Adversarial Search: War Game

Minimax and Alpha-Beta Pruning

David Young

October 2015

Contents

1 Introduction

1.1 Purpose

This writeup summarizes the implementation and results of an adversarial search algorithm. It also contains discussion of said results and attempts to provide some insight and reflection on the behavior of implemented algorithms. This report was produced for course "CS-440: Artificial Intelligence" at University of Illinois Urbana Champaign.

1.2 Problem Definition (War Game)

The goal of this project was to implement an agent to play a simple "warfare" game.

Rules of the Game
The rules of the game are defined as follows:

- The game board is a 6x6 grid representing a city.
- Each square has a fixed point value between 1 and 99.
- There are two players, "blue" and "green". Each player takes turns: blue moves first, then green, then blue, etc.
- Object of the game is to be the player in the end with the largest total value of squares in their possession. Ie: one wants to capture the squares worth the most points.
- Game ends when all the squares are occupied by players since no more moves are left.
- Movement is always vertical and horizontal but never diagonal.
- Pieces can be conquered in the vertical and horizontal direction, but never diagonal.
- Values of the squares are defined at the beginning of each game, and remain constant.
- In each turn, a player can make one of two moves:

Move Option #1: Commando Paradrop. You can take any open space on the board with a Para Drop. This will create a new piece on the board. This move can be made as many times as one wants to during the game, but only once per turn. A Commando Para Drop cannot conquer any pieces. It simply allows one to arbitrarily place a piece on any unoccupied square on the board. Once you have done a Para Drop, your turn is complete.

The image below illustrates a Commando Para Drop. In this case, green drops a new piece on square [C,3]. This square is worth 39, which is a higher number, meaning that it contains some juicy oil wells or other important resources. After that, the score is green 39 : blue 3. A Commando Para Drop could have been carried out on any squares except for [D,4] since blue already occupies it.

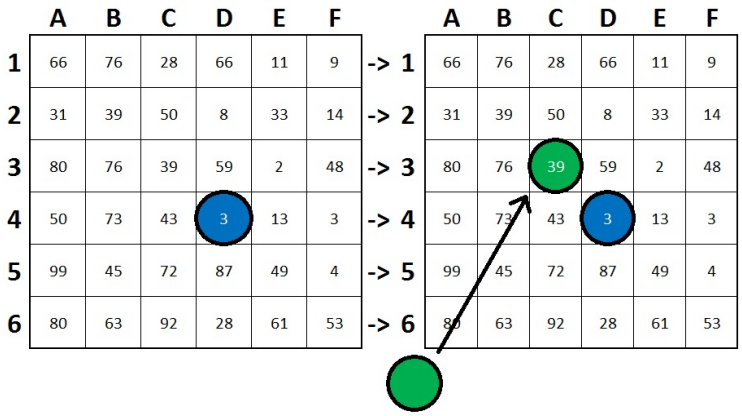

Figure 1: Example Commando Para Drop Move.

Move Option #2: M1 Death Blitz. From any space you occupy on the board, you can take the one next to it (up, down, left, right, but not diagonally) if it is unoccupied. The space you originally held is still occupied. Thus, you get to create a new piece in the blitzed square. Any enemy touching the square you have taken is conquered and that square is turned to your side (you turn its piece to your side). An M1 Death Blitz can be done even if it will not conquer another piece. Once you have made this move, your turn is over.

The image below illustrates an M1 Death Blitz. Green blitzes the piece in [D,4] to [D,3]. This conquers the blue piece in [D,2] since it is touching the new green piece in [D,3]. A blitz always creates a new piece and always moves one square, but it does not conquer another piece unless it is touching it. Thus, another valid move might have been for [D,4] to have blitzed [E,4]. Then the green player would own [D,4] and [E,4] but would have conquered none of blue's pieces. Note, the score before the blitz was green 46 : blue 157 but afterwards is green 113 : blue 149.

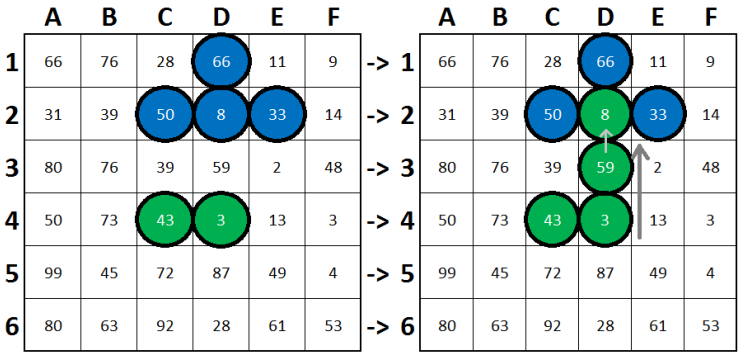

Figure 2: Example M1 Death Blitz Move.

Here is another illustration:

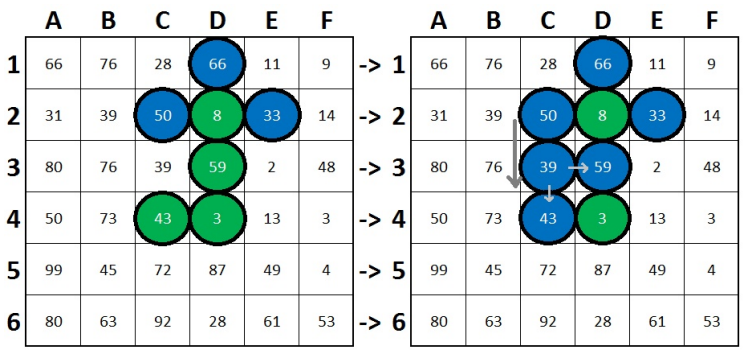

Figure 3: Another Example M1 Death Blitz Move.

Here blue blitzes [C,3] from [C,2]. In the process green's pieces at [D,3] and [C,4] are conquered since they touch [C,3]. Notice that in its next move, green will not be able to conquer any of blue's pieces and only the piece at [D,4] would be able to execute an M1 Death Blitz since [D,2] has no neighboring unoccupied squares.

You can assume that blitzing is mandatory: i.e., if you put a piece adjacent to pieces you already own and you have the opportunity to "convert" neighboring enemy pieces, you have to take it.

Game Board Supplied Format

Game boards are defined in basic text files. The text file include 36 integer values, one for each grid space of the final 6x6 game board. Each number in the .txt file represents the value of the corresponding grid space. An example game board .txt file might look like this:

```
66  76  28  66  11  9
31  39  50  8   33  14
80  76  39  59  2   48
50  73  43  3   13  3
99  45  72  87  49  4
80  63  92  28  61  53
```

Figure 4: Example Gameboard .txt file. Each integer represent the value of a grid space.

1.3 Project Overview

The basic task was to implement agents to play the above game, one using minimax search and one using alpha-beta search. The program used depth-limited search with an evaluation function. The maximum depth feasible for searching was determined for both agents. (For reference, the worst-case number of leaf nodes for a tree with a depth of three in this game is roughly 42,840).

Five different gameboards were supplied for testing. The gameboards are labeled as follows:

1. Keren.txt
2. Narvik.txt
3. Sevastopol.txt
4. Smolenk.txt
5. Westerplatte.txt

For each gameboard tested, the following match-ups were run:

(a) Minimax vs. minimax

(b) Minimax vs. alpha-beta (minimax goes first)

(c) Alpha-Beta vs. minimax (alpha-beta goes first)

(d) Alpha-Beta vs. alpha-beta

For each of the four match-ups, the following statistics were calculated:

- Final state of the board and the total scores for each player.
- Total number of game tree nodes expanded by each player during the game.
- Average number of nodes expanded per move & average amount of time per move.

2 Background

2.1 Adversarial Search & Search Agents

In multi-agent environments, each agent must consider the actions of other agents when planning their own actions. In a competitive multi-agent environment, where all agents are out to maximize their own welfare, agent's goals are often in conflict. Such environments are often abstracted into adversarial search problems. The most basic form of adversarial search problems are also quite familiar to us. Games!

Here, only games of a specific kind will be considered. That is, games that are deterministic, turn-taking, two-player, and present perfect information. Ie: agents act alternately with perfect information about the current state of the environment (fully observable). The agents compete to end the game with the highest utility; but maximizing utility for one agent minimizes utility for another. A problem is definably adversarial specifically because of this quality of opposition between the end utilities.

2.2 Minimax

The minimax algorithm is a way of finding an optimal move in a two player game. The big idea is to evaluate the utility of the leaf nodes such that each node in the tree can be assigned a worth value to the maximizing agent (AI player).

Each layer of the tree alternates as a MAX or MIN node. The goal at a MAX node is to maximize the value of the subtree rooted at that node. To do this, a MAX node chooses the child with the greatest value, and that becomes the value of the MAX node. Similarly, a MIN node tries to minimize the value of the subtree rooted at that node, and will choose the child with the lowest value, and that becomes the value of the MIN node. The MIN/MAX nodes are analogous to our war game where the MAX player is the one who's move it is, and then MIN player is the other player; who will choose the node to minimize the MAX players utility.

The minimax algorithm takes a root node and moves down the tree until it reaches leaf nodes (or the maximum permitted depth, for performance reasons). When it reaches such a stopping condition, it returns a heuristic value that measures or estimates the utility of the root node to the max player.

The basic pseudo-code for minimax search is as follows.

Minimax(*node*) =

- Utility(*node*) if *node* is terminal
- $\max_{action}$ **Minimax**(Succ(*node*, *action*)) if *player* = MAX
- $\min_{action}$ **Minimax**(Succ(*node*, *action*)) if *player* = MIN

Figure 5: Pseudo code for basic minimax search.

2.3 Alpha Beta Pruning

Alpha-beta pruning is a modification to the minimax search. It still finds the optimal minimax solution, but does so while avoiding unnecessary searching. It accomplishes the same task as normal minimax, but reduces the size of the search space by pruning the tree.

Alpha-beta pruning uses two bounds that are passed around in the algorithm. The bounds restrict the set of possible solutions based on the portion of the search tree that has already been seen.

- $\alpha = maximumlowerboundofpossiblesolutions$
- $\beta = minimumupperboundofpossiblesolutions$

Therefore, for any node state to be considered as part of the path to a solution, the current estimate value for that node must fall inside the range bounded by alpha and beta. ie: $\alpha <= estimatevalue <= \beta$

As the algorithm runs, restrictions on the range of possible solutions are updated based on min nodes (which may place an upper bound) and max nodes (which may place a lower bound). Moving through the tree, these bounds typically get closer together and eventually cross, such that beta ¡ alpha. If such a crossing occurs, the range for the node's value becomes nonexistent because there is no overlapping region between alpha and beta. In this circumstance, the node could never belong in a solution path, so the algorithm stops processing the node meaning it stops generating its children and moves back to the parent node. The algorithm must still note the value of this node however, so it passes (to the parent) the value that was changed which caused the crossing of alpha and beta.

The basic pseudo-code for minimax with alpha beta pruning is as follows.

```
Function action = Alpha-Beta-Search(node)
    v = Min-Value(node, −∞, ∞)
    return the action from node with value v

α: best alternative available to the Max player
β: best alternative available to the Min player

Function v = Min-Value(node, α, β)
    if Terminal(node) return Utility(node)
    v = +∞
    for each action from node
        v = Min(v, Max-Value(Succ(node, action), α, β))
        if v ≤ α return v
        β = Min(β, v)
    end for
    return v
```

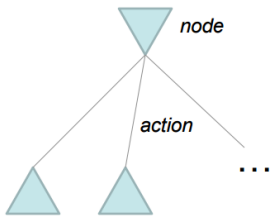

Figure 6: Pseudo code for minimax search with alpha beta pruning.

3 Overview of Source

Obtaining the source code
The entirety of the code written for this project can be found at the following repository:

https://github.com/dcyoung/WarGame

Summary of source code
The following source files were written from scratch. All code is well commented with Javadocs; it should be no burden to browse for specific details.

Filename	Description
AdversarialSearch.java	Conducts minimax or alpha-beta on a game state.
BoardState.java	Holds the state of a game board.
CommandoParaDrop.java	Defines a Commando Para Drop move.
DrawingBoard.java	Draw the state of the war game dynamically.
GameBoardFileReader.java	Reads a file containing initial game board values.
GameStateNode.java	Holds the state of the game at a given point.
GridSpace.java	Holds info about a grid space on the game board.
M1DeathBlitz.java	Defines an M1 Death Blitz Move.
Move.java	Abstract move class.
Player.java	Holds info about a player.
TestRunner.java	Contains basic tests runnable from a static main.

A quick summary of the code:

4 Implementation (Non-Algorithms)

4.1 Reading Info

Reading information is done through GameBoardFileReader class, which takes as input the filename of the game board to be used (in its constructor). This implementation used the java util Scanner class to fill out the grid values.

The first task is to open the file, this is done in initialization:

```
1
  public GameBoardFileReader(File file){
3   try{
      this.sc = new Scanner(file);
5     this.gridVals = this.readGridVals();
      this.numGridRows = this.gridVals.size();
7
      if( this.numGridRows !=0 )
9       this.numGridCols = this.gridVals.get(0).size();
      else
11      this.numGridCols = 0;
    }
13  catch (FileNotFoundException e) {
      System.out.println("File could not be found.");
15    e.printStackTrace();
    }
17 }
```

:

Most of GameBoardFileReader's functionality comes from readGridVals helper function, which populates a 2D ArrayList with the integer values of the squares in the game board file. This code is here:

```
1
  private ArrayList<ArrayList<Integer>> readGridVals(){
3   ArrayList<ArrayList<Integer>> tempGridVals =  new ArrayList<ArrayList<Integer
    >>();
    int lineCount = 0;
5   String tempLine;
    Scanner lineScan;
7
    while(sc.hasNextLine()){
9     tempLine = sc.nextLine();
      lineScan = new Scanner(tempLine);
11    tempGridVals.add(new ArrayList<Integer>());
      while(lineScan.hasNextInt()){
13      tempGridVals.get(lineCount).add(lineScan.nextInt());
      }
15    lineCount++;
    }
17  return tempGridVals;
  }
```

:

Once this arraylist is populated, the board is built. Specifically, the BoardState class' initializeGrid function handle's the translation of this arraylist to game state. This is described in the next section.

4.2 Representing Game State

The game's state is represented using the GameStateNode class. A single instance of game state contains two Players and a Board State. Players are implemented with the Player class, and the Board State is represented with the BoardState class.

A Player consists of a player id, the player's current score, and whether or not that player is a min or max player.

A BoardState consists of a grid of BoardSpaces; a BoardSpace is a single square in the game. The BoardSpace class keeps track of it's index in the BoardState grid, neighboring spaces, and whether or not it is occupied, and by which player.

Initially building the board state from the read 2D ArrayList of integers is done using this function:

```
1  public void initializeGrid(ArrayList<ArrayList<Integer>> initialGridVals){
2
3    //initialize the grid of spaces with the initial grid values
4    for(int row = 0; row < this.numGridRows; row++ ){
5      this.grid.add(new ArrayList<GridSpace>());
6      for(int col = 0; col < this.numGridCols; col++ ){
7        int val = initialGridVals.get(row).get(col);
8        this.grid.get(row).add(new GridSpace(row, col, val));
9      }
10   }
11   this.postInitializeGrid();
12 }
```

:

Establishing the neighbors for all of the GridSpaces is done using a postInitialize function:

```
1  public void postInitializeGrid(){
2    //for each grid space, populate it's neighboring spaces array
3    for(ArrayList<GridSpace> row : this.grid){
4      for(GridSpace gs : row){
5        gs.setNeighboringGridSpaces(this.determineGridSpaceNeighbors(gs));
6      }
7    }
8  }
9
10 public ArrayList<GridSpace> determineGridSpaceNeighbors(GridSpace gridSpace){
11   ArrayList<GridSpace> neighbors = new ArrayList<GridSpace>();
12   //add left, top, right, bottom
13   if( gridSpace.getCol() != 0 )
14     neighbors.add(this.grid.get(gridSpace.row).get(gridSpace.col-1));
15   if( gridSpace.getCol() != this.numGridCols-1 )
16     neighbors.add(this.grid.get(gridSpace.row).get(gridSpace.col+1));
17   if( gridSpace.getRow() != 0 )
18     neighbors.add(this.grid.get(gridSpace.row-1).get(gridSpace.col));
19   if( gridSpace.getRow() != this.numGridRows-1 )
20     neighbors.add(this.grid.get(gridSpace.row+1).get(gridSpace.col));
21
22   return neighbors;
23 }
```

:

4.3 Playing a Game

The war game is played in the TestRunner class by pitting two search algorithms against each other, using two Players. Specifically, a single GameState is shared between both Players, and both Players take alternating moves using their respective search algorithm.

This is all done in a while loop that ends when the shared state hits a leaf node (IE the board is filled). By keeping a count variable, the runner gives Player1 a move on even counts, Player2 on odd counts.

The function that gives each player a chance to run their search and select a move is called getPostSearchedMoveState. This function returns a new state that reflects the player's move.

```
  private GameStateNode getPostSearchedMoveState(GameStateNode state, boolean
    player1Move, boolean useAlphaBeta,int mmDepth, int abDepth){
    Player maximizingPlayer;
    if(player1Move){
      maximizingPlayer = state.getPlayer1();
      state.getPlayer1().setMaximizingPlayer(true);
      state.getPlayer2().setMaximizingPlayer(false);
    }
    else{
      maximizingPlayer = state.getPlayer2();
      state.getPlayer1().setMaximizingPlayer(false);
      state.getPlayer2().setMaximizingPlayer(true);
    }

    ArrayList<GameStateNode> children = state.getNodeChildren(maximizingPlayer);
    GameStateNode bestChoice = null;
    int bestValSoFar = Integer.MIN_VALUE;
    int tempVal;

    int moveExpandedNodes = 0;
    for( GameStateNode child : children ){
      AdversarialSearch mmSearch = new AdversarialSearch(child, mmDepth, abDepth,
    useAlphaBeta);
      tempVal = mmSearch.conductSearch();
      moveExpandedNodes += mmSearch.getNumExpandedNodes();
      if(tempVal > bestValSoFar){
        bestChoice = child;
        bestValSoFar = tempVal;
      }
    }

    if(player1Move){
      this.expandedNodesWriterP1.println(moveExpandedNodes);
    }
    else{
      this.expandedNodesWriterP2.println(moveExpandedNodes);
    }
    return bestChoice;
  }
```

:

getPostSearchedMoveState runs a player's search algorithm for all possible child states of a node (states where the player has made a legal move), and returns the node of maximum utility for the player.

Getting all possible child states of a node involves parsing out the illegal moves. For convenience, moves are represented by the abstract Move class. The core of this functionality is

accomplished by the getAllowableMoves function. Building out the child states, then, just involves crating gamestates using the Move declaration. The getAllowableMoves function is here:

```
public ArrayList<Move> getAllowableMoves(String playerID, GameStateNode state) {
  ArrayList<Move> allowableMoves = new ArrayList<Move>();

  //check every grid space to see if the player can move into it
  for(int row = 0; row < this.numGridRows; row ++){
    for(int col = 0; col < this.numGridCols; col ++){
      GridSpace gridSpace = this.grid.get(row).get(col);
      //if the grid space is unoccupied
      if(!gridSpace.isOccupied()){
        //add a new move of type CommandoParaDrop
        allowableMoves.add(new CommandoParaDrop(playerID, gridSpace, state));

        //Check if a blitz is allowed
        //if the grid space has a neighbor that belongs to the moving player
        for(GridSpace neighbor : gridSpace.getNeighboringGridSpaces()){
          if(neighbor.isOccupied()){
            if(neighbor.getResidentPlayerID().equals(playerID)){
              //add a new move of type M1DeathBlitz
              allowableMoves.add(new M1DeathBlitz(playerID, gridSpace, state));
              break;
            }
          }
        }
      }
    }
  }
  return allowableMoves;
}
```

:

NOTE: While the search algorithms themselves may reorder child states or even prune subtrees, this first depth level before the search algorithms are called contains every possible move. Therefore, a search algorithm of depth n will actually be conducting the search of depth n on every possible move from the current state. This is effectively a search depth of *n+1*. This will be irrelevant for the nodes expanded in minimax, but will yield seemingly high numbers of expanded nodes for alpha beta where the tree depth (in a functional interpretation) is not being pruned. In reality, the alpha beta search is pruning correctly, it's just being run on every child of the first depth. This implementation was chosen to allow for the search algorithms to return only integer values instead of states. This keeps the space requirements at a minimum, which is important in such a bloated data-structure with accessory functionality.

4.4 Conducting Searches

A search is represented by the AdversarialSearch class. This class contains all of the search algorithms, which will be explained in depth in the next section.

The type of adversarial search is determined by the useAlphaBeta flag, and the depth of the search is determined by mmDepth (for minimax searches) and abDepth (for Alpha/-Beta searches). See the previous bolded "NOTE", from sub-section 4.3, for clarity on the meaning of depth here.

A search is performed by calling the conductSearch function, which returns the utility of the move found by the search.

4.5 Displaying the Game Graphically

The game is displayed graphically using StdDraw, a graphical java library. The details of the drawing are in the DrawingBoard class. Drawing a gamestate involves setting the DrawingBoard's internal GameState using a setter method, and then calling the drawCurrentBoardState function. The TestRunner uses this interface to update the board after every move. The class also contains methods to save the draw images after updates, so that simple .gif animations of saved game play can be created.

5 Adversarial Search: Minimax

5.1 Solution Implementation

The Minimax Algorithm

```
public int minimax(GameStateNode root, int depthLimit, boolean bIsMaxNode){
    //if the node is a leaf node report its utility,
    if( root.isLeafNode() || depthLimit == 0 ){
      //treat deep enough nodes as leaf nodes (this will be a utility estimate)
      return evaluate(root, bIsMaxNode);
    }
    else{
      ArrayList<Move> allowableMoves;
      Move move;
      GameStateNode child;
      int childMiniMaxValue;

      if(bIsMaxNode){
        //n is a max node
        int miniMaxValue = Integer.MIN_VALUE;

        //get the allowable moves for the current state
        allowableMoves = root.getAllowableMoves(root.getMaximizingPlayer());
        //consider every child state resulting from one of the allowable moves
        for(int moveIndex = 0; moveIndex < allowableMoves.size(); moveIndex++){
          move = allowableMoves.get(moveIndex);
          child = root.getChildStateAfterMove(root.getMaximizingPlayer(), move);
          //evaluate the child
          this.numExpandedNodes++;
          childMiniMaxValue = minimax(child, depthLimit-1, false);
          //n is a max node, its minimax value will be the max of all its children
          miniMaxValue = Math.max(miniMaxValue, childMiniMaxValue);
        }
        return miniMaxValue;
      }
      else{
        //n is a min node
        int miniMaxValue = Integer.MAX_VALUE;

        //get the allowable moves for the current state
        allowableMoves = root.getAllowableMoves(root.getMinimizingPlayer());
        //consider every child state resulting from one of the allowable moves
        for(int moveIndex = 0; moveIndex < allowableMoves.size(); moveIndex++){
          move = allowableMoves.get(moveIndex);
          child = root.getChildStateAfterMove(root.getMinimizingPlayer(), move);
          //evaluate the child
          this.numExpandedNodes++;
          childMiniMaxValue = minimax(child, depthLimit-1, true);
          //n is a min node, its minimax value will be the min of all its children
          miniMaxValue = Math.min(miniMaxValue, childMiniMaxValue);
        }
        return miniMaxValue;
      }
    }
  }
```

:

6 Adversarial Search: Alpha Beta Pruning

6.1 Solution Implementation

```
1
public int alphaBeta(GameStateNode root, int depthLimit, int alpha, int beta,
    boolean bIsMaxNode){
3   if( root.isLeafNode() || depthLimit == 0 ){
      return evaluate(root, bIsMaxNode);
5   }
    else{
7     //get all the allowable moves for this state
      ArrayList<Move> allowableMoves;
9     GameStateNode childStateNode;

11    if(bIsMaxNode){
        //get the allowable moves for the current state
13      allowableMoves = root.getAllowableMoves(root.getMaximizingPlayer());
        //n is a max node
15      int miniMaxValue = alpha;
        int childValue;
17
        //consider every child state resulting from an allowable move
19      for(int moveIndex = 0; moveIndex < allowableMoves.size(); moveIndex++){
          childStateNode = root.getChildStateAfterMove(root.getMaximizingPlayer(),
    allowableMoves.get(moveIndex));
21        //evaluate the child state
          this.numExpandedNodes++;
23        childValue = alphaBeta(childStateNode, depthLimit-1, miniMaxValue, beta,
    false);
          //n is a max node, its minimax value will be the max of its children
25        miniMaxValue = Math.max(miniMaxValue, childValue);

27        //update alpha and check if alpha and beta crossed
          alpha = Math.max(alpha, miniMaxValue);
29        if(beta < alpha){
            //break;
31          return alpha;
          }
33      }
        return miniMaxValue;
35    }
      else{
37      allowableMoves = root.getAllowableMoves(root.getMinimizingPlayer());
        //n is a min node
39      int miniMaxValue = beta;
        int childValue;
41
        //consider every child state resulting from an allowable move
43      for(int moveIndex = 0; moveIndex < allowableMoves.size(); moveIndex++){
          childStateNode = root.getChildStateAfterMove(root.getMinimizingPlayer(),
    allowableMoves.get(moveIndex));
45        //evaluate the child state
          this.numExpandedNodes++;
47        childValue = alphaBeta(childStateNode, depthLimit-1, alpha, miniMaxValue,
    true);
          miniMaxValue = Math.min(miniMaxValue, childValue);
49        beta = Math.min(beta,  miniMaxValue);
          if(beta < alpha){
51          return beta;
            //break;
53        }
        }
55      return miniMaxValue;
      }
57  }
}
```

:

7 Results

7.1 Results Intro:

The results are a response to the project goals, restated here. Five different gameboards were supplied for testing (Keren, Narvik, Sevastopol, Smolenk & Westerplatte). Each gameboard was tested on four match configurations of minimax and alpha beta agents (MM vs. MM, MM vs. AB, AB vs. MM & AB vs. AB). For each of the four match-ups, the following statistics were calculated:

- Final state of the board and the total scores for each player.
- Total number of game tree nodes expanded by each player during the game.
- Average number of nodes expanded per move & average amount of time per move.

To automate these results, the Adversarial Search class kept a tally of expanded nodes that was incremented upon each recursive call to a search algorithm. This yielded results for expanded nodes at every move for both players. Similar statistics were automated for move times and all the results were compiled in spread sheets.

To see more statistical detail, the included excel spread sheets provide expansions and durations of individual moves.

NOTE: while the final scores of the included implementations are consistent, they will most likely vary from other implementations. While the nodes expanded might be similar for various implementations, the final scores of the players in a game will vary depending on how tie breakers and evaluation functions were implemented. If all moves equated to the same utility, different implementations might choose different moves, resulting in different states later on that affect the outcome of the game. Different evaluation functions could lead to different choices even without a tie breaker.

NOTE: as previously mentioned, the search algorithms are conducted on the resultant states of the first possible set of moves. This was a conscious design choice for space considerations given the functionally large data structures and simplicity of an integer return type with the search functions. This will yield slightly higher node expansions for the alpha beta pruning as the first depth layer will not be pruned, but the search algorithm is still 100% effective and optimal from a supplied root node. The root nodes used for searching just happen to be the game states that result from the first set of allowable moves in the game state currently occupied by the player.

7.2 Minimax Depth 3, Alpha Beta Depth 3

The results of both search algorithms searching to a depth of 3 are shown below.

		Player 1				Player 2				Combined			
Puzzle	Heuristic	Strategy	Score	A.M.T. (ms)	T.N.E	Strategy	Score	A.M.T. (ms)	T.N.E	A.M.T. (ms)	Avg Nodes Expanded/Move	Total Game Time (ms)	Winner
Keren	Score Diff	Minimax	12	38	314,403	Minimax	24	31	280,841	35	16,535	1,251	2
Keren	Score Diff	Minimax	12	31	314,403	Alpha Beta	24	14	137,631	22	12,557	805	2
Keren	Score Diff	Alpha Beta	12	12	121,544	Minimax	24	26	280,841	19	11,177	687	2
Keren	Score Diff	Alpha Beta	12	12	121,544	Alpha Beta	24	13	137,631	12	7,199	438	2
Narvik	Score Diff	Minimax	803	40	335,575	Minimax	997	34	292,719	37	17,453	1,341	2
Narvik	Score Diff	Minimax	803	35	335,575	Alpha Beta	997	13	122,213	24	12,716	872	2
Narvik	Score Diff	Alpha Beta	803	11	104,074	Minimax	997	28	292,719	19	11,022	694	2
Narvik	Score Diff	Alpha Beta	803	10	5,782	Alpha Beta	997	12	122,213	11	3,555	402	2
Sevastopol	Score Diff	Minimax	261	37	328,888	Minimax	117	30	283,778	33	17,019	1,202	1
Sevastopol	Score Diff	Minimax	261	33	328,888	Alpha Beta	117	28	273,077	31	16,721	1,108	1
Sevastopol	Score Diff	Alpha Beta	261	30	320,545	Minimax	117	26	283,778	28	16,787	1,014	1
Sevastopol	Score Diff	Alpha Beta	261	29	320,545	Alpha Beta	117	25	273,077	27	16,490	972	1
Smolensk	Score Diff	Minimax	1,034	38	328,578	Minimax	619	34	291,697	36	17,230	1,279	1
Smolensk	Score Diff	Minimax	1,034	34	328,578	Alpha Beta	619	10	96,718	22	11,814	803	1
Smolensk	Score Diff	Alpha Beta	1,034	14	145,952	Minimax	619	27	291,697	20	12,157	728	1
Smolensk	Score Diff	Alpha Beta	1,034	14	145,952	Alpha Beta	619	9	96,718	12	6,741	419	1
Westerplatte	Score Diff	Minimax	36	45	397,527	Minimax	36	39	349,729	42	20,757	1,520	tie
Westerplatte	Score Diff	Minimax	36	41	397,527	Alpha Beta	36	17	166,037	29	15,655	1,047	tie
Westerplatte	Score Diff	Alpha Beta	36	19	198,114	Minimax	36	32	349,729	26	15,218	920	tie
Westerplatte	Score Diff	Alpha Beta	36	19	198,114	Alpha Beta	36	16	166,037	17	10,115	618	tie

A.M.T. = Average Move Time
T.N.E = Total Nodes Expanded

Figure 7: Adversarial search results for minimax and alpha beta agents both using depth 3.

The first important thing to note here is that the final scores for a puzzle remain the same regardless of the match up. This is expected, as both minimax and alpha beta should return the same optimal result provided the same depth of search. So even though the alpha beta pruning improved on the run time and node expansion, it cannot provide any improved move selection without searching deeper in the tree. It is not considering anything more than minimax, just searching the same space more efficiently.

The second noteworthy observation is better seen from the individual move statistics shown in the excel spread sheets. For a search depth of 3, here is an example number of expanded nodes per turn for all moves made by a minimax agent during the course of a game:

Move #	Expanded Nodes
0	48,300
1	48,810
2	45,534
3	42,134
4	31,750
5	25,371
6	18,604
7	14,026
8	12,160
9	8,462
10	7,293
11	5,184
12	2,996
13	1,945
14	1,166
15	520
16	144
17	4

Figure 8: Example Gameboard .txt file. Each integer represent the value of a grid space.

The worst case number of leaf nodes for a depth 3 search tree in War Game is 42,840. The total number expanded will of course be bigger than the number of leaf nodes, as the algorithm expands the depths approaching leaf depth. But what is clearly shown is a reasonable expanded node count for the first move (which will be near worst case) and then a decrease as the moves accumulate and occupied board spaces restrict potential moves (pruning the tree naturally).

7.3 Minimax Depth 4, Alpha Beta Depth 4

The results of both search algorithms searching to a depth of 4 are shown below. Trends here are similar to the previous results for depth 3. Both algorithms yield identical score results as they search the same space, and alpha beta pruning results in fewer expanded nodes and therefore reduced movement times (faster search).

		Player 1				Player 2				Combined			
Puzzle	Heuristic	Strategy	Score	A.M.T. (ms)	Total Nodes Expanded	Strategy	Score	A.M.T. (ms)	Total Nodes Expanded	A.M.T. (ms)	Avg Nodes Expanded/Move	Total Game Time (ms)	Winner
Keren	Score Diff	Minimax	19	1,009	10,662,815	Minimax	17	795	8,627,923	902	535,854	32,470	1
Keren	Score Diff	Minimax	19	974	10,662,815	Alpha Beta	17	595	6,408,344	785	474,199	28,242	1
Keren	Score Diff	Alpha Beta	19	881	9,391,908	Minimax	17	818	8,627,923	849	500,551	30,576	1
Keren	Score Diff	Alpha Beta	19	899	9,391,908	Alpha Beta	17	614	6,408,344	757	438,896	27,243	1
Narvik	Score Diff	Minimax	701	1,158	11,991,897	Minimax	1,099	895	9,545,837	1,026	598,270	36,949	2
Narvik	Score Diff	Minimax	701	1,125	11,991,897	Alpha Beta	1,099	574	6,019,674	849	500,321	30,567	2
Narvik	Score Diff	Alpha Beta	701	665	6,792,104	Minimax	1,099	930	9,545,837	798	453,832	28,715	2
Narvik	Score Diff	Alpha Beta	701	678	377,339	Alpha Beta	1,099	593	6,019,674	636	177,695	22,881	2
Sevastopol	Score Diff	Minimax	186	1,148	11,596,409	Minimax	192	891	9,218,376	1,019	578,188	36,697	2
Sevastopol	Score Diff	Minimax	186	1,077	11,596,409	Alpha Beta	192	843	9,015,069	960	572,541	34,565	2
Sevastopol	Score Diff	Alpha Beta	186	1,002	11,024,612	Minimax	192	839	9,218,376	921	562,305	33,151	2
Sevastopol	Score Diff	Alpha Beta	186	1,012	11,024,612	Alpha Beta	192	834	9,015,069	923	556,658	33,235	2
Smolensk	Score Diff	Minimax	682	1,043	10,933,937	Minimax	971	995	10,659,741	1,019	599,824	36,678	2
Smolensk	Score Diff	Minimax	682	1,007	10,933,937	Alpha Beta	971	660	7,066,413	834	500,010	30,010	2
Smolensk	Score Diff	Alpha Beta	682	674	7,270,011	Minimax	971	977	10,659,741	826	498,049	29,721	2
Smolensk	Score Diff	Alpha Beta	682	670	7,270,011	Alpha Beta	971	657	7,066,413	664	398,234	23,898	2
Westerplatte	Score Diff	Minimax	37	1,367	14,781,239	Minimax	35	1,046	11,432,518	1,207	728,160	43,438	1
Westerplatte	Score Diff	Minimax	37	1,338	14,781,239	Alpha Beta	35	708	7,718,717	1,023	624,999	36,833	1
Westerplatte	Score Diff	Alpha Beta	37	891	9,717,711	Minimax	35	1,044	11,432,518	968	587,506	34,836	1
Westerplatte	Score Diff	Alpha Beta	37	922	9,717,711	Alpha Beta	35	723	7,718,717	822	484,345	29,599	1

A.M.T. = Average Move Time

Figure 9: Adversarial search results for minimax and alpha beta agents both using depth 4.

7.4 Minimax Depth 4, Alpha Beta Depth 5

Note: animations were generated for every game in this result set. They are included in the form of .gif files. They can viewed easily in an internet browser such as Chrome.

The results of minimax searching to a depth of 4 and alpha beta searching to a depth of 5 are shown below. As expected, the resultant scores are not the same. Here, alpha beta is actually searching a much larger space (tree expands exponentially) than minimax by searching a level deeper. This resulted in slower average move times, despite a more efficient search. Ie: an efficient search algorithm searching an enormous space can still take longer than an inefficient search algorithm searching a small space, but the more efficient algorithm will have been exposed to more information. By searching an extra level deeper, the alpha beta pruning algorithm was able to look further ahead at the results of a move. In practice this didn't yield consistently better move decisions, as the evaluation function was too naive to reliably predict the final outcome from an intermediate state. Additionally the horizon effect could likely play a role in how accurate the predictions were, with a shallower search depth sometimes resulting in better choices in the long run. But this was not the result of information advantage, just luck and game board design.

		Player 1				Player 2				Combined			
Puzzle	Heuristic	Strategy	Score	A.M.T. (ms)	Total Nodes Expanded	Strategy	Score	A.M.T. (ms)	Total Nodes Expanded	A.M.T. (ms)	Avg Nodes Expanded/Move	Total Game Time (ms)	Winner
Keren	Score Diff	Minimax	19	994	10,662,815	Minimax	17	792	8,627,923	893	535,854	32,149	1
Keren	Score Diff	Minimax	14	956	10,419,170	Alpha Beta	22	11,000	116,931,091	5,978	3,537,507	215,199	2
Keren	Score Diff	Alpha Beta	18	9,183	94,079,872	Minimax	18	798	8,620,655	4,990	2,852,792	179,656	tie
Keren	Score Diff	Alpha Beta	14	9,719	100,464,697	Alpha Beta	22	10,998	116,434,838	10,358	6,024,987	372,900	2
Narvik	Score Diff	Minimax	701	1,116	11,991,897	Minimax	1,099	879	9,545,837	997	598,270	35,894	2
Narvik	Score Diff	Minimax	996	1,037	11,319,520	Alpha Beta	804	8,687	91,264,334	4,862	2,849,552	175,038	1
Narvik	Score Diff	Alpha Beta	701	8,020	80,907,049	Minimax	1,099	879	9,545,837	4,450	2,512,580	160,190	2
Narvik	Score Diff	Alpha Beta	700	7,542	4,239,086	Alpha Beta	1,100	8,673	91,182,040	8,108	2,650,587	291,876	2
Sevastopol	Score Diff	Minimax	186	1,057	11,596,409	Minimax	192	831	9,218,376	944	578,188	33,988	2
Sevastopol	Score Diff	Minimax	207	1,007	11,209,443	Alpha Beta	171	20,091	222,430,638	10,549	6,490,002	379,768	1
Sevastopol	Score Diff	Alpha Beta	363	26,624	292,130,483	Minimax	15	751	8,249,451	13,687	8,343,887	492,736	1
Sevastopol	Score Diff	Alpha Beta	253	27,568	298,628,458	Alpha Beta	125	20,488	222,757,348	24,028	14,482,939	865,002	1
Smolensk	Score Diff	Minimax	682	1,025	10,933,937	Minimax	971	986	10,659,741	1,005	599,824	36,185	2
Smolensk	Score Diff	Minimax	1,010	1,057	11,486,707	Alpha Beta	643	10,884	114,219,344	5,971	3,491,835	214,948	1
Smolensk	Score Diff	Alpha Beta	885	16,125	169,097,856	Minimax	768	995	10,722,582	8,560	4,995,012	308,150	1
Smolensk	Score Diff	Alpha Beta	727	11,301	116,614,651	Alpha Beta	926	10,753	112,587,196	11,027	6,366,718	396,967	2
Westerplatte	Score Diff	Minimax	37	1,358	14,781,239	Minimax	35	1,043	11,432,518	1,200	728,160	43,211	1
Westerplatte	Score Diff	Minimax	42	1,342	14,770,130	Alpha Beta	30	11,159	118,008,750	6,250	3,688,302	225,014	1
Westerplatte	Score Diff	Alpha Beta	37	14,814	152,286,754	Minimax	35	1,055	11,279,006	7,934	4,543,493	285,632	1
Westerplatte	Score Diff	Alpha Beta	25	12,758	131,476,235	Alpha Beta	47	11,351	119,589,791	12,055	6,974,056	433,971	2

A.M.T. = Average Move Time

Figure 10: Adversarial search results for minimax (depth 4) and alpha beta (depth 5) agents.

8 Analysis & Discussion

Notes

As mentioned previously, while the final scores of the included implementations are consistent, they will most likely vary from other implementations depending on how tie breakers and evaluation functions were implemented. Additionally, for reasons already stated, the search algorithms are conducted on the resultant states of the first possible set of moves. This will yield slightly higher node expansions for the alpha beta pruning as the first depth layer will not be pruned, but the search algorithm is still 100% effective and optimal from a supplied root node. The root nodes used for searching just happen to be the game states that result from the first set of allowable moves in the game state currently occupied by the player.

Evaluation Functions

The evaluation function has arguably the most impact on the algorithms performance as it must accurately estimate the utility of a state. If the state is a leaf node then the evaluation is very simple because the winner of the game is known by reading the scores off the board. For intermediate states however, the evaluation function can become very complicated. The more complex the game, the more complex the evaluation function generally.

For the listed results, the evaluation function was kept very simple are arguably extremely naive. Because grid spaces can be stolen in WarGame, it wasn't the best as predicting the leaf node utility. But it did provide a very easy way to evaluate the intermediate state in question.

```
public int evaluate(GameStateNode state, boolean bIsMaxNode){
  int heuristic;
  int maxPlyrScore = state.getMaximizingPlayer().getCurrentScore();
  int minPlyrScore = state.getMinimizingPlayer().getCurrentScore();
  int scoreDifference = maxPlyrScore - minPlyrScore;

  //default heuristic
  heuristic = scoreDifference;

  return heuristic;
}
```

:

A few modifications were tried in an attempt to incorporate the possibility of losing utility from another player stealing grid spaces.

```
public int evaluate(GameStateNode state, boolean bIsMaxNode){
  int heuristic;
  int maxPlyrScore = state.getMaximizingPlayer().getCurrentScore();
  int minPlyrScore = state.getMinimizingPlayer().getCurrentScore();
  int scoreDifference = maxPlyrScore - minPlyrScore;

    //Custom Heuristic:  score difference after subtracting the vulnerable points
    from active player's score
  int securedScore;
  if (bIsMaxNode) {
    //it is the max player's turn... best case is positive score difference and
    large portion secured
    int maxPlyrVulnPoints = state.getBoardState().getVulnerablePoints(state.
    getMaximizingPlayer().getPlayerID());
    securedScore = maxPlyrScore - maxPlyrVulnPoints;
    heuristic = securedScore - minPlyrScore;
  }
  else {
    //it is the min player's turn... best case is negative score difference and
    large portion secured
    int minPlyrVulnPoints = state.getBoardState().getVulnerablePoints(state.
    getMinimizingPlayer().getPlayerID());
    securedScore = minPlyrScore - minPlyrVulnPoints;
    heuristic = maxPlyrScore - securedScore;
  }
  return heuristic;
}
```

:

The evaluation function attempted to determine the number of points that were exposed to the opposing player's M1 Death Blitz with the following method from BoardState.

```
public int getVulnerablePoints(String playerID) {
  //determine which spaces are currently occupied by the specified player
  ArrayList<GridSpace> controlled = new ArrayList<GridSpace>();
  for(int row = 0; row < this.numGridRows; row ++){
    for(GridSpace gs : this.grid.get(row)){
      if(gs.isOccupied() && gs.getResidentPlayerID().equals(playerID)){
        controlled.add(gs);
      }
    }
  }

  //determine which of those occupied spaces could be stolen on the nex turn
  Set<GridSpace> vulnerableGridSpaces = new HashSet<GridSpace>();
  for(GridSpace controlledSpace : controlled){
    INNER:
    for(GridSpace potentialEmptyNeighbor : controlledSpace.getNeighboringGridSpaces
    ()){
      if(!potentialEmptyNeighbor.isOccupied()){
        for(GridSpace potentialOpponentSpace : potentialEmptyNeighbor.
    getNeighboringGridSpaces()){
          if(potentialOpponentSpace.isOccupied() && !potentialOpponentSpace.
    getResidentPlayerID().equals(playerID)){
            vulnerableGridSpaces.add(controlledSpace);
            //This space just has to be determined vulnerable by any 1 neighbor,
    other neighbors don't need to be considered
            break INNER;
          }
        }
      }
    }
  }

  //tally the potential loss
  int vulnerablePoints = 0;
```

```
    for(GridSpace gs : vulnerableGridSpaces){
32      vulnerablePoints += gs.getValue();
    }
34  return vulnerablePoints;
}
```

:

Unfortunately, the results did not show statistically significant improvement. Perhaps this was because the implementation above considered all exposed points, and not the max vulnerable to a single blitz.

Revisiting States
The search algorithms did not contain any intelligent check to prevent revisiting a previously visited state via a different path. For example, if two different series of moves resulted in identical game states, then that state was evaluated twice during the search. The search algorithms did not maintain any form of tree structure in order to save on space. The algorithm generated the search tree as it moved along. Despite the improvements to speed, space performance and unnecessary tree formation that would have been pruned, this did mean that repeated state detection was not implemented. It was a trade off, and data from both methods would be required to make a solid statement about the design choice.

Depth Confusion
It is worth noting again that the nodes expanded by the agent using alpha beta pruning was affected by the design choice to conduct the adversarial search on each child of the current state, rather than the current state itself. This means a search depth of 5 is really at most 36*(search depth of 4). Arguably this had a big impact on performance for the agent playing the game, but it did allow for fully functional minimax and alpha beta methods that were compartmentalized enough to not interfere with extra functionality in the data structure.

Horizon Effect
All things the same, alpha beta pruning does not inherently result in a better choice than minimax. In fact it cannot result in anything but the same choice as minimax. The benefit of alpha beta pruning is actually the ability to save time and space. By being more efficient, an agent can conduct a deeper search that may yield a better choice than a shallow search. Indeed the statistics about expanded nodes and time per move yielded a clear advantage to alpha beta, demonstrating it as the more efficient search. Yet, looking at the final scores in the results of minimax to depth 4 vs. alpha-beta to depth 5, alpha-beta did not win consistently. There is a reason for this. It is called the *horizon effect*.

The horizon effect describes the possibility that an agent will make a detrimental move, but the effect is not visible because the computer does not search to the depth of the error (i.e. beyond its "horizon"). Since search depth is often limited for feasibility reasons, evaluating a partial tree is common. This evaluation may give a misleading result and when a significant change exists just over the horizon of the search depth, the agent could have made a bad choice.

For War Game, the M1 Death Blitz presents enormous opportunity for drastic change and turn around between player scores. This is the exact type of situation that manifests as the horizon effect.

Potential Improvements
The first major improvement would be an intelligent evaluation function. The evaluation function is really what limits the end result of the search algorithm and regardless of performance, an algorithm is only as good as its result. Performance could be further improved by intelligently ordering the moves for the alpha beta agent and perhaps implement a new data structure that worked well with a search from the current state rather than a child. Lastly, the horizon effect could be mitigated by implementing quiescence search. A quiescent search would attempt to emulate the intuition of human players by abandoning bad-looking moves and searching deeper into promising moves to make sure there aren't black flags just over the horizon.

Conclusions

This project was fun to implement. The usefulness of the implemented algorithms is immediately obvious. Using these basic adversarial search algorithms, one could code a basic AI player for most simple, turn based, 2 player games. From the results it was clear that evaluation functions can make a big difference in the success of an AI player, while many optimizations exist to improve the speed of an AI player.