Chris Tsuei

cxt240

EECS 391

10-2-16

Project 1 Write-up

Code:

I have separated my code into sections, each dealing with a separate part of the project. For the command-line input, I created a Translator class to read the text file and to execute those commands as they are written. I also created a Solver class to implement all three forms of search (A* h1, A* h2, and beam search) as well as output the solution to the problem. The Solver class uses the StateHolder class in order to store possible states as well as already visited states as nodes. Finally there is the Puzzle class, which represents the 8-puzzle game board.

Translator

The Translator class takes in the directory of a text file containing the text. It uses a FileReader as well as a BufferedReader to read the file line by line provided that a valid text file was entered by the user.

```
FileReader inputFile = new FileReader(input);
BufferedReader read = new BufferedReader(inputFile);
```

Commands are to be written exactly as specified in the project instructions, with any input (ex directions, maxNodes, heuristic) to be specified by a < and ending in a >. An example of this is (`move <left>`). There is also a FileWriter in order to print out the states (if specified by the `printState` command) to a text file named results. This FileWriter will also print out the solution to the problem after a search function is called in the command file (`solve beam <k>`, `solve A-star <h1>`, or `solve A-star <h2>`). A sample command file can be found in the src folder containing my classes and code.

Solver

Solver contains the heuristics that A* and beam search are based on as well as the solver methods themselves. Each solver is similar in structure. To initialize, I created an ArrayList and a Hashtable, stored the puzzle to be solved in a StateHolder class (null vales for direction and parent fields and moves value was 0 and total value was measured using whatever heuristic was required). By using an ArrayList, I could easily sort the Nodes according to the total moves (basically heuristic estimate + total moves) and take the first node for expansion (or first k if the search method was beam search). I used a Hashtable to store the state of the puzzle (in the StateHolder node) because it would be easier to search for already visited states.

Also included in the Solver class a printSolution method. This method takes the solved node (the state should read b12 345 678) and prints out a solution via a while loop that looks at the node, adds the direction to a StringBuilder (after reversing the string and adding a newline character) then moves the pointer to the parent node and continuing until the parent has a null value. If there is no solution, then the command file will print a text input that there was no solution due to a null pointer error.

All search methods had pretty much the same format in order to search for the solution. The heuristics that were used (misplaced tiles and total spaces away from goal) were coded into separate methods that could be called by the different search methods. Each search method started by creating an ArrayList to store expanded nodes that hadn't been considered. The reason I used an ArrayList was because of the sort method available to lists. The search method also created a Hashtable in order to store already considered states (I chose this because the hashtable offers easy access to elements). The puzzle to be solved was added to the ArrayList after being stored in a StateHolder.

The search method would then take the state with the lowest total cost after sorting (beam search removed elements k to list.length()-1 so it could consider k elements). After checking to see if the state was already in the hashtable or was the solved puzzle, the search methods would expand the state as a separate StateHolder and add the expanded states to the ArrayList to be considered if the expanded solution was not already in the ArrayList. Finally the state that was expanded was added to the hashtable. This loop was repeated until maxNodes was hit (Hashtable.size >= maxNodes) or until a solution was found.

StateHolder

StateHolder is a node that contains the state of the puzzle as a String, the number of moves made, the estimated total moves necessary to make, the move made to get to the state, and the parent of the node if there is one. This is just a class to hold a bunch of values to be looked at by the sort methods as well as the print method when it is time to print out a solution. This class is comparable based on the total moves estimate.

Inside the StateHolder class, I overrode the comparison and the equals methods. The comparison method was overridden in order to easily sort the nodes in the ArrayList by the estimated cost. I also overrode the equals method in order to properly compare states to see if a node containing a state already existed in the Hashtable or the ArrayList.

Puzzle

This class is the 8-puzzle board in its various configurations. The default puzzle is in its solved state (b12 345 678), however another state can be specified on creating by entering the string containing the board state. It stores the row and column value of the blank square for easier move method. Here, the puzzle can be told to move (up, down, left, or right) and print its state as a String. Also, there is a randomize(n) method that will randomize the puzzle with n valid moves. The randomizer is seeded with a value of 240. Also with the randomizer, a move sequence of down-up or right-left is valid as those are recognized as valid moves.

Code Correctness:

The random number generator is seeded at 240 (this can be changed if you wish to enter a value manually in the code in the randomize method of the Puzzle class. maxNodes is set to 100 as the default value, but that can be changed if specified in the command file.

The randomize state method is consistent. Entering the following command into the command.txt file any number of times will get you the same puzzle state in the result file

```
setState <b12 345 678>
printState
randomizeState 15
printState
```

This will display in results(for each code block):

```
b12 345 678
125 34b 678
```

The heuristic methods for the different solve methods were written as separate methods so I could easily call them initially for testing, but later for the search methods themselves. I ended up using the distance heuristic for beam search. This heuristic is consistent as it measures the sum of number of moves minimum to move each tile to its default position. It is also admissible as it never overestimates the cost to the default state (the heuristic takes the row and column for the tile position and the rows and columns for the default position and calculates the minimum moves for each tile (right, left, up, down) to get that tile to the default state)

For A* search, the StateHolder class will look at the total moves. This measurement is a sum of the total moves already made as well as the estimated moves that will be made according to the heuristic. I also somewhat simplified the search a tiny bit (and preventing a simple loop) by making impossible to do move pairs up-down or left-right. The search could still potentially do up-up-down-down, but getting rid of the single dimensional pairs simplifies the number of repeated states the search algorithm will visit.

For beam search, as described, the program will only look at heuristic. Beam search will only look at the total moves and ignore the moves already made. Beam search is based on the program looking at the randomized states and selecting the best k states based on how good the heuristic judges them.

Testing the code on with a randomized board of 20 moves (command_test.txt), the results can be seen in (result_test.txt). The only way the code will fail is if the maxNodes is set too low for the problem and if the heuristic considered too many states. For example, running a randomized board of 50 moves with only 5 maxNodes considered and using the first A* heuristic will give a result like the following:

```
534 187 62b
A* search failed. Heuristic h1
java.lang.NullPointerException
```

This result is because the program hit the maximum number of states to be considered, so it can’t continue searching the priority queue for more states.

Experiments

For all of my puzzles, I tested a randomized puzzle of 50+ valid moves (via the randomized function in the Puzzle class) with each of the three different solving methods. Each randomized puzzle (say randomize 50) would generate the same problem given the default state as the start state as demonstrated above since the random number generator is seeded. A solution with a valid solution counted as a success and the number of steps was recorded. Anything caught as a NullPointer Exception counted as a failure since that means that the maxNodes limit was reached by the loop. Moves for failures wasn't counted as only the length of the solution is what we're looking for. All experiments were done with variations of the code in the main statement of the Solver class. For beam search, I used 2 as the number of states.

The number of solvable puzzles from the random initial states increased as I increased the maxNodes limit. As you can see in Table 1 below, beam search did not work on any of the random 50 puzzles that I tested my search methods on since beam search is a greedy search that doesn't consider the number of moves already taken. For both heuristics of A*, the number of solutions steadily increased as the number of considered states increased.

MaxNodes	20	25	40	65	85	150	275
A* <h1>	1	2	2	18	18	18	42
A* <h2>	12	17	17	17	31	40	40
Beam <2>	0	1	19	33	37	42	42

***Table 1**: number of solved puzzles given a maxNodes number of considered states*

For A* search, looking at Table 1, the number of solvable states, the second heuristic, which measures the distance the displaced tiles are away from their default state is better. Also if you can look at Table 2, you can see that the average solution length of an A* search using the second heuristic was far smaller than the solution created using the first heuristic as more puzzles were solved (specifically when the maxNodes considered was 275).

MaxNodes	20	25	40	65	85	150	275
A* <h1>	14	13.5	13.5	16.5	16.5	16.5	29.62
A* <h2>	15.58	16.17	16.18	16.18	19.84	23.85	23.85
Beam <2>	0	13.0	16.37	19.73	22.11	25.0	25.0

***Table 2**: average number of moves per valid solution*

The search method with the best solution length was A* using the distance heuristic once all of the puzzles had about the same number of valid solutions (when maxNodes was about 275). Initially, puzzles that required the least number of moves also were the solutions that required the least amount of states to look at (see Tables 1 and 2). As more states were considered, the harder, more randomized puzzles were considered, so the solutions became correspondingly longer. In table 2, when about 80% of the 50 puzzles were solved, the second A* heuristic had the best solution length. Beam search had a better solution length compared to the A* search using the first heuristic probably because I used the second heuristic as the evaluation as it actually measures the number of moves the computer might need to make.

Overall the second heuristic of A* seemed to be solving most of the problems given a reasonable small of states to consider. However, in the last column of Table 1, we can see that beam search and the first A* heuristic solved a few more puzzles. On the whole, the number of solvable problems will depend on the number of states that the search function considers as well as how randomized the initial problem is. I chose about 50 moves as a reasonable amount of moves a person would make on the 8-puzzle in order to measure the number of valid solutions that can be solved by the computer.

Discussion

On the whole, the best search method was the A* search using the second (distance based) heuristic. This is because it was able to solve significantly more puzzles with a smaller number of states considered to come up with a solution. I would say that using the same measure, beam search with 2 states considered would be better than A* search using the first heuristic as beam search solved more puzzles with fewer states considered.

A* search using the distance heuristic definitely found shorter solutions. In terms of space, beam search would be the best as it only retains the best k states, unlike A*, both of which stored all expanded states until those states were considered. In terms of speed, the distance-based A* solved more problems initially faster since less states were considered.

Overall, the primary difficulty was making sure that the search method checked to see if the state was in the queue or had already been considered. I had a lot of trouble with this until I overrode the equals function in my StateHolder class in order for my Hashtable and ArrayList to properly compare the nodes. Also, in my printState() method inside the Puzzle class, I was adding an extra space to the string version of the puzzle state, which made comparisons of the states difficult as I was only comparing states of the puzzle, not the number of moves made, parent of the state or the g(n) + h(n) measurement of the cost.