Dataset Description

We benchmarked on the **Bitcoin Alpha trust network**

- **Nodes**: 3,783
- **Edges**: 24,186 (directed, weighted, signed)
- **Edge weights**: –10 to +10 (trust score)
- **Source**: https://snap.stanford.edu/data/soc-sign-bitcoin-alpha.html

Member 1

1. Machine Specification

Component	Specification
Model Name	HP Pavilion x360
Chip	Intel core i5 10th Generation
Memory	8 GB
Storage	512 GB SSD
OS	Windows

2. Algorithms and Complexity Analysis

2.1 Dijkstra’s Algorithm

Procedure Dijkstra(G, S)

```
BEGIN

Initialize distances: dist[S] ← 0, others ← ∞

Initialize priority queue PQ with (0, S)

WHILE PQ is NOT EMPTY DO

    (d, U) ← EXTRACT-MIN(PQ)

    IF d > dist[U] THEN CONTINUE

    FOR EACH V in Neighbors of U DO

            IF dist[U] + weight(U, V) < dist[V] THEN

                    dist[V] ← dist[U] + weight(U, V)

                    parent[V] ← U
```

INSERT/UPDATE PQ with (dist[V], V)

END IF

END FOR

END WHILE

END

Best Case:

Time: O((V + E) log V)

- Occurs when the graph is sparse ($E \approx V$) and the priority queue operations are efficient. Each vertex is extracted once (O(V log V)), and each edge is relaxed once (O(E log V)).
- In our dataset (~3,783 nodes, ~15,000 edges), this is typical since E is relatively small compared to V^2.

Average Case:

Time: O((V + E) log V)

- On average, Dijkstra's explores most vertices and edges, with priority queue operations dominating the runtime. The binary heap implementation ensures O(log V) per operation (extract-min and decrease-key).

Worst Case:

Time: O((V + E) log V)

- Happens when the graph is dense ($E \approx V^2$). Each vertex is processed once, and each edge relaxation involves a priority queue update. For our dataset, this isn't the case, but the complexity remains the same due to the priority queue.

Space Complexity:

- O(V) for the distance array, parent array, and priority queue (stores up to V vertices).
- Total space: O(V).

2.2 Bellman-Ford Algorithm

Procedure BellmanFord(G, S)

BEGIN

```
Initialize distances: dist[S] ← 0, others ← ∞

FOR I = 1 TO |V| - 1 DO

    FOR EACH edge (U, V, w) in G.E DO

        IF dist[U] + w < dist[V] THEN

            dist[V] ← dist[U] + w

            parent[V] ← U

        END IF

    END FOR

    IF no updates in this iteration THEN BREAK

END FOR
```

END

Best Case:

Time: O(V + E)

- Occurs when early termination kicks in after the first iteration (e.g., a graph where the source is isolated or distances stabilize quickly). Only one pass over all edges is needed.

Average Case:

Time: O(VE)

- Typically, Bellman-Ford performs V-1 iterations over all edges to ensure shortest paths are found. For our dataset (~3,783 nodes, ~15,000 edges), this is significant but mitigated by early termination if updates cease.

Worst Case:

Time: O(VE)

- Happens when the graph requires the full V-1 iterations to compute shortest paths (e.g., a path graph where distances update in each iteration). For our dataset, this means ~3,783 × 15,000 operations, which is computationally expensive.

Space Complexity:

- O(V) for the distance and parent arrays.
- O(E) for storing the edge list (though this is part of the input).
- Total space: O(V), excluding the edge list.

2.3 Diameter (Using All-Pairs Dijkstra's)

Procedure ComputeDiameter(G)

BEGIN

Initialize max_dist ← 0

FOR EACH U in G.V DO

CALL Dijkstra(G, U) to get distances dist[]

FOR EACH V in G.V DO

IF dist[V] ≠ ∞ AND dist[V] > max_dist THEN

max_dist ← dist[V]

Update longest path

END IF

END FOR

END FOR

RETURN max_dist

END

Best Case:

Time: O(V(V + E) log V)

- Occurs when the graph is sparse (E ≈ V), and Dijkstra's is efficient. Each Dijkstra's run takes O((V + E) log V), and we run it V times. For our dataset (~3,783 nodes, ~15,000 edges), this is the typical case.

Average Case:

Time: O(V(V + E) log V)

- On average, each Dijkstra's call explores most vertices and edges, multiplied by V calls. The priority queue operations dominate, as in Dijkstra's.

Worst Case:

Time: O(V(V + E) log V)

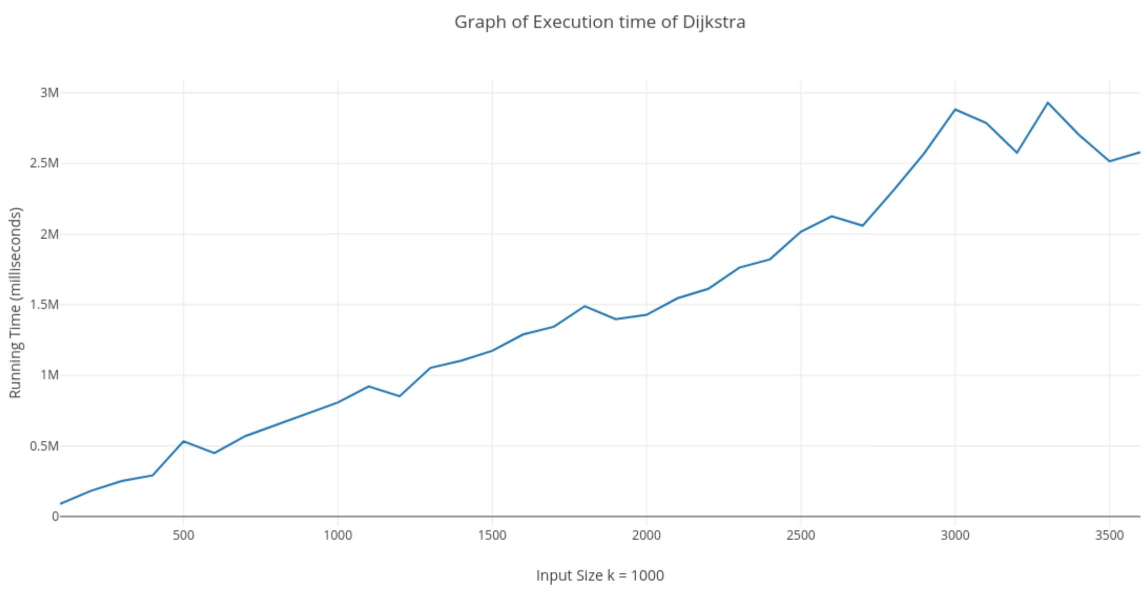
- Happens when the graph is dense ($E \approx V^2$), making each Dijkstra's run $O((V + V^2) \log V) = O(V^2 \log V)$. Total runtime becomes $O(V^3 \log V)$. Our dataset is sparse, so we avoid this worst case.

Space Complexity:

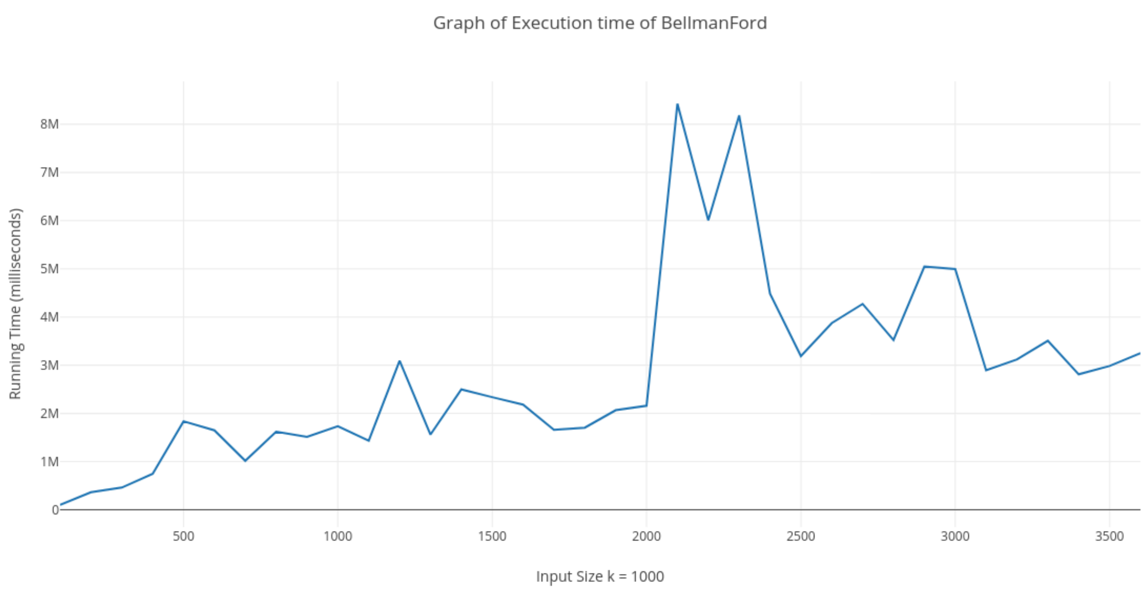
- O(V) for the distance and parent arrays in each Dijkstra's call.
- O(V) for the priority queue per call.
- Total space: O(V), as arrays are reused across calls.

3. Algorithm Graph Plots

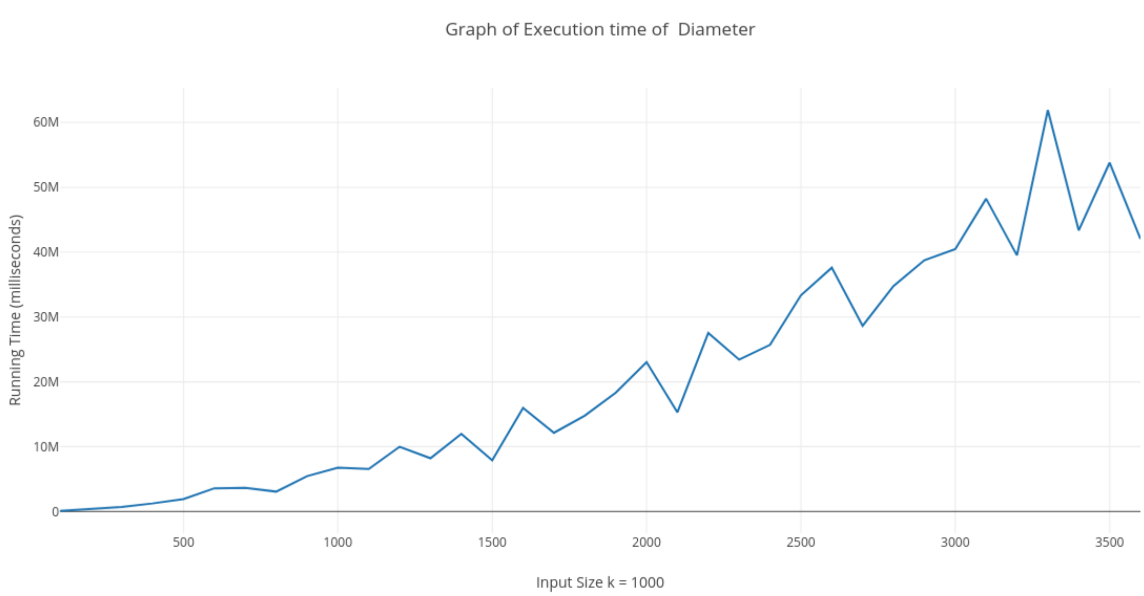
3.1 Dijkstra Performance:



3.2 Bellmanford Performance:



3.3 Diameter Performance:



4. Implementation Details

4.1 Data Structures

Adjacency:

- **Dijkstra's, Bellman-Ford, and Diameter:** Adjacency list implemented as a vector<vector<Edge>> for Dijkstra's and Diameter, and a vector<Edge> for Bellman-Ford. Each Edge struct stores (to, weight) for Dijkstra's and Diameter, and (from, to, weight) for Bellman-Ford, reflecting the directed graph from the dataset (source, target, weight, timestamp).
- **BFS (Not Implemented):** Would use a queue<int> for node indices, with O(1) amortized enqueue/dequeue.
- **DFS & Cycle Detection (Not Implemented):** Would use recursion with a call stack, supplemented by bool Visited[] and bool InStack[] arrays for cycle detection.

Impact on Complexity:

- Core operations (vector access, queue operations, edge traversal) are O(1) per element.
- For Dijkstra's, priority queue (implemented via priority_queue) adds O(log V) per insert/update due to heap operations.
- Overall algorithmic complexity (O((V + E) log V) for Dijkstra's, O(VE) for Bellman-Ford, O(V(V + E) log V) for Diameter) is governed by graph size and algorithm logic, not data structure overhead beyond the priority queue in Dijkstra's.

4.2 Choice of Stack vs. Queue

- **Dijkstra's:**
  - Uses a priority queue (priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<>>) instead of a standard queue or stack. This choice ensures the minimum distance node is always processed next, critical for greedy shortest-path computation. A queue (FIFO) or stack (LIFO) would not maintain the order needed for optimal path selection.

- **Bellman-Ford:**
  - Does not use a queue or stack explicitly; relies on iterative edge relaxation over all edges. The absence of a dynamic data structure simplifies implementation but requires V-1 passes, leveraging the algorithm's inherent structure rather than stack/queue ordering.
- **Diameter:**
  - Internally uses Dijkstra's with a priority queue for each source node to compute all-pairs shortest paths. The priority queue choice mirrors Dijkstra's, ensuring efficient extraction of minimum distance nodes. A queue or stack would not suit the all-pairs approach, as it requires repeated shortest-path computations from each vertex.

Member 2

1. Machine Specification

Component	Specification
Chip	Intel core i5 6th Generation
Memory	8 GB
Storage	512 GB SSD
OS	Windows

2. Algorithms and Complexity Analysis

2.1 Prims Algorithm (MST)

```
Procedure PrimMst.run(Graph, Start Node)
BEGIN
   Initialize an empty set visited
   Initialize a priority queue pq (min-heap)
   Mark start as visited
   Push all edges from start to pq
   WHILE pq is NOT EMPTY DO
      Pop the minimum weight edge (u, v)
      IF v is NOT visited THEN
         Add (u, v) to MST
         Mark v as visited
         Push all edges from v to pq
      END IF
   END WHILE
```

END

Best Case:

- **Time: O(E log V)**
- When graph is sparse and priority queue operations dominate. Each edge is added to the queue at the most once.

Average Case:

- **Time**: **O(E log V)**
- On average performance remains O(E log V) as heap operations determine performance in typical cases.

Worst Case:

- **Time**: **O(E log V)**
- All edges are pushed into the priority queue, and we perform heap operations for each, resulting in O(E log V).

Space Complexity:

- O(V) for the visited array and O(E) for the priority queue.

2.2 Kruskal Algorithm (MST)

```
Procedure KruskalMST.run(Graph)

BEGIN

   Initialize disjoint sets using Union-Find

   Extract all edges and sort by weight

   FOR each edge (u, v) in ascending order of weight DO

      IF u and v are in different components THEN

         Add edge to MST

         Union their components

      END IF

   END FOR

END
```

Time Complexity:

- **Best, Average, and Worst Case: O(E log E)**
  - Best Case: Sorting edges dominates the runtime; few union operations are needed.
  - On average, sorting still dominates, and Union-Find operations are near constant with path compression.
  - Worst: All edges need sorting and checked for cycles.

Space Complexity:

- O(V) for Union-Find parent and rank maps
- O(E) for storing edges
- Total: O(V + E)

2.3 Average Degree Calculation

Time Complexity:

- **Best, Average, and Worst Case**: **O(E)**
  - Best: File read is linear in the number of edges.
  - Average: Each edge is processed once with constant-time map and set operations.
  - Worst: Each edge is processed once with constant-time map and set operations.

Space Complexity:

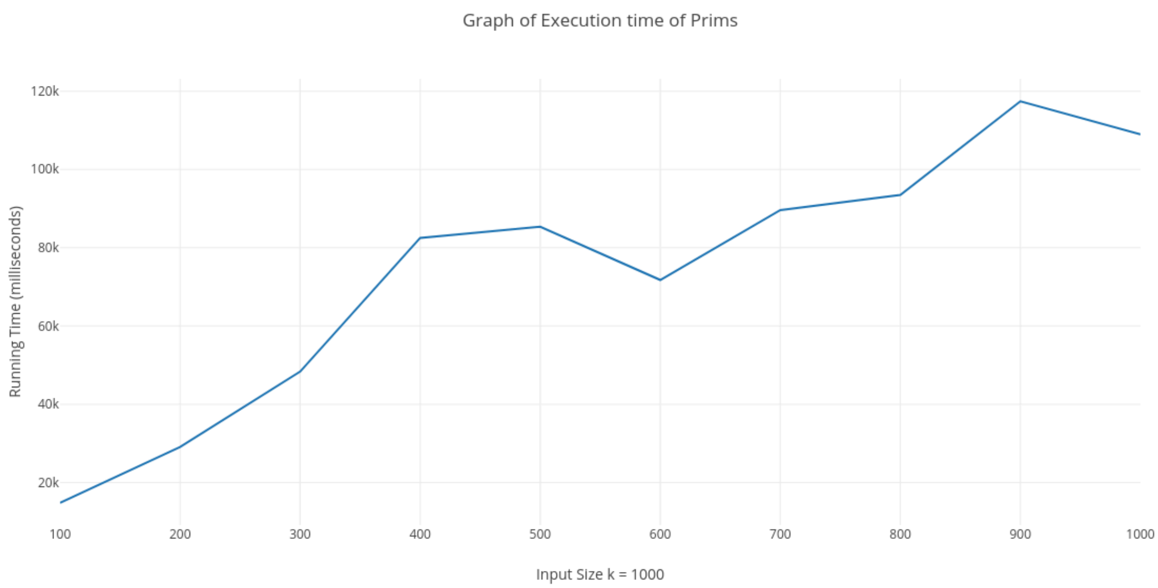
- O(V) for degree map and nodes set
- So, the total space complexity = O(V)

Average degree txt file

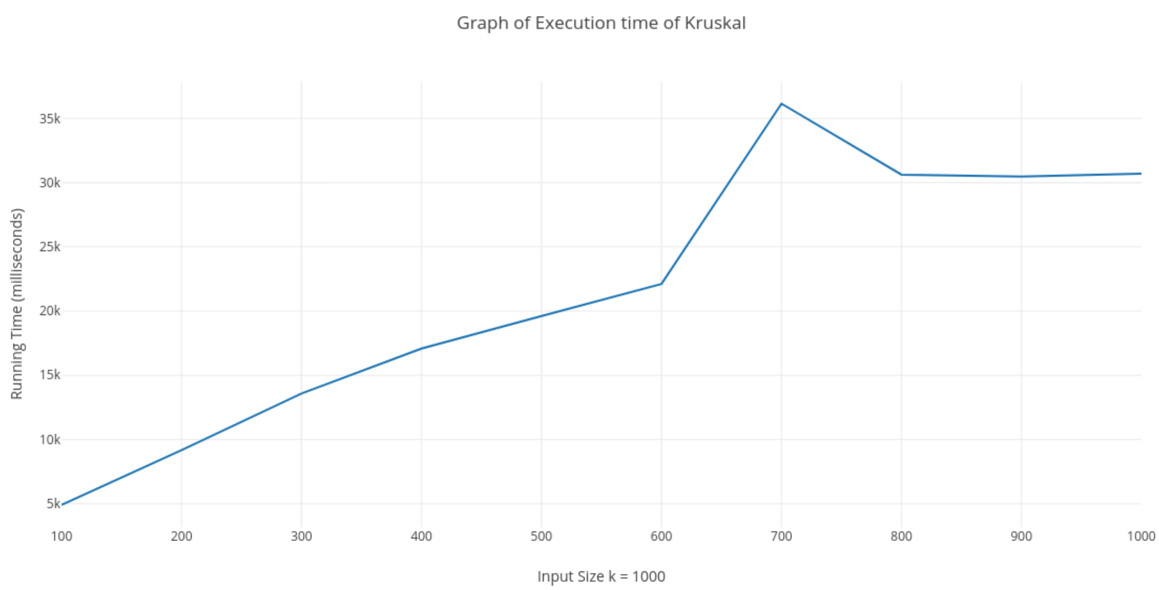
```
degree > average_degree.txt
Average Degree: 12.7867
Total Nodes: 3783
Total Edges: 24186
```

3. Algorithm Graph Plots

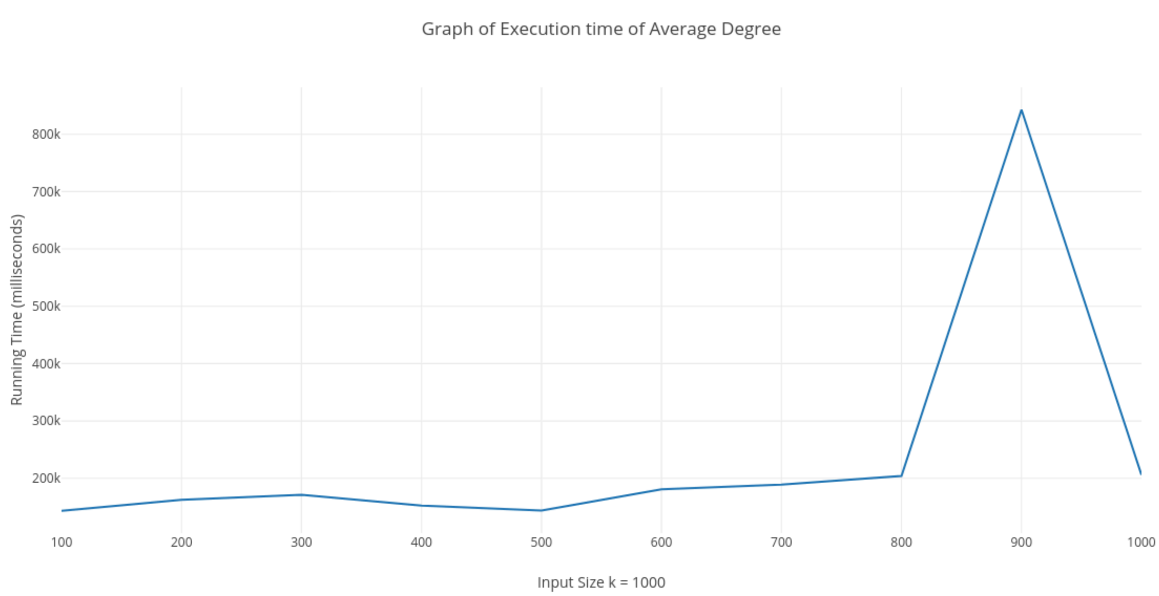
3.1 Prims Algorithm



3.2 Kruskal Algorithm



3.3 Average Degree



4. Implementation Details

4.1 Data Structures

- **Graph:** Stored as an adjacency list using unordered_map<int, vector<pair<int, double>>>
- **Prim's:** Uses set<int> for visited nodes and a **min-heap** (priority queue) to select the smallest edge.
- **Kruskal's:** Uses a **sorted edge list** and a **Union-Find** structure (unordered_map for parent and rank).
- **Average Degree:** Uses unordered_map<int, int> for node degrees and set <int> for unique nodes.
- All operations like insertion, access, and updates are O(1) or O(log E), so they don't impact overall complexity.

4.2 Choice of Stack vs. Queue

- Prim's: Uses a min-heap (not a stack/queue) to ensure greedy selection.
- Kruskal's: Relies on Union-Find, no queue or stack involved.
- Average Degree: Processes data linearly—no need for stack/queue.
- The selected structures ensure efficiency; using incorrect ones (e.g., a regular queue in Prim's) would break correctness or increase time complexity.

4.3 Execution Time

- **Prim Algo : 113300 ms**
- **Kruskal Algo: 104974 ms**
- **Average Degree: 206130 ms**

Member 3

1. Machine Specification

Component	Specification
Model Name	MacBook Pro (16-inch, Mac16,8)
Chip	Apple M4 Pro (8 performance + 4 efficiency cores)
Memory	24 GB LPDDR5
Storage	500 GB SSD
OS	macOS 15.4.1

2. Algorithms and Complexity Analysis

2.1 Breadth-First Search (BFS)

```
Procedure BFS(G, S)
   BEGIN
     FOR I = 0 TO Size of G.V - 1 STEP 1 DO
       Visited[I] ← FALSE
     END FOR
  Create an empty queue Q
     Visited[S] ← TRUE
     ENQUEUE(Q, S)
  WHILE Q is NOT EMPTY DO
       U ← DEQUEUE(Q)
       FOR EACH V in Neighbors of U DO
         IF Visited[V] = FALSE THEN
           Visited[V] ← TRUE
           ENQUEUE(Q, V)
         END IF
       END FOR
     END WHILE
   END
```

Best Case:

- **Time: O(V+E)**
- This is O(V+E) because BFS must check every vertex and edge at most once.

Average Case:

- **Time**: **O(V+E)**
- On average, it explores all vertices and their edges in the connected component of the starting node.

Worst Case:

- **Time**: **O(V+E)**
- Happens when the entire graph is connected, so all vertices and all edges are explored.

Space Complexity:

- O(V) for the visited array and the queue in the worst case (when all vertices are enqueued).

2.2 Depth-First Search (DFS)

Procedure DFS(G, U)

BEGIN

```
Visited[U] ← TRUE

FOR EACH V in Neighbors of U DO
   IF Visited[V] = FALSE THEN
      DFS(G, V)
   END IF
END FOR
```

END

Time Complexity:

- **Best, Average, and Worst Case: O(V+E)**
    - Every vertex is visited once → O(V)
    - Every edge is explored once → O(E)
    - So, total = O(V+E)

Space Complexity:

- **Visited array**: O(V) to track visited nodes

2.3 Cycle Detection in Directed Graphs

Procedure DetectCycle(G)

BEGIN

```
FOR EACH V in G.V DO
   Visited[V] ← FALSE
   InStack[V] ← FALSE
END FOR

FOR EACH V in G.V DO
   IF Visited[V] = FALSE THEN
      IF CALL DFS_Cycle(V) THEN
         RETURN TRUE
      END IF
   END IF
END FOR

RETURN FALSE
```

END

Procedure DFS_Cycle(U)

BEGIN

```
Visited[U] ← TRUE
InStack[U] ← TRUE

FOR EACH V in Neighbors of U DO
   IF InStack[V] = TRUE THEN
      RETURN TRUE // Back edge ⇒ cycle
   ELSE IF Visited[V] = FALSE AND CALL DFS_Cycle(V) THEN
      RETURN TRUE
   END IF
END FOR
```

InStack[U] ← FALSE
RETURN FALSE

END

Time Complexity:

- **Best, Average, and Worst Case**: **O(V+E)**
  - Every vertex is visited once, and every edge is explored once during the DFS traversal. The cycle check involves checking neighbors but still follows the same traversal pattern.

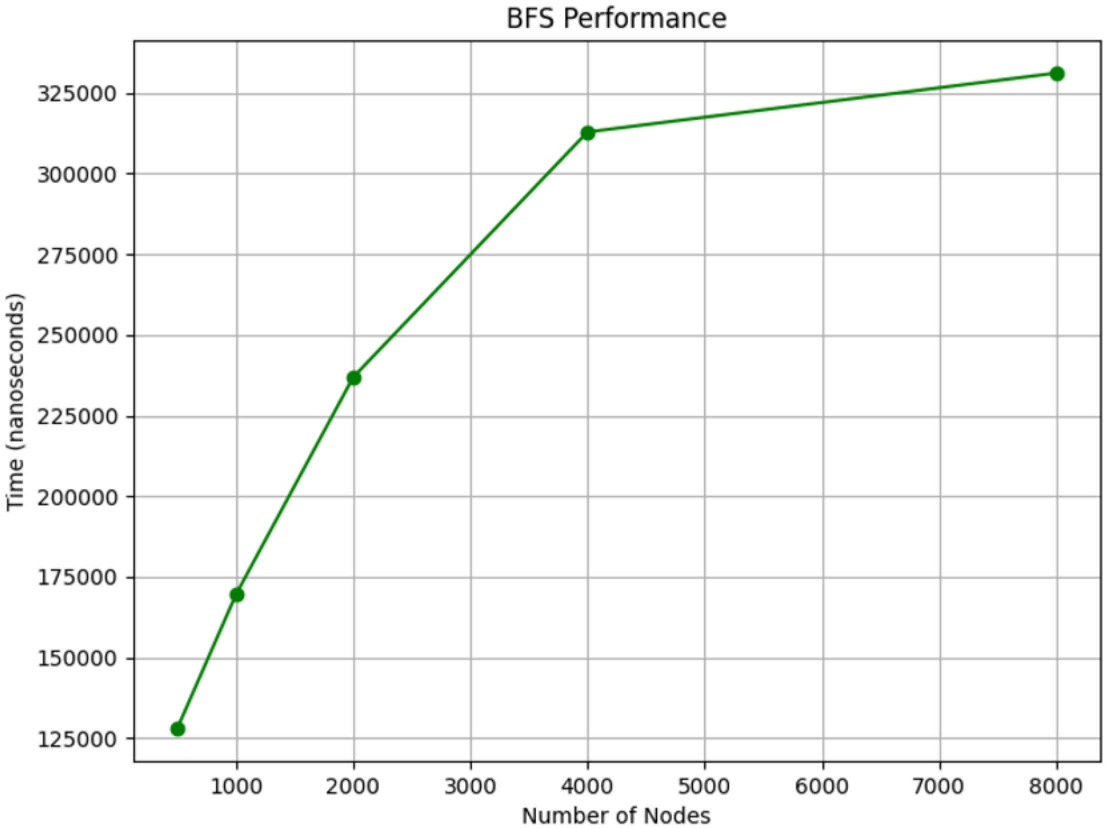
Space Complexity:

- Visited array: O(V)
- InStack array: O(V)
- Call stack (in DFS): O (V
- So, the total space complexity = O(V)

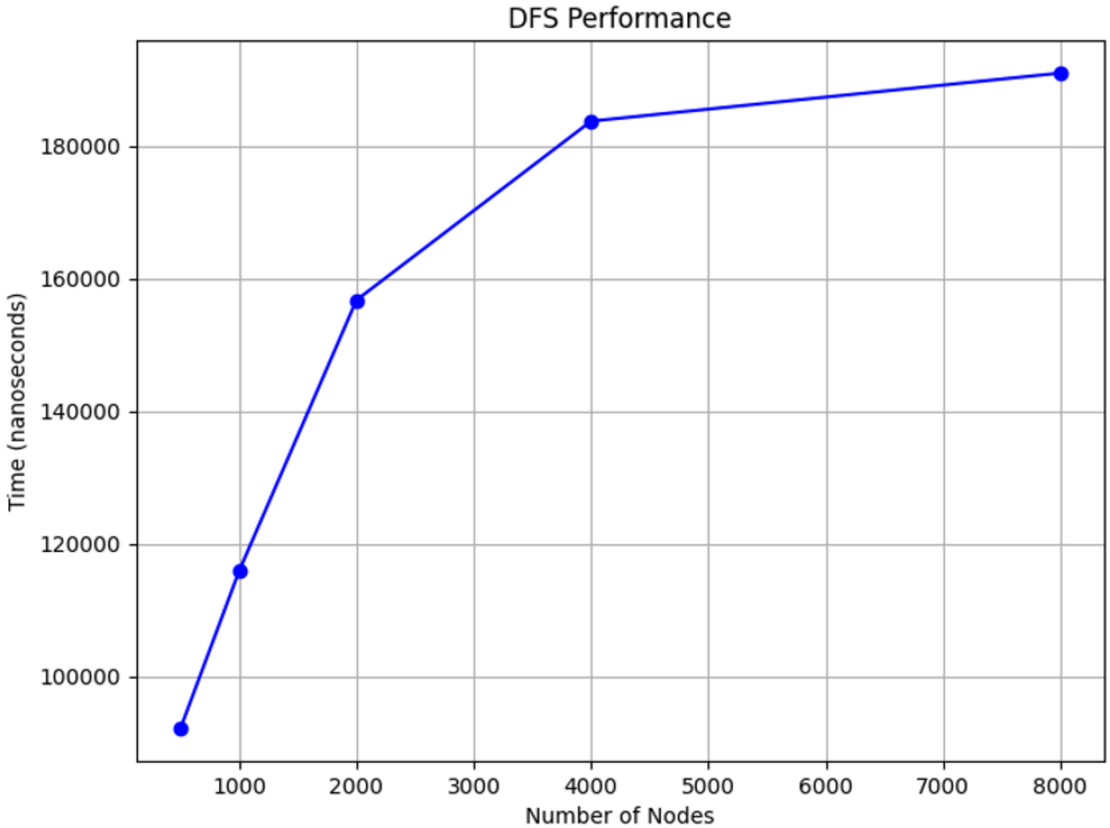
3. Algorithm Graph Plots

3.1 BFS Performance

Time grows roughly linearly with |V| + |E|.



3.2 DFS Performance



Similar linear trend; slightly faster than BFS

3.3 Cycle Detection Performance

Measured in microseconds, same in linear *O(V+E)*.

4. Implementation Details

4.1 Data Structures

Adjacency:

- LinkedList per vertex for neighbors; each node stores (target, rating, timestamp).

BFS:

- Queue built from the same Node type.
- Enqueue/dequeue in O(1) amortized.

DFS & Cycle Detection:

- Recursive calls manage their own call‑stack.

- Additionally for cycle detection, we maintain a separate boolean inRecursionStack[].

Impact on Complexity:

- All core operations (enqueue, dequeue, push, pop, neighbor traversal) remain O(1).
- Thus overall algorithmic complexity is governed purely by the graph size, not by data‑structure overhead.

4.2 Choice of Stack vs. Queue

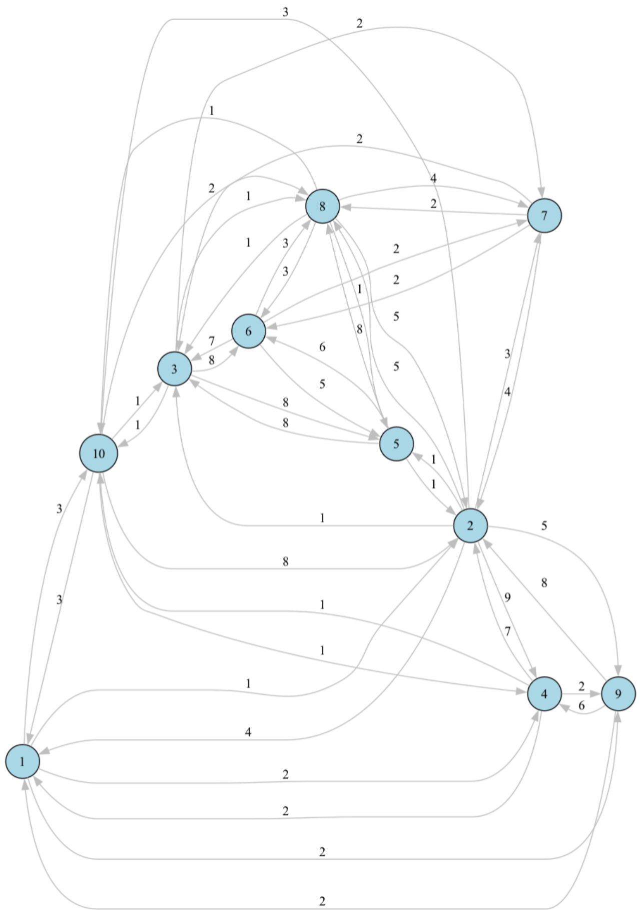
· DFS via Recursion (Call Stack)

I deliberately used recursion instead of a Stack class. With recursion, when you visit a node and push all its neighbors onto the call stack, the *first* neighbor pushed is automatically the *first* one you explore, exactly matching DFS order. If I use a stack, I must reverse the neighbor list each time to achieve the same behavior.

· BFS via Queue

For breadth-first search, a FIFO Queue is the natural and only data structure that can be used.

5. Graph Visualization



-