Writeup for Project 1

By Jieyu Ren(jxr477)

EECS 391 Sep 27 2018

**1. Code Design**

In my program, I used a single array to represent the game board. Here are examples and illustrations of some main methods in my program:

EightPuzzle x = new EightPuzzle()

char[] y = new char[]{'b','1','2','3','4','5','6','7','8'}

1. x.setState(“b12 345 678”)

This method requires a String input. It uses charAt() to store the input in current state.

2. x.randomizeStates(10)

This method requires an int input. It creates a changing int called direction using Math.random() and makes the state continue to move in that direction until the step size entered is reached. Since it starts from the goal state, all states it generates should be solvable.

3. x.printState()

This method will simply print the current state.

4. x.move(“up”)

This method requires a String input and the input should be either “up”, “down”, “right” or “left”. First, it will find the location of the bland tile “b”. Then, if “b” can move in the direction entered, it will swap the value in “b” and the next tile in that direction. Otherwise, it will return false to indicate it cannot move in that direction.

5. x.getDifference(y)

This method requires a char[] state input and will return the number of tiles out of place in that state.

6. x. getManDist(y)

This method requires a char[] state input and will return the total Manhattan distance in that state.

7. x.isSolvable(y)

This method will determine whether a state is solvable based on the number of inversions in that state.

8. x.curRandom(10)

This method is very similar to randomizeStates. The only difference is that it makes n random moves from current state instead of from goal state.

9. x.maxNodes(50)

This method sets the limit of maxNodes.

The main solve algorithms will be explained in the next part.

**2. Code Correctness**

1. x.solveAStar(“h1”)

This method requires a string input which should be either “h1” or “h2”. In both cases, it will first create two array lists to store all states visited and the f-values of these states and then add the starting state into the two array lists. Then, it will create a while loop which continues looping until the current state is equal to goal state. In the while loop, it will create 4 instances of EightPuzzle which are used to test whether it can move in a direction. Then, it will check whether the movement will come back to the previous state and make the movement only if it will not come back. Later on, it will put these states into array lists and start again with the state which has the smallest f-value. The f-value is defined as the sum of g and h. G is an int which starts from 0 and increases 1 in every iteration. H is the cost got from different heuristic functions. In the case of (“h1”), h will be the number of out-of-place tiles while in (“h2”), it will be the Manhattan distance, which is the total steps each tile needs to move to reach the goal state. In the end, the current state will be printed.

Example:

EightPuzzle x = new EightPuzzle();

x.randomizeState(10);

x.printState();

output: 142 375 68b

x.solveAStar("h1")

142 375 68b

142 375 6b8

142 3b5 678

1b2 345 678

b12 345 678 x.randomizeState(10);

x.solveAStar("h2")

output: 142 375 68b

142 375 6b8

142 3b5 678

1b2 345 678

b12 345 678

2. x.solveBeam(4)

This method requires an int input k, which is the number of randomly generated states. Again, it will use two array lists to store states visited and all successor states. At each step, all the successors of all k states are generated. If anyone is the goal state, the algorithm will stop. Otherwise, it will select the k best successors from the complete list and repeat. By the way, in this case the k value should be less than or equal to 4 since we only have 4 possible successor states.

Example:

x.randomizeState(10);

x.solveBeam(4)

output: 142 375 6b8

142 37b 685

142 37b 685

142 375 68b

14b 372 685

142 3b7 685

142 375 68b

14b 372 685

142 3b7 685

142 375 6b8

142 375 6b8

142 375 6b8

142 375 6b8

142 375 6b8

142 375 6b8

142 3b5 678

142 3b5 678

142 3b5 678

142 3b5 678

142 3b5 678

142 3b5 678

1b2 345 678

1b2 345 678

1b2 345 678

1b2 345 678

1b2 345 678

1b2 345 678

b12 345 678

b12 345 678

b12 345 678

**3. Experiments**

a. With the increase of maxNodes, the fraction of solvable puzzles from random initial states will also increase.

|  |  |  |  |
| --- | --- | --- | --- |
| maxNodes | A\*(h1) | A\*(h2) | Beam |
| 10 | 3 | 5 | 2 |
| 20 | 3 | 5 | 3 |
| 50 | 4 | 5 | 5 |

Note: these numbers stand for the number of puzzles each algorithm solved out of 5. The random size is 20.

b. From the table below we can see that for A\* search, h2 is much better because the states it uses is always less than or equal to h1.

c. The solution length of local beam search is the longest while length of h2 is the shortest. The solution length of h1 is between h2 and local beam search.

d. A\*(h1) and beam search both have 3/5 solved while A\*(h2) have 4/5 solved.

|  |  |  |  |
| --- | --- | --- | --- |
| Random size | A\*(h1) | A\*(h2) | Beam |
| 10 | 5 | 5 | 24 |
| 15 | 6 | 6 | 48 |
| 20 | 27 | 7 | 40 |
| 21 | \ | 8 | \ |
| 30 | \ | \ | \ |

Note: these numbers stand for the states this search prints to reach the goal state.

maxNodes = 50, “\” means it did not solve within 50 states.

**4. Discussion**

a. Based on my experiment, A\*(h2) search is better suited for this problem because it always uses the least number of states to solve 8-puzzle. It also finds shorter path and seems superior in terms of time and space. Theoretically, beam search should be good as well but in 8-puzzle, since the successors are only up to 4, it seems not optimal.

b. I found that as the random size increases, the difficulty of solving the puzzle also increases. If the random size is greater than 30, my three algorithms seem almost impossible to solve it. Besides, I think the most difficult part is the storing, comparing and reading from all states visited.