Sandesh Paudel

CSC 242- Project 1

Collaborator: Sidhant Ahluwalia

Project 1

I started the project by defining the class Node.java. I treated Node.java as a data structure that has the following:

1) **Int[] pos (size 9):** used to store the moves in the 3 by 3 tic tac toe board. If an index consists of 1, that corresponds to an X, and similary, -1 corresponds to O. An index with 0 corresponds to an empty spot.
2) **Int turn:** can either be 1 or -1. 1 corresponds to the next player's turn being X and -1 just refers to O.
3) **Int pathCost:** each step in tic tac toe updates the pathcost by adding 1 to it's parent's path cost.
4) **Double utility:** This is the utility value that is later to be backed up by MiniMax

The next step was to implement some basic functions for the user interface, like greetings, select X or O, explanation for how to make a move, etc. I also defined the static variable **usersign** in Main. For the purposes of this project, **usersign** is 1 if the user selects to play as X. It is -1 if the user selects O. This is used to update the board with the right sign (1 for X and -1 for O). To help with debugging the **printNode(Node n)** prints the node to error.

The **userResponse(Node n)** method does exactly this. It takes in a **currentNode** (initially empty board), and asks the user to make a move and returns the node with the updated

move from the user. It also checks if the move is valid. Essentially, this is the transition model for the user's turn.

A valid move for the regular tic tac board is just an integer 1-9, that the user can input. The **userResponse()** method takes that integer, checks if the corresponding spot is empty, and applies the move to the currentNode. Once the user makes the move, a terminal test is done to check if the game is over.

The boolean **terminal(Node n)** method to checks if it is a terminal node. There are 8 different (rows, columns and diagonal) cases to check, and the function returns true if terminal state is reached. Another similar method is **Utility(Node n).** This just checks all the positions, and returns 1 if X wins, and -1 if O wins. It returns 0 if it is a draw.

After coding the basic user side of things, the next step was to code an AI that responds to the user's move. For this, we made the **aiResponse(Node n)** method. This method does the following:

1) Calls the **makeChildren(Node n)** method, that takes the node, generates all possible children and returns them as an ArrayList of Nodes. The **makeChildren(Node n)** takes in the Node n, iterates through the empty positions in that Node, and fills them with the next player's move. It also updates **turn** (alternating), and **pathCost** (adds 1). It then adds all the possible children generated in the ArrayList.
2) For each of the children generated, calls the **MiniMax(Node n)** function to return it's utility. The **MiniMax(Node n)** function is an implementation of the Minimax algorithm as described in the texbook AIAMA page 164. If the **terminal(n)** returns true, the **Utility(n)** is returned. If not, the algorithm checks if **n.turn** is maximizing or minimizing.

For the purposes of this program, 1 (X) is the maximizing player and -1(O) is the minimizing player. The algorithm then recursively calls the **MiniMax(Node n)** function for reach of the children generated. The utility values are then backed up to the top once the algorithm is complete.

3) Once the Utility values are set for each child of **Node n**, the AI picks the child with the smallest value if the player is minimizing, and it picks the child with largest value if the player is maximizing.

The initial testing for the MiniMax caused a lot of hurdles, but once it was working, the following state was reached: (the state is printed like: 1:1 | 2:0 | 3:0|…. |9:0 | meaning that the first position has X, the second and the third positions are empty, and so on.. (-1 represents O).

```
Welcome to tic tac toe. Enter 1 to play tic tac toe. Enter 2 to play 9-board Tic tac toe.
1
Welcome to tic tac toe. Enter to play as x or o.
x

Enter an integer 1-9
1
1:1 |2:0 |3:0 |4:0 |5:0 |6:0 |7:0 |8:0 |9:0 | Turn : -1   | PathCost : 0
1:1 |2:0 |3:0 |4:0 |5:-1 |6:0 |7:0 |8:0 |9:0 | Turn : 1   | PathCost : 1
Numbers of nodes expanded : 119409
```

A counter was used to keep the track of the number of children generated, and it was 119,409 for the particular move made. The A.I then responds in the next line where the PathCost is 1.

We wrapped everything we had upto this point in a **while** loop that persists until a terminal state is reached. So, **userResponse(Node n)** and then **aiResponse(Node n)** keep

repeating until a terminal state is reached. For the record, the Ai's response is printed out as an Integer (1-9) in the **standard output.** Everything else is printed as error output.

The following is a screenshot from the first time the AI beat me:

```
1:1 , 2:-1 , 3:0 , 4:1 , 5:0 , 6:0 , 7:0 , 8:-1 , 9:0 , |   Turn : 1   |   PathCost : 2   |   Utility : 1
AI MINIMAX : 1
1:1 , 2:-1 , 3:0 , 4:1 , 5:1 , 6:0 , 7:0 , 8:-1 , 9:0 , |   Turn : -1   |   PathCost : 3   |   Utility : 1
Enter an integer 1-9
3
1:1 , 2:-1 , 3:-1 , 4:1 , 5:1 , 6:0 , 7:0 , 8:-1 , 9:0 , |   Turn : 1   |   PathCost : 3   |   Utility : 1
AI MINIMAX : 1
1:1 , 2:-1 , 3:-1 , 4:1 , 5:1 , 6:1 , 7:0 , 8:-1 , 9:0 , |   Turn : -1   |   PathCost : 4   |   Utility : 1
X wins
```

This got the regular tictactoe board to work, and the AI made intelligent moves. Most of the work for the regular tic tac toe board was later removed from the Main method and added to **threebthree()** method to make room for the 9-board tic tac toe.

The rest of the paper describes the 9-board tic tac toe:

To keep things organized, another data structure called **nBoard.java** was created. This consists of:

1. **Node[] board (size9)**: This is an array of 9 regular tic tac toe boards.
2. Other variables like **turn, pathcost, utility**, etc like the ones earlier described for the **Node** structure.

To help with organization, we also made the **ai9b9.java** that contains the functional equivalents of the methods used for the regular tictactoe board; the difference being that they handle the **nBoard.java** data structure with the 9 boards instead of 1. For example,

terminal9(nBoard n), Utility9(nBoard n), makeChildren9(nBoard n),

MiniMax9(nBoard n), userResponse9(nBoard n), aiResponse(nBoard n) are functional equivalents of the methods already described for the regular tic tac toe game.

The codes for the methods above are also very similar to the corresponding methods for the regular tictactoe board. However, here are some differences that are worth mentioning:

1. Each move is two digits that the user inputs. The first digit represents the Board number, and the second digit represents the position withint that board where the move is to be made. For example, a response of 19 means "make the move on the 9th position of the first board".
2. The response for AI will also be in the same format of two digits. It will be printed out in Standard output.
3. Each move prints out the state of all 9 boards.

After coding those functions, I ran the MiniMax for the 9 board game. It seemed to me like a never ending supply of children where created. After waiting for more than 30 mins, the **MiniMax9()** was still not complete. The following shows a shreenshot of the counter for the number of nodes exlpored:

```
350078916
350078918
350078920
350078922
350078925
350078929
350078934
350078936
350078939
350078941
```

As the picture shows millions of nodes were explored.

This made it necessary to use alpha-beta pruning to cut down the number of nodes. The **alphaBeta()** function is an implementation of psuedo code in the textbook AIMA pg. 170. In addition to the MiniMax function, it keeps track of the best moves for Max, alpha, and the best moves for Min, beta. If alpha>= beta, it prunes the unnecessary nodes.

When applied to the regular tic tac toe bored, the number of nodes gets reduced from 119,409 nodes to 6,607 nodes for the forst move:

```
Welcome to tic tac toe. Enter to play as x or o.
x

Enter an integer 1-9
1
1:1 |2:0 |3:0 |4:0 |5:0 |6:0 |7:0 |8:0 |9:0 | Turn : -1   | PathCost : 0
1:1 |2:0 |3:0 |4:0 |5:-1 |6:0 |7:0 |8:0 |9:0 | Turn : 1   | PathCost : 1
Numbers of nodes expanded : 6607
```

Although the time delay was not noticable, we replaced the **MiniMax()** function with the **alphabeta3()** function for the regular tic tac toe board. The same was also done for the 9-board tic tac toe board.

Even with the alpha beta pruning implemented, calculations for 9-board tic tac toe was taking too long. The next step was to make a heuristic evaluation function, **H(nBoard b).** The **H()** function works like this:

1) It takes a big 9-board as an input.
2) It iterates through each of the smaller 9 boards.
3) For each smaller board(Node), it checks all the rows, columns and diagonals and adds to the heuristic value (initially 0) according to the following criteria:
    a) 1 X and 2 empty spots: [+0.01]
    b) 2 X and 1 empty spot: [+0.10]

c) 3 X and 0 empty spot: [+1.00]

d) --------values are negated for O's:--------

e) 1 O and 2 empty spots: [-0.01]

f) 2 O and 1 empty spot: [-0.10]

g) 3 O and 0 empty spot: [-1.00]

4) The final value returned by the **H()** function is the average of all the values from the smaller boards.

The following is a screenshot of a gameplay when the heuristic of the **nBoard** is printed:

```
1:0 |2:0 |3:0 |4:0 |5:0 |6:0 |7:0 |8:0 |9:0 |Board: 2
1:0 |2:0 |3:0 |4:0 |5:-1 |6:0 |7:0 |8:0 |9:0 |Board: 3
1:0 |2:0 |3:0 |4:0 |5:0 |6:0 |7:0 |8:0 |9:0 |Board: 4
1:0 |2:0 |3:0 |4:0 |5:0 |6:0 |7:0 |8:0 |9:0 |Board: 5
1:0 |2:0 |3:0 |4:0 |5:0 |6:0 |7:0 |8:0 |9:0 |Board: 6
1:0 |2:0 |3:0 |4:0 |5:0 |6:0 |7:0 |8:0 |9:0 |Board: 7
1:0 |2:0 |3:0 |4:0 |5:0 |6:0 |7:0 |8:0 |9:0 |Board: 8
-- Turn : 1   | PathCost : 1   Board: 4 Pos: 5
Heuristic : -0.0022222222222222222
Numbers of nodes expanded : 325881
```

Once the heuristic function was working, it was added to the **alphabeta()** method. The **alphabeta(nBoard n)** method first checks if the board is a terminal. If it is, it returns the utility value. If it is not, it checks if the **pathCost** is greater than or equal to 7. If it is, then it returns the Heuristic value of that board. This is the Heuristic cutoff. It cuts off search after 7 moves. In practice, this made the algorithm quick and efficient. The wait for the next ai move is only a couple of seconds.

After putting everything together, the 9-board algorithm worked. The game puts **userResponse9()** and **aiResponse9()** in a while loop until a terminal is reached.

One of the improvements we could have made is using the same generic methods from the regular tic tac toe board for the 9-board game. Instead, we made equivalent functions for the 9-board and used those. The code could have been a lot more consice. Another improvement that could have been made is coding an actual GUI for this game. Debugging and testing the game was a lot harder with just printing the numbers. A GUI would have visually laid things out and made testing easier.