CSDS 391 P1

Jhean Toledo

October 2022

Code Design

This programming assignment has been coded in Java. I have chosen a few design choices while programming this assignment. First and foremost was the hierarchical structure I decided to implement. I only have two classes, so the structure isn't complicated but is worth mentioning.

I have a parent class, called SlidingPuzzle, that represents the game itself. In this class, I have all the required methods to make the game itself function. This includes, setState, move, randomizeState, and printState. I also have the algorithms to solve the game in this class, called solveAStar and solveBeamSearch. I wanted to separate the code for the game as much as possible from the rest, such as when to create new states for the search algorithms, when to calculate a states heuristics, and so on as to isolate the functionality of the game to be in only the SlidingPuzzle class.

Things unique to my SlidingPuzzle is that I choosen to represent the puzzle itself as an array of numbers. This allowed me to easily index any position on the puzzle to either reference it for further use in my methods or to simply be printed out easily, allowing me to see the puzzle itself. I set the state of this puzzle with the setState method that takes in a string. I decided to air on the side of caution and remove any spaces that would be inputted as extra spaces may cause issues later down the line. I then set the array puzzle as I read from the string one letter at a time, giving me the puzzle in the order desired.

In my move method, I decided to again air on the side of caution and made it return whether the move inputted is a valid move. For example, if a player wanted to move the blank tile up but the tile is already in the top row, I had the method simply do nothing to the current puzzle but still return a string that read "Invalid move." Not only does this allow the player to know that the move was unsuccessful, it allowed me to reference whether a move was made in creating children nodes later on. Conversely, I made my move method return a string "Valid move" if the move was indeed made successfully. I also update the state of the puzzle each time I make a move as to not worry about it later in other methods. This can be seen in the figure below, showing one of my if statement branches. The input I take in is called "direction." For the sake of the example, lets say that the input was "up." The first if statement simply checks if its on the top most row, in which case the tile can't move. Otherwise, move the tile by manipulating the puzzle array, making sure to save what was previously there and "swapping" the tiles by simply overwriting whats in the puzzle at that location.

```
1. // If the empty tile is on the top most space, tell user its an invalid move
2. if ((direction.equals("Up") || direction.equals("up")) &&
3. (location == 0 || location == 1 || location == 2)) {
4.     return "Invalid move";
5. }
6. // Move the "tile" up
7. else if (direction.equals("Up") || direction.equals("up")) {
8.     String old = puzzle[location - 3];
9.     puzzle[location - 3] = "b";
10.    puzzle[location] = old;
11.    setPuzzle(puzzle);
12.    return "Valid move";
13. }
```

My randomizeState method is simple. It starts by setting the state of the puzzle to be the goal state. I do this because any set of moves that started from the goal state is solvable as all you would really need to do is simply do the reverse moves inputted. I then use the int value inputted to make x amount of random moves. I simply pick a number between 1 and 4, where each number is a separate move, x times.

I have a couple of smaller methods that help me such as getter and setter methods for the puzzle array that I have created, again allowing me to ease of use for other methods. I also have a getState method that simply returns the current state but represented as a string. I also have the required method maxNode. All it takes is an int that I then reference to in my search algorithms to check to see if it ever exceeds the value assigned to maxNodes.

A* algorithm

It should be noted that in both of my search algorithms, I have a hash table that keeps track of states and only ensures that unique states get added into the hash table. I then reference this when I make children of a node as to not make duplicate children unnecessarily. In my A* method, I of course take in an input of which heuristic to use, but I also initialize several things in order to make the algorithm work efficiently. As mentioned earlier, I create a hash table. I also create an ArrayList called path that will store the path found to get to the goal state. I also create a PriorityQueue called queue in order to sort the nodes by which has the lowest heuristic (this was overridden in my SlidingNode class that I will explain later). Finally, I create a startingNode that is set to the current state of the puzzle and with a depth of 0 to represent the root of the tree and add it to the queue. To start searching, I have a while loop that runs until the goal is found. As A* is a complete algorithm, it will eventually find a solution. The first thing I do is poll from the queue and check if the node polled is the goal state. I do this by using the following if branch.

```
1. if(Arrays.equals(node.getPuzzle(),correctPuzzle)){
2.      System.out.println("Found the goal state");
3.      // Return the path back
4.      while(node.getParent() != null){
5.           path.add(node.getDirection());
6.           node = node.getParent();
7.          moves++;
8.     }
9.      Collections.reverse(path);
10.     System.out.println("Number of moves it took: " + moves);
11.     System.out.println("Number of expanded nodes: " + expNodes);
12.     found = true;
13.}
```

The condition in the if statement itself is simply checking whether the puzzle received from the current node is equal to another string array initialized at the beginning that is set to the goal state. If they are equal, I then begin by returning the path back by iterating through each nodes parent. Each time I go back I increment a counter called "moves" that simply keeps track of the number of moves made so far. In each node, I make sure to set the direction of what move it made to create that node in order to be referenced here to add to the path. The path technically worked back words as I work back words to eventually get the root node so I then have to reverse the path to get the correct order. I then simply return the amount of moves it took along with the path to get to the solved state.

It is important to note that this if statement simply checks whether the node currently at the top of the queue is the goal state. In order to actually order the queue properly, I have another branch that executes if the current node is not the goal state. This branch can be seen below.

```
1. else {
2.     // Look through the other neighbors
3.     node.createChildren();
4.     for(int i = 0; i < node.children.length; i++){
5.         if( node.children[i] != null){
6.             expNodes++;
7.             queue.add(node.children[i]);
8.          }
9.         }
10.      }
```

Here, I simply create children for the current node (which is set to the first thing in the queue so it has the best heuristic so far). I then iterate through every child and add that child into the node if the child has been successfully created. I increment a variable expNodes that simply keeps track of the number of nodes expanded. I use this for later portions of the write up.

Code correctness

Taking a look at how A* actually solves a state, lets assume our current state is "142 3b5 678." My algorithm would begin by creating a new SlidingNode called startNode that will be the exact same state as the current puzzle but take in a heuristic to be sorted by. That means that the SlidingNodes state is now "142 3b5 678" and its heuristic is the same as the heuristic inputted into A*. It then begins to put that starting node into both the hash table as to keep track of what states have already been seen and is added to the queue as the first item in the queue. We then begin the while loop and set a new SlidingNode equal to the first thing polled from the queue, in this case it would be our startNode as this is the first iteration. It then compares that nodes state to see if its the correct state. This is false as the state "142 3b5 678" is not the same as the state "b12 345 678". This would then go to the other branch of the if statement. There, we begin by creating the possible children for the node. In this case, all four moves are possible so it creates all 4 children. It then simply adds those children into the queue which sorts them to give the state with the best heuristic. The loop then restarts. However, whenever we poll from the queue, we don't poll the startNode, we now poll whatever is in the front of the queue. In this case it would be the up child we made in the for loop as that results in the lowest heuristic. We then once again check if that node is the correct state, which it is not, so we then again create its children and add them to the queue. This process repeats until we find the goal state.

Once the goal state is found, we begin to enter the first branch of the if statement and use a while loop that runs until a node no longer has a parent (in which case that means we have found the root) and then adds the directions used to the path list, increment our move counter, and then set the node to its parent to repeat the cycle again. Once a node no longer has a parent, we print out the path found and the number of moves it took to get there.

Beam Search Algorithm

In my beam search algorithm, I again use a priority queue but in conjunction with an array of the input size k. I still initialize similar things as in A* such as my hash table, a ArrayList to store my path, and a int counter to keep track of the number of moves. Before I begin the actual search in my algorithm, I again create a starting node that will act as a root for the tree. I add this node to the first index to the array. I also decided to use h2 as my heuristic for beam search as it seemed to perform much better than h1 based on my experiments.

As for my actual search, I again use a while loop that runs until the goal state has been found. In the loop, I immediately check the first index of the array. The array itself is sorted so taking the first element in the array will return the node with the best heuristic so far. I then use if statements to check whether the node with the best heuristic. The first branch of this if statement is the same as my first branch in my A* algorithm. However, my else branch differs as seen below.

```
1. for(int i = 0; i < array.length; i++){
2.     // Create children for everything in the array
3.     if (array[i] != null){
4.         node = array[i];
5.         node.createChildren();
6.
7.         // Loop through every single child created and add to queue
8.         for (int j = 0; j < node.children.length; j++){
9.             if(node.children[j] != null){
10.              queue.add(node.children[j]); // Sort the best of the children
11.              expNodes++;
12.          }
13.        }
14.    }
15.  }
```

The first for loop runs for the length of the array, k. I then check if there is an item at that index in the array and take the node at that index and create its children. This is lines 3-5. I then have another for loop that runs for each child created. If the children was created successfully, add the child to the queue. This allows me to add children for every node in the array if the child is a unique state.

I then have to add everything in the queue into the array outside of for loops. As the queue is a priority queue, the queue itself is already sorted. I use this to my advantage by adding items in the queue one at a time to make my final array sorted without actually having to do any additional sorting on my array. This can be seen in the code below.

```
16.  // Clear the array, then add k things from queue.
17.        Arrays.fill(array,null);
18.        for(int x = 0; x < k; x++){
19.           array[x] = queue.poll();
20.         }
21.         queue.clear();
```

It is important to note that I clear the queue as to allow the queue to be filled with new states in the next iteration of the while loop. Thus resetting everything for the next iteration.

Code correctness

Lets assume that the current state is again "142 3b5 678". The beam search algorithm will start in a similar fashion as A*, initializing a hash table and setting a starting node equal to the current state of the puzzle, however we allocate an array of size k and add that start node to the first index of the array. We then enter the while loop and check the first thing in the array to see if that is the goal state. If not we continue to the other branch of the if statement. Here, we have a for loop that runs for the entire length of the array. We then create children for every present item in the array (lines 3-5), in this case we only create children for one state as we only have one item in the array. We then use another for loop to iterate through each child successfully created and add those children to a queue (lines 9-10). Finally, we clear the array of anything it contains as we want to keep only the k best and add each child to the array in order (line 17-19). This would mean that all four children made from the state "142 3b5 678" are now in the array but sorted by which child had the best heuristic. We then finally clear the queue to begin the cycle again.

Now when we begin the cycle again, the first item in the array will be the node with the best heuristic, that node then gets compared to see if its the correct state. In the event that the first thing in the array is in fact the goal state, we simply return the path the same way we did in A*. The example for both A* and beam search can be seen working in the attached text file named "Code correctness example"

SlidingNode Class

As for my second class, I opted to have it extend the first class since I wanted it to inherit the main functionality of the game. I called this class SlidingNode as it would represent the nodes that would be created in order to be searched through in the searching algorithms. I have a constructor in this class. Every time a node is created, it takes in 4 variables: the puzzle to set the node to, the parent of the node, the heurstic to use for the node, and the depth of the node. I also make sure to implement the Comparable interface so I can make my nodes be compared based on its heuristics.

The most important method in this class would be my createChildren method. The method takes no input, as it simply creates children and stores them in an array called children to keep track of the children created for the specific node instance. I have the method iterate through a for loop 4 times as to create 4 possible children as the max number of moves a state could make is 4. For each iteration, I check if the state can move in one of the four directions. This if statement can be seen below.

```
1. if (i == 0) {
2.         // Check if it can move right
3.         if (child.move("right").equals("Valid move")) {
4.
5.           // Check if the hashtable has already seen this state.
6.           // If its a new state add it to the hashtable and create the new node.
7,           if (!ht.containsKey(child.getState().hashCode())) {
8.             ht.put(child.getState().hashCode(), true);
9.
10.            // Create a node that will represent the new move made.
11.            SlidingNode rightChild =
                 new SlidingNode(child.getPuzzle(),this,this.getHeurstic(),this.depth + 1);
12.            rightChild.setDirection("right");
13.            children[0] = rightChild;
14.          }
15.        }
16.      }
```

It is important to note that "child" is initialized to be the current puzzle. I do this because I don't want to alter the actual current node, but to simply reference to it.Here, we can see that in the first iteration of the for loop, it checks if it can move right in line 3. If the move is in fact valid, it then checks to see if the resulting state is a unique state by checking if the hash table already contains the key for the state. If the hash table doesn't already have this state, add it to the hash table and continue creating the child. I create the child in line 11, setting the puzzle to the child that moved the original state towards the right, setting the parent to the current node, giving it the same heuristic as the current node, and adding the current depth plus 1. I finally set the child as the first index in my children array. I then repeat this for each direction depending on the iteration of the for loop.

I also have evalH1 and evalH2 methods that evaluate the value for heuristic 1 and heuristic 2 respectively. In my evalH1 method, I simply loop through the entire length of the puzzle and check if each index is in its correct spot, otherwise I increment a counter. I then return that counter and the current depth of the node.

My evalH2 method is more complex, as it has to calculate the number of moves needed to bring a specific number to its correct location in the puzzle. To do this, I have two for loops. One for the number of rows and another for the number of columns.This can be seen in the code below.

```
1. // Loop through the rows
2.     for (int k = 0; k < 3; k++) {
3.
4.       // Loop through the columns
5.       for (int j = 0; j < 3; j++) {
6.
7.         // Calculate the distance for each individual tile
8.         switch (currPuzzle[(3 * k) + j]) {
9.           case "1":
10.            x = x + Math.abs(j - 1) + Math.abs(k - 0);
11.            break;
12.
13.          // Continue for each tile number ...
```

I then index the current puzzle at the index (3 * k) + j as that will give me one space at a time, starting from the top left most corner of the puzzle, working its way left to right until it reaches the bottom right most corner. Each case has its own math for calculating the distance. Here we can see the math used to calculate the distance for the tile "1". The tile "1" should be at the index (1,0) so we simply subtract the index it currently is at by the index of where it needs to be. This will give the number of moves it needs to make to reach its correct location. I of course store each calculated value to a variable that then gets added to the current depth and finally returned as an output.

Experiments

a)

Percentage of solvable states with varying max nodes				
Max nodes	A* (h1)	A* (h2)	Beam search (100)	Beam search (1000)
500	4	26	0	0
1000	5	48	1	2
2000	12	80	13	1
5000	25	94	100	1
10000	42	100	100	15
20000	68	100	100	77

Table 1: The values listed is the number of solved puzzles out of 100

b) Based on the first experiment, I have a rough idea that h2 is the better heuristic for A*, as it results in more solved puzzles. Below is another table with data regarding h1 and h2 for A*.

Nodes generated for h1 vs h2 on A*		
	A* (h1)	A* (h2)
Expanded Nodes	11637	1667
Moves	11	22

Table 2: The values listed above is the average amount for a total of 100 runs of solving A*

Here, we can see although on average, h1 solves the state in less moves, it takes up much more space than h2. We also know from our first experiment that h1 doesn't always solve the puzzle with a given maxNode limit since it expands so many nodes, meaning that there is a potential that h1 may not solve a puzzle where as h2 would have a much higher chance. Thus proving that h2 is the better heuristic.

c) As the second experiment eluded to, h1 on average seems to have the shortest solution length. The table below shows the solution length compared to all three search methods.

Average number of moves per search		
A* (h1)	A* (h2)	Beam (100)
11	21	22

Table 3: The values listed above is the average amount for a total of 100 runs for each

This shows that on average, as expected, A* with h1 does in fact find the shortest path but again, expands far to many nodes, wasting a lot of space. A* with h2 however seems to perform much better in terms of number of nodes expanded and finding a solution with a bareble size.

d) Below is a table that shows the percentage of solvable puzzles with each search algorithm.

Percentage of solvable puzzles			
Number of runs	A* (h1)	A* (h2)	Beam (100)
50	64	100	100
100	71	99	100
200	71	100	100
500	67.2	100	100

Table 4: The values listed above had maxNodes set to 20000, causing some algorithms to finish prematurely

Discussion

a) Based on my experiments, I would have to say that in terms of shortest solution, A* with h1 as its heuristic is best. However, taking into account the amount of nodes generated by A* with h1, I feel as though A* with h2 is much better. Beam comes in at a close second but doesn't necessarily always find the most optimal solution. In conclusion, A* with h2 as its heuristic is the best search algorithm in terms of time and space.

b) I had quite a bit of trouble trying to implement these algorithms. For starters, I didn't exactly know how to store the unique states until recently. I simply wasn't sure what data structure to use and where to use it. I then remembered that hash tables seemed like the ideal solution to my problem as adding to a hash table is fast and I can easily check if a hash table contains an item with Java's built in methods. Then it was only a matter of where to use it. I knew that I wanted to add a node (or a state of a node) to the hash table once that node had been created. Which lead me to think of using the hash table in my createChildren method.

   Some other difficulties I had was what data structures to use for my Beam search algorithm. I initially thought of using just an array with size of k as to only keep the k best nodes but I wasn't sure how to sort the array as for whatever reason, even though I implemented Comparable and overrode the compare method, using Arrays.sort wouldn't actually sort my array in terms of the heuristic. I had already finished A* and used a PriorityQueue that worked with no issues so I figured I could just use the same idea for Beam search but put only the k best items from that queue into an array.