Incremental Shortest Path Algorithm for Dynamic Network Optimization

Joel Joseph Kinny (jk2112)
Sumedh Marathe (sam792)
Apurv Paliwal (ap2523)

Agenda

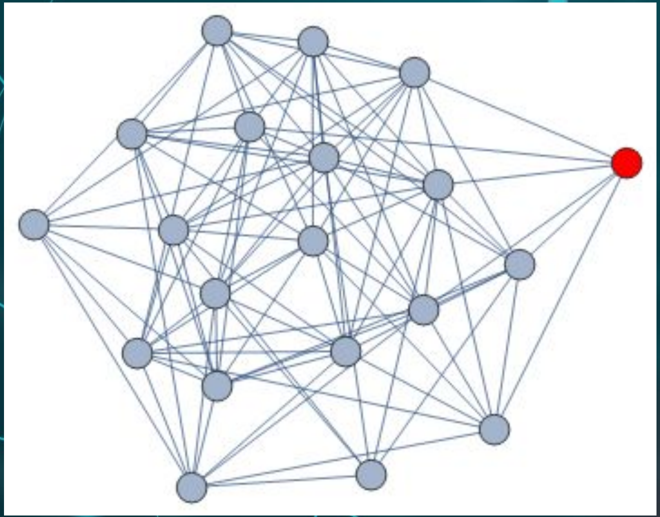
Introduction

In graph theory, computation of the shortest path between node pairs are termed Shortest Path Computation

The All Pairs Shortest Path is generally used for dynamic networks such as airline and railway systems are subject to frequent changes. These changes necessitate rapid recalculations of optimal routes to ensure efficiency and service reliability.

1:05 PM 1:05 PM B11 Scheduled
7:40 AM 10:15 AM C21 Delayed
9:40 AM 10:32 AM C27 Delayed
10:25 AM 10:25 AM A3 Scheduled
10:46 AM 10:46 AM C23

Background



- The All Pairs Shortest Path is a class of graph algorithms designed to calculate the minimal cost to traverse between all nodes of a graph
- Traditional Algorithm: The Floyd-Warshall algorithm is a common solution for finding shortest paths in a weighted graph. It calculates the shortest paths between all pairs of vertices efficiently with a time complexity of $O(V^3)$, where V is the number of vertices.
- Limitation: While effective for static scenarios, Floyd-Warshall's computational expense becomes a significant drawback in dynamic settings where only a subset of weights (costs) changes. Recomputing the entire matrix for minor updates is not practical.

Related Work

- The dynamic weight update has seen a long evolution in graph theory where the shortest path needs to be computed in lesser complexity than $O(V^3)$
- There have been couple of methods that have stood out and achieved better amortized complexities for edge updates/edge insertions
- The Ramalingam and Reps algorithm (1996) for Single Pair Shortest Path (SSSP) and its equivalent version for All Pairs Shortest Path (ASSP) is considered as a basis for achieving a benchmark for dynamic graph shortest path problems
- The QUINCA algorithm (2016) is a improvement over the Ramalingam and Reps algorithm and offers better amortized updates and faster execution time over other state of the art algorithms with a complexity of $O(V^2)$

Goals and Motivation

- Dynamic Updates are divided into Incremental (node insertions/node weight decrease), decremental (node deletions/node weight increase) and full dynamic (handles both)

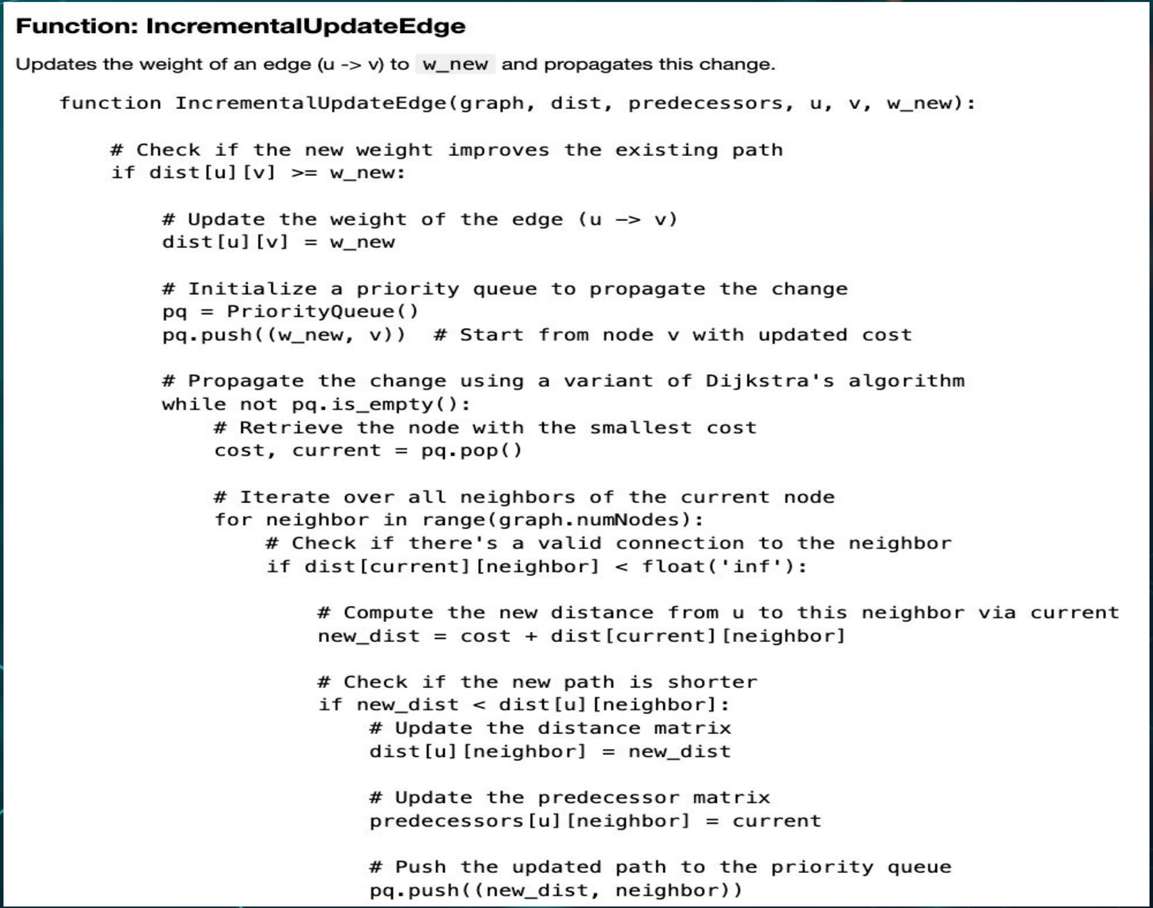
- The primary goal of our project is to design and implement an incremental algorithm that updates shortest paths efficiently in a dynamic network, specifically when minimal changes occur, such as a single edge adjustment or new node insertion as seen in real-world flight networks.

- The algorithm seeks to avoid the computational overhead of recalculating entire path matrices, focusing only on affected paths to enhance responsiveness and efficiency in real-time applications.

Methodology

- The graph has been represented as adjacency matrix for ease of indexing and path reconstruction updates within the dynamic updates.
- The initial shortest path calculations is done using the Floyd Warshall Algorithm which computes the shortest path between all nodes of the graph
- Once the shortest path and predecessor matrix has been calculated for the current set of nodes, any incremental changes in the graph will be handled by two new methods employed in our adaption of incremental shortest paths computation
- On edge updates, the new edge weights are assigned to the shortest path between two nodes if it is lower than the earlier computed shortest path
- On node insertions, we provide a set of incoming and outgoing nodes for this newly added edge and dynamically update the shortest path between all nodes

Incremental Edge Update

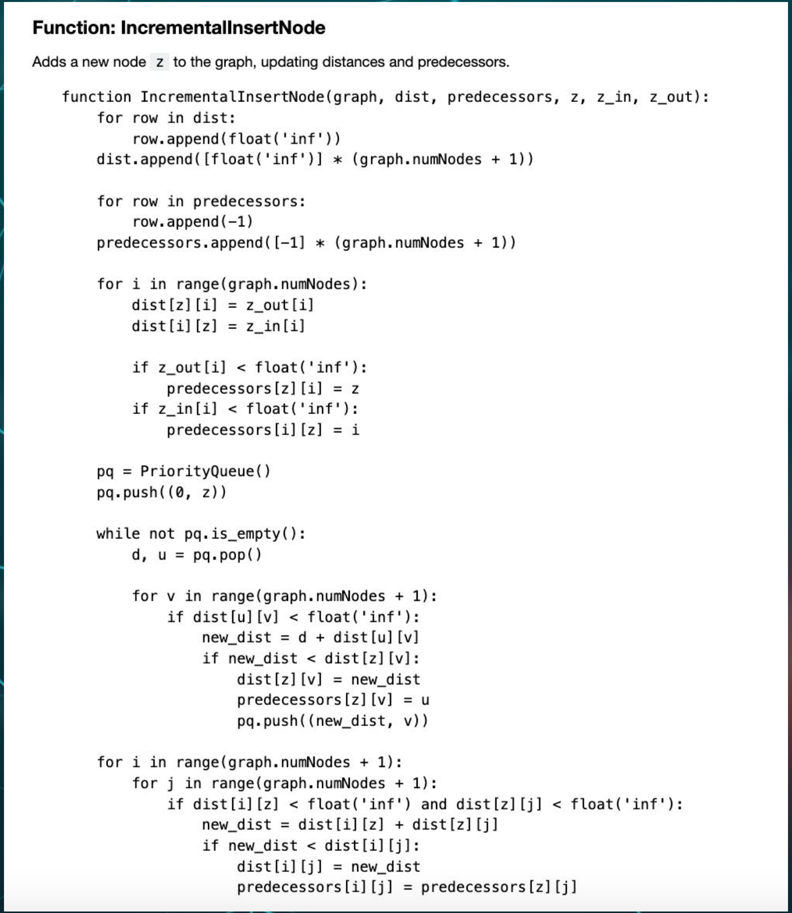


Function: IncrementalUpdateEdge

Updates the weight of an edge (u -> v) to `w_new` and propagates this change.

```
function IncrementalUpdateEdge(graph, dist, predecessors, u, v, w_new):

    # Check if the new weight improves the existing path
    if dist[u][v] >= w_new:

        # Update the weight of the edge (u -> v)
        dist[u][v] = w_new

        # Initialize a priority queue to propagate the change
        pq = PriorityQueue()
        pq.push((w_new, v))  # Start from node v with updated cost

        # Propagate the change using a variant of Dijkstra's algorithm
        while not pq.is_empty():
            # Retrieve the node with the smallest cost
            cost, current = pq.pop()

            # Iterate over all neighbors of the current node
            for neighbor in range(graph.numNodes):
                # Check if there's a valid connection to the neighbor
                if dist[current][neighbor] < float('inf'):

                    # Compute the new distance from u to this neighbor via current
                    new_dist = cost + dist[current][neighbor]

                    # Check if the new path is shorter
                    if new_dist < dist[u][neighbor]:
                        # Update the distance matrix
                        dist[u][neighbor] = new_dist

                        # Update the predecessor matrix
                        predecessors[u][neighbor] = current

                        # Push the updated path to the priority queue
                        pq.push((new_dist, neighbor))
```

Incremental Node Insertion



Function: IncrementalInsertNode

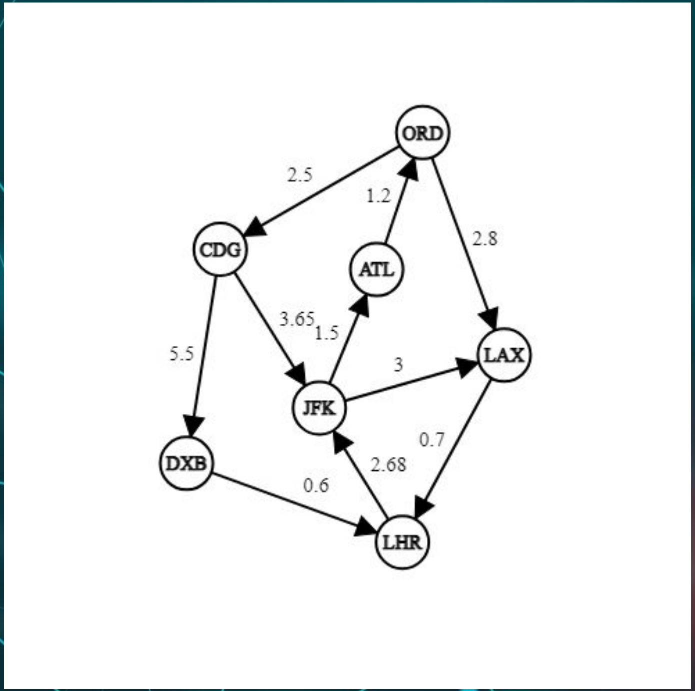
Adds a new node `z` to the graph, updating distances and predecessors.

```
function IncrementalInsertNode(graph, dist, predecessors, z, z_in, z_out):
    for row in dist:
        row.append(float('inf'))
    dist.append([float('inf')] * (graph.numNodes + 1))

    for row in predecessors:
        row.append(-1)
    predecessors.append([-1] * (graph.numNodes + 1))

    for i in range(graph.numNodes):
        dist[z][i] = z_out[i]
        dist[i][z] = z_in[i]

        if z_out[i] < float('inf'):
            predecessors[z][i] = z
        if z_in[i] < float('inf'):
            predecessors[i][z] = i

    pq = PriorityQueue()
    pq.push((0, z))

    while not pq.is_empty():
        d, u = pq.pop()

        for v in range(graph.numNodes + 1):
            if dist[u][v] < float('inf'):
                new_dist = d + dist[u][v]
                if new_dist < dist[z][v]:
                    dist[z][v] = new_dist
                    predecessors[z][v] = u
                    pq.push((new_dist, v))

    for i in range(graph.numNodes + 1):
        for j in range(graph.numNodes + 1):
            if dist[i][z] < float('inf') and dist[z][j] < float('inf'):
                new_dist = dist[i][z] + dist[z][j]
                if new_dist < dist[i][j]:
                    dist[i][j] = new_dist
                    predecessors[i][j] = predecessors[z][j]
```

This function efficiently integrates a new node into a graph by:

- Expanding the distance and predecessor matrices.
- Setting up initial distances.
- Updating paths using Dijkstra's algorithm and considering z as an intermediate node.
- Ensuring accurate shortest paths between all pairs, avoiding full recomputation.

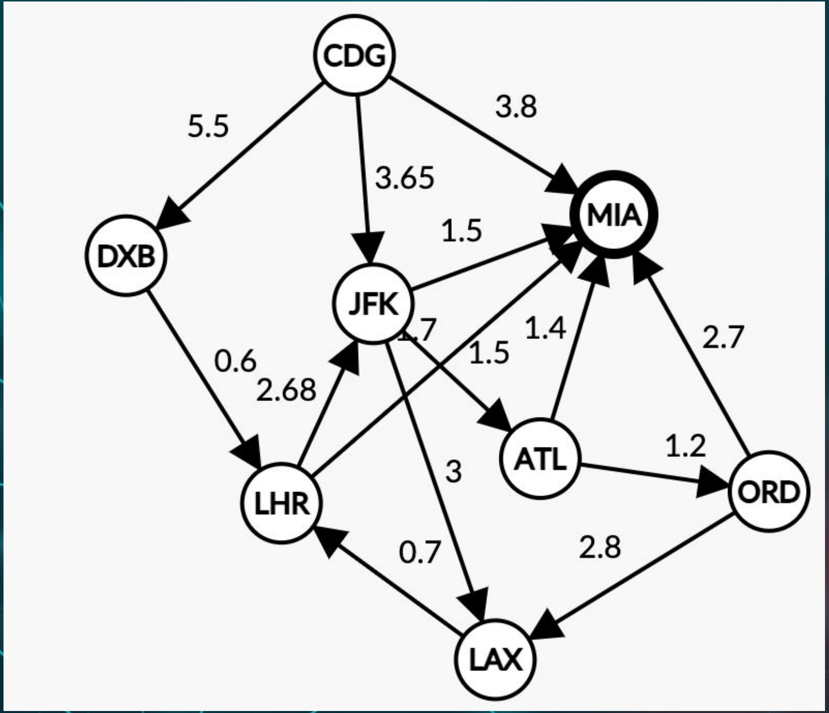
Example Run



The graph represents a directed network of airports, where:

- Nodes: Labeled by airport codes (JFK, LAX, ORD, ATL, LHR, CDG, DXB).
- Edges: Directed arrows between nodes indicate direct flights between these airports.
- Weights: The weights on the edges denote the cost (or time) associated with each flight.

Adding a node

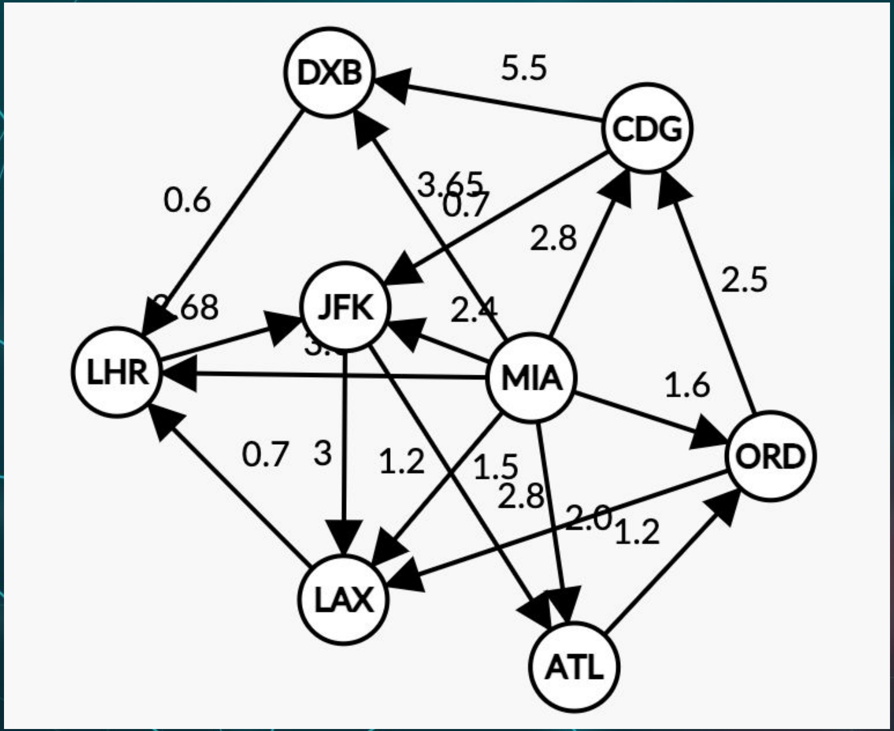


- The initial graph consists of 7 nodes representing different airports and the connection of these nodes are defined by the earlier set distance matrix(dist) and predecessor matrix(predecessors).

- We now add the a new airport, MIA, Weights from existing nodes to "MIA":

  z_in = [1.5, INF, 1.4, 2.7, 3.8, INF, 1.7]
- Weights from "MIA" to existing nodes:

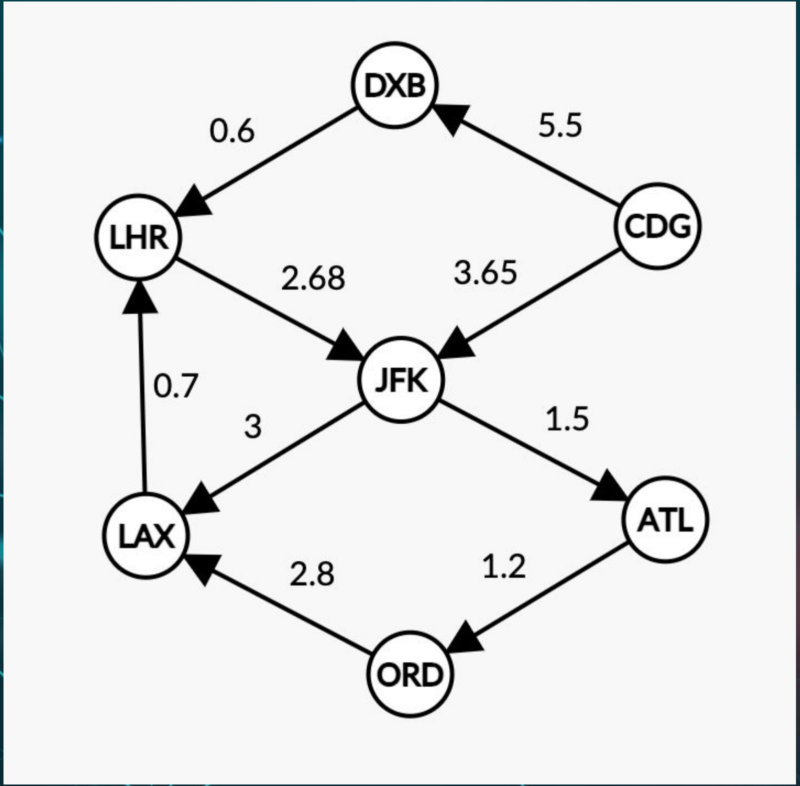
  z_out = [2.4, 1.2, 2.0, 1.6, 2.8, 0.7, 3.0]
- We resize the matrix i.e. the distance and predecessor matrix.
- We update both inbound and outbound distance matrices



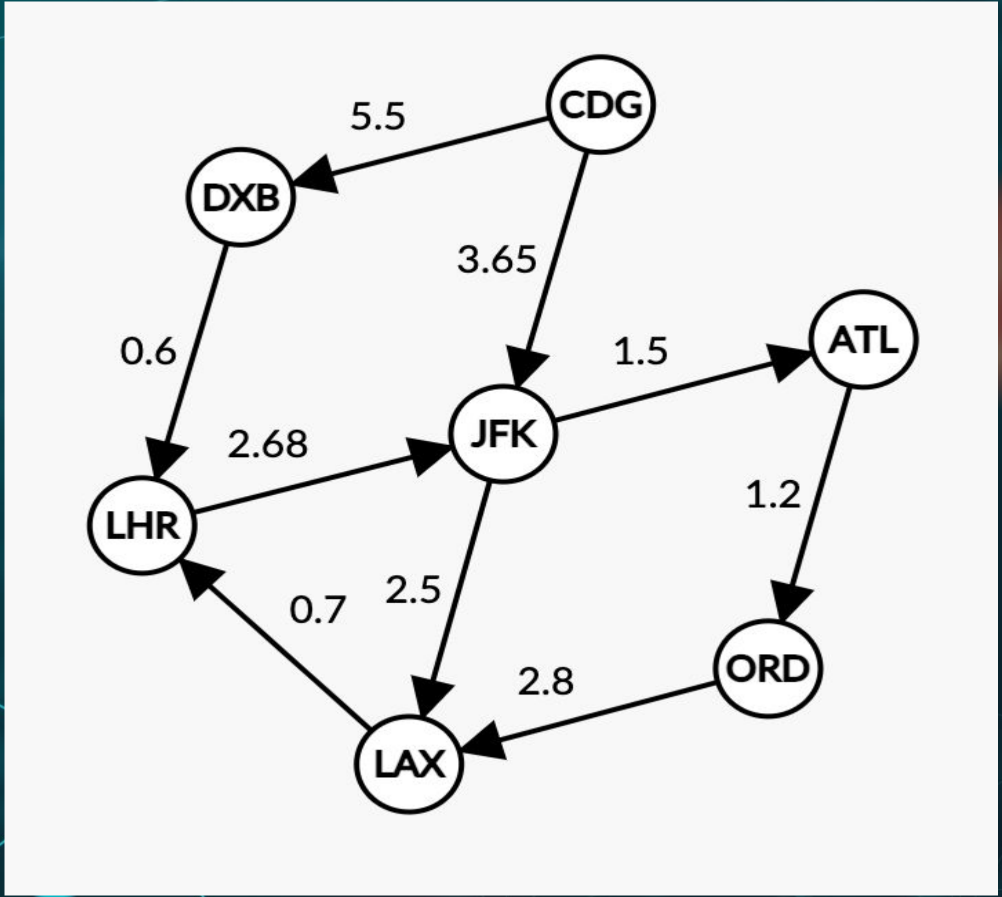
- We use Dijkstra’s algorithm for new paths

- Finally we update the distances and predecessors for all pairs affected by the edge update

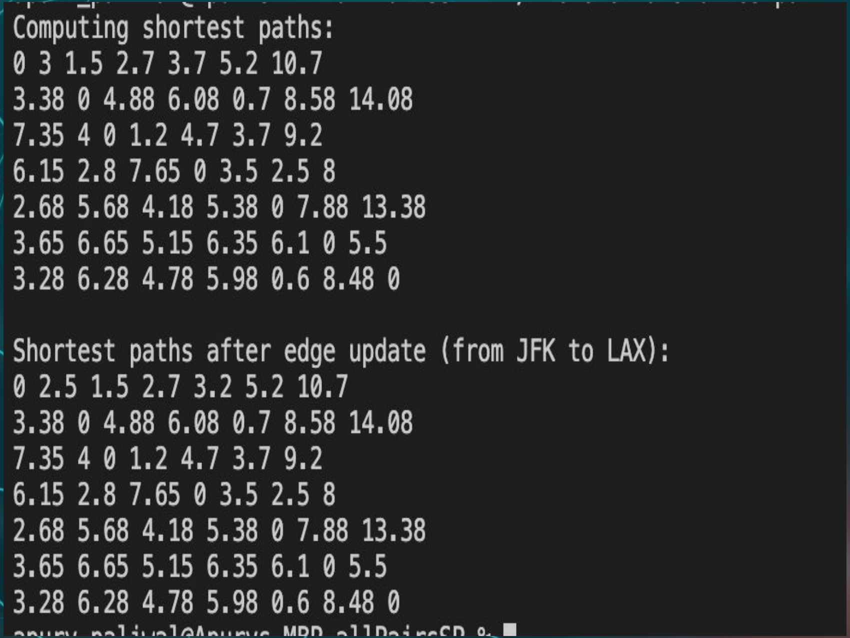
Updating an edge



Let's now update the edge from "JFK" (0) to "LAX" (1) with a new weight of 2.5, and explore how this change propagates through the graph.


$$\begin{tikzcd}
 & & \text{CDG} \arrow[dll, "5.5"'] \arrow[d, "3.65"] & \\
\text{DXB} \arrow[d, "0.6"'] & & \text{JFK} \arrow[r, "1.5"] \arrow[dl, "2.5"'] & \text{ATL} \arrow[d, "1.2"] \\
\text{LHR} \arrow[urr, "2.68"] & \text{LAX} \arrow[l, "0.7"] & & \text{ORD} \arrow[ll, "2.8"]
\end{tikzcd}$$

Example



```
Computing shortest paths:
0 3 1.5 2.7 3.7 5.2 10.7
3.38 0 4.88 6.08 0.7 8.58 14.08
7.35 4 0 1.2 4.7 3.7 9.2
6.15 2.8 7.65 0 3.5 2.5 8
2.68 5.68 4.18 5.38 0 7.88 13.38
3.65 6.65 5.15 6.35 6.1 0 5.5
3.28 6.28 4.78 5.98 0.6 8.48 0

Shortest paths after edge update (from JFK to LAX):
0 2.5 1.5 2.7 3.2 5.2 10.7
3.38 0 4.88 6.08 0.7 8.58 14.08
7.35 4 0 1.2 4.7 3.7 9.2
6.15 2.8 7.65 0 3.5 2.5 8
2.68 5.68 4.18 5.38 0 7.88 13.38
3.65 6.65 5.15 6.35 6.1 0 5.5
3.28 6.28 4.78 5.98 0.6 8.48 0
```

1. Initial Shortest Paths:
   - The first matrix represents the shortest paths between all pairs of nodes, computed using the Floyd-Warshall algorithm.
   - The matrix includes all shortest paths between nodes, with some distances being relatively higher due to indirect paths.
2. Edge Update:
   - The second section shows the shortest paths after an edge update from JFK to LAX, changing the distance to 2.5.
   - Other paths might also show reduced distances, highlighting the propagation effect of the edge update.

Evaluation

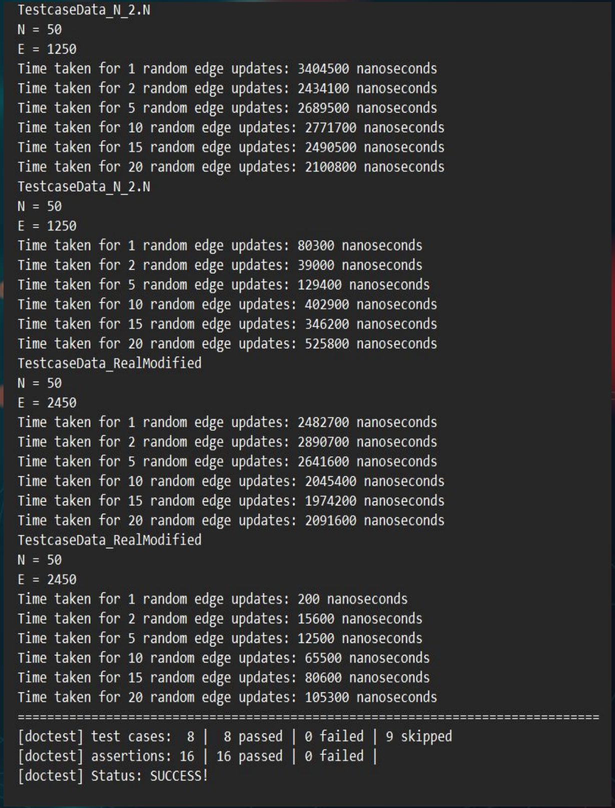
- The algorithm employed in this project can be analyzed based on the time and space complexity
- For the Incremental Edge Update
  - The priority queue operations dominate the time complexity, and each node and edge in the graph may be processed multiple times during the propagation of distance updates.
  - The worst-case scenario occurs when the priority queue processes all potential neighbors for each node, leading to a time complexity of O(N log N + E log N), where N is the number of nodes and E is the number of edges.
  - The space complexity can be given as O(N^2) due to storage of the distance and predecessor arrays, along with the priority queue which has a space complexity of O(N). Hence, the overall complexity is O(N^2)

Evaluation

- For the Incremental Node Insert
  - This method first takes the set of incoming nodes and outgoing nodes to this newly created node and initializes them with the appropriate weights in a $O(N^2)$ operation
  - The major function lies in the variation of Dijkstra's for affected nodes with the help of the priority queue. The worst case time complexity is $O(N^2)$ but it can also go upto $O(N^2 \log N)$ in fully connected graphs
  - Although the worst case complexity is worse for densely connected graphs, it performs better amortized update per node instead of full recomputation
  - The space complexity can be given as $O(N^2)$ due to storage of the distance and predecessor arrays, along with the priority queue which has a space complexity of $O(N)$. Hence, the overall complexity is $O(N^2)$

Experimental Results

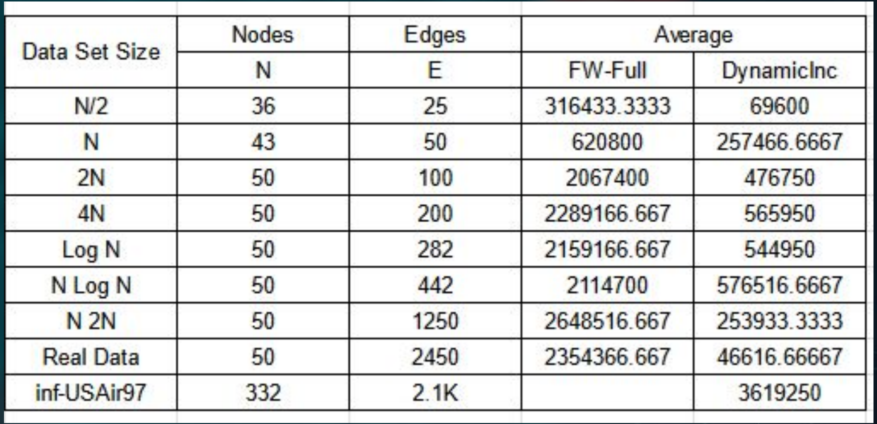
- The experiments to test the algorithms were done on C++ and doctest testing framework on a AMD Ryzen 5 CPU
- The datasets were obtained through public sources and some were synthetically generated for a set of benchmarks as well as real world applications such as the airport flight network graphs
- The datasets used for a preliminary analysis consists of a graph with airport codes and edge lists with the respective cost
- The number of nodes in the test datasets vary from 5 to 300 whereas the number of edges varied from 10 to 25K
- The graphs chosen were mostly sparse graphs as in the real world applications, there are fewer possibilities to see densely populated graphs for aviation networks
- The incremental algorithm has been compared to the Full recomputation of Floyd Warshall algorithm and our dynamic incremental algorithm outperforms it by a huge factor



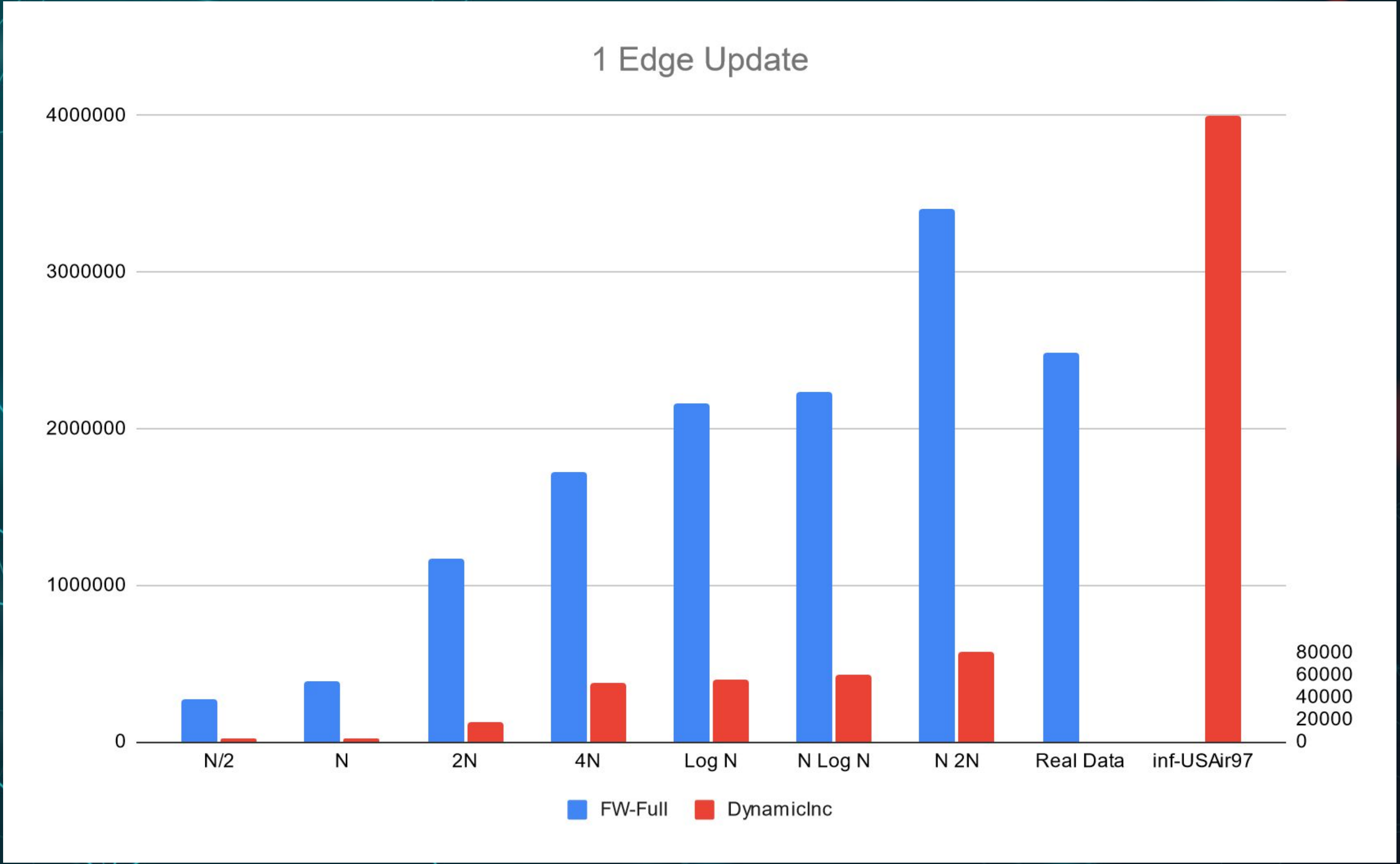
```
TestcaseData_N_2.N
N = 50
E = 1250
Time taken for 1 random edge updates: 3404500 nanoseconds
Time taken for 2 random edge updates: 2434100 nanoseconds
Time taken for 5 random edge updates: 2689500 nanoseconds
Time taken for 10 random edge updates: 2771700 nanoseconds
Time taken for 15 random edge updates: 2490500 nanoseconds
Time taken for 20 random edge updates: 2100800 nanoseconds
TestcaseData_N_2.N
N = 50
E = 1250
Time taken for 1 random edge updates: 80300 nanoseconds
Time taken for 2 random edge updates: 39000 nanoseconds
Time taken for 5 random edge updates: 129400 nanoseconds
Time taken for 10 random edge updates: 402900 nanoseconds
Time taken for 15 random edge updates: 346200 nanoseconds
Time taken for 20 random edge updates: 525800 nanoseconds
TestcaseData_RealModified
N = 50
E = 2450
Time taken for 1 random edge updates: 2482700 nanoseconds
Time taken for 2 random edge updates: 2890700 nanoseconds
Time taken for 5 random edge updates: 2641600 nanoseconds
Time taken for 10 random edge updates: 2045400 nanoseconds
Time taken for 15 random edge updates: 1974200 nanoseconds
Time taken for 20 random edge updates: 2091600 nanoseconds
TestcaseData_RealModified
N = 50
E = 2450
Time taken for 1 random edge updates: 200 nanoseconds
Time taken for 2 random edge updates: 15600 nanoseconds
Time taken for 5 random edge updates: 12500 nanoseconds
Time taken for 10 random edge updates: 65500 nanoseconds
Time taken for 15 random edge updates: 80600 nanoseconds
Time taken for 20 random edge updates: 105300 nanoseconds
===============================================================================
[doctest] test cases:  8 |  8 passed | 0 failed | 9 skipped
[doctest] assertions: 16 | 16 passed | 0 failed |
[doctest] Status: SUCCESS!
```

Experimental Analysis

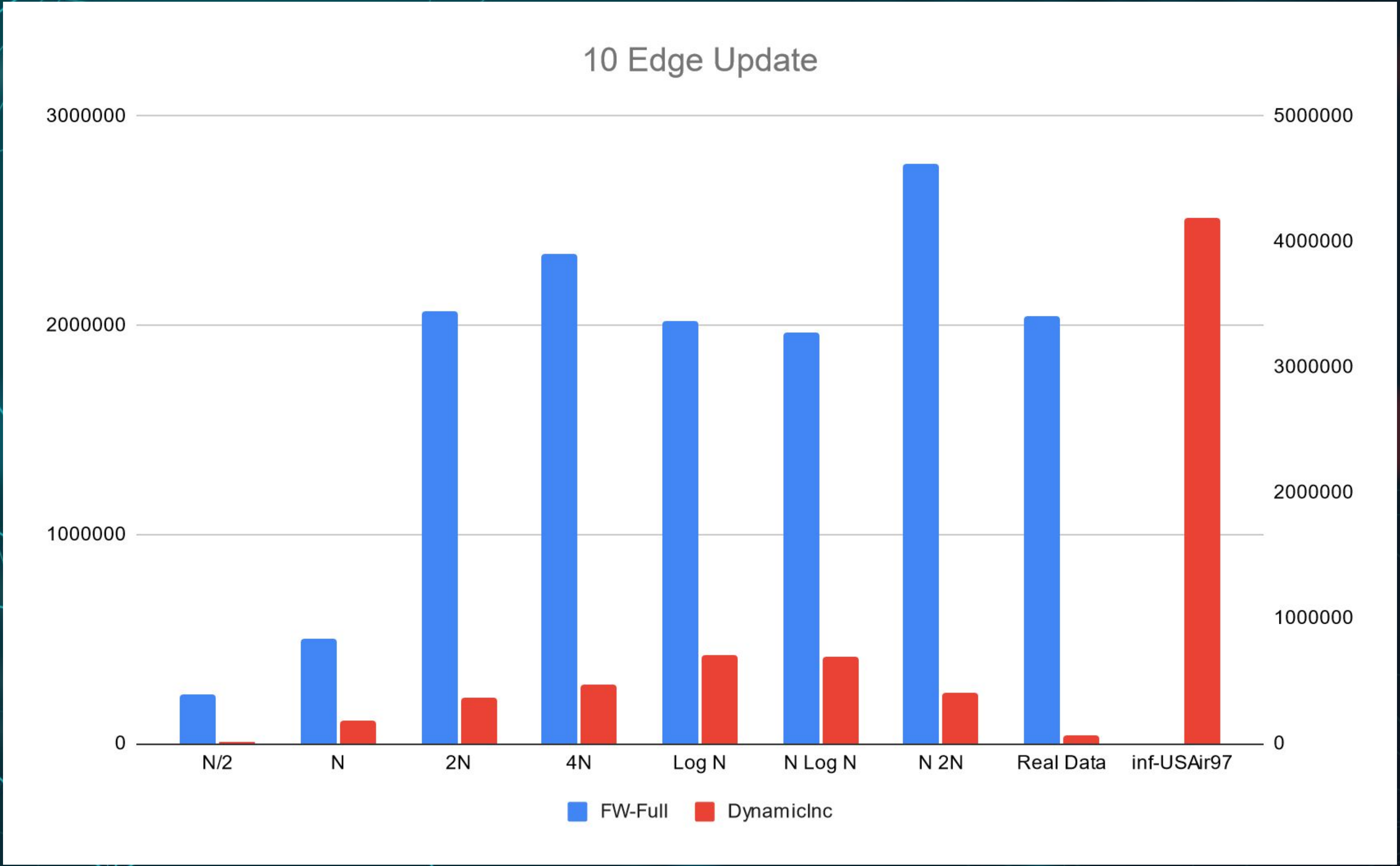
- The node insertions are randomly done on the different graphs and the average over a set of 20 edge/node insertions shows the following runtime in nanoseconds
- The Dynamic Incremental Algorithm significantly outperforms the full recomputation due to the Floyd Warshall by a factor of 10 on average
- This algorithm performs better on dense graphs which is significantly better than other standard algorithms such as RR and QUINCA methods



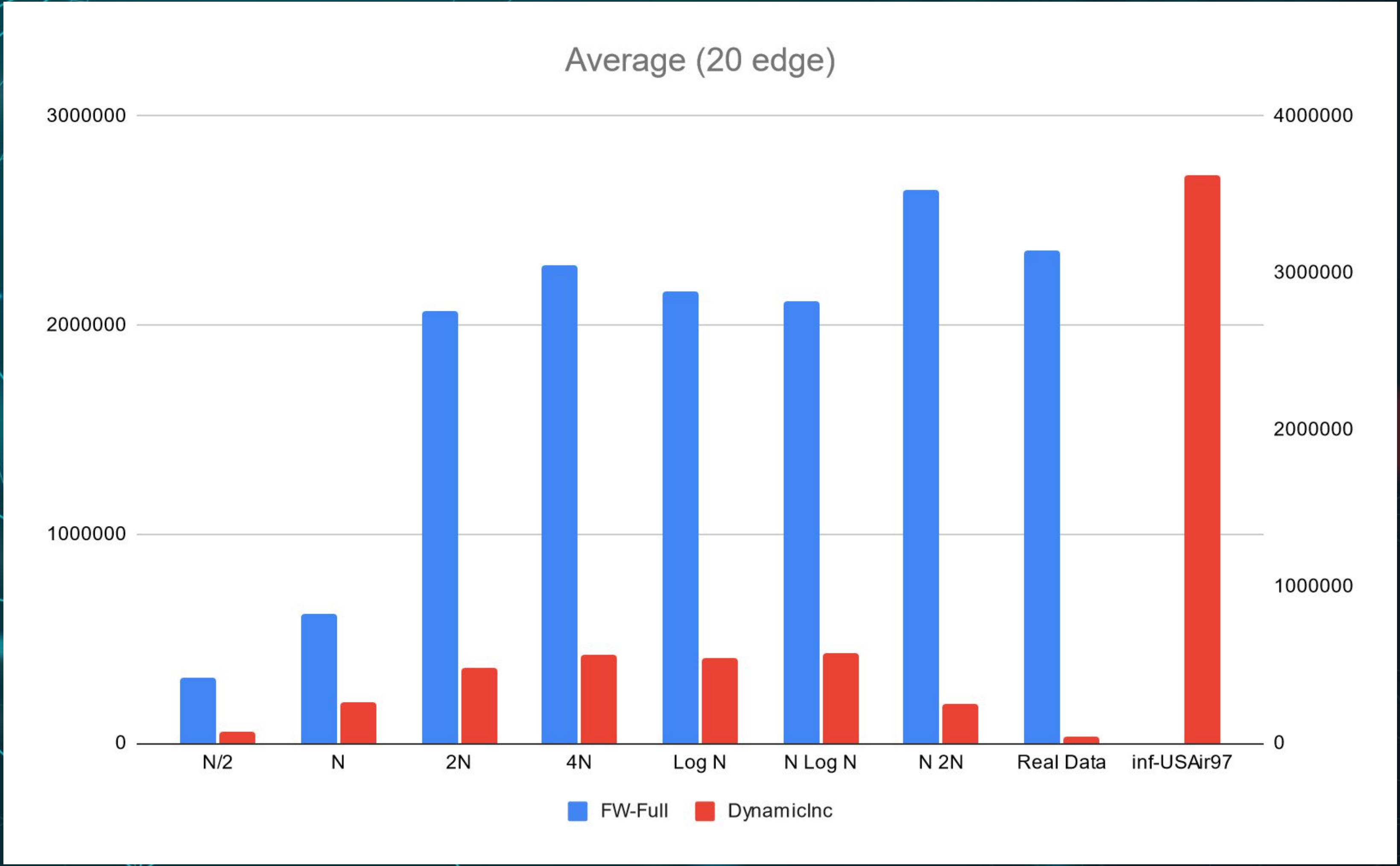
Data Set Size	Nodes	Edges	Average	
	N	E	FW-Full	DynamicInc
N/2	36	25	316433.3333	69600
N	43	50	620800	257466.6667
2N	50	100	2067400	476750
4N	50	200	2289166.667	565950
Log N	50	282	2159166.667	544950
N Log N	50	442	2114700	576516.6667
N 2N	50	1250	2648516.667	253933.3333
Real Data	50	2450	2354366.667	46616.66667
inf-USAir97	332	2.1K		3619250



1 Edge Update
4000000
3000000
2000000
1000000
0
80000
60000
40000
20000
0
N/2
N
2N
4N
Log N
N Log N
N 2N
Real Data
inf-USAir97
FW-Full
DynamicInc



10 Edge Update
3000000
2000000
1000000
0
5000000
4000000
3000000
2000000
1000000
0
N/2
N
2N
4N
Log N
N Log N
N 2N
Real Data
inf-USAir97
FW-Full
DynamicInc



Average (20 edge)
3000000
2000000
1000000
0
4000000
3000000
2000000
1000000
0
N/2
N
2N
4N
Log N
N Log N
N 2N
Real Data
inf-USAir97
FW-Full
DynamicInc

Summary

S

STRENGTHS

Significantly better average performance by a factor of 10 for node updates

W

WEAKNESSES

Performs badly on node insertion for dense graphs and is almost equal to recomputation from scratch

U

USAGE

Uses the principles of truncated Dijkstra for set of affected nodes

A

APPLICATION

Can be applied to infrastructure networks for quick cost computation

Future Work

Adaptive

The graph density is a major factor for most worst case runtimes and hence can be used to switch between different initial algorithms

Redundancies

Despite being fast, further optimizations can be done using affected sources for Node Inserts

Extension

This algorithm only supports incremental updates but further methods would be needed to implement decremental updates

Q&A

References

1. G. Ramalingam and T. Reps, "An Incremental Algorithm for a Generalization of the Shortest-Path Problem," in Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94), 1994, pp. 17-29.
2. G. Ausiello, G. F. Italiano, A. Marchetti Spaccamela, and U. Nanni, "Incremental algorithms for minimal length paths," Journal of Algorithms, vol. 12, no. 4, pp. 615-638, 1991.
3. I. H. Toroslu, "Improving The Floyd-Warshall All Pairs Shortest Paths Algorithm," https://arxiv.org/abs/2109.01872
4. S. Sunita and D. Garg, "Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue," in Journal of King Saud University - Computer and Information Sciences, vol. 13, no. 3, pp 364-373, 2021
5. R. A. Rossi and N. K. Ahmed, "The Network Data Repository with Interactive Graph Analytics and Visualization," in Proceedings of AAAI, 2015. [Online]. Available: https://networkrepository.com
6. A. Slobbe, E. Bergamini, and H. Meyerhenke, "Faster incremental all-pairs shortest paths," Karlsruhe Institute of Technology (KIT), 2016

Thank You

Disclaimer : This algorithm does not calculate the shortest path to success