Solving the All-Pairs Shortest Path Problem after Minor Updates to Large Dense Graphs

Authors: Gangli Liu, Gangli Liu

Date: 2025-02-05T09:14:41+00:00

Abstract

The all pairs shortest path problem is a fundamental optimization problem in graph theory. We deal with re-calculating the all-pairs shortest path (APSP) matrix after a minor modification of a weighted dense graph, e.g., adding a node, removing a node, or updating an edge. We assume the APSP matrix for the original graph is already known. The graph can be directed or undirected. A cold-start calculation of the new APSP matrix by traditional algorithms, like the Floyd-Warshall algorithm or Dijkstra's algorithm, needs $ O(n^3) $ time. We propose two algorithms for warm-start calculation of the new APSP matrix. The best case complexity for a warm-start calculation is $ O(n^2) $, the worst case complexity is $ O(n^3) $. We implemented the algorithms and tested their performance with experiments. The result shows a warm-start calculation can save a great portion of calculation time, compared with cold-start calculation.

Full Text

Preamble

Solving the All-Pairs Shortest Path Problem After Minor Update of a Large Dense Graph

Gangli Liu
Tsinghua University
gl-liu13@mails.tsinghua.edu.cn

ABSTRACT

The all-pairs shortest path problem is a fundamental optimization problem in graph theory. We address the problem of re-calculating the all-pairs shortest path (APSP) matrix after a minor modification of a weighted dense graph, such as adding a node, removing a node, or updating an edge. We assume the APSP

matrix for the original graph is already known. The graph can be directed or undirected. A cold-start calculation of the new APSP matrix using traditional algorithms like the Floyd-Warshall algorithm or Dijkstra's algorithm requires $O(n^3)$ time.

We propose two algorithms for warm-start calculation of the new APSP matrix. The best-case complexity for a warm-start calculation is $O(n^2)$, while the worst-case complexity is $O(n^3)$. We implemented the algorithms and tested their performance experimentally. The results show that warm-start calculation can save a substantial portion of computation time compared to cold-start calculation. Additionally, another algorithm is devised for warm-start calculation of the shortest path between two nodes. Experiments demonstrate that warm-start calculation can save 99% of computation time compared to cold-start calculation by Dijkstra's algorithm on large directed complete graphs.

KEYWORDS: All pairs shortest path; Shortest path problem; Minimax path problem; Widest path problem

ACM Reference Format:
Gangli Liu. 0000. Solving the all pairs shortest path problem after minor update of a large dense graph. In *Proceedings of 000, Beijing, China, 0000* (0000), 10 pages.
DOI: 00.000/000 0

1 INTRODUCTION

The Shortest Path Problem is a fundamental optimization problem in graph theory and computer science. It involves finding the shortest path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

Let $G = (V, E)$ be a graph where: - V is the set of vertices (nodes) - $E \subseteq V \times V$ is the set of edges (connections between nodes) - $w : E \to \mathbb{R}^+ \cup \{0\}$ is a weight function assigning a non-negative weight to each edge

For a given source vertex $s \in V$ and target vertex $t \in V$, the Shortest Path Problem seeks to find a path P from s to t such that $P = \{v_1, v_2, ..., v_k\}$, $v_1 = s$, $v_k = t$, and the total path weight is minimized:
minimize: $W(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$, where $(v_i, v_{i+1}) \in E$ for $i = 1, 2, ..., k-1$.

The all-pairs shortest path (APSP) problem computes the shortest paths between all pairs of vertices $u, v \in V$. A dense graph is a graph in which the number of edges is close to the maximum possible number of edges for the given number of vertices.

Let $G = (V, E)$ be a graph, where $|V| = n$ is the number of vertices and $|E|$ is the number of edges. A dense graph satisfies: $|E| \approx O(n^2)$. This means the number of edges grows quadratically with the number of vertices. For an undirected graph, the maximum possible number of edges is $\binom{n}{2} = \frac{n(n-1)}{2}$. For

a directed graph, the maximum possible number of edges is $n(n-1)$. A graph is considered dense when $|E|$ is close to these upper bounds. Dense graphs are common in applications such as social networks, transportation networks, or communication networks where most entities are interconnected.

In this paper, we address the problem of re-calculating the all-pairs shortest path (APSP) matrix after a minor modification of a weighted dense graph, such as adding a node, removing a node, or updating an edge. We assume the APSP matrix for the original graph is already known. The graph can be directed or undirected. A straightforward method for calculating the APSP matrix of the updated graph is to apply the Floyd-Warshall algorithm to recalculate from scratch, which requires $O(n^3)$ time—a prohibitively expensive cost for large dense graphs. Our goal is to leverage the already computed APSP matrix to make the calculation of the new APSP matrix less expensive.

2 RELATED WORK

The Shortest Path Problem (SPP) is a foundational topic in graph theory and optimization, with numerous applications in transportation networks, telecommunications, and logistics [1, 3, 16, 20, 21]. Over the years, various algorithms and techniques have been developed to solve different variants of the problem efficiently.

2.1 Classical Algorithms

One of the earliest contributions to SPP dates back to the work of Dijkstra (1959), who proposed a greedy algorithm to solve the single-source shortest path problem for graphs with non-negative edge weights in $O(|V|^2)$ time, later optimized to $O(|E| + |V| \log |V|)$ using priority queues [6]. The foundational idea of this algorithm also appears in Leyzorek et al. (1957) [13].

For graphs with negative edge weights, the Bellman-Ford algorithm (1958) provides a reliable solution, albeit with a higher computational cost of $O(|V||E|)$ [2]. The Floyd-Warshall algorithm (1962) extends these techniques to compute all-pairs shortest paths in $O(|V|^3)$ time, leveraging dynamic programming [7].

2.2 Optimizations and Modern Variants

Advances in data structures, such as Fibonacci heaps [8], have further improved the efficiency of Dijkstra's algorithm. More recently, heuristic-based approaches like A^* have been widely adopted for real-world applications, where an admissible heuristic guides the search to improve runtime performance [10]. Parallel and distributed versions of shortest path algorithms have also emerged, leveraging modern computing architectures for large-scale graphs [17].

2.3 Specialized Applications

The SPP has been extended to address specialized scenarios, such as the multi-criteria shortest path problem, which considers trade-offs between multiple objectives like cost and time [9]. In dynamic or time-dependent graphs, edge weights may vary over time, necessitating new algorithms like the time-expanded shortest path [5]. Additionally, the rise of massive graphs in social networks and geographic information systems has spurred the development of approximate methods, such as graph sparsification and sketching.

2.4 Challenges in Dense and Weighted Graphs

For dense graphs, where the number of edges approaches $O(|V|^2)$, naive algorithms often become computationally expensive. Techniques like matrix-based methods for all-pairs shortest paths [11] or GPU-accelerated implementations [12] have shown promise in reducing computational overhead.

2.5 Emerging Trends

Recent research has explored incorporating machine learning into shortest path computations. These methods predict likely paths or edge weights, complementing traditional algorithms in scenarios with incomplete or noisy data [19]. Moreover, shortest path calculations are increasingly being integrated with clustering and community detection tasks to solve problems in network science and biology [18].

3 UPDATING A LARGE GRAPH

In previous work, we proposed Algorithm 1 (MMJ distance by recursion) for solving the all-pairs minimax path problem or widest path problem [14]. It can also be revised to solve the APSP matrix for the shortest path problem, which also takes $O(n^3)$ time.

3.1 APSP After Adding a Node

As discussed in Section 6.1 (Merit of Algorithm 1) of [15], Algorithm 1 (MMJ distance by recursion) has the advantage of warm-start capability. Consider the scenario where we have already computed the APSP matrix M_G for a large graph G, and a new point or node p, which is not part of G, is introduced. The updated graph is referred to as $G + p$. When determining the APSP matrix for $G + p$, conventional algorithms like Floyd-Warshall or Dijkstra's algorithm might necessitate computation from scratch, which takes $O(n^3)$ time.

Algorithm 1 leverages the precomputed M_G to facilitate calculation of the new APSP matrix for graph $G+p$, based on the results of Theorem 3.1, 3.2, 3.5, and Corollary 3.3, 3.4, which are revised from the theorems and corollary in Section 3.3 (Other properties of MMJ distance) of [14]. A warm-start of Algorithm 1

requires only $O(n^2)$ time, which is much less expensive than the $O(n^3)$ time required for a cold-start of conventional algorithms.

Theorem 3.1. Suppose $r \in \{1, 2, ..., n\}$, $f(t) = d(G_{n+1}, G_t) + SPD(G_t, G_r|G[1, n])$, and $X = \{f(t)|t \in \{1, 2, ..., n\}\}$. Then, $SPD(G_{n+1}, G_r|G[1, n + 1]) = \min(X)$. For the meaning of G_t, G_r, $G[1, n]$, and $G[1, n + 1]$, see Table 1 .

Proof. There are n possibilities for the shortest path from G_{n+1} to G_r under the context of $G[1, n+1]$. Set X enumerates them all. Each element of X is the shortest path distance of each possibility. Therefore, according to the definition of shortest path distance, $SPD(G_{n+1}, G_r|G[1, n+1]) = \min(X)$. The n possibilities are not mutually exclusive; multiple possibilities can happen simultaneously if the shortest paths are not unique.

Theorem 3.2. Suppose $r \in \{1, 2, ..., n\}$, $f(t) = SPD(G_r, G_t|G[1, n]) + d(G_t, G_{n+1})$, and $X = \{f(t)|t \in \{1, 2, ..., n\}\}$. Then, $SPD(G_r, G_{n+1}|G[1, n + 1]) = \min(X)$.

Proof. The proof is similar to the proof of Theorem 3.1. Since we are dealing with a directed graph, the order of nodes in a distance notation matters.

Corollary 3.3. Suppose $r \in \{1, 2, ..., N\}$, $p \notin G$, $f(t) = d(p, G_t) + SPD(G_t, G_r|G)$, and $X = \{f(t)|t \in \{1, 2, ..., N\}\}$. Then, $SPD(p, G_r|G + p) = \min(X)$. For the meaning of $G + p$, see Table 1.

Proof. The proof follows the conclusion of Theorem 3.1.

Corollary 3.4. Suppose $r \in \{1, 2, ..., N\}$, $p \notin G$, $f(t) = SPD(G_r, G_t|G) + d(G_t, p)$, and $X = \{f(t)|t \in \{1, 2, ..., N\}\}$. Then, $SPD(G_r, p|G + p) = \min(X)$.

Proof. The proof follows the conclusion of Theorem 3.2.

Theorem 3.5. Suppose $i, j \in \{1, 2, ..., n\}$, $x_1 = SPD(G_i, G_j|G[1, n])$, $t_1 = SPD(G_i, G_{n+1}|G[1, n + 1])$, $t_2 = SPD(G_{n+1}, G_j|G[1, n + 1])$, and $x_2 = t_1 + t_2$. Then, $SPD(G_i, G_j|G[1, n + 1]) = \min(x_1, x_2)$.

Proof. If the shortest paths are not unique, there are two possibilities for the shortest path from G_i to G_j under the context of $G[1, n + 1]$: (1) There exists a shortest path from G_i to G_j that does not contain node G_{n+1}, which means G_{n+1} is not necessary for the shortest path from G_i to G_j under the context of $G[1, n + 1]$. That is, $SPD(G_i, G_j|G[1, n + 1]) = SPD(G_i, G_j|G[1, n])$. (2) All shortest paths from G_i to G_j must contain node G_{n+1}, which means G_{n+1} is necessary for the shortest path from G_i to G_j under the context of $G[1, n + 1]$. That is, $SPD(G_i, G_j|G[1, n + 1]) \neq SPD(G_i, G_j|G[1, n])$. Here, x_1 is the SPD of the first possibility, and x_2 is the SPD of the second possibility. Therefore, according to the definition of shortest path distance, $SPD(G_i, G_j|G[1, n+1]) = \min(x_1, x_2)$. If the shortest path is unique, the reasoning still holds. The two possibilities are mutually exclusive; they cannot happen simultaneously.

Theorem 3.6. Suppose $i, j \in \{1, 2, ..., n\}$ are affected by removing Node C (except Node C itself). Let $t_1 = SPD(G_i, G_{n+1}|G[1, n + 1])$,

$t_2 = SPD(G_{n+1}, G_j|G[1, n+1])$, and $SPD(G_i, G_j|G[1, n+1]) < t_1 + t_2$. Then, $SPD(G_i, G_j|G[1, n+1]) = SPD(G_i, G_j|G[1, n])$, which means G_{n+1} is not necessary for the shortest path from G_i to G_j under the context of $G[1, n+1]$.

Proof. As discussed in the proof of Theorem 3.5, there are two possibilities for the shortest path from G_i to G_j under the context of $G[1, n+1]$, and these two possibilities are mutually exclusive. We only need to negate possibility (2) to arrive at the conclusion. Suppose possibility (2) occurs; then G_{n+1} is necessary for the shortest path from G_i to G_j under the context of $G[1, n+1]$, which means $SPD(G_i, G_j|G[1, n+1]) = t_1 + t_2$. This contradicts the condition that $SPD(G_i, G_j|G[1, n+1]) < t_1 + t_2$. Therefore, possibility (2) cannot occur; only possibility (1) can occur.

Corollary 3.7. Suppose $i, j, k \in \{1, 2, ..., N\}$, $k \neq i$, $k \neq j$, $t_1 = SPD(G_i, G_k|G)$, $t_2 = SPD(G_k, G_j|G)$, and $SPD(G_i, G_j|G) < t_1 + t_2$. Then, $SPD(G_i, G_j|G) = SPD(G_i, G_j|G - G_k)$, which means G_k is not necessary for the shortest path from G_i to G_j under the context of G.

Proof. The proof follows the conclusion of Theorem 3.6. We simply re-index the nodes in graph G.

3.2 APSP After Removing a Node

Sometimes we need to remove a node from a large graph. Suppose we remove the k-th node from graph G; the new graph is denoted $G - G_k$ (see Table 1). A cold-start calculation using conventional algorithms to compute the APSP matrix of graph $G - G_k$ would take $O(n^3)$ time. However, we can employ a smarter method to reduce computational expense.

First, we create a *need-update list* to record which nodes' shortest path distances (SPD) to others are affected by the deletion. For example, in Figure 1 [Figure 1: see original paper], if we delete Node C, we obtain Figure 2 [Figure 2: see original paper] (since the matrices are symmetric, we only show half of them). Removing a node is equivalent to setting that node's distances to and from other nodes to infinity. The need-update list for Figure 2 is completely empty because none of the pairwise shortest path distances are affected:

$SPD(A, B|G) = SPD(A, B|G - C)$
$SPD(A, D|G) = SPD(A, D|G - C)$
$SPD(B, A|G) = SPD(B, A|G - C)$
$SPD(B, D|G) = SPD(B, D|G - C)$
$SPD(D, A|G) = SPD(D, A|G - C)$
$SPD(D, B|G) = SPD(D, B|G - C)$

The need-update list for Figure 2 looks like this:
Node A: empty
Node B: empty
Node D: empty

In Figure 3 [Figure 3: see original paper], we removed Node B. Some of the remaining pairwise shortest path distances are affected, while others are not. For example:

$SPD(A, C|G) \neq SPD(A, C|G - B)$
$SPD(A, D|G) \neq SPD(A, D|G - B)$
$SPD(C, A|G) \neq SPD(C, A|G - B)$
$SPD(C, D|G) = SPD(C, D|G - B)$
$SPD(D, A|G) \neq SPD(D, A|G - B)$
$SPD(D, C|G) = SPD(D, C|G - B)$

The need-update list for Figure 3 looks like this:
Node A: $[A, C], [A, D]$
Node C: $[C, A]$
Node D: $[D, A]$

We use Theorem 3.6 to construct the need-update list by treating the node to be removed as G_{n+1}. If $SPD(G_i, G_j|G[1, n+1]) < t_1 + t_2$, which means G_{n+1} is not necessary for the shortest path from G_i to G_j under the context of $G[1, n + 1]$, then G_{n+1} can be safely removed from the graph without affecting the shortest path distance from G_i to G_j. Thus, node pair $[G_i, G_j]$ will not appear in the need-update list.

Otherwise, if $SPD(G_i, G_j|G[1, n + 1]) = t_1 + t_2$, then we cannot be certain whether the shortest path distance from G_i to G_j will be affected by removing node G_{n+1} from the graph; the SPD from G_i to G_j needs to be recalculated after removal. Therefore, node pair $[G_i, G_j]$ will be appended to the need-update list of node G_i. Corollary 3.7 makes this easier to understand than Theorem 3.6. Constructing the need-update list requires only $O(n^2)$ time.

Definition 3.8. Cost for calculating the new APSP matrix after removing node G_k from G:

$C(G, M_G, G_k) = \frac{\Phi(M_G, G_k)}{N-1}$

where M_G is the APSP matrix of graph G; G_k is the node to be removed; $\Phi(M_G, G_k)$ is the number of non-empty items in the need-update list after removing node G_k from G; and N is the number of vertices in graph G. The range of $C(G, M_G, G_k)$ is $[0, 1]$.

[Figure 1: see original paper]
[Figure 2: see original paper]
[Figure 3: see original paper]

We devise an algorithm for solving the APSP matrix after removing a node. In Algorithm 5, we first construct the need-update list after removing node G_k, then use this list to calculate the Cost defined in Definition 3.8. If the Cost is larger than hyper-parameter δ, we simply use the Floyd-Warshall algorithm to recalculate the APSP matrix of the new graph $G - G_k$. If the Cost is small, we use Dijkstra's algorithm to compute a node's distance to other nodes in the

new graph $G - G_k$. Hopefully, only a few nodes will be affected by removing node G_k, thereby saving time for calculating the new APSP matrix.

Thus, removing a node from a graph is more challenging than adding a node when calculating the APSP matrix. Adding a node requires only $O(n^2)$ time even in the worst case. For removing a node, the best-case complexity is $O(n^2)$, while the worst-case complexity is $O(n^3)$. For example, in the need-update list for Figure 2, all items are empty, so the complexity for solving the new APSP matrix is $O(n^2)$, which is used for calculating the need-update list. In the need-update list for Figure 3, all items are non-empty, resulting in $O(n^3)$ complexity.

3.3 APSP After Modifying an Edge

Modifying an edge can be accomplished by removing one of the edge's vertices, then adding the node back with the updated edge. Therefore, the best-case complexity for modifying an edge is $O(n^2)$, and the worst-case complexity is $O(n^3)$—the same complexity as removing a node.

Algorithm 5 provides the APSP calculation after removing a node, while Algorithm 6 handles APSP after modifying an edge. Algorithm 6 first removes a node associated with the edge from the graph, then adds the node back with the edge updated. Since an edge is associated with two nodes, before removing a node, it calculates which node is cheaper to remove, then removes the cheaper one.

4 WARM-START CALCULATION OF SHORTEST PATH

We can perform a warm-start calculation of the shortest path between two nodes based on the known APSP matrix and the conclusion of Theorem 3.6. Algorithm 7 is devised for warm-start calculation of the shortest path between two nodes using the APSP matrix. It employs Theorem 3.6 to exclude unnecessary nodes from the path between nodes i and j, generating a candidate node list. It then forms a small graph composed of nodes in this candidate list, uses Dijkstra's algorithm to calculate the shortest path from node i to node j on this small graph, and finally translates the path back to the original node indices. Since the candidate node list is usually very small, calculating the shortest path from node i to j on the small graph typically requires only $O(1)$ time. Therefore, the average-case complexity of Algorithm 7 is $O(n)$.

4.1 Correctness Proof of Algorithm 7

The correctness of Algorithm 7 follows from Theorem 3.6.

Theorem 4.1. The small graph composed of nodes in the candidate node list in Algorithm 7 contains all the shortest paths from node i to node j on graph G.

Proof. We can divide nodes in graph G into two sets: nodes in the candidate node list, denoted G_c, and nodes not in the candidate node list, denoted G_r. Suppose there exists a shortest path from node i to node j on graph G that involves a node in G_r; let this involved node be ξ. The path is denoted $p(i,\xi) + p(\xi,j)$. Since the length of path $p(i,\xi) + p(\xi,j)$ is greater than or equal to $M_G[i,\xi] + M_G[\xi,j]$, this contradicts Step 9 of Algorithm 7, which states that the shortest path distance from node i to node j on graph G is less than $M_G[i,\xi] + M_G[\xi,j]$. Therefore, a shortest path from node i to node j on graph G cannot involve a node in G_r, proving the correctness of Theorem 4.1.

4.2 All Shortest Paths Between Two Nodes

The generated small matrix and candidate node list in Algorithm 7 can be used to calculate all shortest paths between two nodes on graph G. Algorithm 8 is devised for warm-start calculation of all shortest paths between two nodes based on the APSP matrix and the conclusion of Theorem 4.1. We can even enumerate all paths from node i to node j to check if they are shortest paths, since the graph defined by the small matrix is small.

4.3 All Shortest Paths on Undirected Graph

When the graph is undirected and the APSP matrix is unknown, Algorithm 9 can be used to calculate all shortest paths between two nodes. Since the APSP matrix is unknown, the calculation is cold-start. The average-case complexity of Algorithm 9 is $O(n^2)$. When the graph is directed, the complexity is $O(n^3)$ because we need $O(n^3)$ time to first calculate the APSP matrix.

4.4 Maintaining a Key Node List

When all shortest paths from node i to node j are known, we can calculate a key node list for node pair (i,j), which collects all essential nodes that form a shortest path from node i to node j. When needing to remove a node, we can simply check each pair of nodes' key node list to determine if the shortest path is affected. Algorithm 10 is a variant of Algorithm 5 that calculates the new APSP matrix by utilizing the key node list. Since the key node list for each pair of nodes is usually small, the average-case space complexity is $O(n^2)$. Steps 2-8 of Algorithm 10 can be calculated in advance of knowing which node will be removed. Algorithm 10 works even when all shortest paths calculated in Step 4 are not complete (e.g., if some shortest paths were missed during Step 4).

4.5 Another Variant of Algorithm 5

Although it can be calculated in advance of knowing which node is to be removed, the key node list in Algorithm 10 is very expensive to calculate, with time complexity at least $O(n^3)$. Therefore, we devise another variant of Algorithm 5 that uses the conclusion of Corollary 3.7 and the technique from Algorithm 10 to calculate the key node list for one pair of nodes, then checks

if the node being removed is in the key node list from node i to node j. To save time, we replace $\{i, j\}$ with $\Psi(i, j)$ in Step 8 of Algorithm 10. This new algorithm is referred to as Algorithm 11. Further experiments in Section 5 show that Algorithm 11 performs better than Algorithm 5.

4.6 A Variant of Algorithm 11

Algorithm 12 is a variant of Algorithm 11 based on the conclusions of Theorem 4.2 and Corollary 3.7. Preliminary tests show Algorithm 12 is slightly faster than Algorithm 11.

Theorem 4.2. Suppose $i, j, k \in \{1, 2, ..., N\}$, $k \neq i$, $k \neq j$. If the shortest path distance from node G_i to G_j on graph $G - G_k$ is larger than on graph G, then G_k is necessary for the shortest path from node G_i to G_j on graph G.

Proof. Suppose G_k is not necessary for the shortest path from node G_i to G_j on graph G, which means there exists a shortest path from G_i to G_j that does not contain node G_k on graph G. This implies that the shortest path distance from node G_i to G_j on graph $G - G_k$ is equal to that on graph G, which contradicts the condition that "the shortest path distance from node G_i to G_j on graph $G - G_k$ is larger than on graph G."

5 TESTING OF THE ALGORITHMS

We tested the algorithms for warm-start calculation of the new APSP matrix after minor updates to a dense graph, such as removing a node or modifying an edge. All experimental code was implemented in Python. To enable more reliable comparisons, we converted the Python code to C++.

5.1 Experiment I

In Experiment I, we tested warm-start calculation of the new APSP matrix after removing a node and compared it with cold-start calculation using the Floyd-Warshall algorithm. In each trial, a random node was removed from a complete graph, and we recorded the time spent calculating the new APSP matrix using both warm-start calculation with Algorithm 5 and cold-start calculation with Floyd-Warshall. The time ratio was calculated using Equation 27. We tested complete graphs of various sizes, from 1,000 to 5,000 nodes. For each graph size smaller than 5,000 nodes, we repeated the experiment 20 times and calculated the average and standard deviation (SD) of the ratios; for the 5,000-node graph, the experiment was repeated five times.

As shown in Table 2 , warm-start calculation requires only 0.57 of the time needed for cold-start calculation with the Floyd-Warshall algorithm on average, meaning we can save 43% of computation time using warm-start calculation.

5.2 Experiment II

The setting of Experiment II is similar to Experiment I, except that we tested modifying an edge rather than removing a node. As shown in Table 3 , warm-start calculation can save 50% of computation time compared to the Floyd-Warshall algorithm.

5.3 Experiment III

In Experiment III, we tested warm-start calculation of the shortest path between two nodes based on the known APSP matrix and Theorem 3.6, comparing it with cold-start calculation using Dijkstra's algorithm. Other experimental settings were similar to Experiments I and II. We tested calculating the shortest path between two nodes on complete graphs of different sizes using both warm-start and cold-start methods, with each method repeated 1,000 times. The results show that warm-start calculation requires only 0.01 of the time needed for cold-start calculation on average, meaning we can save 99% of computation time using warm-start calculation.

5.4 Experiment IV

The setting of Experiment IV is similar to Experiment I, except that we used Algorithm 11 instead of Algorithm 5. By comparing Table 5 with Table 2 , we can see that Algorithm 11 performs better than Algorithm 5.

6 DISCUSSION

The algorithms can be revised for warm-start calculation of the minimax path problem or widest path problem on large dense graphs.

7 CONCLUSION

We propose two algorithms (and several variants) for warm-start calculation of the all-pairs shortest path (APSP) matrix after a minor modification of a weighted dense graph, such as adding a node, removing a node, or updating an edge. We assume the APSP matrix for the original graph is already known and attempt to warm-start from this known matrix to reach the new APSP matrix. A cold-start calculation of the APSP matrix for the updated graph requires $O(n^3)$ time, which is prohibitively expensive for large dense graphs. By leveraging the already computed APSP matrix, we can significantly reduce the computational cost. The best-case complexity for warm-start calculation is $O(n^2)$, while the worst-case complexity is $O(n^3)$.

We implemented the algorithms and tested their performance experimentally. The results demonstrate that warm-start calculation can save a substantial portion of computation time compared to the Floyd-Warshall algorithm. Moreover, we proposed another algorithm for warm-start computation of the shortest path

between two nodes and tested it. The results show that warm-start computation can save 99% of the time required by cold-start computation using Dijkstra's algorithm.

REFERENCES

[1] Ravindra K Ahuja, Kurt Mehlhorn, James Orlin, and Robert E Tarjan. 1990. Faster algorithms for the shortest path problem. *Journal of the ACM (JACM)* 37, 2 (1990).

[2] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.

[3] Quan Chen, Lei Yang, Yong Zhao, Yi Wang, Haibo Zhou, and Xiaoqian Chen. 2024. Shortest path in LEO satellite constellation networks: An explicit analytic approach. *IEEE Journal on Selected Areas in Communications* (2024).

[4] Edith Cohen. 1997. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. System Sci.* 55, 3 (1997), 441–453.

[5] Kenneth L Cooke and Eric Halsey. 1966. The shortest route through a network with time-dependent internodal transit times. *Journal of mathematical analysis and applications* 14, 3 (1966), 493–498.

[6] EW DIJKSTRA. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1 (1959), 269–271.

[7] Robert W Floyd. 1962. Algorithm 97: shortest path. *Commun. ACM* 5, 6 (1962).

[8] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)* 34, 3 (1987), 596–615.

[9] Pierre Hansen. 1980. Bicriterion path problems. In *Multiple Criteria Decision Making Theory and Application*: Proceedings of the Third Conference Hagen/Königswinter, West Germany, August 20–24, 1979. Springer, 109–127.

[10] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[11] Donald B Johnson. 1977. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)* 24, 1 (1977), 1–13.

[12] Gary J Katz and Joseph T Kider Jr. 2008. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. 47–55.

[13] M Leyzorek, RS Gray, AA Johnson, WC Ladew, SR Meaker Jr, RM Petry, and RN Seitz. 1957. Investigation of model techniques–first annual report–6

june 1956–1 july 1957–a study of model techniques for communication systems. *Case Institute of Technology, Cleveland, Ohio* (1957).

[14] Gangli Liu. 2023. Min-Max-Jump distance and its applications. *arXiv preprint arXiv:2301.05994* (2023).

[15] Gangli Liu. 2024. An efficient implementation for solving the all pairs minimax path problem in an undirected dense graph. *arXiv preprint arXiv:2407.07058* (2024).

[16] Tobia Marcucci, Jack Umenberger, Pablo Parrilo, and Russ Tedrake. 2024. Shortest paths in graphs of convex sets. *SIAM Journal on Optimization* 34, 1 (2024), 507–538.

[17] Ulrich Meyer and Peter Sanders. 1998. δ-stepping: A parallel single source shortest path algorithm. In *European symposium on algorithms*. Springer, 393–404.

[18] Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM review* 45, 2 (2003), 167–256.

[19] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[20] Dongbo Zhang, Yanfang Shou, and Jianmin Xu. 2024. A mapreduce-based approach for shortest path problem in road networks. *Journal of Ambient Intelligence and Humanized Computing* (2024), 1–9.

[21] Zhaocheng Zhu, Xinyu Yuan, Michael Galkin, Louis-Pascal Xhonneux, Ming Zhang, Maxime Gazeau, and Jian Tang. 2024. A* net: A scalable path-based reasoning approach for knowledge graphs. *Advances in Neural Information Processing Systems* 36 (2024).

Note: Figure translations are in progress. See original paper for figures.