

Xiaoning Guo

NetID: Xguo24

CSC 242 – Project 1

## Tic-Tac-Toe AI

### Contents

- How to Run
- How to Use the Program
- Expected Output
- Description of Program Design
- How the Program Determines Depth of Search
- How the Program Decides its Next Move
- The Effects of Alpha Beta Handling
- Possible Future Changes

### How to Run

This program is written in Java.

Make sure the build files (.class) from the bin folder are all in the same directory as the terminal.

For example, if the build files are on the desktop, then change the terminal directory to the desktop.

```
C:\Users\Xiaoning Guo>cd desktop
```

```
C:\Users\Xiaoning Guo\Desktop>
```

Then enter the command in this format: `java Driver [game type 0 or 1]`

For example, if you want to play basic Tic-Tac-Toe:

```
java Driver 0
```

However, if you want to play 9-board-Tic-Tac-Toe:

```
java Driver 1
```

If this doesn't work, try `Driver.class` instead.

## How to Use the Program

The program will first ask the user if they want to play “X” or “O”.

Enter a single character either “X” or “O”. This is not case sensitive.

When it is the user’s turn, enter a position from 1 through 9 that correspond to the board’s squares in the following fashion.

1	2	3
4	5	6
7	8	9

For example:

```
C:\Users\XNGJeisenLi\Desktop>java Driver 0
You are now playing basic Tic-Tac-Toe.
Do you want to play X or O? X goes first.
x
Enter your position move:
3
--X
---
---
```

When playing 9-board-Tic-Tac-Toe, the user needs to enter a number corresponding to a board on a grid of boards followed by a space and then a number position for that board. For example:


```
C:\Users\XNGJeisenLi\Desktop>java Driver 1
You are now playing 9-Board-Tic-Tac-Toe.
Do you want to play X or O? X goes first.
x
Enter your position move:
1 5
--- --- ---
-X- --- ---
--- --- ---
--- --- ---
--- --- ---
--- --- ---
--- --- ---
--- --- ---
--- --- ---
```

## Expected Output

After the user inputs a move, the AI will immediately calculate its move and then print out a search summary. It will then say, "My turn. I make the following move:" and then print out its move to System.out as indicated by the red arrow in the figure below. Of course, if your terminal differentiates System.err from System.out by color, the only thing colored will be the AI's move position. Finally, the program will prompt the user for the next move.

For basic Tic-Tac-Toe:


```
Nodes generated: 6138
Leaf nodes generated: 2503
Best utility: 0
Search time (milliseconds): 16

My turn. I make the following move:
3 
--O
-X-
---

Enter your position move:
```

For 9-board-Tic-Tac-Toe:

```
Maximum depth of search: 7
Nodes generated: 342967
Goal-state nodes found: 12948
Heuristic nodes generated: 231294
Best utility or heuristic: 0
Search time(milliseconds): 256

My turn. I make the following move:
5 6 
--- --- ---
-X- --- ---
--- --- ---

--- --- ---
--- --O ---
--- --- ---

--- --- ---
--- --- ---
--- --- ---

Enter your position move:
```

When the game ends, the AI will indicate whether the AI wins, the AI loses, or ties. Then it will prompt for a new game. Of course, the AI should never lose. By playing randomly, the AI wins in about 8-12 moves for 9-board-Tic-Tac-Toe.

For basic Tic-Tac-Toe:

```
Nodes generated: 31
Leaf nodes generated: 16
Best utility: 1
Search time (milliseconds): 1

My turn. I make the following move:
6

XXO
-XO
--O

Hahaha! I, the superior AI, win!
You are now playing basic Tic-Tac-Toe.
Do you want to play X or O? X goes first.
```

```
Enter your position move:
9
XXO
OOX
XOX

We tied.

You are now playing basic Tic-Tac-Toe.
Do you want to play X or O? X goes first.
```

For 9-board-Tic-Tac-Toe:

```
Maximum depth of search: 7
Nodes generated: 204173
Goal-state nodes found: 5392
Heuristic nodes generated: 127624
Best utility or heuristic: 10
Search time(millisecons): 102

My turn. I make the following move:
1 9

--O  X--  -X-
-XO  -O-  ---
--O  ---  ---

---  XO-  X--
---  ---  ---
---  ---  ---

---  ---  ---
---  ---  ---
---  ---  ---

Hahaha! I, the superior AI, win!

You are now playing 9-Board-Tic-Tac-Toe.
Do you want to play X or O? X goes first.
```

## Description of Program Design

The program consists of three class files, namely: Driver, Board, and Grid. Driver contains the main method which runs the game loops. There are two game loops which correspond to basic Tic-Tac-Toe and 9-board-Tic-Tac-Toe, and they create an instance of the game objects.

Board is the basic Tic-Tac-Toe game object which stores the game state in a 9-element integer array. It initializes all entries to 0 which signifies no moves have been made. The program always assumes the AI's piece is a positive 1 and the user to be a negative 1. If the AI makes a move on position 5, then on the 5<sup>th</sup> index of the array, there will be a 1. Additionally, there is a hashset that stores all possible moves that can be made on the current board. The make move method returns a deep copy of the current board except with the applied changes to the move-set and state array if the move were to be made. The current board will be this new board. At the end of each turn, the program checks if the state array is in a terminal state. This is done by summing up the horizontal squares, vertical squares and diagonal squares. The only possible

wins are +3 for AI or -3 for user and if no moves are left and no one has won, then the game is a tie.

Grid is the 9-board version which stores the game state in a 9-element Board array. It initializes all Board objects and the rest is similar to the Board implementation except each action is done for the number of moves that available in the Grid move-set, and then each move available in the Board move-set. To determine if the game ends, it checks if any of the boards in the game state are terminal states or if all the boards are tied.

The program is equipped to handle user error at every step of input. Each input is ran through a while loop to ensure that the user will eventually input a valid answer. The program also checks if the input move is legal and prompts the user to try again if it is not.

Example of error when entering "X" or "O":

```
C:\Users\XNGJeisenLi\Desktop>java Driver 1
You are now playing 9-Board-Tic-Tac-Toe.
Do you want to play X or O? X goes first.
123124
Invalid input. Try again.
Do you want to play X or O? X goes first.
214192501
Invalid input. Try again.
Do you want to play X or O? X goes first.
215825
Invalid input. Try again.
Do you want to play X or O? X goes first.
```

Example of error when inputting position:

```
Enter your position move:
111
Critical error. Enter a valid position move:
bob
Critical error. Enter a valid position move:

Critical error. Enter a valid position move:
99999999 9
Error. Enter a valid position move:
3333 333
Error. Enter a valid position move:
```

## How the Program Determines the Depth of Search

I wanted the program to make decisions almost instantly so I created a function that would maximize the depth of search while keeping the time of operation the same. So I created this equality:

$$Operations = \left( \frac{movesLeft}{9} \right)^{Depth}$$
$$Depth = \lfloor \log_{movesLeft/9} Operations \rfloor$$

\*The number of moves left is 81 minus the total number of moves previously made.

\*Operations is just an arbitrary constant that forces the search time to be 1.2 seconds or less.

After playing around with random numbers, I determined that the initial depth needed to be 7 which would create an almost instant search time of around 1250 milliseconds on turn 1 or 250 milliseconds on turn 2. From testing these random numbers, I found that operations needed to be  $9^7$ .

When the game goes on, less and less moves will be available so the branching factor decreases. When the branching factor decreases, it means more searches can be made at the same time, or in other words, the program can search deeper into the game state tree. This logarithms keeps the search time below a certain threshold. I chose not to use iterative deepening for two reasons. First, it would require running the same algorithm multiple times, and second, it keeps the search time at a maximum. In iterative deepening, if the time limit has not been reached, the program will run the algorithm again, but the last search can potentially exceed the allowable time by a large amount.

## How the AI Decides its Next Move

The AI uses an adversarial state-space-search algorithm called 'minimax'. The basic idea of minimax is to minimize the loss for a worst-case scenario and to maximize the gain for a best-case scenario. Tic-Tac-Toe is a fully observable, deterministic zero-sum game which means that the best move for one player is also the worst move for the opposing player. To determine whether a move is good for one player, a utility value is determined. The minimax function is called recursively until it reaches a terminal state or goal state. If this terminal state results in a win, then it returns a utility of 1. If this terminal state results in a loss, then it returns a utility of -1. If the game is tied, then it returns a utility of 0. Then, upon calling back up the recursive function, depending on whose turn it is, it is either the minimum or maximum of the branch utilities. The opposing player will always choose the path that will minimize the utility, thus if it is their turn, the utility will be the minimum of the branches.

This is for pure minimax where the maximum depth is wherever the terminal states are. In normal Tic-Tac-Toe, there is only  $9!$  transition states so a utility value is feasible. However for 9-board Tic-Tac-Toe, pure minimax is not feasible. There are  $81!$  Transition states so we need to do something clever.

I implemented depth-limited search which cuts off the search at a depth defined by my aforementioned optimum depth function. When the minimax function reaches that point, instead of returning a utility value, it returns a heuristic value which is simply an estimate of what the real utility is. However, a good heuristic must be admissible, which means it never overestimates the true cost. For my heuristic function, I counted the number of two-in-a-rows for each board. For example, if two boards have two-in-a-rows, then the heuristic value would be 2, one for each board. However, if the opposing player also has some two-in-a-rows, then the heuristic value will go down 1 for each of them. Because there are 9 boards, the maximum heuristic value would be +9. Thus, for a 3-in-a-row, the utility must be greater than 9 so I chose 10. My heuristic function is admissible because it is an assessment of the relaxed problem where there is no last move constraint of the 9-board rules. A heuristic for a relaxed problem is always admissible (pg. 105 AI book).

The minimax function returns a value indicating the utility or heuristic of that node. It does not return which node to choose. This is where the argmax comes in. I ran this minimax function on each available move and determined whichever move yielded the highest utility. If another move has the same utility, I added it to a set of moves and randomly picked from that pool of moves.

## The Effects of Alpha-Beta Pruning

To further increase the number of searches that can be made in a certain amount of time, I implemented Alpha-Beta Pruning from Wikipedia's pseudocode. The basic principle is that it is unnecessary to further explore the state nodes where a worst utility value is already found. I cut off search when this occurs and this saves tremendous searching time. Here are the results:

This is the search summary without Alpha-Beta Pruning.

```
Maximum depth of search: 7
Nodes generated: 32268688
Goal-state nodes found: 129159
Heuristic nodes generated: 28202322
Best utility or heuristic: 0
Search time(millisecons): 17836

My turn. I make the following move:
3 8
```



This is the search summary without Alpha-Beta Pruning.

```
Maximum depth of search: 7
Nodes generated: 325106
Goal-state nodes found: 9710
Heuristic nodes generated: 224464
Best utility or heuristic: 0
Search time(millisecons): 246

My turn. I make the following move:
3 7
```

We observe that the search time is significantly smaller with Alpha-Beta Pruning, by about a factor of 1/75. It also only explores 1/100 of the nodes.

### **Things I did not get to/Possible Future Changes**

Sometimes the Tic-Tac-Toe AI makes strange decisions, such as not winning the game in the shortest moves possible. This is because it randomly selects the maximum utility, but other utilities could have the same value. Perhaps, I could create another utility function that takes into account the depth of the terminal state.

Additionally, I could have included a heuristic modifier to handle pointers to tied boards in the 9-board game. As in, pointing to a tied board is generally bad for the AI. However, I am not certain if it is worth doing as it may be redundant with my current heuristic function.