Puzzle game solved with

A\* Algorithm

Ke Ding | CS420 | 04/26/2014

[kding@csupomona.edu](mailto:kding@csupomona.edu)

# Approach

1. Class Plain represents the puzzle state.
2. Interface Action represents the actions which take a state as input and return a new state. There is an Action Factory class which is responsible for creating different kinds of action instances.
3. Interface Heuristic represents the heuristic evaluation, it take a current state and a goal state as input, return the heuristic cost. There is a Heuristic Factory class which is responsible for creating different kinds of heuristics.
4. Class Node represents the node of the search tree. It contains a current state, a heuristic instance and a getCost function which calculates the real step cost from the initial state to current state.
5. Build A\* search tree based on class Node, return the node reaches the goal state as the solution.

# Experimental results

## Experiment Description

1. Run the puzzle program 10,000 times started with a random initial state for each heuristic method.
2. Since puzzles started at random initial state can rarely be solved in a short depth, then the cases solved with depth 1 to 7 are almost zero. I calculate a special state random move 1 to 30 steps from the goal state, run the program 3,000 times using this kind of special state but with different instance for each heuristic method.
3. So the total trials for each heuristic method is 10000 + 3000 = 13,000. Some random moving started from goal state comes back to goal state again, that cause about 300 cases trial start with initial state is goal state which is the depth 0 line comes from.
4. All the experiment codes are in the uTest.ReportTest unit test class.

## Result

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Depth | Distance Heuristic | | | Misplaced Heuristic | | |
| **Search Cost** | **Run Time (milliseconds)** | **# of Cases** | **Search Cost** | **Run Time (milliseconds)** | **# of Cases** |
| 0 | 1 | 0 | 280 | 1 | 0 | 290 |
| 1 | 4 | 0 | 376 | 4 | 0 | 408 |
| 2 | 5 | 0 | 391 | 5 | 0 | 405 |
| 3 | 8 | 0 | 402 | 8 | 0 | 415 |
| 4 | 9 | 0 | 424 | 9 | 0 | 437 |
| 5 | 11 | 0 | 275 | 13 | 0 | 251 |
| 6 | 14 | 0 | 257 | 16 | 0 | 261 |
| 7 | 18 | 0 | 194 | 24 | 0 | 155 |
| 8 | 21 | 0 | 172 | 32 | 0 | 182 |
| 9 | 27 | 0 | 96 | 50 | 0 | 85 |
| 10 | 34 | 0 | 83 | 65 | 0 | 67 |
| 11 | 43 | 0 | 59 | 109 | 0 | 38 |
| 12 | 59 | 0 | 53 | 154 | 0 | 70 |
| 13 | 78 | 0 | 62 | 250 | 0 | 65 |
| 14 | 102 | 0 | 123 | 369 | 1 | 115 |
| 15 | 144 | 0 | 133 | 626 | 2 | 125 |
| 16 | 189 | 0 | 264 | 875 | 3 | 228 |
| 17 | 265 | 1 | 308 | 1443 | 5 | 308 |
| 18 | 328 | 1 | 501 | 2050 | 8 | 523 |
| 19 | 476 | 2 | 591 | 3241 | 14 | 595 |
| 20 | 633 | 3 | 928 | 5244 | 25 | 931 |
| 21 | 894 | 5 | 957 | 8088 | 42 | 917 |
| 22 | 1162 | 7 | 1352 | 11814 | 67 | 1323 |
| 23 | 1639 | 10 | 1107 | 20501 | 132 | 1163 |
| 24 | 2128 | 14 | 1301 | 25632 | 174 | 1322 |
| 25 | 3208 | 22 | 877 | 43809 | 345 | 861 |
| 26 | 4119 | 29 | 851 | 50461 | 411 | 848 |
| 27 | 6112 | 46 | 328 | 82639 | 792 | 336 |
| 28 | 8077 | 63 | 201 | 99222 | 1008 | 216 |
| 29 | 10455 | 85 | 44 | 126953 | 1414 | 49 |
| 30 | 16781 | 147 | 10 | 142629 | 1755 | 11 |
| SUM |  |  | 13000 |  |  | 13000 |

# Analysis

We can see that Heuristic 2(the sum of the distances of the tiles from their goal positions) is do better than Heuristic 1(the number of misplaced tiles), no matter according to search cost nodes or running time.

# Extra finding

## Tail Call Optimization on JAVA

I first used recursion to implement the search tree. Sometime I could get the solution, occasionally I got a [stack over flow] exception. After I changed the main implement using loop, the problem is solved.

So it means when the solution depth is deep, too many recursive function call in and call out lead to stack over flow. Which is a little weird since my recursion is tail call, I thought most modern programming language support Tail Call Optimization which could transport the tail recursion to some kind of loop, I know C++ does. And I looked up materials and find out that java does not support Tail Call Optimization because of some JVM feature. It’s a little disappointed since I would think TCO should be a basic feature for modern programming language.

## Shuffle

About randomly initialize the puzzle state, first I thought about generating random number. Then I realized it is a shuffle problem like we shuffle porker. Thanks for Java Collections.shuffle (List) API make things easy.