Noah Greski

Algorithms Final Project

Rubik’s Cube Algorithms Analysis & METAL Sidewalk Data

April 29, 2024

In this final project for the Algorithms course I studied three different algorithms. Two of which were Rubik’s cube solving algorithms and the third is a modified implementation of Dijkstra's Algorithm for finding the shortest path. When implementing the Rubik’s Cube Algorithms, I found two 2x2 Rubik’s cube solving algorithms that use different techniques to solve the cube, one is by layers and the other is by swapping the corners. I performed an empirical analysis on these two algorithms to see which one is more efficient. For Dijkstra's Algorithm I modified it with a constraint to simulate roads that have sidewalks. To do this, I made it so that vertices within 2 miles of each other have “sidewalks”. Obviously in some cases this is not true but that is why the data is simulated. I then ran these outputs through the METAL graphs and then could see the shortest path from place to place only using roads with sidewalks.

Before getting into the details of each algorithm in this project, there were a few motivations for this project. First, for the Rubik’s Cube algorithms. The first time I ever heard the word algorithm was in a YouTube tutorial on how to solve a Rubik’s Cube. So when starting this project that instantly came to mind. I was not sure on how I wanted to implement this into my project but decided the best way to do it was finding algorithms that other people created and doing my own analysis on them. Next for the sidewalk data, my main motivation was to find the shortest path from place to place only going on roads with sidewalks. The reason to simulate this

type of data would be to make routes to run or walk from place to place with roads with sidewalks, for safety reasons. And obviously running or walking on a highway is illegal.

For the Rubik's Cube algorithms, I did an entire empirical analysis on these two solving algorithms. That analysis is shown below. It can also be found in analysis.pdf in the repository.

An Empirical Study of X and X Rubik's Cube Solving Algorithms

Noah Greski
nj06gres@siena.edu

April 29, 2024

Abstract

In this study, I present and analyze timings and operation counts for two rubik's cube solving algorithms. These algorithms include X1 and X2. These algorithms are programmed for 2x2 cubes rather than a more standard 3x3 cube. This was done to simplify the process of this analysis and remain in the scope of this project. Both solving algorithms and cube scrambling methods are done within several Java programs.

1 Introduction

This study focuses on two rubik's cube solving algorithms in a few different ways. The computing environment section gives information about external factors that may affect the timing results of these algorithms. Both algorithms have different techniques for scrambling and solving these cubes. Testing will be done to compare which algorithm is more efficient overall and why that is. The testing will calculate the number of moves done to solve the cube and the elapsed time for the program to execute the solving algorithm. Tests, results and conclusions will be discussed on why one is more efficient than the other or vice versa. We will be comparing X1 (Layers), a Rubik's Cube solving algorithm that solves the cube layer by layer, and X2 (Corners), a Rubik's Cube

algorithm that solves the algorithm that focuses on solving the cube by swapping the corners.

2 Computing Environment

For this study, a Java program is used to run these two algorithms. There are no inputs by the user, cube scrambling methods and solving algorithms are done by the Java program. The elapsed time, number of moves, and a simple visual representation of the cube on several steps of the solving process are output by the program.

All runs are done using the following Java version:

```
openjdk version "17.0.8" 2023-07-18
OpenJDK Runtime Environment Temurin-17.0.8+7 (build 17.0.8+7)
OpenJDK 64-Bit Server VM Temurin-17.0.8+7 (build 17.0.8+7, mixed
mode, sharing)
```

The computing environment is a Dell PC with a Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz running on Windows 10 Enterprise. The L2 Cache size is 2048 KB and the L3 Cache size is 16384 KB.

3 X1 (Layers)

X1 is a Rubik's Cube solving algorithm that was created by Nabil-1999 and can be found on Github at https://github.com/Nabil-1999/rubik-s-cube. This algorithm was implemented into my own Java program and can be found under the "rubiksSolver1" folder in my repository. I have slightly altered this program for this empirical study, changes can be found in the code paired with "//added by Noah" comment.

X1 is a complex algorithm and is not worth adding the code to this study. The X1 solving method solves the cube layer by layer. First by solving the bottom layer, middle then top layer. Different algorithms are used to solve each layer, combining into one algorithm to solve the cube. First corner placement, edge placement and edge pairing are done on the first two layers. Then OLL (Orientation of Last Layer) and PBL (Permutation of Last Layer). These orient the pieces of the top layer and place the pieces of the top layer into the correct positions to solve the cube. We will call this algorithm X1 (Layers).

3.1 X2 (Corners)

X2 is a Rubik's Cube solving algorithm that was created by ljz112 and can be found on Github at https://github.com/ljz112/rubiks-cube. This algorithm was implemented into my own Java program and can be found under the "rubiksSolver2" folder in my repository. I have slightly altered this program for this empirical study, changes can be found in the code paired with "//added by Noah" comment.

X2 is a complex algorithm and is not worth adding the code to this study. The X2 solving method is known as the Old Pochmann Method. This method is mainly used for blindfolded solves. It uses one specific algorithm to swap corners until they are all in the correct position. In the case of a 2x2 cube, this solves the cube, all pieces on the cube are corners. We will call this algorithm X2 (Corners).

3.2 Tests

To test the X1 and X2 rubik's cube solving algorithms, the program was run 20 different times with three different amounts of cubes being solved for each algorithm. The program loops through solving one rubik's cube, for testing purposes I did a loop with 10, 100 and 1000 solves for the cubes, 20 times each. 120 total program executions. The reason for increasing the amount of cubes being solved per execution was to decrease the weight of the outliers on the data. The average number of moves and elapsed time for each program execution were output.

3.3 Expectations

There are a few expectations I have for both of these two Rubik's cube solving algorithms. The X1 algorithm solves layer by layer using several different algorithms. My expectation is that this allows for X1 to be more efficient than X2. As X2 uses the same algorithm over and over to solve the cube. X1 being more efficient meaning its elapsed time and numbers of moves will be overall less than X2. On the other hand, X2, I predict will have less variance in the results, as it always uses the same algorithm. X1 will have more variance in the number of moves because it can skip certain steps when it sees different patterns using different algorithms, but sometimes cannot skip these steps, resulting in a worse number of moves. The way I solve a Rubik's cube is layer by layer like in the X1 algorithm. When solving I am able to skip steps depending on the original scramble of the cube. The X2 algorithm has no steps, the same moves are repeated over and over. Overall, I predict X1 will be more efficient than X2.

3.4 Results

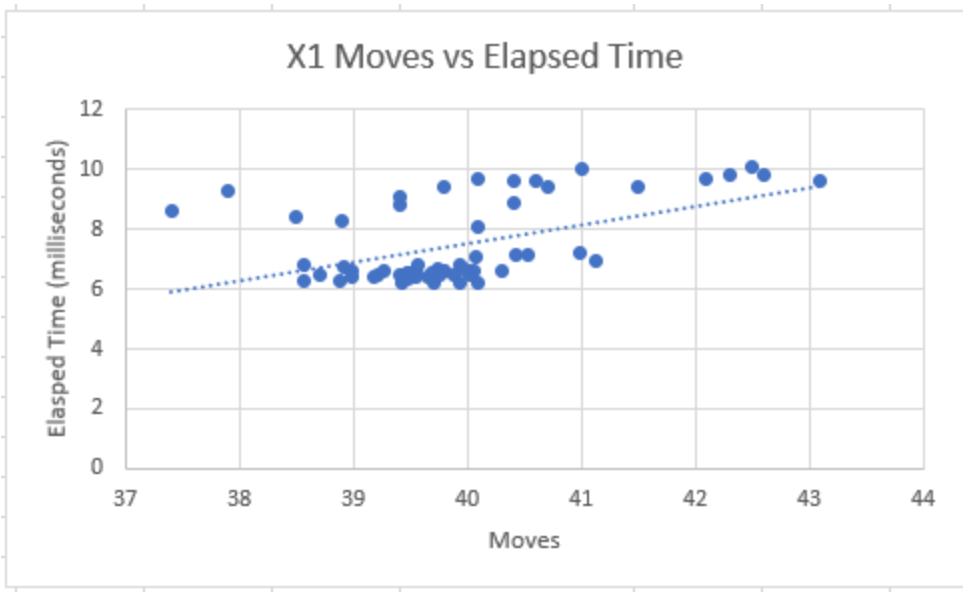
The raw results for tests can be found in the files X1_10, X1_100, X1_1000, X2_10, X2_100, and X2_1000 in the repository. The format is `(number of cubes) (number of moves) (elapsed time)`. The number in the file names indicate the number of cubes being solved per execution. The first number is the number of cubes solved per execution. The second number is the average number of moves made to solve the X number of cubes. The third number is the average elapsed time of solving each cube, in milliseconds. Graphs and charts below display the results as well.

moves		
	X1	X2
mean	39.87233	486.9119
stdev	1.084459	11.67179

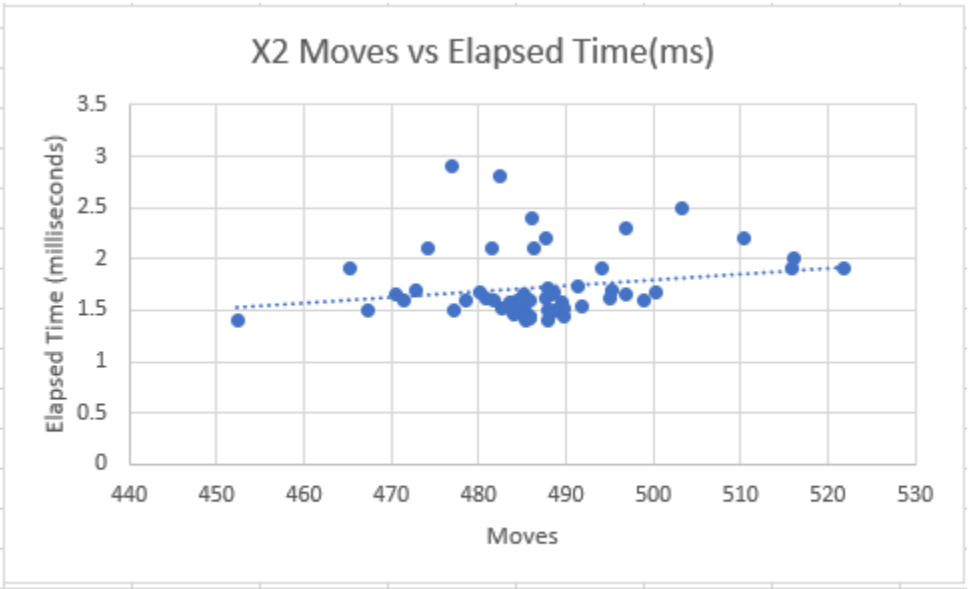
time		
	X1	X2
mean	7.481483	1.721667
stdev	1.340953	0.335515

X1	10	100	1000
mean moves	40.435	39.612	39.70775
mean time	9.28	6.701	6.46345

X2	10	100	1000
mean moves	488.11	486.461	486.1735
mean time	2.045	1.6275	1.4925



correlation
0.500647



correlation
0.198689

3.4 Discussion

There are 6 charts/graphs for the results of testing these two algorithms. The first two charts are a moves chart and a time chart, showing the mean and standard deviation for number of moves and elapsed time. The next two charts are the mean moves for X1 and X2, at 10, 100, and 1000 cubes being solved. The last two charts are scatter plots with trend lines showing number of moves vs elapsed time for X1 and X2. The correlation is also shown here.

First, we will go over the number of moves results of these two algorithms. For X1 (Layers) the mean number of moves to solve a cube was 39.87 moves. The mean for X2 (Corners) was 486.91 moves. In terms of moves, X1 is more efficient than X1. This is most likely because X2 uses the same algorithm over and over to swap the corners until the cube is solved, causing lots of moves to be made. X1 uses several different smaller algorithms when going layer by layer and recognizes patterns quicker to solve the cube in fewer moves.

Next, we will discuss the elapsed time for these algorithms. X1 (Layers), had an average elapsed time of 7.48ms. X2 (Corners) had an average elapsed time of 1.72ms. In this case X2 is more efficient than X1 by the elapsed time. There could be a few reasons for this. First, the program in X2 is just more efficient than X1, not meaning the algorithm is better or more efficient, it just takes less time to run. On the other hand, X2 could just be a more efficient algorithm time wise on a computer. X2 has less code to run through because it performs the same algorithm to swap the corners.

Next I looked at how each algorithm did with solving a different number of cubes when solving,10, 100 and 1000. I did this to check if major outliers could be skewing the data. It looks like in the case for both algorithms, moves and time, the data was not skewed much when solving a lower amount of cubes. This was just to ensure that the averages were real averages, not just a one time high or low average. All the 1000 values are close enough to the actual averages I calculated for all the data points are not being skewed by outliers.

Lastly, we will discuss the last two graphs. First is X1 (Layers), elapsed time vs number of moves made to solve the cube. This graph has a 0.50 correlation coefficient. Meaning as the number of moves increase, the elapsed time also increases. The 0.50 correlation is relativity positive so we can say that there is definitely a relationship between the number of moves and elapsed time for X1. On the other hand in X2 (Corners), the correlation for this graph is much lower at 0.199. Meaning as the number of moves increases the time does not increase as much. In some cases it will because there is a weak positive relationship here. This means that X2 could be more efficient

than X1 because it solves the cube in the same amount of time regardless of how many moves it takes to solve the cube.

4 Conclusions

There are several conclusions that we can draw from the tests, results and discussion of the X1 (Layers) and X2 (Corners) algorithms. First we must decide which Rubik's algorithm is more efficient. There are several factors we must consider. First we will consider the number of moves and elapsed time. X1 had a much lower average than X2 in the number of moves. However X2 had a lower average elapsed time than X1. These don't give us a clear answer on which is more efficient. Now we can consider the correlation of elapsed time and number of moves in each algorithm. X1 has a higher correlation between these two variables than X2. Meaning X2 on average does not increase elapsed time as much as X1 does as the number of moves increases.

After considering those three factors (elapsed time, number of moves and the correlation between the two), we can conclude that X2 (Corners) is more efficient than X1(Layers) in this specific case. Meaning on paper X2 is better at solving Rubik's Cubes in a Java program, in a specific computer environment. There are other circumstances where X1 (Layers) can be better than X2 (Corners). For example, if these exact algorithms and data points were transferred to someone actually solving a 2x2 Rubik's X1 would be significantly better. When solving a Rubik's cube in real life an average of 39 moves (X1) beats an average of 486 moves (X2) everytime. Unfortunately when a computer is solving the cube, the number of moves does not matter as much. Additionally from my experience and background knowledge on Rubik's Cube solving, the layer solving technique is a slower technique but is easier to learn and perform than a corner solving technique. The corner swapping algorithms are usually much faster than the layers and some can be done blindfolded (as stated at the beginning of this analysis). There are several ways to interpret which algorithm is more efficient (either in a program or in real life). In this specific case X2 is more efficient than X1.

Now onto the METAL simulated sidewalk data. I used a constraint where vertices only within 2 miles of each other. This was to simulate the sidewalks, like in a city or populated area there are a lot of intersections, so there would be sidewalks on those roads. In some cases this data is not very accurate at all, which is okay. Only a few edits were made in Dijkstra's Algorithm to simulate sidewalks. The main change was for the algorithm to only consider edges and

vertices that are within 2 miles of each other. And then to change the main method to make sure that each edge has a sidewalk before continuing its search for the shortest path.

I did some testing with my new version of Dijkstra's algorithm. I mostly used the siena 50 mile radius graph to test this algorithm. I did several tests from Siena to different places. All the tests I did can be seen in the sidewalk_testing.txt. Most of the further places would not find a path at all which makes sense. The most successful test I had was from Siena to Scotia (which is the town I live in). It found a path in about 35 miles from Siena to Scotia. The idea is that I could run this path from Siena to my house. 35 miles is a bit too long for me and there is definitely a shorter path that involves back roads. Overall the algorithm works in some cases, but there could be improvements.

There are a few conclusions I can draw from this project. First is that Rubik's Cube algorithms in a Java program do not represent the actual solving of a cube. In the case of this project the corner swapping algorithm was more efficient. However, with only considering the moves in real life, the layers would be much faster. Additionally, the sidewalk data is also simulated and is not directly correlated to real life, as some of the routes I tested would not work. So one main conclusion we can draw from this is that we need to do more testing with real data for both sides of this project. In the case of the Rubik's Cubes we could gather data based on these two algorithms and see which one is more efficient in real life. Additionally with the sidewalks we could gather data on what streets actually have sidewalks and integrate this into Dijkstra's algorithm.