Abdulrahman Abdulrahman, Brian Moran, Fares Easa, Khalid Masuod

CS440

11/5/20

Please Email [aa1684@scarletmail.rutgers.edu](mailto:aa1684@scarletmail.rutgers.edu) if you have any questions!

Assignment 2:

To run the Project, compile the Execv.java file, and run the resulting executable. You will be prompted by many options for the Agent, and then a board will appear and will allow you to move. You can move by clicking on the board and moving your white piece to another part of the board. The letters on the board represent the different pieces for each player, where red is the agent, and white could be the player or agent.

Deliverables:

1. Implement the interface for the game.  The interface should display the board with all of the pieces, it should allow the adversary (human player) to input moves and update the state of the board based on the player's move (including any captures).  It should then indicate the agent's move and update the state of the board based on the agent's move.
   * 1. **Interface for the game**

[DONE]

**Graphics**

In order to display and update the graphical board that the user views, we created a new object class, DrawBoard() that took in the Board object holding the Nodes[][] that represented the board as well as details about the board. It also created a new JFrame from the java swing library for visualization. The board is created as soon as the class object is called in main (DrawBoard()) by going through each node in the board and printing a square as well either a grey circle (pit) or Capital letter in either yellow or marron (Hero, Mage, Wumpus). This is done by the Abstract Window Toolkit library (java.awt) printing squares and characters with spacing being [640px / dimension of board (3,6,9)] in both x and y. Any time the board is changed in the code, we call update() on the DrawBoard() object which re-runs the entire printing function in turn updating whatever nodes were changed in the Board class.

**Moving Pieces**

Again using the java.awt library we implemented a mouseListener override function on mouseClicked to be a function which gets the location of the mouse and sets it to selected, when selected nothing else can happen unless an available square is clicked where it performs the proper task (move, capture, etc). We allow the user and the Ai to go unlimited times in a row so the algorithm can be tested much easier.

1. In this problem it is clearly infeasible to search out the entire depth of the search tree and thus we will be bounding our search by the depth of d. Propose a metric for evaluating the bottom-level nodes of the tree in such a way that a higher metric value indicates a better state for the agent and a lower metric value indicates a better state for the adversary.
   1. Our chosen metric to evaluate the better state for an agent, is based on the number of pieces we have for that state. When we first run the algorithm on our BoardGenerator Grid, the algorithm class creates a new State object, with the current heuristic state value being the number of pieces present of the Agent minus the number of pieces of the opposing player. The white (home team) wants a higher heuristic value, with the number of white pieces being greater than away, and the black(away team) wants the lower heuristic value by having more pieces than white. Some factors that we added in to the possible moves, are when the agent simulates a movement, and the movement causes the agents piece to collide with the opponent's piece of the same type, ex) Mage collides with Mage, then instead of the heuristic just being a value determined by the numbers of pieces, the current calculated metric will either be decremented or incrementing by .5, depending on the individuals turn. By implementing this test case, the agent will be more likely to choose a situation where if they would sacrifice their own piece to win a game they would!
2. Implement minimax search.  This algorithm should take as input a max depth and conduct the minimax search out to that depth.  When evaluating states bottom- level states during minimax you should use the metric you proposed for question 2.
   1. Before generating the game board, the user is being prompted to choose a preferred search algorithm: **Option(0) is for the minimax search** and option(1) for minimax with alpha beta pruning. Once the user picks the search algorithm, the input is being stored as an integer variable (typer) to specify the type of the search. This variable alongside the created board with the specified board size, an integer specifying the playing turn, and the depth of the search which we chose a value of 3 for it.

**Algorithm j = new Algorithm(b.Board, 1 , 3, typer);**

While the minimax search makes a move on the board, it also returns the value of the best heuristic value in the command prompt, giving the best possible value. In our **Algorithm** class we implemented our minimax search by creating the function **minMax** that takes in 4 parameters: The state of the board as it stands, the depth, the turn, and an integer value that holds the depth for reference. All the possible moves that can be made by the agent are saved in a priority queue and explored based on the base heuristic value. The minMax function uses the heuristic described in deliverable 2 to assess the best move for the player. It returns the best heuristic for each piece on the board for the specified player that has the turn. That best heuristic is determined by the possible moves that a piece could perform, so, if a move based on the state of the board at every level in depth results in an opponent losing pieces that would be a good heuristic. So, the algorithm goes through the tree and returns the maximum of the minimum heuristics at every level recursively until we get to the depth which we started at. We also print out the final result of the minimax search after the best move is taken.

1. Implement alpha-beta pruning for your mini-max search.
   1. The same way we implement minMax, we created a function in **Algorithm** called **alphaBeta**, which follows the same steps as regular minimax, but in addition to stopping the search for the branch once a move that results in a minimum heuristic is found. This way it saves time and memory not taking into consideration nodes in the tree that would not give a good heuristic. We also print out the final heuristic with **alphaBeta**.
2. Specify a set of 5 potential heuristics that can be applied to this game and discuss the benefits of each of the heuristics.   Note that the heuristics do not need to be admissible.
   1. One of the possible heuristics could have been based on the average Euclidean distance from all the opponent’s positions. If you make it that it is a more lucrative heuristic to be closer to the enemy, then the AI system will be extremely aggressive, and chase my pieces around the board. If the decreasing average distance from other keys would be less optimal compared to the not, then the AI would be extremely defensive to the player.
   2. Another Heuristic would be optimal, would be the heuristic based on non-clashing keys chasing their optimal battle. So, for example, the Mage would chase the hero of the opponent around the map. This would be great because the AI would be looking for the most battles it can win. This is not as good as the heuristic we implemented, because it does not take into account moves that are going to sacrifice your own keys.
   3. A heuristic that would not be very effective, and would be easily countered by an opponent, would be, if there is an optimal move, then choose the move that isn’t the best but the value that is closest to oscillating between edges of the map. So in the case where there are two movements that can defeat an opponent, the piece would try to find the value that is closer to the opposite side of the Grid.
   4. To twist the heuristic and try and choose the most optimal solution for the pieces to move, the heuristic could be the number of enemy pieces - the number of moves that the agent will take. In this case, the algorithm will choose the moves that require the least number of moves, and that will eliminate the greatest number of enemy pieces. By choosing those turns, especially with a large depth, will end the game as quickly as possible for the agents’ victory.
   5. A last heuristic that would be admissible would be the number of pairs of pieces that a player has. When a player has a pair of different kinds of pieces, they can eliminate any number of pieces based on our game. For example, if I have a hero and a Wumpus, the Wumpus can beat a mage, our hero can defeat a Wumpus, and the hero can cancel out another hero. So, based on the number of different pairs of pieces that a player has, it could be more positive for my team, or negative for the opponent by adding them. If I have 2 pieces, then that would be 1 possible pair, if I have three unique pieces, then I have 3 possible pairs.
3. Implement the minimax algorithm with heuristics described in section 2 and apply it to this problem.
   1. [Completed] The algorithm of minimax returns the best possible heuristic for each move, and we made our code that maximizes on that factor. When calling minimax, our function saves the value with the heuristic value that is most likely to be chosen by the opponent down until the depth specified by the user. At the base level, the first call state, the algorithm runs a minimax on the possible states made by the agent. When the heuristic is returned from the minimax function, the value is saved if it is the best value. When the best value is found, the algorithm saves that state as the finalstate, in the algorithm class. When the algorithm function call is made, the object returned has a state object within it containing a grid, resulting in the move made with the best heuristic value. In our executive function, this grid is set to the new grid, and the board is updated from the resulting 2D array grid.