Algorithms II
Final Project Report

Asra Ahmed - aa03422
Muhammad Ajlal Bawani - mb03748
Ali Shujjat - as03856

December 8, 2019

Abstract

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities exactly once, starting from the origin (e.g. the hometown) and return to the same city. The challenge is that the traveling salesman wants to minimize the total length of the trip. We attempted to solve this problem by using three different implementations: brute-force, a greedy approach and dynamic programming.

Contents

1 Introduction

The Travelling Salesman Problem (TSP) is finding the shortest possible route, given a list of cities and the distances between them, that visits each city and returns to the original city.
It has many applications in the field of logistics, planning and the manufacture of microchips. It is also relevant in astronomy.
The TSP belongs to a group of problems categorized as NP-Hard. No polynomial time solution is known for such problems. Hence, it is possible, and in fact, expected, that the time complexity will increase with an increasing number of cities.

1.1 Defining our problem

The travelling salesman problem asks a simple question:

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"[1]

We hope to solve this problem.

1.2 History of TSP

Historically, the mathematics related to the TSP was developed in the 1800's by the Irish and British mathematicians, Sir William Rowan Hamilton and Thomas Penyngton Kirkman.
In 1857, Hamilton created the Icosian Game, which consisted of a pegboard with twenty holes. Each hole had to be visited only once, no edge to be visited more than once, and the final point had to be the same as the origin. For this reason, this kind of path came to be known as a Hamiltonian circit.
However, the general form of the TSP was first studied by Karl Menger in Vienna and Harvard in the late 1920's or early 1930's.
Applegate, Bixby, Chvatal and Cook solved the TSP using 7,397 and 13,509 cities in the United States, in 1994 and 1994 respectively. In 2001, they found the optimal tour of 15,112 cities in Germany. Later in 2004, the TSP of visiting all 24,978 cities in Sweden was solved. The result was a tour of length of approximately 72,500 km, and it was proven that no shorter tour exists. This is currently the largest solved TSP. [5]

1.3 Setting up the problem

1.3.1 The Input

We defined a list of over 50 cities and created a nodeGenerator function that takes as input the number of cities, and creates a list of nodes, from which we generate a graph containing distances from each city to another. The distances between the cities is calculated by a Google Maps API in real-time for each city. This graph is then passed to the functions of the algorithms that use it as input.

1.3.2 Constraints

For a fair comparison between the algorithms we are testing, we used some constraints:

- Using the same laptop
- Using the same dataset
- Using the same IDE
- Running all of them all together in a single file

1.3.3 Methodologies

The three different algorithms we successfully implemented are

- Brute-Force
- Greedy (Get Nearest Neighbour)
- Dynamic Programming

1.4 Expectations

Before any testing, and from our knowledge of the theory, we expected our Brute-Force algorithm to take the longest time, and the Dynamic Programming algorithm to be the fastest.

2 Our Results

Here we note the time taken in seconds(s) for the program to run till completion.

2.1 The Data Collected

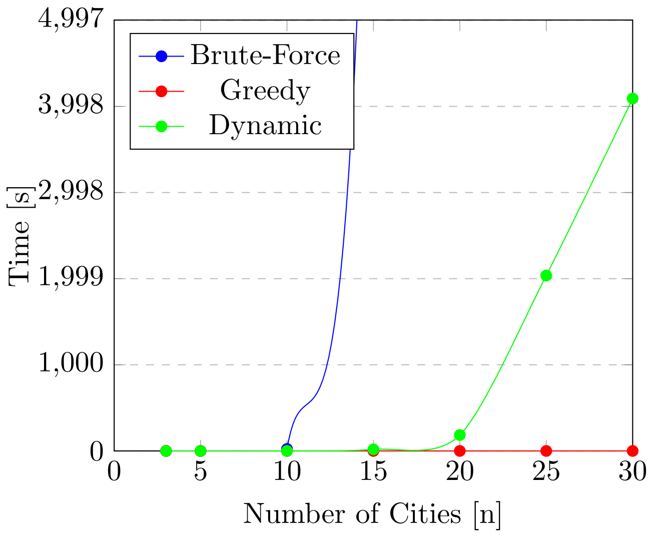
In addition, the time for the creation of the graph and matrix was also noted. However, it must be noted that since distance are being calculated in real-time using the Google API, the times recorded are much higher than expected, as it it also depends on the strength of the internet connection being used.

No. of Cities	Brute-Force	Greedy	Dynamic
3	0.0016	0.0009	0.0030
5	0.0014	0.0000	0.0028
10	23.0051	0.0016	0.0688
15	11520	0.0010	17.8831
20	x	0.0010	184.9863
25	x	0.0020	2034.8235
30	x	0.0040	4089.4320

No. of Cities	Time for Graph	Time for Matrix
3	4.2958	0.0000
5	12.5190	0.0000
10	99.2830	0.0001
15	109.285	0.0020
20	305.128	0.0078
25	593.785	0.0102
30	1045.983	0.1065

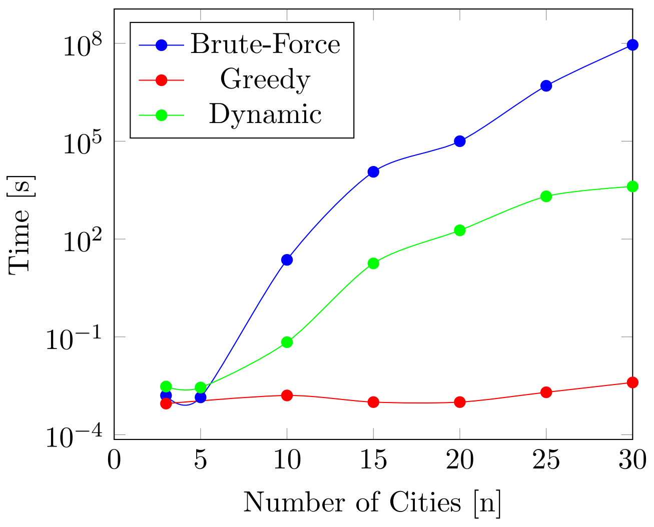
2.2 Comparisons

Runtime complexity for Brute-Force, Greedy and Dynamic Algorithms

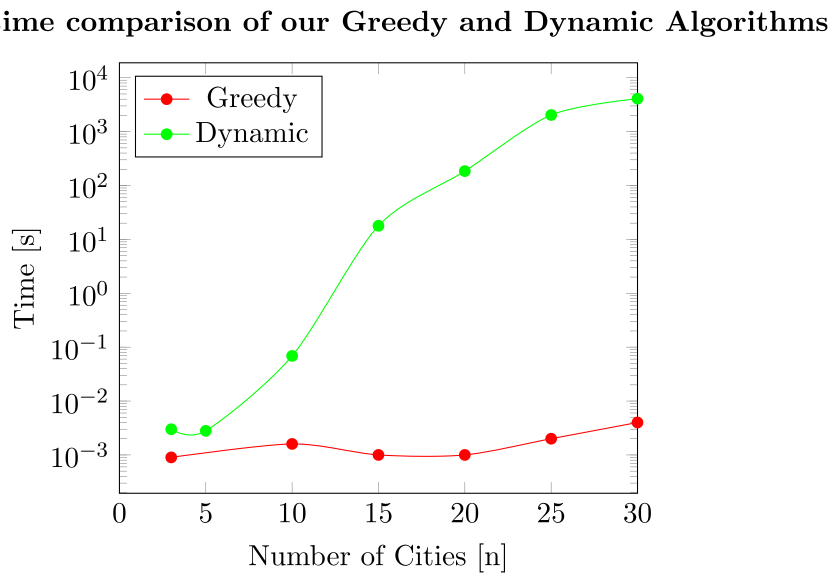


The time scale was changed to a log based scale to help us better visualize and analyse the running times, and the results were plotted on the graph below.

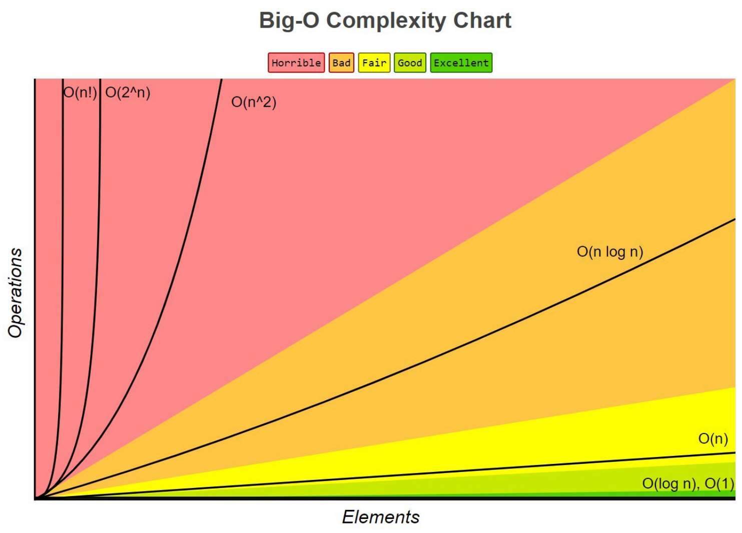
Runtime complexity for Brute-Force, Greedy and Dynamic Algorithms



Further more, the time taken by the Brute-force algorithm is very large, so to give a better sense of the difference between just the times taken by the Greedy and Dynamic algorithms, a separate log based graph is shown below.



Below, we have a graph showing relative complexities, and their ranking, but since we are dealing with complexities in the superpolynomial region, it can not be seen as clearly, as our graphs above are only part of the "horrible" zone shown below.



2.3 Data Analysis

2.3.1 Brute-Force

Brute-force is the most simple and direct way to approach the problem, because it simply goes over all the possible combinations and then chooses the solution with the lowest cost. In this way, an optimum solution is guaranteed.
Brute-force will give (n-1)! possible solutions for an input of n cities. It is (n-1) because we have the origin city, so that is fixed. For each subsequent city chosen, the number of available cities to choose from decreases by 1. For e.g. if we have 3 cities, we will have (3-1)! possible combinations. This is why the time complexity of a brute force algorithm for n cities is O(n!) in all cases.
This is superpolynomial time, which Khan Academy describes as 'often require(ing) more time than available in the universe, even for relatively small input sizes.'[6] This holds true, because when testing the algorithms against an input of 15 cities, the competing algorithms all delivered a solution within minutes, whereas Brute-force took well over 2 hours.
It's space complexity in this case will be $O(n^2)$, because we need to store each potential solution in order to compare later.

2.3.2 Get Nearest Neighbour

The nearest neighbor algorithm was one of the first algorithms used to solve the travelling salesman problem (TSP). In the algorithm, the salesman starts at a given city and repetitively visits the nearest city until all have been visited.

These are the steps of the algorithm:

1. At the beginning, initialize all vertices as unvisited.
2. We are taking a vertex as an input; set it as the current vertex u. Mark u as visited.
3. Now we find the shortest edge connecting the current vertex u and an unvisited vertex v.
4. We then move to v, add the edge weight to the total distance covered. Set v as the current vertex u. Mark v as visited.
5. We will stop when all the vertices in the graph are visited.

The sequence of the visited vertices is the output of the algorithm. The algorithm is easy to implement and executes quickly, but due to its "greedy" nature, it does not always guarantee an optimal path. In other words, it gives the best solution (shortest path covering all the vertices) for some cases, but misses out in other cases when the traversal to the nearest vertices is costlier in an overall image.

Worst Case Performance $\theta(N^2)$

Worst Case Space Complexity $\theta(N)$

To get a better idea of whether the quick response time of the Greedy algorithm is worth letting go of the requirement of an optimal solution, we noted the percentage of times the Greedy Algorithm provided the correct solution. Out of 10 times, it was only correct twice, therefore, 20% of the time.

2.3.3 Dynamic Programming

We expect this algorithm to take the least time, as it does not solve sub-problems that it has already solved. However, because of the nature of our problem, we do not expect an exceptionally better run-time, but it performed significantly better than brute force, giving us an exponential run-time. It did not perform as well as the Greedy Algorithm, however, gave us the optimal solution, unlike the Greedy Algorithm. Also required extra space to store the matrix. Run-time of $O(n^2.2^n)$ because there are at most $O(2^n)$ sub-problems that take linear time each.
For the previous two methods we had given the graph created as input, but for the Dynamic approach we had to create a matrix. The time for creating that matrix was also noted, and counted in the pre-processing time. The space required for the matrix is $O(n^2)$

3 Limitations

Because Google Maps only provides distances between cities that can be travelled to and from each other by car, our implementation is limited in its scope. To account for this, we constructed a list of about 80 cities around Asia, Africa and Europe, that met these requirements, and had the nodeGenerator(n) randomly pick n random cities, each time we tested the algorithms.

4 Conclusion

To summarize, Brute-force is the slowest algorithm, and can thus not be recommended in real life situations with large inputs, where a solution is needed quickly. However, it does give an optimal result. In comparison, the

Greedy Algorithm is much faster, and maintains a relatively standard run-time, despite increasing amounts of data. But there is a tradeoff, and this is at the cost of an optimal result, as the Greedy Algorithm only provides a solution, without a guarantee that the solution is optimal. Whereas, a Dynamic Programming approach, even though works in a superpolynomial time as well, gives a much better run-time complexity as compared to Brute-Force, and at the same time guarantees an optimal solution, unlike the Greedy Algorithm.

5 References

1. https://www.geeksforgeeks.org/
2. https://github.com/phvargas/TSP-python
3. http://www.xm1math.net/texmaker/
4. https://www.csd.uoc.gr/~hy583/papers/ch11.pdf
5. http://cs.indstate.edu/~zeeshan/aman.pdf
6. https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/evaluating-algorithms/a/comparing-run-time-efficiency
7. https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm