CS 457 Fall 2019
Checkers AI
Hailee Kiesecker
Lucas Marchand

Let the record show that we named our program Trevor

Implementation:

The implementation of this checkers artificial intelligence computer program correlated greatly to the implementation covered in class. Using the provided source code and lecture periods we received a clearer idea on how to implement the minimax algorithm along with alpha-beta pruning.

The alpha-beta pruning was able to be accomplished within our FindBestMove method by keeping track of the alpha and beta throughout our recursion of both MinVal and MaxVal. The returned value from our depth recursion of minimax was then compared to the currently found best move value. The following pseudo code illustrates this idea:

```
FindBestMove(int player):
     struct state state
     For each move the player has in state:
          If current move is better than foundBestMove
               foundBestMove = current move
Return foundBestMove

MaxVal:
     Val = -infinity
     For each move in the current board
```

```
          Val = max(MinVal( of passed in alpha and beta))
Return Val

MinVal:
     Val = +infinity
     For each move in the current board
          Val = min(MaxVal( of passed in alpha and beta))
Return Val
```

For the board evaluation we did want to get pretty fancy but ended up running out of time. We changed the code around from the given original to have kings count as a lot more for and against the current player. What we wanted to do was modify the board evaluation to keep our pieces more within the middle of the board in a triangle shape. We also wanted to make it so that our players king board spaces were kept safe, so if they were moved off of when a current other move was available at about the same cost our player would keep themselves in the king position and move the other move.

Issues:

For almost all of our creation time we spent it on the minimax algorithm. We could not figure out why it was not returning the correct move when we thought our implementation was the correct way. After many, many, and let me say again, many hours of messing around with print statements and testing we discovered that the issue was coming in our maxdepth iterating downward too soon. Moments like this is when coding is not so much fun. Bellow is a screen shot for future reference on the correct location of depthLimit iteration, which should happen after we check if the depthLimit has been reached:

```
331  325
332  326    double MinVal(struct State *state,double alpha, double beta, int depthLimit){
333       -     depthLimit--;
334       -     fprintf(stderr, "in Min \n");
335       -     fprintf(stderr, "DepthLimit is %d \n", depthLimit);
336  331        // cutoff test
337  332        if(depthLimit<=0)return evalBoard(state->board);
338  333
          334  +     depthLimit--;
          335  +
```

Another not so much of a problem but rather mutual concern, we ran into was not fully understanding the board and values being passed in. we spent a few hours in just making sure that the provided program was functioning in the way that we assumed that it was.

Thoughts:

This program was oddly complicated to implement given the current code and the lecture periods provided in explaining the program itself. Many hours were spent backtracking and debugging a supposedly fairly easy to comprehend algorithm.

A lot of time was spent reading over Russell Stuart's given pseudo code in his artificial intelligence book located on page 170. Shown below for convenience purposes:

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

When initially implementing this algorithm from just our class time lectures our assignments and evaluations were slightly off. With more careful reading and changing our variables to more closely match the code above, we were able to get our Minimax algorithm to evaluate correctly.

If we had been more successful sooner with the minimax algorithm with alpha-beta pruning we were going to implement iterative deepening also explained on page 170 of Russell, Stuart's Artificial Intelligence.

Testing:

Throughout creating our program we would frequently use the provided other_players/depth with a level of 1 and run our *doit* script. Then after mastering depth one we moved on to two, and then year 1. *doit* shown below:

```
make clean
make
./checkers other_players/depth computer 3 -MaxDepth 1
```

Throughout our creation and testing process we frequently used strategically places fprintf statements to stderr. This allowed us to see current values, especially when running the graphical interface as using that allowed the returned values to appear slower on the screening as well as give a graphic of what those returned values had just caused.

Time Spent:

We spent around five meeting times together about 2-3 hours each time. During that time we evaluated the given program and tried getting our minimax algorithm to work along with alpha-beta. We also communicated through text messages while working alone expressing issues. Our alone time was spent debugging the program with our own styles and messing around with the functionality/implementation of minimax. We worked on everything together and the code we came up with became a mix of both of our implementations.