OSTİM TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

SENG 313 - ANALYSIS OF ALGORITHMS
INSTRUCTOR: DR. ALPASLAN ERDAĞ

FINAL PROJECT

ESMANUR YORULMAZ - 210201008

CANAN KILIÇ - 220201037

08.01.2025

Question 1: Divide and Conquer

1. Introduction

The goal of this project is to improve how products are listed on an e-commerce platform by sorting them based on sales volume and user ratings. We will compare the performance of Merge Sort and Quick Sort using a dataset that includes product IDs, sales volumes, and ratings. The focus is on how fast each algorithm sorts the data, with tests on datasets containing 1,000, 5,000, and 10,000 records. By checking the execution time of both algorithms, we can decide which one sorts product data more efficiently. Merge Sort is a divide-and-conquer algorithm with a time complexity of O(n log n), and it guarantees stable sorting, but it requires extra space for merging. Quick Sort, also a divide-and-conquer algorithm, is generally faster than Merge Sort, but in the worst case, it has a time complexity of $O(n^2)$. Despite this, Quick Sort usually works better in real-world scenarios. The result will be a sorted list of products based on their sales volume and ratings.

2. Algorithm Design

Merge Sort: Merge Sort is a divide-and-conquer algorithm that works by breaking down the dataset into smaller sublists, continuing until each sublist contains only one element. Once this is done, the sublists are merged back together in sorted order. During the merging process, the algorithm ensures stability by comparing both the sales volume and ratings of the products. The process starts by dividing the dataset into two halves, and then both halves are recursively sorted. Afterward, the sorted halves are merged, making sure that the products are ordered based on both their ratings and sales volumes.

Quick Sort: Quick Sort is another divide-and-conquer algorithm, but it uses a different approach. It selects a pivot element, then partitions the dataset into two sublists: one with elements smaller than the pivot and the other with elements larger than the pivot. The algorithm then recursively sorts these sublists. The process begins by selecting a pivot element, followed by partitioning the dataset around the pivot. Finally, the same process is applied recursively to the left and right sublists.

3. Complexity Analysis

Merge Sort:

- **Time Complexity:** Best: O(n log n) | Average: O(n log n) | Worst: O(n log n)
  The time complexity is determined by two factors: the dataset is divided into smaller sublists, which takes O(log n), and the merging process for each pair of sublists takes O(n). Combining these operations gives O(n log n) as the overall time complexity.

- **Space Complexity:** O(n)
  Merge Sort requires extra space for storing temporary sublists during the merging process. As a result, the space complexity is O(n).

Quick Sort:

- **Time Complexity:** Best: O(n log n) | Average: O(n log n) | Worst: $O(n^2)$
  Quick Sort generally performs well, and its average time complexity is O(n log n), depending on the choice of pivot. In the worst case, when the pivot is poorly chosen (e.g., selecting the smallest or largest element), Quick Sort can degrade to $O(n^2)$.

- **Space Complexity:** O(log n)
  Quick Sort requires space for the call stack during recursion. This space is proportional to the depth of the recursive calls, which in the best case is O(log n).

3. Results & Findings

Execution times were measured for both Merge Sort and Quick Sort using datasets of 1,000, 5,000, and 10,000 records. The results were compared in terms of both speed and efficiency, and these were visualized in a graph that shows the execution times relative to the dataset sizes.

Performance Comparison:

- **Merge Sort:**
  Merge Sort is stable and offers a predictable time complexity of O(n log n). This makes it well-suited for sorting large datasets, although it requires additional space for the merging process. Its stable nature ensures that equal elements retain their relative order, which can be important in certain applications. The extra space requirement is a tradeoff for its consistent performance.
- **Quick Sort:**
  Quick Sort is generally faster than Merge Sort with an average time complexity of O(n log n). However, its performance can degrade to $O(n^2)$ in the worst case, especially if the pivot is not chosen optimally. Despite this, it is often faster in practice because of its smaller constant factors. Quick Sort has better space efficiency compared to Merge Sort, requiring only O(log n) space for the call stack during recursion. This makes it more suitable for memory-constrained environments.

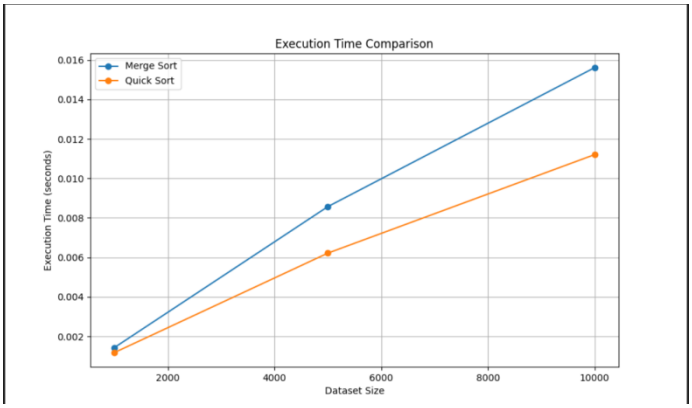
4. Conclusion

```
PS C:\Users\esman\OneDrive\Desktop\question_1> & C:/User
    Algorithm  Dataset Size  Execution Time (seconds)
0  Merge Sort          1000                  0.001430
1  Quick Sort          1000                  0.001176
2  Merge Sort          5000                  0.008569
3  Quick Sort          5000                  0.006214
4  Merge Sort         10000                  0.015611
5  Quick Sort         10000                  0.011205
PS C:\Users\esman\OneDrive\Desktop\question_1>
```

Quick Sort is faster in practice, especially for smaller datasets, because of its efficient partitioning. However, Merge Sort is stable, meaning it keeps the relative order of equal elements, and it has a guaranteed time complexity of O(n log n), making it more predictable.

The choice between the two depends on factors like dataset size, the importance of stability, and memory constraints. If stability matters or the dataset is large, Merge Sort might be better despite needing more memory. But for smaller datasets or when memory is tight, Quick Sort is usually the better choice.

Quick Sort generally performs better in execution time for all dataset sizes, especially with smaller datasets. Its average-case performance is faster, making it a common choice for many applications. But Merge Sort is stable and predictable, with a guaranteed O(n log n) time complexity, which is helpful when maintaining the order of equal elements is important.



For e-commerce platforms dealing with large datasets, Merge Sort may be preferred when stability and performance are key. But if speed is more important, Quick Sort would be a better choice, particularly when execution time matters more than keeping equal elements in order.

Question 2: Greedy Algorithm

1. Introduction

This report explains a way to schedule conference sessions by using a greedy algorithm. Each session has a fixed start time and end time, and no two sessions can happen at the same time. The aim is to pick the most sessions without any overlaps. The input is a list of sessions with their start and finish times, and the output is a list of selected sessions that gives the maximum number of events. The problem is solved by using a greedy method, which means choosing the best option at each step to get the highest number of sessions in the end.

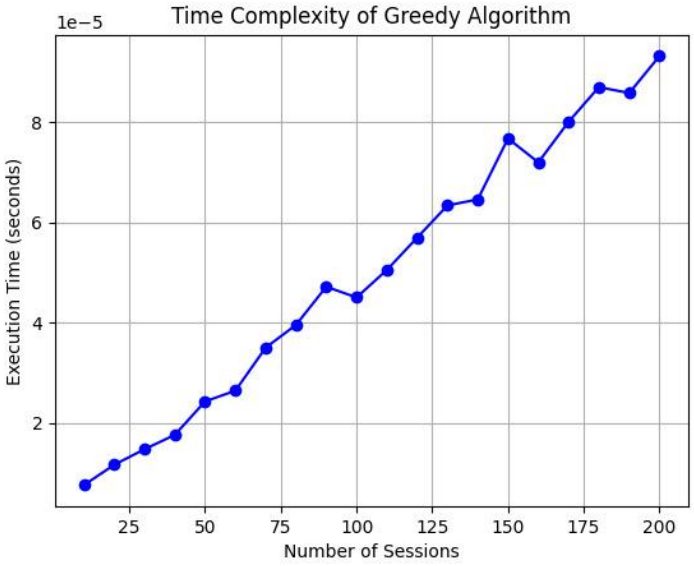
2. Algorithm Desigm

Greedy Algorithm: As the name is greedy this algorithm picks most efficient one at the time of selecting sessions. The greedy algorithm starts by sorting the sessions based on their end times. By choosing sessions that finish the earliest, we leave more place for subsequent sessions. Firstly the algorithm ensures while selecting suitable sessions to leave maxiumum available time for others. Secondly it initialize an empty list for selected sessions, iteratively it adds to the list sessions that are suitable.

3. Complexity Analysis

Time Complexity: Best Case: O(n log n) | Average Case: O(n log n) | Worst Case: O(n log n)

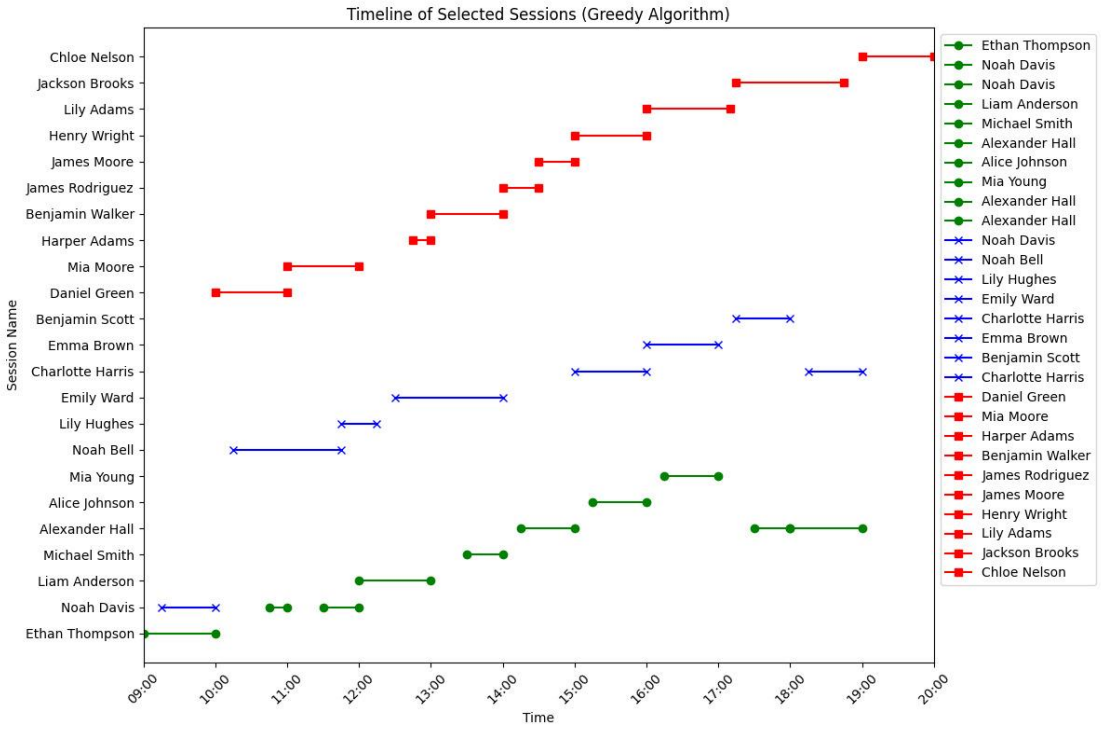
The main step that takes time is sorting the sessions by their end times, which needs O(n log n), where n is the total number of sessions. Sorting ensures sessions are handled in an order that allows more events to fit later. So, the total time complexity is O(n log n).This time complexity means the algorithm works well even for large inputs, especially when n is in the range of thousands, because the log factor increases slower than n.
A graph is used to show how input size (number of sessions) affects execution time. The graph confirms the O(n log n) complexity by showing that execution time rises logarithmically with the number of sessions.



Space Complexity: O(n)
The space complexity is O(n) because we store the selected sessions in a list. Extra space is used for the input data and some temporary variables, but this adds only a linear amount. So, the total space complexity stays O(n).

4. Results & Finding

Empirical Performance on Graph:



As it is seen in the graph the algorithm was tested on three different datasets, each with unique characteristics. In the first dataset, sessions had varying start and end times, and the algorithm correctly selected the maximum number of non-overlapping sessions. For the second and third dataset, where sessions had vary start and end times, the greedy approach still managed to handle smaller gaps effectively and produced an optimal list of non-overlapping sessions. The results confirm that sorting by end time allows the algorithm to consistently maximize the number of scheduled sessions by always leaving room for future events.

5. Conclusion

The greedy algorithm works well for solving the task scheduling problem. It always picks the session that ends the earliest, so there's more space left for other sessions. This method gives the best result while keeping the time complexity low, making it good for handling large datasets. To improve the algorithm, it should handle special cases like sessions with the same start and end times, and there should be checks to confirm if the input data is true. In the future, dynamic programming can be explored for harder problems like weighted datasets.

The greedy approach is efficient despite may not always work for more complex scheduling problems, such as those requiring weighted intervals or prioritization based on other factors. Additionally, it can fail to provide the best solution if the problem involves constraints that cannot be resolved by making local optimal choices at each step. This problems can be optimized.

Question 3: Dynamic Programming

1. Introduction

This project creates a plagiarism detection tool using the Longest Common Subsequence (LCS) algorithm. It compares pairs of text documents, finding the LCS to spot similarities or potential plagiarism. The tool reads data from two CSV files, compares each pair of documents, and shows the LCS length and the common subsequences. The main goal is to find large sections of text that are the same across documents to check if there is any plagiarism happen.

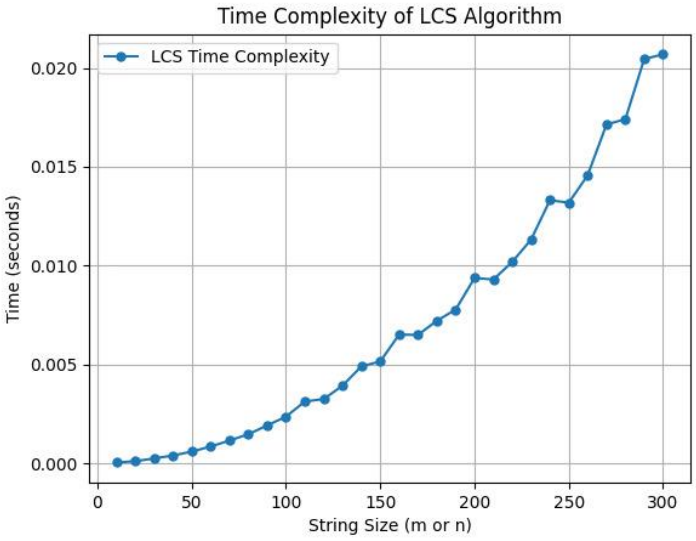
2. Algorithm Design

Longest Common Subsequence: The algorithm uses dynamic programming (DP) to calculate the length of the Longest Common Subsequence (LCS) between two strings. A table, dp[i][j], is created where i and j represent the indices of the two strings being compared. The table is filled by checking if the characters s1[i-1] and s2[j-1] match; if they do, dp[i][j] is set to dp[i-1][j-1] + 1. If not, dp[i][j] takes the maximum of dp[i-1][j] and dp[i][j-1]. This method ensures the LCS length is computed in O(m * n) time, where m and n are the lengths of the two strings. Once the LCS length is calculated, the algorithm uses a backtracking approach to find all possible LCS sequences. The backtracking function explores all possible paths in the DP table that lead to an LCS, building a set of LCS sequences by adding characters when a match is found and exploring multiple subsequences when there are ties. The time complexity of this step can be exponential in the worst case, depending on the number of subsequences.

3. Complexity Analysis

Time Complexity: Best Case: O(m * n) | Average Case: O(m * n) | Worst Case: O(m * n)

The time complexity of the LCS algorithm mainly comes from the dynamic programming (DP) table, which is computed in O(m * n) time, where m and n are the lengths of the two strings. This is the biggest part of the algorithm, so when comparing two documents, the total time complexity is O(m * n). If there are k pairs of documents, the time complexity would be O(k * m * n), where m and n are the average lengths of the documents.

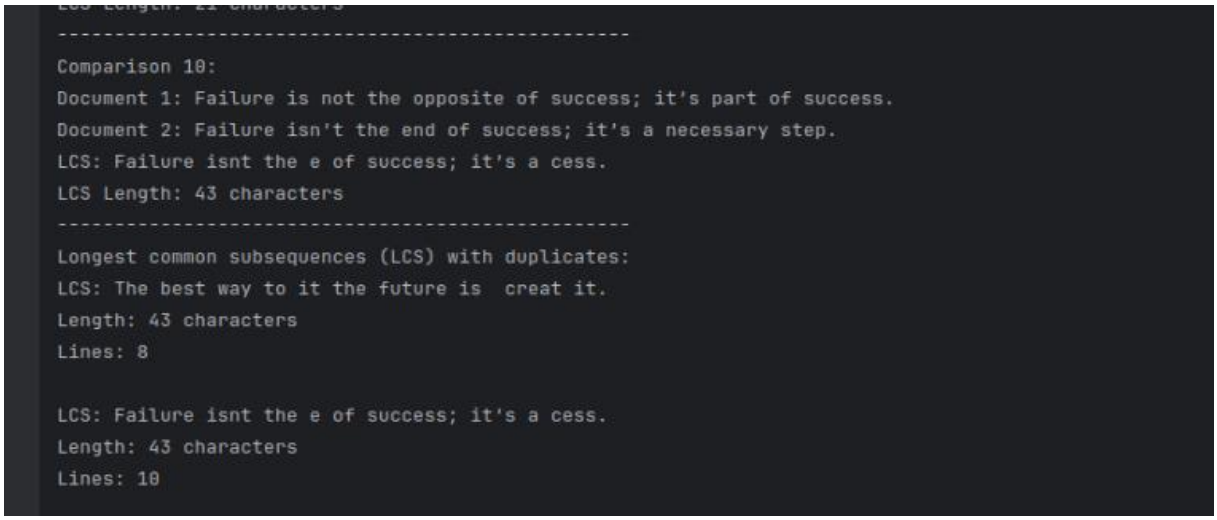


The graph shows the time complexity of the LCS algorithm with dynamic programming. The x-axis represents the size of the strings (m and n), and the y-axis shows the time taken to calculate the LCS. As the size of the strings gets bigger, the execution time grows quadratically, which matches the O(m * n) complexity. This means when the length of the strings doubles, the time to compute the LCS roughly quadruples. The graph confirms that the LCS algorithm has quadratic time complexity, meaning that larger strings need much more time to process.

Space Complexity: O(m * n)

The space complexity of the algorithm is O(m * n), mainly due to the dynamic programming (DP) table. The DP table stores the intermediate results when comparing the two strings. Since the table's size depends on the lengths of the two strings, it has m * n cells, where m and n are the lengths of the two strings. This is the main memory requirement for the algorithm. So, as the input strings get larger, the memory needed also grows. This can be a problem if the strings are very large, but for smaller inputs, it works just fine.

4. Results & Finding



The program shows the LCS length and sequences for each document pair. Longer LCS lengths mean the documents are more similar, which might show potential plagiarism. Multiple LCS sequences show parts of the documents that have the same text, helping to find plagiarized sections.In all lines the longest common subsequence found and written as output.

5. Conclusion

The tool works well for detecting plagiarism, but there are some areas that could be improved. For larger datasets, the algorithm might not perform well. Using more advanced methods like suffix trees or suffix arrays could help with this. Preprocessing the text, such as removing stop words and punctuation, could make the LCS comparison more accurate. For comparing many documents, parallel processing could speed up the work. The backtracking approach used now can generate many subsequences, so improving it by pruning unnecessary paths could reduce computation time.

To improve the tool, we could use more efficient algorithms for large documents, like suffix arrays, for faster LCS calculation. For handling many document comparisons, we could parallelize the task. If memory becomes an issue, optimizing how we store the DP table and improving the backtracking function could help. At the end, adding input checks for things like empty documents or invalid files would make the tool more reliable.

Question 4: Graph Algorithm

1. Introduction

This project is about finding the best delivery routes between cities in Turkey, where the main thing we focus on is distance. The project includes creating a solution that finds the shortest routes using Dijkstra's algorithm, shows these routes, and allows changes like adding or deleting roads between cities. System extended to handle live route updates, like adding or removing roads while it's running.

2. Algorithm Design

To solve the problem of finding the shortest delivery routes, we use *Dijkstra's Algorithm*. This algorithm is chosen because it efficiently finds the shortest path from a starting city to all other cities using a greedy approach. It works by repeatedly selecting the city with the smallest current distance, updating its neighboring cities' distances, and marking it as visited. First, we set the distance of each city to infinity, except for the starting city, which is set to 0. Then, we use a priority queue to pick the city with the smallest current distance. For the selected city, we update the distances of its neighboring cities. This process repeats until all cities have been visited.

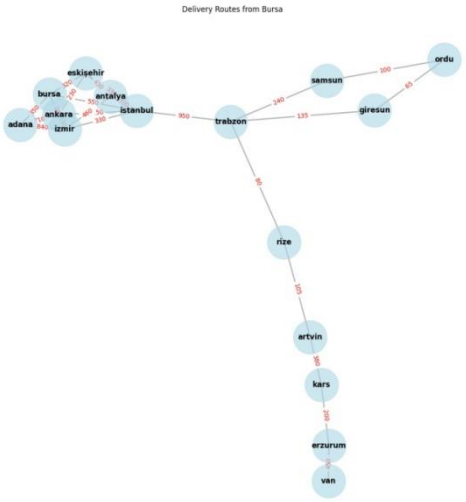
3. Complexity Analysis

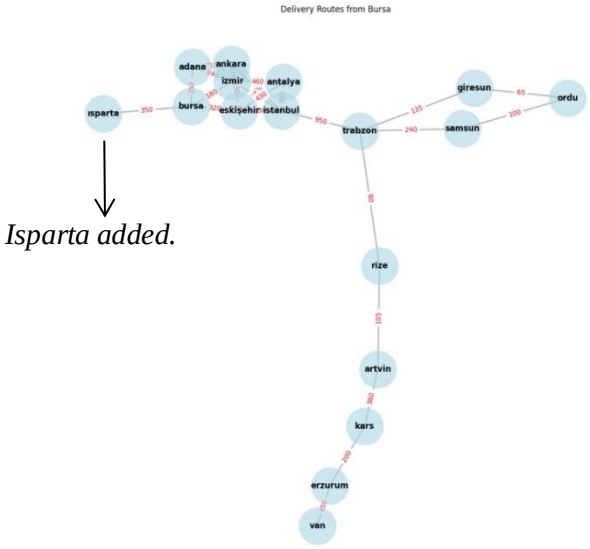
Time Complexity: Best Case**:** O(V * log V)| Average Case: O((V + E) * log V) |
Worst Case: O((V + E) * log V)
The time complexity for the algorithm is as follows: in the worst case, it is O(E log V), where E is the number of edges (roads) and V is the number of vertices (cities). This happens because, in each step, the algorithm performs a priority queue operation, such as insertion and extraction, which takes logarithmic time in the number of vertices. In the best case, the time complexity is O(V log V), which occurs when the graph is sparse and has few edges. For the average case, the time complexity is O(E log V), which is the general scenario where the graph has a normal number of edges and vertices. The space complexity of the algorithm is O(V + E). It uses a priority queue, a distance array, and a set to track visited cities, so the space complexity is proportional to the number of cities and roads in the graph.
Space Complexity: O(V + E). The algorithm uses a priority queue, a distance array, and a set to track visited cities. So, the space complexity depends on the number of cities and roads in the graph.

4. Results & Finding

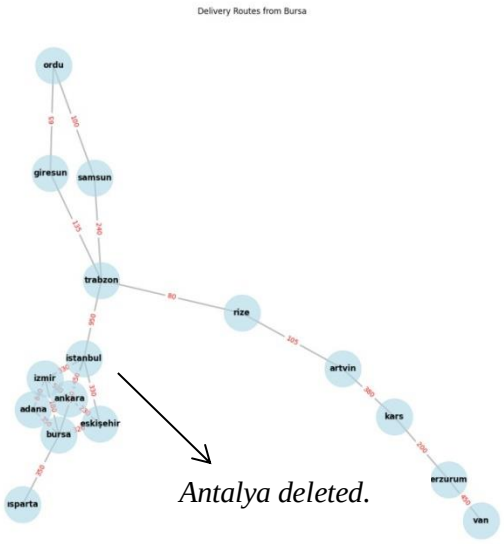
In this section, three stages of graph visualization are shown, each reflecting the state of the graph after different operations: the initial graph, after adding a road, and after deleting a road.

The initial graph shows the delivery routes between cities based on the original data, with roads between all city pairs and their distances marked. This layout shows the original connectivity and distances.this section, three different stages of graph visualization are presented, each reflecting the state of the graph after various operations (initial graph, after adding a road, and after deleting a road).





Isparta added.



Antalya deleted.

After adding a road, a new connection is made between Bursa and Isparta with a distance of 350 units. The graph is updated to include this road, and the distances are recalculated. This new road changes some of the shortest delivery routes, shortening some of the previously longer ones.

After deleting Antalya, several roads to Antalya are removed too, including the ones between Istanbul, Izmir, Eskişehir. The graph reflects these changes, and some cities are no longer directly connected, so alternative routes may need to be found

5. Conclusion

One of the challenges was keeping the graph accurate after adding or removing roads. When roads are added or deleted, the graph must always reflect the current network of cities to ensure the routes are calculated correctly. The algorithm works well for reasonably sized graphs, but as the network grows, performance can slow down. Optimizing it to handle larger datasets, maybe by using more advanced structures like Fibonacci heaps, could help improve speed. Another challenge was handling how dynamic changes affect other roads. Adding or deleting a road can impact other delivery routes, requiring recalculations and adjustments throughout the network. Also, the visualization of graph algorithms is key to understanding how these changes affect delivery routes. As the number of cities grows, it becomes harder to manage overlapping labels and keep the graph easy to read.

Question 5: Graph Algorithm

1. Introduction

This problem aims to optimize internet traffic within a data center by calculating the maximum flow in a network using the Ford-Fulkerson algorithm. The network is represented as a directed graph, with source and target nodes specified by the user.

2. Algorithm Design

The Ford-Fulkerson algorithm is designed to find the maximum flow in a flow network by iteratively searching for augmenting paths using Breadth-First Search (BFS). The algorithm adjusts the flow along these paths until no more augmenting paths are available. In this implementation, two data structures are used: an adjacency matrix and an adjacency list.

3. Complexity Analysis:

- **Time Complexity**: The time complexity of the Ford-Fulkerson algorithm is **O(E * max_flow),** where E is the number of edges and **max_flow** is the maximum flow of the network. Each BFS operation has a complexity of **O(V + E)** and is repeated until the maximum flow is reached. (O(E * max_flow))
- **Space Complexity**: For the adjacency matrix representation, the space complexity is **O(V^2)**, as it requires storing capacities for all possible vertex pairs. For the adjacency list representation, the space complexity is **O(V + E)**, which is more efficient for sparse graphs. (Adjacency Matrix: $O(V^2)$, Adjacency List: O(V + E))

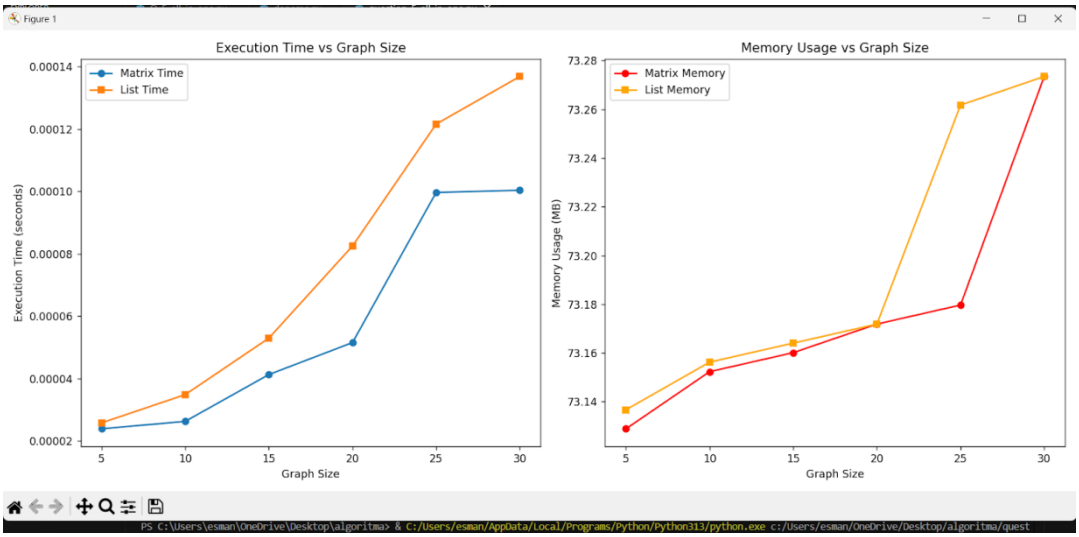
4. Results & Findings

Graphs and Interpretation:

The following graphs illustrate the performance of the Ford-Fulkerson algorithm for different input sizes, showcasing both execution time and memory usage. The graph visualizations help in understanding how the algorithm scales with varying network sizes. There is a noticeable difference between the theoretical time complexity and the observed performance, particularly as the input size increases.

Execution Time vs. Graph Size (Left Graph):

As the graph size increases, the execution time for both the adjacency matrix and adjacency list representations grows. However, the adjacency list shows consistently higher execution times compared to the adjacency matrix. This discrepancy likely arises due to differences in how the two structures handle edge lookups and updates, with the adjacency matrix benefiting from constant-time edge access at the cost of increased memory usage.



Memory Usage vs. Graph Size (Right Graph):

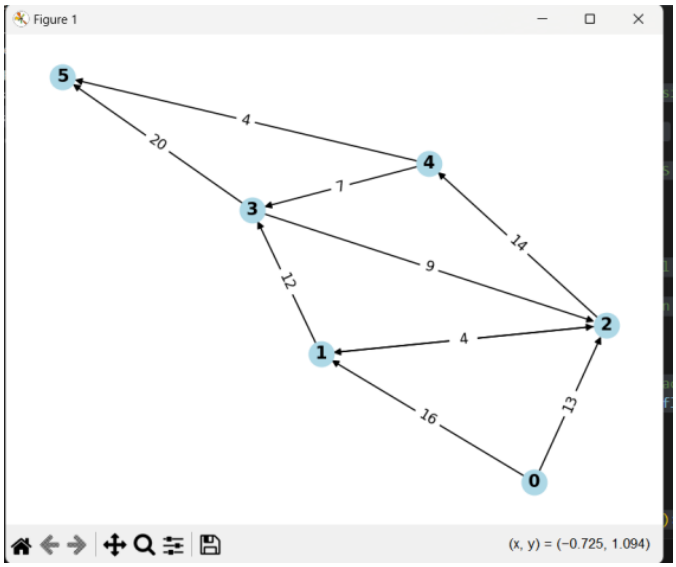
Memory usage also scales with graph size, but the adjacency matrix exhibits a sharper increase in memory usage compared to the adjacency list. This aligns with expectations since the adjacency matrix requires storage for all possible edges, while the adjacency list only stores edges that exist in the graph.

4. Conclusion and Comparison:

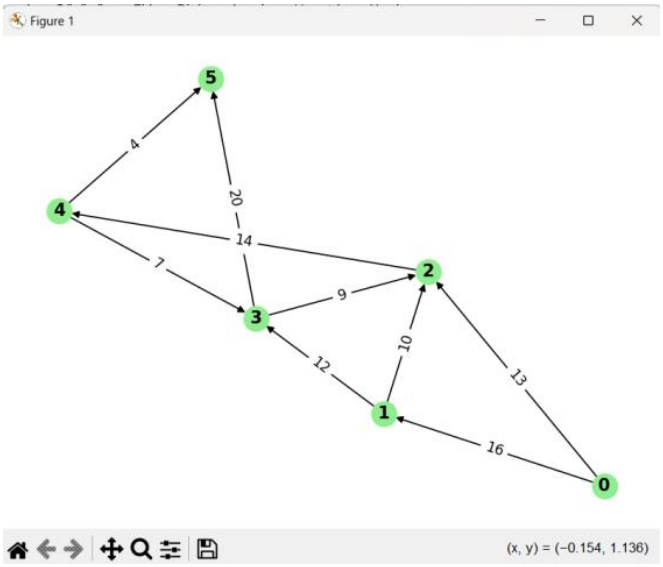
The choice between the adjacency list and adjacency matrix depends on the nature of the graph and the specific use case. The adjacency list is ideal for large sparse graphs, offering efficient memory usage, albeit with some performance trade-offs in terms of edge traversal. On the other hand, the adjacency matrix provides faster edge operations, but it comes with a higher memory cost, especially for larger graphs. When comparing theoretical predictions to observed performance, discrepancies can arise, particularly due to factors like graph density and hardware characteristics. Sparse graphs generally align more closely with theoretical expectations, while denser graphs show more pronounced deviations due to the added computational overhead. In our specific tests, adjacency list showed higher execution times for dense graphs. This may occur due to the specific characteristics of the graph, the implementation details, or the limited size of the dataset used in our experiments.

Outputs:

Matrix



List



Course Feedback Submission

1) In this course, we learned how to improve algorithms by optimizing their time and space complexity. We also learned how to analyze algorithms using Big O notation to understand how they perform with different data sizes. Algorithms are crucial for solving problems efficiently, and understanding them can greatly improve performance. We really thank you teacher, for guiding us and share your experiences.

2) The most challenging part of the project was implementing the Ford-Fulkerson algorithm for maximum flow in a network. Understanding key concepts like augmenting paths and residual graphs was hard at first, as they felt complicated. It took time to apply these ideas in code. Handling edge cases, like cycle nodes or disconnected nodes, was also difficult and needed a lot of trying.

3) This course helped us understand algorithm design and analysis by introducing different algorithms and giving us plenty of practice. We have improved our skills and can now confidently analyze problems and choose the right algorithm for different situations.

REFERENCES:

Question 1: Divide and Conquer

Visualgo. Sorting Algorithms. Retrieved from https://visualgo.net/en/sorting
GitHub. Merge Sort Implementation in Python. Retrieved from https://github.com/TheAlgorithms/Python/blob/master/sorts/merge_sort.py
GitHub. Quick Sort Implementation in Python. Retrieved from https://github.com/TheAlgorithms/Python/blob/master/sorts/quick_sort.py

Question 2: Greedy Algorithm

Khan Academy. Greedy Algorithms. Retrieved from https://www.khanacademy.org/computing/computer-science/algorithms/greedy-algorithms/a/greedy-algorithms
OpenAI. ChatGPT - Random Dataset Generation for Conference Scheduling. Retrieved from https://chat.openai.com/
StackOverflow. Greedy Algorithms and Task Scheduling. Retrieved from https://stackoverflow.com/questions/17378487/greedy-algorithm-for-task-scheduling
Kaggle. Datasets for Algorithm Practice. Retrieved from https://www.kaggle.com/datasets
Kaggle. Conference Scheduling Dataset. Retrieved from https://www.kaggle.com/datasets

Question 3: Dynamic Programming

GeeksforGeeks. Longest Common Subsequence (LCS). Retrieved from https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/
GitHub. Dynamic Programming - Longest Common Subsequence. Retrieved from https://github.com/TheAlgorithms/Python/blob/master/dynamic_programming/longest_common_subsequence.py
Kaggle. Plagiarism Detection Using LCS. Retrieved from https://www.kaggle.com/datasets

Question 4: Graph Algorithm (Dijkstra's Algorithm)

Visualgo. Dijkstra's Algorithm. Retrieved from https://visualgo.net/en/graph
GitHub. Dijkstra's Algorithm Implementation in Python. Retrieved from https://github.com/TheAlgorithms/Python/blob/master/graphs/dijkstra.py

Question 5: Graph Algorithm (Ford-Fulkerson Algorithm)

GeeksforGeeks. Ford-Fulkerson Algorithm for Maximum Flow Problem. Retrieved from https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/
GitHub. Ford-Fulkerson Algorithm for Maximum Flow. Retrieved from https://github.com/TheAlgorithms/Python/blob/master/graphs/ford_fulkerson.py