Project 4 Report - Greedy Approximations (TSP)

COMPSCI 260P - Algorithms with Applications
Cutuli, Andre, cutulia, 803641441
Rodriguez, Antonio, antonr9, 41049941
Trinh, Linh, thitt2, 49172220
Van Riper, Brenda, bvanripe, 96124814
May 24, 2023

1 Introduction

In computer science, the Traveling Salesperson Problem (TSP) is a well-known combinatorial optimization challenge. The goal is to determine the quickest route that visits each city precisely once, travels the smallest distance between them, and then returns to the beginning city. A tree known as a Minimum Spanning Tree (MST) joins every vertex of a given linked, undirected graph with the least amount of edge weight. To put it another way, it is the portion of the graph's edges that form a tree (i.e., have no cycles) and the minimum sum of weights. The MST is an effective method for connecting all of the graph's vertices while reducing the overall cost or distance. This strategy offers a solution that is within a factor of two of the optimal answer, yet it does not ensure an optimal result.

The Minimum Spanning Tree (MST)-based technique for TSP may be classed as a greedy algorithm since it chooses the minimum-weight edges at each step to build the Minimum Spanning Tree, which serves as the basis for producing a 2-approximate solution to the TSP. A popular heuristic that guarantees a solution will only take up to twice as lengthy as the ideal one is the 2-approximation algorithm for TSP. For this project we are implementing the 2-approximation algorithm, here is an examination of this method based on experiments:

1.1 Algorithm Overview

- **Input:** The first line will be an integer, 0..n-1, where n is the number of vertices. The second line will be an integer, indicating the number edges, m, the remaining m lines will each be three integers, indicating an undirected edges. They will be three space-separated integers, representing first the two endpoints and then a positive integer weight for that edge.
- **Output:** Taking such an input, will then output only a single integer that is the cost of the minimum spanning tree (MST).

1.2 Experimental Setup:

Generate random instances of TSP with various numbers of cities (e.g., 10, 50, 100, 500) and random distances between them. Calculate the optimal answer for each situation using a precise solver or a thorough search technique. Calculate the tour length using the 2-approximation approach on the same examples.

1.3 Performance Evaluation:

For each instance, calculate the tour length using the 2-approximation approach. Calculate the approximation ratio, which is the ratio of the algorithm's tour length to the ideal tour length, by comparing the tour length with the best answer. Determine the average approximation ratio for each issue size based on repeated examples.

2 Code Testing and Analysis

2.1 Behavior

When creating the function to find the MST, we decided that all of our test cases would use connected and undirected graphs, so Prim's algorithm was chosen. Using a adjacency list, the implementation has a time complexity of $O(V^2)$ and a space complexity of $O(V + E)$. With Prim's algorithm, it allows for our test cases to include negative weights in the graphs. To test our implementation of Prim's algorithm, we used a combination of test cases from previous projects and our own generated graphs to ensure the paths were correct for small and large scale graphs. Since our tests were solely on connected graphs, verifying there were no cycles with a path size of V was also used to test the correctness of out MST code.

Other than correctness, we tested the optimization of out MST function. Using the time library in python, we can get the timestamps before and after the find MST function to get the run-time. With the cutoff of three minutes for a graph with 500 vertices, our code was able to produce correct MST in 38.75 seconds. Below is the graph for every group size used in testing.

Graph Size	Run Time (Seconds)
100	0.314
200	2.539
300	8.551
400	20.366
500	38.759
600	69.096
700	110.740
800	165.188
900	233.810
1000	320.665

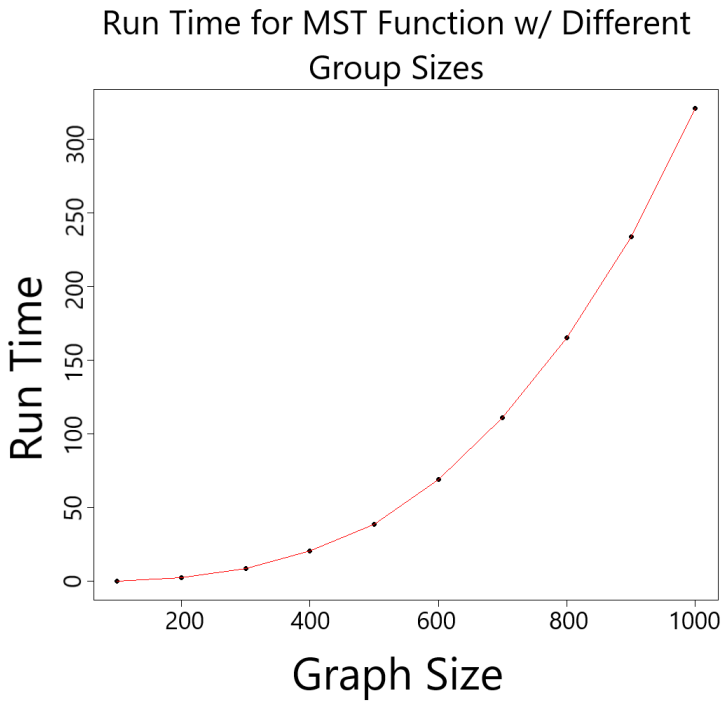


Figure 1: MST Run Time

It is not surprising to see an exponential rise in the MST algorithm's runtime as the graph size rose. The temporal complexity of Prim's method, which is $O(V2)$ for an adjacency list representation, can be linked to this characteristic. The number of vertices (V) and the number of edges (E) are the two variables that have the most effects on the runtime of Prim's algorithm. Both V and E tend to expand as the graph gets bigger. However, the number of edges in linked graphs is roughly proportional to the number of vertices $(E \approx V)$. V is the main factor influencing the runtime.

To determine the minimum-weight edge, Prim's method must look at every vertex's surrounding edges. This requires iterating over the adjacency lists of all vertices when using an adjacency list representation, which results in an $O(V)$ time complexity for each vertex. The total time complexity is $O(V^2)$ since the graph has V vertices. Longer runtimes result from the exponential growth in the number of iterations necessary to discover the MST as the number of vertices rises. This exponential rise is typical of algorithms with quadratic time complexity and is predicted.

2.2 Effectiveness

The main issue with the project assignment was that manually creating test cases for TSP and MST turns into an unreasonable job. As when the number of edges corresponding to the number of vertices in the graph becomes large enough becomes a harrowing job to keep track of all edges and its weights while ensuring all the possible triangles satisfies the inequality. The graph generation wasn't the difficult aspect, but creating a graph where the triangle inequality holds is the real challenge. As a reminder. the triangle inequality means the sum of two sides of a triangle is always greater than the third side $(i.e. a + b > c)$.

In order to achieve this constraint, we had to take a naive approach to the triangle inequality where we put a constraint on the largest possible side of the triangle. By limiting the maximum weight, we can ensure the inequality by making the smallest possible edge weight to half of the max. For instance, say we have an edge with max weight, $A = 100$, then the smallest weight we can have is B = $\lceil \frac{A}{2} \rceil$ = 50. However, the sum of 2 edges needs to be less than the third. So, if $A = 100, B = 50$, and $C = 50$, the inequality doesn't hold. As a result, we can say the highest possible value we can select is A-1 which creates a range, B $\leq$ w $<$ A, that we can randomly assign an integer to any edge without breaking the triangle inequality constraint in a complete graph.

The TSP function is based on Approximate Metric TSP discussed in lecture as it takes the Depth First Search of the graph into an array, J, then iterate through the array by add the edge weight between vertices, $J[i]$ and $J[i + 1]$. So, this will not be functional with incomplete graphs because of the scenario where no edge exists between $J[i]$ and $J[i + 1]$.

Since the general idea of the TSP computation has been explained, the file, test.txt, has two uses in our testing. First, it was to observe the MST function correctly made a tree and lists out the edges within each branch, so we could compare with the our solution done manually. The second purpose is to check if the TSP's journey, $Cost(J)$ is less than or equal to 2 times the cost of the MST, $2 * Cost(MST)$. If it didn't remain consistently less than the walk of the MST, then there was a underlying problem in with the code portion for TSP. After determining that our program was sufficiently test for small graphs, we immediately tested on 500 vertex graph to determine if the code's run time to compute a 2-approximation of TSP within in a 3 minute time limit.

3 Measure Approximation Ratio Results

3.1 Proof that Greedy Algorithm Generates a Solution

Proof: Let's consider a finite connected graph $G = V(G), E(G)$, where $deg(v) = 2$ for all $v \in V(G)$. Our goal is to show that the greedy algorithm generates a solution. To begin, we trace a path P by starting with an arbitrary vertex and iteratively picking a vertex adjacent to the previously added vertex, without repeating edges. Until we cannot add another vertex or we reach a vertex that was visited previously, this procedure continues.

Since G is finite, there is a limited number of vertices and we claim that this process must terminate. Additionally, since $deg(v) = 2$ for all $v \in V(G)$, each vertex has exactly one entering and one exiting edge. Therefore, if the process were to continue indefinitely, we would encounter a repeated vertex, which contradicts the assumption that G is a finite graph. Thus, the process must terminate.

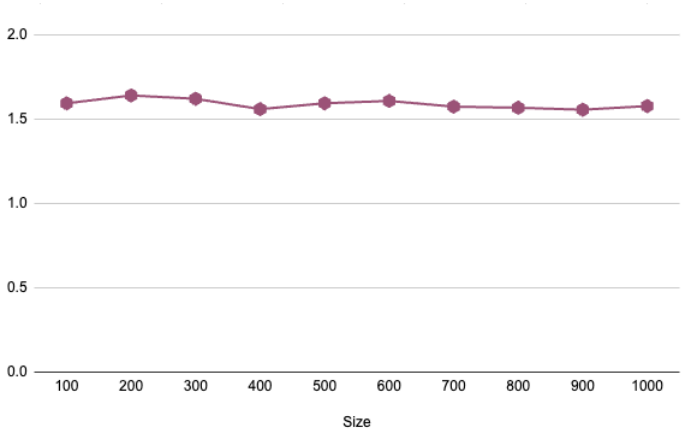
We refer to the terminating vertex as v_t, and the resulting path as P, which forms a cycle containing v_t. It's important to note that this cycle is a subgraph of G. The collection of all the vertices in the cycle P is what we refer to as $V(P)$. Each edge $e_i \in E(G)$ that contains a vertex in $V(P)$ must also be an edge in P because all vertices in G have a degree of 2.

Now, suppose there exists a vertex $y \in V(G)$ that is not in $V(P)$. In other words, there is no path from y to any vertex in

$V(P)$. If such a vertex y existed, there would be an edge not in P connecting a vertex in $V(P)$ to one not in $V(P)$, which contradicts our previous statement that all edges containing a vertex in $V(P)$ are in P. However, since G is a connected graph, every vertex must be connected to at least one vertex in $V(P)$. Thus, there can be no vertex y that does not connect to any vertex in $V(P)$.

Therefore, we draw the conclusion that $V(G) = V(P)$, proving that G is a cycle. The cycle is also Hamiltonian by design since it reaches each vertex exactly once. The power of the restriction $deg(v) \leq 2$ for all $v \in V(G)$ in the greedy algorithm becomes evident. At the end of the algorithm, the degree of each vertex is exactly 2, as the algorithm selects adjacent vertices without repetition. Therefore, the greedy algorithm always generates a solution. □

3.2 Results and Analysis



Size	TSP	MST	Ratio
100	10024	15982	1.59
200	19111	31364	1.64
300	28596	46364	1.62
400	39131	61040	1.56
500	47909	76406	1.59
600	56865	91476	1.61
700	67570	106404	1.57
800	77216	121094	1.57
900	87647	136434	1.56
1000	95773	151140	1.58

Figure 2: MST TSP Ratio

From the results, we can observe the following trends:

1. As the problem size increases, both the TSP and MST values increase. This is expected because larger problem sizes typically require more edges to connect all the vertices, resulting in longer paths and higher costs.

2. The ratio between the TSP and MST was computed by using this formula: $\frac{Cost(TSP)}{2*Cost(MST)}$. From observing our test cases' results, the values for both TSP and MST fluctuates, but generally stays around 1.5 to 1.7. This suggests that the 2-approximation algorithm provides solutions that are approximately 1.5 to 1.7 times the optimal solution (represented by the MST). This indicates that the algorithm is not optimal but still provides reasonably good solutions.

3. The ratio does not exhibit a clear trend with increasing problem size. It is important to note that the quality of approximation algorithms may vary depending on the specific problem instance, so it is possible to encounter cases where the ratio deviates significantly from the average. Additionally, it could depend on the range of values that each edge could randomly select. Especially since the Gtest cases were graphs whose edges' weights whose values could only be with the range: $50 \leq w \leq 99$, so varying ranges of values was not explored.

Overall, the 2-approximation algorithm seems to perform reasonably well for the given problem instances. However, for a comprehensive analysis, it would be beneficial to compare these results with the optimal solutions (if known) or other approximation algorithms to evaluate the performance more accurately.