Project 1 CS 170. Introduction to Artificial Intelligence

Ryan Yuzuki Instructor: Dr Eamonn Keogh

ID 861203598

[ryuzu001@ucr.edu](mailto:ryuzu001@ucr.edu)

2-November-2018

In completing this homework, I consulted…

* <https://www.youtube.com/watch?v=dRMvK76xQJI> (review Uniform Cost search)
* <http://www.cs.ucr.edu/~eamonn/cs170/> (to reference Heuristic Search ­PowerPoint slides and project instructions)
* <https://stackoverflow.com/questions/8378797/stdbad-alloc-am-i-using-too-much-memory> (to diagnose bad\_alloc error)
* <https://stackoverflow.com/questions/4184468/sleep-for-milliseconds> (to see how to sleep() in C++)

All the important code is original. Unimportant subroutines that are not completely original are…

* std::iostream (to use cout to print out statements in my code)
* std::vector (to store the puzzle state)
* unistd.h (to use usleep() during the testing phase of my code. Currently unused)

**Code printout, 1st page:**

**#include <iostream>**

**#include <vector>**

**#include <unistd.h>**

**using namespace std;**

**struct problem{**

**vector<int> puzzle;**

**int algorithm;**

**}; // problem will hold puzzle vector and algorithm choice**

**struct node{**

**vector<int> state; // vector of ints to represent state**

**int cost; // cost to get to state**

**};**

**struct node2{**

**vector<int> state; // vector of ints to represent state**

**int hn; // same as node but with hn, gn instead of total cost.**

**int gn;**

**};**

**int numExpanded = 0;**

**int maxNodes = 0;**

**node findCheapest(vector<node> nodes){**

**// if the tree is super huge, like 1 billion or bigger nodes expanded we are in trouble.**

**node cheapest;**

**cheapest.cost = 1000000000;**

**for(int i = 0; i < nodes.size(); i++){**

**Code printout, last page:**

**cout << "Solved: \n";**

**return currNode;**

**}**

**cout << "The best state to expand with g(n) " << currNode.gn << " and h(n) " << currNode.hn << " is\n";**

**printNode2(currNode);**

**nodes = manhattanExpand(currNode, nodes, visited);**

**}**

**}**

**int main(){**

**problem p;**

**p = getInput();**

**p = getAlgorithm(p);**

**node n;**

**node2 n2;**

**if(p.algorithm == 1){**

**n = uniformCostSearch(p.puzzle);**

**printFinalNode(n);**

**}**

**else if(p.algorithm == 2){**

**n2 = misplacedTileHeuristic(p.puzzle);**

**printFinalNode2(n2);**

**}**

**else{**

**n2 = manhattanDistanceHeuristic(p.puzzle);**

**printFinalNode2(n2);**

**}**

**return 0;**

**}**

**Trace of Manhattan distance A\* on**

**1 2 3**

**4 0 6**

**7 5 8**

**Welcome to Ryan Yuzuki's 8-puzzle solver.**

**Type "1" to use a default puzzle, or "2" to enter your own puzzle.**

**1**

**Using default puzzle**

**1 2 3**

**4 0 6**

**7 5 8**

**Enter your choice of algorithm**

**1 Uniform Cost Search**

**2 A\* with the Misplaced Tile heuristic**

**3 A\* with the Manhattan distance heuristic**

**3**

**We will use the following puzzle:**

**1 2 3**

**4 0 6**

**7 5 8**

**using A\* with the Manhattan distance heuristic.**

**The best state to expand with g(n) 0 and h(n) 2 is**

**1 2 3**

**4 0 6**

**7 5 8**

**The best state to expand with g(n) 1 and h(n) 1 is**

**1 2 3**

**4 5 6**

**7 0 8**

**Solved:**

**1 2 3**

**4 5 6**

**7 8 0**

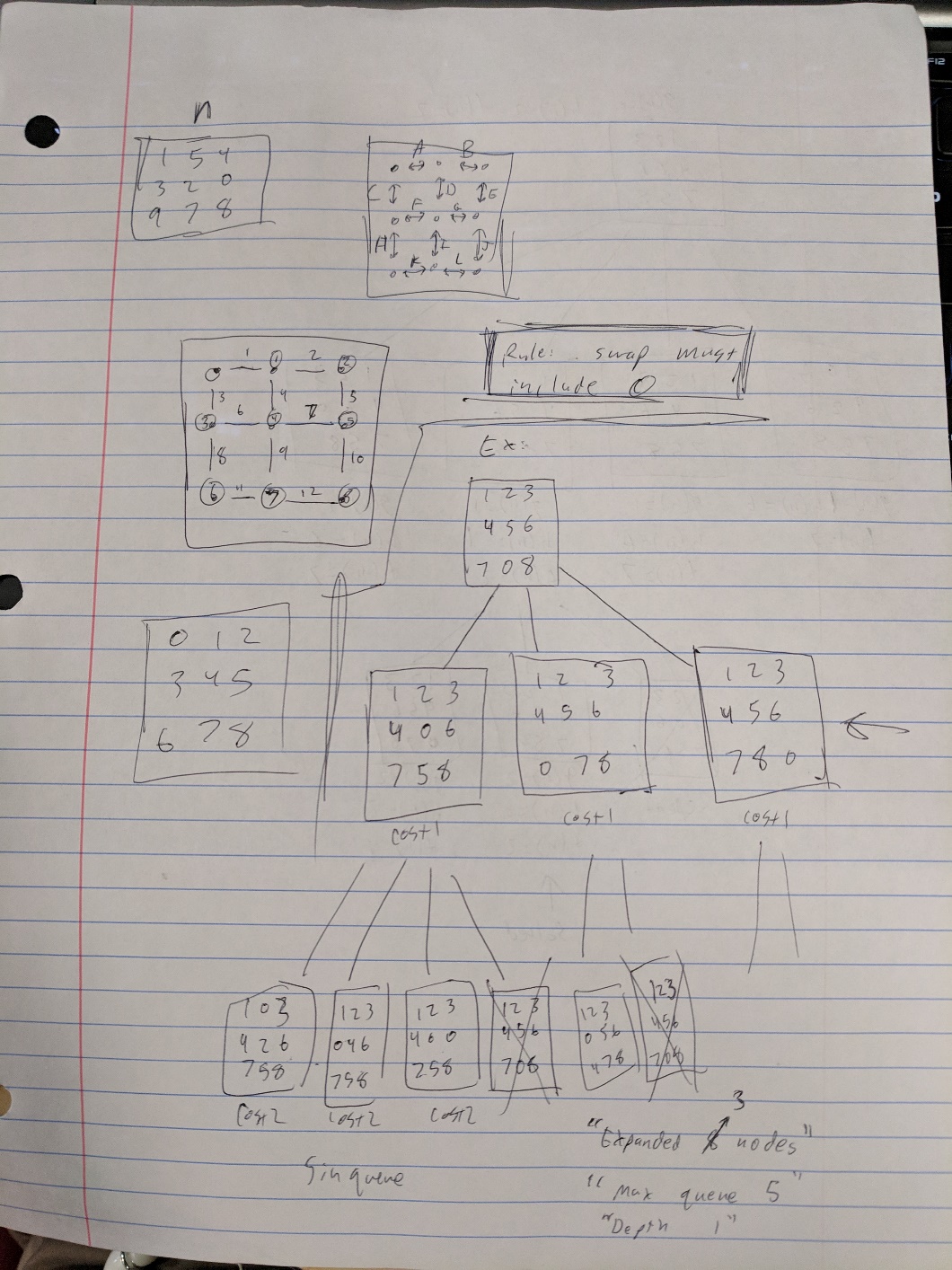
**To solve this problem, the algorithm expanded 2 nodes.**

**The maximum number of nodes in the queue at any one time was 5**

**Depth of goal node: 2**

**SUMMARY**

In general, my code can handle most things that get thrown at it. As for it’s absurd length (currently sitting at 1287 lines of code), it’s just the way I code. I may code inefficiently, but for the most part am able to find the answer with some logical thinking. For example, I created an additional node type, node2 that pretty much served the same purpose as node, except it held g(n) and h(n) instead of total cost. This forced me to rewrite several functions that I had already written. As for the code itself, it behaves as though you would expect it to be. First, a quick explanation. I realized while programming this project that each “swap” that the puzzle can perform had to include the 0 (to represent the blank space). With 9 spaces, this will leave us 12 possible swaps. With the additional requirement that each swap had to include 0, this was considered by hard coding a swap algorithm determined on each individual location of the 0 character. Each swapN (N being a number) in my code was taken from the following diagram, which I drew out to not become confused while programming the project.



In addition, the illustration above shows a trace of the most basic problem using uninformed search. My code would check to see before enqueueing a node if it was a duplicate. If it was, it was discarded. This is shown by the crossed-out nodes. Uninformed search would result in 3 nodes being expanded (8 was a bug) with a max depth of 5 (shown by the uncrossed out nonoriginal nodes). This matches my program output as well. Let’s compare this problem with A\* tile (1 expanded, max queue 3) and A\* Manhattan: (1 expanded, max queue 3).

For this problem, there is no difference between A\* tile and A\* Manhattan. We can show the difference by choosing a more difficult puzzle: (Depth 5)

**1 2 3**

**4 8 0**

**7 6 5**

Uniformed: 65 expanded, max queue: 51

A\* tile: 8 expanded, max queue: 11

A\* Manhattan: 5 expanded, max queue: 7

Here we can see that there is a large difference between Uninformed and A\* search. There is a smaller difference between A\* though Manhattan pulls ahead of tile in this situation. Let’s try with a larger problem: (Depth 22)

**4 6 0**

**2 7 3**

**5 8 1**

Uninformed: (Took 10+ minutes calculating then I just killed the program at that stage)

A\* tile: 6865 expanded, max queue: 3818

A\* Manhattan: 517 expanded, max queue: 335

From this, we can create a rough table:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Uninformed | A\*Tile | A\*Manhattan |
| Depth 1 | 3 expanded  5 max queue | 1 expanded  3 max queue | 1 expanded  3 max queue |
| Depth 2 | 12/9 | 2/5 | 2/5 |
| Depth 3 | 16/13 | 4/6 | 3/6 |
| Depth 4 | 22/19 | 5/6 | 4/6 |
| Depth 5 | 65/51 | 8/11 | 5/7 |
| Depth 6 | 27/18 | 4/6 | 4/6 |
| Depth 8 | 277/178 | 26/21 | 10/12 |
| Depth 12 | 2011/1282 | 105/78 | 19/19 |
| Depth 14 | 5606/3428 | 342/221 | 39/32 |
| Depth 22 | ? | 6865/3818 | 517/335 |

And of a graph of depth vs expanded \* max queue

(If you want to run them yourself to verify)

Depth 12: 236/714/580

Depth 14: 236/701/584

As you can see, Uninformed search is dominating the chart. The sheer inefficiency of uninformed search results in such a large number that the other numbers cannot be seen. According to my collected data, Uninformed search is approximately 20x worse than A\*Tile and 200x worse than A\*Manhattan.