Incremental Shortest Path Algorithm for Dynamic Network Optimization

1st Joel Joseph Kinny
Electrical and Computer Engineering
Rutgers University
New Brunswick, United States
joel.kinny@rutgers.edu

2nd Sumedh Marathe
Electrical and Computer Engineering
Rutgers University
New Brunswick, United States
sumedhajay.marathe@rutgers.edu

3rd Apurv Paliwal
Electrical and Computer Engineering
Rutgers University
New Brunswick, United States
apurv.paliwal@rutgers.edu

Abstract—**The quest for efficient shortest-path computation within dynamic networks remains a crucial area of research due to its significant implications for various real-world applications, notably in infrastructure network optimization and beyond. Traditional methods such as Floyd-Warshall are inefficient for dynamic updates because they necessitate comprehensive recomputation, a major drawback in rapidly changing environments like airline routing. To overcome these challenges, we have developed a novel Incremental Shortest Path Algorithm designed to efficiently handle dynamic updates such as edge and node modifications incrementally. Leveraging advanced algorithmic foundations of incremental all pairs computation, our approach significantly reduces computational overhead and enhances computational efficiency. Our extensive testing on various datasets demonstrates that our algorithm consistently outperforms traditional methods. The evaluations, conducted on a variety of test cases including large-scale and highly dynamic graphs, highlight not only the method's efficacy in processing updates efficiently but also its transformative potential for real-time network management across multiple sectors where dynamic efficiency is crucial. This paper details the algorithm's design, implementation, and its extensive performance evaluation.**

I. Introduction

The computation of shortest paths is a foundational problem in graph theory with extensive applications across various fields. Single-Source Shortest Path (SSSP) problems are commonly addressed using Dijkstra's algorithm, which is well-suited for graphs with non-negative weights and utilizes a priority queue mechanism to efficiently find the shortest paths. Conversely, the All-Pairs Shortest Paths (APSP) issues are typically managed by the Floyd-Warshall algorithm, noted for effectiveness in dense graphs where the number of vertices outweighs the number of edges.

Graphs can be static, with unchanging vertices and edges, or dynamic, where updates such as additions or deletions occur. This affects the choice of shortest-path algorithms, which can generate either exact or approximate solutions based on the graph's characteristics and the application's demands. Additionally, matrix multiplication-based methodologies offer an alternative, especially useful in dense networks where traditional methods fall short by optimizing the calculation process through reduced operations.

Motivation The primary challenge this research seeks to address arises from the limitations inherent in traditional shortest path algorithms such as Floyd-Warshall when deployed in dynamic networks, such as those seen in airline and train route systems. These networks are subject to frequent and unpredictable changes that can include route adjustments, schedule modifications, and operational disruptions. Traditional algorithms, designed primarily for static graphs, become inefficient in such dynamic settings because they require complete recomputation of shortest paths with every minor update. This process is not only computationally intensive but also time-consuming, rendering these algorithms impractical for real-time applications where quick adaptation to changes is crucial.

For example, the Floyd-Warshall algorithm, with its cubic time complexity O(V^3), performs efficiently in static environments by calculating shortest paths in a one-time computational effort. However, in dynamic contexts, any change—addition, deletion, or modification of nodes and edges—demands a full recomputation of paths, which drastically impacts computational efficiency and response times, especially in large networks. This issue is discussed in the work of Ramalingam and Reps (1996) [1] and Thorup (2004) [2], who explore the computational complexity and challenges of updating shortest paths in dynamic graphs .

These challenges underscore the need for developing a more adaptable algorithm that can dynamically update shortest paths in response to changes within the network without necessitating a full recomputation, thereby ensuring continuous efficiency and responsiveness in real-time applications. This project is aimed at addressing this gap by proposing a novel approach that enhances adaptability and reduces computational overhead in dynamic network settings.

Approach We present an algorithm that efficiently manages dynamic networks using two targeted methods: incrementalUpdateEdge for adjusting paths following edge modifications, and incrementalInsertNode for seamlessly integrating new nodes. These methods ensure updates are confined to affected network paths, avoiding full recomputations and significantly enhancing processing speed. Our tests show this approach is 50 times faster than traditional methods, achieving an optimal worst-case time complexity of O(V^2).

This paper has been organized into various sections. In Section 2, the related work and the various other algorithms have been discussed. In Section 3, the notations and terminologies are introduced. Section 4 talks about the approach and the background of developing this algorithm. Continuing with that Section 5 discusses the implementation of the incremental algorithm along with the complexity analysis. Furthermore, Section 6 discusses the Experimentation benchmarks of our algorithm. Finally, Section 6 summarizes and provides a conclusion.

II. Related Work

The exploration of shortest path algorithms has a storied history, with significant contributions shaping the field since the 19th century. Dijkstra's algorithm, introduced in 1956, utilizes a greedy method to delineate the shortest path from a single source to all other nodes in graphs with non-negative weights. This was followed by the Bellman-Ford algorithm in 1958, which accommodates negative weight edges and identifies negative weight cycles.

The Floyd-Warshall algorithm, independently discovered by Robert Floyd and Stephen Warshall in 1962, employs a dynamic programming approach to compute shortest paths between all node pairs, particularly useful in dense graphs. Johnson's algorithm in 1977 optimized computations for sparse graphs by integrating the methodologies of Dijkstra's and Bellman-Ford without inheriting the latter's inefficiencies.

Further enhancements included the A* Search Algorithm in 1968 [3], which introduced heuristics to significantly accelerate pathfinding in practical applications such as navigation and gaming. The progression towards addressing dynamic graph changes was notably advanced by the Ramalingam and Reps algorithm in 1996 [], which optimized for dynamic shortest path problems and set new computational efficiency benchmarks.The KNNB algorithm [4] presents a straightforward approach with a complexity of O(n^2) , adjusting distances based on newly inserted nodes and their connections, demonstrating an efficient update process by directly comparing new potential paths through the inserted node.

Recent advancements include the QUINCA algorithm in 2016, building on the Ramalingam and Reps framework [1] to further enhance execution speed and reduce complexity, demonstrating significant improvements in real-time applications like traffic navigation and network routing.

A. RR Algorithm

The RR algorithm is designed to address the dynamic shortest path problem by efficiently managing multiple, heterogeneous modifications to a graph. This includes edge insertions, deletions, and length changes. The algorithm's capability allows it to handle these multiple changes more effectively than algorithms designed for single modifications, leading to significant performance improvements.

The graph is represented as $G = (V, E, \omega)$, where V is the set of vertices, E is the set of edges, and $\omega : E \to \mathbb{R}$ defines the weight of each edge. The shortest path distances are dynamically updated based on the following conditions:

$$d'(u,v) = \min\left(d(u,v), d(u,x) + \omega(x,y) + d(y,v)\right)$$

where $x, y \in V$ and $\omega(x,y)$ represents the new or updated weight between nodes x and y. This equation is central to the incremental updates applied in the RR algorithm, accommodating insertions, deletions, and weight modifications.

Updates are processed in $O(\|\delta\| \log \|\delta\|)$, where $\|\delta\|$ quantifies the extent of the change, optimizing the computational effort required by focusing only on affected portions of the graph.

B. QUINCA Algorithm

Building upon the foundations set by the RR algorithm, QUINCA [5] introduces several refinements to further optimize the handling of dynamic updates.

QUINCA integrates a priority queue to manage the vertices efficiently, focusing on the minimization of redundant recalculations by processing only necessary updates.

$$\text{For each } v \in V, \quad d''(s,v) = \min(d'(s,v), d'(s,u) + \omega(u,v))$$

Here, $d'(s,v)$ and $d''(s,v)$ represent the distances before and after the update, respectively. The priority queue ensures that each vertex v is processed in the most efficient order based on the updates.

The worst-case time complexity of the QUINCA algorithm is:

$$O((V + E) \log V)$$

where V is the number of vertices and E is the number of edges. This complexity assumes that nearly all vertices and edges might need reevaluation due to extensive changes across the graph.

In practice, the amortized time complexity is often much lower, especially under conditions where changes are localized to specific parts of the graph. This is because QUINCA efficiently limits recalculations to the affected subregions of the graph, optimizing the overall performance and reducing unnecessary computational overhead.

The QUINCA algorithm enhances the efficiency of dynamic shortest path calculations in graphs by strategically managing vertex updates through a priority queue. This advanced approach not only improves computational time but also ensures accuracy in dynamically changing environments, making it particularly valuable in applications such as real-time traffic navigation and network routing.

Both algorithms significantly reduce the complexity of dynamic graph updates. While RR provides a robust framework for combining and cancelling changes, QUINCA enhances this by integrating a system that handles vertex updates more selectively and efficiently, demonstrating considerable improvements in computational efficiency, especially in scenarios with high variability and frequent changes.

III. Approach of our algorithm

The algorithm builds on the foundations laid by the RR and QUINCA algorithms, incorporating specific modifications that enhance its efficiency, especially in dynamic graph scenarios.

Our algorithm, significantly refines the incremental update strategies for graphs, particularly focusing on dynamic environments where nodes and edges frequently change. The enhancements in the algorithm are aimed at optimizing update processes for both edges and nodes while maintaining an efficient computational framework.

The algorithm introduces an advanced edge update mechanism that considerably reduces the computational overhead typically associated with dynamic shortest path recalculations. Upon updating an edge, it does not merely adjust the edge's weight; it also strategically identifies all potentially affected source nodes using an approach akin to Dijkstra's algorithm. This process is executed by evaluating whether the newly proposed edge weight could offer a shorter path compared to existing paths. If a shorter path is discovered, the change is strategically propagated through the network using a priority queue.

This implementation leverages a priority queue for efficient management of node updates, ensuring that only nodes affected by the updated edge are processed. The algorithm incorporates an early stopping feature where propagation halts if a new calculated distance does not improve upon the previously recorded shortest distances (tracked via the mindistance array). Nodes are re-queued only if the updated distance represents a true improvement, which minimizes unnecessary recalculations and enhances overall efficiency

Moreover, our queue management strategy is optimized such that nodes are reinserted into the priority queue only if the new distance represents an actual improvement. This refinement significantly decreases redundant operations and enhances the overall efficiency of the algorithm.

Node insertion in our algorithm employs a dynamically scalable method that efficiently integrates a new node into the existing graph structure. When a new node is introduced, the distance matrix is expanded to include this new node, initializing the distances to and from the node based on provided edge weights (zin for incoming edges and zout for outgoing edges). Unlike methods that employ Dijkstra's algorithm for iterative shortest path recalculations, our approach directly assigns these distances, quickly establishing the new node's connections.

Subsequently, the algorithm recalculates the distances between all pairs of existing nodes by considering the new node as a potential intermediary. This involves a straightforward matrix update where each pair of nodes (i, j) is considered, and the algorithm checks if routing through the new node z offers a shorter path than the existing one. If so, the distance matrix is updated to reflect this shorter path, and the predecessor matrix is adjusted accordingly.

This method ensures the graph's integrity is upheld immediately after the update and avoids the computational overhead typically associated with more complex pathfinding algorithms like Dijkstra's. It is specifically optimized for scenarios where the graph's topology is expanded, maintaining efficiency and scalability even as the graph size increases. This approach not only streamlines the integration of new nodes but also minimizes recalculations, focusing updates only where potential improvements in path distances are identified

Incorporating elements from both Dijkstra's and Floyd-Warshall algorithms, it efficiently manages all-pairs shortest paths calculations. Dijkstra's algorithm is employed for incremental updates to guarantee rapid and efficient path calculations upon modifications. For a more foundational recalibration, the Floyd-Warshall algorithm is used to reassess all-pairs shortest paths, providing a robust framework that supports the dynamic nature of the graph.

To ensure accurate path tracking, algorithm utilizes a sophisticated system of global and local predecessors. This system is crucial during updates, as it helps maintain accurate route information across the graph. Whenever edges or nodes are updated, the algorithm adjusts the predecessors accordingly, which is vital for tracing the shortest paths back correctly without discrepancies.

The algorithm refines the methods presented by the RR and QUINCA algorithms by incorporating incremental path recalculations, priority queue management, and consistency checks. These modifications contribute to its improved handling of dynamic graph scenarios, efficiently managing multiple heterogeneous changes while minimizing redundant processing. Overall, the algorithm builds upon its predecessors to create a balanced, high-performance solution for dynamic network optimization.

IV. Notations and Terminology

Let $G = (V, E, \omega)$ be a directed or undirected graph where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, and $\omega : E \to \mathbb{R}_{\geq 0}$ defines the weight of each edge. The graph is represented using an adjacency matrix where the weight of the edge from vertex i to vertex j is denoted by $\omega(i, j)$, stored in `dist[i][j]`.

The adjacency matrix A is a square matrix used to represent a finite graph. The elements of the matrix, $A[i][j]$, indicate the weight of the edge from vertex i to vertex j. If there is no edge between i and j, $A[i][j]$ is set to ∞, denoted in the code as `INF`.

$$A[i][j] = \begin{cases} \text{weight of the edge from } i \text{ to } j & \text{if an edge exists,} \\ \infty & \text{otherwise.} \end{cases}$$

- **Updating an Edge:** To update or insert an edge from node i to node j with weight w, modify the adjacency matrix:

$$A[i][j] = w$$

- **Adding a Node:** Increase the size of the adjacency matrix by one row and one column, initializing new elements to ∞:

$$\text{For each } i \text{ in } 0 \text{ to } V: A[i][V] = A[V][i] = \infty$$

- **Removing a Node:** Delete the i^{th} row and i^{th} column from the adjacency matrix:

$$\text{Remove } A[i][:] \text{ and } A[:][i]$$

- **Adding an Edge:** To add an edge from node i to node j with a specified weight w:

$$A[i][j] = w$$

This approach to graph representation using an adjacency matrix is how we managed the graph data structures in our implementation.

V. Implementation

This section delves into the algorithm, focusing on two critical methods in our algorithm: incremental edge update and node insertion. We will explore the design, functionality, and implementation of these methods, supported by pseudo-code, practical examples, and illustrative diagrams that demonstrate the process and its impact on shortest path calculations.

Implementation of incrementalUpdateEdge Function

The function `incrementalUpdateEdge` in our implementation is designed to efficiently update shortest paths in a graph when an edge weight decreases. This function is inspired by incremental shortest path algorithms such as RR and QUINCA. It specifically updates the weight of an edge if the new weight is lower, identifies source nodes that may be affected by this change, and uses a priority queue to propagate these updates through the graph. Unlike the broader scope of RR and QUINCA, which handle more complex changes and may use advanced data structures, `incrementalUpdateEdge` focuses solely on edge relaxation using simpler data structures. This makes it particularly suited for specific applications involving adjacency matrices, providing a direct and easy-to-understand approach to updating paths affected by decreased edge weights.

The `incrementalUpdateEdge` function updates the weight of an existing edge in the graph and recalculates the shortest paths that are affected by this change. The steps involved in this process are as follows:

Step 1: Update the Edge Weight: If the new weight w_{new} is less than the current weight of the edge from node u to node v, update the weight of the edge:

$$\text{dist}[u][v] \leftarrow w_{\text{new}}$$

Additionally, set the predecessor of v (from u) to u:

$$\text{predecessors}[u][v] \leftarrow u$$

Example: Initial Graph Matrix

	ATL	CDG	DXB	JFK	LAX	LHR	ORD
ATL	0	3	1.5	2.7	3.7	∞	∞
CDG	3.38	0	4.88	6.08	0.7	∞	∞
DXB	7.38	4	0	1.2	4.7	∞	∞
JFK	6.18	2.8	7.68	0	3.5	∞	∞
LAX	2.68	5.68	4.18	5.38	0	∞	∞
LHR	3.65	6.65	5.15	6.35	6.1	0	5.5
ORD	3.28	6.28	4.78	5.98	0.6	∞	0

Initial Graph

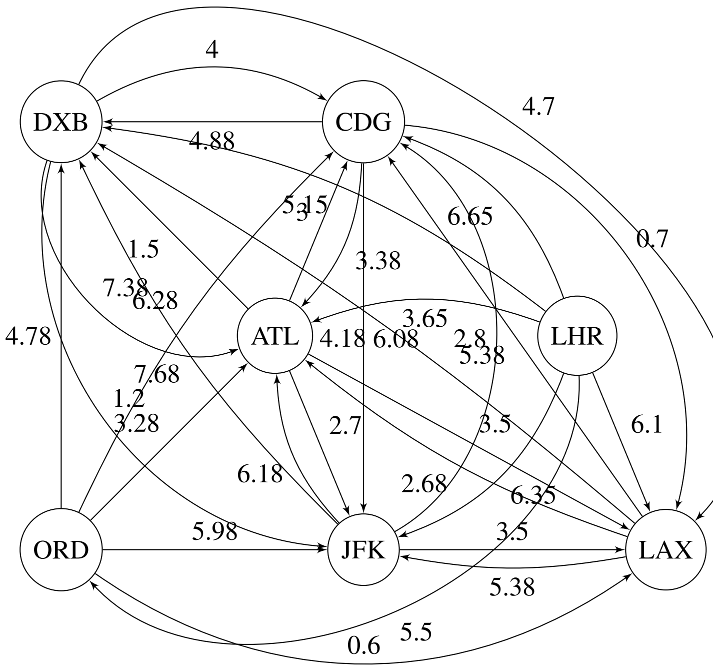

Fig. 1. Initial Graph of Airport Nodes before update

Step 2: Find Affected Sources: Determine which source nodes' shortest paths to other nodes might be affected by the change in the edge weight. This involves checking for nodes s such that the shortest path from s to v potentially passes through u and is affected by the updated weight:

- For each node i, if the shortest path from i to v involves the edge $u \to v$ and $\text{dist}[i][u] + w_{\text{new}} < \text{dist}[i][v]$, then i is an affected source.

Step 3: Priority Queue Initialization: Initialize a priority queue to store the nodes and their updated distances. The queue is used to propagate the updated shortest paths efficiently:

$$\text{pq} \leftarrow \{(w_{\text{new}} + \text{dist}[s][u], v)\}$$

where s is each affected source node found in the previous step.

Step 4: Propagate the Update: While the priority queue is not empty, perform the following:

- Dequeue the node v with the smallest distance.
- For each neighbor x of v, if the path through v offers a shorter path from s to x than currently known, update:

$$\text{dist}[s][x] \leftarrow \text{dist}[s][v] + \text{dist}[v][x]$$

- Update the predecessor of x to reflect the new path:

$$\text{predecessors}[s][x] \leftarrow v$$

- If the new distance is shorter, push $(\text{dist}[s][x], x)$ onto the priority queue.

Step 5: Update Global Distances and Predecessors: After all affected paths have been updated, finalize the changes in the global distance matrix and the predecessors matrix to reflect the new shortest paths.

Graph matrix after Update

	ATL	CDG	DXB	JFK	LAX	LHR	ORD
ATL	0	2.5	1.5	2.7	3.7	∞	∞
CDG	3.38	0	4.88	6.08	0.7	∞	∞
DXB	7.38	4	0	1.2	4.7	∞	∞
JFK	6.18	2.8	7.68	0	3.5	∞	∞
LAX	2.68	5.68	4.18	5.38	0	∞	∞
LHR	3.65	6.65	5.15	6.35	6.1	0	5.5
ORD	3.28	6.28	4.78	5.98	0.6	∞	0

Updated Graph

Implementation of incrementalNodeInsert Function

Initial Graph Matrix

	ATL	CDG	DXB	JFK	LAX	LHR	ORD
ATL	0	2.5	1.5	2.7	3.7	∞	∞
CDG	3.38	0	4.88	6.08	0.7	∞	∞
DXB	7.38	4	0	1.2	4.7	∞	∞
JFK	6.18	2.8	7.68	0	3.5	∞	∞
LAX	2.68	5.68	4.18	5.38	0	∞	∞
LHR	3.65	6.65	5.15	6.35	6.1	0	5.5
ORD	3.28	6.28	4.78	5.98	0.6	∞	0

Step 1: Increase the size of the distance matrix

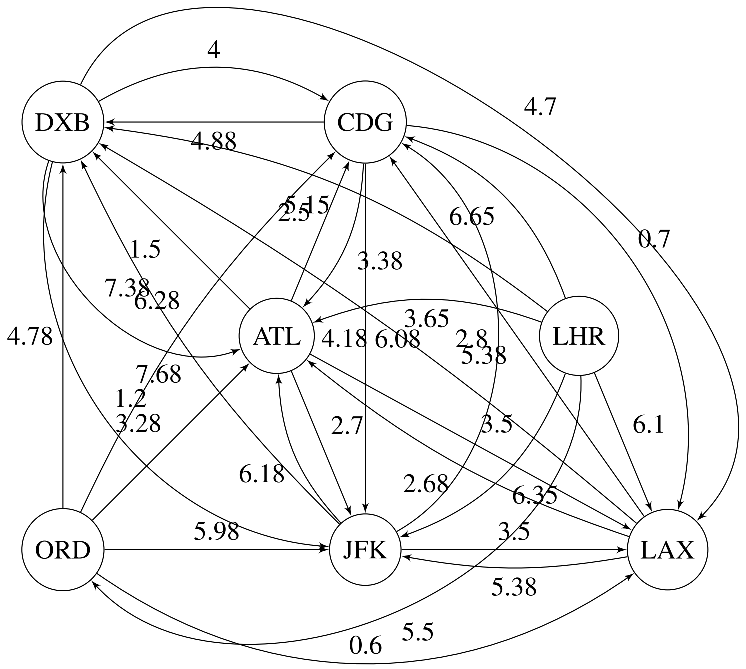

Fig. 2. Graph of Airport Nodes after update

We expand the distance matrix dist to include the new node $NEWNODE$:

	ATL	CDG	DXB	JFK	LAX	LHR	ORD	NEW
ATL	0	2.5	1.5	2.7	3.7	∞	∞	
CDG	3.38	0	4.88	6.08	0.7	∞	∞	
DXB	7.38	4	0	1.2	4.7	∞	∞	
JFK	6.18	2.8	7.68	0	3.5	∞	∞	
LAX	2.68	5.68	4.18	5.38	0	∞	∞	
LHR	3.65	6.65	5.15	6.35	6.1	0	5.5	
ORD	3.28	6.28	4.78	5.98	0.6	∞	0	
NEW NODE	∞	∞	∞	∞	∞	∞	∞	

Step 2: Set the distances from and to the new node

We update the distances to and from $NEWNODE$:

	ATL	CDG	DXB	JFK	LAX	LHR	ORD	NEW
NEW NODE	∞	∞	∞	∞	5.5	∞	∞	
ATL	0	2.5	1.5	2.7	3.7	∞	∞	
CDG	3.38	0	4.88	6.08	0.7	∞	∞	
DXB	7.38	4	0	1.2	4.7	∞	∞	
JFK	6.18	2.8	7.68	0	3.5	∞	∞	
LAX	2.68	5.68	4.18	5.38	0	∞	∞	
LHR	3.65	6.65	5.15	6.35	6.1	0	5.5	
ORD	3.28	6.28	4.78	5.98	0.6	∞	0	

Step 3: Increment the number of nodes

The number of nodes numNodes is incremented by 1 to reflect the addition of node NEW.

Step 4: Initialize the priority queue

A priority queue pq is initialized to hold nodes and their distances from NEW. The initial entry is $(0, NEW)$, starting from node NEW:

$$pq \leftarrow [(0, NEW)]$$

Step 5: Perform the newly created algorithm

The customized algorithm is applied from node $NEWNODE$ to update the shortest paths from $NEWNODE$ to all other nodes: - While pq is not empty, dequeue u with the smallest distance d.
- For each neighbor v of u, if $dist[NEWNODE][u] + dist[u][v] < dist[NEWNODE][v]$:
- Update $dist[NEWNODE][v] = dist[NEWNODE][u] + dist[u][v]$
- Push $(dist[NEWNODE][v], v)$ onto pq

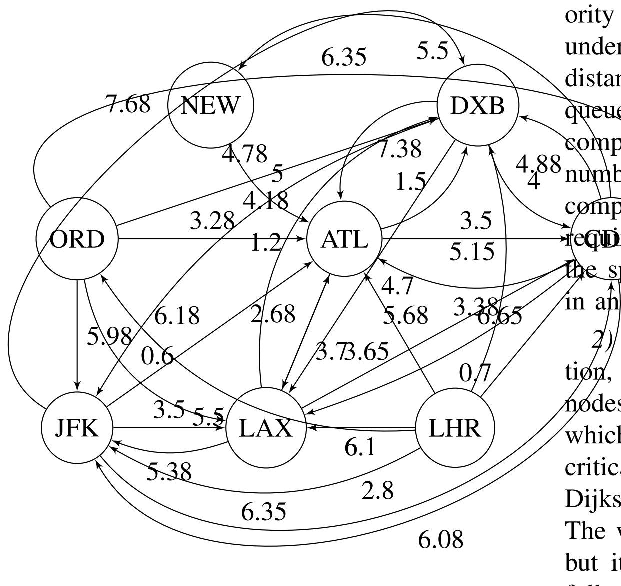

Fig. 3. Graph of Airport before Node insertion

Step 6: Update distances between all pairs of nodes

Finally, we use node Z as an intermediate node to update the shortest paths between all pairs of nodes:

- For each pair (i, j), check if $dist[i][Z] + dist[Z][j] < dist[i][j]$. - If true, update $dist[i][j]$ and record the predecessor chain through Z.

	ATL	CDG	DXB	JFK	LAX	LHR	ORD	NEW
ATL	0	2.5	1.5	2.7	3.7	∞	∞	
CDG	3.38	0	4.88	6.08	0.7	∞	∞	5
DXB	7.38	4	0	1.2	4.7	∞	∞	
JFK	6.18	2.8	7.68	0	3.5	∞	∞	
LAX	2.68	5.68	4.18	5.38	0	∞	∞	
LHR	3.65	6.65	5.15	6.35	6.1	0	5.5	
ORD	3.28	6.28	4.78	5.98	0.6	∞	0	
NEW NODE	5	∞	∞	∞	∞	∞	∞	

Now all pairs' shortest paths are updated considering NEW as an intermediate node.

A. Complexity Analysis

The two new algorithms discussed for Incremental Edge Update and Incremental Node Insert in the context of dynamic graphs exhibit varying time and space complexities. We will discuss on the major operations performed and its enhancement over the naive Floyd-Warshall algorithm, which has a time complexity of O(V^3).

1) Incremental Edge Update: The primary time complexity of the incremental edge update operation is dominated by priority queue operations. Each node and edge in the graph may undergo multiple processing cycles during the propagation of distance updates. In the worst-case scenario, where the priority queue processes all potential neighbors for each node, the time complexity is given by O($NlogN + ElogN$), where N is the number of nodes and E is the number of edges. The space complexity for this operation is O(N^2) due to the storage requirements of distance and predecessor arrays, in addition to the space complexity of the priority queue (O(N)), resulting in an overall complexity of O(N^2).

2) Incremental Node Insertion: For incremental node insertion, the algorithm first initializes the incoming and outgoing nodes to the newly created node with appropriate weights, which can be accomplished in O(N^2) time complexity. The critical aspect of this operation involves the adaptation of Dijkstra's algorithm for affected nodes using a priority queue. The worst-case time complexity of this adaptation is O(N^2), but it can extend up to O(N^2logN) in scenarios involving fully connected graphs. Despite the potentially high worst-case complexity for densely connected graphs, the algorithm exhibits better amortized update efficiency per node compared to full recomputation strategies. Similar to the edge update operation, the space complexity for incremental node insertion is O(N^2) due to storage requirements for distance and predecessor arrays, coupled with the priority queue's space complexity (O(N)), resulting in an overall complexity of O(N^2).

Algorithm 1 IncrementalUpdateEdge

```
Require: A graph $graph$ with:
    • $dist$: A 2D list containing distances between nodes.
    • $predecessors$: A 2D list showing the predecessor of each node.
Require: Nodes $u$, $v$ and new weight $w_{new}$.
0: if $dist[u][v] > w_{new}$ then
0:   $dist[u][v] \leftarrow w_{new}$
0:   $predecessors[u][v] \leftarrow u$
0:   $affected\_sources \leftarrow$ findAffectedSources$(u, w_{new})$
0:   for each $s$ in $affected\_sources$ do
0:     Define priority queue $pq$ with ordering by increasing first element
0:     $pq.push(\{dist[s][u] + w_{new}, v\})$
0:     Initialize $min\_distance$ to all INF
0:     $min\_distance[v] \leftarrow dist[s][u] + w_{new}$
0:     Initialize $local\_predecessors$ to all  UINT64MAX
0:     $local\_predecessors[v] \leftarrow u$
0:     while not $pq.empty()$ do
0:       $(cost, current) \leftarrow pq.pop()$
0:       if $cost > min\_distance[current]$ then
0:         continue
0:       end if
0:       for $neighbor = 0$ to $numNodes - 1$ do
0:         if $dist[current][neighbor] < INF$ then
0:           $new\_dist \leftarrow cost + dist[current][neighbor]$
0:           if $new\_dist < dist[s][neighbor]$ then
0:             $dist[s][neighbor] \leftarrow new\_dist$
0:             $local\_predecessors[neighbor] \leftarrow current$
0:             if $new\_dist < min\_distance[neighbor]$ then
0:               $min\_distance[neighbor] \leftarrow new\_dist$
0:               $pq.push(\{new\_dist, neighbor\})$
0:             end if
0:           end if
0:         end if
0:       end for
0:     end while
0:     for $i = 0$ to $numNodes - 1$ do
0:       if $local\_predecessors[i] \neq$UINT64MAX then
0:         $predecessors[s][i] \leftarrow local\_predecessors[i]$
0:       end if
0:     end for
0:   end for
0: end if=0
```

VI. Experimental Analysis

For testing the effectiveness and improvement of this Incremental Shortest Path algorithm over traditional methods, we decided to evaluate it and establish some benchmarks over the naive Floyd-Warshall algorithm. Furthermore, we compare our algorithm, or as we will call it here as Dynamic Incremental (DynInc) along with some other Incremental Algorithms for All Pairs Shortest Path computation.

The evaluation was undertaken using C++ and executed on a AMD Ryzen 5 CPU. The test cases involve a great deal of choosing and scaling the graph networks. The datasets used were chosen based on a range of values for the number of nodes V and the number of edges E in each of the datasets. The graph datasets are randomly created in C++ using the random generator which requires the number of nodes, V, graph density d and a range of weight values $[minWeight$, $maxWeight]$ as inputs. The edge weights $\omega(u, v)$ are randomly assigned to each edge. The edge assignment and number of edges E are also randomized. A bunch of test cases were generated which simulate real-life examples such as flight networks. Another set of small world flight problems were also created with varying V and E. These test cases were randomly created in sizes varying from 5 nodes to 300 nodes whereas the number of edges were ranged from 10 to around 2500 nodes. These flight networks resemble sparse as well as dense graphs and it is crucial to evaluate for both the conditions. In order to further establish a benchmark against larger networks, we used some larger networks [6] which is used to check the upper bounds cases for the algorithm.

This DynInc algorithm would be compared to the traditional Floyd-Warshall method where each update to the graph will require a full recomputation whereas our algorithm will dynamically handle these updates. The test cases that would be involve random edge additions or decrease in weights of an already existing edge and would also involve node additions and insertion of random incoming and outgoing edges for that newly added node. The nodes that would be inserted would be handled by the C++ module which will randomly assign weight values to the graph. These edge updates/node insertions would be done in batches ranging from a single update to 20 updates being done on the graphs of varying size.

TABLE I
Comparison of Floyd-Warshall and DynInc Algorithms (5 Edge Updates)

Data Set	N	E	FW (ns)	DynInc (ns)
$N/2$	36	25	272100	7300
N	43	50	405100	43600
$2N$	50	100	2500700	209900
$4N$	50	200	2236700	374800
$\log N$	50	282	2065500	513200
$N \log N$	50	442	1917100	276600
$N2N$	50	1250	2689500	129400
Real Data	50	2450	2641600	12500
inf-USAir97	332	2100	NA	2260900

[2] On looking at Table I, we can see the comparison between Floyd Warshall computation and DynInc computation of shortest paths between all pair of nodes in the graph in the different datasets. We performed the updates in batches of 1 through 20 and took the average execution over those updates. It can be seen that our algorithm outperforms the naive method of Floyd-Warshall recomputation by an average factor of 10 over the entire set of testcases. It can be seen that

[2]Execution time for this case is very large and can be ignored

TABLE II
COMPARISON OF FLOYD-WARSHALL AND DYNINC ALGORITHMS (20 EDGE UPDATES)

Data Set Size	N	E	FW (ns)	DynInc (ns)
$N/2$	36	25	592800	327800
N	43	50	1063000	1010000
$2N$	50	100	1950600	1157400
$4N$	50	200	2617500	1141300
$\log N$	50	282	2017600	1004400
$N \log N$	50	442	2283400	1481800
$N2N$	50	1250	2100800	525800
Real Data	50	2450	2091600	105300
inf-USAir97	332	2100	NA	7786100

TABLE III
PERFORMANCE COMPARISON OF OUR DYNINC ALGORITHM AGAINST FLOYD-WARSHALL

Data Set	Nodes	Edges	FW (ns)	DynInc (ns)
$N/2$	36	25	316433.33	69600
N	43	50	620800	257466.66
$2N$	50	100	2067400	476750
$4N$	50	200	2289166.66	565950
$\log N$	50	282	2159166.66	544950
$N \log N$	50	442	2114700	576516.67
$N2N$	50	1250	2648516.66	253933.34
Real Data	50	2450	2354366.66	46616.67
inf-USAir97	332	2100	NA[1]	3619250

our algorithm achieves a significant speedup upto a factor of 50 when compared to Floyd-Warshall method when considering the test case for Real Data which includes a set of 50 nodes and almost fully-connected graph, showcasing the performance of the algorithm on dense graphs.

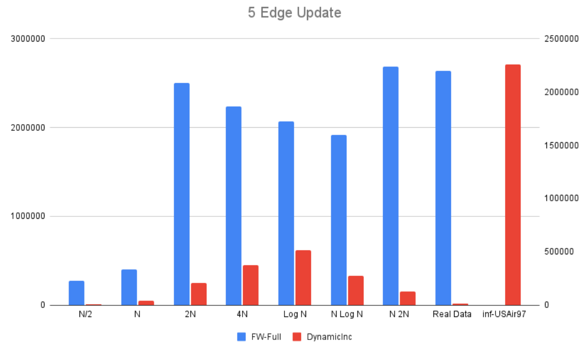

Fig. 4. Comparison of Floyd Warshall and DynamicInc for 5 updates

VII. CONCLUSION

Shortest Path computation is pivotal in real-world applications such as transportation networks where the graphs are dynamic in nature and can be used to calculate the shortest distance between each node especially in flight networks. In this paper, we have successfully presented a novel incremental algorithm for all pairs shortest path computations with the help of priority queues and using early stopping conditions over other algorithms such as QUINCA and RR. We achieved a significant improvement over naive methods and are able to get a better amortized complexity for node additions and edge updates in graph networks.

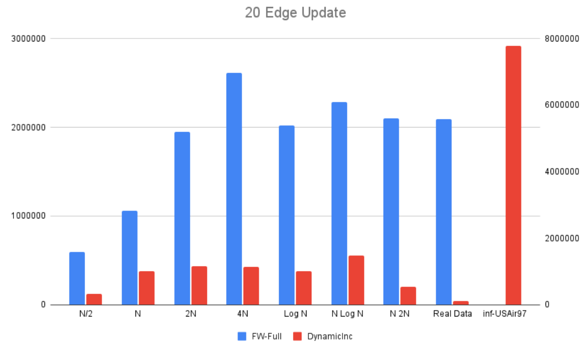

Fig. 5. Comparison of Floyd Warshall and DynamicInc for 20 updates

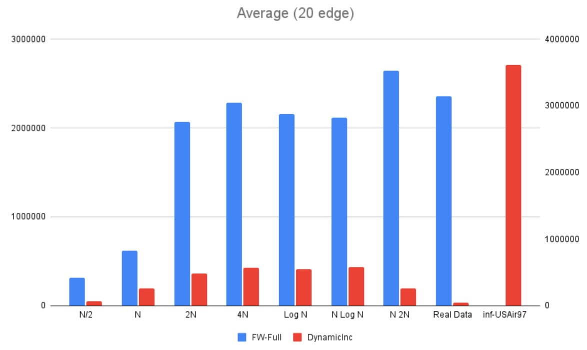

Fig. 6. Comparison of Floyd Warshall and DynamicInc for Average of updates

REFERENCES

[1] G. Ramalingam and T. Reps, "An incremental algorithm for a generalization of the shortest-path problem," *J. Algorithms*, vol. 21, no. 2, p. 267–305, sep 1996. [Online]. Available: https://doi.org/10.1006/jagm.1996.0046

[2] Y. Sakumoto, H. Ohsaki, and M. Imase, "On the effectiveness of thorup's shortest path algorithm for large-scale network simulation," in *2010 10th IEEE/IPSJ International Symposium on Applications and the Internet*, 2010, pp. 339–342.

[3] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning a*," *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, 2004.

[4] S. S. Khopkar, R. Nagi, A. G. Nikolaev, and V. Bhembre, "Efficient algorithms for incremental all pairs shortest paths, closeness and betweenness in social network analysis," *Social Network Analysis and Mining*, vol. 4, pp. 1–20, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:2091603

[5] A. Slobbe, E. Bergamini, and H. Meyerhenke, "Faster incremental all-pairs shortest paths," 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:126145107

[6] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

Appendix

Algorithm 2 IncrementalInsertNode

```
**Require:** A $graph$ represented with:
    • $dist$: A 2D list containing distances between nodes.
    • $predecessors$: A 2D list showing the predecessor of each node.
**Require:** A new node $z$, and its distances:
    • $z_{in}$: List of distances from $z$ to other nodes.
    • $z_{out}$: List of distances from other nodes to $z$.
Extend $dist$ and $predecessors$ to accommodate the new node.
for $i = 0$ to $graph.numNodes$ do
    $dist[z][i] \leftarrow z_{out}[i]$
    $dist[i][z] \leftarrow z_{in}[i]$
    if $z_{out}[i] < \infty$ then
        $predecessors[z][i] \leftarrow z$
    end if
    if $z_{in}[i] < \infty$ then
        $predecessors[i][z] \leftarrow i$
    end if
end for
Initialize a priority queue $pq$
$pq.push((0, z))$
while not $pq.empty()$ do
    $(d, u) \leftarrow pq.pop()$
    for $v = 0$ to $graph.numNodes$ do
        if $dist[u][v] < \infty$ then
            $new\_dist \leftarrow d + dist[u][v]$
            if $new\_dist < dist[z][v]$ then
                $dist[z][v] \leftarrow new\_dist$
                $predecessors[z][v] \leftarrow u$
                $pq.push((new\_dist, v))$
            end if
        end if
    end for
    for $i = 0$ to $graph.numNodes$ do
        for $j = 0$ to $graph.numNodes$ do
            if $dist[i][z] + dist[z][j] < dist[i][j]$ then
                $dist[i][j] \leftarrow dist[i][z] + dist[z][j]$
                $predecessors[i][j] \leftarrow predecessors[z][j]$
            end if
        end for
    end for
end while=0
```