Minimax searches and Alpha-beta pruning for game playing

Sai Srivatsa R | 12EE10059

Indian Institute of Technology

Kharagpur

Abstract

We develop a Minimax and alpha-beta search based intelligent agent for game playing. We implement the above with GUI using Qt

1 Introduction

The Minimax algorithm is commonly used to determine an optimal move in two player zero sum games. The game play can be represented in the form of a minimax tree where the nodes represent moves. The nodes are of two types. That nodes denoting your moves are called *max* nodes and the nodes denoting the opponents move are called a *min* nodes. The leaf nodes of the tree assumes the value +1, 0 or –1 denoting Win, Tie or Lose respectively. A *max* node chooses a child with maximum value and a *min* node chooses a child with minimum value. The time and space complexities of the Minimax Algorithm are $O(b^d)$ and $O(bd)$ respectively where b is the average branching factor and d is the the depth of the tree.

Exponential time complexity makes it impossible to complete the full search and hence after reaching a depth d' ($d' << d$) , we use a heuristic function to return a value based on the status of the game given at that node.

Alpha-beta pruning is an algorithm to reduce the searching during minimax by pruning certain branches. This leads to reduced time complexity and hence the depth limiting the search can be increased. *Alpha* values are related to *max* nodes and the value is non-decreasing. *Beta* values are related to *min* nodes and the value is non-increasing. The search is stopped when *beta* value of a node is less than or equal to alpha.

The report is organized as follows. In section 2, we discuss the implementation. Evaluation and Performance are discussed in section 3, and Conclusion in section 4.

2 Methodology

2.1 Algorithm Implementation

We implement Minimax and Alpha Beta pruning algorithms for the Warfare game with GUI for game-playing using Qt-C++. We first discuss the implementation of Minimax and Alpha-Beta pruning algorithms. Given the current state $Game$, we compute the optimal move $OptMove$ by calling the minimax function for the maximizing player by calling $minimax(game, 0, True)$. Alpha Beta pruning expands a lesser number of nodes when compared to minimax algorithm and we can compute the optimal move by calling $ABPruning(game, 0, True, -\infty, +\infty)$.

The utility function returns a value depending on the current state of the game. We define the utility function for the maximizing player $Player$ as follows :

$$UTILITY(game) = A \times Score(game, player) - B \times Score(game, \overline{player})$$

where A and B are constants. When $|A| > |B|$, the Intelligent agent tries to maximize the score by occupying squares with larger values. When $|A| < |B|$, the agent tries to minimize the score by trying to Blitz the opponent thereby reducing the opponents score However if the values present have a large difference (like board $Narvik$), the larger square is occupied in this case. We set $|A| = |B|$ to be equal in order to given equal preference to both. However A and B can be varied for different board settings for better performance.

2.2 GUI Implementation

We use Qt Widgets to implement several features. We create the board using Push Buttons and Labels. We allow the user to choose among several boards as given in the problem statement. We also give the user a choice to select a random board. Each player can be played by a human himself or can be played by an intelligent agent using minimax or alpha beta pruning algorithm. This gives us 9 modes of playing including totally automated mode (Agent vs Agent) or semi-automated mode(Human vs Agent) or the manual mode (Human vs Human). We use QcomboBox widget for this purpose. We also display the instantaneous score using LCD displays. A label also

Algorithm 1 Minimax Algorithm

```
1: function MINIMAX(game,depth,Player)
2:     if Depth Limit Reached OR No valid moves left then
3:         return UTILITY(game)
4:     if Player then
5:         Best ← −∞
6:         for each valid move M in game do
7:             MAKE-MOVE(M)
8:             val ← minimax(game, depth + 1, Player)
9:             Best ← max(Best, val)
10:            if Depth = 0 AND val = Best then
11:                OptMove ← M
12:            UNDO − MOVE(M)
13:        return Best
14:    else
15:        Best ← +∞
16:        for each valid move M in game do
17:            MAKE-MOVE(M)
18:            val ← minimax(game, depth + 1, Player)
19:            Best ← min(Best, val)
20:            UNDO–MOVE(M)
21:        return Best
```

Algorithm 2 Alpha Beta Pruning Algorithm

```
1: function ABPruning(game,depth,maxPlayer,alpha,beta)
2:     if Depth Limit Reached OR No valid moves left then
3:         return UTILITY(game);
4:     if Player then
5:         for each valid move M in game do
6:             MAKE-MOVE(M)
7:             val ← ABPruning(game, depth + 1, \overline{Player}, alpha, beta)
8:             if val > alpha then
9:                 alpha ← val
10:                if depth = 0 then
11:                    optMove ← M
12:            UNDO − MOVE(M)
13:            if beta ≤ alpha then
14:                return alpha
15:        return beta
16:    else
17:        for each valid move M in game do
18:            MAKE-MOVE(M)
19:            val ← ABPruning(game, depth + 1, \overline{Player}, alpha, beta)
20:            if val < beta then
21:                beta ← val
22:                if depth = 0 then
23:                    optMove ← M
24:            UNDO − MOVE(M)
25:            if beta ≤ alpha then
26:                return beta
27:        return beta
```

shows the status of the game and it keeps updating itself. We also add a Start/Reset button and Quit button. There is also a difficulty slider. Setting higher levels of difficulty would increase the depth limit however this would lead to increase in time taken per move. Additionally, after the completion of every game, a new text file is automatically created and relevant game information such as Game Information, Move Sequences,Nodes expands and Time Elapsed are stored in it. Implementation of these features are trivial and we do not discuss it in detail.

3 Results and Performances

Alpha Beta pruning and Minimax perform equally well when the depth is same. Alpha Beta pruning expands minimum number of nodes and is faster than minimax algorithm. The performance of Alpha-Beta improves drastically as the board gets filled when compared to the Minimax algorithm. We set the default value of depth limit to 4 although until depth limit 5 is accepted for Minimax and 7 for Alpha-Beta pruning.

We extensively evaluated our approach for various setting of the board.

3.1 Comparison based on Game status (Win, Lose or Tie)

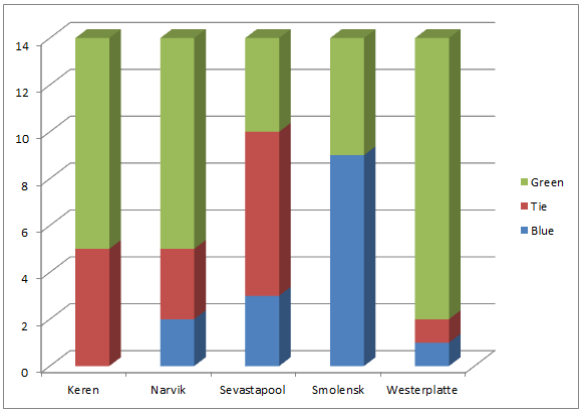
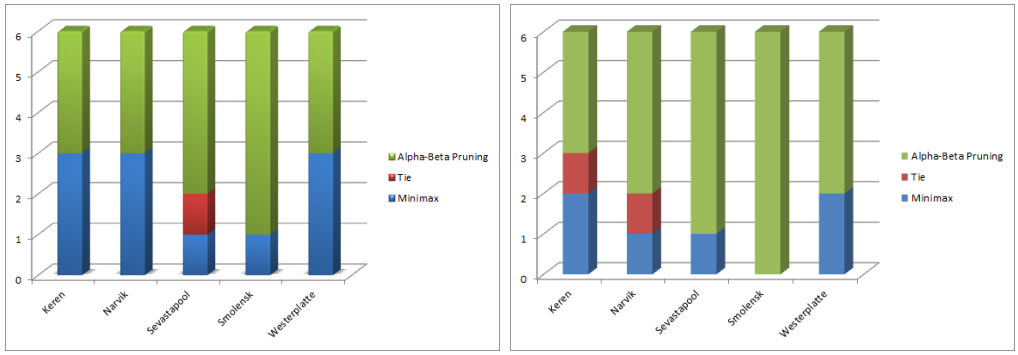


Figure 1: Graph showing # matches won by each player for each type of board

From figure 2, we can clearly conclude that alpha-beta pruning has a better performance when compared to minimax algorithm. We also compare how each agent performs against itself. Results are show in in figure 3. From these graphs, we can



(a) (b)

Figure 2: Graph showing # matches won by each player by different agents for each type of board. (a) has both the agents limited by the same depth whereas in (b) Alpha-beta pruning is limited by a higher depth than agent using minimax

conclude that the performance will improve if the depth limit is increased. This is also in accordance with the theory

We evaluate the approach on boards generated randomly at two settings. The first is the fast mode (default setting) which is limited by a smaller depth. The optimal move is decided quickly however the choice of the move may not be good enough. In the second mode, the depth limit is higher and the choice of move is better, however it is slower than the default mode.

3.2 Number of Nodes Expanded

The Number of nodes expanded by minimax algorithm is much higher than alpha-beta pruning algorithm. The number of nodes expanded falls as the game proceeds for both the algorithms. This is exemplified by the graphs showing in figure 5. In certain cases, the number of nodes expanded by alpha beta pruning increase after initial moves. This peculiar behavior is due to Blitzing.

3.3 Time Elapsed per Move

The time elapsed is proportional to the number of nodes expanded. The Time Elapsed vs the move number is shown in Figure 6 for limiting depths 5 and 6. Time taken is negligible when the limiting depth is 4.

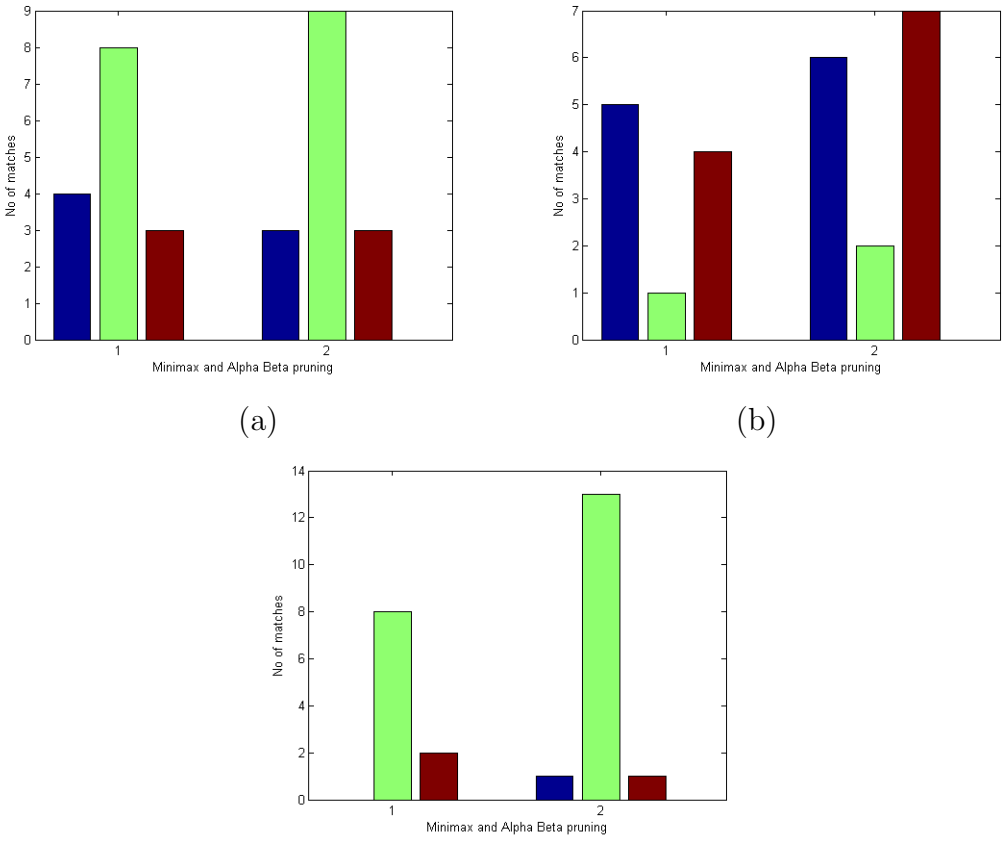
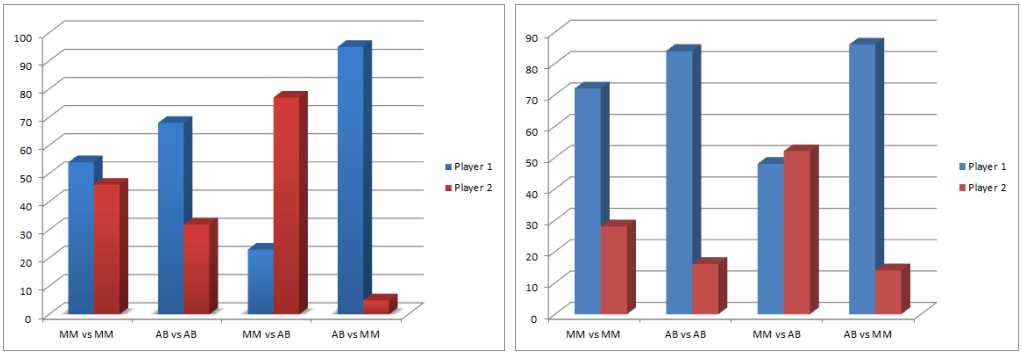
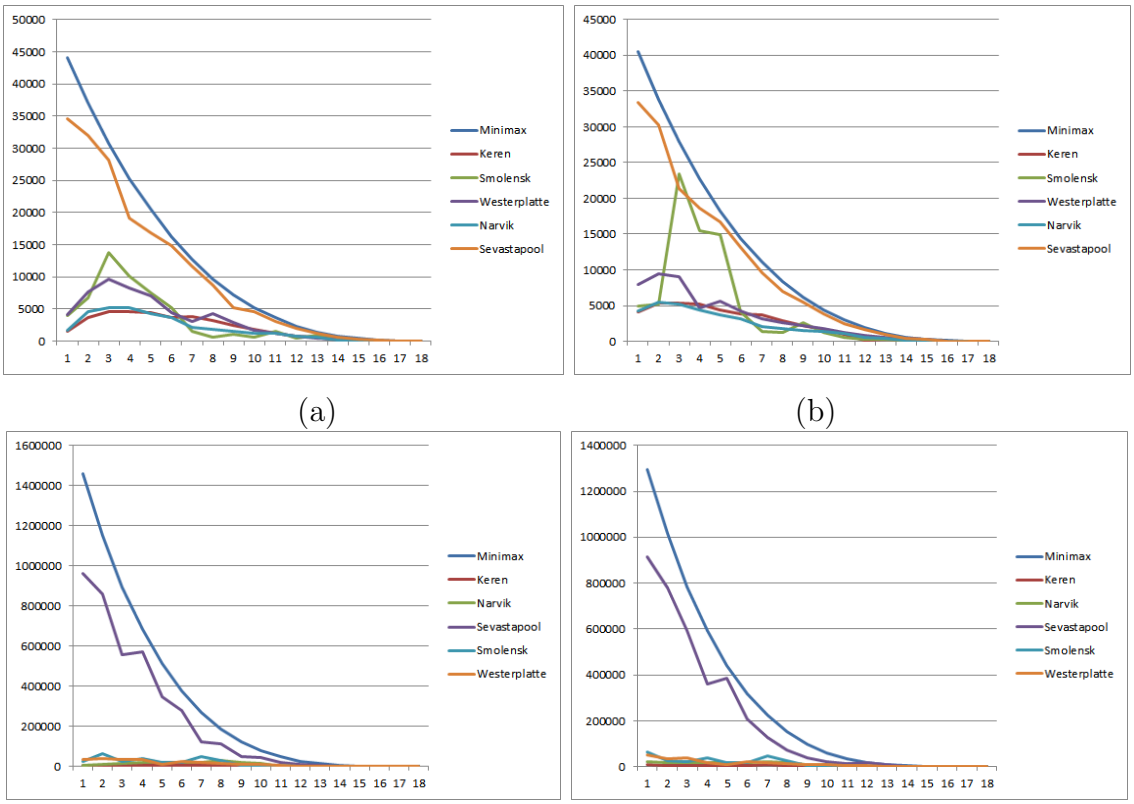


Figure 3: Graph showing # matches won by each agent when its competing against an agent using the same algorithm. The left group shows the performance of minimax against itself and the right group shows the performance of Alpha-beta pruning against itself (a) Both the agents operated at same depth (b) Player 1 limited by a higher depth (c) Player 2 limited by a higher depth



(a) (b)

Figure 4: Graph showing # of matches won for various settings on random boards in (a) default mode (quicker and easier) (b) Difficult mode (slower and tougher)



(a) (b)

(c) (d)

Figure 5: Graph showing # of nodes expanded by each agent at depth 4 as player 1 (a), depth 4 as player 2 (b), depth 5 as player 1 (c), depth 5 as player 2 and (d) at their respective move numbers

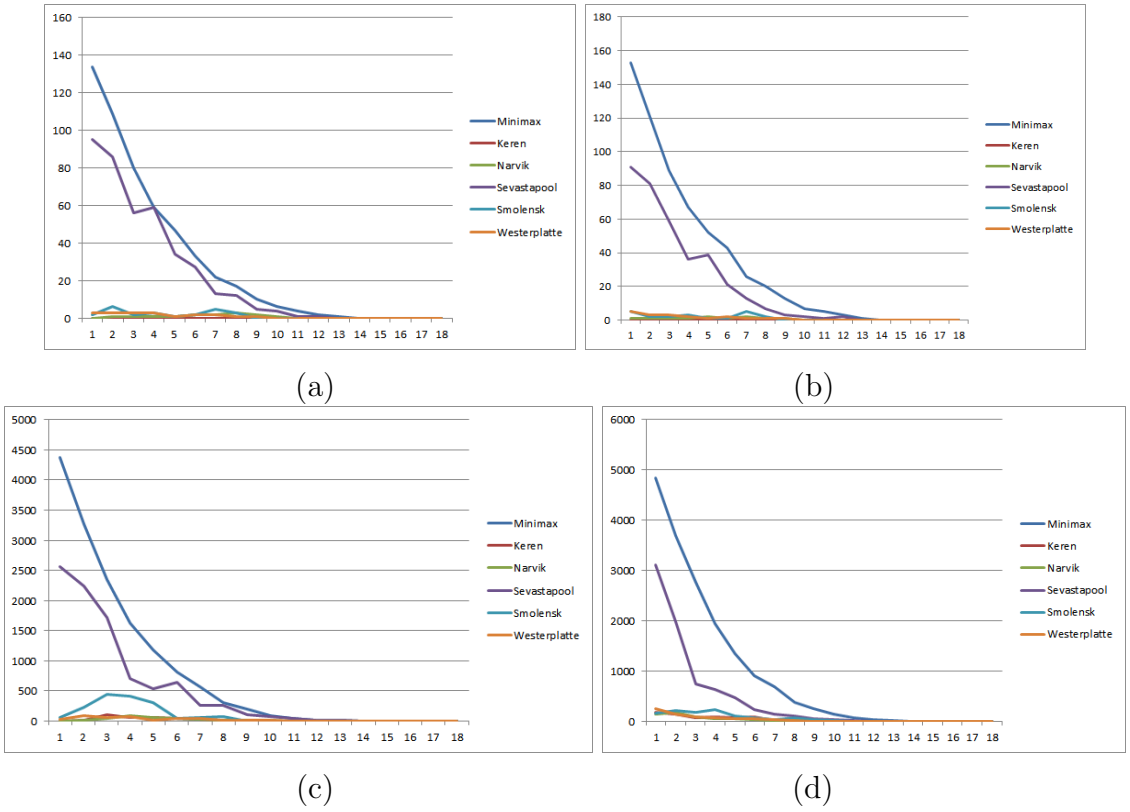


Figure 6: Graph showing time taken by each agent at depth 5 as player 1 (a), depth 5 as player 2 (b), depth 6 as player 1 (c), depth 6 as player 2, (d) at their respective move numbers

4 Conclusion

We have implemented an intelligent agent for playing the warfare game with GUI using Qt. Minimax trees can't be operated at full depth due to computational and space constraints and hence we limit the depth and use a heuristic to evaluate the status of the game. Increase in depth leads to better performance. We use alpha-beta pruning, which can operate at higher depth as branches are pruned saving time and space.

We face a trade-off between time per move and better choice of move that can make the game more difficult. This can also be overcome with a better choice of the heuristic function.