CSDS 391 P1 Due: 10/11/2022
Mason Leung

Code Design:

I had 3 major classes: Puzzle.java, Node.java/BeamNode.java, and Search.java.

Puzzle Class:

The foundation of my code is the puzzle class where all the methods that alter and return the puzzle state. Within my puzzle class, I had 5 private instance variables:

```
private char[][] puzzle;
//{{'b',1,2},{3,4,5},{6,7,8}}
private static int[] indexB;
//e.g. [0,0] initial position
private static int[][] solutionIndex;
 //Used to calculate Manhattan distance, where the first row represents the row index of the desired value and the second row represents the column index of the desired value
private int maxNodes;
private String direction;
```

Puzzle stores the two dimensional, three by three, array. **IndexB** keeps track of where the blank space is at all times. For example, in the solved state, indexB = [0,0], the first index representing the row index, and the second index representing the column index. **SolutionIndex** is a helper variable that allows me to calculate the Manhattan distance by providing quick access to the index of the goal position. **MaxNodes** keeps track of the maximum number of nodes to be considered during a search. **Direction** helps keep track of the previous move that the puzzle made.

Notable Methods within the Puzzle Class:

printState():

- This method returns a concatenated form of the 2d puzzle into a string.

setIndex():

- This method finds ‘b’-blank space and registers those indices into indexB.

setState():

- This method takes a concatenated string of the puzzle and transforms it into a 2d array, which is pretty much the inverse of printState().

move(direction):

- This method takes in 4 options of string: up, down, left, and right. There are 5 helper methods assisting this method: swap [the basis for all moving methods], moveUp, moveDown, moveRight, and moveLeft. The algorithm helps update indexB when it moves the blank space in the indicated direction.

maxNodes(number):

- This method helps set the number of maxNodes to be considered during the search.

heuristics(method):

- This method takes in either “h1” or “h2”, representing the two different heuristic functions: Manhattan distance or the number of misplaced tiles. The two helper methods: heuristicOne and heuristicTwo calculate the respective values for their functions.

randomizeState(number):

- This function generates a random puzzle state by moving the blank space randomly “number” number of times.

addSuccessors():

- This function returns an arraylist of the neighboring puzzle states each representing a possible move: up, down, left and right.

Node/BeamNode Class:

Both Node and BeamNode classes are practically equivalent classes with beamnode class keeping track of one less instance variable than its counterpart. BeamNode has no reason to keep track of the cost, as the heuristic variable is local beam search’s evaluation function. Node assists in representing the puzzle state as an independent node for A-star search. BeamNode represents the puzzle in local beam search.

```
Node parentState;
Puzzle puzzleState;
String direction;
int heuristic; //h(n)
int level; //g(n)
int cost; //f(n)
public Node(Puzzle puzzle, int level, int heuristic, Node parent, String direction){
       parentState = parent;
       puzzleState = puzzle;
       cost = level + heuristic;
       this.level = level;
       this.heuristic = heuristic;
       this.direction = direction;
   }
```

ParentState helps keep track of how the puzzle states are connected to each other. **Direction** provides the same information as the direction variable in the puzzle class. **Level** in the node class represents the g(n) function in A-star search as it represents how many moves made so far. **Heuristic** keeps track of the heuristic cost of the current puzzle state, either in manhattan distance or the number of misplaced tiles. **Cost [f(n)]** is the addition of level [g(n)] and heuristic [h(n)].

```
public String printNode(){
       return puzzleState.printState();
   }

   public void printPath(Node root){
       if(root == null){
           return;
       }
       printPath(root.parentState);
       System.out.print("Move " + root.level + " " + root.puzzleState.getDirection() + ": "+
root.printNode());
```

```
        System.out.println();
    }
```

The two methods within Node/BeamNode class:

printPath():

- The method recursively calls itself to print out the parent node of the parent node of the… etc. of the goal state as well as including the moves that it took to get there. This calls a helper method printNode()

printNode():

- The method calls on the puzzle method printState() to return the string concatenation of the puzzle that is represented by the node.

Search Class:

This class contains the two search algorithms: A-Star Search and Local Beam Search

A-Star Search():

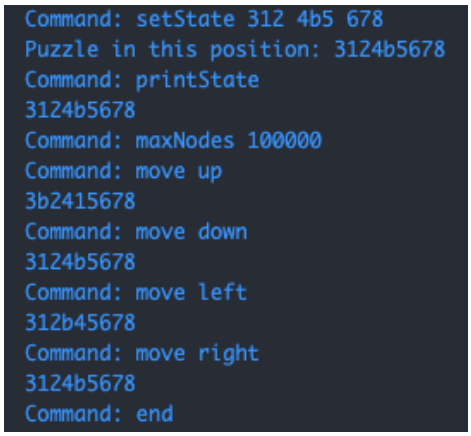
- This algorithm is implemented using a priority queue that utilizes a comparator class in order to sort the nodes within the priority queue. The comparator compares nodes using the cost variable within the node class that is the evaluation function combining the number of moves taken with the heuristic function. The algorithm takes in a puzzle state and a method (either h1 or h2 to utilize the respective heuristic function). The first randomized puzzle state is added into the priority queue. We only check if we found the goal state after we pop the puzzle state from the queue. Next, we pop the initial puzzle state in order to generate its successors into an arraylist that are added into the priority queue. This cycle of popping and adding into the priority queue occurs until either the goal state is found (which should happen because the algorithm is complete) or the algorithm has considered more than the maxNodes allowed.

Local Beam Search():

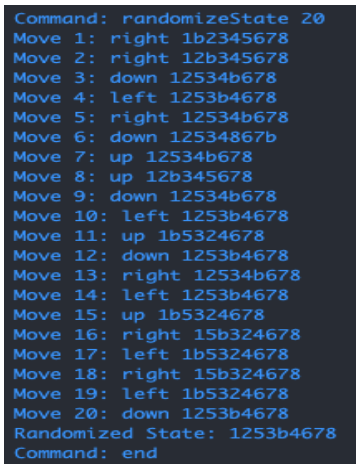
- This algorithm is implemented using both a priority queue and an arraylist. The arraylist helps keep track of k number of paths. The priority queue helps each path expand using the evaluation function. The priority queue is assisted with another comparator class that helps evaluate BeamNodes using the heuristic variable in the BeamNode class that is calculated using the function heuristicTwo(). This function will eventually evaluate to zero when the goal state is reached because the sum of the distance of the tiles from their goal positions will then be zero. Similar to the A-Star search, the initial puzzle state is added and popped from the priority queue. Then I generate the successors and add them to the priority queue. Using the priority queue, we generate the k best states from the initial possible moves. Then those states are added into an arraylist of size k. Using a while loop, we traverse through the arraylist one by one and generate their respective successors into their own priority queue for that specific path. Then the best path is popped from the priority queue and it replaces the parent state that was being held within the array list. Again, the state is checked for the goal state once it is popped from the priority queue. This process is repeated similar to A-Star search until the goal state is found or the algorithm has considered more than the maxNodes allowed.

Code Correctness:

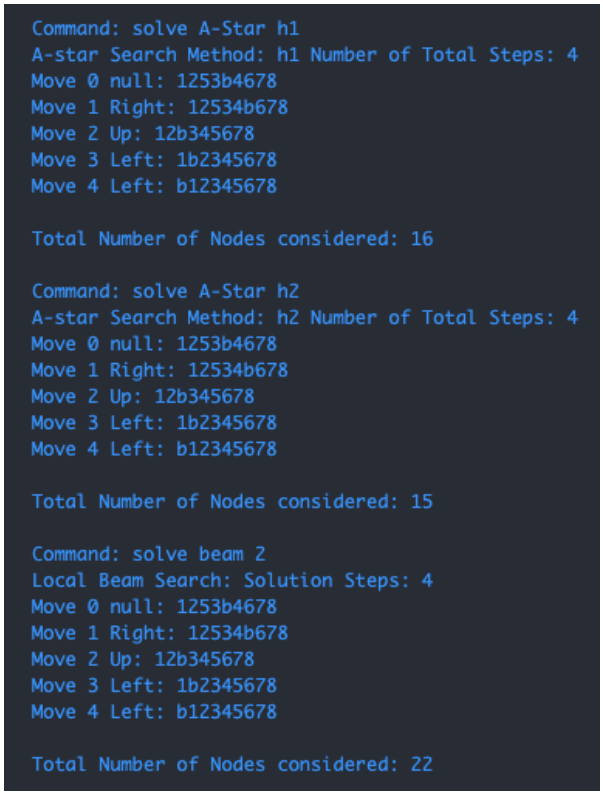
Example 1: Standard functions (not search algorithms)

Input: TestCode.txt	Output
setState 312 4b5 678 printState maxNodes 100000 move up move down move left move right	
As you can see, the move method is shifting the blank space; up, down, left, and right. The set method also initially sets the puzzle state, instead of declaring a new puzzle state and setting it into the goal state.	

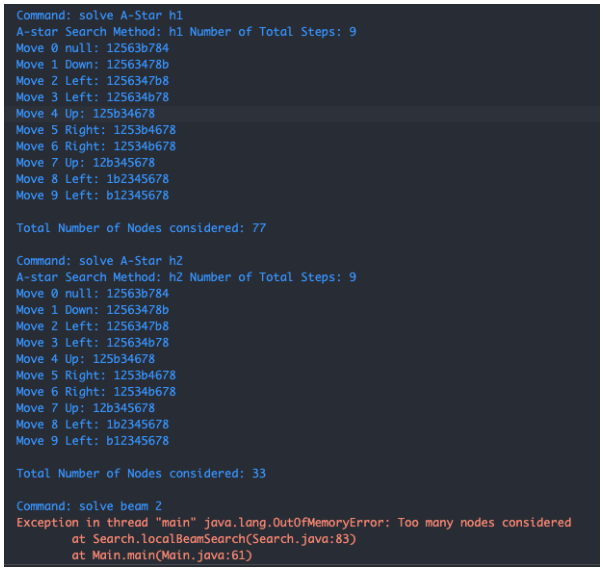
Example 2: Randomize state

Input: TestCode.txt	Output
randomizeState 20	
The algorithm starts with the goal state and performs randomized moves 20 times, printing out each step and then finally the final state.	

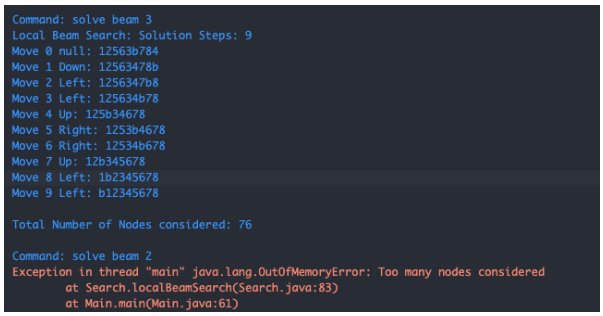
Example 3: Given the randomized state

Input: TestCode.txt	Output:
randomizeState 20 maxNodes 100000 solve A-Star h1 solve A-Star h2 solve beam 2	
Using the same randomized state as above, I performed A star search using h1, h2 and local beam search using k = 2. As you can see, the number of nodes considered by h2 is less than h1 which are both less than local beam search at 22.	

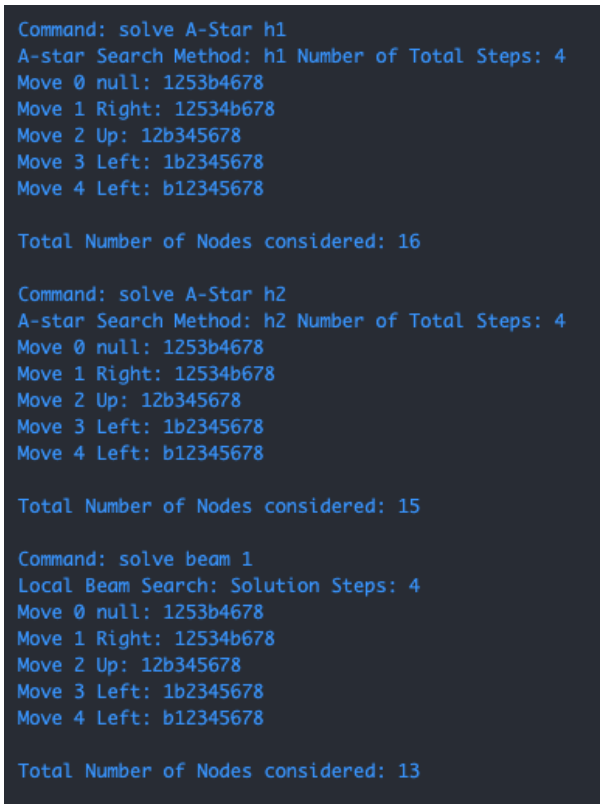
Example 4: Randomized State of 35 moves

Input: TestCode.txt	Output:
RandomizedState: = 12563b784 maxNodes 10000 randomizeState 35 solve A-Star h1 solve A-Star h2 solve beam 2	
Given a max number of nodes of 10000, using both heuristics, both solved the puzzle in 9 step, considering 77 nodes and 33 nodes respectively (h1, h2). However, the local beam search was unable to solve the puzzle due to the maxNodes constraint, when k = 2.	

Example 5: Same randomized state as example 4

Input TestCode.txt	Output:
Same as last example, with the addition of local beam search with k = 3: randomizeState 35 end solve A-Star h1 solve A-Star h2 solve beam 3 solve beam 2	
Interestingly, local beam search, when k = 3, is able to solve the puzzle that k =2 could not.	

Example 6: Randomized State 10

Input: TestCode.txt	Output:
maxNodes 100 randomizeState 10 solve A-Star h1 solve A-Star h2 solve beam 1 Randomized State = 1253b4678	
All the algorithms work properly in this case, and all algorithms had the most optimized path as well, with local beam search considering the least number of nodes.	

Experiments:

a. How does the fraction of solvable puzzles from random initial states vary with the maxNodes limit?
   i. Based on my experiments, solvable puzzles depend on two factors: number of maxNodes and number of moves during the randomizing process. The number of maxNodes is proportional to the number of solvable puzzles. So as maxNodes increase, so does the number of solvable puzzles. As the randomizing process increases, the heuristic also tends to increase causing the puzzle to become more scrambled, which requires more search nodes. However, this plateaus after a certain number. The number of solvable puzzles increases exponentially when you initially increase the number of max nodes. After a certain point, it also plateaus as the percentage of puzzles slowly increases from 90 to 100.

b. For A* search, which heuristic is better?
   i. Manhattan distance has been performing better than the number of misplaced tiles. Both heuristics always provide the most optimal path. However, heuristic 2

(Manhattan distance) typically searches through less nodes to find the shortest path when compared to heuristic 1. Thus, heuristic 2 applied to A-star is able to solve some puzzles that heuristic 1 is unable to solve when the number of maxNodes is limited.

c. How does the solution length vary across the three search methods?
    i. A-star search always has the same paths irrespective of the heuristic that it is using. It always provides the most optimal solution. However, local beam search is not always optimal. Increasing k will improve the solution length to almost that of A-star search. The downside is that it uses up a lot more memory as it has to search through many more nodes.
d. For each of the three search methods, what fraction of your generated problems were solvable?
    i. So for A-star search, 100% of the generated problems are solvable depending on whether there was a limit of maxNodes. However, for local beam search, with a minimal k < 3, the fraction of generated problems is less than 9/10. However, after k >=3, local beam search is able to solve all the generated problems, if maxNodes is infinite.

Discussion:

a. Based on your experiments, which search algorithm is better suited for this problem? Which finds shorter solutions? Which algorithm seems superior in terms of time and space?
    i. A-star search is the most suited for this problem of the two, and heuristic 2 being the better of the two. As it finds the shorter length solutions compared to local beam search and the same length solutions as A-star search with heuristic 1. It is also optimal in terms of time and space as it guarantees an optimal solution in the shortest number of nodes depending on what heuristic is being used. Because heuristic 2 is better than heuristic 1 in that aspect, it can be generalized in this situation where the three types of search are being compared, that A-star search with the Manhattan distance heuristic is most suited for the 8-puzzle..
b. Discuss any other observations you made, such as the difficulty of implementing and testing each of these algorithms.
    i. Local Beam Search was much harder to implement than aStarSearch because it was difficult for me to visualize how local beam search would actually function and store the data for each state until I realized that it was similar to breadth first search in how it expanded each level at the same time for all k paths. After I realized that it was similar to breadth first search, it was much easier for me to decide on which data structures I wanted to use, in this case, an array list paired with a priority queue. It is also hard to test local beam search because there is always an extra parameter to tune compared to aStarSearch where you only have to choose the heuristic function.