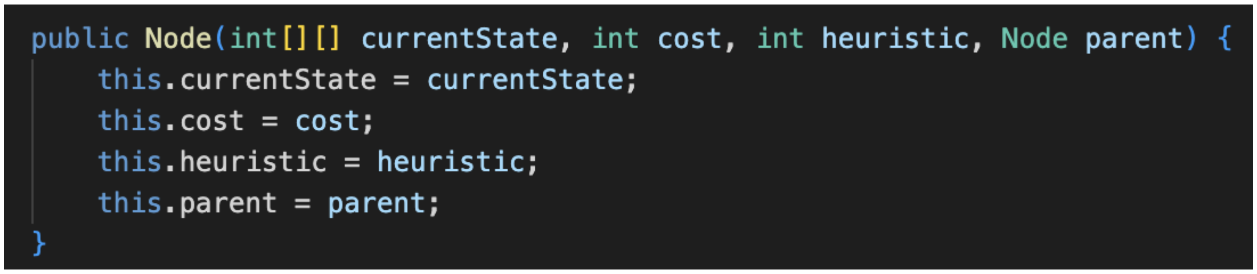
Code Design

Introduction

For my project, the code is separated into two different files, where one file is called: EightPuzzle.java and the other Main.java. Main.java is a fairly simple design, where I have the main method, and it is essentially the driver code for EightPuzzle.java. It has three different text files it is reading, with three different instances of EightPuzzle in order to test A* search with h1 heuristic function, second to test A* search with h2 heuristic function, and third to test local beam search.

Global Variable Explanations

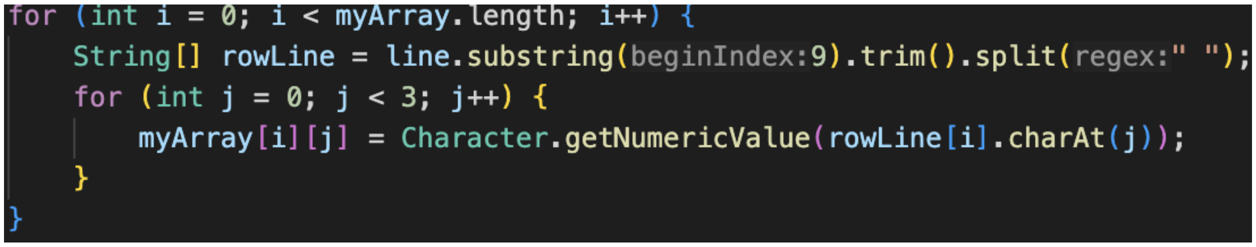
In the file EightPuzzle.java, there are few points to highlight, which starts from what kind of global variables which I decided to use. Main point to highlight from the global variable is Node startNode. My original idea was to have an arrayList, which had all of the states of the method. However, as I continued testing, I realized that my code slowed down significantly due to this, so I decided to get rid of this. So, I have the startNode, and in here, I need to mention that I made an internal node Class which has, the state of the node, cost, heuristic value, and the parent node of that node. Initially, I was thinking of constantly changing the values in the 2D array in order to update the states. However, with this method, it had few errors, which were: we could not keep track of the heuristic values and costs as we kept updating. Even though this is possible, but, constant updates of heuristic value and costs would make this method very inconvenient and very difficult especially while I was trying to code for a pathway to reach the goal. So, it made me realize that I need to store the state, the cost, heuristicValue and parentNode (which I will talk about later while I am talking about path solutions).



```java
public Node(int[][] currentState, int cost, int heuristic, Node parent) {
    this.currentState = currentState;
    this.cost = cost;
    this.heuristic = heuristic;
    this.parent = parent;
}
```

ReadCommand Explanation

In order to read the txt files, I created a method called commandReading(), where it takes in an input of text files. I put a while loop in order to fully read all of the lines, and included conditional statements to intake all of the commands. Most of the conditional statements are self-explanatory, but one issue I dealt with was the setState() section, because within:



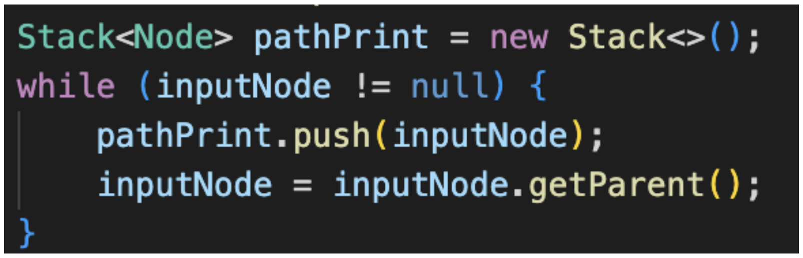
```
for (int i = 0; i < myArray.length; i++) {
    String[] rowLine = line.substring(beginIndex:9).trim().split(regex:" ");
    for (int j = 0; j < 3; j++) {
        myArray[i][j] = Character.getNumericValue(rowLine[i].charAt(j));
    }
}
```

,
The second line I wrote separated the input by spaces, and because my inputs looked like: 012 345 678, so each index had each row. However, by sorting the rowLine[i] by characters, I was able to represent the input in 2D array fashion.

Algorithm Explanation

General Helper Methods

All three of the algorithms share general helper functions called: generateBeamSuccessors(), generateH2Successors(), generateSuccessors(), copyState(), printPathSolution(), and getMove(). To explain the functionalities, generateSuccessors() generates a maximum of four possible states corresponding to ("up", "down", "left", and "right"). Originally, all three of the algorithms used the same method, which was generateSuccessors(). However, I did not make any changes corresponding to the heuristics, so that made me create a successors() method for each of the searches. copyState() copies the state, printPathSolution() prints out the path, and getMove() shows what move is necessary in order to get from parent node's state → current state.
To start off with generateSuccessors() In here, in order to make updated states corresponding to each move, I realized that I should not update the original state, such as: if we updated the original state to up state, then try to get it for down as well, then we would be making an updated in the updated state, so it would not be correct, so I implemented copyState() method.
printPathSolution() is also corresponded with getMove() as well, and the most notable thing should be



```
Stack<Node> pathPrint = new Stack<>();
while (inputNode != null) {
    pathPrint.push(inputNode);
    inputNode = inputNode.getParent();
}
```

The usage of a stack, where the stack initially pushes the inputNode, and the parent nodes of the inputNode are pushed as well, making the first(initial state) to be at the very top. Then, from this stack, we initially start with the starting state, however, by comparing the current state with the parent node's state, which is the purpose of the getMove() method, all of the moves are stored in an arrayList, and this is printed out.

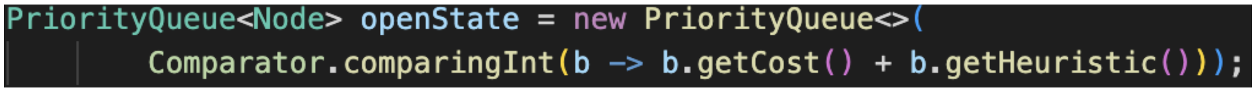
Additionally, I created helper methods which calculated the heuristic values separate from the search algorithm itself, because this would look more organized, and it would be more readable code.

One more thing to add is that, if it reaches the nodeLimit, or the maximum node, then it throws LimitExceededExceptions, saying it reached maximum Node.

A*Search Explanation

For A* Search, I defined heuristic functions as number of misplaced tiles + cost, and the second heuristic function to be distance away from goal state plus the cost.

For my A* Search, although the two functions are in two different methods, but because there are no difference (in terms of structure) other than heuristic calculation, so in here, I will assume the two as the same methods when talking about their structure/design.



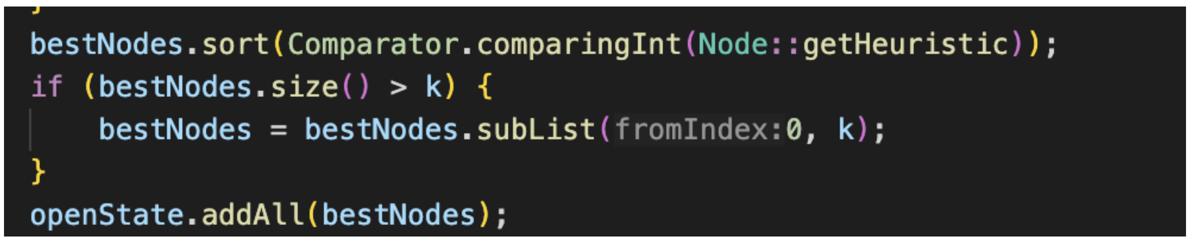
```
PriorityQueue<Node> openState = new PriorityQueue<>(
        Comparator.comparingInt(b -> b.getCost() + b.getHeuristic()));
```

My A* algorithm follows the pseudocode from the textbook and ideas which we talked about during lectures. I use a priority queue to contain all of the nodes, and these nodes are compared through (cost + heuristicValue) of these nodes within the priority queue, where the lowest value has the priority. In the middle, it uses a hash set in order to check for repeated states. While I was working on it, I thought maybe I can edit/make the code look cleaner by changing the type of the hash set to Node instead of string. The main issue with this was the when Node is used, it compares the destinations instead of the actual values. These stored states which are already visited, and store newly updated states from the successor() method. And then, these nodes are compared with the goal state until it is exactly the same.

Local Beam Search Explanation

For my local beam search, I follow the pseudocode of hill climbing combined with ideas mentioned from the lecture. In my local beam search, the evaluation function is the combination of h1 + h2 from A* search, (without including the costs).

From my code, the most important part is:



```
bestNodes.sort(Comparator.comparingInt(Node::getHeuristic));
if (bestNodes.size() > k) {
    bestNodes = bestNodes.subList(fromIndex:0, k);
}
openState.addAll(bestNodes);
```

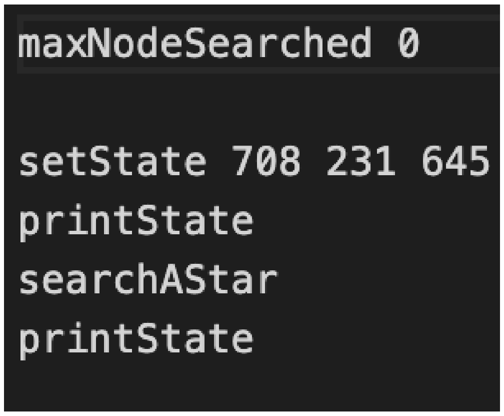
,
And the part that requires a bit more explanation is the subList method, which I defined as (0,k), separates the list from the index of zero to k. From here, bestNodes contains all of the successor (possible states). Then I used comparator,comparingInt to sort this by lowest heuristic value, which is the highest priority for my code. Then, I added the subList to the priority queue again, and this process is repeated until the goal is reached.

Other Method Explanation

Other methods, which are move(), randomizeState(), and maxNodeSearched() were pretty self-explanatory, where move takes in the input of direction, and moves it in that direction, randomizeState() randomly sets the state based on random directions which it sets, and maxNodeSearched() takes in the input of nodeLimit.

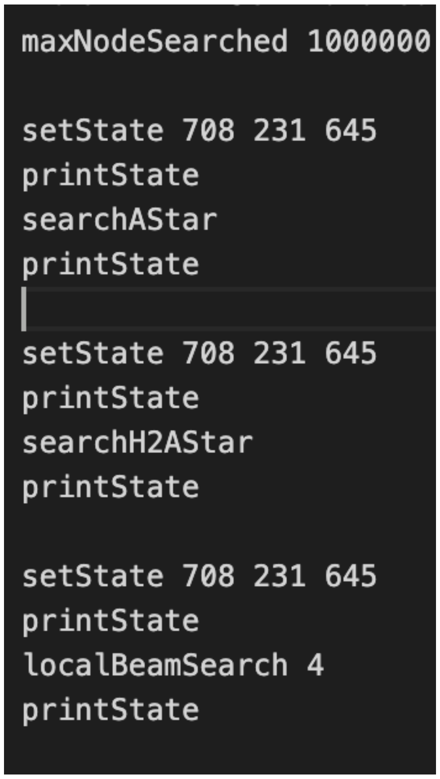
Code Correctness

maxNodes Correctness



```
maxNodeSearched 0

setState 708 231 645
printState
searchAStar
printState
```

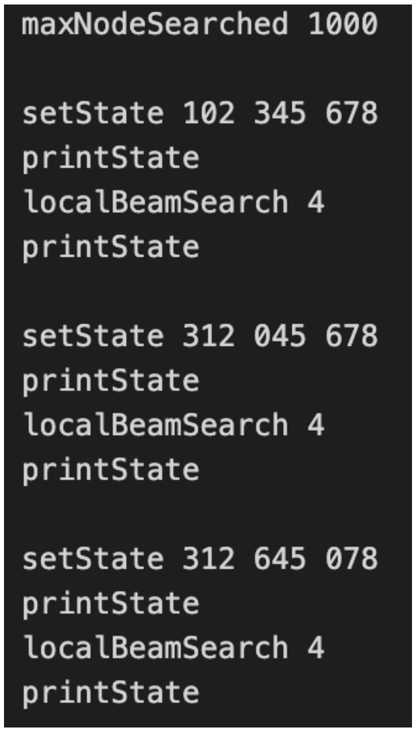
Currently in the txt file, the maxNodeSearched is set to 0, meaning, in the searchAStar, it should throw LimitExceedException based on our command Line, and when I looked at the output, it successfully throws LimitExceedException



```
maxNodeSearched 1000000

setState 708 231 645
printState
searchAStar
printState

setState 708 231 645
printState
searchH2AStar
printState

setState 708 231 645
printState
localBeamSearch 4
printState
```

When this is in, maxNodeSearched became 1000000, though, it did not throw any exceptions, and it successfully outputted the pathway.

BeamSearch Correctness

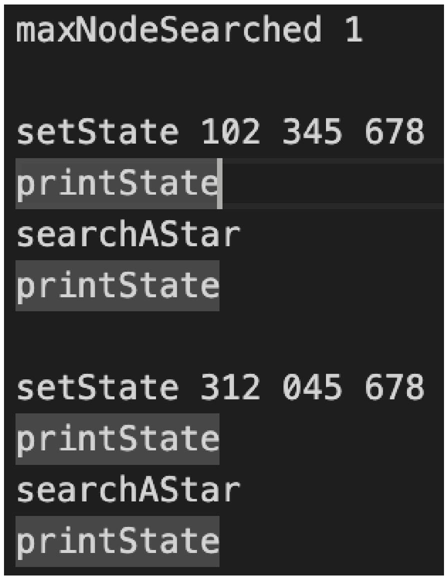


```
maxNodeSearched 1000

setState 102 345 678
printState
localBeamSearch 4
printState

setState 312 045 678
printState
localBeamSearch 4
printState

setState 312 645 078
printState
localBeamSearch 4
printState
```

For localBeam Search, we set our initial state as 102 345 678, and here, based on my output, this successfully outputs the direction and how many nodes which it created. The output is: Goal state found, generated total of: 1 number of nodes.
The directions to the solution are: Starting State, Left. Within our maxNode, all of the methods have successfully outputted the correct directions, so it is correct.

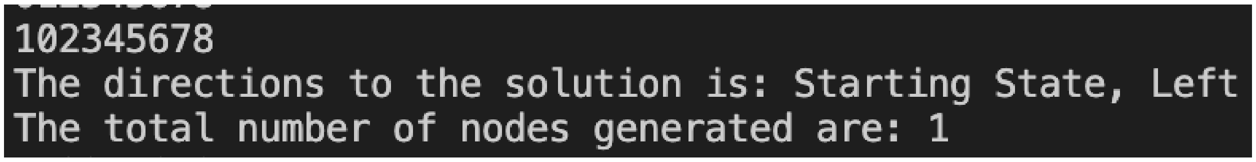
How it fails is already tested through the testing of maxNodeSearched.

A* Search Correctness



```
maxNodeSearched 1

setState 102 345 678
printState
searchAStar
printState

setState 312 045 678
printState
searchAStar
printState
```

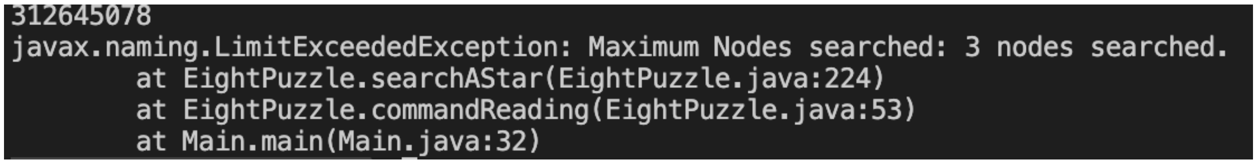
For the first state: 102 345 678, because this only requires one move, meaning it should only generate one node to store. Our output we got was:



```
102345678
The directions to the solution is: Starting State, Left
The total number of nodes generated are: 1
```

, and based on our output, we can say that our A Star search for h1 heuristic function is working successfully.

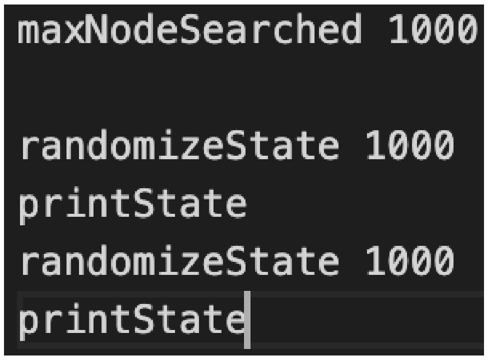
In the second state, our state is 312 045 678. In this state, because it requires two moves, meaning this should fail, as it creates too many nodes based on our maxNodesSearched(), and our output we got was:



```
312645078
javax.naming.LimitExceededException: Maximum Nodes searched: 3 nodes searched.
        at EightPuzzle.searchAStar(EightPuzzle.java:224)
        at EightPuzzle.commandReading(EightPuzzle.java:53)
        at Main.main(Main.java:32)
```

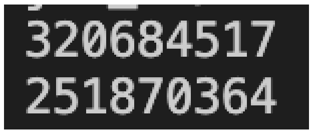
, and because the output outputs LimitExceededException(), an exception we decided to throw, so as a result, we can say that our A Star for h1 heuristic function is working properly.

randomizeStateCorrectness



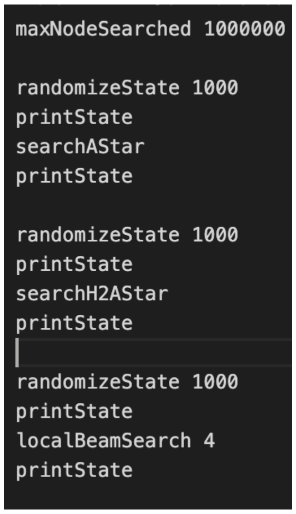
```
maxNodeSearched 1000

randomizeState 1000
printState
randomizeState 1000
printState
```

Because randomizeState() uses move() method within its method, so functionality of randomizeState() would mean that move() method would also be functional. In order to test this, we first randomized the state, and then printed it out. We notice that after randomizeState 1000, it shows a completely random state, which shows that it worked, as seen from the output values.

Output: 

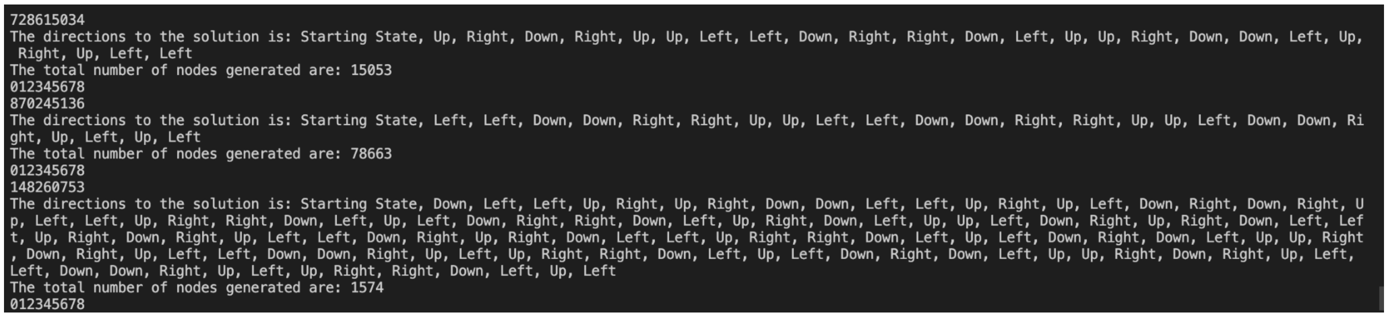
```
320684517
251870364
```

DemoResults:



```
maxNodeSearched 1000000

randomizeState 1000
printState
searchAStar
printState

randomizeState 1000
printState
searchH2AStar
printState

randomizeState 1000
printState
localBeamSearch 4
printState
```

When we randomized the states and then asked it to show the direction:



```
728615034
The directions to the solution is: Starting State, Up, Right, Down, Right, Up, Up, Left, Left, Down, Right, Right, Down, Left, Up, Up, Right, Down, Down, Left, Up,
 Right, Up, Left, Left
The total number of nodes generated are: 15053
012345678
870245136
The directions to the solution is: Starting State, Left, Left, Down, Down, Right, Right, Up, Up, Left, Left, Down, Down, Right, Right, Up, Up, Left, Down, Down, Ri
ght, Up, Left, Up, Left
The total number of nodes generated are: 78663
012345678
148260753
The directions to the solution is: Starting State, Down, Left, Left, Up, Right, Up, Right, Down, Down, Left, Left, Up, Right, Up, Left, Down, Right, Down, Right, U
p, Left, Left, Up, Right, Right, Down, Left, Up, Left, Down, Right, Right, Down, Left, Up, Right, Down, Left, Up, Up, Left, Down, Right, Up, Right, Down, Left, Lef
t, Up, Right, Down, Right, Up, Left, Left, Down, Right, Up, Right, Down, Left, Left, Up, Right, Right, Down, Left, Up, Left, Down, Right, Down, Left, Up, Up, Right
, Down, Right, Up, Left, Left, Down, Down, Right, Up, Left, Up, Right, Right, Down, Left, Up, Left, Down, Right, Down, Left, Up, Up, Right, Down, Right, Up, Left,
Left, Down, Down, Right, Up, Left, Up, Right, Right, Down, Left, Up, Left
The total number of nodes generated are: 1574
012345678
```

The output successfully outputs the correct directions, and the goal state in the end, so it shows that our methods are working.

Experiments

For my experiment, the experiment was run on, Tester.java file, where each node was run 100 times, and it generated random solvable puzzles each time.

a. For my heuristic functions and from my code, for A*(h1), they spiked from 0 to 10500. Then it stayed a bit constant from 1750-2500. It kept exponentially growing until 6000 nodes, where it fluctuated heavily, and this is suspected due to the state differences from randomizeState(). For my heuristic functions of A*(h2), there is a growing trend of exponential growth as maximum nodes increase. For local beam search, it spiked from 1-2500 nodes. However, from 2500 nodes, it fairly stayed constant.

Function	A*(h1)	A*(h2)	Beam Search
Number of nodes for each maxNode for randomly generated states			
1	0	0	0
500	10550	3402	5024
1000	16187	7427	9426
1250	18840	15655	30344
1750	33717	19135	51429
2500	31895	33262	88858
3250	43530	51167	94930
5000	59282	66774	82897

5250	68811	70078	103126
6000	101394	125929	89112
7500	115161	175340	78097
10000	80223	252672	87806
15000	178954	320107	90199
20000	192685	385129	111689
22500	119419	439656	80860

b. H1 generated 1,070,648 nodes, whereas h2 generated 1,9695,433 nodes, so for my methods, H1 is better since h1 generated less nodes.

c. The solution length varied, and it seemed A*(h2) generated the most nodes, Beam Search, and then A*(h1)

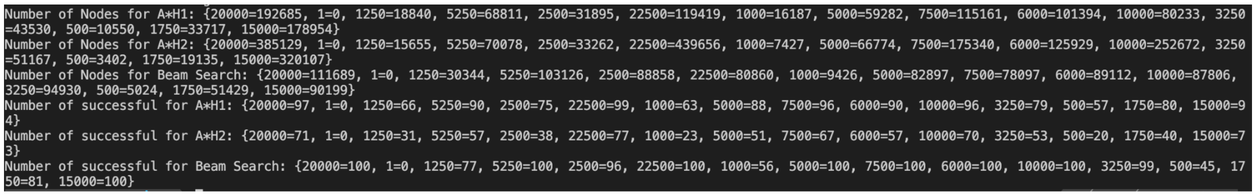
d. A*(h1): solved 1170 problems
A*(h2): solved 731 problems
Beam Search: solved 1271 problems. Most of the problems which the three of the algorithms could not solve was the initial states, as maxNodes were too low for them to solve this.

e.

Function	A*(H1)	A*(H2)	Beam Search
Number of times the search was solved done			
1	0	0	0
500	57	20	45
1000	63	23	56
1250	66	31	77
1750	80	40	81
2500	75	38	96
3250	79	53	99
5000	88	51	100
5250	90	57	100

6000	90	57	100
7500	96	67	100
10000	96	70	100
15000	94	73	100
20000	97	71	100
22500	99	77	100

f.

g. 

```
Number of Nodes for A*H1: {20000=192685, 1=0, 1250=18840, 5250=68811, 2500=31895, 22500=119419, 1000=16187, 5000=59282, 7500=115161, 6000=101394, 10000=80233, 3250
=43530, 500=10550, 1750=33717, 15000=178954}
Number of Nodes for A*H2: {20000=385129, 1=0, 1250=15655, 5250=70078, 2500=33262, 22500=439656, 1000=7427, 5000=66774, 7500=175340, 6000=125929, 10000=252672, 3250
=51167, 500=3402, 1750=19135, 15000=320107}
Number of Nodes for Beam Search: {20000=111689, 1=0, 1250=30344, 5250=103126, 2500=88858, 22500=80860, 1000=9426, 5000=82897, 7500=78097, 6000=89112, 10000=87806,
3250=94930, 500=5024, 1750=51429, 15000=90199}
Number of successful for A*H1: {20000=97, 1=0, 1250=66, 5250=90, 2500=75, 22500=99, 1000=63, 5000=88, 7500=96, 6000=90, 10000=96, 3250=79, 500=57, 1750=80, 15000=9
4}
Number of successful for A*H2: {20000=71, 1=0, 1250=31, 5250=57, 2500=38, 22500=77, 1000=23, 5000=51, 7500=67, 6000=57, 10000=70, 3250=53, 500=20, 1750=40, 15000=7
3}
Number of successful for Beam Search: {20000=100, 1=0, 1250=77, 5250=100, 2500=96, 22500=100, 1000=56, 5000=100, 7500=100, 6000=100, 10000=100, 3250=99, 500=45, 17
50=81, 15000=100}
```

(Data which I received)

Discussion

1.
   a. For this problem, Beam search is better suited because Beam Search can solve more. According to the tests, A* search found shorter paths, and this was clearly noticeable as we started to tests, run experiments on it, and while testing code correctness as we were clearly able to see A* search had much shorter paths. This makes sense, as A* search intakes the cost with the heuristic function, so it will find shorter paths. In general however, beam search is superior in terms of time and space.
   b. For me, personally, if I had more time, I would have loved to test beamSearch() where the k values change every time. Currently, I made the k to equal to 4, but I am certain that as k's value increases, it will help the solve the problem at an efficient manner. Also, I found this project very interesting, as it helped me learn and develop as a programmer. Additionally, I think for me, the most difficult part was definitely the beamSearch(), and currently, I am thinking that my h2 heuristic function of A star search has some errors to it.