Joshua Camacho

Dominick Atanasio

CS 420 – Artificial Intelligence

4/22/18

8 Puzzle Problem with A\* Algorithm

Approach

For this project I implemented A\* in Java using a modified graph search algorithm with two different heuristics. Pseudocode for the algorithm is as follows:

PriorityQueue **frontier** /\* States to be explored, weighted on their evaluation function \*/

Set **explored** /\* States already visited \*/

While (**frontier** is not empty)

**current** = **frontier**.poll() /\* Pull top state from frontier \*/

If (**current** is goal)

return **current** /\* found the solution \*/

**current**.explore() /\* generate child state of current state \*/

For each child of **current**

If **explored** doesn’t contain child state

Add child to frontier

This modified version of graph search uses a priority queue to organize the frontier by lowest evaluation function cost. Note that duplicate states with different paths can end up in the frontier, but because of the priority queue, it does not matter. Two separate evaluation cost functions were used. Note that G(n) is the total path cost leading up to a given state.

1. F1(x): G(n) + Number of puzzle values out of place
2. F2(x): G(n) + Manhattan distance of all values

Simplified EightPuzzle UML Diagram

<Abstract Class>Eightpuzzle

Int[] state  
int heuristic  
int pathcost  
EightPuzzle Parent

Int Fx()

Abstract calcHeuristic()

EightpuzzleH1

calcHeuristic()

EightpuzzleH2

calcHeuristic()

The abstract class Eightpuzzle contains most of the important parts about the state of the node including the array containing the numbers, it’s path cost, and heuristic. Note however, the heuristic is implemented by the child classes, and as such you could create a child class for any heuristic and use the same base class and the driver class for the algorithm.

Results

Below are the results of running 5000 randomly generated puzzle sets through both methods and averaging the results for each depth. Note that if a depth is missing from the table, it was not generated randomly in this test case. Search cost is the number of nodes generated and count is number of nodes that were generated at that depth.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| depth | h1 search cost | h1 time(ms) | h1 count | h2 search cost | h2 time(ms) | h2 count |
| 3 | 9 | 0 | 4 | 9 | 0 | 4 |
| 5 | 24 | 0 | 1 | 15 | 0 | 1 |
| 7 | 35 | 0 | 3 | 27 | 0 | 3 |
| 8 | 48 | 0 | 5 | 26 | 0 | 5 |
| 9 | 68 | 0 | 6 | 37 | 0 | 6 |
| 10 | 113 | 0 | 11 | 52 | 0 | 11 |
| 11 | 169 | 0 | 12 | 61 | 0 | 12 |
| 12 | 251 | 0 | 37 | 91 | 0 | 37 |
| 13 | 422 | 0.095238 | 21 | 131 | 0.095238 | 21 |
| 14 | 606 | 0.169811 | 53 | 144 | 0.018868 | 53 |
| 15 | 1050 | 0.238095 | 84 | 226 | 0.035714 | 84 |
| 16 | 1418 | 0.257143 | 105 | 304 | 0.047619 | 105 |
| 17 | 2499 | 0.476821 | 151 | 440 | 0.072848 | 151 |
| 18 | 3681 | 0.767361 | 288 | 592 | 0.072917 | 288 |
| 19 | 5657 | 1.084806 | 283 | 774 | 0.134276 | 283 |
| 20 | 9315 | 2.157025 | 484 | 1017 | 0.208678 | 484 |
| 21 | 14582 | 3.530786 | 471 | 1458 | 0.282378 | 471 |
| 22 | 21677 | 5.448855 | 655 | 1914 | 0.370992 | 655 |
| 23 | 39078 | 11.07422 | 512 | 2887 | 0.505859 | 512 |
| 24 | 48813 | 14.83943 | 629 | 3687 | 0.739269 | 629 |
| 25 | 89813 | 31.18409 | 440 | 5526 | 1.222727 | 440 |
| 26 | 103827 | 37.98254 | 401 | 7340 | 1.563591 | 401 |
| 27 | 188776 | 81.11892 | 185 | 10650 | 2.491892 | 185 |
| 28 | 235009 | 104.0803 | 137 | 13737 | 3.131387 | 137 |
| 29 | 317967 | 168.2105 | 19 | 20672 | 5.263158 | 19 |
| 30 | 360759 | 178.6667 | 3 | 29515 | 7.333334 | 3 |

Graphs

Findings

Results for both generated nodes and time both show heuristic 2 with a major advantage. At the most often generated depth of 22, heuristic 1 has a search cost of 21677 where heuristic 2 only has a mere 1914. Heuristic 1 appears to scale similar to exponential, while heuristic 2 is much more efficient. Solution depth level of the randomly generated states appear to be distributed normally.