Ben Kurzion
bxk389

Project 1 Write Up

Code Design:

The code begins with the runnable Main function. This function is in charge of setting global variables and reading in user input from a text file. I used the built in java class called BufferedReader to read line by line for a text input. I use the first couple of letters of each text line to recognize the function that the line wants to execute.

The next thing I did was create my State class which is how I represent each configuration of the 8 puzzle. To prepare for writing the search algorithms, I made sure that this class had a 2-D array of integers representing the configuration for that puzzle state, a pointer for the parent so I can recurse back at the end of the search, a String storing the direction moved from the parent to this configuration, two pointers for the row and column of the blank tile so don't need to find them again and again every time, and finally two pointers storing g(n) and f(n) = g(n) + h(n) where h(n) is the given heuristic.

```
private int[][] configuration;
private State parent;
private String parentDirection; //direction to move from parent state to this
state.
private int blankRow;
private int blankCol;
private int movesFromStart; //g(n)
private int cost; //f(n) = g(n) + h(n)
```

The rest of the methods in the State class are getters and setters for the various fields as well as a stringRepresentation function that I discuss later.

For the first method **setState**, I ran through the given input and placed the given values into my global variable currentState. Since my representation of each state is a 2-D array of integers, I decided to represent the blank tile as 0 and the other 8 tiles as their respective numbers. So when the setState algorithm sees the 'b' character in the input string, it treats it as 0.

```
if(Character.isLetter(state.charAt(cursor))){
```

Furthermore, I check if the inputted state is a valid state. I do this with the helper method checkValidStates. This helper convers the 2-D array into a 1-D array and then counts the number of times that a smaller number is placed higher than a larger number. If the number of these inversions is even, then the state is solvable.

If not, the method will throw a NumberFormatException. Only valid states should be considered by the search algorithms.

```
throw new NumberFormatException("This is not a solvable state. Please only set
the puzzle to a solvable state.");
```

The opposite is true for the method **printState**. When this method encounters the 0 in the 2-D array of integers, it will interpret it as a 'b' character.

```
if(currentState.getConfiguration()[i][j] == 0){
   sb.append("b");
}
```

The **move method** is quite repetitive. It checks whether or not the blank tile can be moved up, down, left, and right based on its current position. If the method input asks for a direction that cannot be fulfilled, nothing happens. I figured that there is no need to throw an error because for the most part, this method will be used internally and I don't need to constantly handle errors. Also, this move method directly affects the currentState global variable. This is the same variable that the previous two methods operate on.

For randomizeState, I used the previous method move. I generated a random number from 1 to 4 which correlates to up down left right respectively. If the current configuration does not allow the puzzle to be moved in that direction, then I regenerate the random number until it can be moved. I then use the move method to change the global variable currentState accordingly. This repeats as many times as specified by the input.

```
rand = 1 + (int)(Math.random() * 4);
```

To prepare for writing A* search, I began by writing the two heuristic functions h1 and h2. Both take in a State parameter and return the calculated heuristic value. The h1 heuristic needs to compare the given state with a solved 8 puzzle to determine which values are out of place. To do this, I created a global variable named finalState. To implement finalState, I effectively created a hashtable with a 2-D array. For example, for tile number 3, the correct row and column are 1 and 0 respectively. So finalState[3] = {1,0}. This way, I can quickly compare the given row and column with the correct row and column in the final state without a long lookup time.

```
finalState = new State(new int[][] {{0,0}, {0,1}, {0,2}, {1,0}, {1,1}, {1,2},
{2,0}, {2,1}, {2,2}});
```

Furthermore, both heuristics ignore the blank tile. So if they encounter a 0 in the 2-D array configuration, they ignore it because it is not a tile.

Another helper method for A* is the method generateNextStates(). As the name suggests, this method takes in a state, and generates all of the next possible permutations of this state. It returns these next states as an array of States. This method works similarly to the move method where both consider the position of the blank tile and determine which of the 4 directions the blank tile can be moved. For every possible move, the method creates a new State and computes every important aspect of it such as f(n), g(n), and h(n). Unfortunately, this method is not as space efficient as it could be. In order to create a new configuration for each move, I clone the parent array. That is because an array is non primitive and if two variables point to the same address, then both are affected when the array is changed. To get around this, I use array.clone().

```
int[][] one = new int[3][];
for(int i = 0; i < 3; i++){
   one[i] = state.getConfiguration()[i].clone();
}
```

For A* search, I begin by setting the important variables. Namely, the Hashtable called reached, the PriorityQueue called frontier, and the goalState.
The reached Hashtable uses generic types <K,V> = <String, State>. I chose String to be the key since Java has a built in hashing function for Strings. The string that it stores is the String representation of the 2-D array configuration. If the 2-D array stores {{0,1,2}, {3,4,5}, {6,7,8}}, then the String representation would be “b12 345 678.” The State class has a function stringRepresentation() that calculates the String representation for a 2-D array. Had I used State for the key generic type, I would have had to create a new hashing function for the 2 dimensional array that holds the configuration of the State.
The generic type for value is a State because I need to keep track of each reached State’s cost. I will explain why later.

The PriorityQueue frontier sorts states by their f(n) cost function. It is a min-at-top heap and is a built in class from the Java API.

```
Hashtable<String, State> reached = new Hashtable<>();
PriorityQueue<State> frontier = new PriorityQueue<>(new Comparator<State>() {
   @Override
   public int compare(State o1, State o2) {
       return o1.getCost() - o2.getCost();
   }
});
```

Finally, my goalState is a local variable that is set when the A* algorithm finally reaches the goal state. This is uses to construct the moves in order to solve the puzzle. I will explain how later.

The actual A* algorithm works like so:
While the frontier is populated and the algorithm has not generated more nodes than max (as specified by maxNodes())
The State with the smallest cost is retrieved from the frontier. Then, I use generateNextStates() to give me the next states for top.

```
State top = frontier.poll();
State[] nextStates = generateNextStates(top, heuristic);
```

For each of the non-null next states, I check if they are the goal. If so, the algorithm exits the outer while loop. Otherwise, I check if this state has been reached before.
If the state has never been reached before or if the previous time the algorithm reached this state, the cost was higher than this time, I add the next state to the frontier and to the reached set.
Finally, I add the top node into the reached set.
Eventually, the loop will break and the algorithm will either print out an error message because maxNodes was exceeded or it will call the correctPath method.

correctPath is a helper method that takes in a State and prints out the order of moves from the start state to the goal state.
I do this with a Stack of moves stored as Strings. Every time that a state has a non null parent, I add the parentDirection field to the stack and set the state to its parent.

When I have finally reached a state without a parent, I know that this is the starting state.

```
while(state.parent != null){
   stack.push(state.getParentDirection());
   state = state.getParent();
}
```

Now, I empty the stack and print out the moves. This way, they are in reverse order from how they were inserted into the stack and are in the correct order from start to finish.

For **beam search**, I use many of the same helper methods as A*. The main differences between A* and beam are the k beams and the ordering of the frontier. (The frontier is ordered only by the manhattanDistance heuristic h(n), not f(n)). Also, I chose to exclusively use heuristic 2, or manhattanDistance because it is an admissible heuristic that can work equally well for this search.

In order to consider k best states at each iteration, I have an outer loop which checks that maxNodes is not exceeded. Within that, I have another loop which runs until there are at least k viable states in the frontier.
To do this, I expand the current state, depth by depth using a Queue. In the queue, I store the current depth. For each one of the elements in the current depth, I expand their children. If the children have a lower manhattenDistance (h2) cost that their parent, they are added to the queue and the frontier and make up the next depth. Each successor is also checked to see if it is the goal state. If it is, the loop is broken.
This process repeats until enough depths have been expanded that the frontier has >= k elements.

```
State parent = currentDepth.poll();
State[] successors = generateNextStates(parent, "h2");
for(int j = 0; j < successors.length; j++){
   if(successors[j] != null && manhattanDistance(parent) >
manhattanDistance(successors[j])){
```

However, since beam search is incomplete, I check whether or not there can be a solution. After an entire depth of States are expanded, if none of them have children with a lower manhattenCost, then there are no more states to be expanded. The algorithm has hit a local minimum and will never solve the solution. In this case, I break the outermost loop.

```
if(currentDepth.isEmpty()){
   System.out.println("No solution");
   return;
}
```

Otherwise, the frontier should be filled with at least k states. At this point, the frontier spits out the k best states that it is storing into the currentDepth queue. Finally, I clear the frontier, effectively discarding all of the nonoptimal states.
Then the algorithm repeats.

Finally, when the outer loop is broken somehow, I use the correctPath method to print out the moves, or I print out the corresponding error message if the algorithm failed.

Lastly, the **maxNodes** function is very short. All this does is set a global variable to the inputted value. This global variable is used in all of the search algorithm loops and is decreased every time that a new node is generated. If this function is not called, the global variable defaults to the maximum value for an integer – effectively infinity. This makes sure that the search algorithms can generate as many nodes as they like unless specified otherwise.

Code Correctness:
First and foremost, both search algorithms will fail if the state inputted is not a solvable state. This can easily be checked by setting the start state to an unsolvable configuration such as "b12 348 567". This will throw a NumberFormatException.

<u>A*</u> uses a frontier priority queue ordered by f(n) = h(n) + g(n) where h(n) = given heuristic h1 or h2 and g(n) is the moves from the start node. A* will continuously draw the lowest cost node from the frontier, expand all of its children, and add them into the frontier if they are not redundant. Redundancy is checked via a reached hashtable, If a child is in the reached set but contains a lower cost than the identical state in the reached set, then it will still be added to the frontier.
A* will stop when the state popped from the frontier is the goal state.

If the input is a good input, then the A* is a complete algorithm meaning it will always find a solution.

To see A* fail due to bad input, use the commands:
setState b12 348 567
solve A-star h1
Expected Output:
java.lang.NumberFormatException: This is not a solvable state. Please only set the puzzle to a solvable state.

To see A* fail due to a small maxNodes, use the commands:
maxNodes 10
setState 142 358 67b
solve A-star h1
Expected Output:
Error: Max Nodes is too small for given problem

To see A* succeed, use the commands:
setState 142 358 67b
solve A-star h1

Expected output:
Moves = 4
up left up left

Local Beam Search uses a priority queue ordered by f(n) = h(n) where h(n) is the Manhattan distance heuristic, a reached set using a hashtable like A*, and takes an input k. While the priority queue contains fewer elements than k, the algorithm will expand all of the children of the deepest depth expanded so far. Each child will be added to the queue only if they have a lower cost than their parents.
However, this algorithm can fail if every child of the current depth does not have a lower cost than its parent. This means that all k current states have hit a local minima. This way, the frontier cannot fill up with at least k states and the algorithm must be terminated because it will never find the solution.
If the frontier does fill up with at least k states, then the top k states are popped from the queue and the rest of the elements are discarded. The top k elements are now the new deepest depth and the algorithm repeats until it fails or until the goal state is reached.

To see beam fail due to bad input, use commands
setState b12 348 567
solve beam 5
Expected Output:
java.lang.NumberFormatException: This is not a solvable state. Please only set the puzzle to a solvable state.

To see beam fail due to a small maxNodes, use commands:
setState 142 5b8 367
maxNodes 10
solve beam 5
Expected Output:
Error: Max Nodes is too small for given problem

To see beam fail due to local minima, use commands:
setState b82 764 153
solve beam 5
Expected Output:
No solution - local minima found.

To see beam succeed, use commands:
setState 142 358 67b
solve beam 5
Expected Output:
Moves = 4
up left up left

Experiments:

Every test is being run after randomizing the current state 100 times

# Max Nodes	A*(h1) successes	A*(h2) successes	Beam(99) successes
5,000	3/50	42/50	1/50
10,000	17/50	45/50	0/50
15,000	18/50	50/50	1/50
20,000	20/50	50/50	2/50
50,000	38/50	50/50	1/50
100,000	46/50	50/50	0/50

1. For A*(h1), very small values for maxNodes will have very few successes. This rate dramatically grows with maxNodes until ~10,000. From there, the success rate plateaus until maxNodes grows past 20,000. Then, the success rate grows a lot until finally plateauing at nearly 1.0 for huge values ~100,000. For A*(h2), a very low maxNodes will still yield almost 1.0 success rate simply because h2 generates many fewer nodes than h1. However, around 15,000, A*(h2) will reach 1.0 and will remain there. For beam, the issue is not the number of nodes, but simply that beam search will hit a local minima and not find a solution. MaxNodes do not affect this algorithm as much as the previous two.

MaxNodes = infinity

Randomizations from goal state	A* (h1) nodes generated	A* (h2) nodes generated
10	15	12
20	21	21
30	123	61
40	70	38
60	18805	962
80	12492	2464
100	21449	2284

2. As shown by the data above, in every case, h2 performs better or the same as h1. In most cases, particularly as the state becomes more and more complicated with more random moves, h2 performs much better than h1.

MaxNodes = infinity

Randomizations from start state	A* (h1) moves	A* (h2) moves	Beam(99) moves
10	6	6	6
20	8	8	8
30	6	6	6
40	10	10	10
60	8	8	8
80	22	22	No soln
100	20	20	No soln
120	20	20	No soln

3. As seen in the table above, when all three solutions find a solution, they use the same number of moves. However, when the randomizations grow and the state is more complex, the local beam search will hit a local minima and not get a solution.

This information is an average of the values that I got for question 1

A*(h1) Average	A*(h2) Average	Beam(99) Average
146/300 = 0.47	287/300 = 0.96	5/300 = 0.0167

4. As seen in the table, out of the 300 trials run in part 1 for different max nodes, A*(h1) solves 47% of the problems. A*(h2) solves 96% of the problems. Beam search with k = 99 solves 1.67% of the problems.

Discussion:

1. First and foremost, Beam search is not a viable solution. Even if it were much faster than A*, it only succeeds 1.6% of the time. That is not a reasonable success rate. Between A*(h1) and A*(h2), both algorithms generate the optimal solution. However A*(h2) is far more efficient in the number of nodes generated. Therefore, A*(h2) has better memory complexity. Since A* will consider each generated node the same despite which heuristic is picked, the solution that considers more nodes will take more time than the solution that considers fewer nodes. By process of elimination, A*(h2) must be the best choice.
2. Testing these algorithms was incredibly difficult. Since I did not have any way to check whether or not the solution was optimal, only if it was complete. This was a challenge. I am still not sure if the algorithms give the optimal solution; I am hoping for the best. Also,

in the debugging process, tracking all of the states that were expanded was very challenging as the number of states was large and staying on top of everything to notice mistakes in the algorithm was very difficult.