Writeup Done by Andrew Gross (acg109)

Section 1: Code Design

I chose to implement the project in java as 3 classes: Board.java, State.java, and Node.java. The command file is named Commands.txt . Board.java represents the problem as a whole, and includes the solving methods as well as auxiliary methods for those. State.java represents a single state, and includes methods to represent the state and get the heuristic values at that state. I chose to internally represent the state as an int array of size 9, since that allowed for quick access and straightforward implementation. Node.java represents a node in the search tree generated by solveA_star() and solveBeam(). As such it has variables storing the Node's state, parent, the action used to arrive at that node, and the h(n) value, or path cost of that Node.

Board.java: This class represents the "board" or an instance of the 8-puzzle problem. It includes the search methods for A* and beam search, as well as maxNodes(int n), move(State s, Direction d), randomizeState(int n), setState(State s), and printState(). It also includes other auxiliary methods that help or test other methods, including setHeuristicFunction(int heuristic), testComparator(), findBlankIndex(State s), getLegalMoves(State s), checkMove(State s, Direction d), expand(Node n), and traceNodePath(Node n, boolean solved).

I implemented A* search in the method solveA_star(int heuristic), which takes an int to set the heuristic, and returns either the solution node, or the last node checked before the maxNodes limit was reached. This method first checks if the heuristic input is valid, and if so, it sets the current heuristic as such. It then initializes the priority queue representing the frontier with a Node constructed with the current state & initializes the reached HashMap. This method then follows the A* search procedure of looking for an optimal solution by utilizing the priority queue and the evaluation function. I chose to have the cost of each move be equal to 1, meaning that the evaluation function of a given node is the h(n) plus the number of moves from the root. Before returning a node, this method calls traceNodePath(Node n, boolean solved) with the corresponding parameters, which prints the sequence of moves leading to the final node to the console.

I implemented local beam search in the method solveBeam(int k), which takes in an int k, and chooses the best k states during the search to proceed down the tree with. It returns either the solution node or the last node checked before the maxNodes limit was reached. Although it may not be the most efficient way to implement it, I implemented this algorithm using a PriorityQueue used to sort states according to their h(n) values, a HashMap for reached states, and a HashSet which contains the first k Nodes from the frontier PriorityQueue and is used to expand them. How my implementation works is as follows: The reached HashMap and PriorityQueue are initialized as in solveA_star, and then the algorithm polls a Node from the frontier PriorityQueue k times, putting each into the nodesToExpand HashSet. The frontier is then cleared (so as not to become A* search) and the nodesToExpand HashSet are expanded. At each iteration, it checks whether any expanded node is the solution or if the maxNodes limit has been reached, and returns the current node if either situation is the case. Before returning a node, this method

also calls traceNodePath(Node n, boolean solved) with the corresponding parameters, which again prints the sequence of moves leading to the final node to the console.

State.java: This class represents a single state. It has a constructor that takes in a String (i.e. “b12 345 678” and initializes the int array tiles[] as such. It also has a constructor that takes in another State, and simply creates a new State instance by copying every value in that State’s tiles[] array. This class includes the methods that calculate the value for the h1 and h2 heuristics using the tiles[] array to do so. Finally, it includes a toString() method that returns the contents of the tiles[] array in String format.

Node.java: This class represents a single node in the search tree. It has some instance variables representing the State, g(n) value/path cost, h(n) value, parent node, and action taken to arrive at that node. Additionally, it has a constructor that creates an instance of the class using the input params, and has getPathCost() and getState() which merely return the corresponding variables.

Main.java: This class simply creates an instance of Board.java, and reads the command file, manipulating that instance as the file instructs it to.

I followed standard Java code organization and convention in my project, putting all instance variables at the top, constructors next, useful methods after that, and finally the getters / setters. Board.java also has the testComparator() method, Comparator class, and enum for Directions below the getters / setters.

Section 2: Code Correctness

Commands.txt: this is the text file I am reading my commands for this section of the writeup from.

A* search algorithm (solveA_star()) : With the following input, my implementation of A* finds the solution only expanding 11 nodes, and with 4 total moves. maxNodes is set to 15,000 by default.

Input Commands:

```
randomizeState 10
printState
solve A-star h2
solve A-star h1
```

Output for **h2 AND h1**:

```
NOW TRACING NODE PATH:

SOLUTION:
b 1 2
3 4 5
6 7 8

LEFT
1 b 2
3 4 5
6 7 8

UP
1 4 2
3 b 5
6 7 8

LEFT
1 4 2
3 5 b
6 7 8

UP
1 4 2
3 5 8
6 7 b
STARTING STATE SHOWN ABOVE. MOVES MADE ARE SHOWN IN BETWEEN STATES.
Done in 4 moves.
Number of nodes expanded: 11
```

For n = 10, the change in heuristic did not matter. However, as n increases, there becomes more of a noticeable difference, and the function may start to fail.
Let's try n = 20.
Input:

```
randomizeState 20
printState
solve A-star h2
solve A-star h1
```

Output for **h2**:

```
NOW TRACING NODE PATH:

SOLUTION:
b 1 2
3 4 5
6 7 8

LEFT
1 b 2
3 4 5
6 7 8

UP
1 4 2
3 b 5
6 7 8

LEFT
1 4 2
3 5 b
6 7 8

UP
1 4 2
3 5 8
6 7 b

RIGHT
1 4 2
3 5 8
6 b 7

RIGHT
1 4 2
3 5 8
6 b 7

DOWN
1 4 2
3 b 8
6 5 7
STARTING STATE SHOWN ABOVE. MOVES MADE ARE SHOWN IN BETWEEN STATES.
Done in 6 moves.
Number of nodes expanded: 65
```

Output for **h1:**

```
NOW TRACING NODE PATH:

SOLUTION:
b 1 2
3 4 5
6 7 8

LEFT
1 b 2
3 4 5
6 7 8

UP
1 4 2
3 b 5
6 7 8

LEFT
1 4 2
3 5 b
6 7 8

UP
1 4 2
3 5 8
6 7 b

RIGHT
1 4 2
3 5 8
6 b 7

DOWN
1 4 2
3 b 8
6 5 7
STARTING STATE SHOWN ABOVE. MOVES MADE ARE SHOWN IN BETWEEN STATES.
Done in 6 moves.
Number of nodes expanded: 26
```

In this instance, h2 actually expanded more nodes, but both arrived at the same solution.
If we increase n to something absurd, like 250, A* search will fail and the maxNodes will be hit.
Input:

```
randomizeState 250
solve A-star h2
```

Output:

```
NOW TRACING NODE PATH:

The maximum number of nodes allowed in the search has been exceeded and the most recent node has been returned.
Number of nodes expanded: 15000
```

Local Beam Search:
If k is low, this search method is much more likely to fail, since it only expands the first k nodes. However, if the user chooses a k that is reasonable, for example 10, local beam search is effective in memory usage and time efficiency.
You can see that my implementation works and fails in this way with the following examples:

Input:

```
randomizeState 20
solve beam 10
```

Output:

```
NOW TRACING NODE PATH:

SOLUTION:
b 1 2
3 4 5
6 7 8

LEFT
1 b 2
3 4 5
6 7 8

UP
1 4 2
3 b 5
6 7 8

LEFT
1 4 2
3 5 b
6 7 8

UP
1 4 2
3 5 8
6 7 b

RIGHT
1 4 2
3 5 8
6 b 7

DOWN
```

```
DOWN
1 4 2
3 b 8
6 5 7
STARTING STATE SHOWN ABOVE. MOVES MADE ARE SHOWN IN BETWEEN STATES.
Done in 6 moves.
Number of nodes expanded: 46
```

And now to show it failing with k = 2:
Input:

```
randomizeState 20
solve beam 2
```

Output:

```
NOW TRACING NODE PATH:

The maximum number of nodes allowed in the search has been exceeded. Current Node has been returned.
Number of nodes expanded: 15000
```

Section 3: Experiments

For this section of the writeup, I collected data on each search method, varying heuristics, maxNodes limits, and increasing n values for randomizeState. I used this data to create the following tables:

Number of Nodes expanded & Number of Moves from Solution by N value (F indicates node limit was reached)

MaxNodes	N value	A*(h1)	A*(h2)	Local Beam Search(k=10)
15k	10	11, 4	11, 4	29, 4
	20	26, 6	65, 6	52, 6
	30	42, 8	90, 8	67, 8
	40	165, 10	198, 10	92, 10

	50	**26, 6**	**65, 6**	**52, 6**
	60	**386, 10**	**428, 10**	**F**
	70	**F**	**7256, 16**	**152, 16**
	80	**13868, 16**	**4806, 16**	**F**
	90	**3901, 14**	**863, 14**	**F**
	100	**4405, 14**	**270, 14**	**F**

Failures and Avg. nodes expanded from 1-100 N-values for increasing maxNodes limits:

maxNodes = 100:

```
6 5 7

A* with h1 failed 55 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 30.68888888888889
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

A* with h2 failed 59 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 40.26829268292683
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

Beam Search with k=10 failed 100 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 0.0
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0
```

maxNodes = 1,000:

```
A* with h1 failed 40 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 105.81666666666666
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

A* with h2 failed 32 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 152.76470588235293
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

Beam Search with k=10 failed 45 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 270.0
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

Process finished with exit code 0
```

maxNodes = 10,000:

```
A* with h1 failed 24 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 852.3421052631579
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

A* with h2 failed 0 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 1049.86
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

Beam Search with k=10 failed 44 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 270.0
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0
```

maxNodes = 100,000:

```
A* with h1 failed 2 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 7671.836734693878
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

A* with h2 failed 0 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 1049.86
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

Beam Search with k=10 failed 46 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 270.0
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0
```

maxNodes = 1,000,000:

```
A* with h1 failed 0 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 11250.7
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

A* with h2 failed 0 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 1049.86
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0

Beam Search with k=10 failed 43 times out of 100 tries, n from 1-100
Average nodes expanded for numMoves <= 5: 270.0
Average nodes expanded for 5 < numMoves <= 10: 0.0
Average nodes expanded for 10 < numMoves <= 15: 0.0
Average nodes expanded for 15 < numMoves: 0.0
```

Answers to Section 3 Questions:

A. How does the fraction of solvable puzzles from random initial states vary with the maxNodes limit?

a. As seen in the above screenshots, it is clear that as maxNodes increases, there are fewer and fewer failures for each search method. Notably, with a low maxNodes(100), A* search fails over 50% of the time irrespective of the heuristic used, and local beam search fails 100% of the time, at least for k=10. However, at the other extreme(maxNodes = 1 million), neither heuristic for A* fails once, and beam search with k=10 only fails 43% of the time. As maxNodes increases, we also the difference between h1 and h2, as well as beam search. Since my chosen k-value was only 10, beam search failed much more than the other search methods, and only improved slightly with the increase in node limit. This also reflects the algorithm design, since beam search does not guarantee a solution, and is used in situations where memory and time need to be optimized. Save the first iteration with maxNodes=100, h2 consistently failed less than h2,

demonstrating that it is a better measure of the distance from the goal state.

B. For A* search, which heuristic is better?

a. It is clear from my experiment results that h2 is more efficient, especially as the node limit increases, and as the distance from the goal state increases. My table as well as my screenshots of tests with varying node limits shows that h2 often outperforms h1 in nodes expanded and number of failures. In the table, you can see that in situations where the distance from the goal was higher, h2 expanded less nodes, and even found solutions where h1 failed. The best examples of this are n=80 and n=100, when h2 expanded 9,062 and 4,135 less nodes respectively. However, the difference between heuristics is much less pronounced at lower distances from the goal, with lower node limits, and with lower n-values.

C. How does the solution length vary across the three search methods?

a. Surprisingly, all of the tests I did showed that when each search method succeeded, they found the same length solution. This is perhaps because my data set wasn't incredibly large.

D. For each of the three search methods, what fraction of your generated problems were solvable?

a. The screenshots with increasing node limits demonstrate what fraction of my generated problems were solvable. For higher node limits, there were fewer and fewer failures. h2 performed the best overall, with less failures all around than h1 and beam search with k=10, h1 came in second, and it is hard to judge beam search, since the k value determines much of its performance.

Section 4: Discussion

a. Based on my experiments, I would say that A* search with the second heuristic is best suited for this kind of problem. The extra accuracy of the refined heuristic aids greatly, and since the 8-puzzle isn't an insanely complex problem to solve, the time/space usage is definitely stomacheable. Each algorithm found the same length of solutions when they succeeded. In terms of time and space, it is

difficult to judge the best algorithm. Perhaps the most optimal solution for time/space would be beam search with an optimized choice for k, but it isn't easy to choose the perfect k value for a given instance of the problem. For very low n-values, A* with h1 actually performed better than h2, but h2 soon surpassed it as the problems got more complex. As such, I would conclude that the optimal algorithm is entirely dependent on the nature of the problem instance, but the most consistent has to be A* search with h2.

b. Other observations I made during the course of the project are that it was easier to implement the beam search algorithm correctly, and that I could improve my statistics/data-collection skills when it came to the experimental section. Due to the nature of randomizeState, some high n-values were outliers in that they actually were closer to the goal than some lower n-values. This is because I chose not to weigh against choosing moves you've already chosen, so there is always the possibility of moving back and forth between states. However, given enough data, this quirk would be ironed out a little bit.