%
Step68 (Ply8)	65.31	63.39	96%	98%	98%	99%	99%

Table 4.17 (cont.)

Threshold *t*	Result (Win-Loss)	Winning Percentage
1.0	52-98	34.7%
1.2	58-92	38.7%
1.5	63-87	42%
2.0	60-90	40%

Table 4.18 Tournament results between Multi-ProbCut and Brute-Force (NITKH)

4.4 The Construction of the Evaluation Function

Being able to search game-tree deeper and faster is important for creating a high quality game-playing program. But for almost all interesting games, the corresponding game tree is bushy. Since it is impossible to exhaustively search the whole tree, we need an evaluation function to assess leaf positions. Typically constructing evaluation functions includes two phases: selecting good features and combining selected features. In this section, we tested some existing features used for Amazons and some new features we developed. After that, we tried two methods to combine them into an evaluation function: 1) linear combination using tournament; 2) a pattern classification approach based on Bayesian learning.

4.4.1 New Features

In Chapter 3, we explained three published features for Amazons: Mobility, Territory and Territory-and-Mobility. Based on the ideas of enlarging legal moves and controlling more squares, we developed several new features: Min-Mobility, Regions, Ax and Bx series, and Relative Distance Territory. In this section, we introduce the ideas and implementations of these new features.

4.4.1.1 Min-Mobility

As we played Amazons, we found that keeping all our pieces mobile at the early stage is very important. Neither the Mobility nor the Territory feature can notice that a particular piece is in danger of being trapped at the early stage. We developed a new feature called Min-Mobility to solve this problem. The Min-Mobility of a player is the minimum over the player's four pieces of how many squares it can reach in one step. The

Min-Mobility feature is evaluated by subtracting the opponent's Min-Mobility from the player's.

The Min-Mobility feature can efficiently avoid one or two of a player's pieces being blocked. This is very useful when the territory classification is not clear enough. But in the mid-game and endgame, "sacrificing" a piece and letting it be trapped might allow you to do well elsewhere on the board.

4.4.1.2 Regions

During the play of Amazons, the board will typically be divided into different regions in the middle game. We want to place all four amazons in coordinated way: the number of queens in a specific region should be proportional to the empty squares in the region, and we do not want to be outnumbered by our opponent in an important region. However, neither Mobility nor Dominance can detect this. The territory feature assumes queens can go in all directions at once, and defend against attackers approaching from different sides at the same time. Therefore, it optimistically evaluates unbalanced situations. Mobility can make sure amazons don't get trapped in very small regions, but it tends to move all pieces into the biggest regions of the board.

The Regions feature is designed to place all four amazons in different regions in coordinated way. The idea of this feature is illustrated by the following.

```
int Regions(pos) {
    Find out all regions in the current position pos.
    double whiteScore = 0.0;
    double blackScore = 0.0;
    // A "Shared region" means both sides have at least one place inside.
```

```
    // The Totally dominant region is more important then the shared region.
    For each white piece i
        whitePieceValue[i] = 2*(the number totally dominant regions) + (the number of
        "shared" regions);
    For each black piece i
        blackPieceValue[i] = 2*(the number totally dominant regions) + (the number of
        "shared" regions);
    For each region  {
        score = the number of empty squares in the region.
        blackNum = 0.0;
        whiteNum = 0.0;
        For each white piece i which can reach the region
            whiteNum = whiteNum + 1.0 / whitePieceValue[i];
        For each black piece i which can reach the region
            blackNum = blackNum + 1.0 / blackPieceValue[i];
        whiteScore += score * whiteNum/(whiteNum+blackNum);
        blackScore += score * blackNum/(whiteNum+blackNum);
    }
    value = (int)((1000*(whiteScore - blackScore))/(whiteScore + blackScore));
    if (white's turn to play)
        return value;
    else
        return –value;
}
```

Figure 4.16 The procedure of the Regions feature

There are two obvious shortcomings in the Regions feature. First, the Regions feature does not consider how quickly the amazons can get to an empty square. Thus it is quite easy to enter some bad positions. By combining Regions with the MSP feature, this problem can be solved. Second, the horizon effect can also affect the value of Region

feature, since Regions cannot detect when a region is almost, but not quite enclosed; that is, if a wall between regions still has holes in it at the horizon. Because of this, the effect of the Region feature strongly depends on the search depth.

4.4.1.3 Ax & Bx

Ax represents a series of features we derived from the Mobility feature. The score of Ax is based on how many squares each side can reach in x steps. The basic idea behind Ax is that the squares which can be reached in different numbers of steps should be weighted differently. So combining A1, A2, etc with appropriate weights may give a more precise evaluation of the board.

Similarly, Bx is a series of features which are essentially the same as the territory feature. It represents how many squares the player can reach in x steps that the opponent cannot. Using B1, B2, etc, we can evaluate not only whether a square belongs to the player's territory, but also how fast the player can reach it.

4.4.1.4 Relative Distance Territory (RDT)

As we mentioned above, the Territory feature does not work well in unbalanced situations, such as when 1 queen is outnumbered by two or more. To solve this problem, we developed a new feature called the Relative Distance Territory (RDT). The implementation of the RDT feature is described as the following.

```
// pos : current board position
// evaluate the RDT feature for the board position "pos" and return the result.
double RDT(pos)
{
    blackPoints = 0;
```

whitePoints = 0;
for all empty squares x do
{
    blackFraction = 0;
    whiteFraction = 0;
    minStonePly = the minimum number of moves that a piece needs to arrive at x;
    for (i=0; i<4; i++) {
        b = the number of moves that i th black piece needs to arrive at x;
        w = the number of moves that i th white piece needs to arrive at x;
        b = b – minStonePly;
        w = w – minStonePly;
        blackFraction = blackFraction + $(1/weight)^b$;
        whiteFraction = whiteFraction + $(1/weight)^w$;
    }
    whitePoints = whitePoints + $\frac{whiteFraction}{blackFraction + whiteFraction}$;
    blackPoints = blackPoints + $\frac{blackFraction}{blackFraction + whiteFraction}$;
}
if (white's turn to play)
    return (whitePoints – blackPoints);
else
    return (blackPoints – whitePoints);
}

Figure 4.17 The procedure of the RDT feature

As we mentioned before, the Territory feature optimistically assumes queens can go all directions at once. But in reality, a queen can't defend against multiple attackers approaching from different sides at the same time. In the Relative Distance Territory, for each square each player gets a fractional score. The fractional score that each player gets

depends on two factors: how many of his/her pieces can reach that square, and their relative speeds, i.e., how quickly they can get there. So the RDT feature can evaluate the position more precisely in unbalanced situations. By considering relative speed, the RDT feature may also be able to use “barricade” positions intelligently to maximize the moves necessary for the opponent to reach empty squares.

The main shortcoming of the RDT feature is its evaluation speed. There are two reasons why evaluating the RDT feature is more time-consuming than the Territory feature. First, RDT needs to evaluate how quickly each piece can reach a square, while the Territory feature only needs to know which player’s pieces can reach a square first. Furthermore, dividing the square score into two fractions introduces expensive floating-point or integer operations, while the Territory feature can be implemented efficiently using bitmap operations.

4.4.2 Choosing good features

We described several new features for the game of Amazons in the previous section. Now we need to determine which features are the most valuable ones. Selecting good features is important and difficult. First, we need to avoid too few features as well as redundant ones. Furthermore, there is the well-known tradeoff between speed and accuracy of evaluation. We used Mulan as a test bed for this investigation.

Ideally, every feature should measure characteristic of the board position. To avoid redundant features, we divided all features into three groups based on the characteristic they represented: mobility (including Mobility, Ax series and Min-Mobility), coordination (including TM and Regions) and territory (including MSP, Bx

series and Relative Distance Territory). First, we need to find out the best feature in each group, i.e. the best measure of that characteristic of the board position. Then we can combine them according to the characteristics of different stages of the game.

We used two methods to examine the performance of the different features: computer tournaments and human-computer playing experience. We modified Mulan to produce different versions, each applying different features in its evaluation function. We set two of them to play at a time on a set of random initial board positions. Each version played both white and black on each board position. The other method we used was our experience of playing Amazons against different versions of Mulan. This method of feature selection is a very intuitive process; Humans are good at trying different strategies during the game. So Human vs. computer playing experience can help us to further improve the program to handle different situations.

First, we tested territory features. For Bx and MSP, we found that the result of combining Bx series was very similar to MSP. To keep the evaluation function simple and reduce the effort of combining features, we eliminated the Bx feature first. Then we had two choices left: MSP and RDT. As we mentioned before, the RDT evaluator is inherently slower than MSP. To remove the speed difference, we limited each search in the tournament to 100,000 leaf positions. The results listed in Table 4.19 shows that MSP beat RDT in the computer tournament even without considering the speed factor. To further investigate RDT's effect in the stage of the early game, we did an additional tournament where we used RDT only in the first 15 moves. The results are given in Table 4.20. They show that RDT is superior to MSP when we set the RDT weight to 2.8. This result seems also to imply that we should set different RDT weights in different game

stages. Our playing experience also shows that the RDT feature is able to use barricade positions in a smarter way than MSP by considering relative speed. However, the evaluation speed of the RDT feature is extremely slow compared with MSP. When we include time (not just number of leaf positions) in our criteria, the MSP is much superior to RDT. Because of this reason, we decided to select MSP as territory feature.

RDT Weight	**1.4**	**1.6**	**1.8**	**2.0**	**2.2**	**2.4**	**2.6**	**2.8**	**3.0**	**3.2**	**3.4**
Win-Loss	12:88	25:75	38:62	34:66	39:61	41:59	31:69	38:62	38:62	44:56	42:58

Table 4.19 RDT VS Dominance with 100000 Leaf Positions per Move

RDT Weight	**1.6**	**1.8**	**2.0**	**2.2**	**2.4**	**2.6**	**2.8**	**3.0**	**3.2**	**3.4**
Win-Loss	42:58	42:58	47:53	45:55	41:59	49:51	56:44	50:50	42:58	41:59

Table 4.20 Using RDT in First 15 moves against Dominance with 100000 Leaf Positions per Move

We next chose a feature from the mobility-related group. We eliminated Ax first since the result of combining the Ax series is very similar to Mobility. Then we compared Mobility and Min-Mobility. We found that neither of them can form a strong evaluation function alone. According to our playing experience, mobility-related features are only valid in the early stage, typically during the first 20 moves. So we used a linear combination of a mobility-related feature and MSP in the first 20 moves, and pure MSP after that. Our experimental results show that Min-Mobility is superior to Mobility. We didn't find any obvious improvement from the combination of Mobility and MSP, whereas we got an unequivocal improvement from using Min-Mobility and MSP. Table 4.21 illustrates the effect of Min-Mobility, which is combined with MSP using the

equation (weight*mobility_feature + (1 – weight)*MSP). The best result is 194 -111 when the weight is set to 0.5.

Weight	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**
Win-Loss	143:157	176:124	170:130	189:111	194:106
Weight	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**
Win-Loss	189:111	174:126	175:125	129:171	92:208

Table 4.21 Min-Mobility+Dominance VS Dominance

To test the effect of Min-Mobility, we compared Mulan with several existing programs for Amazons. Using an evaluation function consisting of Min-Mobility and Dominance, Mulan can compete with many world championship level Amazons programs, including Yamazon and Arrow. Appendix A lists the playing record of Mulan vs Yamazon and Mulan vs Arrow. From Figure 4.18 – Figure 4.21, we can see that Mulan can effectively block one of its opponent's pieces during the opening stage. But the new evaluation function still has a weakness: it cannot place all four of its amazons in a coordinated way. Figure 4.21 is an example, in which the opponent occupied a big region while Mulan was attacking one of their pieces. The situation becomes worse when playing with a human. For example, my adviser, Prof. Moore, developed a strategy against this version of Mulan. He used one of his pieces as the bait, then blocked all of Mulan's pieces in a big region while they attacked this bait. He finally successfully beat this version of Mulan by more than ten squares.

To solve this problem, we need a feature to coordinate all four pieces' moves. We tested two features related to coordination: TM and Regions. Though Hashimoto et al. **[26]** believes that the TM feature allows the program to place all four amazons in a coordinated way, our experiments show that it cannot improve the playing strength of

Mulan at all. Our results on TM appear in Tables 4.22 and 4.23. The test results of our newly developed feature, Regions, are far more promising. After Regions is applied, Mulan's coordination ability was improved significantly. When we tuned the combination coefficients of the evaluation function to 0.2*Regions+0.5*Dominance +0.3*Min-Mobility, Mulan's playing strength become so strong that we have never beaten her.

The Computer's self-play tournaments also show that the evaluation function with Regions is superior to the original one. To further examine the performance of the Regions feature, we used the new version of Mulan to play against Yamazon and Arrow again. The result is surprising: Mulan won all the games! The tournament records are shown in Appendix B. From Figure 4.22—4.25, we can see that Mulan not only blocked one of her opponent's pieces successfully but also placed all four amazons in different regions in a coordinated way in the early stage.

Now we have found three effective features: MSP, Min-Mobility and Regions. By combining these features linearly, we got a very effective evaluation function. In our experiment, we determined the coefficients of the linear combination from our experience and from tournament results. Though the outcome is promising, there are still three problems left. First, it is very difficult for humans to estimate these coefficients precisely since we don't use game tree search and evaluation functions. Furthermore, though we try to reduce redundancies between features, it is difficult to totally avoid correlations among them. Finally, we divide the whole Amazons game into only two stages. This is obviously not enough. To solve these problems, we will test a pattern classification approach and Automatic Evaluation Function Construction in next section.

Weight	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Win-Loss	40:60	30:70	31:69	30:70	16:84	23:77	22:78	12:88	6:94	13:87

Table 4.22 TM+Dominance vs Dominance (100000 leaf pos. per move)

Weight	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Win-Loss	106:94	77:123	62:138	32:168	30:170	18:182	12:188	7:193	1:199	1:199

Table 4.23 TM+Dominance vs Dominance (10 Sec. per Move)

Tournament criterion	100000 leaf pos. per move	10 sec. per move
Win-Loss	106:94	108:92

Table 4.24 Regions+Min-Mobility+Dominance vs Min-Mobility+Dominance

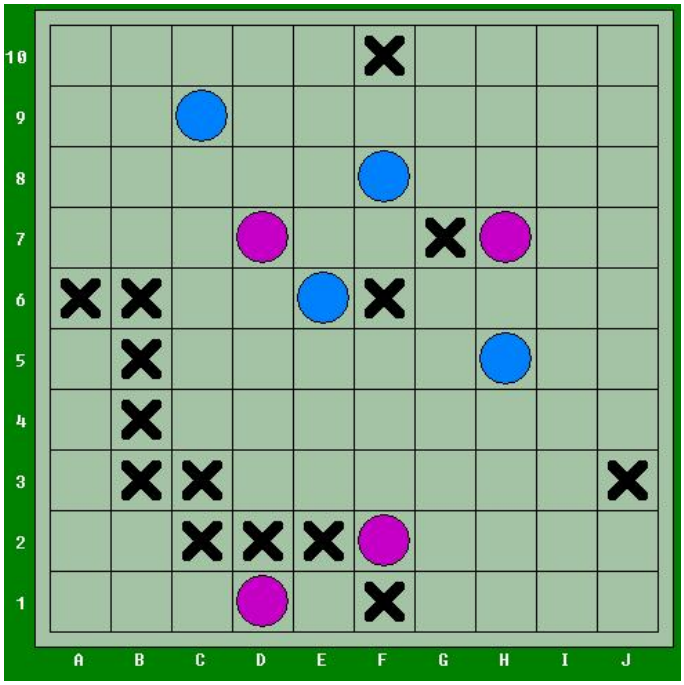

Figure 4.18 Yamazon vs Mulan (without Regions feature): Yamazon plays red

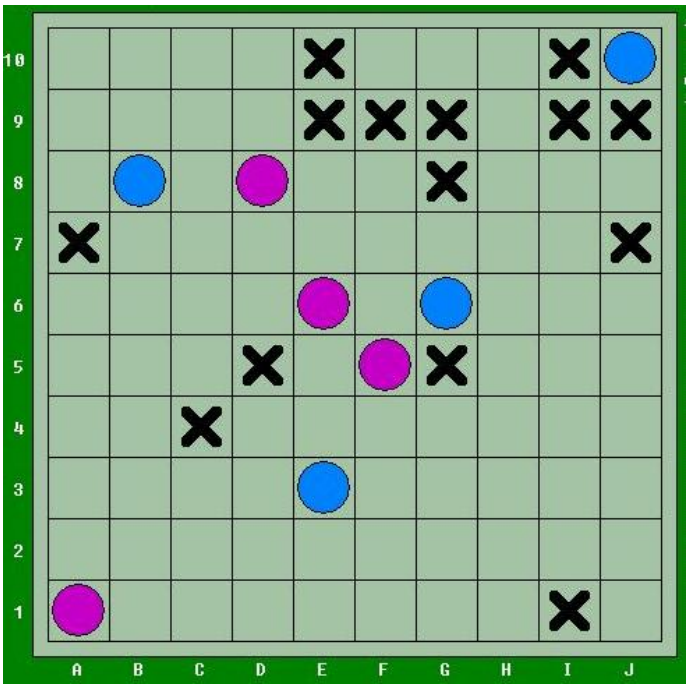

Figure 4.19 Yamazon vs Mulan (without Regions feature): Mulan plays red

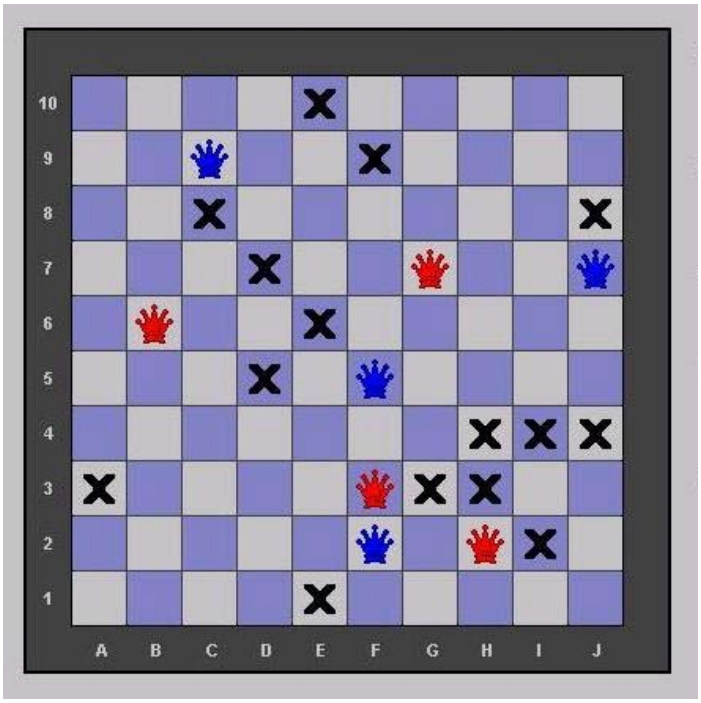

Figure 4.20 Invader vs Mulan (without Regions feature): Invader plays red

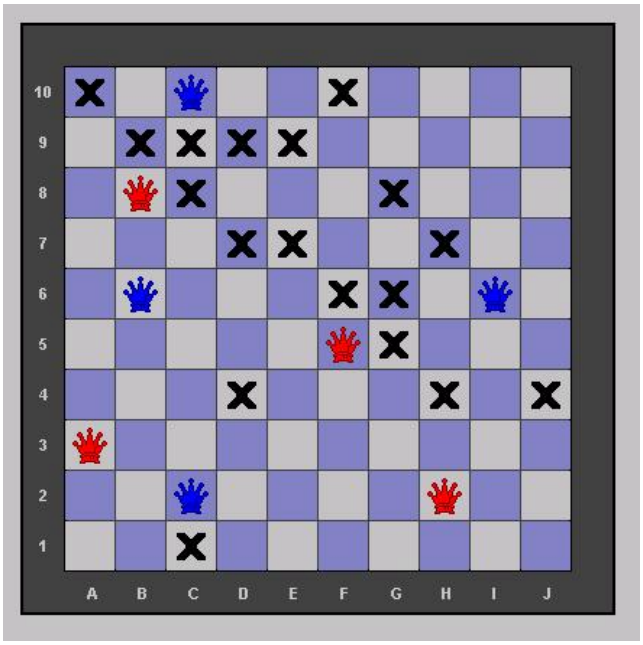

Figure 4.21 Invader vs Mulan (without Regions feature): Mulan plays red

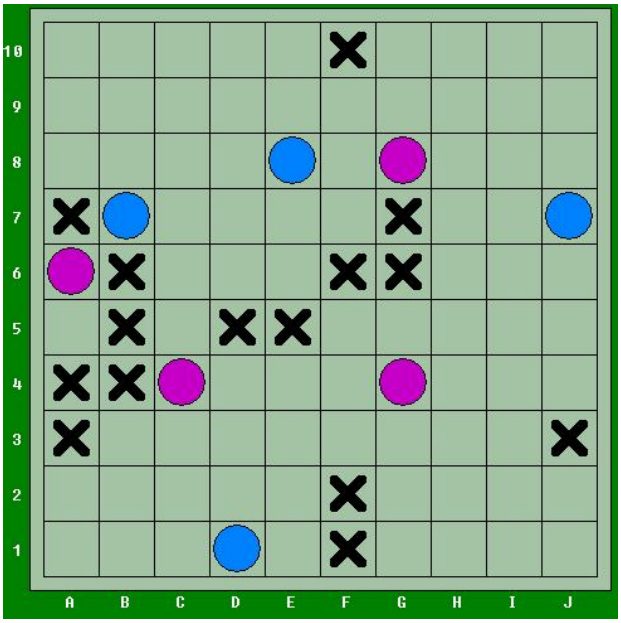

Figure 4.22 Yamazon vs Mulan (with Regions feature): Yamazon plays red

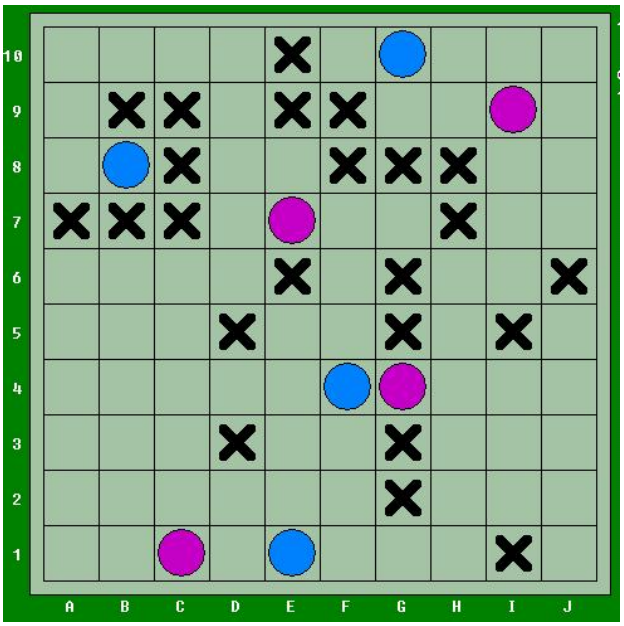

Figure 4.23 Yamazon vs Mulan (with Regions feature): Mulan plays red

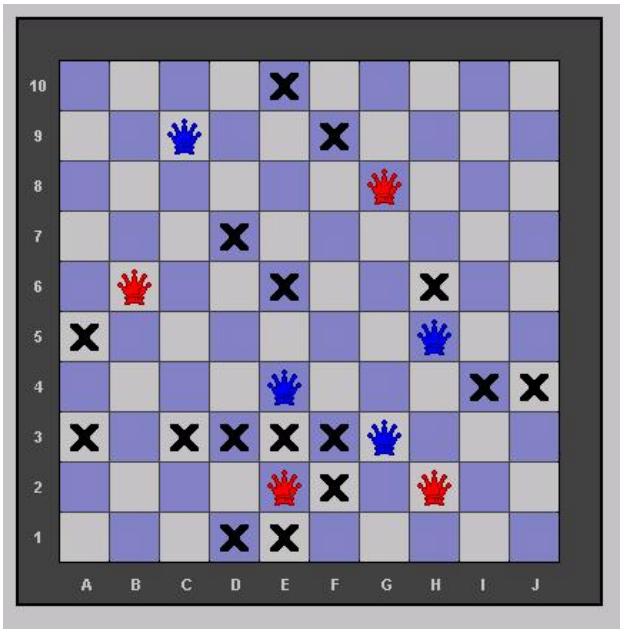

Figure 4.24 Invader vs Mulan (with Regions feature): Invader plays red

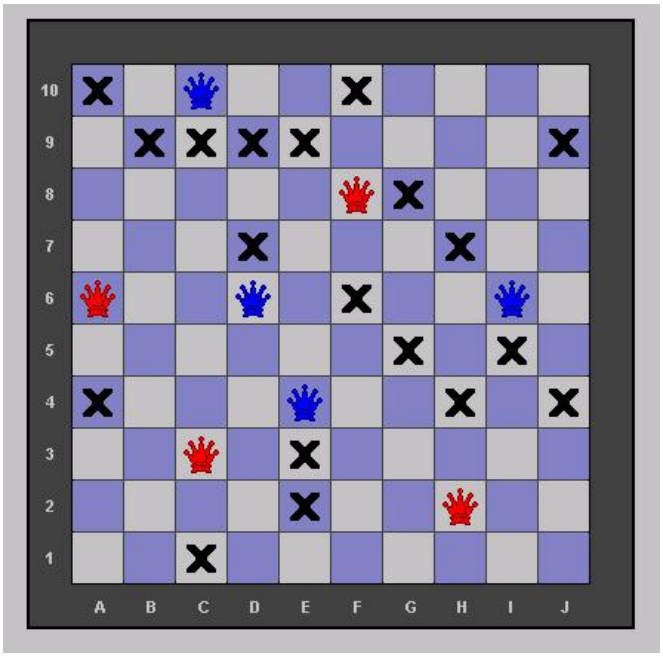

Figure 4.25 Invader vs Mulan (with Regions feature): Mulan plays red

4.4.3 Using Pattern Classification and Bayesian Learning to combine features

In this chapter, we will apply Bayesian learning to our Amazons program—Mulan. Similar to Lee et al's Bayesian learning approach to the game of Othello, our approach also includes two stages: training and recognition.

In the training stage, a database of labeled training positions is required. We obtained these positions from the actual games generated from Mulan's self-play. We generated 2 random initial moves for each game, after which, two copies of Mulan played against each other. During the game, each side is given 30 seconds for each move. Usually, one player will get quite a big advantage in the beginning and go on to win the game. After the game is terminated, all positions are recorded as training data. All positions of the winning player are labeled as winning positions, and all positions of the losing player are labeled as losing positions. A total of 2000 games were played and recorded as the training data.

It is well known that different strategies should be used for different stages of Amazons. We used the number of burnt squares to classify stages. For a stage with N burnt squares, we used training positions with N burnt squares to estimate the parameters. For each training position, the three features were calculated and represented as a feature vector. Then, the mean vectors and covariance matrices for both categories, winning and losing, were estimated. After that, the corresponding evaluation function was calculated via the following equation:

$$g(x) = -\frac{1}{2}(x-\mu_{win})^T V_{win}^{-1}(x-\mu_{win}) - \frac{1}{2}\log|V_{win}| + \frac{1}{2}(x-\mu_{loss})^T V_{loss}^{-1}(x-\mu_{loss}) + \frac{1}{2}\log|V_{loss}|$$

Using this method, a series of slowly varying discriminant functions, which can provide a fine measure of game positions for different stages, were generated. The functions for different stages are listed as the following, where x, y and z represent evaluation values of Dominance, Min-Mobility and Regions respectively.

Move 5:

$$-0.997422 - 0.0015418\,x - 5.42157\times10^{-7}x^2 + 0.00140515\,y + 5.20371\times10^{-6}xy - 8.21226\times10^{-7}y^2 - 0.00604553\,z + 3.71359\times10^{-7}xz + 5.61574\times10^{-6}yz + 0.00202669\,z^2$$

Move 6:

$$-0.275975 + 0.000544858\,x - 3.43234\times10^{-6}x^2 + 0.00139084\,y + 1.60483\times10^{-6}xy - 7.66591\times10^{-7}y^2 + 0.00411963\,z + 0.0000605542\,xz + 2.46082\times10^{-7}yz + 0.00184263\,z^2$$

Move 7:

$$-2.15991 + 0.000231133\,x - 4.25596\times10^{-7}x^2 + 0.00404007\,y + 1.40761\times10^{-6}xy - 2.21537\times10^{-6}y^2 + 0.0047651\,z + 0.0000160546\,xz - 0.0000313477\,yz + 0.00132382\,z^2$$

Move 8:

$$0.571769 + 0.00177234\,x + 5.98155\times10^{-7}\,x^2 + 0.00148389\,y - 2.65102\times10^{-6}\,x\,y - 1.55761\times10^{-6}\,y^2 + 0.0037633\,z + 0.000042359\,x\,z + 0.0000105338\,y\,z - 0.00116309\,z^2$$

Move 9:

$$-1.88075 + 0.00163066\,x + 2.2281\times10^{-7}\,x^2 + 0.0026073\,y + 2.27333\times10^{-7}\,x\,y - 7.82549\times10^{-7}\,y^2 + 0.00779871\,z - 0.0000109059\,x\,z - 0.0000233588\,y\,z + 0.000603111\,z^2$$

Move 10:

$$0.675081 + 0.00196876\,x - 2.93109\times10^{-6}\,x^2 + 0.00168869\,y + 2.12433\times10^{-6}\,x\,y - 1.7376\times10^{-6}\,y^2 + 0.00444221\,z + 9.34201\times10^{-6}\,x\,z + 0.0000187061\,y\,z - 0.000425721\,z^2$$

Move 11:

$$-1.87805 + 0.00385968\,x + 1.4842\times10^{-7}\,x^2 + 0.00285252\,y - 3.64952\times10^{-6}\,x\,y - 5.38946\times10^{-7}\,y^2 + 0.00505107\,z + 0.0000142436\,x\,z - 0.0000149833\,y\,z + 0.000277929\,z^2$$

Move 12:

$$0.457485 + 0.0027442\,x - 2.54617\times10^{-6}\,x^2 + 0.00149188\,y + 2.02435\times10^{-6}\,x\,y - 1.0778\times10^{-6}\,y^2 + 0.0098241\,z + 0.0000242371\,x\,z + 1.80991\times10^{-6}\,y\,z - 0.000164668\,z^2$$

Move 13:

$$-1.82138 + 0.00404531\,x + 4.19231\times10^{-7}\,x^2 + 0.00179823\,y - 3.43449\times10^{-6}\,x\,y + 1.20298\times10^{-7}\,y^2 + 0.0103534\,z + 0.0000208647\,x\,z - 0.0000203898\,y\,z + 0.000308181\,z^2$$

Move 14:

$$0.669872 + 0.00380293\,x - 5.31363\times10^{-6}\,x^2 + 0.00122248\,y + 7.01452\times10^{-7}\,x\,y - 1.58175\times10^{-6}\,y^2 + 0.0128523\,z + 9.28096\times10^{-6}\,x\,z + 7.47522\times10^{-6}\,y\,z - 0.000148753\,z^2$$

Move 15:

$$-1.77574 + 0.00351681\,x + 5.29749\times10^{-6}\,x^2 + 0.00081019\,y - 2.15915\times10^{-7}\,x\,y + 8.44543\times10^{-7}\,y^2 + 0.0146666\,z + 0.0000127563\,x\,z - 0.0000157202\,y\,z + 0.000202421\,z^2$$

Move 16:

$$0.341292 + 0.00494561\,x - 3.93172\times10^{-6}\,x^2 + 0.000858505\,y + 2.15312\times10^{-6}\,x\,y - 8.49635\times10^{-7}\,y^2 + 0.0174589\,z - 0.0000154577\,x\,z - 6.86939\times10^{-6}\,y\,z - 0.0000201464\,z^2$$

Move 17:

$$-1.27815 + 0.00538262\,x - 2.53047\times10^{-6}\,x^2 + 0.000623036\,y - 1.55175\times10^{-6}\,x\,y + 8.55594\times10^{-7}\,y^2 + 0.0142406\,z + 0.0000247121\,x\,z - 9.99416\times10^{-6}\,y\,z + 0.0000797396\,z^2$$

Move 18:

$$0.199416 + 0.00529466\,x - 8.19045\times10^{-7}\,x^2 + 0.000320933\,y + 3.09733\times10^{-6}\,x\,y - 3.14474\times10^{-7}\,y^2 + 0.0198604\,z - 0.0000308927\,x\,z - 5.67722\times10^{-6}\,y\,z - 0.0000300763\,z^2$$

Move 19:

$$-1.41592 + 0.00733786\,x - 3.6445\times10^{-6}\,x^2 + 0.000555011\,y - 1.45849\times10^{-6}\,x\,y + 1.08294\times10^{-6}\,y^2 + 0.0157919\,z + 0.0000247817\,x\,z - 0.0000142676\,y\,z + 0.0000977267\,z^2$$

Move 20:

$$0.261509 + 0.00544465\,x + 1.28232\times10^{-6}\,x^2 + 0.000523453\,y + 1.08382\times10^{-6}\,x\,y - 7.71907\times10^{-7}\,y^2 + 0.019619\,z - 0.000023834\,x\,z + 6.02909\times10^{-6}\,y\,z - 0.000061187\,z^2$$

Move 21:

$$-1.2491 + 0.00632449\,x - 2.63308\times10^{-6}\,x^2 + 0.000384885\,y + 7.10138\times10^{-7}\,x\,y + 8.4948\times10^{-7}\,y^2 + 0.0200342\,z + 0.0000102646\,x\,z - 0.0000163035\,y\,z + 0.0000953523\,z^2$$

Move 22:

$$0.197349 + 0.00669165\,x + 3.31637\times10^{-6}\,x^2 + 0.000741352\,y - 1.72331\times10^{-6}\,x\,y - 8.79389\times10^{-7}\,y^2 + 0.0197532\,z - 0.0000218525\,x\,z + 0.0000180918\,y\,z - 0.0000935729\,z^2$$

Move 23:

$$-1.4514 + 0.00840189\,x - 4.09187\times10^{-6}\,x^2 + 0.000493894\,y - 6.61764\times10^{-7}\,x\,y + 9.46314\times10^{-7}\,y^2 + 0.0173127\,z + 0.0000176894\,x\,z - 0.0000169801\,y\,z + 0.000105015\,z^2$$

Move 24:

$$0.148123 + 0.00759464\,x + 2.14168\times10^{-6}\,x^2 + 0.000507131\,y - 4.01306\times10^{-7}\,x\,y - 9.19167\times10^{-7}\,y^2 + 0.0193556\,z - 0.0000275413\,x\,z + 0.0000150671\,y\,z - 0.0000722716\,z^2$$

Move 25:

$$-1.47002 + 0.00890548\,x - 3.89289\times10^{-6}\,x^2 + 0.000531111\,y + 3.9986\times10^{-7}\,x\,y + 8.84134\times10^{-7}\,y^2 + 0.0148957\,z + 0.0000223929\,x\,z - 0.0000158818\,y\,z + 0.0000776215\,z^2$$

Move 26:

$$0.159465 + 0.008858\,x - 3.54746\times10^{-6}\,x^2 + 0.000625495\,y - 1.33753\times10^{-6}\,x\,y - 7.06728\times10^{-7}\,y^2 + 0.0174154\,z - 0.0000189698\,x\,z + 0.000014148\,y\,z - 0.0000622167\,z^2$$

Move 27:

$$-1.48221 + 0.00992909\,x - 4.87572\times10^{-6}\,x^2 + 0.000303827\,y + 7.04131\times10^{-7}\,x\,y + 8.29595\times10^{-7}\,y^2 + 0.0136949\,z + 0.0000243875\,x\,z - 0.0000161583\,y\,z + 0.0000629963\,z^2$$

Move 28:

$$0.0234448 + 0.0116827\,x - 9.22348\times10^{-6}\,x^2 + 0.000572712\,y - 1.79459\times10^{-7}\,x\,y - 4.54154\times10^{-7}\,y^2 + 0.0128658\,z - 2.52242\times10^{-6}\,x\,z + 7.25174\times10^{-6}\,y\,z - 0.0000400919\,z^2$$

Move 29:

$$-1.52986 + 0.010753\,x - 2.09874\times10^{-6}\,x^2 + 0.000376512\,y + 6.18421\times10^{-7}\,x\,y + 6.92063\times10^{-7}\,y^2 + 0.0136772\,z + 7.87649\times10^{-6}\,x\,z - 0.0000120484\,y\,z + 0.0000460114\,z^2$$

Move 30:

$$-0.117816 + 0.0117813\,x - 6.94781\times10^{-6}\,x^2 + 0.000813082\,y - 2.0933\times10^{-6}\,x\,y - 4.62371\times10^{-7}\,y^2 + 0.0118742\,z - 3.95867\times10^{-6}\,x\,z + 7.50346\times10^{-6}\,y\,z - 0.0000168311\,z^2$$

Move 31:

$$-1.40136 + 0.0105505\,x - 1.25999\times10^{-6}\,x^2 + 0.000719051\,y + 1.69102\times10^{-6}\,x\,y + 3.31923\times10^{-7}\,y^2 + 0.0107472\,z + 7.10553\times10^{-6}\,x\,z - 8.24051\times10^{-6}\,y\,z + 0.0000275633\,z^2$$

Move 32:

$$-0.181637 + 0.0132378\,x - 5.57432\times10^{-6}\,x^2 + 0.000997469\,y - 3.09664\times10^{-6}\,x\,y - 1.70471\times10^{-7}\,y^2 + 0.00914412\,z + 2.50853\times10^{-6}\,x\,z + 5.82315\times10^{-6}\,y\,z - 0.0000178108\,z^2$$

Move 33:

$$-1.35425 + 0.0120342\,x + 1.36855\times10^{-6}\,x^2 + 0.000710516\,y + 1.55918\times10^{-6}\,x\,y + 2.37216\times10^{-7}\,y^2 + 0.0103306\,z + 5.01664\times10^{-6}\,x\,z - 8.22007\times10^{-6}\,y\,z + 0.0000115233\,z^2$$

Move 34:

$$-0.221935 + 0.0143446\,x - 6.53812\times10^{-6}\,x^2 + 0.000958794\,y - 1.0975\times10^{-6}\,x\,y - 2.96128\times10^{-7}\,y^2 + 0.00895052\,z - 5.55709\times10^{-6}\,x\,z + 3.30902\times10^{-6}\,y\,z + 9.75586\times10^{-7}\,z^2$$

Move 35:

$$-1.34031 + 0.0128117\,x + 2.8601\times10^{-6}\,x^2 + 0.000666913\,y - 2.8446\times10^{-7}\,x\,y + 2.68348\times10^{-7}\,y^2 + 0.00981992\,z + 5.02738\times10^{-6}\,x\,z - 3.89887\times10^{-6}\,y\,z + 1.49128\times10^{-6}\,z^2$$

Move 36:

$$-0.255715 + 0.0145341\,x - 5.31408\times10^{-6}\,x^2 + 0.000641949\,y + 1.22053\times10^{-6}\,x\,y - 7.35374\times10^{-8}\,y^2 + 0.00903889\,z - 0.0000167525\,x\,z - 6.8738\times10^{-7}\,y\,z + 0.0000126681\,z^2$$

Move 37:

$$-1.17315 + 0.0136987\,x - 1.66189\times10^{-6}\,x^2 + 0.000662974\,y - 5.23591\times10^{-7}\,x\,y + 3.183\times10^{-7}\,y^2 + 0.00799443\,z + 0.0000150853\,x\,z - 3.006\times10^{-6}\,y\,z - 5.52443\times10^{-6}\,z^2$$

Move 38:

$$-0.108538 + 0.0145749\,x - 1.88254\times10^{-6}\,x^2 + 0.000699398\,y + 4.22236\times10^{-7}\,x\,y - 2.41759\times10^{-7}\,y^2 + 0.00727268\,z - 0.0000147027\,x\,z + 1.50719\times10^{-6}\,y\,z + 9.72286\times10^{-7}\,z^2$$

Move 39:

$$-1.281 + 0.0150188\,x + 1.28283\times10^{-6}\,x^2 + 0.000544859\,y + 1.45805\times10^{-6}\,x\,y + 1.55606\times10^{-7}\,y^2 + 0.00656979\,z + 9.40797\times10^{-7}\,x\,z - 4.21504\times10^{-6}\,y\,z + 9.3528\times10^{-6}\,z^2$$

Move 40:

$$0.00821834 + 0.0163908\,x - 6.71716\times10^{-6}\,x^2 + 0.000534039\,y - 1.36918\times10^{-7}\,xy - 8.74644\times10^{-8}\,y^2 + 0.00699702\,z - 3.21866\times10^{-6}\,xz + 9.74231\times10^{-7}\,yz - 9.09629\times10^{-6}\,z^2$$

Move 50:

$$0.478788 + 0.0238406\,x - 0.0000104875\,x^2 + 0.000705451\,y - 9.36295\times10^{-7}\,xy + 8.79864\times10^{-8}\,y^2 + 0.00503608\,z + 5.84642\times10^{-7}\,xz - 6.46336\times10^{-7}\,yz - 4.99071\times10^{-6}\,z^2$$

Figure 4.26 Evaluation Functions Generated by Bayesian Learning

Using Evaluation Functions Generated by Bayesian Learning and the same three features as the original version of Mulan, a new version, Mulan--Bayesian, was created. The original version, Mulan-Original, used a linear combination of three features, 0.2*Regions+0.5*Dominance+0.3*Min-Mobility, which was tuned using tournaments as described in the previous section. We set Mulan-Bayesian and Mulan-Original to play against each other based on two criteria: time and number of boards evaluated per move. For each tournament, we selected 100 initial positions from the opening book. For each selected initial position, two games between the two versions of Mulan were played. Each version played once as white and once as black. The results are listed in Table 4.25.

Criterion	Result (Win-Loss)	Winning Percentage
30 Seconds per Move	55: 145	27.5%
100000 leaf nodes per Move	68: 132	34%

Table 4.25 Tournament Results between Mulan-Bayesian and Mulan-Origianl

Although Lee et al got a very good improvement by applying Bayesian learning to the game of Othello, the results of applying it to Amazons are not very promising. By

comparing the characteristics of two games, we think there are three possible reasons for this.

First of all, we observed that the Mulan-Bayesian's performance is better when using board evaluations instead of time as criterion. This is a major problem with Bayesion learning, efficiency. Using the functions obtained from Bayesion learning, the number of floating point multiplications needed to combine N features is 2N(N+1). This substantially affects the speed of evaluation.

Secondly, mislabeled positions may decrease the accuracy of the evaluation function's coefficients. This problem is more harmful for Amazons than the game of Othello. In Amazons, we know less about how to play Amazons, so we don't have good experts to provide training data. So a poor subsequent move may cause a losing result for a winning position. In Lee et al's experiments on the game of Othello, this less likely happens since typically Logistello plays better than any expert.

Finally, the underlying distributions of the features we used in Mulan do not obay the Multivariate normal distribution assumption on which Bayesian learning largely depends. To investigate whether this assumption is true in our domain, the distribution of the three features from all 2000 training games were plotted in Figure 4.27-4.38. From these figures, we can see this assumption is not quite reasonable for our selected features.

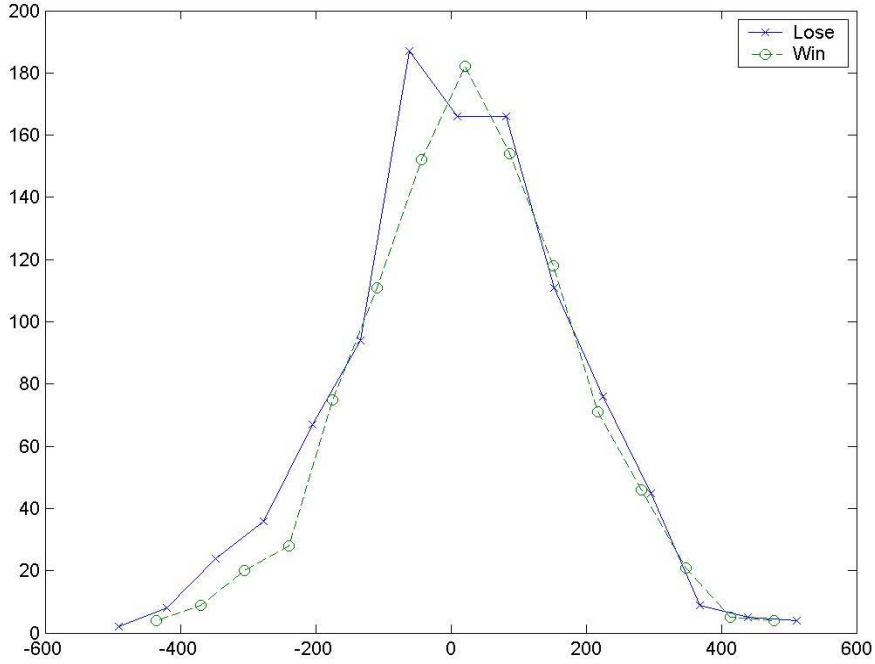

Figure 4.27 Normalized Dominance Feature at Move 10

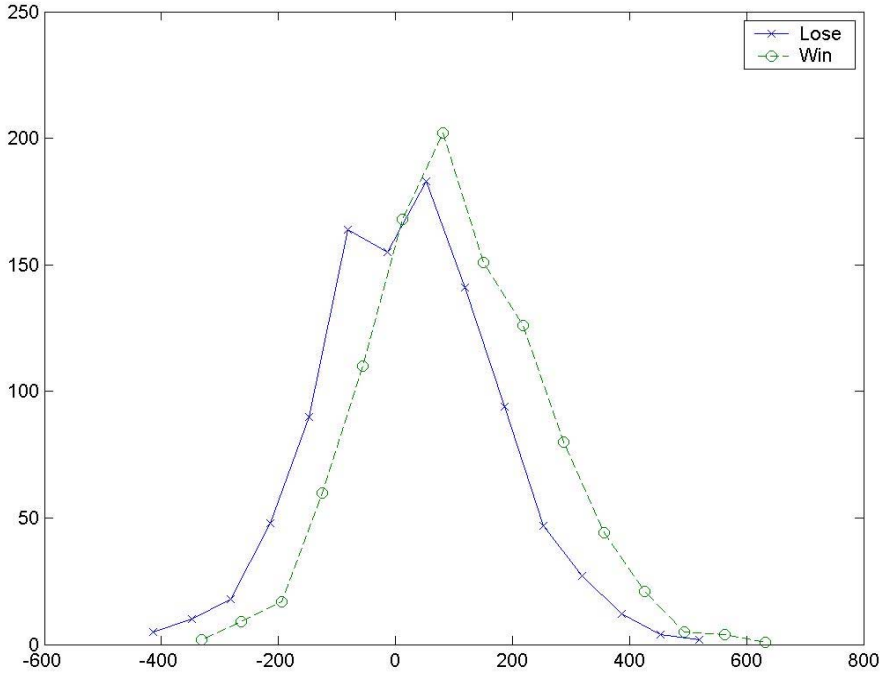

Figure 4.28 Normalized Dominance Feature at Move 20

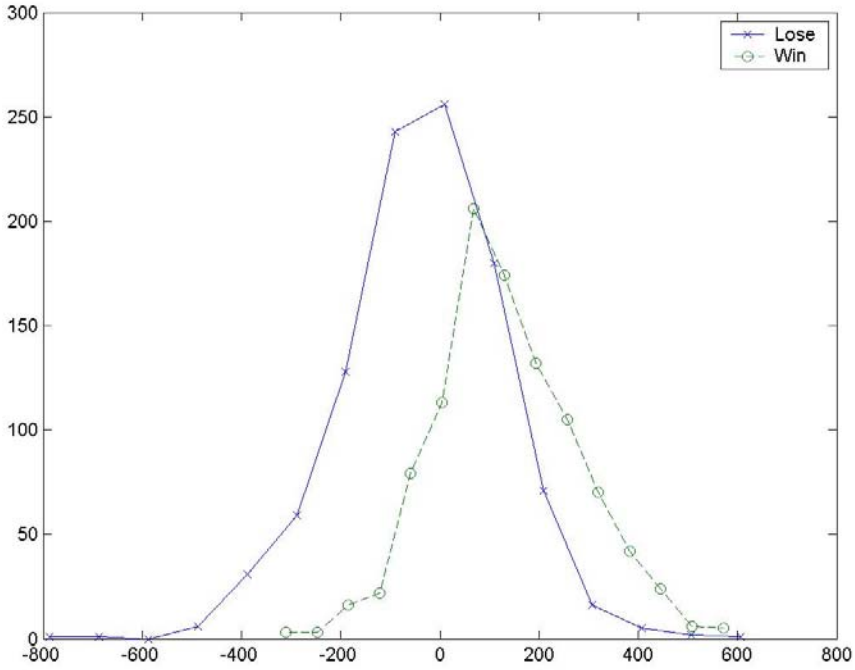

Figure 4.29 Normalized Dominance Feature at Move 30

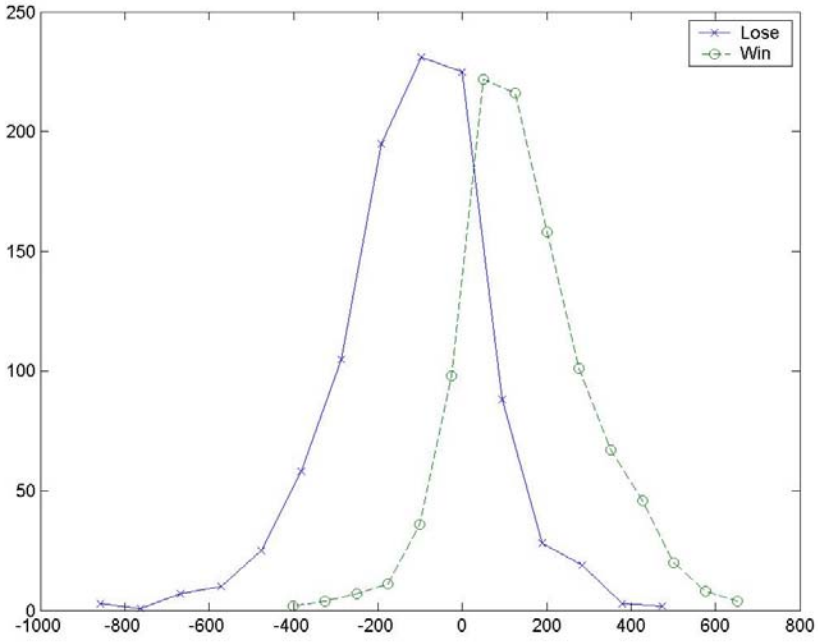

Figure 4.30 Normalized Dominance Feature at Move 40

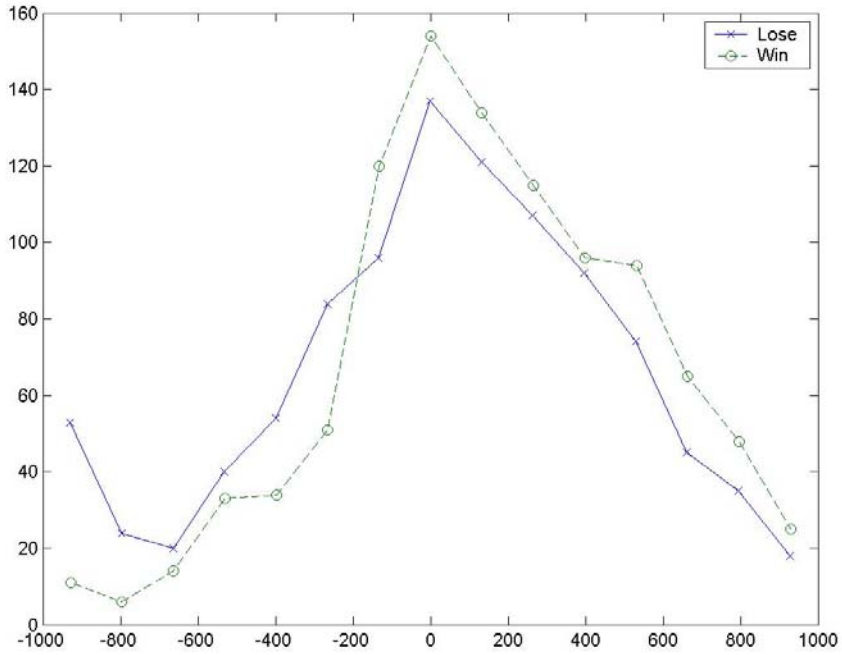

Figure 4.31 Normalized Min-Mobility Feature at Move 10

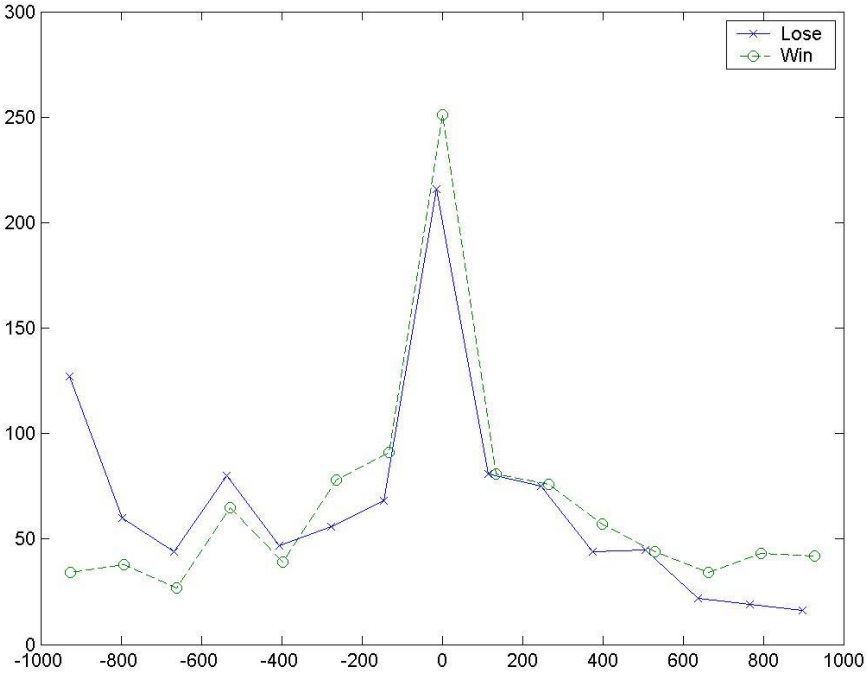

Figure 4.32 Normalized Min-Mobility Feature at Move 20

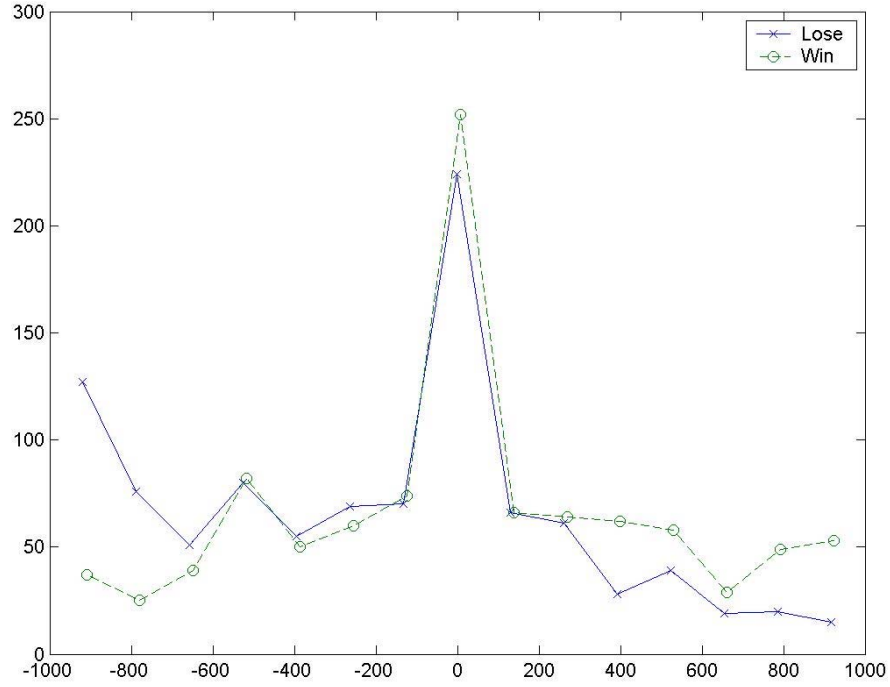

Figure 4.33 Normalized Min-Mobility Feature at Move 30

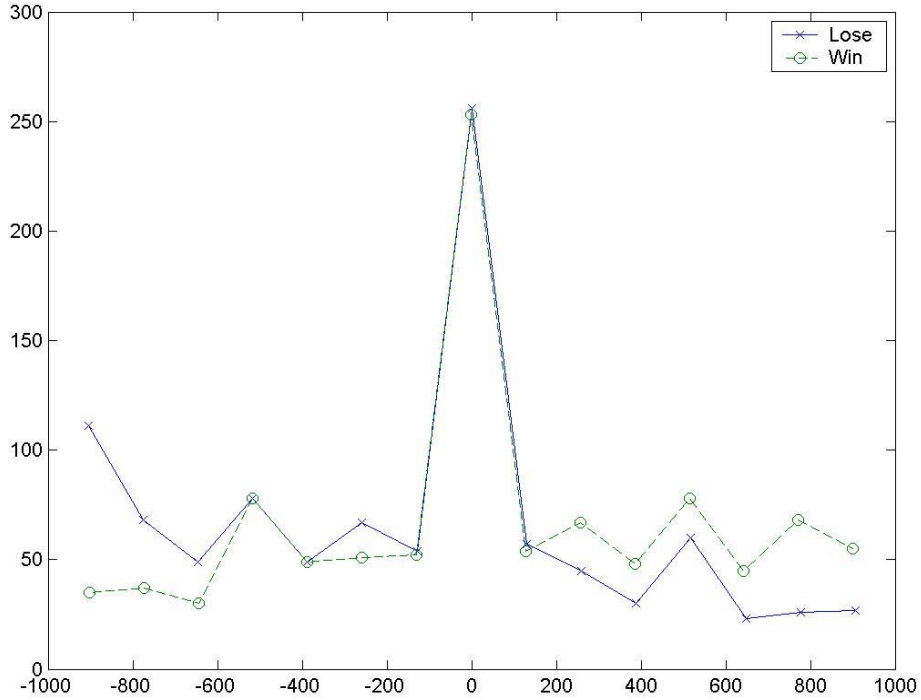

Figure 4.34 Normalized Min-Mobility Feature at Move 40

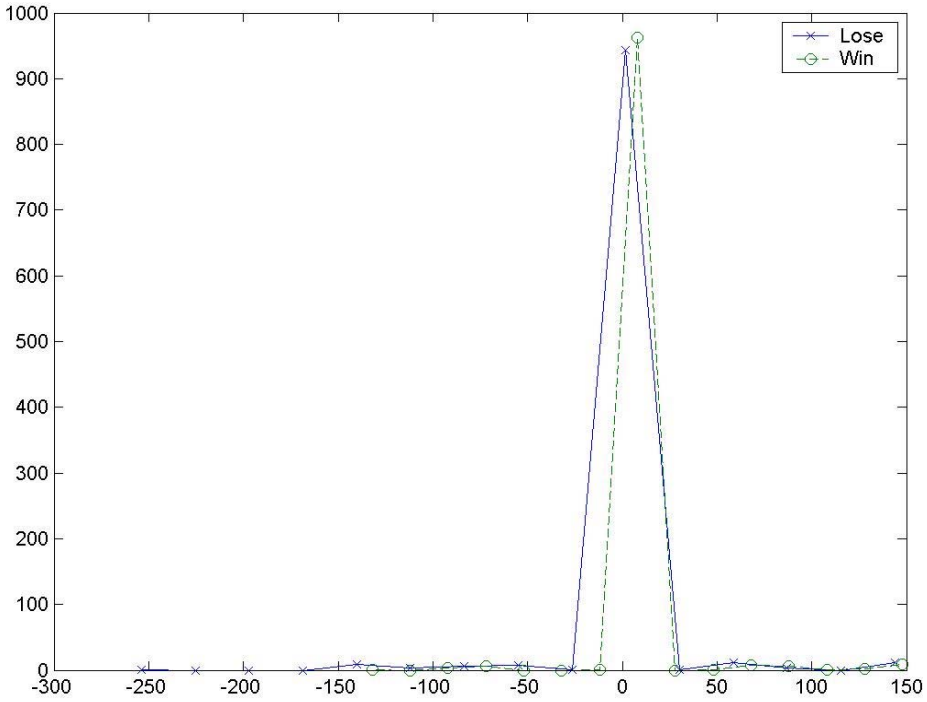

Figure 4.35 Normalized Regions Feature at Move 10

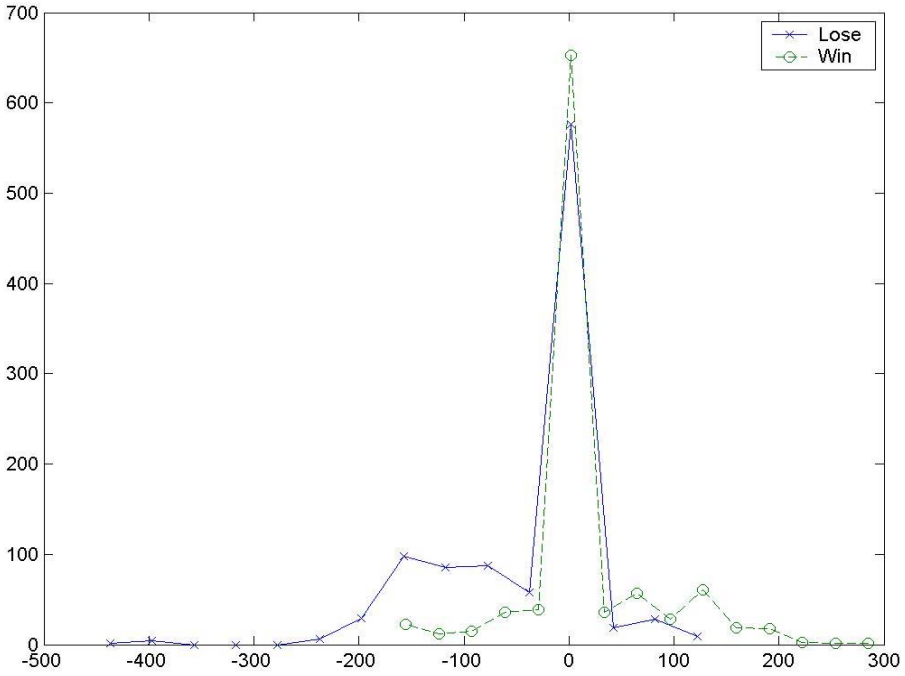

Figure 4.36 Normalized Regions Feature at Move 20

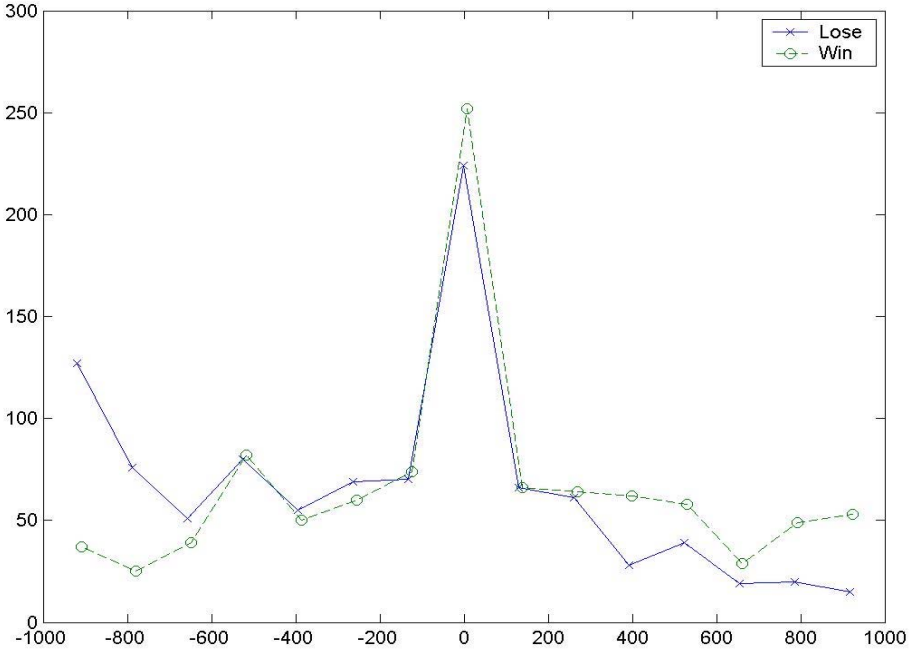

Figure 4.37 Normalized Regions Feature at Move 30

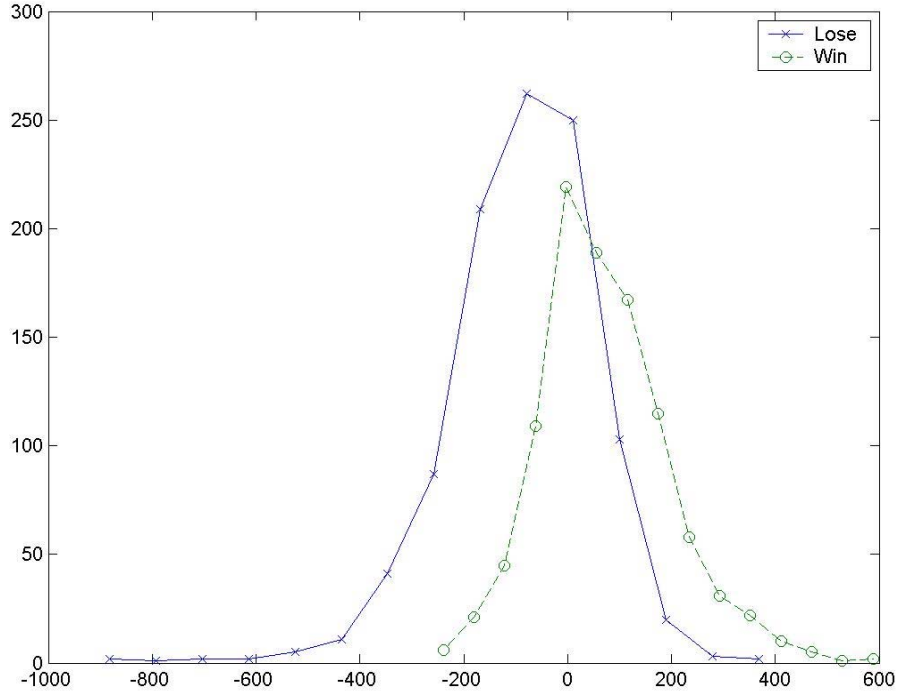

Figure 4.38 Normalized Regions Feature at Move 40

CHAPTER 5 CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

Throughout this thesis, we implemented and tested some major algorithms for selective and non-selective search. All these algorithms have been proved valuable in many domains, but their performance may vary for different games. Currently there are no published results about how well they perform in Amazon. Our experiments investigated this question experimentally. We found that for non-selective search, a combination of NegaScout, Iterative Deepening, Transposition Table, Killer Heuristic and History Heuristic can achieve the best result. We also tested two kinds of forward pruning enhancements, N-best search and Multi-ProbCut. Our results show that N-best search is superior to Multi-ProCut in Amazons. Two kinds of N-best selective search were implemented and tested in our experiment. One divides a move into two separated operations: queen-move and barricade-move. For each node, N promising queen-moves are selected first, and then M favorable barricade-moves are determined for each queen-move. The other kind of N-best selective search simply selects F promising moves for each node. We found that this second approach gets better results; when we chose F=40, we got a 69% winning percentage comparing with the best non-selective search.

Most successful game-playing programs apply heuristic evaluation functions at terminal nodes to estimate the probability that the current player will win. Typically, constructing a good evaluation function includes two working phrases:

1) Selecting good features.

2) Combining them appropriately to obtain a single numerical value.

Selecting features is important and difficult. We have to avoid too few features as well as redundant ones. We implemented the three most important published features of Amazons. Based on our practical playing experience on Amazons, we also designed four new features. To avoid redundant features, we divided all these features into three groups based on the board characteristics they evaluated. For each group, we used tournaments to select the most effective feature. The features we finally selected are Dominance, Min-Mobility and Regions.

Furthermore, we used tournaments to combine these features linearly. We divided the whole game to two phases: Before and after 30 moves. We found the best combination of features is 0.2*Regions+0.5*Dominance+0.3*Min-Mobility for the first 30 moves, and a pure Dominance feature after that.

The last question is whether we can apply Bayesian learning to Amazons effectively. To answer this question, we used our linear combination of features to create a version of Mulan called Mulan-Original. Using Mulan-Original as an expert, we collected training data and applied the Bayesian learning method to get a new version called Mulan-Bayesian. We used tournaments to compare these two versions of Mulan. The result shows that Bayesian learning doesn't work out in Mulan. There may be three possible reasons: 1) The efficiency problem of floating point multiplications needed for evaluations generated by Bayesian learning; 2) Mislabeling of positions in Bayesian learning; 3) The underlying distributions of the features we selected don't meet the Multivariate normal distribution assumption required by Bayesian learning.

5.2 Future Work

In this section, some techniques and ideas related with future studies are discussed.

5.2.1 Finding more useful features

Finding more useful features is critical for creating a strong game playing program. It also requires both expert game knowledge and programming skill because of the well-known tradeoff between the complexity of the evaluation function and the number of positions we can evaluate in a given time. Thought Mulan has combined three important features and evolved to a strong Amazons playing program, there is still considerable room for improvement.

5.2.2 Designing new automatic features combination method

Though our experimental results on Bayesian Learning are not very impressive, we still believe automatic evaluation function construction has a brilliant future in Amazons. Many other new methods of automatic evaluation function have been proposed and achieved promising results on some complicated games **[34],** and we believe that it is also possible to find out a good approach to suit Amazons.

5.2.3 Quiescence Search

Although we don't have experiments to show this, we believe that Quiescence Search is beneficial for improving the play strength of Mulan. In Amazons, some "quiescent" positions such as "dead piece", "region invasion" should be assessed

accurately, but they cannot be evaluated correctly without further search. Quiescence Search can increase the search depth for positions that have potential and should be explored further. Currently, applying Quiescence Search in Amazons is difficult since humans are still in the learning stage and haven't accumulated enough information about "quiescent" positions. So further research is necessary to investigate how to implement Quiescence Search in Amazons and how useful Quiescence Search is in Amazon.

5.2.4 Opening books and Endgame improvement

Creating large endgame databases can extend the search horizon in the endgame phase. This is the approach taken by Arrow using combinatorial game theory. However, we find that combinatorially complicated positions rarely occur in play and the large-scale dynamics of the early and mid-game seems to be more important.

At the beginning of the game, a large opening book would be helpful. But one of the charms of Amazons is that we still don't know enough about the game to be sure what the good opening moves are! This is another area for future research.

APPENDICES

Appendix A. Tournament results of Mulan (Min-Mobility + Dominance version)

Game1: Yamazon vs Mulan (10 Sec. per Move and Yamazon plays first)

Mulan wins 5 squares.

D1-D7 (G7)	D10-F8 (B4)
J4-J6 (F10)	G10-E8 (J3)
J6-I6 (F6)	E8-E2 (F1)
G1-F2 (B6)	J7-H9 (B3)
I6-H7 (C2)	E2-H5 (B5)
A4-A1 (A6)	A7-C9 (C3)
A1-D1 (D2)	H9-E6 (E2)
H7-I8 (H9)	H5-I5 (G3)
F2-F5 (E5)	F8-H8 (H2)
F5-F4 (I4)	I5-H5 (H7)
F4-G4 (H4)	E6-D5 (F7)
G4-E4 (G4)	H5-J7 (I7)
I8-I9 (J8)	H8-C8 (H8)
D1-E1 (F2)	D5-D3 (F3)
I9-H10 (C5)	J7-J5 (F5)
D7-D9 (E8)	D3-D6 (G9)
E4-D5 (E6)	C9-D10 (C9)
D9-C10 (A8)	C8-B9 (B7)
C10-D9 (D7)	D10-E9 (D10)

D9-D8 (B8)

D5-E4 (D5)

D8-E9 (F8)

E9-D8 (E7)

D8-C7 (C6)

H10-I10 (H10)

C7-D8 (C7)

E1-A1 (E1)

A1-A4 (A5)

A4-A3 (A4)

A3-C1 (D1)

E4-D4 (E4)

D4-D3 (D4)

D3-C4 (D3)

C1-A3 (C1)

I10-J10 (I10)

J10-J9 (J10)

J9-I9 (J9)

I9-I8 (I9)

A3-A2 (A3)

A2-B2 (B1)

B2-A2 (B2)

A2-A1 (A2)

J5-G5 (E3)

E9-F9 (G10)

B9-C8 (D9)

F9-E10 (E9)

G5-H6 (F4)

H6-H5 (J7)

C8-B9 (C8)

B9-B10 (B9)

B10-A9 (A10)

A9-B10 (A9)

B10-C10 (B10)

H5-H6 (H5)

H6-I5 (I6)

I5-J6 (J5)

J6-I5 (J6)

I5-H6 (G5)

E10-F9 (E10)

H6-I5 (H6)

I5-J4 (I5)

J4-I3 (J4)

I3-I2 (I3)

I2-H3 (I2)

H3-G2 (H3)

Game2: Mulan vs Yamazon (10 Sec. per Move and Mulan plays first)

Mulan wins 9 squares.

D1-D8 (E9)	A7-E3 (A7)
J4-J5 (E10)	D10-B8 (I1)
G1-G8 (F9)	J7-G4 (C4)
G8-H9 (G9)	G10-J10 (J7)
A4-A1 (I9)	G4-E6 (D5)
J5-F5 (J9)	E6-H6 (G5)
H9-E6 (I10)	H6-G6 (G8)
D8-C7 (B7)	B8-C8 (B8)
C7-H7 (I7)	C8-D7 (D6)
E6-E8 (C8)	E3-C3 (A3)
E8-D8 (E8)	C3-B2 (B1)
F5-G4 (E2)	G6-I4 (G6)
G4-H3 (C3)	B2-B6 (B2)
H3-E3 (C5)	I4-G4 (F3)
E3-F4 (F7)	D7-F5 (D7)
H7-H6 (H3)	F5-E5 (E3)
F4-G3 (H4)	E5-G7 (H7)
H6-H5 (H6)	G7-E5 (E7)
G3-E1 (G3)	E5-D4 (D2)
E1-D1 (A4)	D4-D3 (C2)
D8-C7 (C6)	B6-B3 (B6)

A1-A2 (A1)

D1-E1 (F2)

E1-F1 (G2)

F1-G1 (H2)

G1-H1 (I2)

H1-G1 (H1)

G1-F1 (G1)

F1-E1 (F1)

E1-D1 (E1)

D1-C1 (D1)

H5-I6 (H5)

I6-I5 (I6)

I5-J6 (J5)

J6-I5 (J6)

I5-I4 (I5)

I4-I3 (I4)

I3-J4 (J3)

J4-I3 (J4)

I3-J2 (I3)

J2-J1 (J2)

C7-D8 (C7)

D3-D4 (D3)

D4-H8 (J8)

H8-H10 (F10)

H10-G10 (H10)

G10-H9 (G10)

H9-I8 (H9)

I8-H8 (I8)

H8-G7 (F8)

G7-H8 (G7)

B3-B4 (B3)

B4-B5 (B4)

B5-A5 (B5)

A5-A6 (A5)

G4-D4 (G4)

D4-E4 (D4)

E4-F4 (E4)

F4-F5 (F4)

F5-E5 (F5)

E5-E6 (F6)

E6-E5 (E6)

Game3: Invader vs Mulan (10 Sec. per Move and Invader plays first)

Mulan wins 11 squares.

1. G1 - G7 (D7)	2. G10 - E8 (I4)
3. A4 - A6 (E10)	4. D10 - F8 (A3)
5. A6 - B6 (E6)	6. F8 - F2 (E1)
7. J4 - H2 (J4)	8. A7 - C9 (H4)
9. D1 - C2 (C8)	10. E8 - G6 (G3)
11. C2 - F5 (F9)	12. G6 - G4 (I2)
13. F5 - F4 (J8)	14. G4 - F5 (H3)
15. F4 - F3 (D5)	16. J7 - I7 (I6)
17. F3 - D3 (C2)	18. I7 - H8 (G8)
19. H2 - G1 (G2)	20. F2 - C5 (F2)
21. D3 - E2 (C4)	22. C5 - B4 (C3)
23. E2 - H5 (D9)	24. F5 - F4 (C7)
25. G1 - F1 (D3)	26. F4 - E3 (E2)
27. G7 - F8 (F4)	28. C9 - A7 (B7)
29. H5 - H7 (E4)	30. B4 - E7 (B4)
31. B6 - A6 (D6)	32. E7 - G7 (E7)
33. H7 - I8 (I7)	34. H8 - H9 (H7)
35. A6 - A5 (C5)	36. E3 - C1 (D1)
37. A5 - A4 (A6)	38. C1 - B2 (B3)
39. A4 - A5 (B6)	40. G7 - F7 (G7)
41. F8 - D10 (B8)	42. A7 - A9 (B10)

43. D10 - C9 (B9)
44. F7 - F8 (D8)
45. C9 - D10 (C10)
46. F8 - E8 (E9)
47. I8 - I9 (J9)
48. E8 - F8 (H10)
49. I9 - I8 (I10)
50. H9 - H8 (H9)
51. I8 - J7 (I8)
52. F8 - F5 (J5)
53. J7 - J6 (I5)
54. F5 - F6 (F7)
55. D10 - C9 (D10)
56. F6 - F5 (F6)
57. J6 - J7 (J6)
58. F5 - G6 (G5)
59. A5 - A4 (A5)
60. G6 - H5 (H6)
61. A4 - C6 (A4)
62. H5 - G6 (H5)
63. C6 - B5 (C6)
64. G6 - F5 (G6)
65. F1 - H1 (F1)
66. F5 - G4 (F5)
67. H1 - H2 (H1)
68. G4 - F3 (G4)
69. H2 - I1 (J2)
70. F3 - E3 (C1)
71. I1 - H2 (G1)
72. E3 - D4 (E5)
73. H2 - I3 (J3)
74. D4 - E3 (D4)
75. I3 - H2 (I3)
76. B2 - B1 (B2)
77. H2 - I1 (H2)
78. E3 - D2 (E3)
79. I1 - J1 (I1)
80. B1 - A2 (A1)

Game4: Mulan vs Invader (10 Sec. per Move and Mulan plays first)

Invader wins 13 squares.

1. D1 - D8 (E9)	2. G10 - G3 (G8)
3. J4 - F4 (F10)	4. D10 - C10 (C1)
5. D8 - C8 (D9)	6. A7 - D4 (D7)
7. C8 - C9 (B9)	8. D4 - F6 (G5)
9. G1 - H2 (J4)	10. G3 - F2 (H4)
11. F4 - E4 (D4)	12. J7 - I6 (H7)
13. A4 - A3 (A10)	14. F6 - D6 (E7)
15. E4 - F5 (G6)	16. D6 - B6 (F6)
17. C9 - B8 (C9)	18. F2 - C2 (C8)
19. H2 - I2 (E2)	20. I6 - I4 (H5)
21. F5 - E6 (C6)	22. C2 - B3 (D5)
23. E6 - F7 (F8)	24. I4 - I5 (G7)
25. B8 - G3 (D3)	26. I5 - I4 (I3)
27. I2 - J1 (H3)	28. C10 - E10 (F9)
29. G3 - C7 (B7)	30. I4 - J3 (H1)
31. J1 - I1 (I2)	32. B6 - B4 (B6)
33. I1 - F4 (D2)	34. J3 - J2 (J1)
35. A3 - A5 (C5)	36. J2 - I1 (G3)
37. F4 - F2 (H2)	38. B3 - D1 (A4)
39. A5 - B5 (A5)	40. D1 - G1 (E1)
41. B5 - C4 (C2)	42. E10 - B10 (E10)

43. C7 - E5 (A9)
44. G1 - G2 (E4)
45. C4 - B3 (C3)
46. G2 - F3 (F5)
47. E5 - F4 (E3)
48. F3 - G2 (F1)
49. B3 - A3 (B3)
50. B4 - B5 (B4)
51. F2 - F3 (F2)
52. B5 - A6 (A8)
53. F4 - B8 (A7)
54. B10 - D10 (B10)
55. A3 - A2 (A3)
56. D10 - C10 (D10)
57. A2 - A1 (A2)
58. A6 - C4 (A6)
59. A1 - B2 (B1)
60. C4 - B5 (C4)
61. B2 - A1 (B2)
62. I1 - J2 (I1)
63. F7 - E8 (F7)
64. J2 - J3 (J2)
65. E8 - D8 (C7)
66. J3 - I4 (I10)
67. D8 - E8 (D8)
68. I4 - I8 (I9)
69. F3 - F4 (F3)
70. G2 - G1 (G2)
71. F4 - E5 (E6)
72. I8 - J7 (J10)
73. E5 - F4 (G4)
74. J7 - I8 (J9)
75. F4 - E5 (D6)
76. I8 - J7 (I7)
77. E5 - F4 (E5)
78. J7 - H9 (H8)

Appendix B. Tournament results of Mulan (0.3*Min-Mobility + 0.5*Dominance + 0.2*Regions version)

Game1: Yamazon vs Mulan (10 Sec. per Move and Yamazon plays first)

Mulan wins 11 squares.

D1-D7 (G7)	D10-F8 (B4)
J4-J6 (F10)	G10-E8 (J3)
J6-I6 (F6)	F8-C5 (F2)
G1-G4 (G6)	C5-D5 (B5)
D7-C7 (E5)	D5-B3 (A3)
A4-A6 (B6)	A7-B7 (A7)
I6-G8 (D5)	B3-D1 (A4)
C7-C4 (F1)	D1-F3 (F5)
C4-C5 (E3)	B7-C6 (B7)
C5-C4 (E2)	C6-C5 (C9)
G4-H4 (D4)	F3-I3 (H3)
G8-F9 (F8)	J7-H9 (H5)
F9-D9 (C8)	C5-D6 (C5)
D9-E9 (D9)	D6-A9 (C7)
E9-F9 (E9)	H9-G9 (G8)
H4-I5 (I4)	G9-I7 (G9)
F9-G10 (I8)	I7-J8 (H10)
I5-G3 (F4)	J8-H6 (H9)

G3-J6 (I7)

J6-J8 (J5)

C4-D3 (C4)

J8-J10 (J6)

D3-E4 (F3)

A6-A5 (A6)

J10-J8 (J7)

J8-J9 (J8)

J9-J10 (J9)

J10-I10 (J10)

E4-D3 (E4)

D3-D1 (E1)

D1-A1 (D1)

A1-A2 (A1)

A2-D2 (D3)

D2-A2 (D2)

A2-B1 (C1)

B1-A2 (B1)

A2-B2 (A2)

B2-C2 (B2)

C2-C3 (C2)

C3-B3 (C3)

I3-J4 (I5)

E8-F9 (E10)

J4-H2 (G2)

H6-H8 (I9)

H8-H7 (I6)

H2-I3 (J4)

I3-H4 (G5)

H4-I3 (H4)

I3-I2 (I3)

I2-I1 (I2)

I1-J2 (J1)

J2-I1 (J2)

I1-H1 (I1)

H1-G1 (H1)

G1-H2 (G1)

H2-G3 (H2)

G3-G4 (G3)

H7-H8 (H6)

H8-H7 (H8)

F9-E8 (F9)

E8-D8 (E8)

D8-D7 (D8)

Game2: Yamazon vs Mulan (10 Sec. per Move and Mulan plays first)

Mulan wins 20 squares.

D1-D8 (E9)	A7-E3 (A7)
J4-J5 (E10)	D10-B8 (I1)
G1-G7 (B7)	J7-H5 (H8)
A4-C4 (C7)	H5-F5 (D5)
C4-C2 (D3)	F5-F6 (G6)
D8-C8 (B9)	G10-J7 (H7)
G7-E7 (C9)	E3-G5 (G2)
C8-E6 (C8)	G5-E5 (I5)
J5-I6 (J6)	J7-G10 (G8)
C2-C1 (G5)	F6-F4 (F8)
E6-G4 (G3)	E5-E1 (E6)
I6-I9 (F9)	E1-H1 (D1)
C1-E3 (E4)	H1-H3 (I4)
E7-F6 (E5)	F4-F2 (B2)
E3-D2 (E3)	G10-I8 (J9)
D2-E1 (E2)	B8-A9 (B8)
F6-F7 (F3)	A9-B10 (D10)
F7-E8 (C10)	F2-F1 (F2)
G4-H4 (G4)	H3-I3 (H3)
H4-H5 (H4)	I8-H9 (J7)
H5-I6 (I8)	I3-J4 (J5)

I9-H10 (I10)

H10-H9 (H10)

H9-I9 (G9)

I6-H5 (I6)

I9-H9 (I9)

H9-G10 (F10)

G10-H9 (G10)

E1-D2 (E1)

D2-C2 (D2)

C2-C1 (C2)

C1-B1 (C1)

B1-A1 (B1)

A1-A2 (A1)

A2-A4 (A2)

A4-B4 (A3)

H9-I9 (J10)

I9-J8 (H6)

J8-I7 (J8)

B10-A10 (B10)

A10-A9 (A8)

A9-A10 (A9)

J4-J1 (J4)

F1-G1 (F1)

G1-H1 (G1)

H1-H2 (H1)

J1-J2 (J1)

J2-I2 (J2)

I2-I3 (J3)

I3-I2 (I3)

Game3: Invader vs Mulan (10 Sec. per Move and Invader plays first)

Mulan wins 2 squares.

1. G1 - G7 (D7)	2. G10 - E8 (I4)
3. A4 - A6 (E10)	4. D10 - F8 (A3)
5. A6 - B6 (E6)	6. F8 - F2 (E1)
7. J4 - H2 (J4)	8. J7 - F3 (D3)
9. D1 - C2 (D1)	10. A7 - C9 (C3)
11. G7 - G8 (F9)	12. E8 - H5 (A5)
13. C2 - D2 (H6)	14. F3 - E4 (E3)
15. D2 - E2 (F3)	16. F2 - G3 (F2)
17. B6 - B8 (B3)	18. G3 - G7 (G1)
19. H2 - H4 (H3)	20. E4 - F4 (G4)
21. G8 - H8 (C8)	22. G7 - G9 (G8)
23. H8 - I9 (E5)	24. G9 - H8 (F6)
25. H4 - G3 (H4)	26. F4 - C4 (F4)
27. I9 - I8 (I5)	28. C4 - C7 (B7)
29. B8 - A7 (D4)	30. C7 - B6 (C7)
31. A7 - A10 (D10)	32. H5 - J7 (H7)
33. I8 - I9 (J8)	34. C9 - B9 (B10)
35. A10 - A7 (A10)	36. B9 - B8 (A8)
37. I9 - G9 (H9)	38. B8 - C9 (E7)
39. G9 - F8 (D8)	40. J7 - H5 (E8)
41. F8 - H10 (J10)	42. H5 - I6 (I10)

43. H10 - G9 (F10)
44. I6 - H5 (J7)
45. G9 - H10 (G10)
46. H8 - I7 (I9)
47. H10 - F8 (E9)
48. I7 - H8 (G7)
49. A7 - A6 (C4)
50. B6 - B5 (B6)
51. A6 - A7 (B8)
52. H8 - G9 (H10)
53. F8 - F7 (G6)
54. G9 - F8 (G9)
55. A7 - A6 (A7)
56. H5 - I6 (J5)
57. E2 - C2 (B1)
58. I6 - I7 (H8)
59. G3 - I1 (I3)
60. I7 - I8 (J9)
61. C2 - B2 (A1)
62. I8 - I7 (I8)
63. I1 - H2 (H1)
64. I7 - I6 (I7)
65. B2 - C2 (A2)
66. I6 - H5 (G5)
67. C2 - B2 (C1)
68. H5 - I6 (H5)
69. B2 - C2 (B2)
70. I6 - J6 (I6)
71. C2 - D2 (C2)
72. B5 - A4 (B5)
73. D2 - E2 (D2)
74. A4 - B4 (A4)
75. E2 - F1 (E2)
76. B4 - C5 (B4)
77. F1 - G2 (G3)
78. C5 - C6 (C5)
79. G2 - F1 (G2)
80. C6 - D6 (C6)
81. H2 - I2 (H2)
82. D6 - D5 (D6)
83. I2 - I1 (J1)
84. D5 - E4 (D5)
85. I1 - I2 (I1)
86. E4 - F5 (E4)
87. I2 - J2 (I2)
88. C9 - C10 (D9)

Game4: Mulan vs Invader (10 Sec. per Move and Mulan plays first)

Mulan wins 13 squares.

1. D1 - D8 (E9)	2. G10 - G3 (G8)
3. J4 - F4 (F10)	4. D10 - C10 (C1)
5. D8 - C8 (D9)	6. A7 - D4 (D7)
7. C8 - C9 (B9)	8. D4 - F6 (G5)
9. G1 - H2 (J4)	10. G3 - F2 (H4)
11. A4 - A6 (A10)	12. F2 - C2 (A4)
13. C9 - C3 (C9)	14. F6 - D6 (F6)
15. F4 - E4 (J9)	16. J7 - I6 (H7)
17. E4 - E8 (E2)	18. C2 - E4 (E3)
19. E8 - F8 (I5)	20. I6 - I9 (G7)
21. H2 - H3 (F3)	22. I9 - G9 (F9)
23. H3 - G4 (F4)	24. E4 - G6 (I6)
25. G4 - H3 (I2)	26. G6 - I4 (E8)
27. F8 - F7 (D5)	28. D6 - B4 (C5)
29. A6 - B6 (B5)	30. I4 - I3 (I4)
31. H3 - H2 (H3)	32. I3 - J2 (J1)
33. F7 - G6 (B1)	34. G9 - I7 (G9)
35. H2 - G1 (I1)	36. J2 - I3 (J3)
37. G1 - H1 (H2)	38. B4 - B3 (B2)
39. C3 - B4 (D4)	40. B3 - D3 (A3)
41. B4 - C3 (C2)	42. I7 - H6 (J8)

43. B6 - D6 (E5)
44. C10 - B10 (E10)
45. D6 - C7 (A9)
46. B10 - C10 (D10)
47. H1 - G1 (G4)
48. I3 - J2 (I3)
49. C3 - C4 (C3)
50. D3 - D1 (F1)
51. C4 - D3 (D2)
52. D1 - E1 (G3)
53. G1 - G2 (F2)
54. E1 - D1 (E1)
55. G6 - H5 (G6)
56. C10 - B10 (C10)
57. C7 - C8 (C7)
58. H6 - I7 (H6)
59. C8 - B7 (B8)
60. I7 - I9 (H8)
61. B7 - C8 (B7)
62. I9 - H9 (I10)
63. C8 - D8 (C8)
64. H9 - I9 (I7)
65. D8 - E7 (F8)
66. I9 - H9 (G10)
67. E7 - E6 (F7)
68. H9 - I9 (H10)
69. E6 - E7 (D8)
70. I9 - I8 (H9)
71. E7 - E6 (E7)
72. I8 - I9 (J10)
73. E6 - F5 (E6)
74. I9 - I8 (I9)
75. F5 - E4 (F5)
76. I8 - J7 (I8)
77. D3 - C4 (D3)
78. J7 - J5 (J7)
79. C4 - B3 (C4)
80. J5 - J6 (J5)
81. B3 - A2 (A1)

REFERENCES

[1] M. Buro. The Othello Match of the Year: Takeshi Murakami vs. Logistello. International Computer Chess Association Journal, 20(3):189-193, 1997.

[2] J. Schaefer. One Jump Ahead: Challenging Human Supremacy in Checkers. Springer-Verlag, 1997.

[3] M. S. Campbell, A. J. Hoane Jr., and F.-h. Hsu. Deep Blue. Artificial Intelligence, to appear in 2002

[4] J. Schaeffer and A. Plaat. Kasparov versus Deep Blue: The Re-match. International Computer Chess Association Journal, 20(2):95-101, 1997.

[5] A. L. Samuel. Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3(3):210-229, 1959.

[6] A.D. de Groot, Thought and Choice in Chess, Mouton, The Hague, 1965. Also 2nd Edition 1978.

[7] A. Newell, J.C. Shaw and H.A. Simon, "Chess playing programs and the problem of complexity", IBM J. Res. and Devel., 3, 1959, pp. 211 – 229.

[8] D.E. Knuth and R.W. Moore, An Analysis of Alpha-beta Pruning. Artificial Intelligence 6, 4 (1975), 293-326.

[9] A.L. Brudno, Bounds and Valuations for Abridging the Search of Estimates, Problems of Cybernetics 10, (1963), 225-241. Translation of Russian original in Problemy Kibernetiki 10, 141-150 (May 1963).

[10] D.E. Knuth and R.W. Moore, An Analysis of Alpha-beta Pruning, Artificial Intelligence 6, 4 (1975), 293-326.

[11] Marsland, T. A. (1986). A Review of Game-Tree Pruning. ICCA Journal. Vol. 9, No. 1, pp. 3-19.

[12] D. J. Slate and L. R. Atkin. CHESS 4.5 – The Northwestern University Chess Program. In P. Frey, editor, Chess Skill in Man and Machine, Pages 82-118. Springer-Verlag, 1977.

[13] Zobrist, A.L. (1970). A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA.

[14] Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1994). Replacement Schemes for Transposition Tables. ICCA Journal, Vol. 17, No. 4, pp. 183-193.

[15] T.A. Marsland and M. Campbell, Parallel Search of Strongly Ordered Game Trees, Computing Surveys 14, 4 (1982), 533-551.

[16] R.D. Greenblatt, D.E. Eastlake and S.D. Crocker, The Greenblatt Chess Program, Fall Joint Computing Conf. Procs. vol. 31, (San Francisco, 1967), 801-810. Also in D. Levy (ed.), Computer Chess Compendium, Springer-Verlag, 1988, 56-66.

[17] Akl, S.G. and Newborn, M.M. (1977). The principal Continuation and the Killer Heuristic. 1977 (ACM Annual Conference Proceedings, pp. 466-473. ACM, Seattle.

[18] Schaeffer, J. (1983). The History Heuristic. ICCA Journal, Vol. 6, No. 3, pp. 16-19.

[19] A. Reinefeld, An Improvement of the Scout Tree-Search Algorithm, Int. Computer Chess Assoc. J. 6, 4 (1983), 4-14.

[20] T.A. Marsland. Relative Performance of Alpha-Beta Implementations. In Proceedings of International Joint Conferences on Artificial Intelligence (IJCAI'83), pages 763-766, Karlsruhe, Germany, 1983.

[21] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-First Fixed-Depth Minimax Algorithms. Artificial Intelligence, 87:55-293, 1996.

[22] M. Buro. ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm, ICCA Journal 18(2), 71-76.

[23] M. Buro. Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. Technical Report No. 96, NEC Research Institute, 1997.

[24] Hashimoto, T., Kajihara, Y., Sasaki, N., Iida, H. and Jin, Y. (1999). An Evaluation Function for AMAZON, Shizuoka University, Japan.

[25] Kotani, Y. (1999). An Evaluation Function for Amazons. Proceedings of the 40th Programming Symposium, Information Processing Society of Japan, Tokyo, Japan.

[26] T. Hashimoto, Y. Kajihara, N. Sasaki, H. lida, J. Yoshimura. An evaluation function for amazons, in: H.J. van den Herds, B. Monien (Eds.), Advances in Computer Games, Vol. 9. Universiteii Maastricht, Maastricht, 2001,pp. 191-203.

[27] Kai-Fu Lee and Sanjoy Mahajan. *A pattern classification approach to evaluation function learning*. Artificial Intelligence Journal, 36:1--25, 1988.

[28] A. L. Samuel. Some studies in machine learning using the game of checkers, II. IBM J. 11 (1967) 601-617.

[29] D. Heckerman. A Bayesian approach for learning causal networks. In Proceedings of Eleventh Conference on Uncertainty in Artificial Intelligence, Montreal, QU, pages 285-295. Morgan Kaufmann.

[30] D. Kopec and I. Bratko, The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess, in Advances in Computer Chess 3, M. Clarke (ed.), Pergamon Press, Oxford, 1982, 57-72.

[31] J. Gillogly, Performance Analysis of the Technology Chess Program, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, 1978.

[32] R.M. Hyatt, Cray Blitz – A Computer Chess Playing Program, M.Sc. thesis, University of Southern Mississippi, 1983.

[33] P.P.L.M. Hensgens, A knowledge-based Approach of the Game of Amazons, MS thesis, Department of Computer Science, Universiteit Maastricht, 2001.

[34] M. Buro. *From simple features to sophisticated evaluation functions*. In H.J. van den Herik and H. Iida, editors, Computers and Games, Proceedings of CG98, LNCS 1558, pages 126--145. Springer Verlag, 1999 .