Project 2 Report

Introduction:

In this project we were tasked with coding an algorithm that solves or approximates a solution to the traveling salesman problem. This problem asks the question: “Given a list of n cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" Thus we model the problem as a complete graph consisting of n vertices and attempt to create a tour of the graph, a hamiltonian cycle, where each city is visited once and the salesman finishes his tour where he began.

Furthermore, each trip from city x to city y has a cost which we assume in our implementation satisfies the triangle inequality. Hence, we are able to use the distance formula derived from Pythogaroean’s Theorem to calculate the cost from traveling from one city to the next. Essentially, we are attempting to find the tour with the cheapest total cost. Naturally, it is a NP problem; however, when implementing the decision problem

TSP = {<G, c, k>} : G = (V, E) is a complete graph,

C is a function from V x V -> Z

K ∈ Z

G has a traveling-salesman cost tour with cost at most k}.

it becomes a NP-Complete problem, meaning that a solution can be verified to the problem in polynomial time and it is at least as hard as the hardest NP problem thus finding a solution to these problems means we can solve any NP problem. A problem is considered P, if it could be solved in polynomial time and NP, if it could be verified in polynomial time. Thus a P problem is

also an NP problem. Furthermore, this report provides an analysis of the performance of the approximation algorithm provided in the textbook and a description of its implementation.

Background:

The first known record of the traveling salesman problem dates back to 1832 with examples of tours through Germany and Switzerland; however, it was not viewed in any mathematical regard. It was not until the late 19th century where the Irish mathematician by the name of W.R Hamilton and the British mathematician Thomas Kirkman viewed it in a different framework hence the name Hamiltonian cycle.

In the 1930s the general form of the TSP problem was studied in Vienna and at Harvard in a mathematical form. They analyzed the performance of the brute-force algorithm and of the nearest neighbor heuristic. The conclusion was that the brute force algorithm could be verified in polynomial time; however, it would run in O(n!) time to find the solution and that the rule that one should visit the closest point does not always yield the optimal solution. However, the Rand Corporation in the 1930s developed the cutting plane method for its solution and utilized this new methodology to solve an instance consisting of 49 cities. They found they only needed 26 cuts to come to a solution for this instance. While the paper and research on this approach was not algorithmic, the ideas were imperative to creating exact solutions to the TSP.

Furthemore, in the 1960s a new approach was created based on the ideology of producing an answer whose length is bounded by a multiple of the optimal length. Thus this results in lower bounds for the problem which when combined with branch bound approaches create a near optimal solution. This is done by creating a minimum spanning tree of the graph and doubling all its edges bounding the problem to the length of an optimal tour being at most twice the weight of

a minimum spanning tree. Eventually in 1976 a new algorithm dubbed the Christofides-Serdyukov algorithm yielded a solution which in the worst case was at most 1.5 times longer than the optimal solution which was the catalyst for a new generation for near optimal solutions.

Implementation:

In order to look at the TSP mathematically it is necessary to define various classes. The first class to define are the vertices from which each connection is being drawn. These vertices can be defined as a pair of x, y coordinates. Furthermore, the next class to define is an edge which is a pair of vertices, v_1, v_2. Lastly we must define the undirected graph class which condenses the list of vertices and list of edges into an adjacency matrix or an adjacency list. In this instance an adjacency list was utilized which was a 2D vector with each index in the outer vector corresponding to the position of the vertex in the list of vertices. Moreover, in each adjList[i], there is a vector of corresponding edges where one of the vertices is vertex[i] and the other is the vertex that was connected to it, thus organizing the adjacency list into a vector of vector of edges. Lastly, it is imperative to define a cost function, c(u,v), because that is the nature of the problem, to find a hamiltonian cycle, a tour of G with minimum cost.

In our rendition of the problem we assume that the cost function satisfies the triangle equality, essentially meaning that if an intermediate step is removed from a connection vertex a, to b then the cost would not increase. Put formally, cost function c satisfies the triangle inequality if for all $u,v \in V$

$$c(u, v) \leq c(u, v) + c(v, w).$$

In addition, we utilize the Euclidean distance formula utilizing the attributes of the vertices to create a nonnegative cost function that satisfies the triangle inequality in an everyday manner.

The algorithm utilized to approximate the optimal solution uses a structure called a minimum spanning tree which finds the least costly way to connect each vertex. The difference between this and a hamiltonian cycle is that it does not form a tour but rather a tree. For example, this structure is utilized when doing electrical in house and circuit boards. The cost of the minimum spanning tree, MST, gives a lower bound on the length of an optimal TSP tour. Thus we can use the MST to create a tour whose cost does not exceed 2 times that of the MST's since the cost function satisfies the triangle inequality.

The main file, called driver.cpp, consists of the auxiliary function needed to make that MST and the files containing the vector, edge, and graph classes. Once the driver file reads in the input from the user about the filename that it wishes to use consisting of the list of vertices and edges in a specific format it condenses that information into a list of edges and a list of vertices and uses an adjacency list to create a graph. The user then selects the root vertex of the MST, indicating the starting point from which the structure will attempt to map each vertex.
This information is used when calling the MST-Prim function, as well as the Graph and the cost function. However, in my implementation I included the cost function as an attribute of each edge.

To make an MST there are two standard algorithms. The algorithm I utilized was the Prim algorithm which utilizes another structure called a Minimum Heap to find the least costly way of discovering a new vertex. The function requires that each vertex has a parent attribute and a key attribute showing the lowest cost associated with that edge thus I added those attributes

to the vertex class and initialized them in the constructor, vertex.key = INT_MAX and vertex.parent = “NIL”. After calling the MST function with the parameters G, r, where r represents the root vertex selected by the user, we set the key of the root vertex to 0. Then the algorithm proceeds to copy the list of vectors from G, G.V to the minHeap class and heapifies it to create the minHeap property structure, all of which is done in the constructor.

Furthermore, while the minHeap is not empty the algorithm extracts the vector with the smallest key value, u, and evaluates if all of the possible vertices that can connect to it are in the minHeap Q, and if their edge cost is less than its current key. If this condition is satisfied then we change the parent attribute of said vertex to the vertex that discovered it, u, and call a decrease key function that uses the map data structure to map the vertices with their position index in the minHeap. Thus, the decrease key can run in O(lg V). Since the decrease key updates the positional index of the vertex in the map we can access the index of any vertex at any time in O(1) and use its index to call heapify(i), which swaps the parent index with its child if its child is smaller than the parent node.

Now when we call extractMin(), it takes the smallest vertex since the heap property is always maintained after changing the key of the vertex and we send that vertex to another list of vertices at each extractMin call. Hence when we exit the function we have a list of the vertices in the order that they were discovered which is essentially the preorder traversal of the MST. However, this implies that each vertex succeeding the vertex prior is a valid edge belonging to the list of edges used in the graph. Furthermore, it is also necessary to add the root vertex to the end of this list to make it a tour.

Anaylsis:

When analyzing the runtime of the TSP approximation algorithm we find the function or process that creates the upper limit of the run time. In this scenario this is the MST-Prim function which is responsible for creating the minimum spanning tree. The first step of the function is to build the binary min heap which uses the Build Heap() in its constructor. This function iterates through the heap array and heapifies at each iteration ensuring that the entire array maintains the heap property. Thus the constructor runs in O(V) time. Moreover, the while loop runs until there are no more vertices thus it executes |V| times and each line in it is executed |V| times.

The first line inside the while loop is the Extract-Min() which heapifies the height of the tree times. Thus it runs in O(log v) time and since it's executed |V| times it's total cost through the duration of the program is O(V log V). Since we are using the adjacency implementation of a graph the for loop inside the while loop is executed 2|E| times throughout the full duration of the while loop since it counts each edge twice. In addition, the decrease key function inside the for loop runs in O(log v) time because it heapifies the height of the tree times. So during the whole execution of the MST-Prim() it runs in O(E log V) time. Furthermore, we stated that Extract-Min() runs in O(V log V) thus the run time of the entire MST-Prim() is the sum of these run times, O(E log V + V log V), hence the run time of our approximation TSP algorithm is O(E log V).

Conclusion:

In essence, the TSP is a NP-hard and NP which we define as NP-Complete. We prove that it is so by reducing an instance of the problem to another reduction problem, the hamiltonian cycle which can be further reduced until we eventually get a 3-Sat problem.

This is the problem wherein we are given a conjunction of 3-clause disjunctions,

(x_v11 OR x_v21 OR x_v31) AND (x_v12 OR x_v22 OR x_v32) AND ... AND (x_v1n OR x_v2n OR x_v3n)

where each x_vij is a boolean variable. Thus, if one NP-Hard problem can be solved in P time then every function can be solved NP-Hard problem can be solved in polynomial time and by definition then every NP problem can be solved in polynomial time, hence the unsolved problem P = NP.

Furthermore, since we are not aware that this is the case but we cannot prove that it is not we must approximate the solution to these decision problems which result in being NP-Complete. This report highlights the implementation of such an algorithm and its runtime which is subject to the runtime of the essential operations and implementations of the data structures utilized. By utilizing a minHeap we obtain O(log v) for most basic heap operations and by using implementing the graph as an adjacency list we only check known edges as opposed to checking each combination of vertices and verifying if they make an edge thus we are able to obtain a runtime of O(E log v) as opposed to O(V^2) which is asymptotically quicker.

Bibliography

DarthVaderDarthVader 48.5k6969 gold badges199199 silver badges293293 bronze badges, et al. “What Are the Differences between NP, NP-Complete and NP-Hard?” *Stack Overflow*, 1 Nov. 1957, https://stackoverflow.com/questions/1857244/what-are-the-differences-between-np-np-complete-and-np-hard.

Limited, Wordy. “Writing an Academic Report.” *Wordy*, https://www.wordy.com/writers-workshop/writing-an-academic-report/.

“Time Complexity of Building a Heap.” *GeeksforGeeks*, 18 Sept. 2017, https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/.

“Travelling Salesman Problem.” *Wikipedia*, Wikimedia Foundation, 30 Jan. 2022, https://en.wikipedia.org/wiki/Travelling_salesman_problem#History.

“Which Data Structure Has a Constant Time of Finding an Element?” *Quora*, https://www.quora.com/Which-data-structure-has-a-constant-time-of-finding-an-element.