Algorithm Design and Analysis Project Final Report

Time-Dependent Traveling Salesman Problem

BY:

Cassie Valleria Garcia (2702391453)

Cindy Reginia Wang (2702334815)

Ella Raputri (2702298154)

Ellis Raputri (2702298116)

Class L3AC
Computer Science Program
School of Computing and Creative Arts

Bina Nusantara International University
Jakarta
2024

Final Project Report

A. Introduction

Background

The Traveling Salesman Problem (TSP) is an algorithmic problem that involves finding the shortest possible route at which all locations in a given set of points are visited once, and end at the starting point [1]. It was introduced by a mathematician and economist, Karl Menger, in the 20th century [2]. TSP was referred to as the "Messenger Problem", due to its popularity amongst lettermen, who must distribute a handful of letters daily. There are millions of ways to travel to each of their destinations, however, without proper and accurate planning, they may travel unnecessarily long distances, which in turn will take up too much time, cost more, and certainly tire them out. Over time, the problem becomes prominent in the sales industry, hence the name "Traveling Salesman Problem".

Importance of TSP

TSP remains to be important for industries where transportation and logistics are crucial. In today's digital age, where online transactions have become a cornerstone of commerce, the need for efficient delivery routes is crucial. Jobs and companies that include itinerant employees and transportation like e-commerce, food delivery, courier services, and ride-hailing platforms tend to depend on optimized logistics to meet customer demands swiftly and cost-effectively. For example, online retailers such as Amazon or food delivery services like Uber Eats must carefully plan their delivery routes to minimize travel time and reduce fuel costs. Managing this directly affects customer satisfaction and operational efficiency. Without considering the TSP problem, this can lead to inefficient routing and in turn lead to higher costs, delays, and wasted resources. Using TSP principles, can in theory help businesses streamline their operations, reducing the time and distance required to complete their services. Thus, TSP is often utilized to diminish their spending by reducing the distance and time taken for the task to be completed [2]. Beyond minimizing expenses, optimized routes can also contribute to reducing environmental footprint by lowering fuel consumption and emissions. This not only enhances corporate sustainability efforts but also improves public perception in a world increasingly concerned with environmental impact.

Applications in Modern Technology

Aside from lowering travel costs, TSP is critical for optimization in the more technical field. Some examples from everyday sight are cellular towers and cabling. Cellular towers are built in locations at which the area will receive a fair share of signals, whilst ensuring the number of towers is within the budget's limits. Similarly, with cabling, TSP is employed to guarantee that electricity is equally distributed with the least amount of cables. That said, the algorithm has been essential in a diversity of services and the daily routines of civilization all over the globe. With this comes the likelihood of its further assistance in future applications. Given TSP's wide-ranging applicability, it has the potential to be the driving force behind many future innovations. As technology advances, there will be the need to apply TSP in industries such as autonomous vehicles, smart cities, and drone delivery services. These applications require precise planning to minimize fuel consumption, reduce travel time, and make efficient use of resources. With continuous improvements in computational algorithms and machine learning, TSP will remain useful for problems in both established and emerging industries.

B. Problem

Limitations on Classical TSP

Despite its widespread application, TSP has some limitations. One of the key issues with TSP is that it generally only contains constant time windows, which may be unsuitable in several conditions where there could be potential delays or setbacks in the journey at which something travels from one point to another. Moreover, research says that "failing to solve the TSP efficiently can lead to wasted time, increased fuel costs, late deliveries, and ultimately, dissatisfied customers" [3]. The lack of flexibility towards potential traffic or other impediments comes with the increased probability of the mentioned risks. Another aspect to consider is in the real world, deliveries are subject to last-minute changes. In real-life logistics, new orders might come in unexpectedly, customers might request altercations to the delivery place and time, or the package might not be ready by the time the driver gets there. This requires constant route adjustments, which the statistical nature of classical TSP does not support.

Time-dependent TSP

With this in mind, our group has decided to investigate and explore the time-dependent traveling salesman problem. In time-dependent TSP, the time needed to travel from one point to another will vary throughout the day [4]. Realistically, this is much more effective in implementations considering the road conditions, such as traffic, whose severity changes over time, making it difficult to accurately determine the fastest route to the destination. We figured that by analyzing and solving this, we would be able to further scrutinize the topic and apply our results in a more realistic scenario.

Project Objective

Our project aims to investigate the TD-TSP (Time-dependent Traveling Salesman Problem) and explore how incorporating the time-dependent variables allows the algorithm to be more dynamic and improve upon route planning. We plan to incorporate fluctuating travel times based on traffic conditions and road closures into our analysis and explore how different periods of the day, such as rush hour or off-peak times, affect the solution to the TSP. We aim to analyze the effect of travel times on the TSP problem and hope to provide a more realistic solution to the TSP problem that reflects real-world traffic conditions. We hope that with this paper, we are able to contribute to the research on TSP.

C. Method

Algorithms

The traveling salesman problem (TSP) is a classical problem that already has many solving algorithms. In general, the solving algorithms are divided into two types: full and partial solutions. The full solution emphasizes finding the most optimal solution for the problem, thus it is also known as the exact solution [5]. On the other hand, partial solutions consist of heuristic techniques that do not aim to solve the problem completely. Still, it provides a near-optimal solution by emphasizing the algorithm speed [5]. Heuristic methods have proven to be better than the other methods, solving TSP with many nodes [6]. However, in this study, we will only consider the exact solution of TSP.

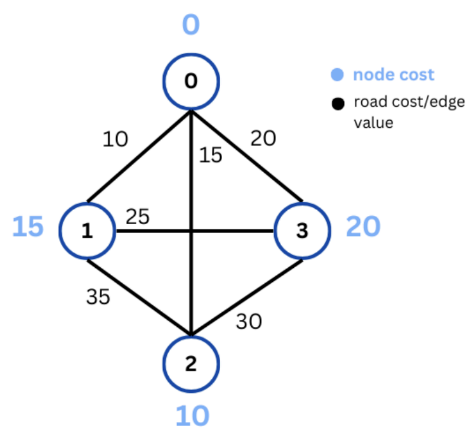
- Brute Force (Naive solution)

  Since TSP is a combinatorial problem, the naive solution just enumerates all possible routes with a total of (n-1)! routes [7]. It explores all possible roads while calculating the total time taken on each road. After that, it compares all the calculation values, which ultimately result in the most optimal value [8]. The advantage of this algorithm is that it guarantees the global maximum value or the most optimal solution. However, its time and space increased exponentially with every new node added, indicating that the algorithm was not good enough to solve the problem.

- Dynamic Programming
  The dynamic programming approach is similar to the naive solution, which explores all possible paths. However, it has better time complexity since it utilizes memoization to store the explored paths, so it does not need to recalculate the value of the explored path. This algorithm utilized the bitmasking technique in the recursion to minimize the algorithm running time [9]. The bit represents each node visited status, 0 for unvisited and 1 for visited. Each bit will be then "turned on" or "turned off" based on the needs by using bitwise operations, like and (&), or (|), and left shift (<<) [10]. In other words, all possible routes are stored in the memo in the form of their bit representations, so it is easier to save and compare the visited nodes in each recursion.

- Branch-and-Bound
  The branch-and-bound algorithm is one of the most often used approaches to solve TSP. The method is similar to the naive brute force method, but it prunes some of the recursion tree by calculating the bound of branches of the recursion tree [7]. The branch-and-bound method does not explore all the possible path combinations like brute force and dynamic programming. Instead, for each level of the recursion tree, it checks whether the bound is greater than the current bound. If it is greater, then it does not traverse that branch. The worst-case scenario for branch-and-bound is that it has to traverse all the nodes without any memoization, so it will take the same time as the brute force method.

Next, we consider these basic algorithms to be insufficient in solving the problem of Time-Dependent TSP since they do not consider the congestion time in each road segment. We can see the example of a classic TSP problem in the figure below.



Now, we consider three possible scenarios:

- No congestion
- Roads (1-3) and (3-1) got congestion from timestamp 0 to 100, which caused a 0.2 increase in travel time
- Roads (1-3) and (3-1) got congestion from timestamp 5 to 70, which caused a 3.0 increase in travel time

With the basic TSP algorithm, the minimum cost of each scenario will always be 125, and the optimal path will be 0-1-3-2-0. This may be the case when no congestion is involved. However, for the second scenario, the minimum cost should be 130 since there is a 0.2 increase in travel time on the road (1-3).

Another problem would be the third scenario. If we took the road 0-1-3-2-0, it should get an additional time of 75 in the road (1-3), resulting in the final cost being 200. However, it is possible to get a final cost of 125, that is, through the road 0-2-3-1-0. When the calculation reaches the road (3-1), the time is already at 75, which exceeds the congestion time (5 to 70), thus it does not get any additional travel time from the congestion.

Hence, we refactored the basic algorithms to incorporate the additional travel time due to congestion. The principle of each algorithm is still the same, but they have the same new feature to calculate the optimal path while anticipating the additional time caused by congestion. The time and space complexity of these algorithms are shown in the big-O table below. The n in the table is the number of nodes, e is the number of edges that have congestion and c is the average number of congestion per edge.

Complexity	**Brute Force**	**Dynamic Programming**	**Branch and Bound**
Time	$O(n!)$	$O(n^2 \cdot 2^n)$	$O(n!)$
Space	$O(n^2 + e \cdot c)$	$O(n \cdot 2^n + n^2 + e \cdot c)$	$O(n^2 + e \cdot c)$

The time for the brute force method is dominated by the TD-TSP recursive function. Meanwhile, the space is dominated by the size of the adjacency matrix, which is n^2.

On the other hand, dynamic programming can solve the problem in exponential time complexity (better than brute force approach). The dynamic programming table has n * 2^n states, which is why the space complexity for dynamic programming is the space for brute force added by n * 2^n. For the time complexity, the algorithm attempts to fill all the values in the memo table and for each state in the table, loops through n nodes to calculate the minimum cost for visiting unvisited nodes. That is why the time complexity of dynamic programming is O(n^2 * 2^n).

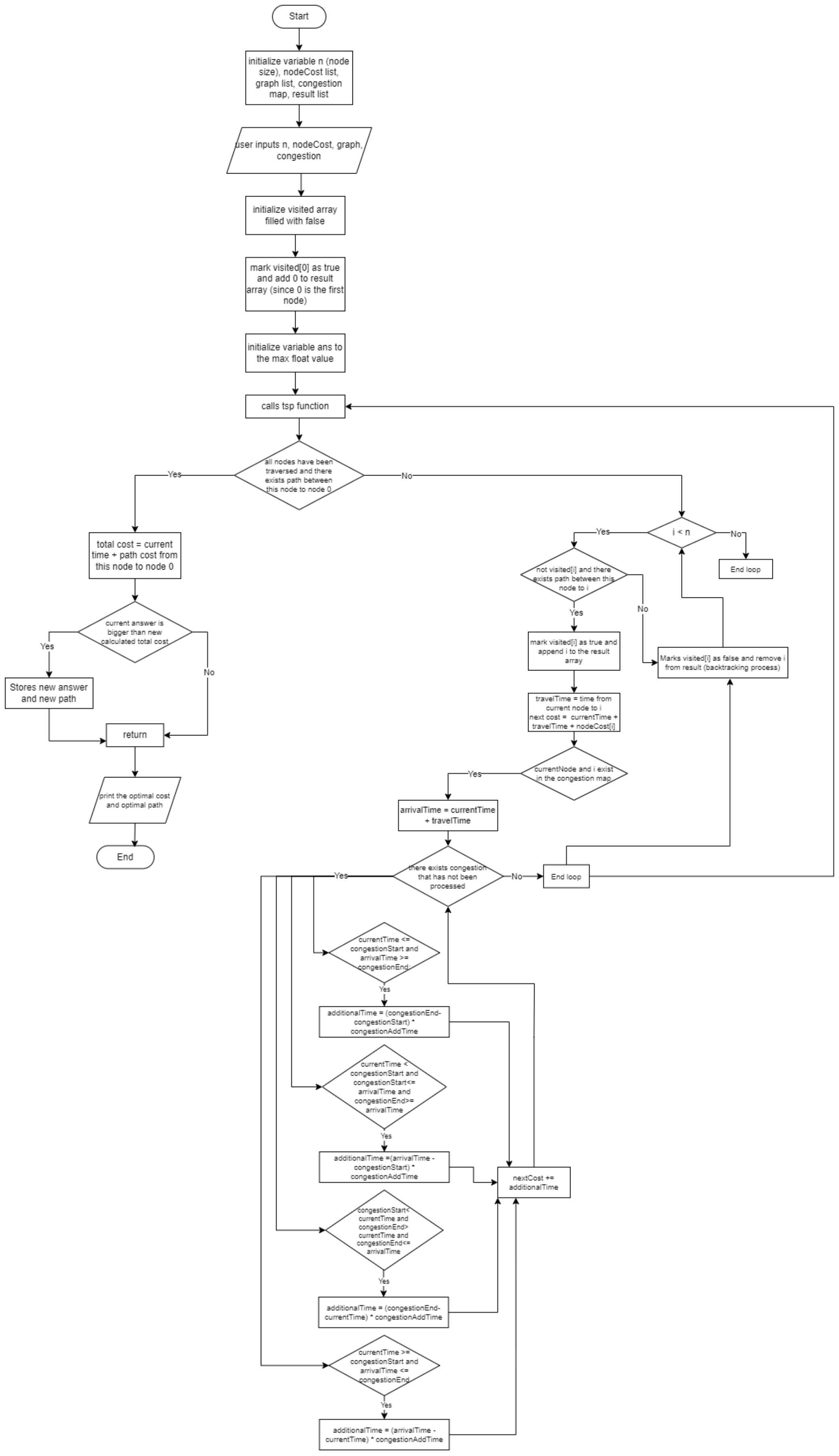
Lastly, the branch-and-bound method has similar time and space complexity with brute force because in the worst case, branch-and-bound is the brute force itself. However, branch-and-bound has better average time complexity than the brute force method, depending on the graph structure and pruning method because it prunes the recursion tree and does not traverse all the possible combinations like the brute force approach.

Algorithm Flowcharts

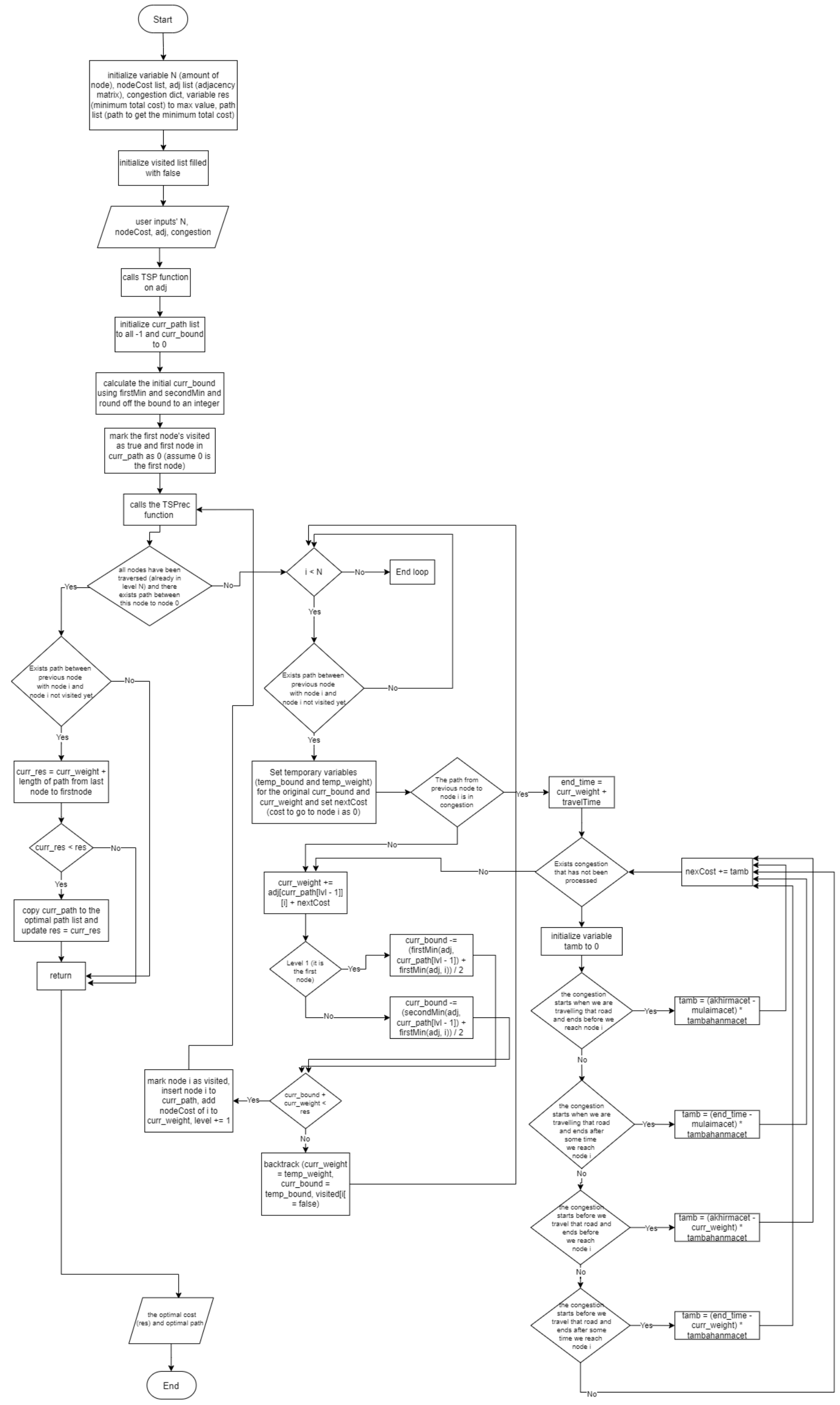
Below are the rough estimation of flowcharts for each algorithm:

(The flowchart pdf files are available in the folder https://drive.google.com/drive/folders/1elUZ9KWQCSl7VQ4FRrfvy6x3iQcDS-nZ?usp=drive_link).

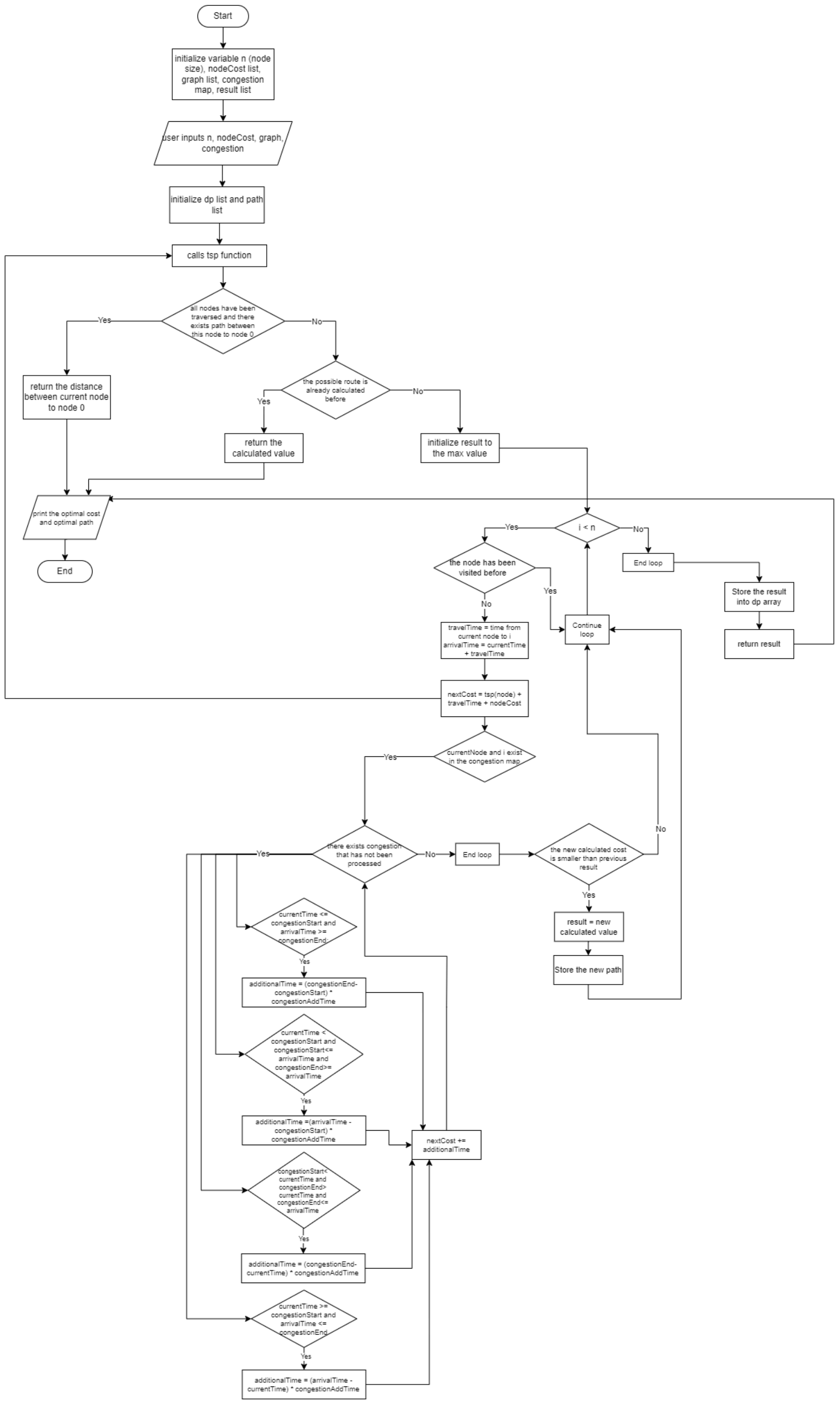
- Brute Force Algorithm


Start
initialize variable n (node size), nodeCost list, graph list, congestion map, result list
user inputs n, nodeCost, graph, congestion
initialize visited array filled with false
mark visited[0] as true and add 0 to result array (since 0 is the first node)
initialize variable ans to the max float value
calls tsp function
all nodes have been traversed and there exists path between this node to node 0
Yes
No
total cost = current time + path cost from this node to node 0
current answer is bigger than new calculated total cost
Yes
No
Stores new answer and new path
return
print the optimal cost and optimal path
End
i < n
Yes
No
End loop
not visited[i] and there exists path between this node to i
Yes
No
mark visited[i] as true and append i to the result array
Marks visited[i] as false and remove i from result (backtracking process)
travelTime = time from current node to i
next cost = currentTime + travelTime + nodeCost[i]
currentNode and i exist in the congestion map
Yes
arrivalTime = currentTime + travelTime
there exists congestion that has not been processed
Yes
No
End loop
currentTime <= congestionStart and arrivalTime >= congestionEnd:
Yes
additionalTime = (congestionEnd- congestionStart) * congestionAddTime
currentTime < congestionStart and congestionStart<= arrivalTime and congestionEnd>= arrivalTime
Yes
additionalTime =(arrivalTime - congestionStart) * congestionAddTime
nextCost += additionalTime
congestionStart< currentTime and congestionEnd> currentTime and congestionEnd<= arrivalTime
Yes
additionalTime = (congestionEnd- currentTime) * congestionAddTime
currentTime >= congestionStart and arrivalTime <= congestionEnd
Yes
additionalTime = (arrivalTime - currentTime) * congestionAddTime

- Branch and Bound Algorithm


Start
initialize variable N (amount of node), nodeCost list, adj list (adjacency matrix), congestion dict, variable res (minimum total cost) to max value, path list (path to get the minimum total cost)
initialize visited list filled with false
user inputs' N, nodeCost, adj, congestion
calls TSP function on adj
initialize curr_path list to all -1 and curr_bound to 0
calculate the initial curr_bound using firstMin and secondMin and round off the bound to an integer
mark the first node's visited as true and first node in curr_path as 0 (assume 0 is the first node)
calls the TSPrec function
all nodes have been traversed (already in level N) and there exists path between this node to node 0
Yes
No
Exists path between previous node with node i and node i not visited yet
Yes
No
curr_res = curr_weight + length of path from last node to firstnode
curr_res < res
Yes
No
copy curr_path to the optimal path list and update res = curr_res
return
the optimal cost (res) and optimal path
End
i < N
No
End loop
Yes
Exists path between previous node with node i and node i not visited yet
No
Yes
Set temporary variables (temp_bound and temp_weight) for the original curr_bound and curr_weight and set nextCost (cost to go to node i as 0)
The path from previous node to node i is in congestion
Yes
No
end_time = curr_weight + travelTime
Exists congestion that has not been processed
No
curr_weight += adj[curr_path[lvl - 1]][i] + nextCost
Level 1 (it is the first node)
Yes
No
curr_bound -= (firstMin(adj, curr_path[lvl - 1]) + firstMin(adj, i)) / 2
curr_bound -= (secondMin(adj, curr_path[lvl - 1]) + firstMin(adj, i)) / 2
curr_bound + curr_weight < res
Yes
No
mark node i as visited, insert node i to curr_path, add nodeCost of i to curr_weight, level += 1
backtrack (curr_weight = temp_weight, curr_bound = temp_bound, visited[i[= false)
nexCost += tamb
initialize variable tamb to 0
the congestion starts when we are travelling that road and ends before we reach node i
Yes
tamb = (akhirmacet - mulaimacet) * tambahanmacet
No
the congestion starts when we are travelling that road and ends after some time we reach node i
Yes
tamb = (end_time - mulaimacet) * tambahanmacet
No
the congestion starts before we travel that road and ends before we reach node i
Yes
tamb = (akhirmacet - curr_weight) * tambahanmacet
No
the congestion starts before we travel that road and ends after some time we reach node i
Yes
tamb = (end_time - curr_weight) * tambahanmacet
No

- Dynamic Programming Algorithm


Start
initialize variable n (node size), nodeCost list, graph list, congestion map, result list
user inputs n, nodeCost, graph, congestion
initialize dp list and path list
calls tsp function
all nodes have been traversed and there exists path between this node to node 0
Yes
No
return the distance between current node to node 0
the possible route is already calculated before
Yes
No
return the calculated value
initialize result to the max value
print the optimal cost and optimal path
End
i < n
Yes
No
End loop
Store the result into dp array
return result
the node has been visited before
Yes
No
Continue loop
travelTime = time from current node to i arrivalTime = currentTime + travelTime
nextCost = tsp(node) + travelTime + nodeCost
currentNode and i exist in the congestion map
Yes
there exists congestion that has not been processed
Yes
No
End loop
the new calculated cost is smaller than previous result
No
Yes
result = new calculated value
Store the new path
currentTime <= congestionStart and arrivalTime >= congestionEnd:
Yes
additionalTime = (congestionEnd- congestionStart) * congestionAddTime
currentTime < congestionStart and congestionStart<= arrivalTime and congestionEnd>= arrivalTime
Yes
additionalTime =(arrivalTime - congestionStart) * congestionAddTime
nextCost += additionalTime
congestionStart< currentTime and congestionEnd> currentTime and congestionEnd<= arrivalTime
Yes
additionalTime = (congestionEnd- currentTime) * congestionAddTime
currentTime >= congestionStart and arrivalTime <= congestionEnd
Yes
additionalTime = (arrivalTime - currentTime) * congestionAddTime

Measurements

To compare the performance of each algorithm after incorporating the congestion feature, we decided to test each algorithm with the time-dependent traveling salesman problem that consists of three to ten nodes, with the congestion amount ranging from one to ten. The graph weight, node cost, and congestion cost are randomized using the Python Random library. Each road can have more than one congestion. We conducted ten tests for each case, then averaged all test results before putting them as the final result.

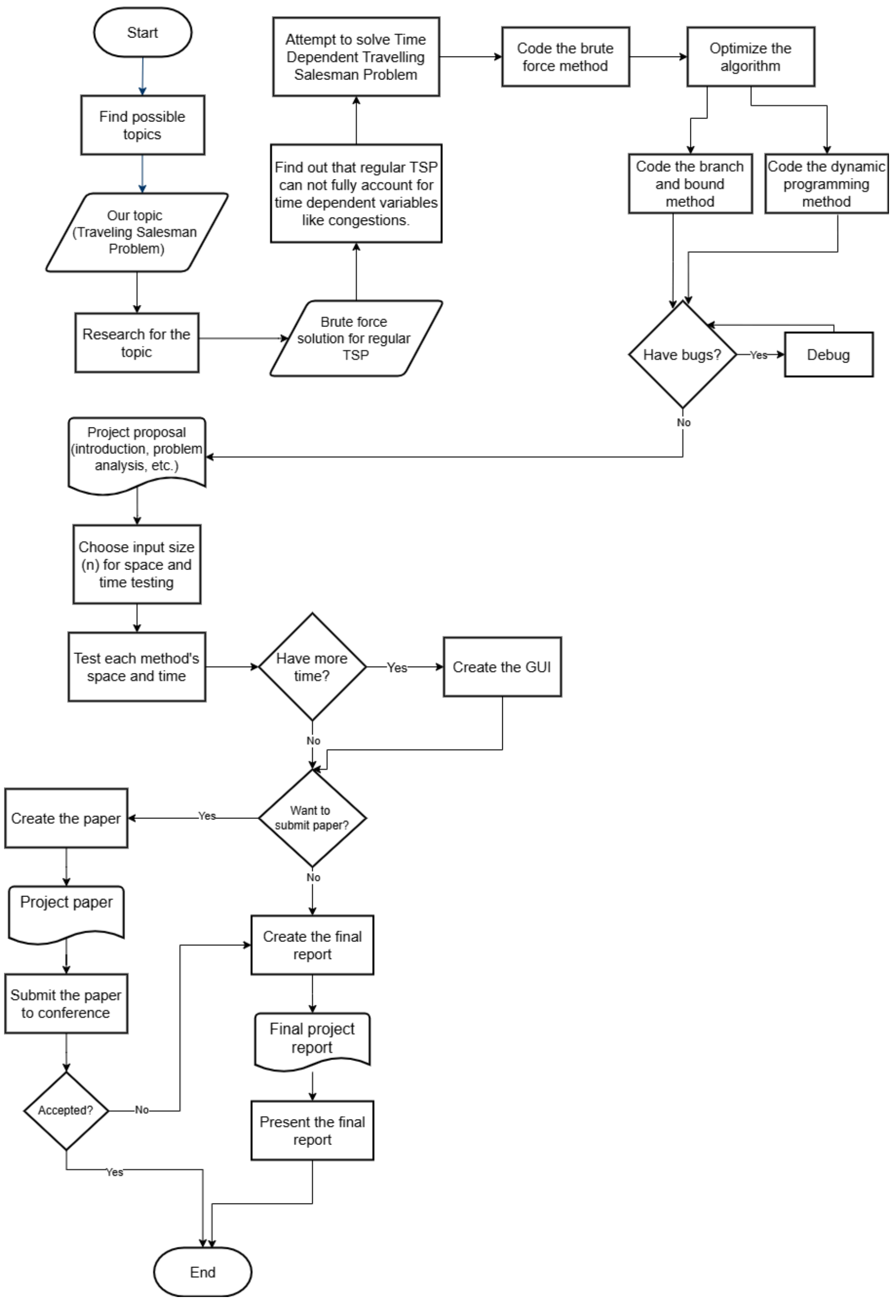
After that, we wanted to perform a t-test to see whether the algorithm difference was significant or not. However, we need to know first whether our data satisfies the parametric assumption of the t-test (the data distribution has to be normal). Therefore, we used the Shapiro-Wilk test for this purpose. Shapiro-Wilk test is usually utilized to verify the normality of a data distribution [11]. It is used for independent observations, such as ours where the results for brute force, branch-and-bound, and dynamic programming are independent of each other.

Then, from the Shapiro-Wilk test, we found that the data distribution is not normal. Hence, we perform the Mann-Whitney test to see whether the algorithm difference is significant. We decided to use the Mann-Whitney U test because our data is not normally distributed. The Mann-Whitney U test or the Wilcoxon rank-sum test is often used to test the difference in significance between two groups of data with no specific distribution [12]. Mann-Whitney U is similar to the t-test, but it is used for data that do not meet the parametric assumption of the t-test, such as our data.

D. Flow

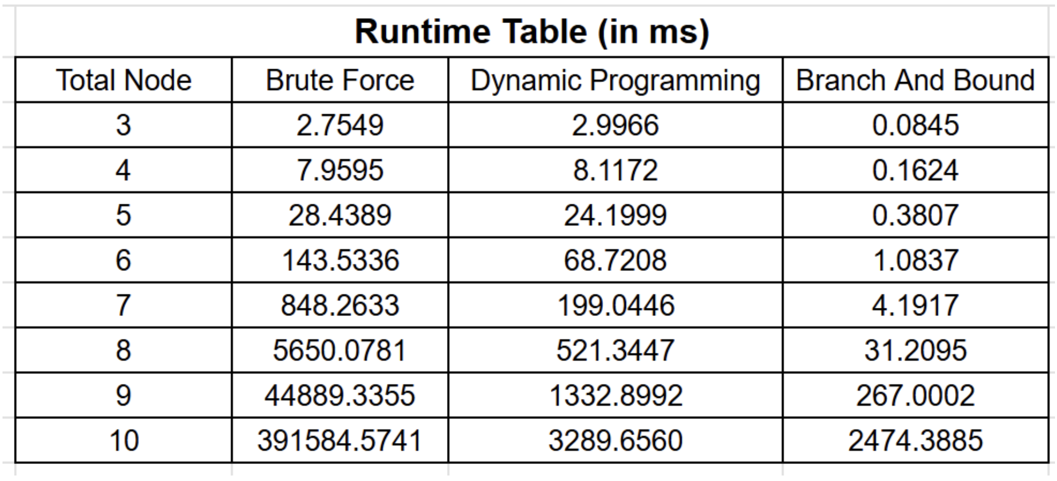
We already finished everything including the paper. However, since we still do not know whether the paper is accepted, we also created this report. The overall research flow is shown below:

(https://drive.google.com/file/d/1Z3-KxnJWeEyV_SZOBm-gb1ZCxpl5EWn7/view?usp=sharing)


Start
Find possible topics
Our topic (Traveling Salesman Problem)
Research for the topic
Brute force solution for regular TSP
Find out that regular TSP can not fully account for time dependent variables like congestions.
Attempt to solve Time Dependent Travelling Salesman Problem
Code the brute force method
Optimize the algorithm
Code the branch and bound method
Code the dynamic programming method
Have bugs?
Yes
Debug
No
Project proposal (introduction, problem analysis, etc.)
Choose input size (n) for space and time testing
Test each method's space and time
Have more time?
Yes
Create the GUI
No
Want to submit paper?
Yes
Create the paper
No
Project paper
Submit the paper to conference
Accepted?
No
Yes
Create the final report
Final project report
Present the final report
End

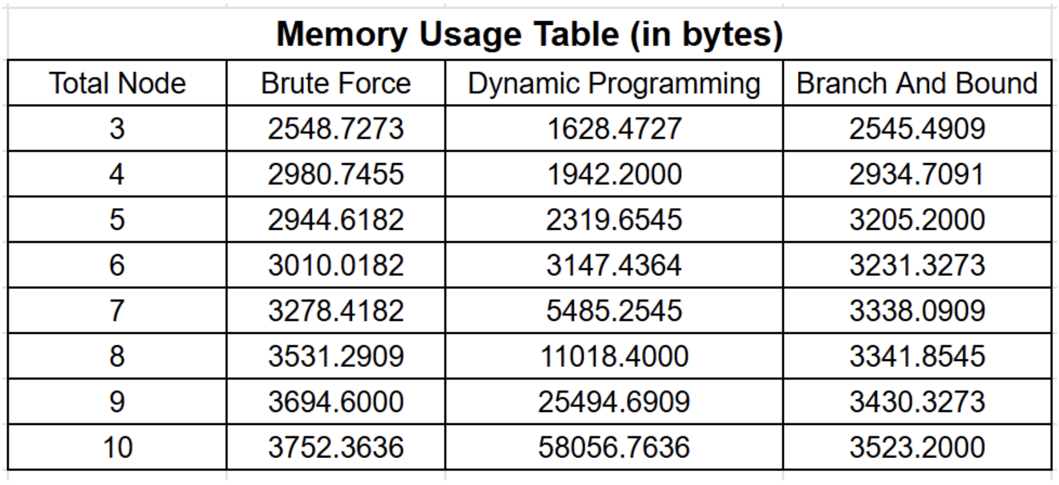
E. Results and Discussion

We tested the three algorithms across the total node of three to ten, then recorded the runtime and memory usage in each run. The runtime is measured in milliseconds, while the memory usage is measured in bytes. The runtime result can be seen in the runtime table, while the memory usage result can be seen in the memory usage table. The full result can be seen at https://docs.google.com/spreadsheets/d/1TZl7SleevPdP5eyd4b5Gq-ZMM16lsZTNJfkn4NXvQJo/edit?usp=sharing.



Runtime Table (in ms)

Total Node	Brute Force	Dynamic Programming	Branch And Bound
3	2.7549	2.9966	0.0845
4	7.9595	8.1172	0.1624
5	28.4389	24.1999	0.3807
6	143.5336	68.7208	1.0837
7	848.2633	199.0446	4.1917
8	5650.0781	521.3447	31.2095
9	44889.3355	1332.8992	267.0002
10	391584.5741	3289.6560	2474.3885



Memory Usage Table (in bytes)

Total Node	Brute Force	Dynamic Programming	Branch And Bound
3	2548.7273	1628.4727	2545.4909
4	2980.7455	1942.2000	2934.7091
5	2944.6182	2319.6545	3205.2000
6	3010.0182	3147.4364	3231.3273
7	3278.4182	5485.2545	3338.0909
8	3531.2909	11018.4000	3341.8545
9	3694.6000	25494.6909	3430.3273
10	3752.3636	58056.7636	3523.2000

We will first explore the runtime result. As seen in the runtime table, the branch-and-bound algorithm is the most optimal algorithm that consumes the least time regardless of the number of nodes. This could happen since the algorithm prunes some of the paths while traversing the graph, so it does not explore all the paths like brute force or dynamic programming [7]. On the other hand, we can see that the results of brute force and dynamic programming are quite varied. At first, the brute-force method runs faster than dynamic programming. However, as the total number of nodes increases, dynamic programming becomes faster and more effective. At first, dynamic programming is slower since it needs some pre-processing time to initialize the memo.

Next, we will explore the memory usage result. As seen in the memory usage table, brute force and branch-and-bound methods have a more stable memory usage. It increases slowly as the total number of nodes increases. On the other hand, the dynamic programming method's memory usage increases by a considerable amount in each node addition. This could happen since dynamic programming utilizes a memo to store the visited path, which results in a faster execution but larger memory usage [13].

Furthermore, we tested each algorithm with the Mann-Whitney test to determine the significance of their differences. The calculation process can be seen in our GitHub repository ‘Statistics’ folder. The p-value results of each hypothesis test are shown in the p-value table below, which we utilized α = 0.05. From the test, we can conclude that brute force and dynamic programming differ significantly from each other, both in time and space. This also applies to dynamic programming and branch-and-bound, which differ significantly from each other. Lastly, the brute force and branch-and-bound methods differ significantly too, but only in the time usage. In terms of space, we cannot say that they have significant differences.

Algorithm	Time	Space
Brute Force and Dynamic Programming	$6.14 * 10^{-15}$	$2.03 * 10^{-7}$
Brute Force and Branch and Bound	$1.46 * 10^{-105}$	0.377767765
Dynamic Programming and Branch and Bound	$1.49 * 10^{-74}$	$1.39 * 10^{-7}$

From the result and analysis of the Mann-Whitney test, we could further summarize the result. We can confidently say that dynamic programming is faster than brute force, but requires significantly more memory. Then, the branch-and-bound method is faster than dynamic programming and requires less space. Lastly, the branch-and-bound method is faster than the brute force method, but their memory usage does not differ much.

F. Conclusion and Recommendation

To conclude, this study analyzes the performance of each Time-Dependent Traveling Salesman Problem algorithm that we developed, which are the brute force, dynamic programming, and branch-and-bound methods. We aimed to create these algorithms so that we can account for congestion in the Traveling Salesman Problem, which the classic Traveling Salesman Problem can not resolve. We tested each algorithm with different amounts of nodes and congestion to see their usage time and space. After that, we conducted a statistical test to test the significance of the difference in the result.

The comparison of each algorithm's average time taken reveals that the branch-and-bound algorithm is the fastest one, regardless of the number of nodes. It is likely due to the pruning nature of branch-and-bound, so it does not need to explore all the nodes, like brute force and dynamic programming approach. Meanwhile, dynamic programming is slower than brute force in a small number of nodes, but faster in a large number of nodes due to the time to initialize the memo. Dynamic programming is also the algorithm that uses the largest memory since it utilizes a memo to store the results of the smaller subproblems.

The statistical test that we conducted further determined that the difference in the time taken between each algorithm is significant, meaning that branch-and-bound is significantly faster than dynamic programming and brute force, while dynamic programming is significantly faster than brute force. Meanwhile, for space complexity, we can only conclude that dynamic programming takes significantly larger memory than brute force and branch-and-bound, while branch-and-bound and brute force's memory usage does not differ significantly.

Looking ahead, future studies should test with more nodes for each algorithm. Furthermore, comparing the performance of the exact algorithm with the heuristic algorithm may also provide insights into the time and space usage of each algorithm. Therefore, we can further determine which algorithm is the best for certain scenarios of the Time-Dependent Traveling Salesman Problem.

G. GitHub and Presentation Link

GitHub: https://github.com/ellisraputri/ADA-FinalProject

Presentation:
https://www.canva.com/design/DAGbCMHJLQ8/JFntIrS1PUabxRUUwHyQsQ/edit?utm_content=DAGbCMHJLQ8&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

H. Member Contributions

Cassie: Paper (Introduction, Related Works), benchmarking process, PPT

Cindy: Paper (Introduction, Related Works), benchmarking process, PPT

Ella: Paper (Abstract, Methodology, Conclusion), branch-and-bound codes

Ellis: Paper (Methodology, Result and Discussion), brute-force codes, dynamic programming codes, statistic test

REFERENCES

[1] D. Wawan Saputra, “Optimalisasi Rute Distribusi Kurir Menggunakan Metode Traveling Salesman Problem (Studi Kasus: JNE Balige),” (in Indonesian), *G-Tech: J. Tek. T.*, vol. 6, no. 2, pp. 159–165, Aug. 2022, doi: 10.33379/gtech.v6i2.1577.

[2] David Biron, “The Traveling Salesman Problem: Deceptively Easy to State; Notoriously Hard to Solve,” 2006.

[3] M. S. N. Afif, M. I. A. Tsauri, and S. Hadiwijaya, “Optimisasi Rute Pengiriman Produk Komponen Otomotif (Traveling Salesman Problem) Melalui Pendekatan Heuristik,” (in Indonesian), *J. TEK. IND.*, vol. 3, no. 1, pp. 38–46, May 2022, doi: 10.37366/JUTIN0301.3846.

[4] R. Fontaine, J. Dibangoye, and C. Solnon, “Exact and anytime approach for solving the time dependent traveling salesman problem with time windows,” *Eur. J. Oper. Res.*, vol. 311, no. 3, pp. 833–844, Dec. 2023, doi: 10.1016/j.ejor.2023.06.001.

[5] K. L. Hoffman and M. Padberg, *Traveling salesman problem*, Springer US, 2001, p. 849–853.

[6] C. Rego, D. Gamboa, F. Glover, and C. Osterman, “Traveling salesman problem heuristics: Leading methods, implementations and latest advances,” *Eur. J. Oper. Res.*, vol. 211, no. 3, p. 427–441, Jun. 2011.

[7] S. Sangwan, “Literature review on Traveling salesman problem,” *Int. J. Res.*, vol. 5, p. 1152, 06 2018.

[8] M. Muneeb Abid and M. Iqbal, “Heuristic approaches to solve traveling salesman problem,” *Indones. J. Electr. Eng. Comput. Sci.*, vol. 15, pp. 390–396, 09 2015.

[9] D. Messon, D. Verma, M. Rastogi, and A. Singh, *Comparative Study of Time Optimization Algorithms for Traveling Salesman Problem*. Springer Nature Singapore, 2022, p. 555–566.

[10] A. Tripathy, “Traveling salesman problem, an approach to optimize the brute force method by dynamic programming and bit masking,” p. 10, 06 2020.

[11] J. Jureˇckov´a and J. Picek, “Shapiro–wilk-type test of normality under nuisance regression and scale,” *CSDA*, vol. 51, no. 10, p. 5184–5191, Jun. 2007.

[12] P. E. McKnight and J. Najab, “Mann-whitney u test,” p. 1–1, Jan. 2010.

[13] S. Rostami, S. Creemers, and R. Leus, “Precedence theorems and dynamic programming for the single-machine weighted tardiness problem,” *European Journal*

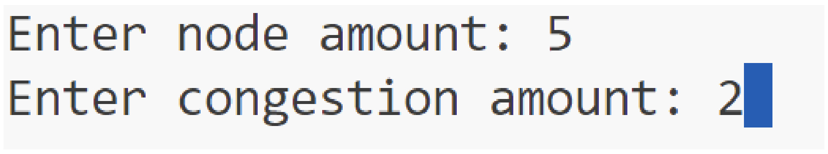
of Operational Research, vol. 272, no. 1, pp. 43–49, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221718305277

SCREENSHOTS AND USER MANUAL

Benchmarking (Time)

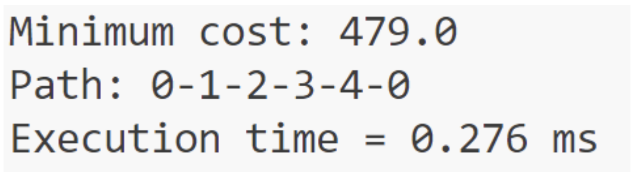
1. Go to folder Time, choose the algorithm you want to test (brute force: backtracking_time.py, dynamic programming: dp_time.py, branch-and-bound: bnb_time.py)

2. Run the program, and enter the node amount and congestion amount.



```
Enter node amount: 5
Enter congestion amount: 2
```

3. See the results

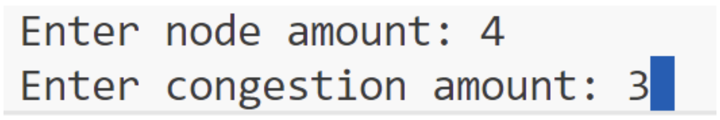


```
Minimum cost: 479.0
Path: 0-1-2-3-4-0
Execution time = 0.276 ms
```

Benchmarking (Space)

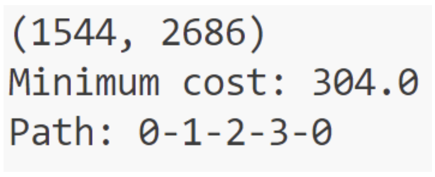
1. Go to folder Space, choose the algorithm you want to test (brute force: backtracking_space.py, dynamic programming: dp_space.py, branch-and-bound: bnb_space.py)

2. Run the program, and enter the node amount and congestion amount.



```
Enter node amount: 4
Enter congestion amount: 3
```

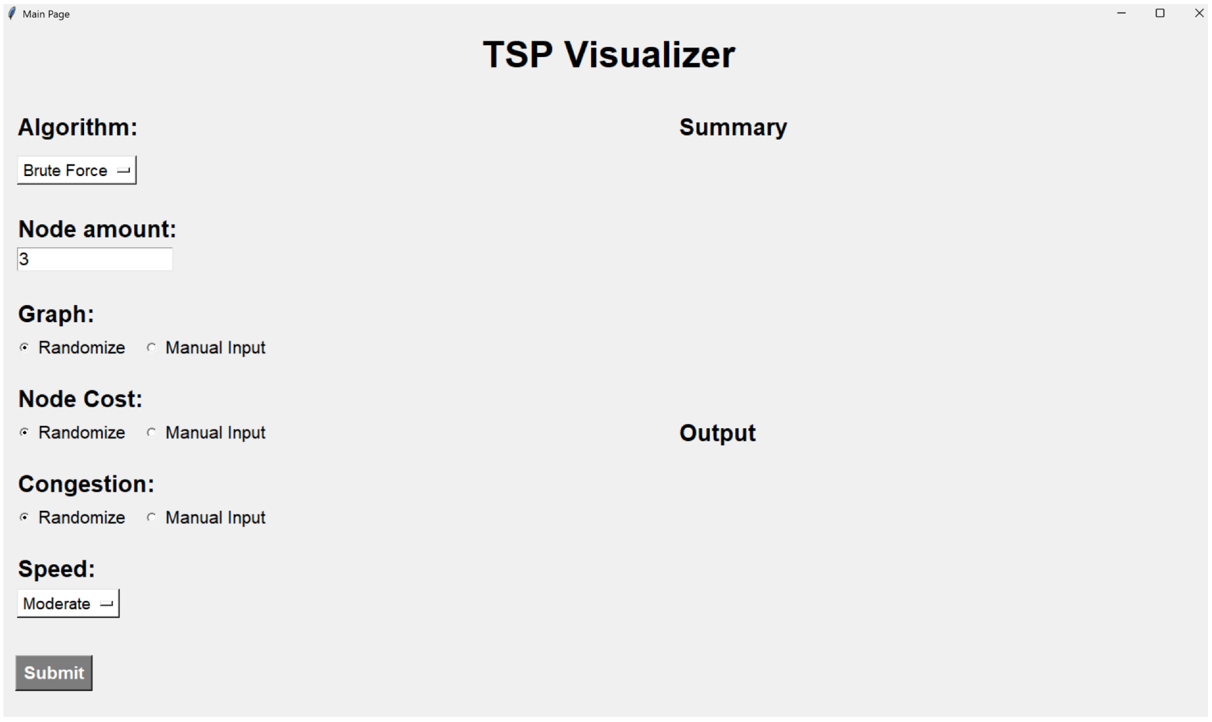
3. See the results. The first number in the tuple is the current memory used, while the second is the peak memory used.



```
(1544, 2686)
Minimum cost: 304.0
Path: 0-1-2-3-0
```

GUI

1. Go to folder GUI and run the mainGUI.py



Algorithm: Brute Force

Node amount: 3

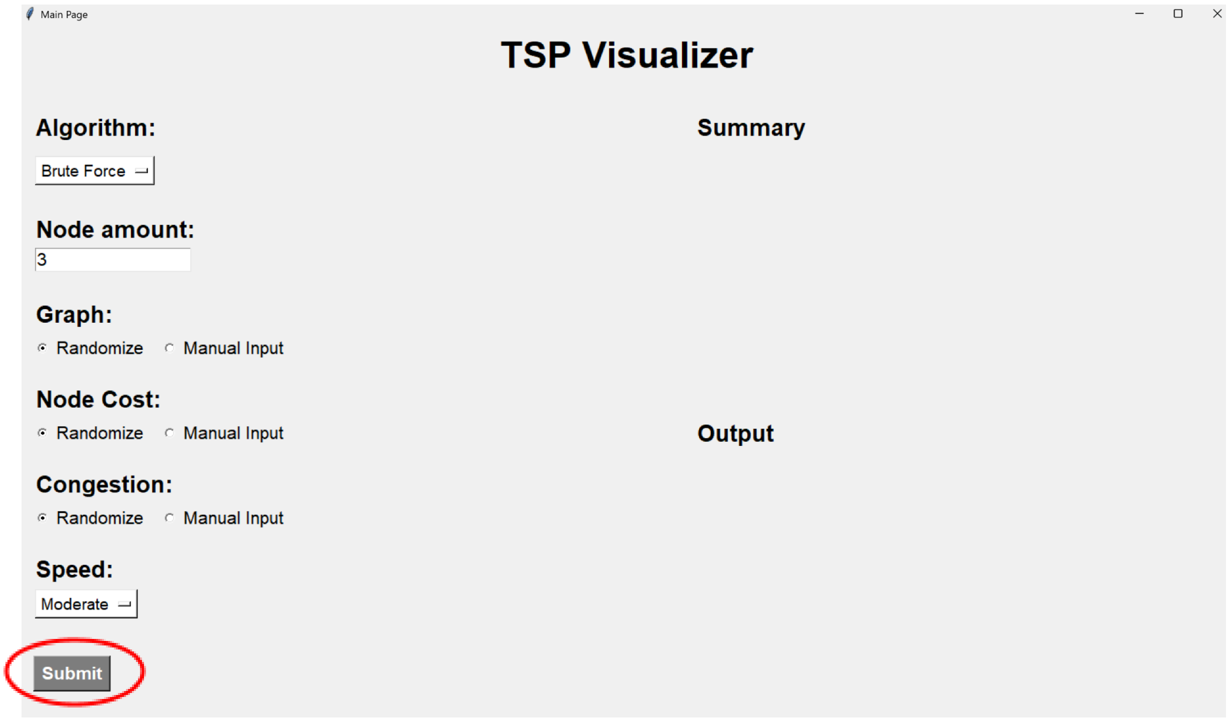
Graph: [✓ Randomize] Manual Input

Node Cost: [✓ Randomize] Manual Input

Congestion: [✓ Randomize] Manual Input

Speed: Moderate

2. Select the algorithm and visualization speed you want. Also, enter your node amount. Lastly, select whether the graph, node cost, and congestion are randomized or inputted manually. If done, click the submit button.



Algorithm: Brute Force

Node amount: 3

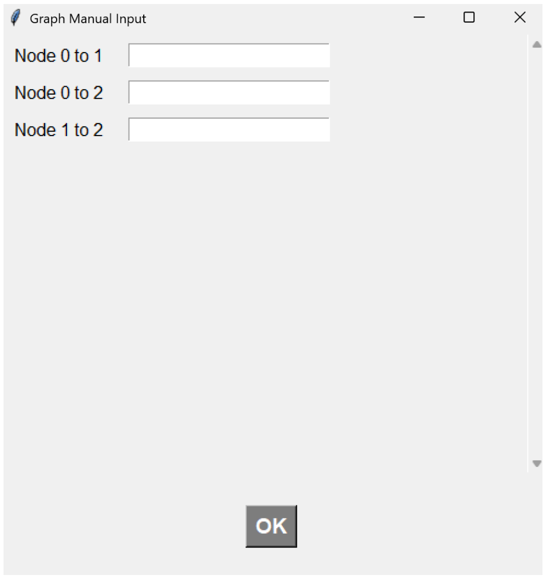
Graph: [✓ Randomize] Manual Input

Node Cost: [✓ Randomize] Manual Input

Congestion: [✓ Randomize] Manual Input

Speed: Moderate

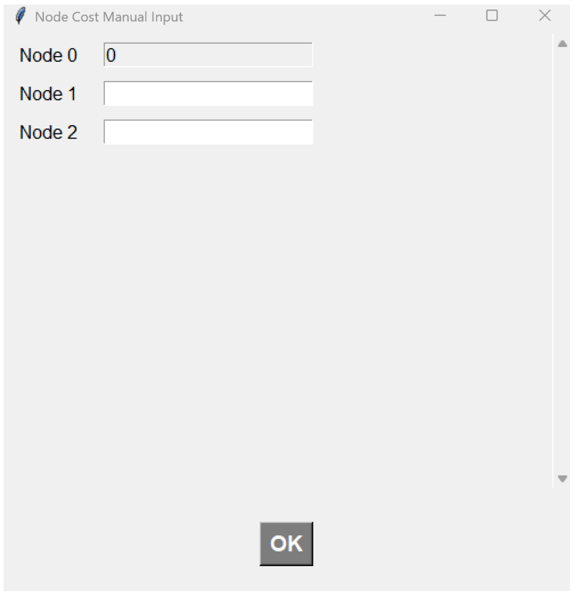
3. If you select the manual input for the graph, you need to input your graph edge weight manually.



Graph Manual Input

Node 0 to 1	
Node 0 to 2	
Node 1 to 2	

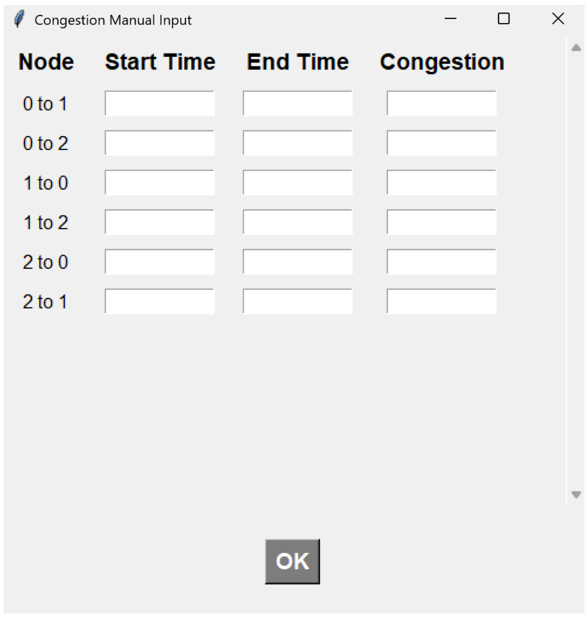
4. If you select the manual input for node cost, you need to input each node cost manually. Node 0 cost is set to zero since it is the starting point.



Node Cost Manual Input

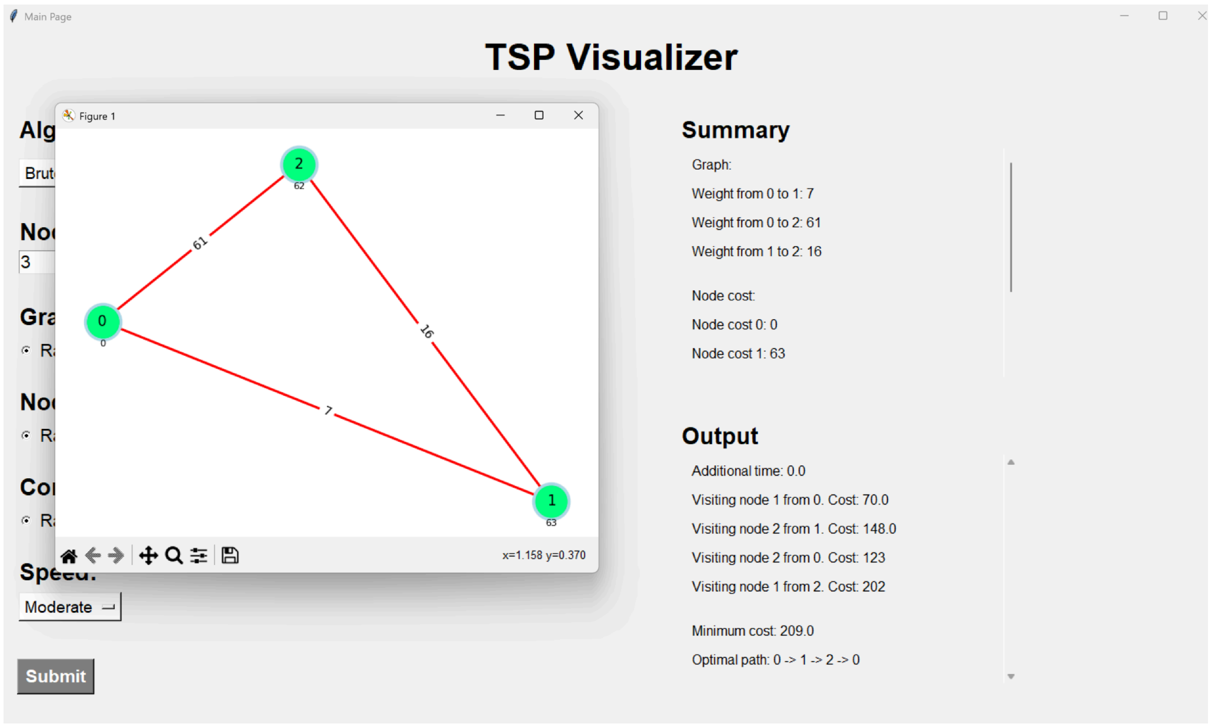
Node 0	0
Node 1	
Node 2	

5. If you select the manual input for congestion, you need to input all possible congestions manually. If you do not want congestion in that path, please input 0.


Congestion Manual Input

Node	Start Time	End Time	Congestion
0 to 1			
0 to 2			
1 to 0			
1 to 2			
2 to 0			
2 to 1			

6. After you click the submit button, you can see the graph visualization in a new small window. You can also see the summary of the graph weight, node cost, and congestion that is randomized or manually inputted. Lastly, you can see the output to help you understand what the visualization is currently doing.



Summary

Graph:

Weight from 0 to 1: 7

Weight from 0 to 2: 61

Weight from 1 to 2: 16

Node cost:

Node cost 0: 0

Node cost 1: 63

Output

Additional time: 0.0

Visiting node 1 from 0. Cost: 70.0

Visiting node 2 from 1. Cost: 148.0

Visiting node 2 from 0. Cost: 123

Visiting node 1 from 2. Cost: 202

Minimum cost: 209.0

Optimal path: 0 -> 1 -> 2 -> 0