All Pair Shortest Path Algorithms on Dynamic Graph Setting

Beomseok Park
Electrical and Computer Engineering
Rutgers University - New Brunswick
Piscataway, USA
bp593@scarletmail.rutgers.edu

Suhas Kollur
Electrical and Computer Engineering
Rutgers University - New Brunswick
Piscataway, USA
sk2870@scarletmail.rutgers.edu

Shuyuan Fan
Electrical and Computer Engineering
Rutgers University - New Brunswick
Piscataway, USA
sf850@scarletmail.rutgers.edu

***Abstract*—We study three important incremental shortest path algorithms, which are designed for dynamic updates to graph. We implemented naive Floyd-Warshall algorithm as the baseline, and implemented three dynamic algorithms: RR algorithm, QUINCA algorithm, and LDSP algorithm. We also benchmarked those three algorithms: The RR algorithm and QUINCA algorithm are fast. The LDSP algorithm, which supports fully-dynamic updates, underperforms those two algorithms, but still faster than the naive Floyd-Warshall algorithm. Our work shows that incremental shortest path algorithms not only have low complexity in theory, but also run fast in practice.**

***Index Terms*—Algorithms, Discrete Mathematics, Graph Theory, Shortest Path**

I. INTRODUCTION

Recent surges in research on shortest-path algorithms stem from its wide-ranging applications in areas such as network routing protocols, route planning, traffic control, social network path-finding, and transportation systems. Computer scientists and graph theorists define the shortest-path problem as one that identifies the path with the minimum length from a source to a destination.

The shortest-path problem is typically divided into the single-source shortest-path (SSSP) and the all-pairs-shortest paths (APSP). The SSSP aims to find shortest path between a given vertex and every other vertex in the graph. The APSP is solved by seeking all the shortest paths between every pair of vertices. These two algorithms work efficiently only in a static graph where vertices and edges are constant. However, in real-world scenarios, graph problems are extended to dynamic graph settings where vertices and edges change over time. For example, airline routes are dynamic in the real world, with changes in price, origin, and destination due to various factors.

In the realm of dynamic graphs, the complexity of managing changing vertices and edges presents unique challenges, leading to the categorization into incremental, decremental, and fully-dynamic graphs. An incremental graph is one where edges are added over time, but none are removed (i.e., edge insertion by weight decrease). This incrementally changes the topology of the graph and requires algorithms that can efficiently update the shortest paths without recalculating from scratch. Conversely, a decremental graph involves the removal of edges, necessitating updates to existing shortest paths that may no longer be valid due to these deletions. A fully dynamic graph involves both incremental and decremental graphs. Our project focuses on solving the incremental all-pairs shortest path (APSP) problem, where we aim to continuously update the shortest paths between all pairs of vertices as new edges and vertices are added. This is crucial for applications such as continually updating routing information in networks or traffic systems where new routes are frequently established.

II. METHODOLOGY

A. Floyd Warshall's Algorithm

The objective of this study is to implement the Floyd-Warshall algorithm for computing shortest paths in a transportation network represented by fictitious transportation costs. By achieving this objective, we aim to explore the practical applications of graph algorithms in transportation optimization.

The Floyd-Warshall algorithm is a classic algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. By implementing this algorithm for transportation costs, we aim to improve route planning, enhance resource allocation, and optimize overall network efficiency.

The implementation of the Floyd-Warshall algorithm on transportation costs involves applying a dynamic programming approach to find the shortest paths between all pairs of vertices in a weighted graph representing a transportation network. This algorithm has significant applications in optimizing transportation routes, minimizing costs, and improving efficiency in various industries, including logistics, public transportation, and supply chain management.

1) Floyd-Warshall Algorithm Overview: The Floyd-Warshall algorithm, developed by Robert Floyd and Stephen Warshall, is a classic algorithm used for solving the all-pairs shortest path problem in weighted graphs. It works efficiently for both dense and sparse graphs, with a time complexity of $O(V^3)$, where V is the number of vertices in the graph. The algorithm iterates through all pairs of vertices and computes the shortest path between them by considering all possible intermediate vertices.

Algorithm 1 Floyd-Warshall Algorithm for Transportation Costs

```
Input: $V$: Number of vertices in the transportation network.
Input: $E$: Number of edges in the transportation network.
Input: $graph$: Adjacency matrix representing the transportation network.
Output: $distance$: Matrix containing shortest distances between all pairs of vertices.
 1: Initialize $distance$ matrix as $graph$.
 2: for $k = 1$ to $V$ do
 3:     for $i = 1$ to $V$ do
 4:         for $j = 1$ to $V$ do
 5:             if $distance[i][k] + distance[k][j] < distance[i][j]$ then
 6:                 $distance[i][j] \leftarrow distance[i][k] + distance[k][j]$
 7:             end if
 8:         end for
 9:     end for
10: end for
11: return $distance$
```

2) Related Work: Several algorithms have been proposed for computing shortest paths in transportation networks, each with its strengths and limitations. Dijkstra's algorithm, for example, efficiently finds the shortest path from a single source to all other vertices in a graph, making it suitable for real-time route planning.

However, Dijkstra's algorithm is not well-suited for computing shortest paths between all pairs of vertices in a dense graph. Bellman-Ford algorithm, on the other hand, can handle negative edge weights but has a time complexity of O(VE), where V is the number of vertices and E is the number of edges. In contrast, the Floyd-Warshall algorithm has a time complexity of $O(V^3)$, making it suitable for dense graphs. Previous studies have applied the Floyd-Warshall algorithm to various domains, including network routing, urban planning, and logistics optimization, demonstrating its versatility and effectiveness in solving the all-pairs shortest path problem.

In this section, the c programming approach employed by the Floyd-Warshall algorithm is explored in detail. It explains how the algorithm breaks down the problem of finding the shortest paths between all pairs of vertices into smaller subproblems and utilizes memoization to store and reuse intermediate results. The concept of optimal substructure is discussed, emphasizing how the optimal solution to a larger problem can be constructed from optimal solutions to its subproblems. Examples and illustrations are provided to illustrate the application of dynamic programming techniques in solving the all-pairs shortest path problem.

3) Dataset Generation: For this study, we generated a directed graph to represent transportation costs using an adjacency matrix. The dataset was designed to simulate a transportation network, with considerations made to ensure symmetry and handle unreachable locations by assigning INF (infinity) values. Random costs were assigned between points to mimic real-world transportation scenarios.

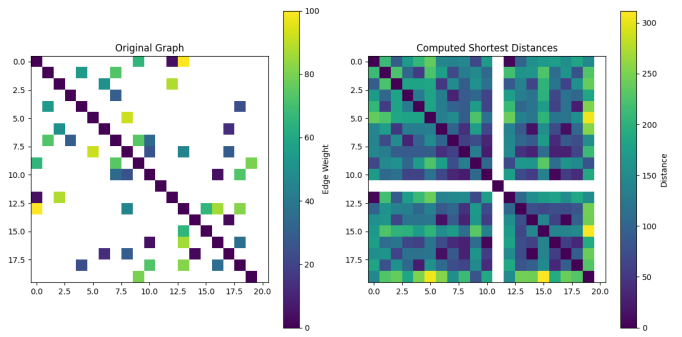


Fig. 1. Original Graph vs Computed Shortest Distances

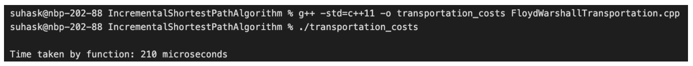
4) Algorithm Implementation: The implementation of the Floyd-Warshall algorithm begins with initializing a matrix to represent the shortest distances between pairs of vertices. Then, the algorithm iteratively updates the matrix by considering all possible intermediate vertices and recalculating the shortest distances. This process continues until all pairs of vertices have been considered, resulting in a matrix containing the shortest distances between all pairs of vertices.

5) Analysis and Optimization: After implementing the algorithm, it is essential to analyze its performance in terms of execution time, memory usage, and accuracy of results. Optimization techniques may be applied to improve the algorithm's efficiency, such as reducing redundant calculations or optimizing data structures. Benchmarking against other algorithms or datasets can provide insights into the algorithm's effectiveness and scalability.

The results of this study have significant implications for transportation planning, logistics management, and urban infrastructure development. By leveraging the Floyd-Warshall algorithm, transportation authorities and logistics companies can improve route planning, reduce congestion, and optimize resource allocation in transportation networks. However, it is essential to consider the limitations of the algorithm, particularly its computational complexity for large-scale graphs. Future research directions may focus on developing hybrid algorithms that combine the strengths of different approaches to achieve faster and more scalable solutions for transportation optimization. Additionally, incorporating real-time data and dynamic updates into the algorithm can further enhance its applicability in dynamic transportation environments. Overall, the Floyd-Warshall algorithm offers a promising avenue for addressing the challenges of transportation optimization and advancing the efficiency and sustainability of transportation systems worldwide.

Algorithm 2 Naive Floyd-Warshall's Algorithm

```
1:  Create a $|V| \times |V|$ matrix, $M$, to describe the distances between vertices
2:  for each cell $(i, j)$ in $M$ do
3:      if $i == j$ then
4:          $M[i][j] = 0$
5:      else if $(i, j)$ is an edge in $E$ then
6:          $M[i][j] = weight(i, j)$
7:      else
8:          $M[i][j] = \infty$
9:      end if
10: end for
11: for $k$ from 1 to $|V|$ do
12:     for $i$ from 1 to $|V|$ do
13:         for $j$ from 1 to $|V|$ do
14:             if $M[i][j] > M[i][k] + M[k][j]$ then
15:                 $M[i][j] = M[i][k] + M[k][j]$
16:             end if
17:         end for
18:     end for
19: end for
```



```
suhask@nbp-202-88 IncrementalShortestPathAlgorithm % g++ -std=c++11 -o transportation_costs FloydWarshallTransportation.cpp
suhask@nbp-202-88 IncrementalShortestPathAlgorithm % ./transportation_costs

Time taken by function: 210 microseconds
```

Fig. 2. Benchmarking Floyd Warshall's Algorithm

Algorithm 3 Find Affected Sources

Input: distance matrix d, edge update $(u, v, w'(u, v))$
Output: Affected Source S

```
1:  $N \leftarrow (d.size)$
2:  $visited \leftarrow$ [False] $* n$
3:  $S \leftarrow \emptyset$
4:  if $d[u][v] > w'$ then
5:      $Q \leftarrow \emptyset$
6:      $S$.enqueue($u$)
7:      $visited[u] \leftarrow$ True
8:      while $Q$ is not empty do
9:          $x \leftarrow Q$.dequeue()
10:         for $n \in$ range($N$) do
11:             if not$visited[n]$ and $d[n][v] > d[n][u] + w'$ then
12:                 $Q$.enqueue($z$)
13:                 $vis[z] \leftarrow$ True
14:                 $S$.insert($z$)
15:             end if
16:         end for
17:     end while
18: end if
```

B. Implementation of RR

The RR (Ramalingam-Reps) algorithm addresses dynamic shortest path problems by efficiently updating shortest paths when changes [1]. The algorithm requires two arguments; distance matrix d, new edge $u, v, w'(u, v)$ where u represents the source node, v represents the destination node, and w represents the updated weight of the edge between nodes u and v. The distance matrix input d is pre-computed by the floyd-warshall algorithm before the start of the algorithm.

The RR algorithm starts by identifying the source nodes (sources) that are affected by the change in edge weight between nodes u and v with the new weight w'. The `identify_affected_source` function identifies nodes that can achieve shorter paths to a target node v through a new or updated edge $E(u, v)$ with weight w', using a breadth-first search approach to propagate potential path improvements through the graph. For each affected source node s, it initializes a queue and an array to explore nodes connected to v. If the updated edge weight offers a shorter path from s to v, it updates the shortest path to d and marks v as visited, then enqueues it for further exploration. The main loop processes each node in the queue, exploring their neighbors to update distances where the newly adjusted path offers an improvement, ensuring all shortest paths are recalculated only where necessary, thus optimizing the overall update process. In other words, RR algorithm identify affected sources and update the distance matrix by visiting vertices in BFS manne. The tight bound complexity of RR is $\Theta(S \times (T + E_T))$ where S represents the affected source and T refers the targeted node and E_t is the number of edges w.r.t T.

Algorithm 4 RR Algorithm

Input: distance matrix d, edge update $(u, v, w'(u, v))$
Output: updated distance matrix d

```
1:  $S \leftarrow$ find_affected_source($d, (u, v, w'(u, v))$)
2:  $N \leftarrow$ len($d$)
3:  for $s \in S$ do
4:      $visited \leftarrow$ [False] $* N$
5:      $Q \leftarrow \emptyset$
6:      if $d[s][v] > d[s][u] + w'$ then
7:          $d[s][v] \leftarrow d[s][u] + w'$
8:          $Q$.enqueue($v$)
9:          $visited[v] \leftarrow$ True
10:         while $Q$is not empty do
11:             $y \leftarrow Q$.dequeue()
12:             for $n \in N$ do
13:                 if not$visited[n]$ and $d[s][n] > d[s][u] + w' + d[v][n]$ then
14:                     $d[s][n] \leftarrow d[s][u] + w' + d[v][n]$
15:                     $visited[n] \leftarrow$ True
16:                     $Q$.enqueue($n$)
17:                 end if
18:             end for
19:         end while
20:     end if
21: end for
```

C. Implementation of QUINCA

The QUINCA algorithm is another advanced method designed to efficiently update all-pairs shortest paths (APSP) in graphs following incremental changes such as edge insertions or weight decreases [2]. Similar to RR algorithm, it first identifies all the nodes whose shortest paths to a given node v can be improved by a new edge $E(u,v)$. Once these affected source nodes are determined, QUINCA employs a modified breadth-first search (BFS) that does not necessarily process nodes in the order of increasing distances, which differs from traditional Dijkstra's method. This approach minimizes recomputation by ensuring that each node and edge is processed fewer times compared to previous methods. The authors of QUINCA algorithm argue that the worst case complexity of $O(n^2)$ where n is the number of sources since the QUINCA algorithm has doubly nested loop. This makes QUINCA alorithm outperform RR method.

Algorithm 5 QUINCA algorithm

```
Input: distance matrix $d$, edge update $(u, v, w'(u, v))$
Output: updated distance matrix $d$
1: if $\omega'(u,v) < d(u,v)$ then
2:     $S \leftarrow$ find_affected_source$(d, (u, v, w'(u, v)))$
3:     $d(u,v) \leftarrow w'(u,v)$;
4:     $Q \leftarrow \emptyset$;
5:     visited $\leftarrow$ [False] $* N$;
6:     $Q$.push$(v)$;
7:     visited$(v) \leftarrow$ true;
8:     while $Q$ is not empty do
9:         $y \leftarrow Q$.front();
10:        for all $x \in S$ do
11:            if $d(x,y) > d(x,u) + w' + d(v,y)$ then
12:                $d(x,y) \leftarrow d(x,u) + w' + d(v,y)$;
13:            end if
14:        end for ▷ enqueue all neighbors that get closer to $u$
15:        for all $n \in N$ do
16:            if not visited$(n)$ and $d(u,n) > w' + d(v,n)$ then
17:                $Q$.enqueue$(n)$;
18:                visited$(n) \leftarrow$ true;
19:            end if
20:        end for
21:    end while
22: end if
```

D. Implementation of LDSP

The LDSP algorithm [3] is an old algorithm proposed in 2004. Unlike the previous two algorithms, it supports fully-dynamics update, which means it could accept both weight increase and weight decrease as its input. Also, it could process all changed edges incident to a certain vertex in single update, which is useful when all the edges incident to a vertex are requiring an update. Here we will simply demonstrate the definitions used in this algorithm.

Definition II.1. *A path π_{xy} is LOCALLY SHORTEST in Graph G if either:*
(i) π_{xy} consists of a single vertex, or
(ii) Every proper subpath of π_{xy} is a shortest path in G.

It's obvious that all shortest paths are locally shortest paths, but locally shortest paths are not necessarily shortest. The LDSP algorithm maintain the following 7 sets:

$P(x,y)$	set of locally shortest path from x to y
$P^*(x,y)$	set of shortest path from x to y
$L(\pi_{xy})$	set of pre-extension paths which are locally shortest
$L^*(\pi_{xy})$	set of pre-extension paths which are shortest
$R(\pi_{xy})$	set of post-extension paths which are locally shortest
$R^*(\pi_{xy})$	set of post-extension paths which are shortest
$w(\pi_{xy})$	weight of a path

With those definitions, we could easily get the shortest path and its distance with the following two functions:

Algorithm 6 distance(x, y)

```
1: return the minimum weight of paths in $P(x,y)$
```

Algorithm 7 path(x, y)

```
1: return the path with minimum weight in $P(x,y)$
```

Then we will introduce the algorithm to maintain the sets to be correct after each update.

Algorithm 8 LDSP algorithm

```
Input: distance matrix $d$, multiple edge update $(u, \mathbf{w})$
Output: updated correct sets $P(x,y)$, $P^*(x,y)$, $L(x,y)$, $L^*(x,y)$, $R(x,y)$, $R^*(x,y)$, $w(x,y)$
1: cleanup$(u)$
2: fixup$(u, \mathbf{w})$
```

Algorithm 9 cleanup(u)

```
1: $Q \leftarrow \langle v \rangle$
2: while Q $\neq \emptyset$ do
3:     extract any $\pi$ from Q
4:     for all $\pi_{xy} \in L(\pi) \cup R(\pi)$ do
5:         add $\pi_{xy}$ to $Q$
6:         remove $\pi_{xy}$ from $P_{xy}$, $L(r(\pi_{xy}))$, and $R(l(\pi_{xy}))$
7:         if $\pi_{xy} \in P^*$ then remove $\pi_{xy}$ from $P^*$ , $L^*(r(\pi_{xy}))$, and $R^*(l(\pi_{xy}))$
8:         end if
9:     end for
10: end while
```

The clean up function is simple. It removes every paths containing vertex u from $P(x,y)$, $P^*(x,y)$, $L(x,y)$, $L^*(x,y)$, $R(x,y)$, $R^*(x,y)$. This is done iteratively by deleting the path and adding more paths into the set Q.

Algorithm 10 fixup($u, \mathbf{w}$)

```
1:  for all  $u \neq v$ do                                    $\{Phase1\}$
2:      $w_{uv} \leftarrow w'_{uv}$;$w_{vu} \leftarrow w'_{vu}$
3:      if $w_{uv} < \infty$ then
4:          $w(\langle u,v\rangle) \leftarrow w_{uv}$; $l(\langle u,v\rangle) \leftarrow \langle u\rangle$;
5:          $r(\langle u,v\rangle) \leftarrow \langle v\rangle$, add $\langle u,v\rangle$ to $P_{uv}$, $L(\langle v\rangle)$,$R(\langle u\rangle)$
6:      end if
7:      if $w_{vu} < \infty$ then
8:          $w(\langle v,u\rangle) \leftarrow w_{vu}$; $l(\langle v,u\rangle) \leftarrow \langle v\rangle$;
9:          $r(\langle v,u\rangle) \leftarrow \langle u\rangle$, add $\langle v,u\rangle$ to $P_{vu}$, $L(\langle u\rangle)$,$R(\langle v\rangle)$
10:     end if
11: end for

12: $H \leftarrow \emptyset$                                   $\{Phase2\}$
13: for all $(x,y)$ do
14:     add $\pi_{xy} \in P_{xy}$ with minimum $w(\pi_{xy})$ to $H$
15: end for

16: while $H \neq \emptyset$ do                               $\{Phase3\}$
17:     extract $\pi_{xy}$ from $H$ with minimum $w(\pi_{xy})$
18:     if $\pi_{xy}$ is the first extracted path for pair $(x,y)$ then
19:         if $\pi_{xy} \notin P^*_{xy}$ then
20:             add $\pi_{xy}$ to $P^*_{xy}$ , $L^*(r(\pi_{xy}))$, and $R^*(l(\pi_{xy}))$
21:             for all $\pi_{x'b} \in L^*(l(\pi_x y))$ do
22:                 $\pi_{x'y} \leftarrow \langle x',x\rangle\pi_{xy}$;
23:                 $w(\pi_{x'y}) \leftarrow w_{x'x} + w(\pi_{xy})$;
24:                 $l(\pi_{x'y}) \leftarrow \pi_{x'b}$;
25:                 $r(\pi_{x'y}) \leftarrow \pi_{xy}$
26:                 add $\pi_{x'y}$ to $P_{x'y}, L(\pi_{xy}), R(\pi_{x'b})$, and $H$
27:             end for
28:             for all $\pi_{ay'} \in R^*(r(\pi_{xy}))$ do
29:                 $\pi_{xy'} \leftarrow \pi_{xy}\langle y,y'\rangle$
30:                 $w(\pi_{xy'}) \leftarrow w(\pi_{xy}) + w_{yy'}$;
31:                 $l(\pi_{xy'}) \leftarrow \pi_{xy}$;
32:                 $r(\pi_{xy'}) \leftarrow \pi_{ay'}$
33:                 add $\pi_{xy'}$ to $P_{xy'}, L(\pi_{ay'}), R(\pi_{xy})$, and $H$
34:             end for
35:         end if
36:     end if
37: end while
```

There are three phases of this fixup process. In phase 1, all edges incident to the vertex u are added to the corresponding sets. In phase 2, a priority queue H is initialized with the existing locally shortest paths. In phase 3, locally shortest paths are extracted from the phase 3, added to the shortest path set, and longer paths are considered by searching the pre-extension and post-extension sets.

III. Experiment

We have implemented four different incremental APSP algorithms: the naive Floyd-Warshall, LDSP, RR, and QUINCA. Our implementations are written in C++ and all algorithms have been unit-tested under various graph conditions. The implementations are contained in `main.cpp`, while unit testing can be found in the `test.cpp`. For comparison purposes, we use a Python implementation because the Python `matplotlib` library offers decent visualization capabilities. Python implementation and visualization can be found `in Plot Incremental Algorithms/func.py` and `Plot Incremental Algorithms/plot.py` respectively. We measured the running time for each incremental APSP algorithm. Our experimental procedure is as follows: (i) generate a random graph, (ii) select an edge to update, (iii) compute the distance matrix using the Floyd-Warshall algorithm, and (iv) run the incremental shortest path algorithms. We measure the running time during step (iv), as the previous steps are constant and do not affect the performance comparison. We generated random graphs where the size of the graph ranges from 10 to 200 with an interval of 5. We ensure that a new edge $E'(u,v)$ that is randomly picked for insertion must be less than the existing $E(u,v)$ before replacement. As we believe that the number of edges is less influential than the number of vertices, we use the same number of edges across the multiple runs. Lastly, we aggregated the running time for each algorithm and averaged them for a singular comparison. The experiment result shows (figure. 3) that all incremental APSP algorithms outperform the naive floyd-warshall. The QUINCA algorithm achieves the best performance since its worst complexity is more efficient than the other two incremental algorithms. While the LDSP algorithm underperforms the RR and QUINCA algorithms, it is a fully dynamic APSP algorithm and can update all the edges incident to a vertex in a single step.

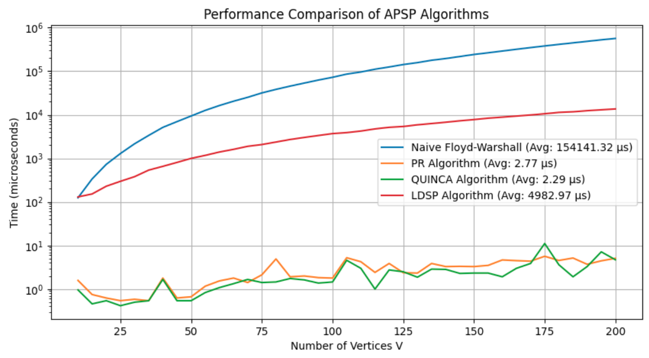


Fig. 3. Comparison between the four algorithms

IV. Conclusion

In this study, we addressed the incremental shortest all-pairs-shortest-paths (APSP) problems by focusing on the implementation and analysis of three algorithms: LDSP, RR, and QUINCA. Through thorough unit testing and a meticulously conducted experiment, we demonstrated the efficiency of these algorithms against static APSP algorithms under various conditions. Our experimental setup was designed to ensure fairness, with each algorithm tested under identical conditions to measure true performance differences accurately. The results indicate that the QUINCA algorithm outperforms LDSP and RR algorithms, due to its sophisticated mechanism that efficiently handles updates by minimizing unnecessary recomputation and optimizing node and edge processing. While the QUINCA algorithm is the most efficient among our implementations, LDSP can handle the fully dynamic graph setting including decremental APSP problems, which encourages us to investigate decremental APSP problems for our future study. Our study not only highlights the effectiveness of QUINCA in solving incremental APSP problems but also illustrates its potential for addressing real-world challenges.

Acknowledgment

We appreciate Prof.Kruger for instructing the course and advising project ideas.

References

[1] G. Ramalingam and T. Reps, "On the computational complexity of dynamic graph problems," *Theoretical Computer Science*, vol. 158, no. 1-2, pp. 233–277, 1996.

[2] A. Slobbe, E. Bergamini, and H. Meyerhenke, *Faster incremental all-pairs shortest paths*. KIT, 2016.

[3] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths," *Journal of the ACM (JACM)*, vol. 51, no. 6, pp. 968–992, 2004.