Project Report

Theo Bragstad

Code: https://github.com/Theobragstad/tictactoe

Explanation of what your code does and the libraries and frameworks you used

My code is a Tic-tac-toe game that runs in the terminal. It has the following game modes: 2-player (normal), Random vs You, Random vs Random, AI (Minimax) vs You, AI (Minimax) vs AI (Minimax), AI (Minimax) vs Random, AI (Alpha-beta) vs You, AI (Alpha-beta) vs AI (Alpha-beta), AI (Alpha-beta) vs Random, and AI (Minimax) vs AI (Alpha-beta). Game modes not involving the user (a human player) can be simulated for multiple games, and stats are displayed at the end of each simulation. The game also allows the user to set custom player icons, set a custom board size, and set a custom max depth for the AI player algorithm.

Libraries used: time, random, datetime
Frameworks used: none

Test results and a description of how you tested it

Test results as per the project directions (note: additional test results are available in the ("An example of how your application works" section below):

100 games of random player against random player:

```
Random (❌, P1) vs Random (⭕, P2)
3x3
100 games played

Average game duration (moves only) (sec): 2e-05


❌ wins: 56 (56.0%)
⭕ wins: 28 (28.0%)
   ties: 16 (16.0%)

❌ average move duration (sec): 0.0
⭕ average move duration (sec): 0.0

❌ average total moves to win: 4.17857
⭕ average total moves to win: 3.67857
   average moves to reach end game: 7.62
```

AI player that uses minimax to choose the best move with a variable number of plies:
(I wasn't really sure what this meant, so I made the Minimax player select a random number from 1-10 on each turn to use as its max depth parameter, and then played 100 games of it against a random player. (Disregard the max depth that is displayed; the random selection was done internal to the AIPlayer class, and the default in the TicTacToe class is 9.))

```
AI (Minimax) (❌, P1) vs Random (⭕, P2)
3x3, Max Depth: 9
100 games played

Average game duration (moves only) (sec): 1.50061


❌ wins: 95 (95.0%)
⭕ wins: 4 (4.0%)
   ties: 1 (1.0%)

❌ average move duration (sec): 0.46458
⭕ average move duration (sec): 0.0

❌ average total moves to win: 3.21053
⭕ average total moves to win: 3.25
   average moves to reach end game: 5.5
```

100 games with the random player against the minimax AI player at a depth of 5 plies:

```
Random (❌, P1) vs AI (Minimax) (⭕, P2)
3x3, Max Depth: 5
100 games played

Average game duration (moves only) (sec): 0.08615


❌ wins: 11 (11.0%)
⭕ wins: 77 (77.0%)
   ties: 12 (12.0%)

❌ average move duration (sec): 0.0
⭕ average move duration (sec): 0.02587

❌ average total moves to win: 3.0
⭕ average total moves to win: 3.27273
   average moves to reach end game: 6.89
```

Is your AI player better than random chance? Write a paragraph or two describing or why not:
Yes, it is better than random chance. This is because it utilizes the minimax algorithm to select the most optimal move at each state. Although the max depth of 5 is non-optimal for a 3x3 board (since it does not search the whole game tree), it is still doing some calculations, so it selects better moves on average than random, and therefore wins the majority of the games, although with a non-optimal max depth there will be more ties and more opponent wins.

100 games with the random player against the Alpha-Beta AI player at a depth of 5 plies:

```
Random (❌, P1) vs AI (Alpha-beta) (⭕, P2)
3x3, Max Depth: 5
100 games played

Average game duration (moves only) (sec): 0.01434


❌ wins: 11 (11.0%)
⭕ wins: 78 (78.0%)
   ties: 11 (11.0%)

❌ average move duration (sec): 0.0
⭕ average move duration (sec): 0.00423

❌ average total moves to win: 3.18182
⭕ average total moves to win: 3.33333
   average moves to reach end game: 7.0
```

Are your results for this part different from those for your minimax AI player? Write a paragraph or two describing why or why not:
The win percentages are very similar. The main difference is the time taken. The average game duration for the alpha-beta AI player is ~0.01, while the average game duration for the minimax AI player was ~0.09. The average move duration for the alpha-beta AI player was ~0.004, while the average move duration for the minimax AI player was ~0.03. Clearly, the alpha-beta AI player is much faster (about 10x faster).

100 games with the random player against the Alpha-Beta AI player at a depth of 10 plies:

```
Random (❌, P1) vs AI (Alpha-beta) (⭕, P2)
3x3, Max Depth: 10
100 games played

Average game duration (moves only) (sec): 0.03298


❌ wins: 0 (0.0%)
⭕ wins: 82 (82.0%)
   ties: 18 (18.0%)

❌ average move duration (sec): 0.0
⭕ average move duration (sec): 0.00916

❌ average total moves to win: 0
⭕ average total moves to win: 3.5122
   average moves to reach end game: 7.38
```

Does increasing the number of plies improve the play for our AI player? Why or why not?
Yes, the win percentage increased for the AI player, and now the random player never won. The time taken increased very slightly. Both of these results make sense. The AI player is exploring deeper into the game tree, so it finds more optimal moves, but this takes a bit longer. Also note that for a 3x3 board, a max depth of 10 is slightly excessive: a max depth of 9 is equivalent, because there are at most 9 moves (i.e. when the AI player goes first). A max depth of 10 for a 3x3 board will always return the most optimal move, so it is impossible to beat it; the best its opponent (of any type) could do is tie it.

An example of how your application works

My code implements functionality for 3 types of players: human, random, and AI.

It also implements a user-friendly interface that allows the user to select from one of 10 game modes that involve the various player types. The code is encapsulated in the TicTacToe class. Most of the code for the game interface is not especially relevant to the AI concepts we've learned in class, but the core functionality includes:

- play(), which manages calling of all the required functions for each round of the game, including displaying the board, getting the next move, and checking if the game is over
    - It also handles stat calculations for simulated games
- get_move(), which handles turn-taking by requesting moves from the corresponding player depending on the selected game mode
- update_board(), which adds a move to the current board
- check_game_over(), which checks if either player has won or if there is a tie

- functions for displaying the board and the available moves
- various helper functions for handling user input

The TicTacToe class imports the random and AI players which are in external classes.

The random player code is very simple. It selects a random move from the available moves and returns it.

The AI player code implements minimax and alpha-beta minimax players.
To build my code for this part, I first made sure I understood the content from class where we covered minimax and alpha-beta pruning. I also reviewed content on game trees, depth, and utility functions. Since Tic-tac-toe is relatively simple, I did not need to create an example decision tree but rather began coding it and testing it to see the progress I made. The hardest part of the code was implementing the minimax() and minimax_alphabeta() functions, so I saved those for last and implemented the other functions first, because I knew I would need them in the minimax algorithms:

- init()
    - Sets the icon for the player that we want to act as the AI
    - Sets the icon for the player that we want to act as the opponent
    - Sets the desired algorithm type
    - Sets the specified max depth. In the TicTacToe class, this defaults to the number of squares on the board (i.e. 9 for a 3x3 board), but the user can specify it. The default in the AIPlayer class is positive infinity, so if a max depth is never specified anywhere it will explore the whole tree.
- get_move()
    - Simply calls either minimax() or minimax_alphabeta(), depending on the specified algorithm type
- get_available_moves()
    - Returns the moves that are currently available based on the specified board (available moves are squares that are empty)
- make_move()
    - Returns a new board, where the specified player has made a move on the specified board
- is_board_full()
    - Checks if there are no empty squares on the board, i.e. there is a tie.
- check_winner()
    - For the passed board, checks if the passed player marker has won (on an n x n board, if they have gotten any n moves in a row, column, or diagonal)
- update_icons()
    - Since the TicTacToe class allows the user to set custom icons, this is called when they are set, so that the AI player uses the current set of icons when it runs.

How minimax() works:

Takes the board, current player icon, current depth (initialized to 0), and max depth as parameters. Checks if the opponent has won. If so, returns -1 for the score and no move.

Checks if the AI player has won. If so, returns 1 for the score and no move.
Checks if there is a tie. If so, returns 0 for the score and no move.

Initializes a list of scores.
Goes through every available move for the current board.
Creates a new board where the current available move has been made.
Checks if the current depth is less than the max depth.

If so, recurses using the new board, swaps the current player and increments the depth. The result of this call will be the score for this move.

If not, sets the score to 0 and the move to none.

Adds the score to the list of scores.

If the current player icon for this call to minimax() is the AI player's, then we set the best score to the one with the maximum score in the score list, and return the corresponding score as well as the available move corresponding to that index.

Else (if the current player icon is the opponent's), we set the best score to the one with the minimum score in the score list, and return the corresponding score as well as the available move corresponding to that index.

This algorithm was fairly straightforward to implement. The initial checks make sense, and the loop through each available move is logical. The final checks are the key part of the algorithm: maximize the score for the AI player, and minimize the score for the opponent.

How minimax_alphabeta() works:

Very similar to minimax(), except it adds two new parameters: alpha and beta. Alpha is initialized to negative infinity while beta is initialized to positive infinity. This ensures that on the first setting of alpha and beta we choose the first score for the starting point (the max of negative infinity with any number > 0 will always be that other number, and vice versa for positive infinity).

Everything is the same as minimax(), except for the part after the score is added to the list of scores. Within the loop, after getting the new score, we update alpha or beta depending on the current player icon. If it's the AI player's, we update alpha to the maximum of itself and the score we just added. If it's the opponent's turn, we update beta to the minimum of itself and the score we just added. This is a key step, because it's what allows us to prune the game tree in the next step: if alpha ever becomes greater than or equal to beta, we know we don't have to explore down that branch in the game tree any deeper, and we can break out of the for loop and continue recursion down another branch, saving time.

This algorithm was also fairly straightforward to implement, but a bit more difficult to conceptualize than the simple minimax algorithm. I knew what to initialize alpha and beta to, and I knew to maximize alpha on the AI player's turn and minimize beta on the opponent's turn, but the pruning condition was

a bit tricky at first. Initially I had it reversed, but when I realized I saw no performance improvements over the simple minimax player in terms of runtime, I reassessed the pruning algorithm and realized that we want to prune when we have found an alpha greater than or equal to beta, because this represents that we have found a move for the AI player with a score higher than or equal to the best move for the opponent based on the current state, so there's no point in exploring any further because the opponent has already been beaten on that particular branch of the game tree.

How did you validate your results? What are the results?

I validated the success of my AI player implementations in two main ways: by playing against them myself, and by simulating multiple games with varying types of opponents against each other using my game interface implemented in the TicTacToe class. It's hard to capture all the results here; the best way to see them is to run simulations yourself to see results in real-time.

- When I played against both versions of the AI, I was unable to beat them and our 3x3 games always resulted in ties. I experimented with other board sizes as well, and the best I could do was tie them, regardless of who went first. This helped me validate my results because it was clear they were making optimal decisions. My alpha-beta code was validated because it made moves substantially faster than the basic minimax algorithm. Variable depths also followed logical patterns (for example, when the max depth was less than the number of possible moves, the AI was not as good, but it was faster).
- My results were also validated when I simulated games of the AI players against each other and against a random player. When the max depth was optimal, the AI always beat or at least tied the random player. This makes sense, because the AI players should be better than random, but very rarely, the random player may make optimal moves. Results vary due to randomness, but in one set of simulations with a standard 3x3 board and a max depth of 9, the results were as follows:

```
AI (Minimax) (❌, P1) vs Random (⭕, P2)

100 games played

Average game duration (moves only) (sec): 3.76765

❌ wins: 100 (100.0%)
⭕ wins: 0 (0.0%)
   ties: 0 (0.0%)

❌ average move duration (sec): 1.15218
⭕ average move duration (sec): 0.0

❌ average total moves to win: 3.27
⭕ average total moves to win: 0
   average moves to reach end game: 5.54
```

```
AI (Alpha-beta) (❌, P1) vs Random (⭕, P2)

100 games played

Average game duration (moves only) (sec): 0.14091

❌ wins: 98 (98.0%)
⭕ wins: 0 (0.0%)
   ties: 2 (2.0%)

❌ average move duration (sec): 0.04169
⭕ average move duration (sec): 0.0

❌ average total moves to win: 3.34694
⭕ average total moves to win: 0
   average moves to reach end game: 5.76
```

Results for a 4x4 board with max depth of 3:

```
AI (Minimax) (❌, P1) vs Random (⭕, P2)


100 games played

Average game duration (moves only) (sec): 0.1403


❌ wins: 77 (77.0%)
⭕ wins: 0 (0.0%)
   ties: 23 (23.0%)

❌ average move duration (sec): 0.02505
⭕ average move duration (sec): 0.0

❌ average total moves to win: 4.88312
⭕ average total moves to win: 0
   average moves to reach end game: 10.43
```

```
AI (Alpha-beta) (❌, P1) vs Random (⭕, P2)


100 games played

Average game duration (moves only) (sec): 0.01462


❌ wins: 79 (79.0%)
⭕ wins: 2 (2.0%)
   ties: 19 (19.0%)

❌ average move duration (sec): 0.00266
⭕ average move duration (sec): 0.0

❌ average total moves to win: 4.8481
⭕ average total moves to win: 7.0
   average moves to reach end game: 10.19
```

Since the underlying algorithms are the same, simulations of Minimax vs Minimax, Alpha-beta vs Alpha-beta, and Minimax vs Alpha-beta result in ties 100% of the time, assuming the max depth is set to a value high enough so that the players can explore the full game tree. On simulations where this was not the case, player 1 won, because the first player has the advantage and can win if the other player does not always make the optimal move. Examples are below.

Results for a 3x3 board with a max depth of 9 for Minimax vs Alpha-beta:

```
AI (Minimax) (❌, P1) vs AI (Alpha-beta) (⭕, P2)


10 games played

Average game duration (moves only) (sec): 3.74845


❌ wins: 0 (0.0%)
⭕ wins: 0 (0.0%)
   ties: 10 (100.0%)

❌ average move duration (sec): 0.74627
⭕ average move duration (sec): 0.00428

❌ average total moves to win: 0
⭕ average total moves to win: 0
   average moves to reach end game: 9.0
```

Results for a 3x3 board with a max depth of 3 for Minimax vs Alpha-beta:

```
AI (Minimax) (❌, P1) vs AI (Alpha-beta) (⭕, P2)


10 games played

Average game duration (moves only) (sec): 0.01899


❌ wins: 10 (100.0%)
⭕ wins: 0 (0.0%)
   ties: 0 (0.0%)

❌ average move duration (sec): 0.00415
⭕ average move duration (sec): 0.0008

❌ average total moves to win: 4.0
⭕ average total moves to win: 0
   average moves to reach end game: 7.0
```

Since the max depth is non-optimal for the board size, player 1 wins because of the player 1 advantage in Tic-tac-toe.

Results for a 5x5 board with a max depth of 3 for Alpha-beta vs Random:

```
AI (Alpha-beta) (❌, P1) vs Random (⭕, P2)
5x5, Max Depth: 3
100 games played

Average game duration (moves only) (sec): 0.04701


❌ wins: 79 (79.0%)
⭕ wins: 0 (0.0%)
   ties: 21 (21.0%)

❌ average move duration (sec): 0.00614
⭕ average move duration (sec): 0.0

❌ average total moves to win: 6.24051
⭕ average total moves to win: 0
   average moves to reach end game: 14.32
```

Despite non-optimal max depth given the board size, AI still outperforms Random because it makes more optimal moves than random choice, even if it does not make the absolute most optimal move every time.

Results for 3x3 Random vs Random:

```
Random (❌, P1) vs Random (⭕, P2)


100 games played

Average game duration (moves only) (sec): 2e-05


❌ wins: 58 (58.0%)
⭕ wins: 23 (23.0%)
   ties: 19 (19.0%)

❌ average move duration (sec): 0.0
⭕ average move duration (sec): 0.0

❌ average total moves to win: 4.10345
⭕ average total moves to win: 3.73913
   average moves to reach end game: 7.61
```

Clearly, player 1 has an advantage in Tic-tac-toe no matter the player type.

2-3 paragraph conclusion about what you learned

In this project, I learned a lot about how to write the code for an AI player that uses the minimax and alpha-beta algorithms. It helped me understand game trees and how concepts like depth and alpha-beta pruning can be used to help an algorithm make optimal decisions in an adversarial environment. I also gained an improved understanding of Tic-tac-toe (and similar games in general), particularly around how the game plays out with different methods of play and larger-than-standard boards.

I gained a deep understanding of the minimax algorithm and how it works by using recursion to evaluate possible moves in the game tree so it can find the optimal move for the AI player, based on the current state of the board, the icon of the current player making the move, and the current depth of the game tree. I learned about the utility function, and how the algorithm must act so that it selects the move with the maximum score for the AI player, while selecting the move with the minimum score if it is the opponent's turn. The alpha-beta player also utilizes minimax, but applies the concept of alpha-beta pruning during the loop. If the current player is the AI, it updates the alpha value by taking the maximum of the current alpha and the score. If the current player is the opponent, it updates the beta value by taking the minimum of the current beta and the score. If alpha is greater than or equal to beta, the loop ends early, because we know that the remaining moves will not affect the final result due to pruning.Compared to the normal minimax algorithm, the alpha-beta pruning helps to reduce the number of nodes explored in the search tree, making the algorithm more efficient by eliminating unnecessary branches. This is done by maintaining alpha (the best score found so far for the maximizing player) and beta (the best score found so far for the minimizing player) and cutting off branches when it's clear they won't affect the final result. During the search, if the current score is better than beta for the minimizing player, it means that the maximizing player (AI) has found a move that is at least as good as beta, and the minimizing player (opponent) will not choose this path, so the algorithm can prune the remaining branches.

I also learned about the depth parameter, which represents how deep the algorithm should explore the game tree during its recursion. A deeper exploration means considering more moves into the future, but limiting the depth can be useful in situations where the game tree is too large to explore

fully, and we want to balance computational efficiency with the quality of the AI's decision. For example, not setting a max depth value and leaving it to its default value of infinity will mean the AI will explore the entire game tree until it reaches a terminal state, but on larger games, such as 10x10 Tic-tac-toe, the algorithm will take unreasonably long to finish if the max depth is too high.