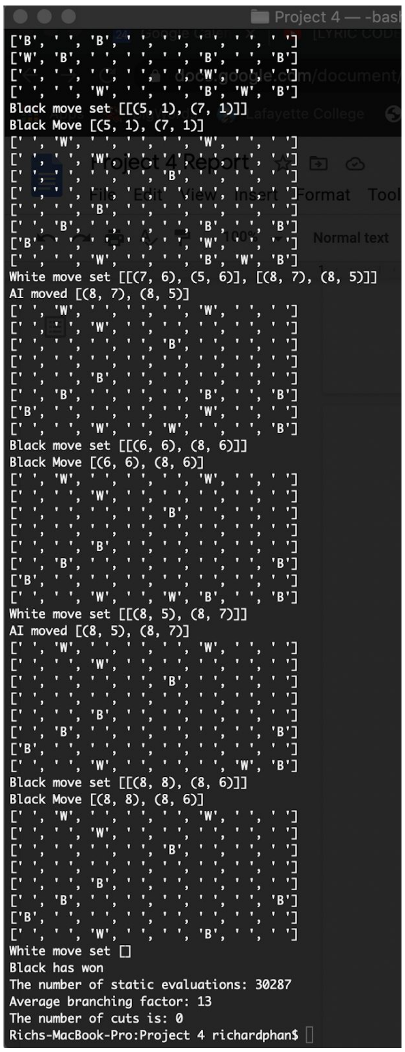
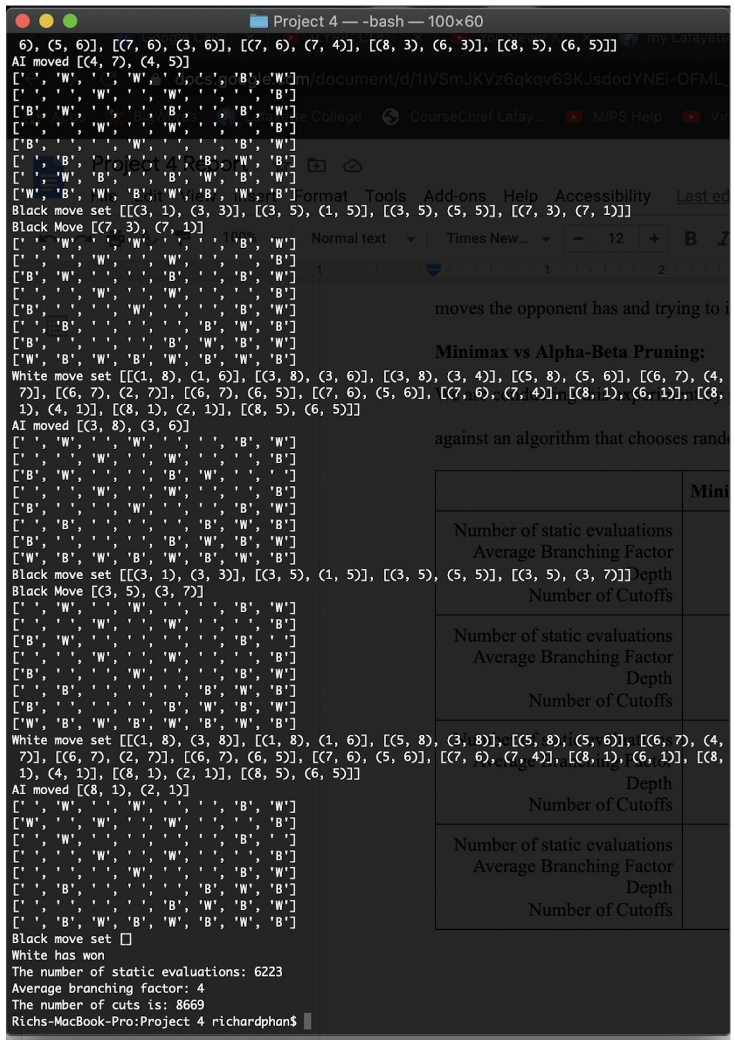
Richard Phan and Amanuel Zewge
CS 420 Artificial Intelligence
Professor Tao
October 25, 2020

Project 4 Report

Screenshots:





Pseudo Code for Minimax, Minimax + alpha-beta pruning:

Minimax Algorithm

1 function minimaxDecision(state, turn, iteration):

```
2       t = all of the descendants of a game tree node gb
3       if turn equals white:
4               return arg min_{a in actions(s)} minValue(result(state, a))
5
6       return arg max_{a in actions(s)} maxValue(result(state, a))
7
8 function maxValue(turn, tree, state, iterations):
9       if terminalTest(state) or iterations == 0:
10              evaluations += 1 find state in tree
11              return utility(state)
12      add to branching factor
13      if turn equals black, then turn = white
14      else if turn equals white, then turn = black
15      for each a in actions(state):
16              v is the max(v, minValue(tree, result(a, state), iteration - 1)
17      return v
18
19 function minValue(turn, tree, state, iterations):
20      if terminalTest(state) or iterations == 0:
21              evaluations += 1 find state in tree
22              return utility(state)
23      add to branching factor
24      if turn equals black, then turn = white
25      else if turn equals white, then turn = black
26      for each a in actions(state):
27              v is the min(v, maxValue(tree, result(a,state), iterations - 1)
28      return v
```

Variations:

- Lines 1, 8, 9, 19, 20 include “iterations” because we do not want to traverse through the entire game tree. This parameter allows us to stop the algorithm at a certain depth
- Line 2 is needed to create the tree that the minimax algorithm will traverse through. It references an instance of the Tree class
- Line 3 allows the different players to maximize or minimize respectively.
- Lines 10, 21 and 12,23 allow the algorithm to calculate the number of static evaluations and the branching factor respectively

Minimax Algorithm with Alpha Beta Pruning

```
1 function miniMaxAB(color of current player, depth of tree needed):
2       Add the number of valid moves to branching factor sum
3       Make the tree we will be using t
4       if it is whites turn:
5               return arg min_{a in actions(s)} minValueAB(t.root,-inf,inf)
6       else:
7               return arg max_{a in actions(s)} maxValueAB(t.root,-inf,inf)
8
9 function maxValueAB(node, alpha, beta):
10      if terminalTest(node) :
11              Return evaluation of the node and count one evaluation done
12      v = -inf
13      For every child of node:
14              v ← MAX(v, minValueAB(child,,α, β))
15      if v ≥ β:
16              count a cut
17              Add the number branches explored so far to branching factor
18               return v
19      α ← MAX(α, v)
20
21 function minValueAB(node, alpha, beta):
22      if terminalTest(node) :
23              Return evaluation of the node and count one evaluation done
24      v = inf
25      For every child of node:
26              v ← MIN(v, maxValueAB(child,α, β))
27      if v ≥ β:
28              count a cut
29              Add the number branches explored so far to branching factor
30              return v
31      α ← MAX(α, v)
```

Variations:

- Line 3 creates a tree from the current game state.
- Lines 2,17,29 where I keep track of the branching factor

- Lines 16,28 where I count the number of cuts the algorithm makes
- Lines 11 and 23 where I keep track of the number of static evaluations done

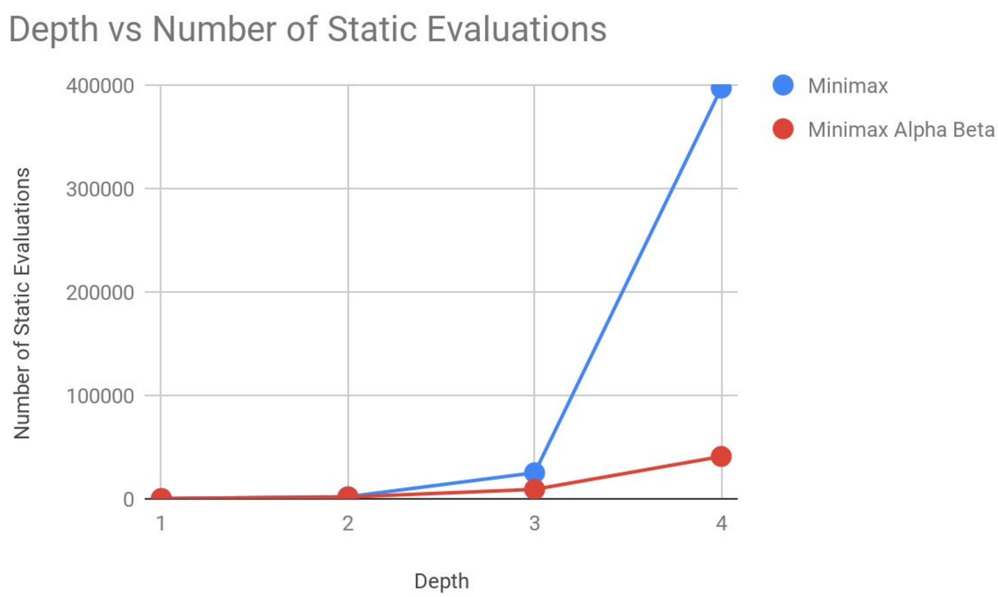
Explanation of static evaluation function:

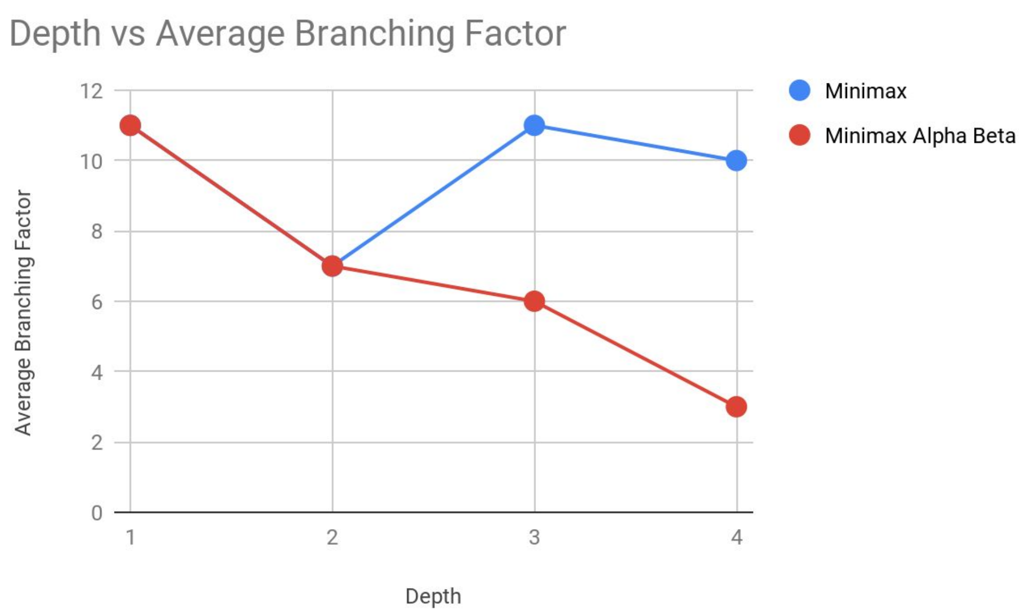
Our evaluation function takes in a board as a parameter and subtracts the number of legal moves for the black player and the number of legal moves for the white player. This makes it so that the black player looks for the most positive utility and the white player looks for the most negative utility. This means that each player is simultaneously trying to decrease the number of valid moves the opponent has and trying to increase their own moveset.

Minimax vs Alpha-Beta Pruning:

We are conducting this experiment by playing either minimax or minimax alpha-beta pruning against an algorithm that chooses random moves.

	Minimax	**Alpha-Beta Pruning**
Number of static evaluations Average Branching Factor Depth Number of Cutoffs	275 11 1 N/A	255 11 1 0
Number of static evaluations Average Branching Factor Depth Number of Cutoffs	2,081 7 2 N/A	1,791 7 2 0
Number of static evaluations Average Branching Factor Depth Number of Cutoffs	25,218 11 3 N/A	9,168 6 3 8,834
Number of static evaluations Average Branching Factor Depth Number of Cutoffs	396,981 10 4 N/A	40,944 3 4 53,590





As we can see from the daa, Alpha Beta pruning does not give many advantages on shallow trees. This is probably because the algorithm needs to check multiple levels of a tree before performing a cut and with only 1 or 2 levels this may not be possible. For deeper trees however, Alpha Beta pruning requires significantly fewer evaluations, and it dramatically reduces average branching factor. This effect seems to scale up especially for number of evaluations which increases much slower with alpha beta pruning that without. In general, this causes the algorithm to run faster and more efficiently.