Shannon Stork
EECS 391 P1 Writeup

1. Code Design

For my PuzzleSolver implementation, I created two inner classes, PuzzleNode and PuzzleQueue. The PuzzleNode class represents a single state of the puzzle, with fields for the puzzle state itself (represented as a two-dimensional int array with the blank tile indicated by 0), the parent state, all four children, the level in the tree (used to compute the number of steps taken to reach the solution), the next node if the node is part of a PuzzleQueue, and the move taken to reach the node if the node was traversed in either A* or beam search. The PuzzleQueue class is a queue that facilitates adding to the beginning and removing from the end, or a FIFO queue. It stores the root of the queue and the end of the queue, as well as the number of elements. The PuzzleSolver class itself stores the initial state of the puzzle, the maximum number of nodes to expand if set by maxNodes(), the root of the PuzzleQueue if A* is used, and whether or not the solution has been found. The overall structure is as follows:

```
public class PuzzleSolver {

	public class PuzzleNode {
		// PuzzleNode class body
	}

	public class PuzzleQueue {
		// PuzzleQueue class body
	}

	// PuzzleSolver class body
}
```

After much research into A* search and many problems with recursive implementations, I decided to implement it iteratively using a PuzzleQueue. The method first initializes the PuzzleNode containing the root of the puzzle. It then handles the base case by testing whether the root node is the solution. If so, it returns the solution. Otherwise, it continues by initializing two PuzzleQueues, one to hold the nodes that are still candidates for expansion and the other to hold the nodes that have already been examined. Then it enters the main loop, which continues as long as !open.isEmpty() && solnFound == **false**, or as long as there are still nodes to be explored and the solution has not yet been found. It pops the node in front of the PuzzleQueue and call expand(), which generates the possible children of that node. It checks if maxNodes() has been exceeded by this expansion, and if so, throw an exception. It calls PuzzleNode[] order = orderChildren(current, heuristic); to dump all of the current nodes' children into an array and order them frm lowest to highest heuristic. For each child that exists (is not null, or results from a valid move on the parent state) in order, the algorithm checks if that node is the goal. If so, we

stop and return the level of the node (number of steps) and path to the goal printed out by node.tracePath(). If it is not the goal, the algorithm makes sure none of the higher level nodes in the queue of nodes to explore and none of the nodes that we have already expanded have an equivalent puzzle state but a lower heuristic. If not, the child is added to the array of elements to explore. Before entering the main loop again, the node that was expanded at the current iteration is added to the queue of already explored nodes.

I also implemented beam search iteratively, but instead of a queue, I used arrays to store the best k PuzzleNodes and their successors at each step. I chose h2 as my evaluation function because it is optimal and admissible. Like A*, beam starts by creating a PuzzleNode for the root and checking whether the root is the goal. If not, it creates an array for the best k successors with the root node being stored at bestk[0] and enters the main loop, which continues as long as solnFound == **false**. Within the loop, it creates an array successors of size 4 * k which stores the maximum of 4k successors of the k best nodes in bestk. All of the nodes in bestk are expanded, and their children that exist and are not duplicates are stored in successors. As each node is added to successors, it is tested to see if it is the goal. If not, the number of nodes explored is incremented and tested to make sure it does not exceed the value specified by maxNodes(). If none of the successors for this iteration are the goal, the heuristics of the successors are computed, and PuzzleNode[] order = order(successors, heuristics) is called to order the successors by lowest to highest heuristic. The first k of these values are stored in the bestk array, and the main loop repeats.

2. Code Correctness

The text file that I have included starts with A* search. The algorithm correctly solves the base case of a sorted puzzle with h1 (scenario 1), a puzzle with 10 random moves with h1 (scenario 2), and the same two examples with h2 (scenarios 3 and 4). These examples were all solved in the minimum number of moves, 0 for the sorted puzzles and 6 for the randomized puzzles. The next two examples are a sorted and randomized puzzle with 10 moves being solved with beam search where (k = 10). Again, the output is proper and involves the minimum number of steps. However, when beam width = 2 for a sorted puzzle (second to last scenario), my implementation somewhat fails, printing out the proper solution 3 separate times. The algorithm breaks when the solution takes more moves than the bean width. For example, “randomize state 10 solve beam 2” would loop infinitely. When I specified maxNodes to be 100 for this test (last scenario), it threw an exception.

3. Experiments

Here is the data compiled from my testing() method on the search cost and path to solution from my experiments. The base case is one case, and all other columns represent the average of two cases with the given shortest path to the solution.

	Search Cost			Steps to Solution		
	h1	h2	beam	h1	h2	beam
base case	0	0	0	0	0	0
1	1	1	0.5	1	1	1
2	2	2	3.5	2	2	2
3	5	5	13.5	3	3	3
4	8	8	25.5	4	4	4
6	58	50	98	6	6	6
8	110.5	105.5	142	8	8	8

a. I ran testing2() 3 times with maxNodes limits of 10, 50, 100, and 200. For 10 nodes, there was 1 h1 failure, 1 h2 failure, and 4 beam failures. For 50 nodes, there were 3 h1 failures, 3 h2 failures, and 5 beam failures. For 100 nodes, there were 2 failures of each. For 200 nodes, there were no A* failures and 1 beam failure.

	P(success)		
nodes	h1	h2	beam
10	0.9	0.9	0.6
50	0.94	0.94	0.9
100	0.98	0.98	0.98
200	1	1	0.995

b. The heuristics were similar, but h2 was better. I expect that if I had continued experimenting on more and more scrambled puzzles, there would be even more of a difference between h1 and h2.

c. The solution length was the same across all three search methods. As explained in the discussion, this is due to my extra condition for beam search. The condition I added decreased time efficiency while making more problems solvable and staying true to the original algorithm.

d. I tested 100 scrambled states using my testing2() method. As expected, h1 and h2 performed identically and well. Each had no failures, making the success rate of both A* heuristics approximately 100%. Beam search failed once, making its success rate approximately 99%.

4. Discussion

A* search is better suited to this problem- in all situations, it finds the best possible solution, whereas beam search sometimes fails. Beam search is also more computationally expensive because of the need to store and order k states for each iteration rather than just focusing on the most likely state. It takes both more time and more space than A* search in my implementation. Theoretically, A* search would also find shorter solutions; it does not just find a solution, it finds the best solution. Beam search is greedy and therefore is only looking for a solution. Because I added a condition in my beam search implementation to make more problems solvable by only adding states to the k state set, fewer problems generate infinite loops,

and the problems that are solvable tend to find equivalent length solutions to A*. This change does also slow down beam search; without checking whether states have already been duplicated among already investigated nodes, beam search could be faster because it does not have to check paths until the shortest is found.

When I initially tried recursive implementations, both algorithms were very difficult to implement. Since the solution could be found in any of the 4 recursive calls (because there are up to 4 children per node), making sure the stopping conditions were met was more of a hassle than I expected. That was when I went back to research and found resources that suggested implementing iteratively. Then, both algorithms were much easier. For both of my attempts, beam search was much easier to implement than A* since it was simpler to keep track of whether or not the algorithm could stop; a path needed to be found for the algorithm to stop. It did not need to be finalized as the best path with consideration of other paths. My algorithms are both technically correct implementations based on the book definitions, but especially beam search behaves a bit differently thanks to the design choices I made.