Adele Fuchs
391 P1 WriteUp

1. **Code Design:** How your code is organized and what design choices you made. Include code in write up but not all of it. Only use relevant portions for specific points.

I have one class that represents the board and operations that can be done on the board called puzzle. I have another class called solver that contains all of the necessary classes and methods to solve the 8 puzzle. This was done so that the classes did not get muddled and so that the solver class could potentially be used to solve a different puzzle or game.

In the puzzle class I chose to represent the board as an ArrayList because the time complexity to access elements is O(1) and I knew that the elements would be accessed often because the move function requires access to elements and their locations. No other choices made in the construction of the puzzle class are particularly unique or interesting.

In the solver class I created a class of Nodes that are comparable based on whichever of the 3 heuristics is passed in as a parameter. The following is the constructor for the Node class:

```
public Node(String state, int g, Node parent, String move, String heuristic) {
. . .
}
```

This choice was made so that I could easily override the compareTo method of the Node class to use the heuristic values:

```
public int compareTo(Node other) {
if(this.heuristic.equals("beam")){
return Integer.compare(this.h, other.h);
}
else return Integer.compare(this.g + this.h, other.g + other.h);
}
```

This allowed me to use a priority queue to “naturally” order the nodes as they now had comparable values defined by a heuristic. A priority queue was used as the data structure behind both the A star search and the beam search as in both searches, values are considered methodically based upon a function and thus the searches lend themselves to priority queues.

Also within the solver class I defined general methods generateNextNodes and tracePath as well as a small class called moveInfo. Both methods and the class are used in both the beam search and the A star search as they perform actions that are necessary for both types of searching. moveInfo is somewhat redundant as the same functionality could be implemented using the node class. However, to simplify retracing the path to determine the moves, I decided to create this class.

When a solution is found and then the moves from the initial state are printed I made the choice to begin printing from this initial state so that the first move is more in context. As a result the “first” move and state printed are null and the initial state and the next moves and states are the actual actions that should be performed to reach the goal state. The number of moves calculated and printed does not include this first null “move.”

An important note about the functionality of my searching algorithms is that they do not alter the state of the puzzle. They do not move the pieces on the board and put the board in the goal state but rather output a sequence of moves that could then be put into the move function of the puzzle class to actually put the puzzle into the goal state.

My A*search works by:

- Creating a priority queue and a hashmap of costs from the start to the current node
- The start state is added to both data structures
- While we have not surpassed the nodeLimit
    - Pop the head off of the queue and check if it is the goal state
    - If it is, we are done and can retrace our path through node parents
    - If not, we will find and evaluate (based on heuristic) the states reachable from the state which we just popped off of the queue and add them to our priority queue. We will only add them if we have not been in the state before or if this way of reaching the state has a lower cost than the way previously found.

My beam search works by:

- Initializing a priority queue and adding the initial state to the queue
- For each node in the queue we then check if it is the goal state
- If it is the goal state, we are done searching and we can retrace the path again using the node class’ parent child relationship
- If it is not the goal state, we will generate all states reachable from the current state and add them to a compounding list of potential next states from any of the visited states.
    - While doing this we will check to make sure that the node limit is not violated.
- Since beam search only considers the k best states, we will sort this list of potential next states based on our evaluation function and add the k best next states to our priority queue.
    - The evaluation function uses a combination of h1 and h2. To do this I simply summed the h1 and h2 value for a given state and defined this to be the state evaluation function. I did not include any parameters to account for the cost from the start to reach the current state. I made this decision because I assumed that while neither heuristic is perfect, they both have advantages and thus the combination of the two could lead to a more advantageous evaluation function.

2. **Code Correctness:** Demonstrate your code on examples with the goal of proving to the grader that your search algorithms function correctly. This should explain and illustrate how the search algorithms work and how they can fail. Choose examples that are concise and easy to verify.

My program, including the searches, runs from the puzzle class. I have included the txt files that I used to test the program in my zip file. In general I aimed to show the basic functionality of my algorithms as well as cases where they may fail or exhibit poor performance. I set a seed of 678 in my test files and created a command to update this seed from the txt file. In addition, when a line of the txt file is blank or contains a description of what is being tested, the file reader will output an empty line.

The first txt file named “inputpuzzle.txt” demonstrates the correctness of my puzzle class and its methods. Within the file I set the puzzle state multiple times using both legal puzzles and illegal puzzles, print this state, make a series of both legal and illegal moves, and randomize the state multiple times and by multiple amounts. These are the basic functions of the puzzle class and as illustrated by the output of the input file, function properly.

My next testing file is called searchTests.txt and shows a variety of situations where A*(h1), A*(h2) and beam search are used to solve an 8 puzzle. My first case is one in which not many moves are required to solve the puzzle, and in this case as my example shows, all algorithms are able to solve the problem and produce a solution of the same number of moves. Next, I implemented a node limit to illustrate that if given a limit, none of the searches continue to execute if they have surpassed the node limit. My next test in this file is one which illustrates how h2 is superior to h1 as it results in the creation of fewer nodes to solve a complex puzzle state. This test also illustrates how both A* searches are superior to beam search as beam search is unable to solve the puzzle. Since my implementation of beam search cannot solve the puzzle, in order to end this test you must kill the command in the terminal.

My final test file is called beamTests.txt and it is meant to illustrate the impact of changing the width of the beam search. I execute 3 beam searches of widths 1,2, 5, and 50. From these cases it can be seen that up until a certain point, the larger the beam, the more nodes will be created.

3. Experiments:

I used a seed of 678 to ensure that each method was solving the same problem. In addition, I imposed no limit on maxNodes when collecting this data. Approximate Effective Branching Factor equation from https://stackoverflow.com/questions/71514682/how-to-calculate-effective-branching-factor.
As noted on this site, the approximation is better for larger b* values.

	Search Cost (nodes generated (N))			Effective Branching Factor (B*) = N^(1/d)		
Depth (d)	A*(h1)	A*(h2)	Beam k=5	A*(h1)	A*(h2)	Beam k=5
1	4	4	4	4	4	4
2	8	8	21	2.83	2.83	2.83

3	11	11	58	2.22	2.22	3.87
4	16	13	313	2	1.90	4.21
6	31	25	4589	1.77	1.71	4.08
9	89	33	182044	1.65	1.47	3.84
11	206	88	2668114	1.62	1.50	3.84
12	206	50	7222837	1.56	1.39	3.73
14	569	324	Won’t solve	1.57	1.51	Can’t compute
15	1111	338	Won’t solve	1.60	1.47	Can’t compute
17	2358	347	Won’t solve	1.58	1.41	Can’t compute
19	6240	697	Won’t solve	1.58	1.41	Can’t compute
22	21093	2540	Won't solve	1.57	1.43	Can’t compute
27	143594	10207	Won't solve	1.55	1.41	Can’t compute

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

As the maxNodes limit increases, the fraction of solvable puzzles does as well. Based on my test below, the fraction of solvable puzzles using beam search from random initial states does not vary greatly depending on the maxNodes limit because beam search consistently requires very many nodes to solve the problem*. The difference in solvable puzzles is larger for an A star search using either heuristic h1 or h2. Both A star searches are not able to solve all of the puzzles using very few nodes but, eventually are able to solve all 10 puzzles. The difference between the two A star searches is that when h2 is used, more of the puzzles can be solved with a lower node limit. (* the result for beam search could vary depending on the k value used).

To test this I ran each algorithm 10 times on different puzzles with different maxNodes values. I then converted how many of the 10 puzzles were solvable into a percentage to find the fraction of solvable puzzles at each maxNodes limit. I randomized my puzzle with 5-50 moves each time using a seed of 678.
For example:
For A* with h1 I ran and recorded the success or failure at each of these parameters:

maxNodes =10 and randomizeState=5,
maxNodes = 10 and randomizeState = 10,
…
maxNodes = 10 and randomizeState = 50
…
maxNodes = 500,000 and randomizeState = 5
…
maxNodes = 500,000 and randomizeState = 50

I did not save each run as that would be an excessive amount to show in this report.

MaxNodes	A* with h1	A* with h2	Beam with k = 5
10	4/10	4/10	3/10
50	5/10	7/10	4/10
100	6/10	8/10	4/10
500	8/10	10/10	4/10
1,000	9/10	10/10	4/10
5,000	10/10	10/10	5/10
10,000	10/10	10/10	5/10
50,000	10/10	10/10	5/10
100,000	10/10	10/10	5/10
500,000	10/10	10/10	6/10

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

For A star search, heuristic h2 (the Manhattan distance), is significantly better than heuristic h1 (the number of misplaced tiles). Using h2 consistently required fewer nodes to solve a puzzle than h1. In addition, when the branching factors differed, that of h2 was always lower than that of h1. This means that less time and space are needed to solve a puzzle when h2 is used. In addition, if we were constrained tightly by time and space, it would mean that h2 could solve a more difficult problem than h1.

c. How does the solution length vary across the 3 search methods?

The solution length is consistent from A*(h1) to A*(h2). When given the same puzzle, they have in all tested cases found a solution of the same number of moves. A*(h1) requires more nodes to come to this conclusion that A*(h2) in most cases, although there are some in which they create the same number of nodes. Beam search's solution length is not always consistent with the other 2 methods, although I cannot recall which situation to recreate the results, beam search's solution was a few moves longer than that of either A* search.

d. For each of the 3 search methods, what fraction of your generated problems were solvable?

When no node limit was imposed, A* using either h1 or h2 was able to solve all of the generated problems (all problems generated backward from goal state and therefore have a solution). Beam search (with k=5) on the other hand was unable to solve problems that required over 12 moves to solve. The program would run for minutes without producing a solution. It is hard to give an exact fraction to represent how many beam search could solve. The max number of moves needed to solve any 8 piece puzzle is 31 (http://w01fe.com/blog/2009/01/the-hardest-eight-puzzle-instances-take-31-moves-to-solve/). Therefore, making the assumption that there are an equal number of puzzles that can be solved with each number of moves, the beam search can only solve 12/31 or around 38% of possible puzzles.

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 A* search with the h2 heuristic is best suited for this problem of the 3 choices. Although in my testing h1 and h2 produced solutions of the same length, h2 outperforms h1 and beam search in terms of number of nodes created to solve any puzzle and time to solve each puzzle. That makes h2 superior in terms of both time and space. Fewer nodes created means first, less space is required and second, less time as the search space is smaller.

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

I found the A* search to be more difficult to implement than the beam search. It took me a while to realize that I needed to implement my "gMap" and track if I am encountering a node for the first time or if a node has already been visited and had a lower or higher g value. Initially I had tried to implement this functionality within my node class but I soon realized that a separate map would function much better and not risk disrupting a path by messing up node pointers. In addition, I decided to not implement a state memory in my beam search algorithm because it was not very effective depending on the beam width. For cases with a small width imposing this constraint meant that once a state was added to the top k list, even if it was not expanded upon, it was considered visited (until I later removed it from the visited list). This seemed like an excessive amount of operations and checks, especially given that even when I implemented this state memory, the performance of beam search did not improve. It would still become stuck on puzzles with complex solutions (> around 12 moves to solve). Removing this state memory from beam search likely made its implementation much easier.