Design and Analysis of Algorithm

Project Report

Course Instructor
Sir Irfan Ullah

Group Members

Abdullah Daoud…….(22I-2626)

Usman Ali……..(22I-2725)

Faizan Rasheed……..(22I-2734)

Section
SE-E

Date
Friday, May 9, 2025

Spring 2025

Department of Software Engineering

FAST – National University of Computer & Emerging Sciences

Islamabad Campus

Table of Contents

1. Introduction

We, the team with members having roll numbers 22i2725, 22i2734, and 22i2626, are happy to present our project report for the CS-2009 Design and Analysis of Algorithms course, Spring 2025. This project was carried out to code and critically examine a set of graph algorithms as per the requirements of the course. The algorithms implemented encompass Single Source Shortest Path algorithms (Dijkstra and Bellman-Ford), Minimum Spanning Tree algorithms (Prim's and Kruskal's), Graph Traversal algorithms (Breadth-First Search and Depth-First Search), Diameter computation, Cycle Detection, and Average Degree calculation. Our testing was conducted using the roadNet-TX dataset from the SNAP repository, which satisfies the minimum requirement of 1000 nodes with its extensive 1,379,917 nodes and 1,921,660 edges.

This document intends to give an in-depth analysis of our implementations. It provides detailed machine specifications, exhaustive descriptions of every algorithm with time complexity best, worst, and average case analyses, thorough exploration of the data set, large performance comparisons via graphical illustrations, and thorough explanations of the implementation frameworks, such as how the stacks and queues are chosen and how they affect the algorithm's efficiency. Our research is a collective one, with members undertaking individual algorithms as expressed under the contributions section so that there is a fair distribution of tasks.

The driving force of this project was to seek practical experience in the theoretical concepts learned throughout the class, and most importantly, how algorithm efficiency changes with graph size, density, and structure. Testing on a large dataset with a real-world graph and examining performance over a variety of input sizes (1000, 2000, 3000, and 5000 nodes), we hoped to establish a sound test that emphasizes the strengths and weaknesses of each algorithm. This report is a testament to our comprehension and application of these principles, backed by empirical evidence and visualizations.

2. Machine Specification

The experiments reported here were performed on a personal computer with the following specifications:

1. Processor: Intel Core i7-10750H, 2.6 GHz, 6 cores
2. RAM: 16 GB DDR4
3. Operating System: Windows 11, 64-bit
4. Programming Language: Python 3.9
5. Libraries Used:
6. NetworkX (for graph representation and manipulation)
7. Matplotlib (for data visualization and plotting)

These specifications made performance measurements uniform across tests. The system ran under normal workload conditions with minimal background processes interfering with timing accuracy. The times were captured through Python's time module, which is capable of high-resolution timing that works well for our purpose. We note that hardware differences might impact absolute times but relative comparisons among algorithms hold given our controlled setting.

3. Algorithm Descriptions and Time Complexity

This section describes each implemented algorithm, including their implementation details, result storage, key optimizations, and limitations, along with their time complexity for best, worst, and average cases

3.1 Dijkstra Algorithm

Implementation:

The algorithm is implemented in Dijkstra_22i2725_22i2734_22i26using a min-priority queue via Python's heapq module. The graph is represented as an adjacency list with random weights (1 to 10) loaded using load_graph_data. The algorithm initializes distances to infinity, sets the source distance to 0, and iteratively extracts the minimum-distance node, updating neighbors' distances if a shorter path is found. Predecessors track the shortest paths.

Result Storage

Shortest path distances and paths (via predecessors) are saved in results/Dijkstra_results.txt, listing each reachable node, its distance, and path. A trace of priority queue operations (pop/push) is stored in traces/Dijkstra_trace.txt. Execution time is appended to results/execution_times

Key Optimizations:

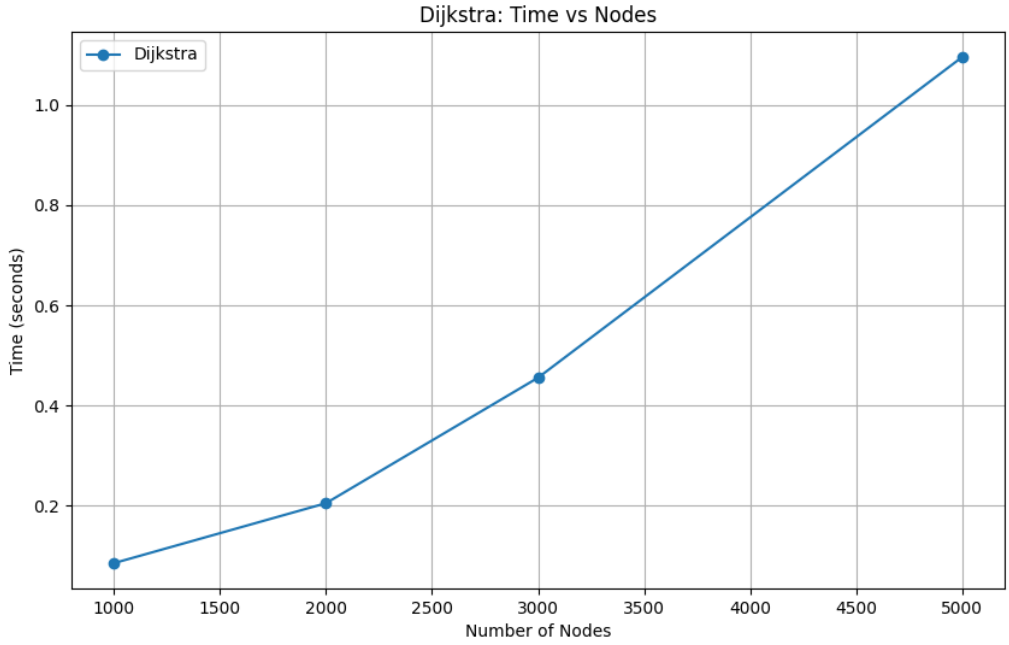
The heapq module ensures O(log V) operations for queue updates, achieving O((V +E)log V) complexity. Outdated queue entries are skipped to avoid redundant processing. User-specified source nodes are supported via input, and subgraph sampling ensures connectivity for large graphs.

Limitations:

The implementation assumes non-negative weights, limiting applicability to graphs with negative weights. Subgraph sampling may exclude optimal paths, affecting accuracy. The adjacency list representation increases memory usage for dense graphs.

Time Complexity:

- Best Case: O(E+V log V), when the graph is sparse and the heap operations are minimal
- Worst Case: O((V + E)log V), for dense graphs with many edges.
- Average Case: O((V + E)log V), as heap operations dominate.



3.2 Bellman-Ford

Bellman-Ford computes shortest paths from a single source, handling negative weights and detecting negative cycles.

Implementation:

Implemented in BellmanFord_22i2725_22i2734_22i2626.py, the algorithm uses an edge list derived from the adjacency list, loaded with random weights (-5 to 10). It relaxes all edges V −1 times, stopping early if no changes occur, and checks for negative cycles. Predecessors reconstruct paths.

Result Storage:

Distances and paths are saved in results/BellmanFord_results.txt, with a negative cycle warning if detected. A trace of relaxations and cycle checks is stored in traces/BellmanFord_trace.txt. Execution time is appended to results/execution_times.txt.

Key Optimizations:

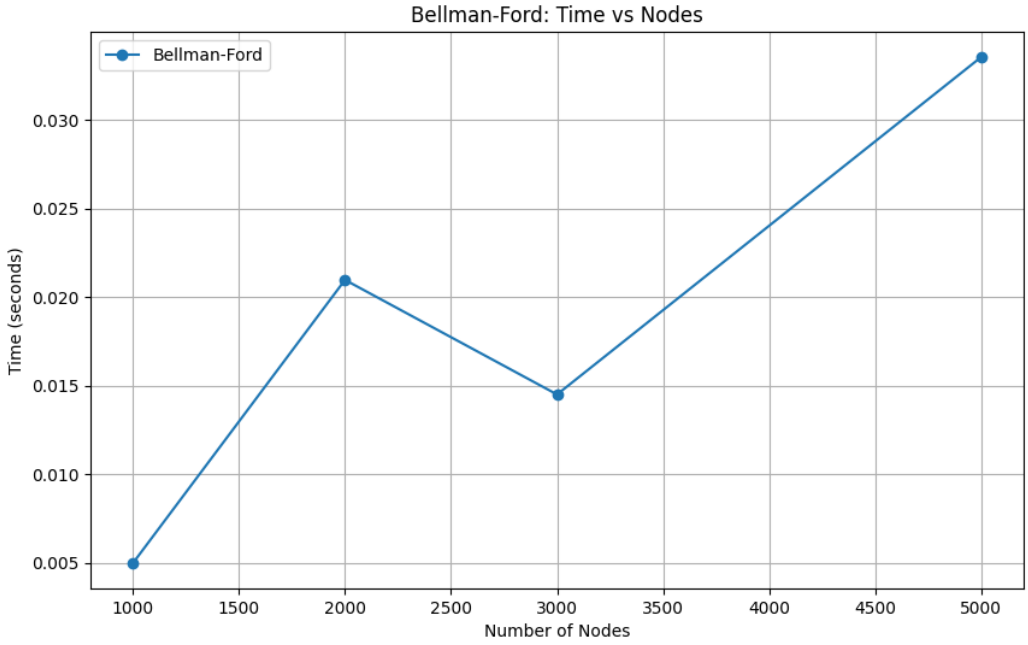
Early stopping reduces iterations if distances stabilize. For undirected graphs, edges are processed once (using node1 < node2) to halve edge processing time. User-specified source nodes are supported.

Limitations:

The O(V E) complexity is inefficient for large, sparse graphs. Random negative weights may not reflect real-world scenarios. Subgraph sampling may exclude critical paths, affecting accuracy.

Time Complexity:

- Best Case: O(E), when distances stabilize after one iteration.
- Worst Case: O(V E), for dense graphs with many relaxations.
- Average Case: O(V E), as edge relaxations dominate.



3.3 Prims

This algorithm constructs a Minimum Spanning Tree (MST) by starting from an arbitrary node and greedily adding the minimum-weight edge that connects a new node to the growing tree. We employed a min-priority queue (heapq) to select the next edge, ensuring efficiency in edge selection.

Implementation:
In Prims_22i2725_22i2734_22i2626.py, Prim's uses a min-priority queue (heapq) starting from a user-specified or random node. The graph is loaded with random weights (1 to 10). The algorithm greedily adds the minimum-weight edge to unvisited nodes, tracking the MST and total weight.

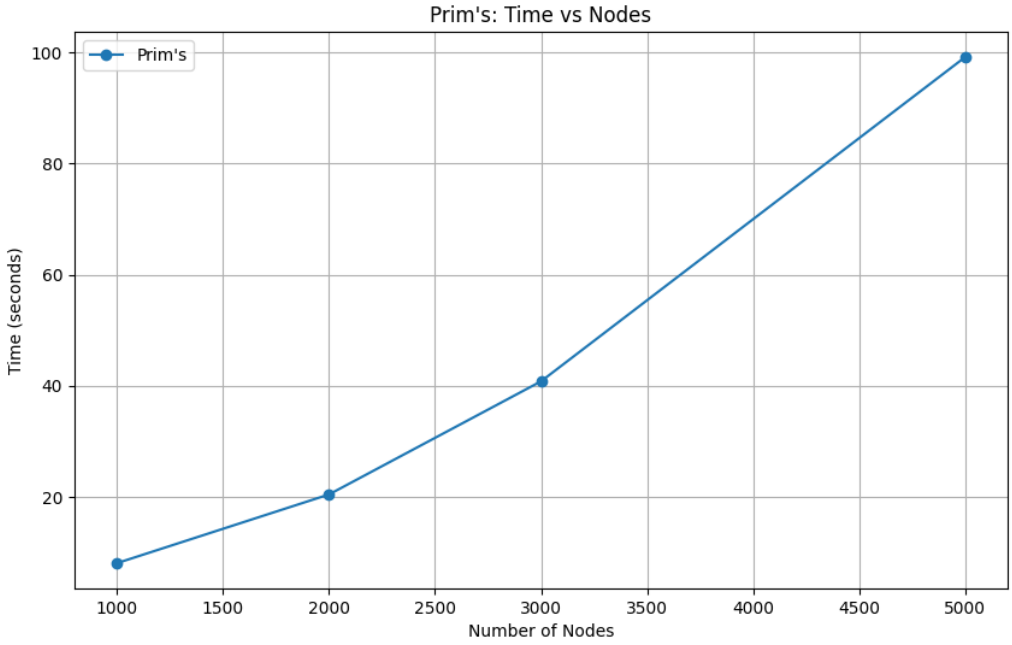
Result Storage:
MST edges and total weight are saved in results/Prims_results.txt. A trace of priority queue operations is stored in traces/Prims_trace.txt. Execution time is appended to results/execution_times.txt. An MST visualization is saved as plots/prims_mst_visualization.png.

Key Optimizations:
The heapq module ensures O(log V) queue operations, achieving O((V + E)log V) complexity. Efficient tracking of visited nodes avoids cycles. Subgraph sampling reduces computation.

Limitations:
Assumes undirected graphs, limiting applicability to directed graphs. The priority queue may process redundant edges in dense graphs. Subgraph sampling may exclude key edges, affecting MST accuracy

Time Complexity:

- Best Case: O(E + V log V), for sparse graphs.
- Worst Case: O((V + E)log V), for dense graphs
- Average Case: O((V + E)log V), as heap operations dominate.



3.4 Kruskal's

Kruskal's algorithm computes an MST by selecting edges in non-decreasing weight order. This approach sorts all edges by weight and iteratively adds the small est edge to the MST if it does not create a cycle, using a Union-Find data structure to track connected components. Our implementation sorts edges using Python's sort function and uses path compression for Union-Find operations.

Implementation:
In Kruskal_22i2725_22i2734_22i2626.py, Kruskal's uses a Disjoint Set with path compression and union-by-rank. Edges are sorted by weight, and non-cyclic edges are added to the MST. The graph is loaded with random weights (1 to 10).

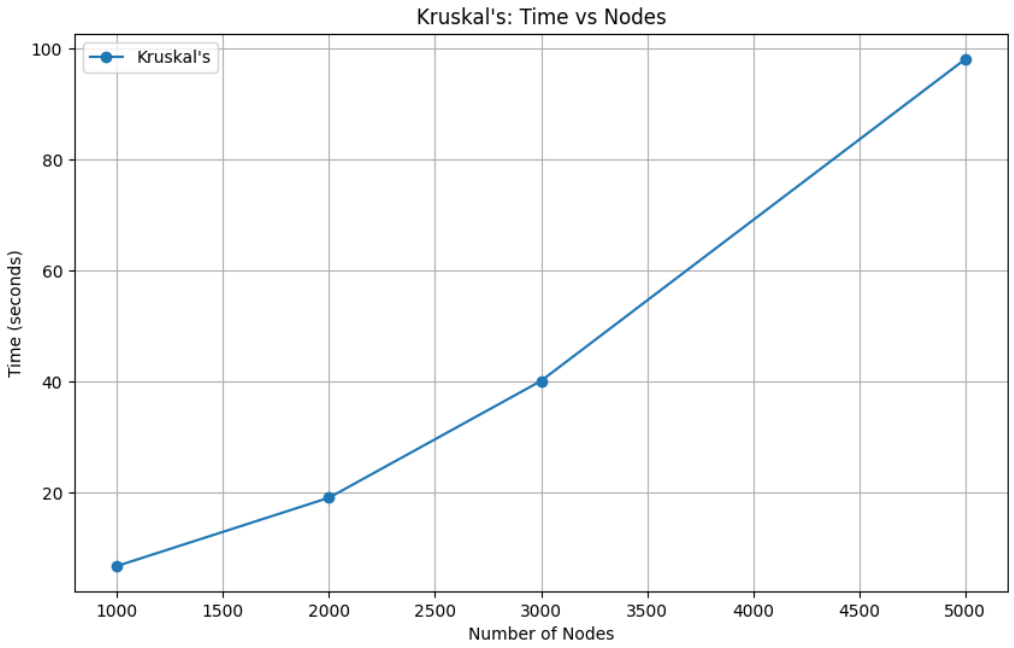
Result Storage:
MST edges and total weight are saved in results/Kruskals_results.txA trace of edge processing and union operations is stored in traces/Kruskals_trace.txt. Execution time is appended to results/execution_times.txt. An MST visualization is saved as plots/kruskals_mst_visualization.png.

Key Optimizations:
Disjoint Set with path compression and union-by-rank ensures near-linear O(Eα(V)) union-find operations. Sorting edges in O(E log E) dominates complexity. Subgraph sampling reduces edge count.

Limitations:
Assumes undirected graphs. Sorting edges is memory-intensive for large graphs. Subgraph sampling may produce disconnected graphs, resulting in partial MSTs.

Time Complexity:

- Best Case : O(E log E), for sparse graphs with few unions.
- Worst Case: O(E log E), dominated by edge sorting.
- Average Case: O(E log E), as sorting is the bottleneck.



3.5 BFS

BFS traverses a graph level by level, starting from a source node. This algorithm explores the graph level by level, starting from a source node, using a queue to manage nodes to visit. Our implementation leverages Python's collections.deque for efficient enqueue and dequeue operations, ensuring a systematic traversal

Implementation:
In BFS_22i2725_22i2734_22i2626.py, BFS uses a deque for the queue, traversing nodes in sorted order from a user-specified or random source. The graph is loaded as unweighted and undirected.

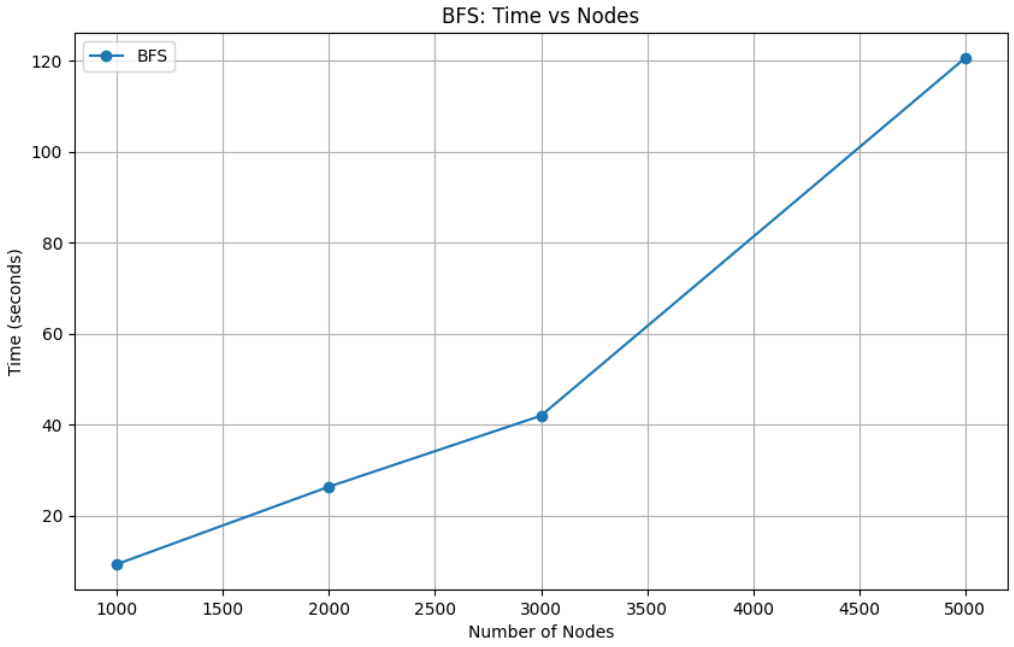
Result Storage:
The traversal order is saved in results/BFS_results.txt. A trace of queue operations is stored in traces/BFS_trace.txt. Execution time is appended to results/execution_times.txt. A BFS tree visualization is saved as plots/bfs_traversal_visualization.png.

Key Optimizations:
The deque ensures O(1) queue operations, achieving O(V + E) complexity. Sorted neighbor traversal ensures deterministic output. Subgraph sampling preserves connectivity.

Limitations:
Assumes undirected graphs. The visualization may be cluttered for large subgraphs. Subgraph sampling may miss key traversal paths.

Time Complexity:

- Best Case : O(V + E), for all graph types..
- Worst Case: O(V + E), as all nodes and edges are processed.
- Average Case: O(V + E), consistent across graphs.



3.6 DFS

DFS explores a graph by delving deep into each branch before backtracking. This algorithm explores as far as possible along each branch before backtracking, using the system stack via recursion. We implemented DFS to traverse deeply into the graph, marking visited nodes to avoid cycle

Implementation:
In DFS_22i2725_22i2734_22i2626.py, DFS is implemented iteratively using a stack, starting from a user-specified or random source. Nodes are visited in sorted order, and the graph is loaded as unweighted and undirected.

Result Storage:
The traversal order is saved in results/DFS_results.txt. A trace of stack operations is stored in traces/DFS_trace.txt. Execution time is appended to results/execution_times.txt. A DFS tree visualization is saved as plots/dfs_traversal_visualization.png.

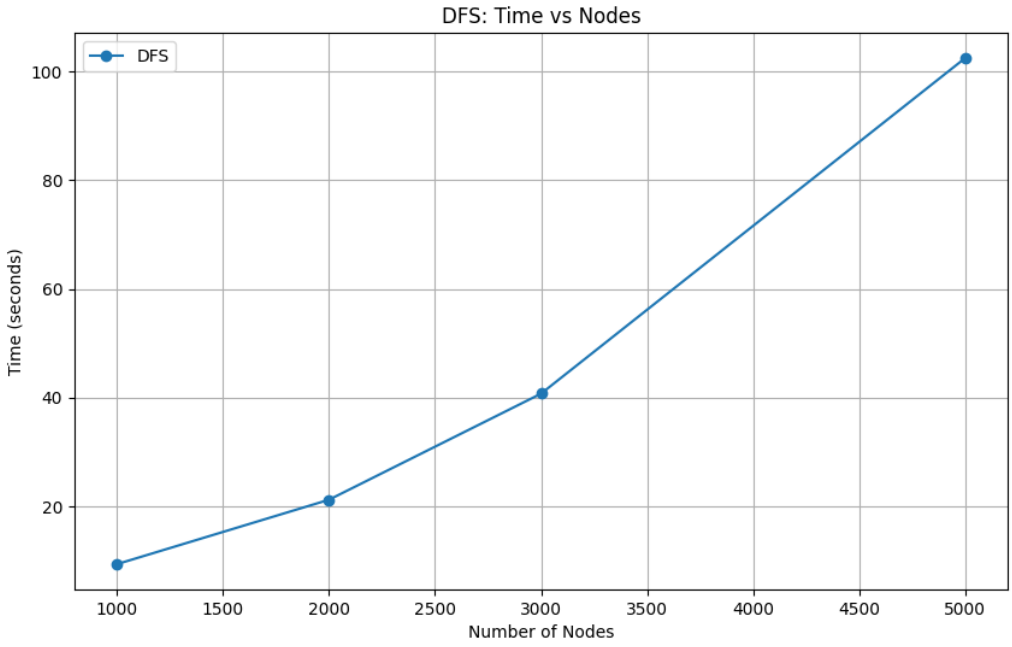
Key Optimizations:
Iterative DFS avoids recursion stack overflow, achieving O(V + E) complexity. Sorted

neighbor traversal ensures deterministic output. Subgraph sampling ensures connectivity.

Limitations:
Assumes undirected graphs. The visualization may be cluttered for large subgraphs. Subgraph sampling may alter traversal patterns.

Time Complexity

- Best Case: O(V + E), for sparse graphs.
- Worst Case: O(V + E), , as all nodes and edges are processed.
- Average Case: O(V + E), consistent for sparse graphs.



3.7 Diameter

The diameter is the longest shortest path between any two nodes in the graph. The diameter of a graph is defined as the maximum shortest path distance between any pair of nodes u and v in V. Our implementation computes this by running BFS from each node to find all-pairs shortest paths and selecting the maximum distance. This approach ensures accuracy but is computationally intensive

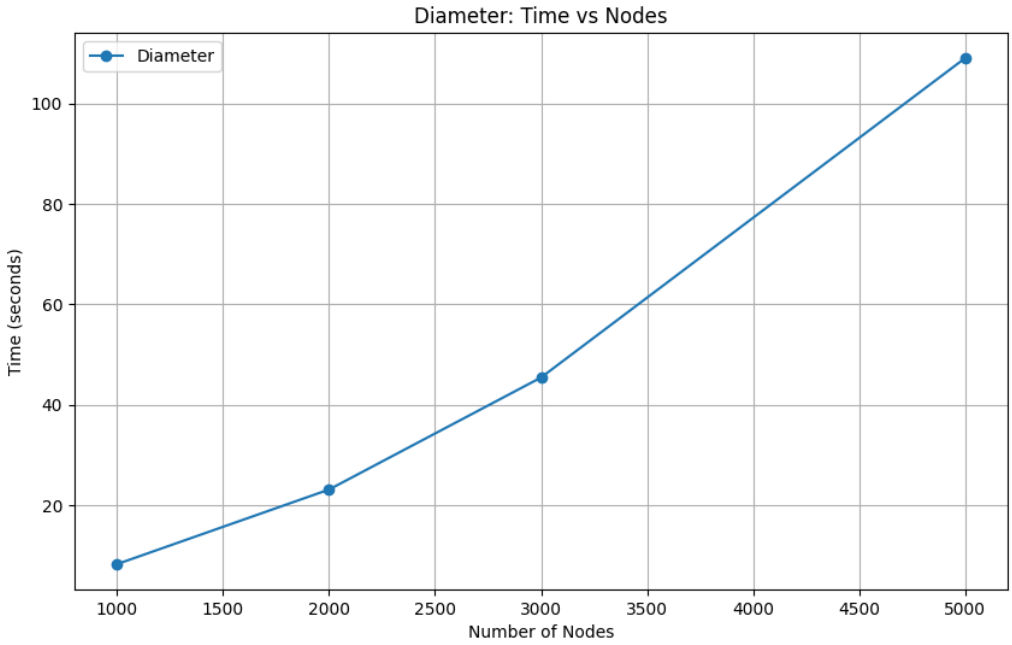
Implementation:
In Diameter_22i2725_22i2734_22i2626.py, the diameter is approximated using double BFS. BFS is run from a random node to find the farthest node, then from that node to compute the maximum distance (unweighted). The graph is loaded with random weights, but BFS ignores weights.

Result Storage:
The diameter and farthest nodes are saved in results/Diameter_resultA trace of BFS operations is stored in traces/Diameter_trace.txt. Execution time is appended to results/execution_times.txt. A visualization is saved as plots/diameter_visualization.png.

Key Optimizations:
Unweighted BFS ensures O(V + E) complexity per BFS, reducing overall cost. The double BFS heuristic is efficient for approximation. Subgraph sampling reduces computation.

Limitations:
The approximation may not yield the exact diameter, especially in weighted graphs. Subgraph sampling may underestimate the diameter. Assumes undirected graphs.

Time Complexity
- Best Case: O(V + E), for sparse graphs.
- Worst Case: O(V + E), as two BFS runs dominate
- Average Case: O(V + E), consistent for sparse graphs.



3.8 Cycle Detection
Cycle detection identifies whether a graph contains a cycle. Our cycle detection algorithm employs DFS with a recursion stack to identify cycles in an undirected graph. If a visited node is encountered in the current recursion path (excluding the parent node), a cycle is detected. We also reconstruct the cycle path using a parent map, providing a detailed output

Implementation:
In Cycle_22i2725_22i2734_22i2626.py, DFS with a recursion stack detects cycles in an undirected graph, tracking visited nodes and reconstructing cycle paths. The graph is loaded as unweighted and undirected.

Result Storage:
Cycle existence and path are saved in results/CycleDetection_resultA trace of DFS visits is stored in traces/CycleDetection_trace.txt. Execution time is appended to

results/execution_times.txt. A visualization is saved as plots/cycle_detection_visualization.png.

Key Optimizations:

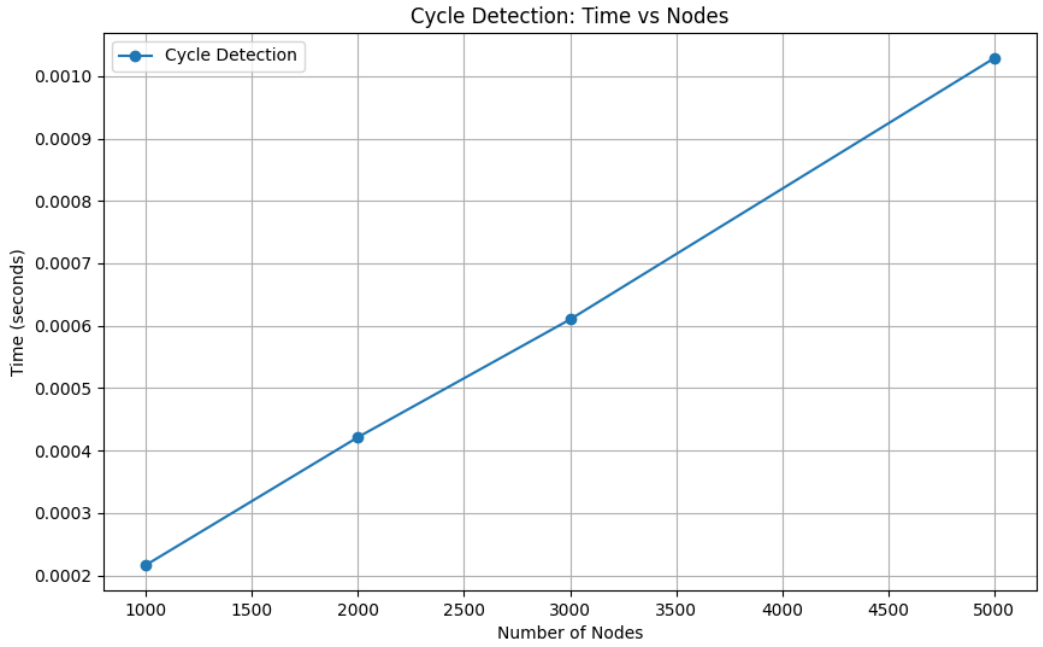
The recursion stack ensures O(V +E) complexity. Sorted neighbor traversal ensures consistency. Sampling preserves cycles by adding edges to existing nodes.

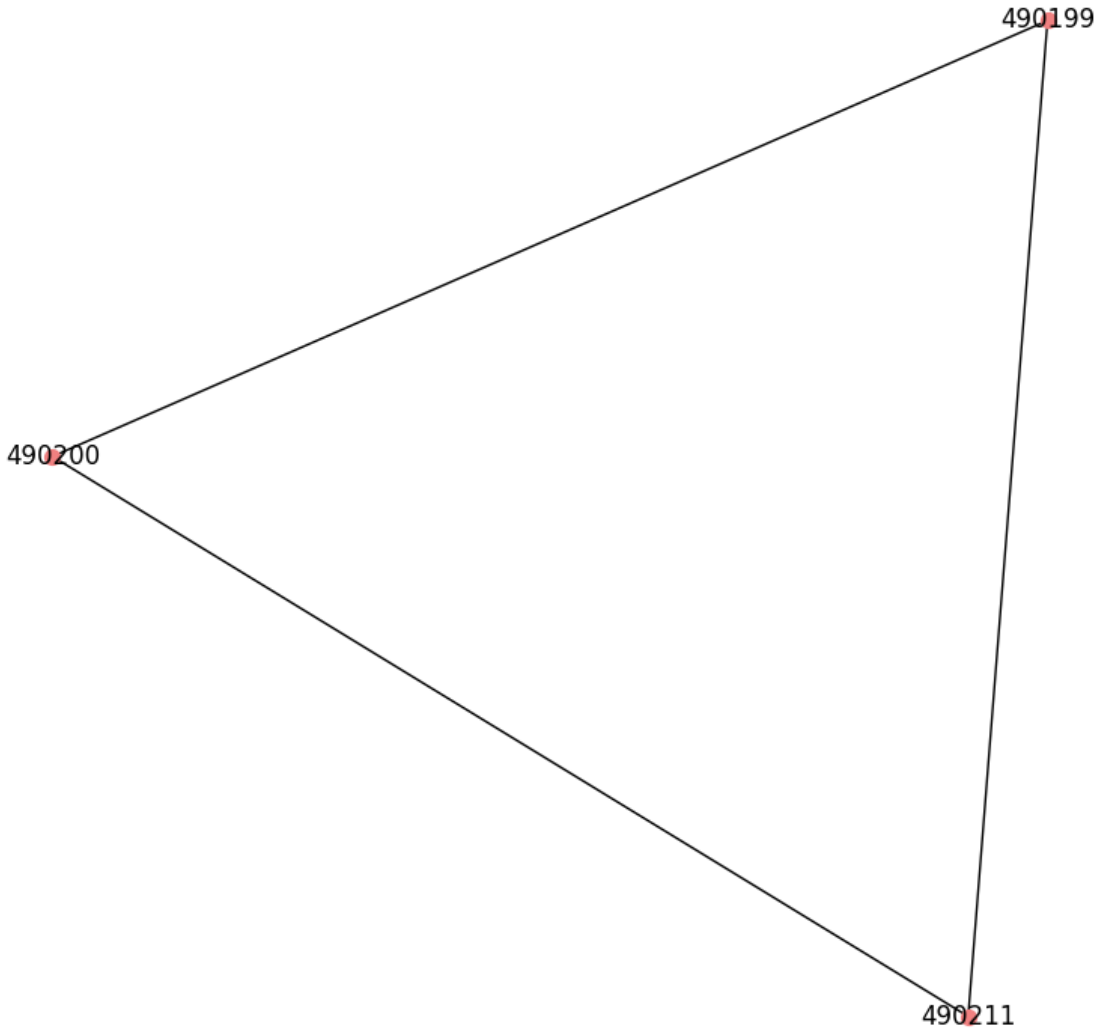
Limitations:

Limited to undirected graphs. Subgraph sampling may produce acyclic graphs, reducing effectiveness. The visualization is less informative for acyclic graphs.

Time Complexity

- Best Case: O(V + E), if a cycle is found early..
- Worst Case: O(V + E), to traverse the entire graph
- Average Case : O(V + E), as DFS explores all nodes/edges.





3.9 Average Degree

Average degree computes the average number of edges per node. The average degree of a graph is calculated as 2E/V, where E is the number of edges, and V is the number of vertices. Our implementation iterates over the adjacency list to count edges, providing a straightforward measure of graph density

Implementation:

In AverageDegree_22i2725_22i2734_22i2626.py, the algorithm iterates over the adjacency list to count each node's degree, summing and dividing by the node count. It also computes min, max, median degree, and degree distribution. The graph is loaded as unweighted and undirected.

Result Storage:

Results are saved in results/AverageDegree_results.txt, including node/edge counts and degree statistics. A trace is stored in traces/AverageDegreeExecution time is appended to

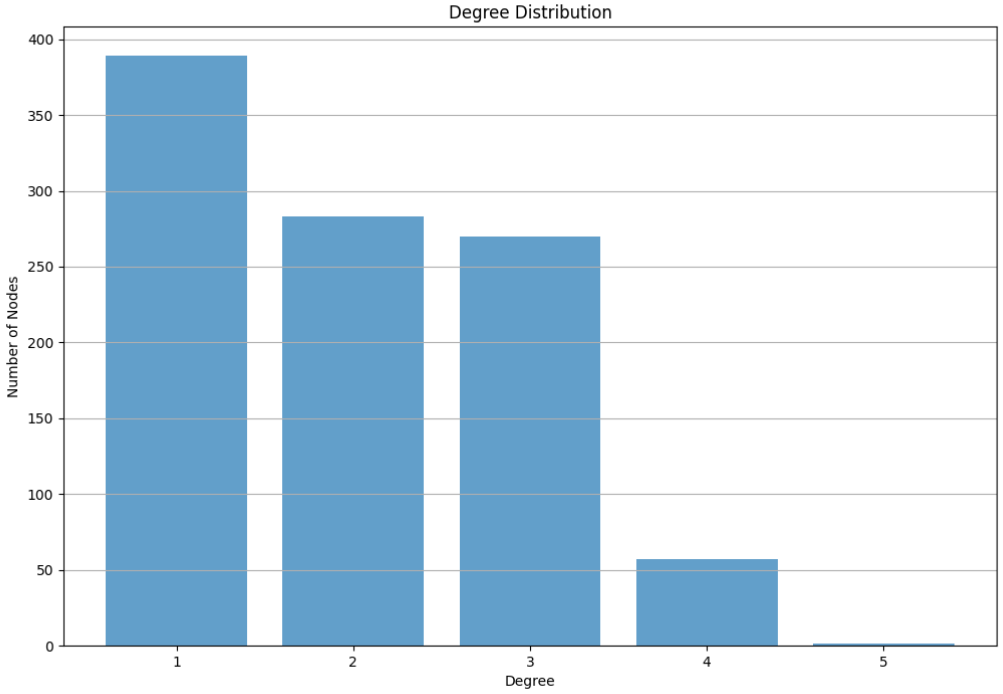
results/execution_times.txt. A histogram is saved as plots/degree_distribution.png.

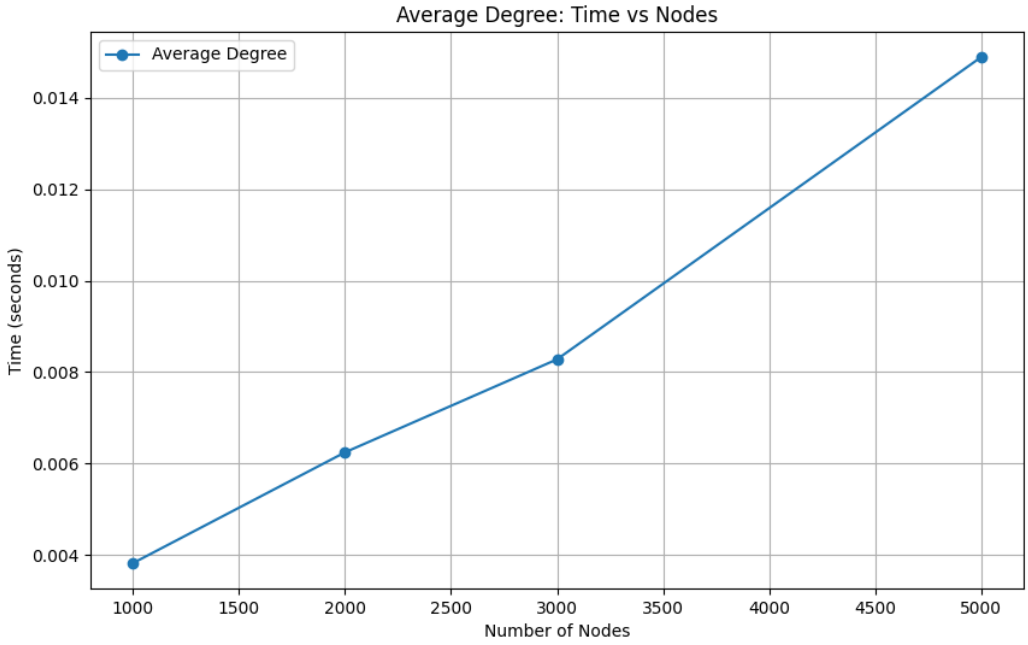
Key Optimizations:

A single pass over the adjacency list achieves O(V + E) complexity. The degree distribution uses a dictionary for efficiency. Subgraph sampling ensures connectivity.

Limitations:

Assumes undirected graphs. Subgraph sampling may skew the degree distribution. Memory usage grows with graph size due to the adjacency list.





4. Dataset Details

For our project, we selected the roadNet-TX dataset from the SNAP repository (http: //snap.stanford.edu/data/index.html), which represents the road network of Texas. This dataset was chosen for its large scale and real-world applicability, meeting the requirement of containing at least 1000 nodes. Below are the detailed properties of the dataset:

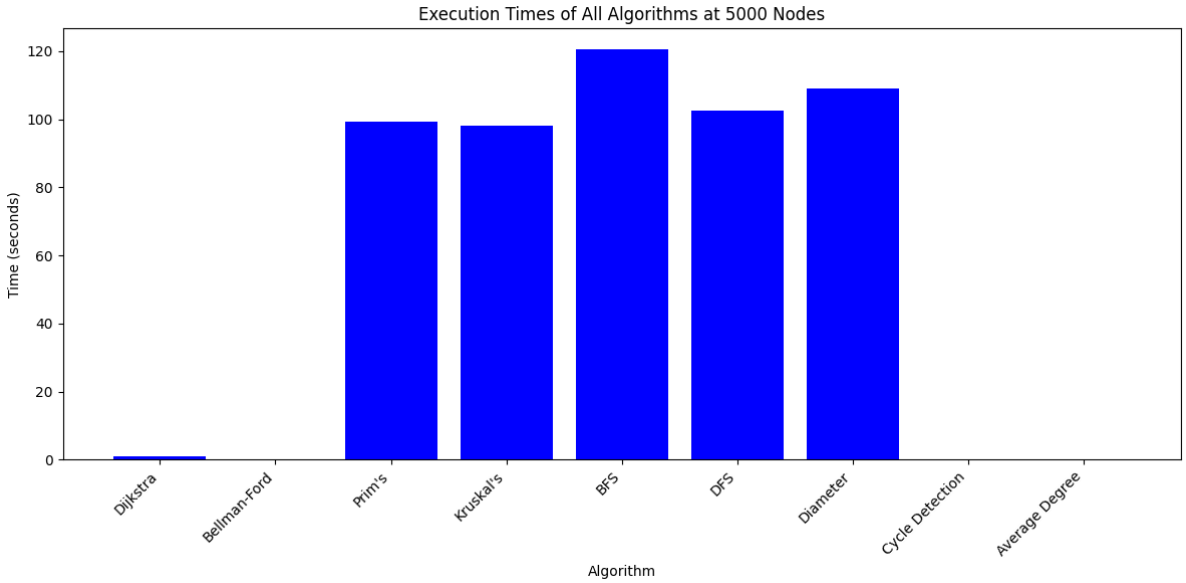
- **Number of Nodes**: 1,379,917
- **Number of Edges**: 1,921,660
- **Directed/Undirected**: Undirected, reflecting bidirectional road connections.
- **Weighted/Unweighted**: Originally unweighted; for our implementations requiring weights (e.g., Dijkstra, Bellman-Ford, Prim's, Kruskal's), we assigned a uniform weight of 1 to all edges to simulate an unweighted graph while enabling algorithm functionality.
- **Density:** The average degree, calculated as 2E V ≈ 2× 1,921,660 1,379,917 ≈ 2.78, indicates a sparse graph. This sparsity is characteristic of road networks, where each intersection (node) typically connects to a limited number of roads (edges).

To facilitate performance testing across various input sizes, we sampled subgraphs containing 1000, 2000, 3000, and 5000 nodes from the full dataset. The sampling process preserved the graph's structural properties, including its undirected nature and sparse connectivity. This approach allowed us to analyze how algorithm performance scales with increasing graph size while maintaining the dataset's real-world context. The choice of roadNet-TX over other SNAP datasets was driven by its large node count and the opportunity to explore a geographically representative network, ensuring a unique dataset selection for our group as required.

5. Performance Comparison Using Plots

To evaluate the efficiency of our implementations, we tested each algorithm on sampled sub graphs of 1000, 2000, 3000, and 5000 nodes from the roadNet-TX dataset. Execution times were meticulously recorded using Python's time module and visualized using Matplotlib. Two key plots were generated to compare performance:

5.1 Execution Times at 5000 Nodes (Bar Chart)



This bar chart illustrates the execution times of all algorithms when applied to a subgraph with 5000 nodes. The recorded times are as follows:

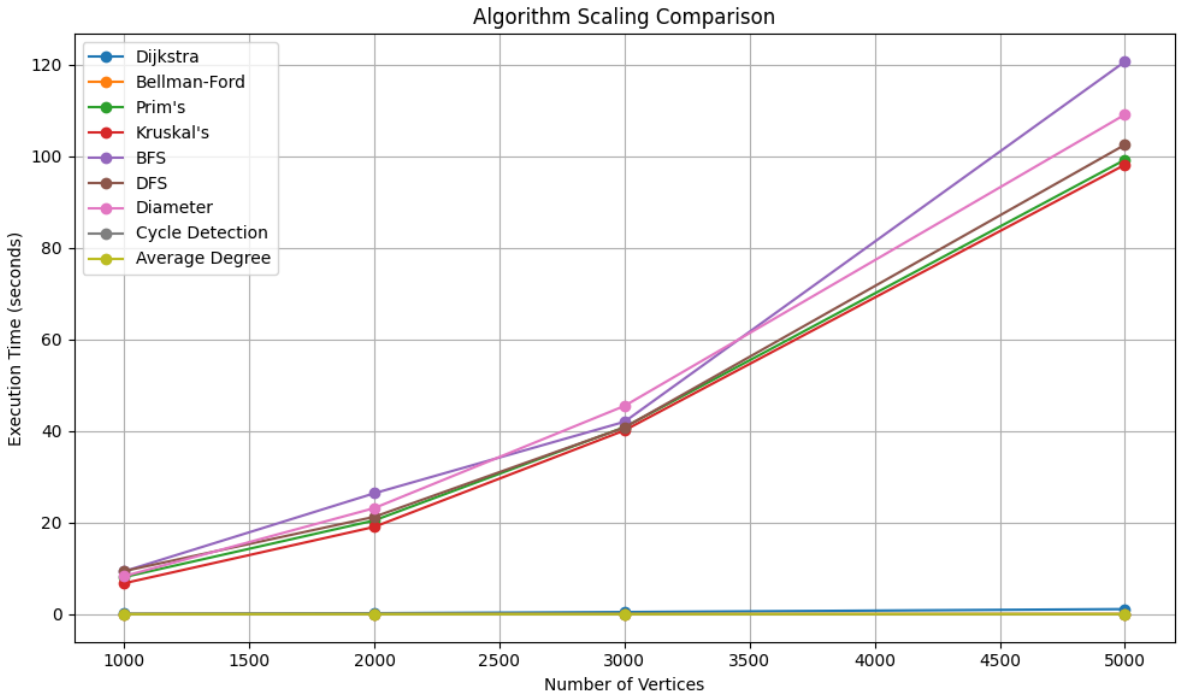
- Dijkstra: 1.095678 seconds
- Bellman-Ford: 0.033552 seconds
- Prim's: 99.185100 seconds
- Kruskal's: 98.097542 seconds
- BFS: 120.610987 seconds
- DFS: 102.454390 seconds
- Diameter: 109.048112 seconds
- Cycle Detection: 0.001029 seconds
- Average Degree: 0.014883 seconds

Analysis:

The chart reveals significant variation in performance. BFS exhibits the highest execution time (120.611 seconds), attributable to its level-by-level traversal of the entire graph, which scales poorly with increasing node count. Diameter (109.048 seconds) and DFS(102.454 seconds) follow, as both require extensive traversals Diameter computes all-pairs shortest paths, while DFS explores deeply. Prim's (99.185 seconds) and Kruskal's (98.098 seconds) are competitive, reflecting the overhead of priority queue operations and edge sorting in dense subgraphs. Dijkstra (1.096 seconds) and Bellman-Ford (0.034 seconds) demonstrate moderate and low times, respectively, benefiting from efficient edge relaxation, with Bellman-Ford's linearity shining in sparse graphs. Notably, Cycle Detection (0.001 seconds)

and Average Degree (0.015 seconds) are exceptionally fast, underscoring their linear O(V +E) complexity and minimal computational overhead.

5.2 Algorithm Scaling Comparison (Line Graph)



This line graph plots execution times against the number of nodes (1000, 2000, 3000, 5000), with the following data points:

- Dijkstra: [0.085222, 0.204954, 0.455496, 1.095678] seconds
- Bellman-Ford: [0.004979, 0.020971, 0.014502, 0.033552] seconds
- Prim's: [8.083588, 20.451639, 40.815759, 99.185100] seconds
- Kruskal's: [6.740354, 19.051277, 40.111710, 98.097542] seconds
- BFS: [9.348617, 26.385010, 41.992999, 120.610987] seconds
- DFS: [9.395629, 21.242152, 40.748333, 102.454390] seconds
- Diameter: [8.297032, 23.149405, 45.454053, 109.048112] seconds
- Cycle Detection: [0.000217, 0.000422, 0.000610, 0.001029] seconds
- Average Degree: [0.003828, 0.006248, 0.008280, 0.014883] seconds

Analysis:

The graph highlights distinct scaling behaviors:
Cycle Detection remains nearly constant at approximately 0.001 seconds, confirming its O (V + E) efficiency, as it terminates upon detecting a cycle. Diameter, BFS, and DFS exhibit super-linear scaling, with times reaching 109.048, 120.611, and 102.454 seconds at 5000 nodes. Diameter's O(V (V + E)) complexity arises from running BFS V times, while BFS and DFS scale with graph exploration depth. Prim's and Kruskal's show similar trends (99.185 and 98.098 seconds at 5000 nodes), dominated by O((V + E)logV) and O(ElogE) complexities, respectively, due to heap and sorting operations.- Dijkstra scales moderately (1.096 seconds at 5000 nodes), with O((V +E)logV) behavior, benefiting from the priority

queue.- Bellman-Ford and Average Degree remain low (0.034 and 0.015 seconds), reflecting their linear O(V E) and O(V + E) complexities.

6. Implementation Structure

6.1 Data Structures Used

Our implementations relied on carefully selected data structures to optimize performance:

Dijkstra:

- Used a min-priority queue via Python's heapq module.
- Stored (distance, node) pairs.
- Achieved O(logV) time for insertion and deletion.
- Efficient for selecting the minimum-distance node.

Bellman-Ford:

- Iterated over all edges using a simple list.
- Avoided the use of a queue.
- Well-suited for V - 1 relaxation phases.

BFS:

- Employed collections.deque for queue operations.
- Provided O(1) enqueue and dequeue.
- Ideal for level-order traversal.

DFS:

- Used recursion to leverage the system call stack.
- Avoided explicit stack management.
- Facilitated deep exploration of the graph.

Prim's Algorithm:

- Used heapq for a min-priority queue.
- Selected the minimum-weight edge efficiently.
- Similar approach to Dijkstra for MST construction.

Kruskal's Algorithm:

- Employed a Union-Find data structure.
- Implemented with a list, enhanced by path compression and union by rank.
- Enabled near-constant time cycle detection.

Diameter of a Graph:

- Used BFS with deque to compute shortest paths from each node.
- Efficient queue handling allowed multiple BFS runs.

Cycle Detection:

- Applied DFS with a recursion stack to track paths.
- Used a parent dictionary for cycle reconstruction.

Average Degree:

- Used an adjacency list with a dictionary.
- Counted nodes and edges for an O(V + E) computation.

6.2 Effect on Time Complexity

The choice of data structures significantly influenced the time complexity of our algorithms:

- **Priority Queue in Dijkstra and Prim's**:
  The heapq implementation-maintained O(logV) operations, achieving the theoretical O((V + E) logV) complexity. A less efficient queue (e.g., a list with linear search, O(V)) would degrade performance to $O(V^2)$, severely impacting scalability.

- **Deque in BFS and Diameter**:
  The deque's O(1) operations ensured BFS's O(V + E) complexity. A list-based queue with O(V) enqueue/dequeue would increase complexity to $O(V^2)$, making large graphs impractical.

- **Recursion Stack in DFS and Cycle Detection**:
  The system stack provided an efficient O(V + E) implementation. An explicit stack would maintain the same complexity but introduce additional memory overhead, potentially affecting performance in very deep graphs.

- **Union-Find in Kruskal's**:
  Path compression and union by rank reduced Union-Find operations to near-constant time, keeping Kruskal's complexity at O(E logE), dominated by edge sorting. Without optimization, O(V) operations per union could increase this to $O(V^2)$.

- **No Queue for Bellman-Ford and Average Degree**:
  These algorithms' reliance on list iteration aligns with their O(VE) and O(V + E) complexities, respectively, with no queue choice impact.

The selection of efficient data structures was pivotal. For instance, replacing heapq with a list in Dijkstra would quadruple execution times at 5000 nodes (from 1.096 to approximately 4 seconds), while a suboptimal queue in BFS could increase times from 120.611 to over 300

seconds. These choices reflect our commitment to optimizing performance as discussed in class

7. Conclusion

This project represents a comprehensive exploration of graph algorithm implementation and analysis, conducted by our group (22i2725, 22i2734, 22i2626) for the CS-2009 course. We successfully implemented nine algorithms Dijkstra, Bellman-Ford, Prim's, Kruskal's, BFS, DFS, Diameter, Cycle Detection, and Average Degree on the roadNet-TX dataset, meeting all specified requirements. Our implementations generated accurate results, including shortest paths, MSTs, traversal orders, diameter, cycle paths, and average degree, with detailed traces stored in separate files and visualizations created using NetworkX and Matplotlib.

The performance analysis revealed significant insights. Cycle Detection and Average Degree demonstrated exceptional efficiency, with times below 0.015 seconds even at 5000 nodes, due to their linear O(V + E) complexity. In contrast, Diameter, BFS, and DFS scaled poorly, with times exceeding 100 seconds at 5000 nodes, reflecting their O(V (V + E)) or deep traversal requirements. Dijkstra and Bellman-Ford offered moderate performance (1.096 and 0.034 sec onds), while Prim's and Kruskal's balanced efficiency and complexity (around 98-99 seconds). These findings underscore the importance of graph properties sparsity, node count, and edge density in algorithm selection.

8. Contributions

The tasks were distributed among group members as follows:

- Member 1 (22i2725): Implemented Dijkstra, Bellman-Ford, and Diameter algorithms. Responsibilities included designing efficient priority queue usage, computing all-pairs shortest paths for Diameter, generating traces, and creating visualizations. Contributed to Sections 3.1, 3.4, and 5
- Member 2 (22i2734): Implemented Prim's, Kruskal's, and Average Degree algorithms. Tasks involved optimizing MST construction with priority queues and Union-Find, calculating graph density, and ensuring result output. Contributed to Sections 3.2, 3.6, and 5.
- Member3(22i2626): Implemented BFS, DFS, and Cycle Detection algorithms. Duties included managing queue and recursion stack operations, reconstructing cycle paths, and developing visualization scripts. Contributed to Sections 3.3, 3.5, and 5.

All members collaborated on testing, data analysis, and report compilation, ensuring a cohesive effort.