EECS 391
Programming Assignment #1
Andrew Hwang
ath50
Professor Michael Lewicki

Writeup

1. Code Design

I placed all of my code into one class called EightPuzzle. The class contains 11 methods excluding the main method. The main method functions in two ways; the method can either act upon an inputted text file containing a list of commands (each command on a separate line), or the method can act according to user inputs from the command line, the command to stop the program from accepting user inputs is “quit”.

Each command is passed to a method called “command”, which takes in the command line as a string and runs a switch case statement to call a specific method for the desired inputted command.

There is a method for each command from the assignment document. The setState(String state) method takes in a state in the form of a String where each row of the puzzle board is separated by a space in the String. The state is stored as a private global variable, where it can be altered by any method within the class, but since all methods are private methods, the state’s modification is allowed only within the class, and this prevents the state becoming an unpredicted variable. The row and column of the blank tile is calculated from the inputted state and stored as private global variables, the reason for this is to make later calculations easier.

```
int i = 0;
while (tempState.charAt(i) != 'b') {
    tempCol++;
    if (tempCol == 3) {
        tempRow++;
        tempCol = 0;
    }
    i++;
}
```

```
int r = rand.nextInt(100) + 1;
// depending on the randon number, move up, down
int result = 0;
if (r <= 25)
    result = EightPuzzle.move("left");
else if (r > 25 && r <= 50)
    result = EightPuzzle.move("down");
else if (r > 50 && r <= 75)
    result = EightPuzzle.move("right");
else if (r > 75 && r <= 100)
    result = EightPuzzle.move("up");
else
    System.out.println("Something went wrong!");

if (result < 0)
    i--;
```

The randomizeState(int n) method takes in the number of random moves inputted as n. A random number generator was used to produce a random number, and depending on where the number falls between 1 and 100 the blank tile will move either left, right, up, or down. Also, the method makes sures to not waste a move if that move is not possible, such as the blank

tile being on the top row of the board and the next random move is “up”, which is not possible from that position, so the method will skip that iteration and will continue to generate random moves until a possible move has been generated.

The printState() method does not take in a parameter, and outputs the current state, as well as, the state represented as a puzzle board to the command line.

The move(String direction) method takes in a specified direction as a String, which specifies either to move the blank tile left, right, up, or down. The method will return a “-1” if the specified direction is not possible for the blank tile’s current position, this will let other methods know whether to do certain functionalities with appropriate moves, such as the randomizeState method, where it will skip an invalid movement. Here is where specifying the row and columns of the blank tile comes in, so to specify if a move is invalid, a condition is placed to check if the the row is at the top or bottom or if the column is at the right or left. Swaps are made for the blank tile and the position of where the blank tile will be moved to. The swap was made with temporary variables.

```
case "up":
    if (row > 0) {
        replaceRow -= 1;
        replaceChar = board[replaceRow].charAt(replaceCol);
        board[replaceRow] = board[replaceRow].replace(replaceChar, 'b');
        board[row] = board[row].replace('b', replaceChar);
        row -= 1;
    } else {
        return -1;
    }
```

The solveAStar(String heuristics) method takes in a String that specifies the either to use “h1” heuristic (the number of misplaced tiles) or to use “h2” heuristic (the sum of the distances of the tiles from their goal positions). This method uses a HashMap to store all unique states and their path cost from the start state, parent node, and the direction that led to the state. The method mainly uses a priority queue that prioritizes by the sum of a state’s heuristic value and the path cost from the start state (f(n) = g(n) + h(n)). The sum (f(n)) and heuristic value (h(n)) are calculated with helper methods; getF(String state, int pathCost, String heuristic), getH1(String state), and getH2(String state). The main loop is a while loop have conditions to check if the queue size has reached the max node limit and if the current node being expanded is the goal state. If the queue size reaches the max node limit, then the search ends and outputs an error message. If the goal state is the next node being expanded then the search has reached the goal state and is done. Inside the while loop is a for loop where every possible state from the current node is produced, which is checked from the move method return value. When a state is revisited the path cost is compared to the path cost that is in the HashMap to see if the current path was a better path than the previous path, and if so, it is updated in the map and the queue. When the search is done, the program outputs the number of nodes used for the search, the number of tile moves to get to the goal state

from the start state and the moves needed to obtain the solution as a sequence of moves.

The getF(String state, int pathCost, String heuristic) method was metioned in the solveAStar search method, and it takes in a state to calculate the total cost (f(n) = g(n) + h(n)) for, the path cost (g(n)) to the state, and the heuristic (h(n)) to use to calculate the total cost (f(n)). Depending on which heuristic is inputted the sum will call the getH1 method or the getH2 method.

The getH1(String state) method takes in a state and calculates the number of misplaced tiles. The method does this by iterating through the state and checking the numeric value of the tile and comparing it to the index of the state, except for the blank tile, which is compared to the zeroth index. The sum of the misplaced tiles is returned at the end of the method.

```
char tile = tempState.charAt(i);
if (tile != 'b') {
    if (i != Character.getNumericValue(tile))
        h1++;
} else {
    if (i != 0)
        h1++;
}
```

The getH2(String state) method takes in a state and calculates the sum of the distances of the tiles from their goal positions. The method does this by iterating through the state and calculating the difference of the numeric value of the current tile and the index of the current tile, except for the blank tile, which subtracts zero from the index of the blank tile. The distance of the tile from its goal position is the sum of the difference divided by 3 (calculates the number of rows the tile is from the goal position) and the difference modulo by 3 (calculates the number of columns the tile is from the goal position); written in math tems: h2 = diff/3 + diff%3. The sum of the distances is returned at the end of the method.

```
diff = (tile != 'b') ? Math.abs(i - Character.getNumericValue(tile)) : i - 0;
h2 += diff/3 + diff%3; // number of rows from target + number of cols from target
```

The solveBeam(int k) method takes in an integer that represents the number of states that the search can keep track of. The search is very similar to the A* search that was implemented, but if the queue size specified by k has reached its limit, the search will not end, but instead replace the lowest prioritized state in the queue with the newly visited state if the new state has a higher priority value than the lowest priority state. The evaluation function used to calculate the priority queue uses the path cost value and the “h2” heuristic value, which was used for A* search. The output is the same as A* search.

The maxNodes(int n) method takes in an integer that represents the maximum number of nodes the A* search is allowed to use for its priority queue. The method sets the global variable MAXNODES to “n”, so that it may be used by the A* search method.

2. **Code Correctness**

	Number of Nodes Used (Max Nodes at 1000)		**Path from start to goal/ Time for search to complete(milliseconds)**			
Initial State	**A*(h1)**	**A*(h2)**	**A*(h1)**	**A*(h2)**	**Beam(10)**	**Beam(20)**
312 7b5 468	11	9	start -> left -> down -> right -> up -> left -> up -> goal (6 moves) 6 msecs	start -> left -> down -> right -> up -> left -> up -> goal (6 moves) 6 msecs	start -> left -> down -> right -> up -> left -> up -> goal (6 moves) 6 msecs	start -> left -> down -> right -> up -> left -> up -> goal (6 moves) 6 msecs
15b 374 682	29	20	start -> down -> down -> left -> up -> up -> right -> down -> left -> up -> left -> goal (10 moves) 14 msecs	start -> down -> down -> left -> up -> up -> right -> down -> left -> up -> left -> goal (10 moves) 10 msecs	start -> left -> down -> right -> up -> left -> left -> down -> right -> right -> up -> left -> left -> down -> right -> right -> down -> left -> up -> left -> up -> right -> right -> down -> left -> left -> up -> goal (26 moves) 66 msecs	start -> down -> down -> left -> up -> up -> right -> down -> left -> up -> left -> goal (10 moves) 12 msecs
836 175 b24	Exceeds max node limit (would take 20210 nodes)	Exceeds max node limit (would take 7162 nodes)	(28 moves) 1902 msecs	(28 moves) 500 msecs	(70 moves) *too many moves to put in the table 150 msecs	(232 moves) *too many moves to put in the table 290 msecs

The simplest scramble (312 7b5 468) shows that all the searches produced the same solution and completed with similar. The only difference is the amount of nodes

used by the searches, and the A* search using the h2 heuristic used the least amount of nodes.

The next more difficult scramble (15b 374 682) shows how the A*(h2) search uses fewer nodes than the A*(h1) search and completed the search the fastest. The beam search produced different results when the number of states it can hold increased. The solution path and amount of time to complete the search were both larger for the 10 state beam search than the 20 state beam search.

The most difficult scramble (836 175 b24) made both A* searches run out of nodes to use for their searches, but without the limit this scramble showed a larger difference of number of nodes used between the two A* searches, but the h2 heuristic search still used the fewest number of nodes. Both A* searches produced the same number of moves to get to the solution, but the A*(h1) search was much slower than the A*(h2) search. The two beam searches produced very interesting results. The beam search with fewer states produced a solution with less moves, and to completed faster.

3. Experiments

	Number of Nodes Used (Max Nodes = 1000)		Number of moves to Solution/ Time for search to Complete(milliseconds)			
Initial State	**A*(h1)**	**A*(h2)**	**A*(h1)**	**A*(h2)**	**Beam(10)**	**Beam(20)**
312 645 b78	2	2	2 moves 1 msecs	2 moves 1 msecs	2 moves 1 msecs	2 moves 1 msecs
b12 365 748	10	8	6 moves 7 msecs	6 moves 3 msecs	6 moves 4 msecs	6 moves 6 msecs
312 7b5 468	11	9	6 moves 6 msecs	6 moves 6 msecs	6 moves 6 msecs	6 moves 6 msecs
15b 374 682	29	20	10 moves 14 msecs	10 moves 10 msecs	26 moves 66 msecs	10 moves 12 msecs
b32 841 657	775	332	18 moves 106 msecs	18 moves 69 msecs	20 moves 45 msecs	34 moves 136 msecs
4b2 735 816	Exceeds max node limit (would take 1363 nodes)	315	19 moves 79 msecs	19 moves 66 msecs	47 moves 207 msecs	43 moves 164 msecs
568 317 42b	Exceeds max node limit (would take 3997 nodes)	924	22 moves 180 msecs	22 moves 119 msecs	134 moves 182 msecs	38 moves 143 msecs
836 175 b24	Exceeds max node limit (would take 20210 nodes)	Exceeds max node limit (would take 7162 nodes)	28 moves 1902 msecs	28 moves 500 msecs	70 moves 150 msecs	232 moves 290 msecs

a. The fraction of solvable puzzles from initial states varies with how many randomized moves are performed on the state, and so the more random moves the more likely the A* searches start to exceed the maxNodes limits. Around 150 random moves will get the A* searches to use more than 1000 nodes. The beam searches start to produce large solution paths, but do not get stuck that often.
b. Heuristic h2 is the better heuristic for A* search, since every move will produce a different heuristic value versus the h1 heuristic, which only calculates the number of misplaced tiles that may not change when a move is made.
c. The A* searches produce the shortest solutions out of the three searches, and the beam search produces much larger solutions depending on the number of k states it's allowed to use.
d. All problems were solvable, except for the times when the maxNodes limit was reached. Even with very difficult scrambles like the "836 175 b24" scramble the searches will produce a solution, and it may depend on what the maxNodes limit is set to and it may produce very long solutions similar to what the beam searches produced.

4. **Discussion**
    a. The A*(h2) search is the best suited for simple to semi-difficult scrambles, but the beam search performs on par to the A*(h2) search when scrambles become really difficult. The A*(h2) and A*(h1) searches produce the shortest solutions compared to the beam search. The beam search overall is superior in terms of time and space, since it's able to produce a solution for all difficulties of scrambles with only 10 or 20 nodes available, and produces them at similar times as the A* searches.
    b. The difficulties of implementing the searches were keeping track of the amount of nodes the searches were allowed to use and making sure that revisited states were updated properly if a better path was found. Testing the algorithms produced some very interesting results, particularly with the beam searches, since some results would be in favor of less states available than more states.