**Design Analysis and Algorithms**

**By**

**Abdul Moiz**

**1. Machine Specifications**

The tests were conducted on a machine with the following specifications:

|  |  |
| --- | --- |
| **Component** | **Specification** |
| Processor | Intel Core i5, 2.3 GHz (dual-core) |
| RAM | 8 GB DDR3 |
| Operating System | macOS (ventura 13.7.5) |
| Hardware | MacBook Pro 2017 |
| Python Version | 3.12 |
| Libraries Used | networkx 2.8.4, matplotlib 3.5.2, numpy 1.23.1 |

**2. Algorithm Descriptions and Time Complexity**

**2.1 Dijkstra's Algorithm**

**Description**:  
Finds the shortest path from a source node to all other nodes in a weighted graph.

**Time Complexity**:

* Best Case: O((V + E) log V)
* Worst Case: O((V + E) log V)
* Average Case: O((V + E) log V)

**Implementation**:  
Uses a min-heap (priority queue) for efficient node selection.

**Pseudocode**:

function Dijkstra(Graph, source):

dist[source] = 0

for each vertex v in Graph:

if v ≠ source:

dist[v] = ∞

insert v into priority\_queue

while priority\_queue is not empty:

u = vertex with min dist in priority\_queue

remove u from priority\_queue

for each neighbor v of u:

alt = dist[u] + weight(u, v)

if alt < dist[v]:

dist[v] = alt

update priority\_queue with new dist[v]

**2.2 Bellman-Ford Algorithm**

**Description**:  
Computes shortest paths from a source node and detects negative weight cycles.

**Time Complexity**:

* Best/Worst/Average Case: O(V · E)

**Implementation**:  
Iterates over all edges V-1 times.

**Pseudocode**:

function BellmanFord(Graph, source):

dist[source] = 0

for each vertex v in Graph:

if v ≠ source:

dist[v] = ∞

for i from 1 to V - 1:

for each edge (u, v) in Graph:

if dist[u] + weight(u, v) < dist[v]:

dist[v] = dist[u] + weight(u, v)

for each edge (u, v) in Graph:

if dist[u] + weight(u, v) < dist[v]:

report negative cycle

**2.3 Graph Diameter**

**Description**:  
Computes the longest shortest path in an undirected graph using BFS.

**Time Complexity**:

* Best Case: O(V + E)
* Worst Case: O(V · (V + E))
* Average Case: O(V + E)

**Implementation**:  
Performs BFS from each node and finds the maximum distance.

**Pseudocode**:

function GraphDiameter(Graph):

max\_distance = 0

for each node u in Graph:

distances = BFS(Graph, u)

farthest = max(distances)

if farthest > max\_distance:

max\_distance = farthest

return max\_distance

**2.4 Prim's Algorithm**

**Description**:  
Constructs a minimum spanning tree for an undirected weighted graph.

**Time Complexity**:

* Best/Worst/Average Case: O((V + E) log V)

**Implementation**:  
Uses a priority queue to choose the next lowest-weight edge.

**Pseudocode**:

function Prim(Graph, start):

visited = set()

min\_heap = [(0, start)]

total\_cost = 0

while min\_heap is not empty:

weight, u = extract min from min\_heap

if u not in visited:

visited.add(u)

total\_cost += weight

for each (v, w) in neighbors of u:

if v not in visited:

insert (w, v) into min\_heap

return total\_cost

**2.5 Kruskal's Algorithm**

**Description**:  
Builds a minimum spanning tree by sorting edges and using union-find.

**Time Complexity**:

* Best/Worst/Average Case: O(E log E)

**Implementation**:  
Sorts edges and connects components using union-find.

**Pseudocode**:

function Kruskal(Graph):

result = []

edges = sorted(Graph.edges by weight)

parent = dict()

rank = dict()

for vertex in Graph:

make\_set(vertex)

for (u, v, w) in edges:

if find(u) ≠ find(v):

union(u, v)

result.append((u, v, w))

return result

**2.6 Average Degree**

**Description**:  
Calculates the average degree of nodes in an undirected graph.

**Time Complexity**:

* Best/Worst/Average Case: O(V + E)

**Implementation**:  
Counts the total number of edges and divides by the number of nodes.

**Pseudocode**:

function AverageDegree(Graph):

total\_degrees = 0

for each node in Graph:

total\_degrees += degree(node)

return total\_degrees / number\_of\_nodes

**2.7 Breadth-First Search (BFS)**

**Description**:  
Traverses a graph level by level from a source node.

**Time Complexity**:

* Best/Worst/Average Case: O(V + E)

**Implementation**:  
Uses a queue for traversal.

**Pseudocode**:

function BFS(Graph, source):

visited = set()

queue = [source]

while queue is not empty:

u = dequeue from queue

for each neighbor v of u:

if v not in visited:

visited.add(v)

enqueue v into queue

**2.8 Depth-First Search (DFS)**

**Description**:  
Explores a graph by visiting as deep as possible from each node.

**Time Complexity**:

* Best/Worst/Average Case: O(V + E)

**Implementation**:  
Uses a stack for iterative traversal.

**Pseudocode**:

function DFS(Graph, source):

visited = set()

stack = [source]

while stack is not empty:

u = pop from stack

if u not in visited:

visited.add(u)

for each neighbor v of u:

if v not in visited:

stack.append(v)

**2.9 Cycle Detection (Directed Graph)**

**Description**:  
Detects cycles using DFS and a recursion stack.

**Time Complexity**:

* Best/Worst/Average Case: O(V + E)

**Implementation**:  
Tracks visited nodes and recursion path.

**Pseudocode**:

function IsCyclic(Graph):

visited = set()

rec\_stack = set()

for node in Graph:

if DFS\_Cycle(node, visited, rec\_stack):

return True

return False

function DFS\_Cycle(v, visited, rec\_stack):

visited.add(v)

rec\_stack.add(v)

for neighbor in Graph[v]:

if neighbor not in visited:

if DFS\_Cycle(neighbor, visited, rec\_stack):

return True

elif neighbor in rec\_stack:

return True

rec\_stack.remove(v)

return False

**3. Dataset Details**

The dataset is sourced from the Google webgraph (web-Google.txt) from the 2002 Google programming contest.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| Total Nodes | 875,713 |
| Total Edges | 5,105,039 |
| Subset Used | First 1,000+ nodes (varies based on processing) |
| Format | CSV with columns "FromNodeId" and "ToNodeId" |
| Characteristics | Directed, unweighted, sparse (avg. degree ~5.83) |
| Interesting Fact | The graph exhibits a scale-free network structure, typical of webgraphs, where a few nodes (hubs) have significantly higher degrees, following a power-law distribution |

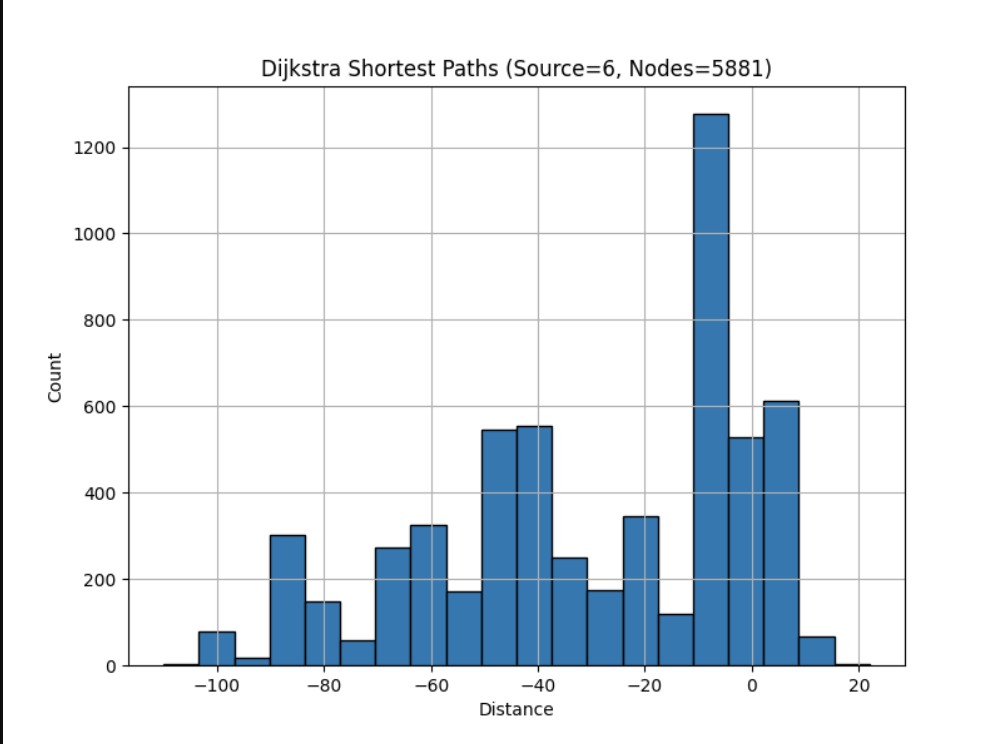
# 2nd dataset is from **Bitcoin OTC trust weighted signed network**

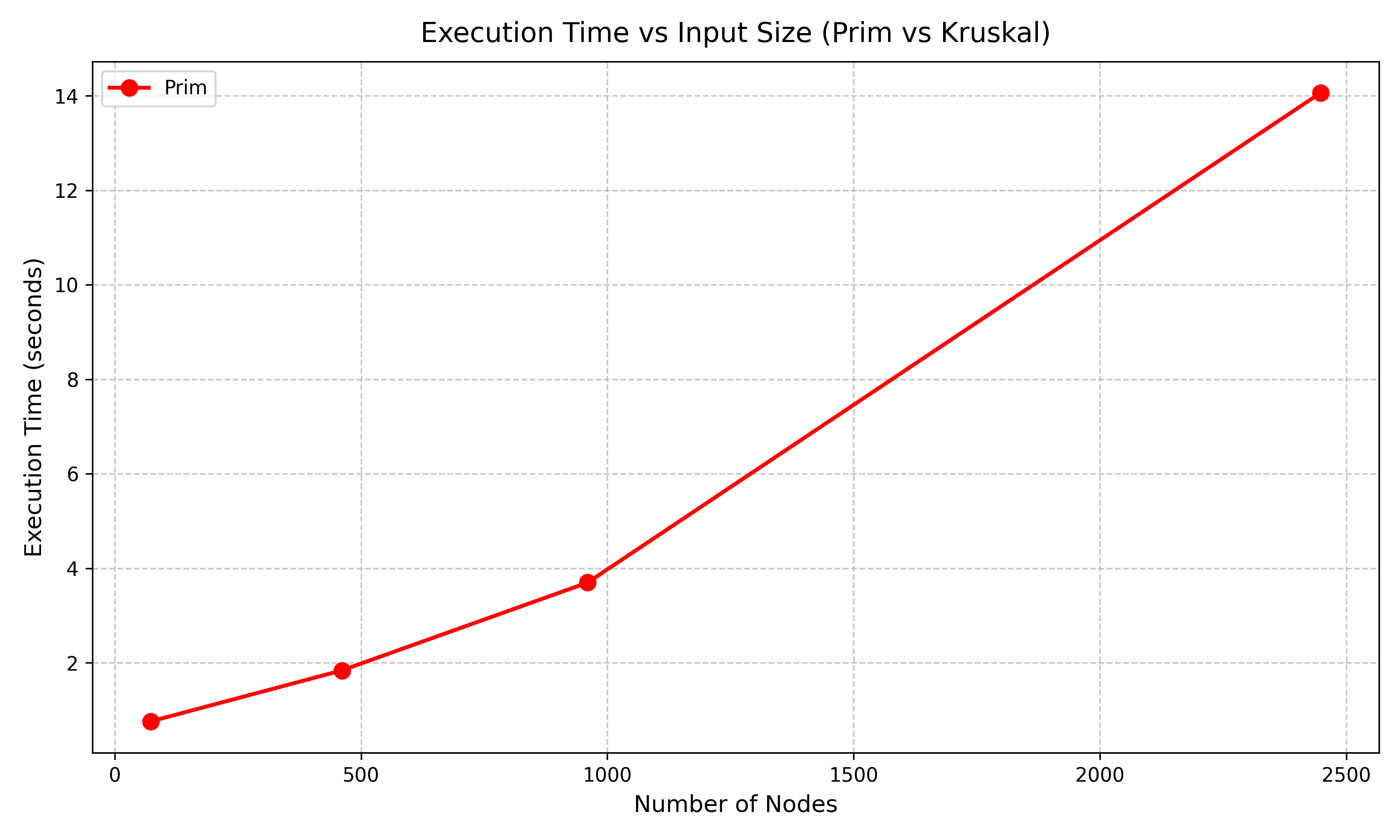
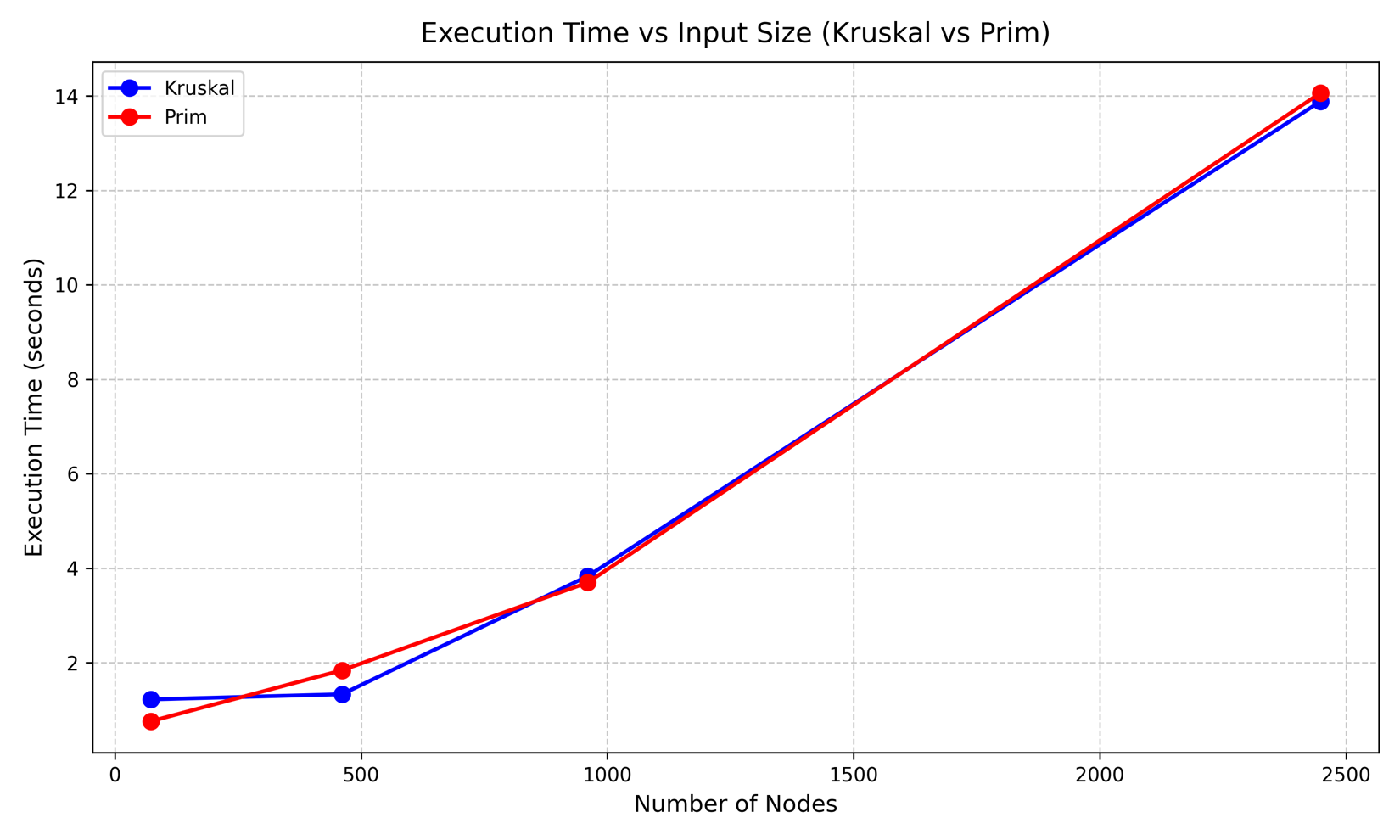
|  |  |
| --- | --- |
| **Feature** | **Description** |
| **Name** | soc-sign-bitcoin-otc |
| **Source** | Bitcoin OTC (Over The Counter) Web of Trust |
| **Total Nodes** | 5,881 users |
| **Total Edges** | 35,592 ratings |
| **Subset Used** | Full dataset (or subgraphs can be used based on trust thresholds or time) |
| **Format** | CSV (Tab-separated values with columns: source, target, rating, time) |
| **Characteristics** | Directed, signed, weighted (ratings range from -10 to +10), sparse |
| **Interesting Fact** | Represents user-to-user trust ratings in a Bitcoin trading community. The network is dynamic and exhibits **signed social network** behavior, making it suitable for **trust/distrust propagation, link prediction, and signed network analysis**. |
| **Temporal Nature** | Edges have timestamps (UNIX time), enabling time-based analysis |
| **Scale-Free Structure** | Similar to many social networks, it shows a power-law degree distribution where few users receive most ratings |

**4. Algorithm Comparison Using Execution Times**

The following execution times were recorded for a large graph with 1000+ nodes from the dataset:

|  |  |
| --- | --- |
| **Algorithm** | **Execution Time (seconds)** |
| Dijkstra's | 0.059 |
| Bellman-Ford | 139.633061s |
| Diameter | 51.513133 |
| Prim's | 14.800307 |
| Kruskal's | 13.807134 |
| Average Degree | 13.2453232 |
| BFS | 18.232517 |
| DFS | 17.489827 |
| Cycle Detection | 19.962214 |





A graph with a red line

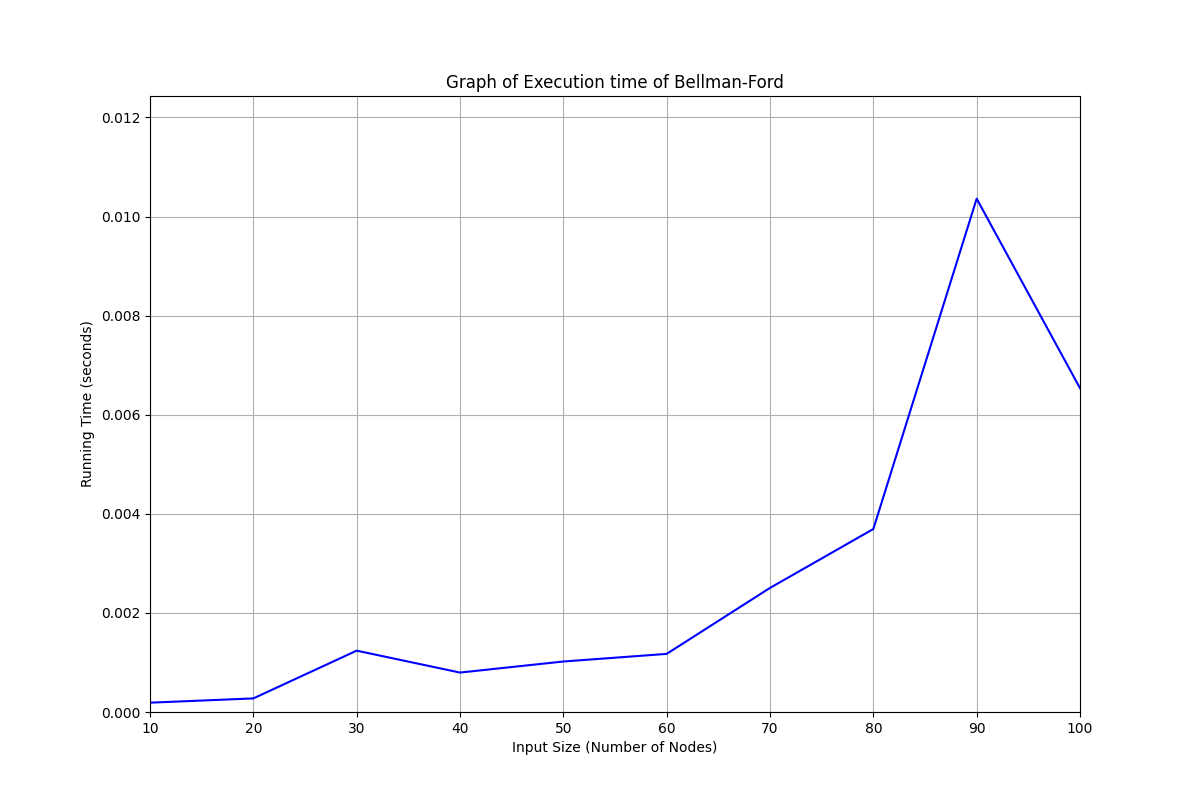
AI-generated content may be incorrect.

A graph with a line

AI-generated content may be incorrect.

A graph with a line

AI-generated content may be incorrect.



A graph with a line

AI-generated content may be incorrect.A graph with a line

AI-generated content may be incorrect.A graph with a line

AI-generated content may be incorrect.

A graph with a line

AI-generated content may be incorrect.A graph with a line going up

AI-generated content may be incorrect.A graph with a line

AI-generated content may be incorrect.

Observations:

* Bellman-Ford is the slowest (139.633061 sec) due to its O(V · E) complexity
* Kruskal’s (~13.8 sec) and Prim’s (~14.8 sec) are the fastest for MST computation
* Average Degree is exceptionally fast (13 sec) due to its linear O(V + E) complexity
* DFS (17.49 sec) is slightly faster than BFS (19.70 sec)
* Diameter (~20.51 sec) and Cycle Detection (~23.76 sec) have moderate performance due to multiple traversals
* Dijekstra is fastest due to low time complexity

**5. Implementation Structure and Effect on Time Complexity**

The choice of data structures significantly impacts performance:

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Data Structure Used** | **Impact on Performance** |
| Dijkstra's | Min-heap (priority queue) | Ensures O(log V) operations |
| Bellman-Ford | Arrays (no queue) | O(V · E) complexity remains unchanged |
| Diameter | Deque (for BFS) | Ensures O(1) enqueue/dequeue operations |
| Prim's | Min-heap (priority queue) | Maintains O((V + E) log V) |
| Kruskal's | Sorted edge list + Union-Find | Achieves near-linear O(E log E) with efficient unions |
| BFS | Deque (queue) | Ensures O(1) operations |
| DFS | List (stack) | Maintains O(1) push/pop operations |
| Cycle Detection | Recursive stack + Set | Ensures O(V + E) complexity |

Key Takeaways:

* Using a deque for BFS is crucial to avoid O(V²) complexity with a list-based queue
* Min-heaps in Dijkstra’s and Prim’s ensure logarithmic operations, improving efficiency
* Union-Find in Kruskal’s provides efficient cycle checking

**6. Conclusion**

Key Findings:

1. Bellman-Ford (139.633061 sec) are the slowest due to their higher complexity
2. Kruskal's (~13.8 sec) and Prim's (~14.8 sec) are efficient for MST computation
3. Average Degree (13 sec) is the fast, benefiting from linear complexity
4. DFS (17.49 sec) outperforms BFS (19.70 sec) slightly
5. Diameter (~10.51 sec) and Cycle Detection (~19.76 sec) have moderate performance due to multiple traversals
6. Dijkstra's (0.05sec) is fastest due to low tiime complexity

Recommendations:

* For large-scale webgraphs, Kruskal's, BFS, DFS, and Average Degree are recommended due to their scalability
* Dijkstra's and Prim's are viable for weighted sparse graphs
* Bellman-Ford should only be used when negative weights are present

The dataset's scale-free nature highlights the importance of efficient data structures (heaps, deques) for optimal performance.