Tristan Maidment [tdm47]

Artificial Intelligence Programming Assignment 1

Code Design

EightPuzzle.java

The first design choice that I had to make was a decision on how to define the various states. My goal was to provide a simple interface for the various algorithms I was about to implement. As such, I created the class EightPuzzle, with the idea that each instance of the object would represent a state.

Within EightPuzzle, I created an enum to declare the move status of the state which it represents. EightPuzzle has a command for each direction of movement. When called, a new state is created with its move type as the move status. For example, running eightPuzzle.left() will return a new object of type EightPuzzle, with the status Move.LEFT or Move.INVALID if the move is completed or cannot be completed. If the move is completed, the object reflects the changes made to the tiles.

```
public EightPuzzle left(){
    if(validLocation(blank.getLeft())){
        return new EightPuzzle(stateSwap(this.state, blank, blank.getLeft()),
Move.LEFT);
    }
    return new EightPuzzle(state, Move.INVALID);
}
```

This decision was made to provide easy propagation of successor states, since each state generates all possible moves, and invalid moves can be filtered out. This step is done by creating a list of type EightPuzzle. This method also allows easy unbiased way to randomize selected state for Beam search, by randomizing the order of the list.

In addition, an inner class called Location was created to abstract puzzle index selection, and provide code that is easy to read. This provides an interface to select a location and then use functions left(), up(), etc. to acquire other locations on the board.

EightPuzzle also contains methods to calculate various heuristics at each step. Since the heuristics are state-specific, it provides a clean way to determine the weight of each node.

AStar.java

AStar creates a PriorityQueue of type Node, which is an inner class that implements Comparator. This was done to make sure the PriorityQueue returns the node with the smallest f(n) each time its polled. In addition, the equals method was modified to only check if the state of the puzzle is equal, and does not compare the current path cost. This is important so that the queue doesn't contain duplicates of the same state, which significantly lowers the number of nodes in the queue, speeding up the runtime. The algorithm works as follows:

1. Remove the node with the lowest f(n) from the queue.
2. Calculate the successor states in each direction for that node.
3. Filter out the invalid moves, and states which have already been reached.
4. Add the new states to the queue.

This continues until either the goal state is reached, node limit is reached, or the queue empties.

```
while(!priorityQueue.isEmpty()){
    Node current = priorityQueue.remove();

    if(current.eightPuzzle.equals(goal)) {
```

```
        System.out.println(printMoves(current));
        break;
    }

    for (EightPuzzle e : current.getEightPuzzle().listMoves()) {
        if(e.getMove() != EightPuzzle.Move.INVALID){
            Node node = new Node(current, e, current.getPath_cost() + 1,
Heuristic.calculateHeuristic(heuristic, e, goal));
            if(!priorityQueue.contains(node)) {
                priorityQueue.add(node);
            }

            if(priorityQueue.size() > EightPuzzle.getMaxNodes()){
                try {
                    throw new EightPuzzle.NodeLimitReached();
                } catch (EightPuzzle.NodeLimitReached nodeLimitReached) {
                    nodeLimitReached.printStackTrace();
                }
            }
        }
    }
}
```

Heuristic.java

This class contains an enum which represents each heuristic. It takes an input of the heuristic type, as well as the goal state and current state, and provides the correct output for each condition.

Beam.java

Beam contains an inner class, BeamSlave, which compartmentalizes each initialized search state. Beam contains a list of all initialized BeamSlaves, with a max size of k. Each BeamSlave contains an function called executeIteration(). While not the most beautiful code, this code provides a randomized Beam Search. Due to how it works, it requires only one BeamSlave be initialized, and will create k-1 different BeamSlaves. It works by generating all successor states, randomizing the order of consideration and creating new BeamSlaves for the following cases:

1. The first optimal successor state. Since the successor states are randomized, this simulates the beam search randomly moving toward a maximum.
2. If there are less than k BeamSlaves, it creates a BeamSlave for other optimal successor states.
3. If it's a non-optimal state, but there are less than k BeamSlaves, create a successor state for the non-optimal state, in the hopes that it leads to a more optimal state.
4. If we're at a local max, remove the BeamSlave, to make space for other BeamSlaves that are still climbing.

```
public void executeIteration() {
    boolean executedSlave = false;
    int totalDirection = 0;
    int badDirection = 0;
    ArrayList<EightPuzzle> moves = current.listMoves();
    Collections.shuffle(moves, EightPuzzle.random);
    for (EightPuzzle e : moves) {
        if (e.getMove() != EightPuzzle.Move.INVALID) {
            totalDirection++;
            if (Heuristic.calculateHeuristic(heuristic, e, goal)
                    < Heuristic.calculateHeuristic(heuristic, current, goal)) {
```

```
                if (!executedSlave) {
                    slaveHashMap.remove(this);
                    new BeamSlave(e, new Node(node, e.getMove()));
                    executedSlave = true;
                } else {
                    if(slaveHashMap.size() <= k) {
                        new BeamSlave(e, new Node(node, e.getMove()));
                    }
                }
            } else {
                badDirection++;
                if(slaveHashMap.size() <= k){
                    new BeamSlave(e, new Node(node, e.getMove()));
                }
            }
        }
    }
    if (badDirection == totalDirection) {
        slaveHashMap.remove(this);
    }
}
```

In addition, each BeamSlave creates a linked list of Nodes, containing the moves required to get to the current state. These nodes are not considered when the algorithm is executing, but rather when returning the steps for the solution, to ensure the search is "stateless". When a BeamSlave is initialized, if it's current state is equal to the goal state, it breaks a conditional loop in Beam. Which stops executeIteration() from executing, and returns the steps used to find the solution.

Code Correctness

A file "input.txt" has been included. The program can be run with the filename as the input, and it will run the contents of the file, line by line. The format of each line in the file is "Randomizations,MaxNodes,k", where each number will change the tests run. For example, the line "50,5000,100" will randomize the input 50 moves, limit execution to 5000 nodes, and run beam search of size k.

The file contains the following:

//Cases with varying RandomMoves MaxNodes and K-Value
50,5000,100
50,5000,200
50,5000,300
50,5000,400
50,5000,500
//Varying Randomization
10,5000,300
20,5000,300
30,5000,300
40,5000,300
50,5000,300
//Failure Cases
50,499,500
50,10,1

Please note that the console, on my machine at least, occasionally has a delayed output. As such, the errors thrown by the last two cases are sometimes output at an incorrect time. Each of the 10 tests before will NOT fail. Failure Case #1 occurs since Beam Search will attempt to create 500 BeamSlaves, when the maximum allocation is 499. Failure Case #2 fails because A* tries to populate more nodes than the maximum allocation of 10. The output of executing the included file is depicted below. Please note that H1 was used for Beam solely out of curiosity, and that Beam Search should be executed with H2 only.

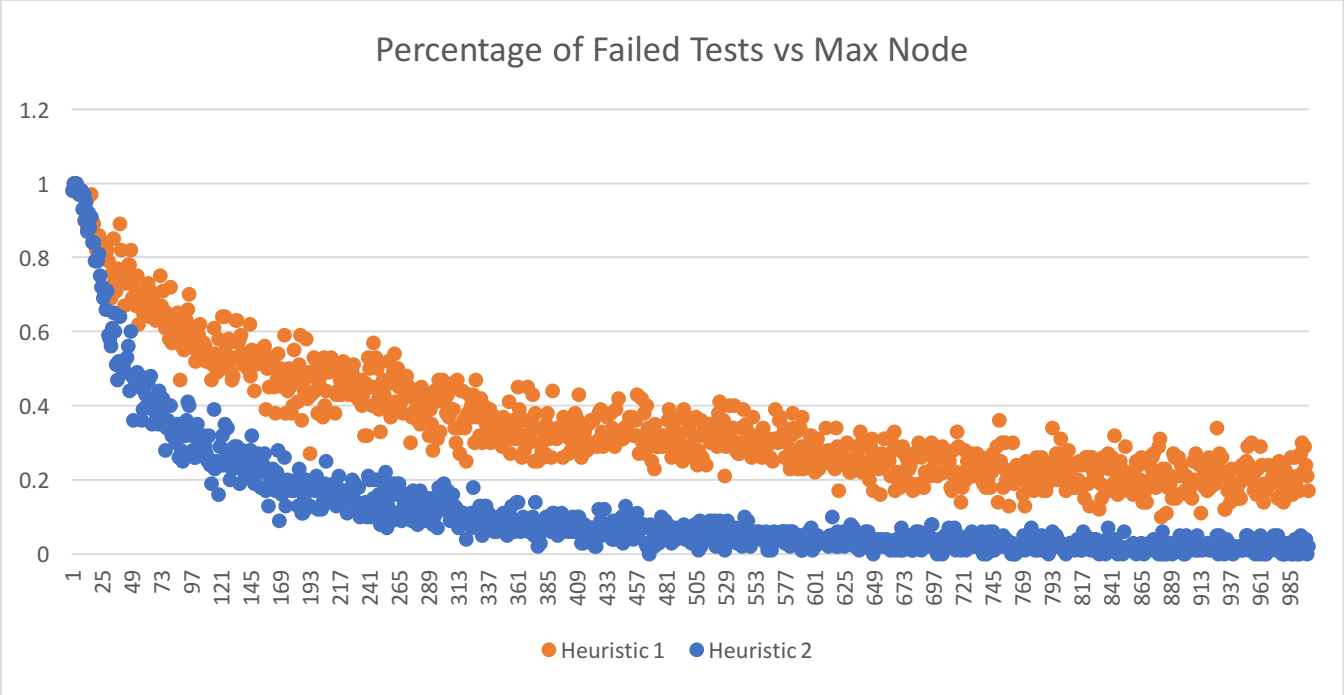
	K-Value					Randomization					Failure	
	100	200	300	400	500	10	20	30	40	50		
A* H1	12	20	8	18	18	4	8	18	6	4	X	X
A* H2	12	20	8	18	18	4	8	18	6	4	X	X
Beam H1	42	28	8	30	38	4	8	1396	6	4	X	X
Beam H2	16	40	8	22	36	4	8	72	6	4	X	X

In addition, the program input can be specified as a set of arguments. Running the program with the input "b12 345 678" will run all 4 algorithms on the specified puzzle.

Experiments

Please note that the functions used for the statistics are contained in *Main.java*. However, they need to be manually run, and cannot be ran from the compiled version of the program. The reason for this is that they don't provide output that's very easy to read or understand. I will be using graphs or other data presentation techniques to make sense of the output.

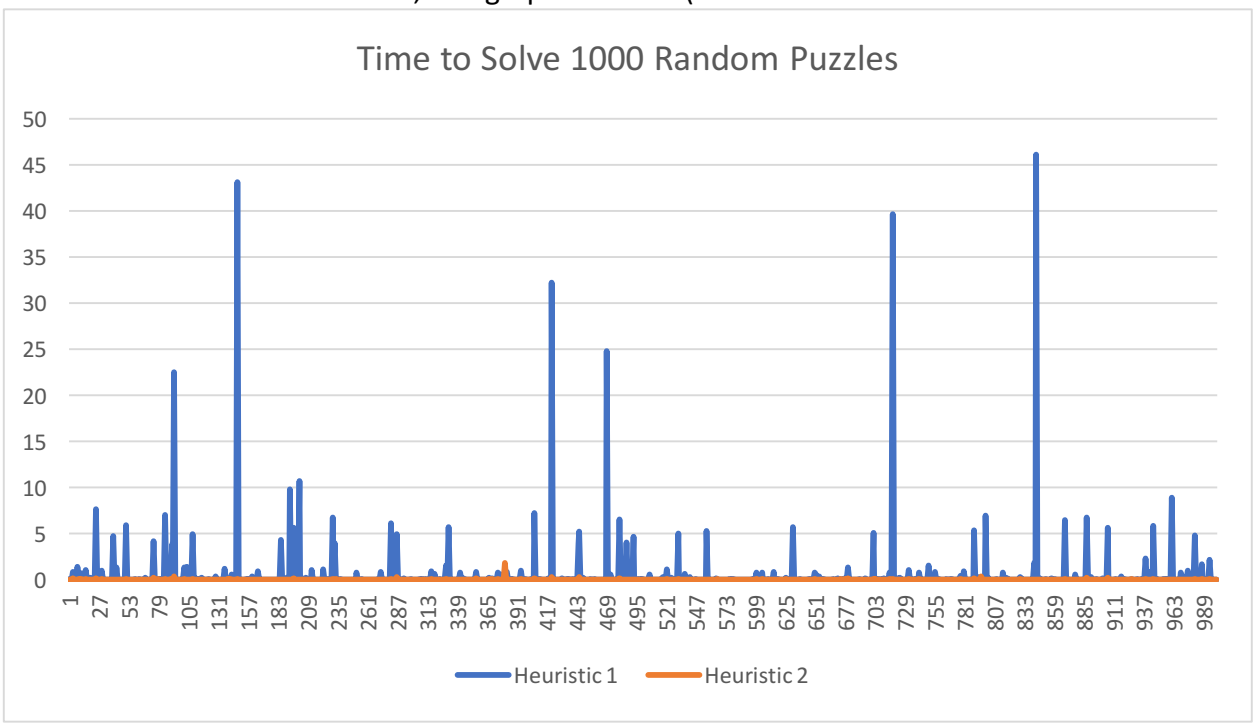
a) How does the fraction of solvable puzzles from random initial states vary with the maxNodes limit?
    a. For this experiment, I used A*, since Beam is solely dependent on k value, it's completely dependent on only the k value. I created a method, maxNodeStatistics(), that calculates the fraction of failed tests for each maxNode value from 0 to 1000. At each maxNode value, I ran A* on a 50x randomized puzzle 100 times. I then divided the amount of failed tests by total tests, giving the fraction of failed tests at each maxNode value. I then graphed the percentage of failed tests as a function of maxNode (see next page). From the graph, we can see that for H2, the percentage of failed tests quickly decreases and appears to reach a minimal at a maxNode of around 650. For this reason, that value could provide an optimal value for space complexity and completion rate. However, even with a maxNode of 1000, ~20% of tests fail with H1. A test with more iterations would provide a more accurate result, but I don't want to melt my laptop.



b.

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

a. As we saw in part A, H2 provides a much lower space complexity. For this reason, we would most likely conclude that H2 is a superior algorithm. However, although the memory usage is lower, H1 may provide a faster runtime. As I have learned in algorithms, the two most important factors when comparing algorithms is time and space complexity. To test this, I timed 1000 iterations of both H1 and H2, and graphed them. (Please note that the colors are switched for



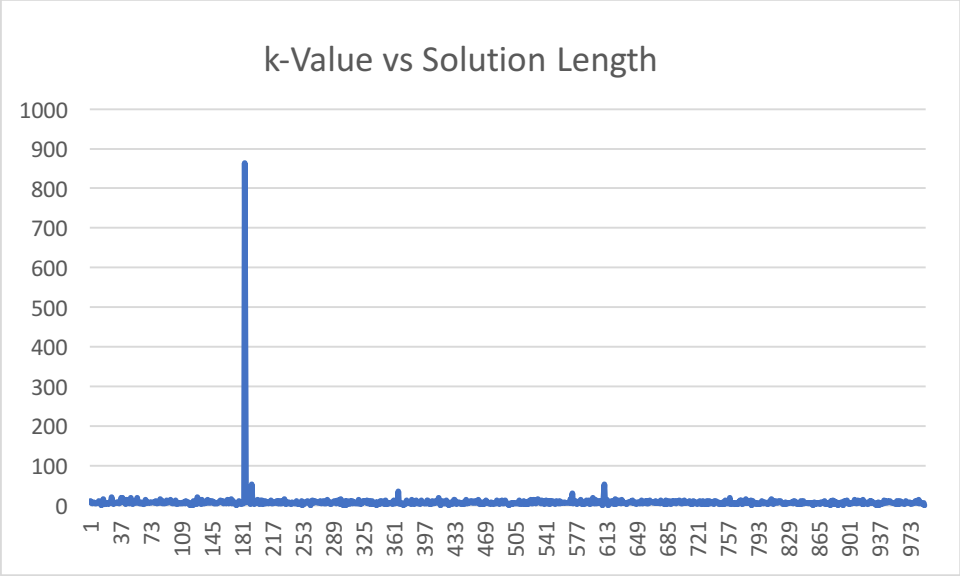
some odd reason). We can see that H1 not only takes more space, but is significantly slower. At this point, we can conclude that H2 is the superior algorithm. This corresponds and confirms the data presented in the book. In addition, using a method I created to calculate nodes generated at each depth, we can find the Effective Branching Factor (using estimate N^(1/d)). We can see that as depth increases the EBF of H1 increases faster than that of H2, proving that H2 is faster.

d	**Search Cost (nodes generated)**		**Effective Branching Factor**	
	A*(h1)	A*(h2)	A*(h1)	A*(h2)
0	1	1	1.898828922	1.861209718
4	13	12	1.752802563	1.68637688
6	29	23	1.685055345	1.553942213
8	65	34	1.666281584	1.503436886
10	165	59	1.663182223	1.490270497
12	448	120	1.644835715	1.525017559
14	1061	368	1.630811642	1.493993798
16	2503	616	1.628401946	1.492905142
18	6482	1357	1.623306611	1.473147733
20	16144	2317	1.616157932	1.45967419
22	38605	4108		

b.

c) How does the solution length vary across the three search methods?
   a. Solution length of all A* solutions, no matter the heuristic, is the same. This is because while A* uses heuristics, it is not to guess the solution, but to guess which nodes are best to check. Beam on the other hand, offers non-optimal solutions. We can see in the chart on page 4 that Beam search using H1 took a whopping 1396 steps for a solution. Although I did mention that H1 was a poor heuristic for this algorithm, it goes to show that Beam does not always provide a very short solution length. I plotted a chart of k vs solution length. Strangely enough, it appears that there is no correlation between the two. However, we can see a case where one solution was almost 900 steps long. I believe this is a case where the heuristic hit a plateau. I am not entirely sure what causes this to happen.

b.



d) For each of the three search methods, what fraction of the generated problems are solvable?
    a. For A* with both heuristics, 100% of problems are solvable, assuming maxNodes is set high enough. maxNodes is set to default at Java's max integer. Since the 8-puzzle has a total of 181,440 distinct states (from the book), it should always be able to calculate an optimal solution. With the way I wrote beam search, it could technically fail (aka hang forever) if every single state exists on a plateau. If this occurred, it's possible that it would keep creating new states on the same plateau, but due to the randomization of successor states, would eventually find its way out. However, at this point, it is most likely to cause a stack overflow or run out of memory. Technically, however, in a perfect world with infinite memory and infinite time, all generated problems, as long as they are valid, are solvable by all algorithms.

Discussion

a) Based on my observations the best search algorithm for this problem is A* with H2. It is guaranteed to find the shortest solution each time, and is significantly better than A* with H1. In terms of space and time, beam search appears to provide an answer faster, and uses no more than k nodes. However, for low values of k, and due to the random nature of beam search, the solution length is not always the shortest. In some occasions, such as using Beam with H1, it can provide a solution 100s of times longer than the optimal.

b) During this project, I noticed that proper planning, such as the definition of *EightPuzzle.java* was required in order to make the algorithm implementation easier. In addition, since I didn't have a concrete example of how beam search was supposed to function, my algorithm is based off of assumptions made by looking at the slides in class. I'm not sure if beam search is a specific algorithm, like A*, or a generalized idea of a hill climbing search with k states.