CSCE 420 - Spring 2023

Assignment P2

due: Tues, Mar 21, 5:00pm, pushed to your Github account

Objective

The objective of this assignment is to write an interactive (text-based) program in C++ or Python for playing tic-tac-toe using the **Minimax** algorithm with **alpha-beta pruning**. The point is to observe that Minimax search produces intelligent decisions within the game, which is as good as any rule-based strategy. Although alpha-beta pruning is not necessary for tic-tac-toe, it does improve the efficiency of the search (measured as number of nodes or game states searched).

Tic-tac-toe is a relatively easy game, in that the search space is actually not that large (depth at most 9). However, the same algorithm could be used for more complex games, like checkers, chess, or connect4. In these games, the search tree is much deeper, and the branching factor much higher. In addition to alpha-beta pruning and checking for visited states, it would probably be necessary to also truncate the search at a predetermined depth-limit and apply a heuristic board evaluation function (to estimate the probability of winning from incomplete board states). Despite these additional tricks, most AI programs for playing such games still basically rely on Minimax search. I predict that, once you get this program working, you will be surprised at how it autonomously figures out the right (i.e. most intelligent) move to make for tic-tac-toe in each situation.

Assignment Tasks Overview (TLDR):

- Create command line program for tic-tac-toe (specific requirements below)
- Allow the user pick moves (for both X and O)
- Implement a minimax algorithm as an AI player
- List the minimax value of all possible moves for a particular player and number of nodes explored
- Implement alpha-beta pruning for the minimax algorithm
- Record the output of test cases

Program Interface:

The program will be driven from the command-line using commands. The pieces are X and O ('.' represents empty). The coordinates of positions on the board are: rows=A,B,C (top-down), columns=1,2,3 (left-to-right).

```
 (A 1) (A 2) (A 3)
 (B 1) (B 2) (B 3)
 (C 1) (C 2) (C 3)
```

'move' places a piece in a position (player's choice). 'choose' asks the computer to decide where to place the piece (by calling Minimax). Here is a brief example to illustrate:

```
> ttt
Welcome to Tic-Tac-Toe
. . .
. . .
. . .

> move X A 2
. X .
. . .
. . .

> choose O
O X .
. . .
. . .
> move X B 2
O X .
. X .
. . .

> choose O
O X .
. X .
. O .  // O's move is forced to block possible win by X
```

Note that 'move' and 'choose' specify the piece, which allows us (as the developer) to place pieces (X or O) in any desired positions, or you can **simulate the computer playing against itself** by iterating 'choose X' and 'choose O' repeatedly. Whenever a terminal state (end game) is reached, the program should detect this and print either 'X wins', 'O wins', or 'it's a draw'.

Importantly, if your implementation of Minimax is working correctly, then the program should autonomously: 1) take winning moves when it can, and 2) choose positions to block potential wins by the opponent (i.e. when the opponent has set up 2 in a row). If you play the computer against itself from scratch, it should end in a draw.

With each call to 'choose', the program should also print out some tracing information. You should **display the minimax scores for each of the possible initial moves** being considered (at the top-level) (relative to the piece that is being played). This will probably help you in debugging. Also **print out number of nodes searched** for each call to choose() (i.e. use a global counter that is incremented for each call of *minscore*() or *maxscore*()). For example:

```
O X .
. X .
. . .
> choose O
move (A,3) mm_score: -1.0  // these values are based on the default utility function
move (B,1) mm_score: -1.0
move (B,3) mm_score: -1.0
move (C,1) mm_score: -1.0
move (C,2) mm_score: 0.0
move (C,3) mm_score: -1.0
```

```
number of nodes searched: 373
O X .
. X .
. O .
```

Notice how the computer realizes that any move other than (C 2) is a guaranteed loss, while putting an O in (C 2) guarantees at least a draw.

Here are the commands your program should accept at the command line:

- **show**: shows the current state of the board
- **reset**: resets the state of the board to empty (removes all the pieces)
- **move P R C**: put piece P (X or O) in row R (A, B, or C) and column C (1, 2, or 3)
- **choose P**: invoke Minimax to figure out the optimal place to put piece P on the board
- **pruning**: show the state of pruning (on or off)
- **pruning on | off**: turn alpha-beta pruning on or off
- **quit**

Print out the updated board configuration after every command.

Breaking Ties with a Modified Utility Function

The default utility function for tic-tac-toe assigns +1 for a win (for a given player), -1 for a loss, and 0 for a draw. However, in some cases, this creates ambiguity and leads to the appearance of sub-optimal choices by the computer. For example, if the minimax score of all possible top-level moves for a given piece are tied at -1, the computer realizes it can't win and chooses an action at random (instead of intentionally blocking an imminent win by the opponent, for example). To help break these ties in Minimax, you can implement a **modified utility function** as follows:

- `win  = +1 - 0.1*depth`
- `loss = -1 + 0.1*depth`
- `draw = 0`

This is simple to do, and merely requires adding a 'depth' argument to *minscore*() and *maxscore*(), which gets incremented by 1 with each recursive call descending into the search tree.

The rationale behind this modified utility function is that paths to *shallower winning nodes* are preferred, and otherwise paths to *deeper losing nodes* are preferred. This should produce more intelligent decisions in Minimax by opportunistically taking wins whenever possible, and effectively deferring/delaying losses as long as possible.

Hints:

- It is not necessary to keep track of whose "turn" it is; this is why the piece is specified for the 'move' and 'choose' commands; this makes things more flexible for debugging.
- When you generate children states (moves), be sure to make them as copies of the parent state, so you don't modify the object itself.
- You can put in error-checking if you want, but we won't intentionally test it with illegal inputs.
- Your program should have **2 recursive functions** that call each other: *minscore*() and *maxscore*(). For the base case (where one of the players has won the game), be sure to return the appropriate value (depending on which piece wins). You need to pass in the id of the player making the move to *maxscore*() and *minscore*(), so the utility at the leaves (i.e. end games) can be given the appropriate sign. For example, in a state s where X has 3 in a row, if player=X then utility(s)=+1, and if player=O, then utility(s)=-1. The pseudocode in the AIMA textbook left out the 'player' argument in these 2 functions. For either player, the outer loop always chooses the action that maximizes the maxscore for that player.
- Do not worry about precisely replicating the examples shown in this hand out. Everybody's implementation will be slightly different. The results can be affected by things like the specific order in which successor nodes are generated. This will cause variations in the number of nodes searched, etc. Still, your program should produce intelligent decisions in the game.

What to Turn In:

Put these files in subdirectory called 'programming_assignment_2' of your github project for this course (same repo as for PA1).

- **if C++:**
  - **ttt.cpp** (and any other source code files)
  - **makefile** (with any flags necessary to compile on compute.cs.tamu.edu)
- **if Python: ttt.py**
- **README.txt –** documentation of how to run your program (command-line arguments and flags; interactive commands, etc)
- **Transcripts –** text files showing show transcripts of the Test Cases (games) defined below (you can call them Transcript1.txt, Transcript2.txt, etc)
- **RESULTS.txt** - Show the effect of alpha-beta pruning on number of nodes expanded in the search tree after each step for Game1, for where the computer plays a complete game against itself with and without alpha-beta pruning. (See the 'summary of number of nodes searched' below.)

Grading:

- 20% - If C++, does it compile on compute.cs.tamu.edu (by running 'make')? If Python, does it run with python3 on compute.cs.tamu.edu?
- 20% - Does it run on simple test cases?
- 20% - Does the implementation of Minimax look correct? (e.g. minscore() and maxscore() functions)
- 20% - Does it produce the right decisions? (does 'choose' make optimal moves, as shown in the Transcripts?)
- 20% - Does alpha-beta pruning improve the efficiency of the search? (as shown in your table of number of nodes searched in RESULTS.txt)

Test Cases

Game 1: computer plays against itself (iterate 'choose X' and 'choose O' till end game)

```
. . .
. . .
. . .
> choose X
> choose O
…
```

Game 2: Start with an X in the middle; then let computer choose all subsequent moves.

```
> move X B 2
. . .
. X .
. . .
> choose O
…
```

Game 3: Start with an X in the U-L corner; then let computer choose all subsequent moves.

```
> move X A 1
X . .
. . .
. . .
> choose O
…
```

Game 4: Start with an X in the middle of top edge; then let computer choose all subsequent moves.

```
> move X A 2
. X .
. . .
. . .
> choose O
…
```

Game 5: Let's see if we can force a win for X by making a sub-optimal move for O. Try placing an X in the middle and O on the side; then let the computer play the rest of the game against itself.

```
> move X B 2
> move O A 2 // this is sub-optimal for O
. O .
. X .
. . .
> choose X
> choose O
…
```

__**Transcript1.txt** (using the modified utility function and alpha-beta pruning on game1)__

```
> ttt
Welcome to Tic-Tac-Toe
. . .
. . .
. . .
> pruning on
pruning=1
> choose X
move (A,1) mm-score: 0.0
move (A,2) mm-score: 0.0
move (A,3) mm-score: 0.0
move (B,1) mm-score: 0.0
move (B,2) mm-score: 0.0
move (B,3) mm-score: 0.0
move (C,1) mm-score: 0.0
move (C,2) mm-score: 0.0
move (C,3) mm-score: 0.0
number of nodes searched: 34202
X . .
. . .
. . .

> choose O
move (A,2) mm-score: -0.5
move (A,3) mm-score: -0.5
move (B,1) mm-score: -0.5
move (B,2) mm-score: 0.0
move (B,3) mm-score: -0.5
move (C,1) mm-score: -0.5
move (C,2) mm-score: -0.5
move (C,3) mm-score: -0.5
number of nodes searched: 6304
X . .
. O .
. . .
> choose X
move (A,2) mm-score: 0.0
move (A,3) mm-score: 0.0
move (B,1) mm-score: 0.0
move (B,3) mm-score: 0.0
move (C,1) mm-score: 0.0
move (C,2) mm-score: 0.0
move (C,3) mm-score: 0.0
number of nodes searched: 1630
X X .
. O .
. . .
> choose O
move (A,3) mm-score: 0.0
move (B,1) mm-score: -0.9
move (B,3) mm-score: -0.9
move (C,1) mm-score: -0.9
move (C,2) mm-score: -0.9
move (C,3) mm-score: -0.9
number of nodes searched: 220
X X O
. O .
. . .
> choose X
move (B,1) mm-score: -0.9
move (B,3) mm-score: -0.9
move (C,1) mm-score: 0.0
move (C,2) mm-score: -0.9
move (C,3) mm-score: -0.9
number of nodes searched: 97
X X O
. O .
X . .
> choose O
move (B,1) mm-score: 0.0
move (B,3) mm-score: -0.9
move (C,2) mm-score: -0.9
move (C,3) mm-score: -0.9
number of nodes searched: 32
X X O
O O .
X . .
> choose X
move (B,3) mm-score: 0.0
move (C,2) mm-score: -0.9
move (C,3) mm-score: -0.9
number of nodes searched: 13
X X O
O O X
X . .
> choose O
move (C,2) mm-score: 0.0
move (C,3) mm-score: 0.0
number of nodes searched: 4
X X O
O O X
X O .
> choose X
move (C,3) mm-score: 0.0
number of nodes searched: 1
X X O
O O X
X O X
```

```
*** it's a draw! ***
> quit
```

(Use grep to extract these lines from the transcripts...)

```
summary of nodes searched WITHOUT alpha-beta pruning for game1:
number of nodes searched: 549945
number of nodes searched: 59704
number of nodes searched: 7331
number of nodes searched: 934
number of nodes searched: 197
number of nodes searched: 46
number of nodes searched: 13
number of nodes searched: 4
number of nodes searched: 1

summary of nodes searched WITH alpha-beta pruning for game1:
number of nodes searched: 34202
number of nodes searched: 6304
number of nodes searched: 1630
number of nodes searched: 220
number of nodes searched: 97
number of nodes searched: 32
number of nodes searched: 13
number of nodes searched: 4
number of nodes searched: 1
```