CSC 242 Project 1 Writeup

Part 1: 3 x 3 board

Steps to run the 3 x 3 board

1. Go to the directory project1/part1/
2. Run the following commands:

javac Main.java

java Main

Implementation

Main.java

main() makes an instance of the problem class and then starts the tic-tac-toe game using startGame(). startGame() asks the human agent if she wants to play as “X” or “O”. If their response is “X” then the following functions are executed sequentially until goal state is found: humanMove(), aiMove() and displayBoard(). A response of “O” would run the sequence aiMove(), displayBoard() and humanMove().

Problem.java

humanMove() asks the human agent to enter the position of her move. If the spot is already taken, then the human agent is asked again, until a valid response is given. After the human agent’s correct response, activeState is set to the result of applying that response to the active state. This is implemented on line 119, shown below:

activeState = result(activeState, new Action(position));

result() first makes a copy of the State that is passed in as the first parameter and then initializes a new State (the successor state) with the contents of the passed in State. Something that I learned working on this project was that fiddling with the passed in State causes a side-effect on the original State. My very first implementation of the copy() function failed because of this very reason. I initially had the following:

board = sCopy.board

Whereas the correct implementation is:

System.arraycopy(board, 0, sCopy.board, 0, 9);

The lesson to learn here is that “=” copies the *reference* of the array rather than the value, and if we want to prevent side-effects, we must either use System.arrayCopy, or iterate over the elements of the array, using “=” to copy individual elements.

aiMove() is a two liner function that calls minimaxDecision(), which returns a decision. Then aiMove() sets the activeState to the result of applying decision on activeState. This is shown in the following two lines:

Action decision = minimaxDecision(activeState);

activeState = result(activeState, decision);

minimaxDecision() returns the most optimal action available for the AI agent. There is one conditional in this function, whose decision is dependent on the value of aiMaximize, which is set in startGame() based on the human agent’s decision to be “X” or “O”. If the human agent had chosen “X”, then aiMaximize would have been set to false, running the part of the conditional where the AI agent seeks to minimize its utility. On the other hand, if the human agent had chosen “O”, aiMaximize would be false, running the part of the conditional where the AI agent seeks to maximize its utility.

minVal() and maxVal() are the helper functions of minimaxDecision().

minimaxDecision(), minVal() and maxVal() call applicableActions() while passing in a State. applicableActions returns a HashSet of Actions that are applicable at the State that is passed in as a parameter.

Utility function returns 1 if it finds that the state passed in is won by “X”, -1 if by “O” and 0 if it’s a draw.

State.java

State.java contains the variables for the board as an array of Strings and the activePlayer (either “X” or “O”). It also contains the displayBoard function, that is called when we display the tic-tac-toe board.

Gameplay

The gameplay is very simple. First, we tell the program if we want to play as “X” or “O”, and then enter a number for each move, which specifies the location of the move. 

Figure : 3x3 board gameplay

Part 2: 9 x 9 board

Steps to run the 9 x 9 board

1. Go to the directory project1/part2/
2. Run the following commands:

javac Main.java

java Main

Implementation

Main.java

Similar to the 3 x 3 board, main() makes an instance of the problem class and then initializes the game using startGame(). startGame() asks the human agent if they want to play as “X” or “O”.

Problem.java

If their response is “X” then the following functions are executed sequentially until goal state is found: humanMove(), aiMove() and displayBoard(). A response of “O” would run the sequence aiMove(), displayBoard() and humanMove().

humanMove() first checks if the human agent has a choice as to what board she should play on. If board has the value of 0 in humanMove(), then the human agent is free to choose the board position. If board has any other value, then that value refers to the board where the human agent must to play her next move. If the human agent inputs any other board number, she would be prompted to input the value again, until the correct value is entered.

aiMove() first sets the depth of the activeState to 0. This is important because we are implementing a depth limited search with a depth of 10. So each time hMinimaxDecision() is called by aiMove(), setting the depth of activeState to zero makes sure that sure hMinimaxDecision() always searches ten levels deeper than the current state. hMinimimaxDecision() returns the best action as far as the AI agent can tell based on the heuristic, so we set the new activeState to result(activeState, hMinimaxDecision(activeState)).

result() function does four things:

1. Adds the move on the board.
2. Changes the activePlayer from “O” to “X”, or from “X” to “O”
3. Sets the cost of the successor state one more than the cost of previous state
4. Sets the depth of the successor state one more than the cost of previous state

eval() returns the sum of cost() and heuristic().

State.java

cost() is the cost of reaching that node.

heuristic() is implemented to check how many times we can find two X’s or O’s horizontally, vertically or diagonally, given that the third space is empty.

Gameplay

The gameplay for the 9 x 9 board is a little more engaging. First the human agent inputs if she wants to play as “X” or “O”. Then they first input the board which they want to play on, and then the position on that board.

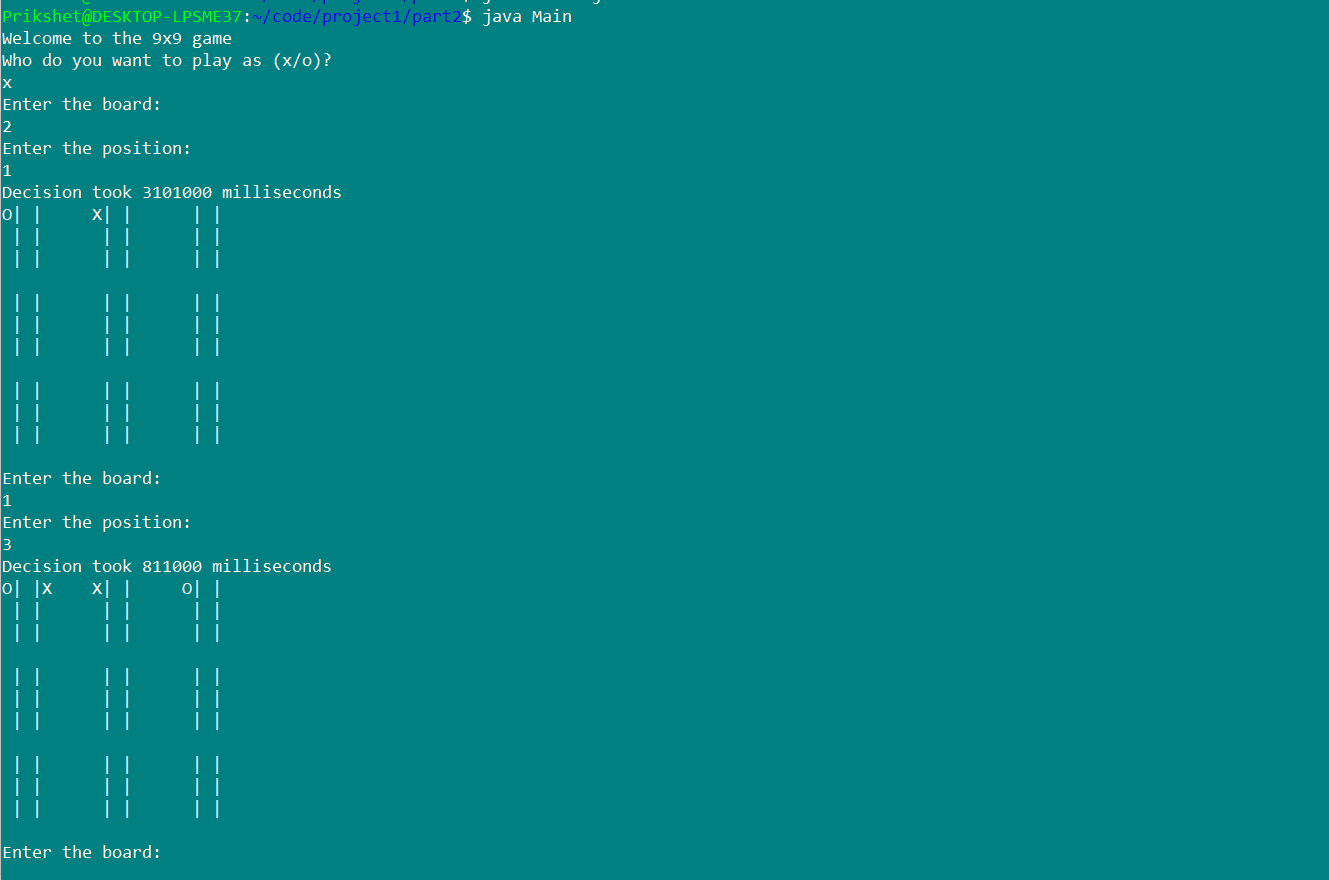


Figure : 9x9 board gameplay

Effects of Alpha Beta Pruning

Here we compare the computation times for alphaBetaDecision() versus the hMinimaxDecision()

These times were computed by sandwiching these two functions between two System.nanoTime() statements who value we store in two different variables. The difference of those two variables gives the time it took to compute the functions.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Computation times | | | | | | | | |
|  | hMinimaxDecison (ms) | | | | alphaBetaDecision (ms) | | | |
| Move | Trial 1 | Trial 2 | Trial 3 | Average | Trial 1 | Trial 2 | Trial3 | Average |
| 1 | 4E+06 | 4561000 | 4151000 | 4378000 | 4759000 | 4627000 | 4357000 | 4581000 |
| 2 | 976000 | 1024000 | 1066000 | 1022000 | 831000 | 953000 | 1435000 | 1073000 |
| 3 | 330000 | 208000 | 314000 | 284000 | 311000 | 398000 | 211000 | 306666.7 |
| 4 | 195000 | 158000 | 131000 | 161333.3 | 158000 | 185000 | 130000 | 157666.7 |
| 5 | 267000 | 250000 | 193000 | 236666.7 | 282000 | 167000 | 214000 | 221000 |
| 6 | 176000 | 280000 | 166000 | 207333.3 | 256000 | 188000 | 222000 | 222000 |
| 7 | 288000 | 218000 | 185000 | 230333.3 | 219000 | 157000 | 271000 | 215666.7 |

Result: In the 9x9 state space search, hMinimaxDecision() fared as well as alphaBetaDecision()

Problems and Proposed Solution

Although the 3 x 3 board never loses, the 9 x 9 board unfortunately tends to lose sometimes. That means that either something in our implementation is causing this problem, or that the heuristic is not effective. A way to make the AI agent play better games would be to use a better heuristic.