Dr. Goldsmith

CS 463G

September 18, 2018

Programming Assignment 2: A* On the Megaminx

In this programming assignment, we were tasked with expanding on programming assignment 1, and make an algorithm that solves the Megaminx. Specifically, we will be using the A* algorithm to solve the Megaminx, which will always find a solution. In my version of the code, I was able to use A* algorithm to find solutions for when my Megaminx is randomized between 1 and 13 times. For K values greater than this, my computer ran out of memory and proceeded to kill the process.

My code that is turned in is an adaptation to the code I turned in for part one. The representation for the Megaminx is the same (see assignment one for description), but I did add some code to the main file for running A* and adding in timing capabilities. The main addition to this project is the astar.py file, which contains the representation for A*. At first I did struggle a bit with coding up A* properly and correctly, however thanks to Red Blob Games, I was able to fully understand and grasp how to code the A* formula. My code starts off with a new Node class. This node class contains data information such as the actual Megaminx, the priority weight value from f, and the clockwise and counterclockwise movements. Since this Node class is added to a priority queue later on, I also had to readjust the comparisons for the Node, and have them compare based off of the priority.

In the A* formula, I initialize a priority queue and a start node, with priority value of 0. Every time a new Node is created, it also creates a new Megaminx in memory. This prevents the same Megaminx from being called each time, which led to many errors in code in my first version. This node is then added to the queue, and two dictionaries are created. These dictionaries use the Megaminx as a string as the key, and the parent lists which one it came from, also as a string, and then the cost records the cost to this node so far. We then enter the loop, which is execute while the queue is not empty, basically ensuring it continues until solution is found. In this loop, we start out by getting the lowest priority item in the queue. This item is tested to see if it is solved, and if it is it breaks out of the loop. If not solved, it finds all the neighbor nodes, based on rotating the Megaminx on that side, and add 12 to the number that is expanded. The code loops over these neighbor nodes, and calculates the cost of this node, which is cost of the current Megaminx plus one. If this Megaminx is not in the dictionaries or if the cost is less than a previously found cost, it computers the priority f value with the heuristic, creates a new node, and adds this node to the queue. The parent dictionary is updated with the information of the current as parent, and the index number showing which direction it came from.

After it finds solution, it finds the path, starting with a new Megaminx, until it equals the start Megaminx passed into A*. It does this through backtracking through the parent dictionary, and finds the correct parent for each that leads to the parent key that has a None value. Since this is the solution from start to finish, it is reversed, and this is the path to solve the Megaminx.

The heuristic I used was the number of pieces out of place, divided by 10, and then the floor of that. This is admissible because it always predicts a lower estimate for how many moves are left. For example, for the number of k we rotate, it should be seen that we should not have a heuristic higher than the k. This is because the heuristic should be a lower estimate of the number of moves left to solve. So if the heuristic for the initial mixed Megaminx is higher than k, then this is not a good heuristic to use, as it over estimates. In testing, I have not found an instance where my version of the heuristic over estimates. I chose 10 based on it being the number of parts that actually move in a rotation.

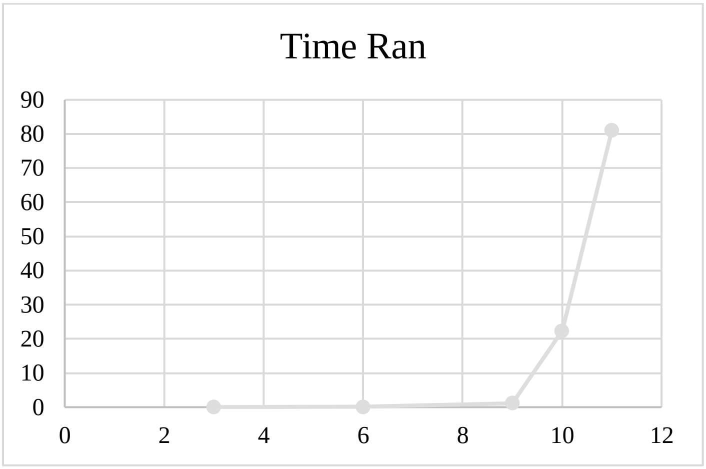
In testing my code, I found the following solutions for the k values. Each k value was ran three times, minus k = 13 due to the large time it took the first time. The seconds and number of nodes are recorded for each K.

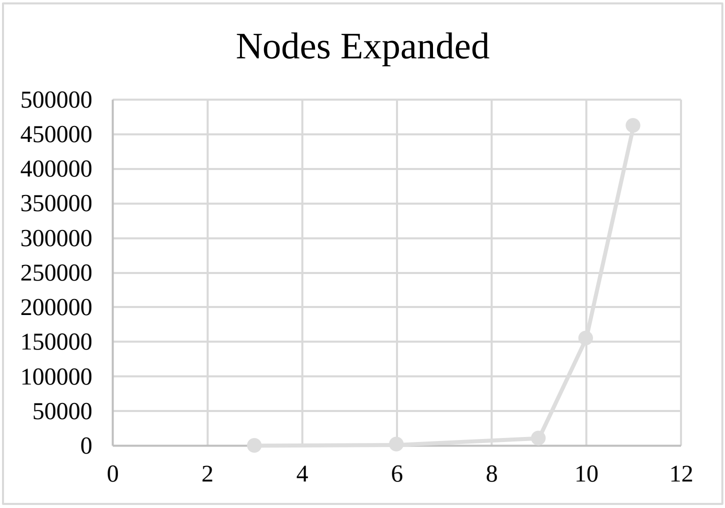
K-Value	**Seconds1**	**Nodes1**	**Seconds2**	**Nodes2**	**Seconds3**	**Nodes3**
3	0.009125	60	0.010332	72	0.013153	108
6	0.160344	1668	0.106244	1188	0.117172	924
9	0.587622	6024	1.090799	9924	1.882003	15444
10	37.94443	291888	22.449777	141468	6.631533	32136
11	120.453618	835080	80.057309	319500	42.467827	232584
13	11094.1823	6099000	N/A	N/A	N/A	N/A

That resulted in average second and nodes for each to be:

K-Value	**Avg. Seconds**	**Average Nodes**
3	0.01087	60
6	0.12792	1668
9	1.186808	6024
10	22.3419133	291888
11	120.453618	835080
13	11094.1823	6099000

This shows that as we increase K, the average seconds to solve increases as well as the average nodes expands, as expected. Graphing this out matches how we expect A* algorithm to work. A* algorithm has a complexity and time requirements for this problem that is $O = (12^d)$, where d is the depth of the shortest path, which in this case should always be k rotations since we are rotating the opposite way it was mixed. The two graphs below show how it grows exponentially.





In this assignment, I learned a better grasp on search algorithms, priority queues, dictionaries, and overall how the A* algorithm works. I learned that while the A* algorithm isn't the fastest search algorithm, it does return the shortest path to the solution. However, it will only do this if it has an admissible heuristics. To compile and run my code type *<python3 main.py>*, and follow the instructions to mix and solve.

Works Cited

Implementation of A*. *Red Blob Games.* <https://www.redblobgames.com/pathfinding/a-star/implementation.html> Published 6 July 2014. Accessed 14 September 2018