EzTravel: The Simple Travel Assistant

Nathan Loria

April 29, 2020

Motivation

The problem that I attempted to solve by implementing this project is known as the travelling salesman problem. The travelling salesman problem is a problem which looks for the most efficient route between nodes in a graph such that each node except for the starting node, is visited once, and the starting node is visited twice. The relevance of the name of this problem becomes clear when the definition is understood, as you can imagine a salesman travelling from city to city attempting to sell a certain product. If the salesman has a way to develop the most efficient route through each city then they can save both time and energy.

The reason that I wanted to approach this problem is that it has a myriad of practical applications in the real world. A list of these applications includes things like navigation services (Google Maps, iMaps, Waze), computer chip manufacturing, delivery routes for USPS and UPS, and urban planning in cities to link popular destinations [1]. From just this

small list, it is clear that the applications of this problem in the real world are extremely important. Without a solid algorithm to approach this problem, many systems such as the postal service, or even computers, would not be as efficient as they currently are. Because of the importance of this algorithm and the services that it can provide society, I wanted to research it further and attempt to implement it in a practical way.

Background

The travelling salesman problem was first mathematically formulated by the Irish mathematician W.R. Hamilton, and Thomas Kirkman, a British mathematician [2]. The initial brute force approach to this problem, which calculates every possible solution to the problem, has a time complexity of O(n!). This is problematic for graphs with a lot of notes and this becomes clear when looking at examples. Table 1 shows how quickly the number of permutations increases as n increases by 1. Because of this extremely large time complexity, a search for a more efficient solution to the problem began. In the coming years, there were several different algorithms proposed, such as the branch and bound algorithm. This algorithm divides the problem into subsets and, before the subset is checked again, it is compared against an upper and lower bound estimate. If the subset cannot produce a path that is better than one that has already been calculated, it is discarded. This solution was effective in reducing the time complexity of the problem, but it was not the optimal solution and it can become very exhaustive given the right conditions [3]. In 1962, an algorithm

Table 1: n! permutations

n	Permutations
5	120
6	720
7	5,040
8	40,320

known as the Held-Karp algorithm was proposed. The Held-Karp algorithm was theorized by mathematician Richard Bellman and computer scientist Richard Karp. This algorithm takes advantage of something called dynamic programming which breaks a problem down into multiple sub-sections and uses these to recursively find a solution [4]. The algorithm takes an adjacency matrix and uses the data from this to develop the shortest possible route, also known as the lowest weight hamiltonian cycle. The time complexity of this algorithm is O($n^2 2^n$), making it the most efficient solution to the problem that provides a path that is the most efficient solution.

EzTravel Overview

EzTravel is divided up into 5 java files that work together to develop the results that are presented to the user. The main method is located within TravelMain.java. This method orchestrates the entire program and displays a trip that has been planned based on parameters that the user entered. These parameters include the city they wish to depart from, the cities they would like to visit, the date they would like to check-in and check out, how many guests are coming on the trip, and what their ideal price per night is for the trip.

Algorithm 1: Create Distance Matrix

Input: an n-element ArrayList al of string values.

Output: an n x n-element distance matrix d of double values.

1: $DistanceFinder\ df \leftarrow new\ DistanceFinder()$
2: **for** $(int\ i = 0;\ i\ <\ al.size();\ i++)$ **do**
3: $d[0][i] \leftarrow df.getWeight(al.get(0),\ al.get(i))$
4: $d[i][0] \leftarrow df.getWeight(al.get(0),\ al.get(i))$
5: **if** $i\ !=\ (al.size() - 1)\ \&\&\ i\ !=\ 0$ **then**
6: $d[i][i+1] \leftarrow df.getWeight(al.get(i),\ al.get(i+1))$
7: $d[i+1][i] \leftarrow df.getWeight(al.get(i),\ al.get(i+1))$
8: **end if**
9: **end for**

These parameters are stored in TripPlan.java to make it easy for all of the other methods to access them. Once these parameters are given, the main method creates a distance matrix using the names of all of the locations that were inputted by the user.

The main method accomplishes this by calling a method from DistanceFinder.java. When this method is called, and two location strings, which are just addresses, are passed as parameters. DistanceFinder takes advantage of the Google Geocode API. This API allows the program to insert the address of a location into a query link and then get an HTTP Response as a result. This response contains the latitude and longitude of a location located within an XML file. After parsing this XML and extracting the latitude and longitude values, DistanceFinder then calculates the distance from location one to location two. This value is returned, and inserted into the distance matrix. Each location String that is added to the distance matrix represents a vertex in a graph. Every distance that is added between two locations represents an edge in a graph. The program automatically creates an edge between the vertex at index 0 and all other vertices. More edges are then created between the vertices at index 1 to index 2, index 3 to index 4, all the way up to index n-2 to index n-1. This is done to provide the smallest combination of possible cycles through the graph that contains the most efficient path. Algorithm 1 shows this process. The time complexity of this algorithm is O(n) where n is equal to the size of the input ArrayList.

Once the distance matrix has been created, it is passed in the constructor for the class TripPath. A UML diagram for TripPath can be seen in figure 1. The method solve() is then invoked from the TripPath

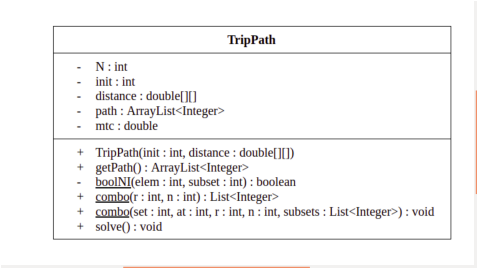

Figure 1: TripPath UML

object, and this initiates my implementation

of the Held-Karp algorithm. As previously mentioned, the Held-Karp algorithm is typically recursive. For this program, I decided to implement an iterative Held-Karp algorithm since it was easier for me to understand and it operates with the same time complexity. The way that this algorithm saves time is by storing all previously developed paths as binary values. The program can then access these paths more quickly than the brute force approach which allows for smaller time complexity. Table 2 shows a sample output from the getPath method in TripPath. The getPath method returns an ArrayList of integer values that are equal to the indexes of the cities that should be visited from first to last.

After the path ArrayList is returned, the main method re-orders the locations ArrayList based on the values from the path. This allows the locations ArrayList to follow chronological order from the first destination to the last destination. This newly sorted array is then passed to TripData and a new object of TripData is initialized containing all of the data for the trip. The next step in the program is to scrape the web for Airbnb listings that match the users' requests. The method that accomplishes this task is located in WebScraper.java. When the method scrapeData is called, WebScraper creates a connection to a query link based on user parameters and connects to this link using Jsoup, which is a library that allows you to programmatically extract specific HTML elements from websites. The element that is extracted from each query website is a link

Table 2: Travel Path From Batavia, NY

Locations	Visiting Order	Index
Chicago, IL	2	0
Buffalo, NY	4	1
New York, NY	1	2
Boston, MA	3	3

to the most relevant house in a certain location based on user preferences. This is then added to an ArrayList. Once this process is complete for each desired destination, the method returns an ArrayList of strings containing the links to book a house in every location located within locations. Finally, the main method prints the location name along with the link to book a stay there. Since these ArrayLists are already sorted, there is no need to do it again at the end of the program.

Results & Analysis

Because this algorithm has a large time complexity, I wanted to devise a way to test the amount of time that it would take to run my implementation of the algorithm with a different number of cities. In order to do this, I took advantage of Java's System.nanoTime() method. This method returns the number of nanoseconds that the program has been running for. When this number is taken before and after the method call that contains the Held-Karp algorithm, this data can be used to calculate the execution time of the algorithm. Table 3 shows the execution time in milliseconds for each value of n (number of cities) that was tested. As you can see, execution time does not increase linearly as n does. The, however, is much smaller than that of the brute force algorithm.

Table 3: TSP Execution Times

n	Execution Time (micro seconds)
7	1,243
8	1,964
9	3,816
10	6,267
11	17,404

When I implemented the brute force algorithm for testing purposes, I ran experiments for all values of n 1-7. When I attempted to execute the program with n equal to 8, my computer crashed. The execution time increased so greatly that an increase of 1 in n made the program not executable. Figure 2 shows the execution times of these two algorithms for different values of n. When viewed graphically, it becomes very clear the advantage of the dynamic program and the Held-Karp algorithm over finding each possible permutation of a problem using a brute force approach.

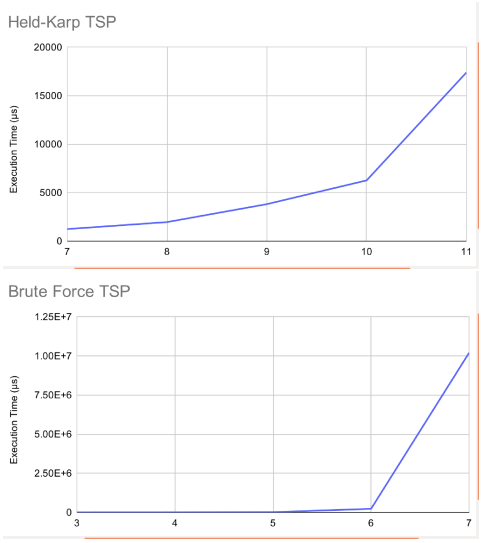

Figure 2: Held-Karp Vs Brute Force

Conclusion

This project was an extremely valuable learning experience. While implementing my solution to the traveling salesman problem, I developed many skills that will allow me to be even more successful in the coming years of my education. One of the skills that I feel I improved most is working with multiple different data structures in order to produce an efficient solution. In this project, it was very important that I maintained a lot of data, and developing data types was necessary to accomplish this. Through this process, I most definitely became a more experienced and mature programmer. Another important skill that was improved during this project was algorithm analysis. Although CS202 is an algorithm analysis course, using the knowledge that I gathered throughout the year to analyze

extremely complex algorithms enhanced my skills greatly.

Although coding was a large portion of what taught me so much in this project, the thing that I mostly learned from was the obstacles that I had to navigate to complete this project. The first major obstacle that I encountered was getting the Google Geocode API to function properly. I had to do a lot of research on how to do this and it definitely made my understanding of HTTP and APIs much more solid. The next major obstacle was the Held-Karp algorithm itself. Although it seems straight forward at first, implementing this problem is an entirely new way of programming that required a lot of research to understand. A lot of the concepts used were new to me so this obstacle was extremely helpful in teaching me new skills.

The biggest reward for me was the first successful execution of the program that printed out answers that made sense when you looked at them. Prior to this attempt, the program would print meaningless data that was clearly not the most efficient route and solution to the problem. When the results began to make sense, the whole process of trial and error became well worth the time and effort that was put in. In the end, this project was a very valuable experience for me and it was one that taught me many different things that will allow me to be a stronger programmer in the future.

References

[1] G. Laporte, A. Asef-Vaziri, and C. Sriskandarajah. Some Applications of the Generalized Travelling Salesman Problem. *Journal of the Operational Research Society*, 47(12):1461–1467, Dec 1, 1996.

[2] TSP History Home, www.math.uwaterloo.ca/tsp/history/index.html.

[3] “Branch and Bound Algorithm.” GeeksforGeeks, www.geeksforgeeks.org/branch-and-bound-algorithm/.

[4] Joshi, Vaidehi. “Speeding Up The Traveling Salesman Using Dynamic Programming.” Medium, Basecs, 13 Nov. 2017, medium.com/basecs/speeding-up-the-traveling-salesman-using-dynamic-programming-b76d7552e8dd.