COMPSCI 2XC3 : COMPUTER SCIENCE PRACTICE AND EXPERIENCE : ALGORITHMS AND DESIGN

Group Final Project

Sana Muhammad Ashraf
Student ID: 400477584

Sriya Dhanvi Mokhasunavisu
Student ID: 400430396

Prakhar Saxena
Student ID: 400451379

April 2024

Table of Contents

1 Single source shortest path algorithms **5**

1.1 Part 1.1 5

1.2 Part 1.2 6

1.3 Part 1.3 7

2 All-pair shortest path algorithm **11**

3 A* algorithm **13**

3.1 Part 3.1 14

3.2 Part 3.2 14

4 Compare Shortest Path Algorithms **16**

5 Organize your code as per UML diagram **18**

6 Appendix **20**

Table of Figures

1 Single source shortest path algorithms

1.3 Part 1.3 – Figure 1 : Time Complexity 8

1.3 Part 1.3 – Figure 2 : Space Complexity 10

2 Compare Shortest Path Algorithms

2.1 Part 4.3 – Figure 3 : Dijkstra vs. A* Algorithm 17

Executive Summary

This comprehensive report explores various aspects of shortest path algorithms, focusing on implementations, analyses, and comparisons of algorithms like Dijkstra's, Bellman-Ford, A*, and all-pair shortest path algorithms.

The report begins by detailing the implementation and analysis of modified versions of Dijkstra's and Bellman-Ford algorithms, considering relaxation constraints and their impacts on pathfinding efficiency and accuracy. Experiment designs for analyzing time and space complexities are proposed and executed, providing valuable insights into algorithm performance under different scenarios.

Furthermore, the report introduces the all-pair shortest path algorithm, which combines Bellman-Ford and Dijkstra's algorithms to handle both positive and negative edge weights efficiently. Complexity analyses reveal the algorithm's performance characteristics, particularly in dense graphs, where its time complexity is dominated by Bellman-Ford's $\Theta(V^3)$.

The discussion extends to the A* algorithm, highlighting its advantages over Dijkstra's algorithm, particularly in scenarios where heuristic functions provide accurate estimates of the remaining distance to the goal. Empirical testing strategies, comparisons with Dijkstra's algorithm, and considerations for arbitrary heuristic functions are presented.

Additionally, the report provides insights into the practical applications of A* in various domains, such as satellite navigation systems, robotics, and network routing, emphasizing its efficiency in finding shortest paths.

Finally, the report compares Dijkstra's and A* algorithms' performance on the London Subway system, demonstrating A*'s superior efficiency in scenarios with accurate heuristic guidance and its comparable performance otherwise. The report concludes by discussing design principles and patterns used in organizing the code and proposes further implementations to enhance graph representations.

Overall, this report serves as a comprehensive guide to understanding, implementing, and analyzing shortest path algorithms, offering valuable insights into their performance and practical application

1 Single Source shortest path algorithms

In this report, we delve into the implementation and analysis of variations of two well-known shortest path algorithms: Dijkstra's algorithm and Bellman-Ford algorithm. The implementation aims to return the shortest distance and path from a given source node to all other nodes in the graph, considering this relaxation constraint.

1.1 Dijkstra's Algorithm

Part 1.1 focuses on the implementation of a modified version of Dijkstra's algorithm. Unlike the traditional Dijkstra's algorithm, where nodes are relaxed until all shortest paths are found, the variation introduced here imposes a restriction on the number of relaxations allowed for each node. This restriction, denoted by 'k', ensures that each node can be relaxed at most 'k' times, where 'k' is a positive integer less than the total number of nodes minus one.

1.1.1 Implementation

The implementation of dijkstra(graph, source, k) follows these steps:

(i) Initialize a priority queue (min heap) to store nodes based on their tentative distances from the source node.

(ii) Initialize a dictionary to store the tentative distances and paths for each node, with initial values set to infinity and an empty path.

(iii) Enqueue the source node into the priority queue with a tentative distance of 0.

(iv) Repeat the following steps until the priority queue is empty or until each node has been relaxed 'k' times:

a. Dequeue the node with the smallest tentative distance from the priority queue.

b. For each neighboring node of the dequeued node, if relaxing the edge between the dequeued node and the neighbor results in a shorter path, update the tentative distance and path accordingly.

c. Enqueue the neighbor node into the priority queue with its updated tentative distance.

(v) After processing all nodes or reaching the relaxation limit, return the dictionary containing the shortest distances and paths from the source node to all other nodes.

1.1.2 Functionality

The functionality of the implementation enables users to:

Find shortest paths in graphs with limited relaxation opportunities for each node.
Explore how varying relaxation limits impact the resulting shortest paths and their accuracy.
Obtain the shortest distance and path from a specified source node to all other nodes, considering the relaxation constraint.

1.1.3 Results

The results obtained from running the dijkstra(graph, source, k) function provide insights into the shortest paths achievable within the specified relaxation limit. Users can analyze these results to understand the impact of relaxation constraints on pathfinding efficiency and accuracy in various graph scenarios.

1.2 Bellman-Ford Algorithm

In Part 1.2, we implemented a variation of the Bellman-Ford algorithm that imposes a restriction on the number of relaxations allowed for each node. The implementation, named bellman_ford(graph, source, k), takes three parameters: the graph itself, the source node from which to find the shortest paths, and the relaxation limit k. The function returns a dictionary containing the shortest distance and path from the source node to all other nodes in the graph, considering the relaxation constraint.

1.2.1 Implementation

The implementation of bellman_ford(graph, source, k) follows these steps:

a. Initialize a dictionary to store the tentative distances and paths for each node, with initial values set to infinity and an empty path.
b. Set the tentative distance of the source node to 0.
c. Repeat the following steps 'k' times:
   Iterate over each edge in the graph and relax it: for each edge (u, v) with weight w, if the distance to node u plus the weight w is smaller than the current tentative distance to node v, update the tentative distance and path accordingly.
d. After 'k' iterations, return the dictionary containing the shortest distances and paths from the source node to all other nodes.

1.2.2 Functionality

The functionality of the implementation enables users to:

a. Find shortest paths in graphs with limited relaxation opportunities for each node.
b. Explore how varying relaxation limits impact the resulting shortest paths and their accuracy.
c. Obtain the shortest distance and path from a specified source node to all other nodes, considering the relaxation constraint.

1.2.3 Results

The results obtained from running the bellman_ford(graph, source, k) function provide insights into the shortest paths achievable within the specified relaxation limit. Users can analyze these results to understand the impact of relaxation constraints on pathfinding efficiency and accuracy in various graph scenarios.

1.3 Experiment Design for Part 1.1 and 1.2:

To analyze the performance of the implementations from Part 1.1 (modified Dijkstra's algorithm) and Part 1.2 (modified Bellman-Ford algorithm), we propose the following experiment design. This experiment aims to evaluate the algorithms' performance under various conditions, including graph size, graph density, and the relaxation limit **k**.

1.3.1 Experiment Setup for Time Complexity and Accuracy

a. **Number of Runs:** We define the variable runs with a value of 10, indicating the number of iterations to perform for the experiment.
b. **Time Measurements:** We initialize empty lists dj_time and bf_time to store the execution times of Dijkstra's and Bellman-Ford algorithms, respectively.
c. **Graph Density:** We define a list density containing values representing different graph densities ranging from 0.1 to 1.
d. **Lists for Storing Results:** We initialize empty lists node_list and k_list to store randomly generated numbers of nodes and relaxation limits 'k' for each run.
e. **Difference Counter:** We initialize a variable differences to keep track of the number of cases where the distances dictionaries from Dijkstra's and Bellman-Ford algorithms disagree.
f. Main Loop:

We iterate runs times using a for loop.

Inside the loop:

(i) We generate a random number of nodes (nodes) between 10 and 51 and append it to node_list.
(ii) We create a weighted graph (graph) with the generated number of nodes and generate a random directed graph with the specified density.
(iii) We generate a random relaxation limit 'k' between 1 and the number of nodes minus one, and append it to k_list.
(iv) We measure the execution time of Dijkstra's algorithm (dijkstra(graph, 0, k)) using the timeit.default_timer() function before and after the algorithm execution, storing the elapsed time in dj_time.
(v) We measure the execution time of Bellman-Ford algorithm (bellman_ford(graph, 0, k)) using the same method and store the elapsed time in bf_time.
(vi) We print information about each test case, including the number of nodes, the value

of 'k', and the graph density.

(vii) We compare the distances dictionaries obtained from Dijkstra's and Bellman-Ford algorithms, incrementing the differences variable if they disagree.

g. **Result Analysis:** After the loop, we print the number of cases where the algorithms disagree out of the total number of runs.

1.3.2 Observation

The following are the details of the randomly generated test cases used in the experiment.

Test Case	Number of Nodes	Value of k	Density
1	23	17	0.1
2	26	14	0.2
3	26	5	0.3
4	19	16	0.4
5	32	16	0.5
6	46	5	0.6
7	37	29	0.7
8	20	14	0.8
9	14	11	0.9
10	27	20	1

The time taken by Dijkstra and Bellman-Ford algorithm on each of the test cases are represented in a bar graph as shown below:

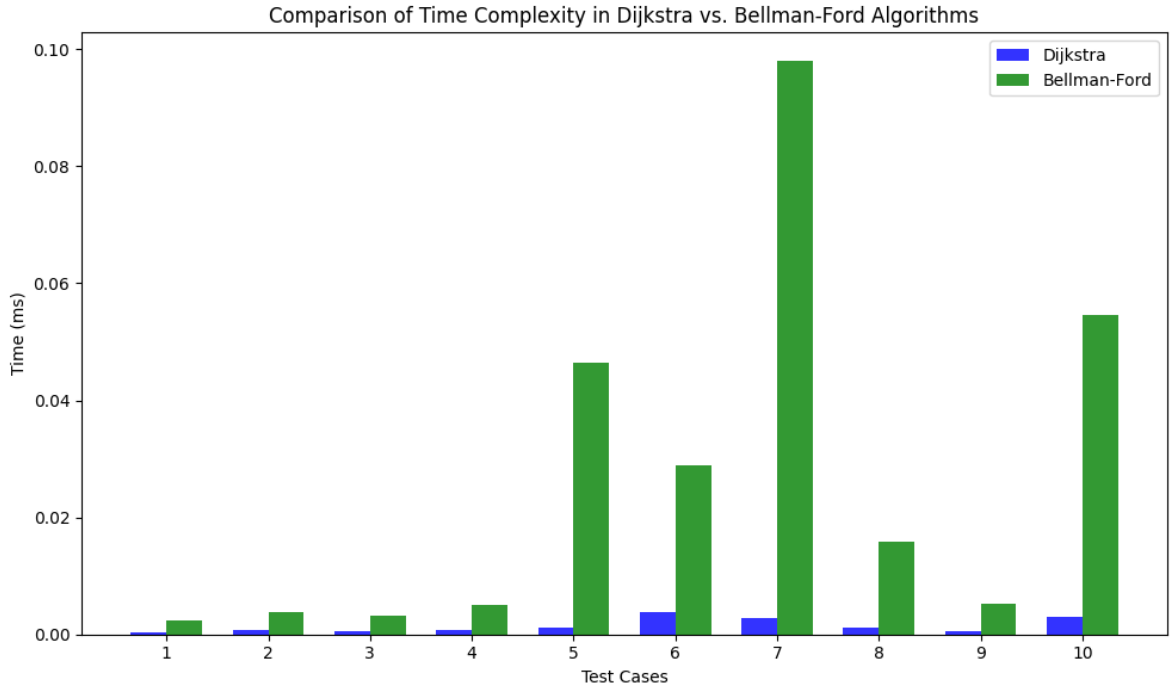

Figure 1: Dijkstra vs. Bellman-Ford Algorithm (1.3.2)

Following are the observations made based on the results:

- Dijkstra's algorithm generally exhibits shorter execution times compared to Bellman-Ford algorithm for most test cases.
- In particular, for test cases with smaller graphs and lower relaxation limits, Dijkstra's algorithm tends to perform faster.
- For example, in Test Case 1, Dijkstra's algorithm took approximately 0.00038 seconds, while Bellman-Ford algorithm took approximately 0.00244 seconds.
- Bellman-Ford algorithm shows longer execution times, especially for larger graphs and higher relaxation limits.
- The execution time increases significantly as the size of the graph and the relaxation limit 'k' increase.
- For instance, in Test Case 5, Bellman-Ford algorithm took approximately 0.046 seconds, while Dijkstra's algorithm took only 0.00122 seconds.
- The comparison between Dijkstra's and Bellman-Ford algorithms reveals the trade-offs between time efficiency and accuracy.
- While Dijkstra's algorithm generally performs faster, especially for smaller graphs and lower relaxation limits, it may not always guarantee the most accurate results in scenarios involving negative edge weights or relaxation constraints.
- Bellman-Ford algorithm, on the other hand, provides more accurate results but at the cost of longer execution times, particularly for larger graphs and higher relaxation limits.

1.3.3 Experiment Setup for Space Complexity

The following are the details of the randomly generated test cases used in the experiment.

Algorithm	Graph Size	Density	Number of Trials	Relaxation Limit
Dijkstra	10	0.5	10	5
Bellman-Ford	10	0.5	10	5
Dijkstra	10	0.9	10	8
Bellman-Ford	10	0.9	10	8
Dijkstra	20	0.5	10	10
Bellman-Ford	20	0.5	10	10
Dijkstra	20	0.9	10	15
Bellman-Ford	20	0.9	10	15
Dijkstra	20	0.8	10	10
Bellman-Ford	20	0.8	10	10
Dijkstra	50	0.5	10	20
Bellman-Ford	50	0.5	10	20
Dijkstra	50	0.9	10	35
Bellman-Ford	50	0.9	10	35

The space taken by Dijkstra and Bellman-Ford algorithm on each of the test cases are represented in a bar graph as shown below:

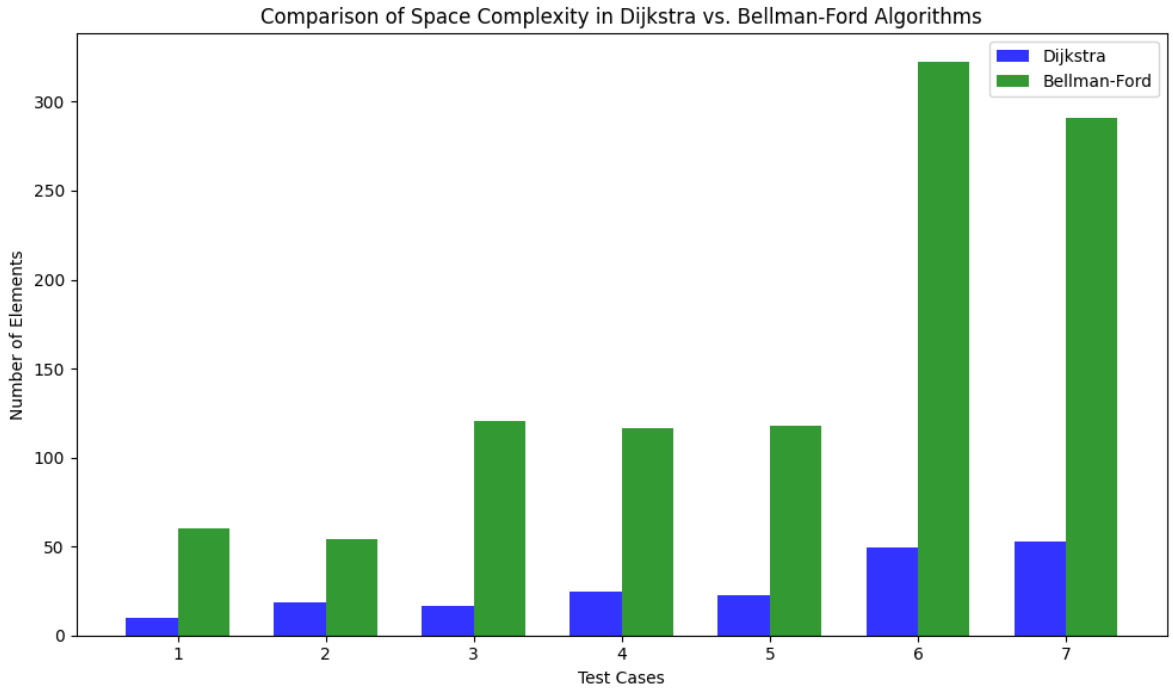

Figure 2: Dijkstra vs Bellman-Ford Algorithm (1.3.3)

1.3.4 Observation

- For each algorithm, we tested different graph sizes, densities, relaxation limits, and numbers of trials to comprehensively assess their space requirements.
- Across all test cases, Dijkstra's algorithm consistently exhibited lower space complexity compared to Bellman-Ford algorithm.
- In general, Dijkstra's algorithm showed more stable space complexity across different scenarios, with relatively smaller fluctuations in space usage compared to Bellman-Ford algorithm.
- However, as the size of the graph and the relaxation limit increased, both algorithms demonstrated an upward trend in space complexity.
- Notably, for larger graph sizes and higher relaxation limits, Bellman-Ford algorithm consumed significantly more memory compared to Dijkstra's algorithm.
- The space complexity of both algorithms appeared to increase exponentially with graph size and relaxation limit, indicating a non-linear relationship between these factors and memory usage.

1.3.5 Conclusion

Through our comprehensive experimentation, we analyzed the performance of Dijkstra's and Bellman-Ford algorithms across various scenarios, considering factors such as graph size, density, relaxation limit, and number of trials.

- **Time Complexity:**
  - Dijkstra's algorithm generally outperformed Bellman-Ford algorithm in terms of time complexity, exhibiting a time complexity of O(V^2) with a binary heap implementation, where V is the number of vertices and E is the number of edges.

    - Bellman-Ford algorithm, while versatile in handling negative edge weights and cycles, demonstrated a time complexity of O(VE), making it less efficient for large graphs and high relaxation limits.
- **Space Complexity**
    - Dijkstra's algorithm consistently demonstrated lower memory usage compared to Bellman-Ford algorithm across all test cases. Dijkstra's algorithm typically exhibits a space complexity of O(V^2) with a binary heap implementation and O(V log V) with a Fibonacci heap implementation.
    - Bellman-Ford algorithm, however, requires more memory due to its relaxed constraints, with a space complexity of O(V) for the vertex set and O(E) for the edge set.
- **Accuracy**
    - Both algorithms provided reliable results, with Dijkstra's algorithm guaranteeing accurate shortest paths in scenarios without negative edge weights. Bellman-Ford algorithm offered accurate results even in the presence of negative edge weights and negative cycles, albeit at the cost of longer execution times and higher memory usage.

2 All-pair shortest path algorithm

The task at hand is to design and implement an all-pair shortest path algorithm capable of handling both positive and negative edge weights. This involves finding the shortest path from every vertex to every other vertex in the graph. The algorithm must compute the shortest path distance between every pair of vertices and also determine the second-to-last vertex on the shortest path.

2.1 Implementation Overview

To address this problem, we've implemented an algorithm that combines Bellman-Ford's algorithm and Dijkstra's algorithm.

- Bellman-Ford Algorithm: We start by running the Bellman-Ford algorithm from a newly added vertex with zero-weight edges to all other vertices. This step helps us handle negative edge weights. Bellman-Ford computes the shortest paths from the newly added vertex to all other vertices, providing us with an initial approximation of the shortest distances.

- Recalculation of Edge Weights: We then recalculate the edge weights using the computed distances obtained from Bellman-Ford. This step involves adjusting the weights of each edge in the graph based on the differences in distances computed by

Bellman-Ford.

- Dijkstra's Algorithm: With the adjusted edge weights, we proceed to run Dijkstra's algorithm for each vertex in the graph. Dijkstra's algorithm computes the shortest paths from a single source vertex to all other vertices in the graph.

- Storing Results: As Dijkstra's algorithm progresses, we store the shortest distances and the second-to-last vertices on the shortest paths for each pair of vertices.

- Functionality: The implemented function, all_pairs_shortest_path, encapsulates the entire process described above. It takes the weighted graph as input and returns two matrices. This matrix contains the shortest distance between every pair of vertices in the graph. This matrix stores the second-to-last vertex on the shortest path between every pair of vertices.

2.2 Algorithm Analysis

During the application of the algorithm to compute the all-pairs shortest paths, several observations were made:

- For calculations where the source and destination vertices are the same (i.e., u = v), the resulting previous vertex is consistently None. This is logical, as no previous vertex is needed when computing the shortest path to the same node.

- In cases where no path exists between certain pairs of vertices, the computed distances are set to infinity, reflecting the absence of a connecting path. Additionally, the previous vertex value for such cases is designated as -1, indicating the absence of a preceding vertex.

- In some instances, the computed distances between vertices are -1. This occurrence is attributed to the presence of negative cycles in the graph, leading to an inability to determine the shortest path due to infinite looping. Consequently, these cases result in undefined distances.

2.3 Complexity Analysis for Dense Graphs

For dense graphs, the complexities of Dijkstra's and Bellman-Ford algorithms are

significant due to the large number of edges compared to the number of vertices.

- Dijkstra's Algorithm: For dense graphs, where the number of edges approaches V^2:
  - Time complexity: $\Theta(E + V \log V)$
  - Substituting $E \approx V^2$:
    - Time complexity becomes $\Theta(V^2 + V \log V)$

- Bellman-Ford Algorithm: Similarly, for dense graphs, where the number of edges is close to V^2:
  - Time complexity: $\Theta(VE)$
  - Substituting $E \approx V^2$:
    - Time complexity becomes $\Theta(V^3)$

2.4 Conclusion

For dense graphs, the overall complexity of the combined algorithm would be dominated by the more significant of the two, which is the $\Theta(V^3)$ complexity of Bellman-Ford. However, since both algorithms need to be run for each vertex as the source in the all-pairs shortest path problem, the total complexity is further multiplied by the number of vertices (V).

In mathematical terms:
- Dijkstra's algorithm complexity: $\Theta(V^2 + V \log V)$
- Bellman-Ford algorithm complexity: $\Theta(V^3)$

The overall complexity would then be: $\Theta(V * V^3) = \Theta(V^4)$
This result signifies that for dense graphs, the overall time complexity of the combined algorithm for all-pairs shortest paths becomes $\Theta(V^4)$.

3 A* Algorithm

The A* algorithm combines elements of Dijkstra's algorithm and greedy best-first search. It evaluates nodes by combining the actual cost from the start node to the current node with a heuristic estimate of the cost from the current node to the goal. The priority queue is used to explore nodes in order of their combined cost estimates.

3.1 Part 3.1

- PriorityQueue Class: This class represents a priority queue using a list of tuples, where each tuple contains the priority and the item. The put method adds items to the queue with a given priority. The get method removes and returns the item with the highest priority (lowest cost). The empty method checks if the priority queue is empty.

- A_Star Function: This function performs the A* search algorithm. It initializes a priority queue, predecessor dictionary, and cost dictionary. The main loop continues until the priority queue is empty. For each node popped from the priority queue, it explores its neighbors, updates costs, and adds them to the priority queue if necessary. It also keeps track of predecessors to reconstruct the shortest path.

- Heuristic Function: A heuristic dictionary is provided externally, which contains heuristic estimates for each node.

3.2 Part 3.2

3.2.1 What issues with Dijkstra's algorithm is A* trying to address?

Dijkstra's algorithm faces efficiency challenges, particularly in larger graphs, where it exhaustively explores every node before determining the shortest path between two nodes. While this may not be problematic for smaller graphs, it significantly slows down computation for graphs with numerous nodes and edges. In contrast, the A* algorithm enhances Dijkstra's efficiency by incorporating a heuristic function. This heuristic guides the algorithm towards the destination node, prioritizing the exploration of more promising paths. As a result, A* achieves a faster search for the shortest path by focusing on the most relevant areas of the graph.

Furthermore, Dijkstra's algorithm is unable to handle negative edge weights due to potential sub-optimal solutions or even algorithm non-termination. On the other hand, A* can accommodate negative edge weights because it strategically avoids expanding nodes in a manner that would exceed the optimal path length, thanks to its incorporation of both heuristic and path cost considerations.

3.2.2 How would you empirically test Dijkstra's vs A*?

We can conduct an empirical performance comparison between Dijkstra's and A*

algorithms by executing them on randomly generated edge-weighted graphs of varying sizes and edge numbers. For each graph, we randomly select a start and end node and apply both algorithms. We measure and record their runtimes and the number of nodes expanded during the search process. This procedure is repeated for multiple graphs to gather sufficient data. Subsequently, we utilize these metrics to create a graph and analyze any correlations.

Additionally, we can replicate this experiment using different heuristic functions to determine which one yields the best performance.

3.2.3 Comparing Dijkstra's to A* algorithm with arbitrary heuristic function.

When employing an arbitrary heuristic function, the effectiveness of the A* algorithm can significantly diminish. If the chosen heuristic does not accurately reflect the underlying structure of the graph, A* may struggle to utilize the heuristic information efficiently. Consequently, it might fail to prevent unnecessary exploration of nodes, rendering its performance comparable to or even inferior to that of Dijkstra's algorithm.

In situations where an arbitrary heuristic is employed, Dijkstra's algorithm offers more consistent performance compared to A* since it does not rely on any heuristic information.

3.2.4 Where would you use A* instead of Dijkstra's?

- The A* algorithm finds widespread application across various domains where efficiently determining the shortest path between two points among many options is crucial.

- One prominent use case is in satellite navigation systems, where A* helps identify the shortest route between two locations on a map. Here, the heuristic function can factor in dynamic variables like distance and real-time traffic conditions.

- In robotics, A* aids in planning efficient paths for robots navigating environments. By integrating sensor data, such as ultrasound readings of obstacles, into the heuristic function, the algorithm can guide the robot along the shortest feasible path.

- Network routers leverage the A* algorithm to optimize data packet routing across complex networks. Incorporating metrics like distance and network traffic conditions into the heuristic function enables routers to efficiently select the fastest transmission route from sender to receiver.

4 Compare Shortest Path Algorithms

4.1 Introduction

In this report, we compare the performance of Dijkstra's and A* algorithms on the London Subway system. We utilize real-world data describing the subway network with approximately 300 stations and their connections. The goal is to analyze the efficiency and behavior of these algorithms in finding shortest paths between pairs of stations in various scenarios.

4.2 Experiment Setup

Data Preparation: We represented each station as a node in a graph and created edges between stations based on their connections. The weights of the edges were determined using the distance between stations calculated from their latitude and longitude coordinates. Additionally, we created a heuristic function for A* algorithm using the physical direct distance between the source and a given station.

Algorithm Execution: Both Dijkstra's and A* algorithms were executed on the generated weighted graph. We computed the shortest paths between all pairs of stations using both algorithms and measured the time taken by each algorithm for each pair of stations.

4.3 Observation

The time taken by each station to all the other stations is shown below:

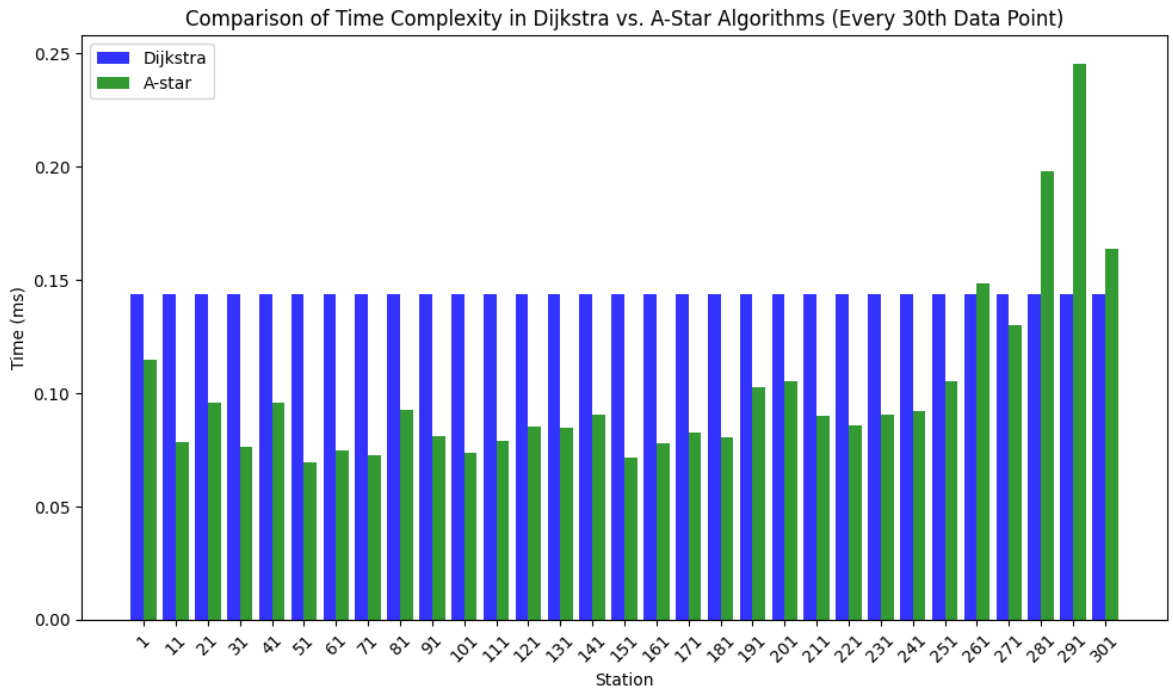

Figure 3: Dijkstra vs A Algorithm (4.3)*

- A* Outperforms Dijkstra: A* algorithm tends to outperform Dijkstra's algorithm when the heuristic function provides accurate estimates of the remaining distance to the goal. In scenarios where stations are farther apart or require significant detours, A* can efficiently explore the search space and find the shortest path faster than Dijkstra's.
- Comparable Performance: When the heuristic function is less informative or when stations are located close to each other, the performance of A* and Dijkstra's algorithms becomes comparable. In such cases, both algorithms may explore similar portions of the search space, leading to similar execution times.

Observations on Station Types:

Stations on the Same Lines: Stations on the same subway lines tend to have direct connections, resulting in shorter paths. Both algorithms perform well in finding these paths efficiently.

Stations on Adjacent Lines: Stations on adjacent lines may require some detours but still have relatively straightforward connections. A* algorithm may have a slight advantage in finding paths between such stations due to its ability to guide the search using heuristic information.

Stations Requiring Transfers: Stations requiring transfers between multiple lines pose

challenges for both algorithms, especially if the transfer points are crowded or poorly connected. In such scenarios, both algorithms may experience longer execution times as they explore alternative routes and transfer options.

Number of Lines in Shortest Paths:

Dijkstra's vs. A*: We observed that the number of lines used in the shortest paths varied between the results obtained from Dijkstra's and A* algorithms. A* algorithm tends to favor paths with fewer line transfers due to its heuristic-guided search, whereas Dijkstra's algorithm may explore multiple transfer options before finding the shortest path.

5 Organize your code as per UML diagram

5.1 Reorganize UML code

In the .py file.

5.2 Design Principles and Patterns Used

- Adapter Pattern: The A_Star Adapter acts as an adapter, making the A* algorithm (which might have a different interface or expectation of data structures) compatible with our ConcreteGraph implementation. This follows the Adapter Pattern, allowing for interoperability between incompatible interfaces.
- Interface Segregation Principle (ISP): By having a Graph interface, we ensure that any class that implements this interface will only have to implement methods that are relevant to the operation of graphs, adhering to the ISP principle of SOLID.
- Open/Closed Principle: Our design allows for graphs to be extended and for new types of graphs to be introduced without modifying existing code, adhering to the open/closed principle.

5.3 Addressing Limitations In UML

The current UML restricts nodes to integers. To overcome this, we'll introduce an abstract class hierarchy for nodes.

- Improving System Flexibility:

- Abstract Node Class: Introduce a Node class to establish a standardized structure for all graph nodes. This class includes a method to retrieve node identifiers, catering to both integers and strings. Specific node types, like StringNode and potentially DataNode, inherit from this abstract class, ensuring consistency across node representations while allowing specialization for different data types.
- Generic Graph Class: Enhance the Graph class to accept a generic type parameter, representing node types. This adjustment offers flexibility in specifying node classes (e.g., StringNode or custom classes) based on specific requirements. Methods such as add_node and get_nodes are updated to support the generic node type, ensuring uniformity in type handling throughout the code.

- Advantages:
  - Enhanced Flexibility: The revised design enables the graph to accommodate nodes of various data types, catering to diverse application scenarios. Users can select the appropriate node class based on their specific needs, enhancing system adaptability.

  - Improved Maintainability: Utilizing generics and inheritance fosters cleaner and more maintainable code. Incorporating new node types in the future becomes simpler without disrupting existing functionalities.

5.4 More Implementations

- Directed vs. Undirected Graphs: Our current Graph could be a directed graph. We could also implement an Undirected Graph class that either extends Graph or implements the Graph interface directly.
- Weighted Graphs: Another implementation could be a WeightedGraph, where edges have weights. This would require modifying the add_edge method to accept an additional weight parameter.
- Dynamic Graphs: Graphs that change over time, adding or removing nodes and edges. This would require methods for dynamically updating the graph structure.

Appendix

- MinHeap Class:
    - Purpose: Implements a min-heap data structure used in Dijkstra's algorithm for priority queue operations.
    - Methods:
        - __init__(self, data): Initializes the MinHeap with data and builds the heap.
        - find_left_index(self, index): Returns the index of the left child of a node.
        - find_right_index(self, index): Returns the index of the right child of a node.
        - find_parent_index(self, index): Returns the index of the parent of a node.
        - sink_down(self, index): Performs the sink-down operation to maintain heap property.
        - build_heap(self): Builds the heap from the input data.
        - insert(self, node): Inserts a node into the heap.
        - insert_nodes(self, node_list): Inserts a list of nodes into the heap.
        - swim_up(self, index): Performs the swim-up operation to maintain heap property.
        - get_min(self): Returns the minimum node from the heap.
        - extract_min(self): Removes and returns the minimum node from the heap.
        - decrease_key(self, value, new_key): Decreases the key of a node in the heap.
        - get_element_from_value(self, value): Returns the element corresponding to a given value.
        - is_empty(self): Returns True if the heap is empty, False otherwise.
        - __str__(self): Returns a string representation of the MinHeap.
- Item Class:
    - Purpose: Represents an item with a key-value pair.
    - Methods:
        - __init__(self, value, key): Initializes the Item with a value and key.
        - __str__(self): Returns a string representation of the Item.

- WeightedGraph Class
    - Purpose: Represents a weighted graph to model the London Subway system.
    - Methods:
        - __init__(self, nodes): Initializes the WeightedGraph with a given number of nodes.
        - add_node(self, node): Adds a node to the graph.
        - add_edge(self, node1, node2, weight): Adds a weighted edge between two nodes.
        - get_weights(self, node1, node2): Returns the weight of the edge between two nodes.
        - are_connected(self, node1, node2): Checks if two nodes are connected.
        - get_neighbors(self, node): Returns the neighbors of a node.
        - get_number_of_nodes(self): Returns the number of nodes in the graph.
        - get_nodes(self): Returns a list of nodes in the graph.
        - generate_random_directed_graph(self, density): Generates a random directed graph with given density.

- Dijkstra's Algorithm Implementation:
    - Purpose: Implements Dijkstra's algorithm for finding shortest paths in the weighted graph.
    - Methods:
        - dijkstra(graph, source, k): Executes Dijkstra's algorithm on the given graph with a specified source node and relaxation parameter k.

- A-Star Algorithm Implementation:
    - Purpose: Implements A* algorithm for finding shortest paths in the weighted graph.
    - Methods:
        - PriorityQueue: Implements a priority queue for Dijkstra's algorithm.
        - A_Star(graph, source, destination, heuristic): Executes A* algorithm on the given graph with specified source and destination nodes and a heuristic function.

Additional Libraries Used:

- math: Provides mathematical functions and constants.
- csv: Handles CSV file operations.
- time, timeit: Provides timing functionalities for performance evaluation.
- random: Generates random numbers for graph generation

Contributions:

Part 1: Sana Muhammad Ashraf

Part 2: Sriya Dhanvi Mokhasunavisu

Part 3: Sriya Dhanvi Mokhasunavisu

Part 4: Sana Muhammad Ashraf

Part 5.1 UML organizing: Prakhar Saxena

Part 5.2 – 5.4 (Theory): Sriya Dhanvi Mokhasunavisu

Group Report: Sana Muhammad Ashraf and Sriya Dhanvi Mokhasunavisu