Austin Stau

Professor Graham

CSC 482

1 December 2021

Lab0xFF

**Greedy Heuristic** -

The first algorithm I am using is the greedy heuristic algorithm. First, I will show correctness of the algorithm itself. This algorithm takes it first point at 0 and moves to the closest point and compares from that point where the closest point is until returning to back to vertex 0. For this I’ve made two functions that creates a circular Euclidian graph. The first one creates the vertex points and the second creates the distances between everything. As you have shown and can be seen on a graph, the point around the edges creates the shortest distance moving in an either clockwise or counterclockwise direction depending on the preparation was done for the algorithm. Testing on this circular graph makes it easy to tell which direction the algorithm should go and to verify where the algorithm’s shortest path should be.

With help from <https://www.geeksforgeeks.org/travelling-salesman-problem-greedy-approach/>, I’ve used a modified algorithm because this only lists in C++/Java/C#. So, I’ve had to do slight modifications to the code to make it work in C.

First, testing with a low N, for this instance N = 6 to check validity by eye.

Calendar

Description automatically generated

Matrix with -1 denoting self-edge:

A black screen with white text

Description automatically generated with low confidence

Looking at graph:

A picture containing calendar

Description automatically generated

Looking the graph we can see that this route is correct following a counter-clockwise around the circle.

These are the points around the circle and the radius being 100. The first print out is the creation of points, where everything moves in clockwise fashion. Then I mix up the point’s vertex numbers, this is what is shown in the second block of print out. Then by eye we can see the route follows the counterclockwise direction around the circle giving us the min distance of 600 which is correct because the shortest distance between each point is 100.

A little higher N = 10, with the same radius, but still able to see that the algorithm is moving in the circle.

A picture containing calendar

Description automatically generated

Matrix:

A picture containing text, keyboard

Description automatically generated

Here you can see that distance is the same or close to it. These are float numbers so decimals do have an effect. But we can see that the route the algorithm is moving is counterclockwise again, we can see this as well when looking at a graph:

A picture containing scatter chart

Description automatically generated

Now let try a large number at N = 40, here is the list of vertices mixed up:

A picture containing calendar

Description automatically generated

The result of the route:

A picture containing text, scoreboard, meter

Description automatically generated

You can eyeball this fairly easily, we can look at our halfway point at vertex 33 sitting at spot 20, although this would be spot 21, but the 0 is added in front. Looking at vertex 15 sitting next to 0 we have (-15.40, 99) so this is as small on the negative x side we go and very close to 100 on the y-axis. Looking at 27, which is at the end of the route it is a copy of vertex 15, but with 2 positive axis.

So why does this algorithm find the correct solution for circular Euclidean graphs? Because of how these graphs are set up, not so much the algorithm itself. The graph is purposefully made so that the nearest point is always the point along the outside of the circle. Thus, when the algorithm is working correctly it will follow along the circle in its route and find the minimal route.

For the first time test I have done the low N up to 30. To get an idea of where this stands, I didn’t include a matrix because this would have been large.

Graphical user interface

Description automatically generated

It went up to 30 N, but my doubling rations were not hitting the expected exactly, but still a good result of performance. My expected doubling ratio I used was n^2\*log2(n), I used n/2 calculating my doubling ratio. Now for the doubling starting at 50 and going until I cannot anymore.

Looking at graph for this we can see the increase

Chart, line chart

Description automatically generated

The top graph is what my n to 30 looks like. I went up to N=20 and the timing had to be multiplied by 100,000, so I could get legible results, but the data stays intact. So, my first at N=1 the time would 0.49 and I multiplied each one after that by the same amount. The bottom graph is n^2\*log2(n) and you can see that the graph for n^2\*log2(n) is quite abrupt about going straight up, whereas my curve is more evened out.

Chart, line chart

Description automatically generated

So, we can see that

Text

Description automatically generated

I could get about this far before Visual Studio Code started crashing. This was using a large amount of ram to compute. But we can see that my experimental doubling ratio was closer to my expected ratio than my previous N increase.

Looking at graph for this one we have

Chart, line chart

Description automatically generated

We can see it is increasing like our actual n^2\*log2(n), but with the exception of it is a little more flattened whereas the original curve shoots up fast.

**Brute Force**

First off I got my start with this code here <https://www.codesdope.com/blog/article/generating-permutations-of-all-elements-of-an-arra/> I used this to find permutations of an array of integers and then ran them through the graph. Here are some correctness tests:

Calendar

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated with low confidence

Text

Description automatically generated

I could not figure out the routes, although my code seems to be correct, I think there is an issue with it. Although the cost for the trips seems to be correct.

Text

Description automatically generated with medium confidence

My doubling ratios are little higher than the expected rations I used T(n) = n! my ratio I used was n!/n/2!. This algorithm an expected n! time complexity because of how the algorithm process’s each vertex. A permutation of let’s say a small amount like 5 gives 120 different permutations to work through. Although my algorithm takes all self-edges and disregards them this still takes time to go through each and every permutation. So the runtime is multiplied by N times so I technically run through 120 iterations of finding the lowest cost amount.

Looking at a graph of a factorial

A picture containing shoji

Description automatically generated

We can see it goes straight up and fast, much faster than the last greedy algorithm.

Here are my values on a graph:

A picture containing shoji

Description automatically generated

So, my graph looks very similar to the plotted curve of n!, because it does rise so quickly.

Let’s take a look at solution quality exact vs heuristic

**SQR**

For the SQR I used the Greedy vs the brute force method and I used random Euclidian graphs. I started at N = 4 and went to 13, as past 13 the exact algorithm started taking quite a long time, and with N\_max at 100.

A picture containing table

Description automatically generated

As you can see my average SQR seems to be dipping after each run and the exact vs average seem to get a larger margin between them as N gets bigger.