Austin Feydt
EECS 391
Project 1 Writeup

1. Code Design

I broke up this project into a few different classes.

1. StateNode

This class contains all of the code that manipulates the puzzle state. It has a lot of very import fields, such as:

- the puzzle state (2D int array)
- the heuristic cost to get to this state from the previous state
- the actual cost from the initial state (number of slides)
- The total cost (sum of heuristic and actual costs)
- The path, or the sequence of moves to get to this state from the start state

The constructor takes the puzzle orientation, the heuristic function cost of this orientation, and the step cost to get to it, which is just the **actual cost of the previous node + 1**, and also the path to get to tit, which is just **the path to get to the previous node + the move made to get to this state.**

This class also contains basically all of the command line functions, like **setting the state** (which means we first validate the incoming state to make sure that every number is used once, no invalid numbers, etc), **randomizing the state** (which starts the state at the goal and works backwards, overwriting whatever state existed before), **printing the state, moving a tile, populating the puzzle from the string input.**

It also includes some auxiliary functions necessary for the searches later, like **finding the coordinates of the blank tile, swapping two tiles,** as well as **overriding hashcode() and equals():**

```
public boolean equals(Object obj){
        StateNode compare = (StateNode)obj;
        for (int i = 0; i < this.getState().length; i++){
                if (!Arrays.equals(this.getState()[i], compare.getState()[i])){
                        return false;
                }
        }
        return true;
}
```

This overridden equals function makes comparisons based **solely** on puzzle states. Later on, I do a check to see if the state should be added to a hashset of all visited states. Hashset.add(StateNode) returns **false** if the element is already in the HashSet (which uses the **equals** function to compare). But, I'll go more in depth later :)

2. StateNodeComparator

This class is my custom Comparator for StateNodes. The PriorityQueue is best utilized if the top of the queue contains the best node to explore next. Therefore, I decided that it should rank StateNodes on their totalCost (which is the **heuristic + actual cost**).

```
public int compare(StateNode node1, StateNode node2){
        if (node1.getTotalCost() < node2.getTotalCost())
                return -1;
        else if(node1.getTotalCost() == node2.getTotalCost())
                return 0;
        else
                return 1;
}
```

This way, the top of the queue always contains the StateNode with the lowest cost. This helps shorten the search and also ensures that a more costly path to a state is not explored before a less costly one.

3. InformedSearches

This class contains my search methods and a bunch of helper methods as well. It's fields include:

- HashSet<StateNodes> visitedNodes
    - This is my hashset that keeps track of all visited states.
    - To limit the size of the hash, I implemented a resizing method. Once the size of the hash exceeds a reasonably large number (I chose 3000 nodes), the following code executes:

```
public void resizeHash(){
        Iterator<StateNode> refactorer = this.getVisitedNodes().iterator();
        ArrayList<StateNode> tempNodes = new ArrayList<StateNode>();
        while(refactorer.hasNext()){
                tempNodes.add(refactorer.next());
        }
        this.getVisitedNodes().clear();
        tempNodes.sort(new StateNodeComparator());
```

```
            this.getVisitedNodes().addAll(tempNodes.subList(0,
tempNodes.size() - 500));
            }
```

The general format of the method is that it creates an Arraylist of StateNodes, copies the hashSet to the ArrayList, sorts the arrayList using my StateNodeComparator, and then returns all but 500 of the nodes back to the Hash. This way, the 500 largest-costing nodes are thrown away, since they probably weren't worth visiting anyways.

- PriorityQueue<StateNode> explorableNdoes
    - This field is the priority queue that ranks based on lowest total cost
    - For the beam search, this priority queue must be trimmed anytime its size exceeds 'k'. To account for this, I implemented a trim method:

```
public void trimQueue(){
        ArrayList<StateNode> tempStorage = new ArrayList<StateNode>();
        if(this.getExplorableNodes().size() > this.beamLimit){
                for (int i = 0; i < this.beamLimit; i++){
                        tempStorage.add(this.getExplorableNodes().poll());
                }
                this.getExplorableNodes().clear();
                this.getExplorableNodes().addAll(tempStorage);
        }
```

In general, it creates an ArrayList for temporary storage, copies the first 'k' elements from the queue to the arrayList, clears the queue, then adds the contents of the arrayList back to the queue. This way, we are only every considering the k-best nodes.

- String heuristic
    - It's either "h1" or "h2".
    - I wanted to just define it as a field to let the searches know which heuristic to use
- Int maxNodes
    - User-defined max number of nodes to explore before returning an error
- Int nodesExplored
    - Keeps track of number of nodes explored thusfar
- Int beamLimit
    - The same as the user defined 'k' for local beam search
- Boolean solved
    - Keeps track of whether or not the puzzle was solved

Explanation for A*:

The A* search relies on the recursive method “exploreNodeAStar()”. It polls a StateNode from queue and does some checks:

- Does the hash need resized?
    - If yes, call the resize method
- Is the node that was polled null?
    - If yes, there are no more nodes to explore, so the puzzle is unsolveable
- Did we exceed maxNodes?
    - If yes, alert user of the error
- Is the heuristic cost of the node 0?
    - If yes, we reached the goal state!
- If it’s not the goal state, has it already been visited?
    - If yes, then ignore it and call the recursive function
- If it hasn’t been visited, find its children, add them to the queue, then call the recursive function.

This is a brief outline of the algorithm. A few helper methods were introduced.

- **findChildren**
    - Attempts to move the blank tile in all 4 directions, and returns an arrayList of the resulting StateNodes (if the blank tile couldn’t be moved in the given direction, no StateNode was constructed.)
- **calculateHeuristic**
    - Calculates the appropriate heuristic
- **calculateH1**
    - Calculates the number of misplaced tiles by looping through the puzzle array and checking to see if at each index, the appropriate number is there. ******It is worth noting that to make array methods easier, I changed ‘b’ to the number 0.******
- **CalculateH2**
    - Calculates the Manhattan distance for each node. In a perfect board, the row index of the number would just be the number divided by 3 (integer division, so we drop the remainder), and the column index of the number would just be the number modulo 3. So, for each number, we just take the absolute value of the difference between actual and perfect row and column indices, as shown below:

```
public static int calculateH2(int[][] puzzle){
        int manhattanDistance = 0;
        for(int i = 0; i < puzzle.length; i++){
                for(int j = 0; j < puzzle[i].length; j++){
                        manhattanDistance += Math.abs(puzzle[i][j]/3 - i);
                        manhattanDistance += Math.abs(puzzle[i][j]%3 - j);
```

```
                }
            }
        }
```

Explanation for Beam Search:
The beam search algorithm relies on the recursive function “exploreNodeBeam”. It polls the top element off of the queue and does some checks:

- Do we need to resize the hash?
- Is the StateNode we polled null?
- Have we exceeded maxNodes?
- Is the heuristic cost 0?
- Have we already expanded this state?
- Else, we find all the children of this node and add them to the queue
    - Then, we see if the size of the queue has exceeded the beamLimit
        - If yes, then we trim the queue
    - Then, we call the recursive function.

The same calculateHeuristic function is used, but I decided to go with H2 (Manhattan Distance), because from my A* tests, it is a much better heuristic than H1, and usually results in exploring way less nodes.

4. EightPuzzleSolver
This is the java class that interacts with the user. It has a main method that serves as both the user interface backend, and also the filereader. If no args are entered when running the program, then it uses the console to get commands and display results. **If an arg is entered,** then it treats the arg as a file name, and attempts to open the file. Regardless, the bulk of the program is the **evaluateInput method.** It is a big switch block that determines what command was entered, and what to do. It takes the user input string, sets it all to lowercase (to avoid annoying errors if they forgot to capitalize something), and then tokenizes the string based on spaces. The switch is controlled by the first token. It determines if the first token was setstate, randomizestate, printstate, etc... And within each case, it checks to makes sure appropriate values followed the command (like that a positive k was chosen for beamsearch, for example), and executes code from StateNode and InformedSearches as needed.

2. Code Correctness

The beginning of my test file (p1test.txt) shows that the puzzles can be solved if the goal state is only 1 slide away, and if 10 random moves were made (just to ensure it works for more than 1 slide).

Then, I created a test to calculate both heuristics on a relatively simple puzzle state (blank tile shifted down twice).

Next, we see how a search can fail: If the method explores more than maxNode nodes, then it exits early and notifies the console. Immediately after, I tried to make maxNodes ridiculously big, and showed that there are some puzzle states that even require more than 10,000 node explorations. However, with a large enough beamLimit, beamSearch gets the job done!

Next, i picked an unsolvable state, and showed that the algorithms can determine that it is unsolveable (AKA, the queue is empty but the goal hasn't been found). I found the unsolveable state on somewhere on StackOverflow. Unfortunately, the computer runs out of room before all of the nodes can be visited, thus it returns the memory allocation error, rather than getting to the end of the priority queue.

3. Experiments (use command "runExperiments")

(a) How does fraction of solvable puzzles from random initial states vary with the maxNodes limit?

My test function **testSearchMethods()** generates 500 random puzzles, and sees if the algorithms can solve in under 100 node generations, under 500, under 1000, and under 2000. It then takes the fraction of completed puzzles of all 500 puzzles. In general, **as maxNodes increases, so too does the fraction of solvable puzzles.** At a maxNodes of 500, only 5.8% of puzzles could be solved using A* and H1, whereas at 15.8%, it jumped up to 15.8%. **A* with H2 and Beam Search with k = 50 both had the similar upward trends.**

(b) For A* Search, which heuristic is better?

By observing the fractions obtained from above, it is clear that **H2 (Manhattan Distance) is a much better heuristic than H1 (misplaced tiles).** At maxNodes = 2000, only 15.8% of the puzzles could be solved by H1, whereas 56.4% of them could be solved by H2. This means that H2 was able to explore less than 2000 nodes and come across a solution much more often than when A* used H1. **Therefore, the Manhattan heuristic does a much better job at predicting how close you are to the goal than by just counting the number of misplaced tiles**.

(c) How does the solution length vary across the three search methods?

From my test function, I was able to determine average path lengths for A* with H1, A* with H2, and Beam with k = 50 for 500 randomly generated puzzles and maxNodes = 2000. This was done by summing up all of the path lengths (when the puzzles were actually successfully solved), and then dividing that by the total number of solved puzzles. As you can see, **A* with H1 had on average, the shortest path**. However, considering it cannot solve 85% of the puzzles, this number should be taken with a grain of salt. **A* with H2 took, on average, 4 more steps to solve the puzzle than H1.** Also, **beam with k = 50 took about 9 more steps to solve than H2, and about 13 more than H1.** A difference of

4 steps may not be that big of a difference, but it is clear that **beam search is taking long paths to reach its goals.**

(d) For each of the three search methods, what fraction of your generated problems were solvable?

This was already eluded to in part **(a),** but there is a very evident difference in solvable states between the three searches. **A* with H1 is by far the least likely to find a solution.** It only was able to solve about 15% of puzzles, even when the maxNodes was set to a relatively high value of 2000. **A* with h2 did significantly better than A* with H1.** It solved 56.4% of puzzles under the same conditions as H1. **However, beam search was most likely to find a solution.** It solved 82.2% of the 500 random puzzles that it attempted, making it significantly more reliable than A*.

4. 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 believe that, based on my experiments, **A* with H2 is much better suited for this problem, but Beam search also has its advantages**. Even though **A* with H1 had the shortest solutions**, it rarely was able to actually come up with a solution, especially compared to the other 2 algorithms. **Beam search** seems superior in terms of space. It was able to solve about 85% of puzzles without exploring more than 2000 nodes, thus saving a lot of room. However, it took significantly more steps for beam search to find the goal state, showing that it is not very optimal. **A* with H2** was in the middle of the pack for all of the metrics considered. It produced path lengths that were about 9 steps shorter than beam search, making it superior in terms of time. With a large enough maxNode size, A* with H2 is also able to solve a significant number of puzzle states, without exploring **too** many states.

(b) Discuss any other observations you made, such as difficulty of implementing and testing each of these algorithms.

By completing this assignment, I have noticed a few things worth mentioning. First, **a solid foundation to you algorithms is crucial.** I spent much more time setting up StateNode and choosing proper data structures than I did actually coding the search algorithms. With the correct choices for my queue and hash, and creating a smart comparator, I was able to easily keep track of the multitude of nodes being generated, stored, polled, ignored, etc. Also, **implementing these algorithms takes patience.** Writing recursive methods can be difficult, especially when there are so many special edge cases to consider, like in A* and local beam search. Again, I tried to plan and iterate my planning before I started coding. Ultimately, I still had to change my code quite a few times, but not nearly as often as I would have had to if I went into it without atleast writing my own pseudocode and going through examples of the search algorithms first.