Caleb Cain
Cjc166
EECS 391: Intro to Artificial Intelligence
Programming Assignment One Writeup

1. **Code Design**

The code is separated in two classes, Board.java and Main.java.

The Board class is a representation of the different "boards" of the 8 puzzle. Each time a tile is moved or switched around, the board needs to be represented differently. Each board has a representation of the corresponding physical state. The board class also has methods for calculating heuristics one and two for the A* search, and a method for the evaluation function for beam search. Each board has a unique value for each of these so different representations of the board return different values for these. An important thing to know about the board is where the blank tile is at in each different state of the board, so the "index", or the "row" and "column" of the blank tile are kept track of so that I know where the blank tile is, what moves are valid for each board, and how the blank tile can be changed from each state of the board. While the main representation of each board is a String, I also represent the board as a one dimensional array of characters that is 9 elements long, and as a 3x3 array of characters.

```
private char[] states = new char[9];
private char[][] boardRep = new char[3][3];
```

This is because this is helpful in calculating the respective heuristics for each board. I use the one dimensional array to determine how many tiles are not in their correct place, which is essentially our heuristic one. Then I use the two dimensional array to help calculate the second heuristic, and determine where each tile is in the board and how far it is away from where it should be, the goal board, by using rows and columns indices of each tile. I use a method called neighbors, to get each neighbor of the current board, or in other words, the other board states that are accessible from the current one.

```
public Iterable<Board> neighbors() {
    Stack<Board> boardStack = new Stack<Board>();
    //move right
    if (blankTileColumn < 2) {
       Board boardRight = new Board(this, moveBlank(1), "Right");
       boardStack.push(boardRight);
    }

    //move left
    if (blankTileColumn > 0) {
       Board boardLeft = new Board(this, moveBlank(-1), "Left");
       boardStack.push(boardLeft);
    }

    //move up
    if (blankTileRow > 0) {
       Board boardUp = new Board(this, moveBlank(-3), "Up");
```

```
        boardStack.push(boardUp);
    }

    //move down
    if (blankTileRow < 2) {
        Board boardDown = new Board(this, moveBlank(3), "Down");
        boardStack.push(boardDown);
    }

    return boardStack;

  }
```

This function checks the up, down, right and left moves of the blank tile in the current board, and if it is possible to move the blank tile that way, meaning it is not on that edge of the puzzle, then it generates a new board with the blank tile moved in that respective direction, using the moveBlank method.

```
public String moveBlank(int offset) {
      states = stateNoSpaces.toCharArray();
      char blank = states[blankTileIndex];
      char temp = states[blankTileIndex + offset];
      char[] newStates = states;
      newStates[blankTileIndex] = temp;
      newStates[blankTileIndex + offset] = blank;

      String newState = convertToStateString(newStates);

      return newState;
   }
```

This method moves the blank tile by swapping the blank tile with the tile it should be swapping with in the puzzle. The offset is different depending on which direction the blank tile is being moved because we are only using a one dimensional array to do the movements. After this method swaps the blank tile, it returns a new String, which is used to create the board with the new state, and then this board is added it to the stack in neighbors, and is returned at the end of the method. We use the Iterable that is returned in our searches in the Main class to get the neighboring boards of each state, and we can then loop through this Iterable stack to look at each neighbor that exists and add it to the queue in our search.

The Main class is where the work and commands are implemented that can be done on the states of the Board. The board is represented as a String, so when I move tiles I just change where they are in the String to represent the different movements of the board. There are seven different commands a user can execute. The main class has methods to take user input and then process that user input to determine if it is an appropriate command and if it is it determines what command to run from there based on what is inputted. These commands are, setState, randomizeState, printState, move,

solve A-star, solve beam, and maxNodes. The details on how to use the program are listed below.

In the terminal you can compile this program with the command

>javac Main.java Board.java

Then you have two different options to run the program. If you want to run the program with command input from a text file, you enter

>java Main nameoftextfile.txt

The program will read in the text file, and the commands specified in the txt file will be executed. This has to be a .txt file, and the commands need to be listed correctly, and with a single command on each separate line. After all the commands in the text file are executed, the program will then prompt you with a ">" on a new line. It is here you can manually enter commands in to execute in the program, provided they have the correct syntax. When one wants to stop executing the program, all they need to do is type "exit" after the prompt character ">" and the program will exit. If there is no text file with commands to enter, then if the user just enters "java Main" on the command line after compiling, the program will run and immediately prompt you for a command.

COMMANDS

setState <state>

This command sets the state of the puzzle to whatever state is inputted. The parameter has to be three, three character sequences, each separated by a space. For example, if I wanted to set the state to be the goal state, the command line command would be:

>setState b12 345 678

and this would set the state of the puzzle to be that state. The state must be inputted exactly like that, the order of the characters can be switched up to represent different states of the puzzle, but there must be three characters, then a space, then another three characters, then a space, and then the last three characters. These nine characters must consist of the numbers 1-8 and the letter 'b'.

randomizeState <n>

This command starts at the goal state of the puzzle, and makes an inputted amount, n, of random moves from the goal state. This is a method of randomizing the puzzle, but when we start at the goal state and do n moves, we ensure that a solution is possible because the command to enter for this is simple

>randomizeState 5

This would make 5 random moves of the blank tile from the goal state and then tell you what the current state of the puzzle is after making those random moves, the number can be any integer but it must be a number that follows the randomizeState command.

printState
This is a rather simple command, all you do is type in the command as it is listed, and it will print back to you what the current state of the puzzle is.

move <direction>
This command is to move the blank tile in any direction you want to from the command line. The only accepted inputs are ‘right’, ‘left’, ‘up’, ‘down’. These are the four directions you can move the blank tile. However, depending on where the blank tile is in the puzzle, it may not let you move the blank tile in a certain direction. For instance, if the blank tile is in the top left corner of the puzzle, you can only move it to the right, or down. If you try to enter a command to move the tile to the left, or upward, it will not let you because that is not a valid move. After you move the blank tile a certain direction, if it is a valid move, the program will tell you what the new state of the puzzle is after that move is performed.

solve A-star <heuristic>
This command is used to solve the current puzzle state, using a-star search, with a heuristic that you provide. There are two valid heuristics to provide for this program, heuristics one and two. These are inputted as parameters, either as ‘h1’, or ‘h2’. ‘h1’ uses the heuristic of misplaced tiles, and ‘h2’ uses the heuristic of manhattan distance, which is a sum of the horizontal and vertical tiles a specific tile is away from where it should be. Part of the heuristic calculation is also the cost to get to the current state of the puzzle, which in this case is just the amount of moves to get to the puzzleThis command should be entered as
>solve a-star h1
OR
>solve a-star h2
and if the puzzle is solvable with the current value of max nodes, the program will find the solution to the puzzle and print out the steps from the starting state to the goal state and how to get to the goal. The max nodes value is set at the beginning of the program to a default value, but can be changed using the max nodes command. This is to control how many nodes a search will search through. If the search algorithm searches more nodes than the max nodes value is set to, without finding the goal state, then the search will stop and the program will tell the user that the goal solution has not been found and the number of nodes exceeded the max number of nodes to search.

solve beam <k>
This command is used to solve the current puzzle state using beam search. Beam search uses a heuristic evaluation function to determine what state to go to next. In this case, I used the manhattan distance as the evaluation function, and this determines what state to go to next in beam search. The manhattan distance is the same as the one described in a-star search for heuristic two. It is the sum of the horizontal and vertical distances that a specific tile is away from where that tile should be, in other words the distance it is away in the current state from the goal state. The parameter k, is the max number of

nodes to keep in the search space for beam search. So the user will specify how big the search space is, and if a solution of moves to the goal is found in that search space, then the solution to go from the initial state to the goal state is printed to the screen, but if the search space is exceeded without finding a solution, then it tells the user that a solution has not been found within the given search space. A correct command line input for this command would be
>solve beam 500
where the number 500 can be any number, and represents how big the search space can be in looking for a solution.

maxnodes <n>
This command specifies the max number of nodes to be considered during a search. If the number of nodes is exceeded, then the search will stop and an error message will be printed. This is set to a default value when the program is ran, but can be changed with a command line command of
>maxnodes 500
This would set the max number of nodes to be searched to 500, and this number 500 can be set to any integer value and it will change the maximum number of nodes to be searched during a search.

Every one of these commands is implemented in the Main.java class. They all manipulate the board or the search space to solve the puzzle. All the move and randomize methods just take the current state of the board and change that state in a way that was directed by the user. To implement the search method, I used a priority queue. The priority queue was defined with a unique comparator, because I needed items in the queue to be listed based on a heuristic defined by me. Below is the code where I instantiate the priority queue in A-star, and the comparators for each of those heuristics.

```
priorityQueue = new PriorityQueue<Node>(20, new Comparator<Node>() {
        @Override
        public int compare(Node node, Node t1) {
          if (heuristic == 1) {
            if ((node.moves + node.board.heuristicOne()) < (t1.board.heuristicOne() + t1.moves))
              return -1;
            if ((node.moves + node.board.heuristicOne()) > (t1.board.heuristicOne() + t1.moves))
              return 1;
            return 0;
          }
          if (heuristic == 2) {
            if ((node.moves + node.board.heuristicTwo()) < (t1.board.heuristicTwo() + t1.moves))
              return -1;
            if ((node.moves + node.board.heuristicTwo()) > (t1.board.heuristicTwo() + t1.moves))
              return 1;
            return 0;
          }
          return 0;
        }
      });
```

The comparator used depends on which heuristic is specified for use in the search, and each comparator calculates the respective heuristic function for each given board, and adds that to the boards number of moves it takes to get to it, to determine which to order first in the queue. The queue is a queue of nodes, and I defined an inner Node class in the main class. Each node has a game board associated with it, along with the previous node that it came from and the moves it takes to get to that node. I keep track of the previous node so that when the goal state is reached I can trace back the path that was taken to get to the goal state, and the number of moves contained in each node helps keep track of the number of moves it takes to get to the goal. To print the solution out, I use the following method, which makes use of an ArrayList of nodes.

```
public void printSolution() {
    ArrayList<Node> solution = new ArrayList<Node>();
    while(currentNode.previous != null) {
       solution.add(currentNode);
       currentNode = currentNode.previous;
    }

    Collections.reverse(solution);

    System.out.println("Start state is: " + solution.get(0).previous.board.getState() + "\n");

    for (int i = 0; i < solution.size(); i++) {
       Node printNode = solution.get(i);
       System.out.println("Move Number: " + printNode.moves +
            "\nCurrent State: " + printNode.board.getState() +
            "\nMove Direction: " + printNode.board.getMoveDirection());

       if (printNode.previous != null)
          System.out.println("Previous State: " + printNode.previous.board.getState() + "\n");
       else
          System.out.println("");
    }
  }
```

Because the node that is the current node is the goal state node, if we loop back through the previous nodes before that until we get to the initial node, and add each node to the ArrayList, the result will be a list of the moves taken to get to the goal state. Then when the list is reversed and each is printed, it will be the steps taken to get from the initial state to the goal state.

2. Code Correctness

For this we run through a small sample set of commands specified in the file included with this project. The commands are listed below in the order they appear in the file

setState b12 345 678
move right
move down

move left
move down
Printstate
solve a-star h2
randomizestate 250
solve a-star h2
randomizestate 250
Solve a-star h1
Randomizestate 250
Solve beam 5000
randomizestate 250
Solve beam 50
maxNodes 200
randomizestate 3
Solve a-star h1
Randomizestate 3
Solve a-star h2
Randomizestate 3
Solve beam 100
Randomizestate 3
Solve beam 5
randomizeState 10
printstate

To run these commands first need to compile the files and then run it with the text file input

>javac Main.java Board.java
>java Main example391.txt

This will run the commands in this file through the program in the order they appear in the file. The first command sets the state to be b12 345 678, then it goes through four different move commands and then calls printstate, which will print the state after the moves have been performed. Then the command solves the puzzle, it is a simple solve since we originally set the state to be the goal state and then made four moves. We use the a-star search and heuristic two to do this, and we see that the moves follow the heuristic, for each state the neighbors are calculated and then the board moves in the direction of the smallest heuristic, which brings the goal state closer and it solves the puzzle in four moves, the optimal solve. After we do this solve, we run through four iterations of trying to solve the same puzzle using both algorithms of solving, the a-star with both heuristics, and then beam search with two different search space states. When we do this, we make 250 random moves from the goal state. This ensures the the puzzle is very much randomized. Then we try to use a-star search with heuristic two to solve it, and this is able to solve the puzzle in 18 moves. However, when we try to use

heuristic one to solve this same puzzle, it exceeds the max number of nodes. This is because the heuristic two is better than heuristic one, and will provide better and faster results. Beam search also was not able to solve this very randomized puzzle with the given number of states allowed in its search space. After this we demonstrate that we can change the value of maxnodes, and then we do small randomizations of states. We make three random moves from the goal state, and we know that when we randomize a state with the same number the puzzle will always be the same because we seeded the random number generator. Both a-star algorithms using both heuristics were able to solve this in a short and optimal manner, and so was beam search when we made the search space 100. However, when we make the search space 5, there is not enough space for the solution to be found so it prints an error message. We then randomize the state again and print out the new state to demonstrate new random state and the printing function.

3. **Experiments**
    a. The fraction of solvable puzzles increases exponentially as the maxNodes limit increases, as this allows for many more nodes to be searched and added to the possible solution path.
    b. Based on the tests we have run, shown above in the sample file and performed on top of that, the heuristic number two for A* search works better than heuristic one. This is because it is a more comprehensive representation of which possible next puzzle state is closer to the goal. Rather than just saying the tiles are “wrong” per say, it tells you almost “how wrong” each tile is. It calculates how far the tiles are from their correct position, and uses that as comparison, rather than just saying that a tile is in its incorrect position.
    c. The solution length does vary but not a huge amount. The reason being that even the longest solutions aren’t necessarily that long. They are very similar between beam search and a* with the second heuristic, because they have very similar, almost the same heuristic evaluation functions. The a* using the first heuristic function has a little bit longer solutions, but not always and sometimes they are the same length.
    d. This depended on the maximum amount of nodes that were allowed to be searched and the search space specified for beam search, but the majority of generated problems that were attempted to be solved with a* search and the second heuristic were able to be solved and solutions were found. This dropped a little bit for beam search, just because it depended how big the search space was allowed to be and then the number able to be solved did drop when using a* with the first heuristic, this is because this heuristic was not as helpful as the other one for a*. It was possible that a puzzle might get stuck going back and forth and hitting the same states because the heuristic was not very specific.

4. **Discussion**
    a. The a-star search with the heuristic two, or the manhattan distance, is the better search algorithm for this puzzle. It finds shorter solutions and quicker than the other

options. It is superior in terms of time, but in terms of space it could be rivaled by beam search. Beam search allows you to specify the amount of space you are going to use in the search algorithm so it is possible that this would be an improvement on space. Beam search is relatively same in time as the a-star search with heuristic two, but it does not count for moves to get to current node as part of its heuristic which makes the a-star with heuristic two a little bit better in terms of time. The beam search can be better in terms of space though.

b. It was difficult to figure out exactly how to determine how to keep track of the exact moves it took to get to the goal state and list them as a sequence. That part was a little tough because you have to represent the count of moves individually for each board and the direction. Testing the algorithms was not too difficult once everything was represented correctly, it was just a matter of making sure the right nodes are entered in to and removed from the queue.