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 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. In addition, another algorithm is devised to warm-start calculate of the shortest path between two nodes. Experiment shows warm-start calculation can save 99% of calculation time, compared with cold-start calculation by Dijkstra's algorithm, on directed complete graphs of large sizes.

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),* 9 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:

$$P = \{v_1, v_2, \ldots, v_k\}, \quad v_1 = s,\ v_k = t,$$

and the total path weight is minimized:

$$\text{minimize: } W(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}),$$

where $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$.

The all-pairs shortest path (APSP) problem compute 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 like social networks, transportation networks, or communication networks where most entities are interconnected.

In this paper, 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 straightforward method for calculating the APSP matrix of the updated graph is to use the Floyd–Warshall algorithm to recalculate the updated graph, it needs $O(n^3)$ time. It is a very expensive time cost for a large dense graph. We are trying to utilize the already calculated APSP matrix to make 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

0000, Beijing, China
0000. 000-0000-00-000/00/00…$00.00
DOI: 00.000/000_0

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 idea of this algorithm is also given 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 $O(|V|^3)$, 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, the 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 [4].

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 a previous paper, we propose 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 of 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 $\mathbb{M}_G$ 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 algorithm or Dijkstra's algorithm might necessitate a computation beginning from scratch, which takes $O(n^3)$ time.

Algorithm 1 leverages the precomputed $\mathbb{M}_G$ to facilitate the calculation of the new APSP matrix of graph $G+p$, in accordance with 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 a cold-start of conventional algorithms, which takes $O(n^3)$ time.

Theorem 3.1. *Suppose* $r \in \{1,2,\ldots,n\}$,

$$f(t) = d(G_{n+1}, G_t) + SPD(G_t, G_r \mid G_{[1,n]}) \tag{1}$$

$$\mathbb{X} = \{f(t) \mid t \in \{1,2,\ldots,n\}\} \tag{2}$$

then,

$$SPD(G_{n+1}, G_r \mid G_{[1,n+1]}) = min(\mathbb{X}) \tag{3}$$

For the meaning of $G_t, G_r, G_{[1,n]}$, *and* $G_{[1,n+1]}$, see Table 1.

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

Theorem 3.2. *Suppose* $r \in \{1,2,\ldots,n\}$,

$$f(t) = SPD(G_r, G_t \mid G_{[1,n]}) + d(G_t, G_{n+1}) \tag{4}$$

$$\mathbb{X} = \{f(t) \mid t \in \{1,2,\ldots,n\}\} \tag{5}$$

then,

$$SPD(G_r, G_{n+1} \mid G_{[1,n+1]}) = min(\mathbb{X}) \tag{6}$$

Proof. The proof is similar to 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* $r \in \{1,2,\ldots,N\}, p \notin G$,

$$f(t) = d(p, G_t) + SPD(G_t, G_r \mid G) \tag{7}$$

$$\mathbb{X} = \{f(t) \mid t \in \{1,2,\ldots,N\}\} \tag{8}$$

then,

$$SPD(p, G_r \mid G+p) = min(\mathbb{X}) \tag{9}$$

For the meaning of $G+p$, see Table 1.

Proof. The proof follows the conclusion of Theorem 3.1. □

Corollary 3.4. *Suppose* $r \in \{1,2,\ldots,N\}, p \notin G$,

$$f(t) = SPD(G_r, G_t \mid G) + d(G_t, p) \tag{10}$$

$$\mathbb{X} = \{f(t) \mid t \in \{1,2,\ldots,N\}\} \tag{11}$$

then,

$$SPD(G_r, p \mid G+p) = min(\mathbb{X}) \tag{12}$$

Table 1: Table of notations

G	A weighted dense graph of N nodes, with each node indexed from 1 to N. Graph G is supposed to be directed, an undirected graph can be considered as a special case of a directed graph;
$G_{[1,n]}$	A graph that is composed of the first n nodes of G, the nodes are indexed from 1 to n;
G_{n+1}	The $(n+1)$th node of G;
G + p	Graph G plus one new node p. Since $p \notin G$, if G has N nodes, this new graph now has $N+1$ nodes;
$G - G_k$	A new graph by removing the kth node from graph G. If G has N nodes, this new graph now has $N-1$ nodes;
$G^{'}$	A new graph by modifying weight of one edge of graph G, or by removing a node from G, or by adding a node to G;
$\Psi_{(i,j,n,G)}$	$\Psi_{(i,j,n,G)}$ is a sequence from node i to node j, which has a total number of n nodes. All the nodes in the sequence must belong to graph G. That is to say, it is a path starts from i, and ends with j. The path is not allowed to have loops, unless the start and the end is the same node;
$d(i,j)$	$d(i,j)$ is the adjacency distance from node i to node j on graph G. Note the graph is directed;
$len(\ \Psi_{(i,j,n,G)}\ )$	$len(\ \Psi_{(i,j,n,G)}\ )$ is the length of path $\Psi_{(i,j,n,G)}$, which is the sum of edge weights on the path;
$\Theta_{(i,j,G)}$	$\Theta_{(i,j,G)}$ is the set of all paths from node i to node j. A path in $\Theta_{(i,j,G)}$ can have arbitrary number of nodes (at least two). All the nodes in a path must belong to graph G;
$SPD(i,j \mid G)$	$SPD(i,j \mid G)$ is the shortest path distance (SPD) from node i to node j, where G is the ***Context*** of the shortest path distance. The ***Context*** of a distance is defined in [14]. A node's shortest path distance to itself is always 0;
$\mathbb{M}_{k,G_{[1,k]}}$	$\mathbb{M}_{k,G_{[1,k]}}$ is the pairwise shortest path distance matrix of $G_{[1,k]}$, which has shape $k \times k$. The shortest path distances are under the ***Context*** of $G_{[1,k]}$;
$\mathbb{M}_G$	The APSP matrix of G, $\mathbb{M}_G = \mathbb{M}_{N,G_{[1,N]}}$;

Proof. The proof follows the conclusion of Theorem 3.2. □

Theorem 3.5. *Suppose* $i, j \in \{1, 2, \ldots, n\}$,

$$x_1 = SPD(G_i, G_j \mid G_{[1,n]}) \tag{13}$$

$$t_1 = SPD(G_i, G_{n+1} \mid G_{[1,n+1]}) \tag{14}$$

$$t_2 = SPD(G_{n+1}, G_j \mid G_{[1,n+1]}) \tag{15}$$

$$x_2 = t_1 + t_2 \tag{16}$$

then,

$$SPD(G_i, G_j \mid G_{[1,n+1]}) = min(x_1,\ x_2) \tag{17}$$

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 one shortest path from G_i to G_j which 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 to say:
$$SPD(G_i, G_j \mid G_{[1,n+1]}) = SPD(G_i, G_j \mid G_{[1,n]})$$

(2) All the 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 to say:
$$SPD(G_i, G_j \mid G_{[1,n+1]}) \neq SPD(G_i, G_j \mid G_{[1,n]})$$

x_1 is the SPD of the first possibility; x_2 is the SPD of the second possibility. Therefore, according to the definition of shortest path distance, $SPD(G_i, G_j \mid G_{[1,n+1]}) = min(x_1,\ x_2)$. If the shortest path is unique, the reasoning still holds. The two possibilities are mutually exclusive; they cannot happen simultaneously. □

Theorem 3.6. *Suppose $i, j \in \{1, 2, \ldots, n\}$,*

$$t_1 = SPD(G_i, G_{n+1} \mid G_{[1,n+1]}) \tag{18}$$

$$t_2 = SPD(G_{n+1}, G_j \mid G_{[1,n+1]}) \tag{19}$$

if,

$$SPD(G_i, G_j \mid G_{[1,n+1]}) < t_1 + t_2 \tag{20}$$

then,

$$SPD(G_i, G_j \mid G_{[1,n+1]}) = SPD(G_i, G_j \mid G_{[1,n]}) \tag{21}$$

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 the two possibilities are mutually exclusive; they cannot happen simultaneously. We only need to negate possibility (2), then we can arrive to the conclusion of Equation 21. Suppose possibility (2) happens, then G_{n+1} is necessary for the shortest path from G_i to G_j, under the context of $G_{[1,n+1]}$. Then $SPD(G_i, G_j \mid G_{[1,n+1]}) = t_1 + t_2$, which is contradicted to Equation 20. Therefore, possibility (2) cannot happen; only possibility (1) can happen. □

Corollary 3.7. *Suppose $i,\ j,\ k \in \{1, 2, \ldots, N\}, k \neq i, k \neq j$,*

$$t_1 = SPD(G_i, G_k \mid G) \tag{22}$$

$$t_2 = SPD(G_k, G_j \mid G) \tag{23}$$

if,

$$SPD(G_i, G_j \mid G) < t_1 + t_2 \tag{24}$$

then,

$$SPD(G_i, G_j \mid G) = SPD(G_i, G_j \mid G - G_k) \tag{25}$$

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 just 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 removed the kth node from graph G, the new graph is noted $G - G_k$ (Table 1). A cold-start of conventional algorithms for calculating the APSP matrix of graph $G - G_k$ will take $O(n^3)$ time.

However, we can take a smarter method to make the computation less expensive. Firstly, we make a *need_update_list* to record which nodes' shortest path distance (SPD) to others are affected by the deletion. E.g., in Figure 1, if we delete Node C, we get Figure 2 (since the matrices are symmetric, we only show half of them). Removing a node is equivalent to set the node's distances (to and from) to other nodes to infinity. The *need_update_list* is totally empty for Figure 2. Because none of the pair-wise shortest path distances are affected by removing Node C. Except Node C itself. E.g.,

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

The *need_update_list* for Figure 2 looks like this:

$$Node\ A : empty$$
$$Node\ B : empty$$
$$Node\ D : empty$$

In Figure 3, we removed Node B. Some of the remaining pair-wise shortest path distances are affected, some are not. E.g.,

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

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 setting the node to be removed as G_{n+1}. If

$$SPD(G_i, G_j \mid G_{[1,n+1]}) < t_1 + t_2$$

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. So, node pair $[G_i,\ G_j]$ will not appear in the *need_update_list*. Otherwise, if

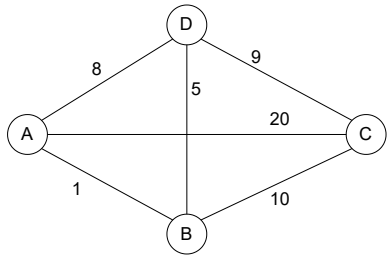
$$SPD(G_i, G_j \mid G_{[1,n+1]}) = t_1 + t_2$$

then we are not sure 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 re-calculated after the removing. So, node pair $[G_i,\ G_j]$ will be appended to the *need_update_list* of node G_i. Corollary 3.7 makes it easier to understand than Theorem 3.6. Constructing the *need_update_list* only needs $O(n^2)$ time.

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

$$C(G, \mathbb{M}_G, G_k) = \frac{\Phi(\mathbb{M}_G, G_k)}{N - 1} \tag{26}$$

Where $\mathbb{M}_G$ is the APSP matrix of graph G; G_k is the node to be removed; $\Phi(\mathbb{M}_G, G_k)$ is the number of non-empty items in the *need_update_list*, after removing node G_k from G; N is the number of vertices in graph G. The range of $C(G, \mathbb{M}_G, G_k)$ is $[0, 1]$.



(a) Graph

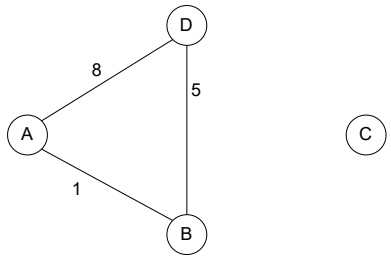
	A	B	C	D
A	0	1	20	8
B		0	10	5
C			0	9
D				0

(b) adjacency matrix

	A	B	C	D
A	0	1	11	6
B		0	10	5
C			0	9
D				0

(c) APSP matrix

Figure 1: Graph, adjacency matrix, and APSP matrix.



(a) Graph

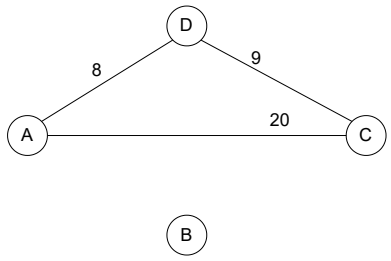
	A	B	C	D
A	0	1	∞	8
B		0	∞	5
C			0	∞
D				0

(b) adjacency matrix

	A	B	C	D
A	0	1	∞	6
B		0	∞	5
C			0	∞
D				0

(c) APSP matrix

Figure 2: Graph, adjacency matrix, and APSP matrix, after removing node C.



(a) Graph

	A	B	C	D
A	0	∞	20	8
B		0	∞	∞
C			0	9
D				0

(b) adjacency matrix

	A	B	C	D
A	0	∞	17	8
B		0	∞	∞
C			0	9
D				0

(c) APSP matrix

Figure 3: Graph, adjacency matrix, and APSP matrix, after removing node B.

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 the *need_update_list* to calculate the *Cost* defined in Definition 3.8. If the *Cost* is larger than hyper-parameter δ, we just use the Floyd-Warshall algorithm to re-calculate the APSP matrix of the new graph $G - G_k$. If the *Cost* is small, we use Dijkstra's algorithm to calculate 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, therefore, saved time for calculating the new APSP matrix.

So, removing a node from a graph is harder than adding a node to the graph, for calculating the APSP matrix. Adding a node only needs $O(n^2)$ time even for the worst case. For removing a node, the best case complexity is $O(n^2)$, the worst case complexity is $O(n^3)$. E.g., in the *need_update_list* for Figure 2, all the items are empty, 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 the items are non-empty, the complexity is $O(n^3)$.

3.3 APSP after modifying an edge

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

Algorithm 5 APSP after removing a node

Input: G, APSP of G: $\mathbb{M}_G$, node to be removed: G_k, hyper-parameter: δ
Output: $APSP\ of\ G - G_k : \mathbb{M}_{G-G_k}$

function APSP_REMOVE_NODE(G, $\mathbb{M}_G$, G_k, δ)
  $remaining\_node\_list \leftarrow G - G_k$
  $need\_update\_list \leftarrow cal\_need\_update\_list(G_k, \mathbb{M}_G)$
  $Cost \leftarrow cal\_cost\_of\_remove(need\_update\_list)$
  if $Cost > \delta$ **then**
    $\mathbb{M}_{G-G_k} \leftarrow Floyd\_Warshall(G - G_k)$
    return $\mathbb{M}_{G-G_k}$
  end if
  $\mathbb{M}_{G-G_k} \leftarrow copy(\mathbb{M}_G)$
  for i *in* $remaining\_node\_list$ **do**
    if $len(need\_update\_list[i]) > 0$ **then**
      //We can stop early if the shortest path tree has covered all the nodes in $need\_update\_list[i]$.
      $temp \leftarrow dijkstra\_one\_to\_all\_others(G - G_k,\ i)$
      for j *in* $need\_update\_list[i]$ **do**
        $\mathbb{M}_{G-G_k}[i, j] \leftarrow temp[j]$
      end for
    else
      $pass$
    end if
  end for
  return $\mathbb{M}_{G-G_k}$
end function

Algorithm 6 is devised to calculate the APSP matrix after modifying an edge. It firstly remove a node associated with the edge from the graph, then add the node back, with the edge being updated. An edge is associated with two nodes. So, before removing a node, it calculates which node is cheaper to remove, then remove the cheaper one.

4 WARM-START CALCULATION OF SHORTEST PATH

We can carry out a warm-start calculation of the shortest path between two nodes, based on the already 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, based on the APSP matrix. It use the conclusion of Theorem 3.6 to exclude unnecessary nodes from node i to node j, generate the *candidate_node_list*; then form a small graph which is composed of nodes in *candidate_node_list*; then use Dijkstra's algorithm to calculate the shortest path from node i to node j, on the small graph; then translate the path into original node index.

Since the *candidate_node_list* is usually very small, calculating the shortest path from node i to j on the small graph usually needs only $O(1)$ time. So the average case complexity of Algorithm 7 is $O(n)$.

Algorithm 6 APSP after modifying an edge

Input: G, APSP of G: $\mathbb{M}_G$, edge nodes: e_n, edge weight: e_w, hyper-parameter: δ
Output: APSP of new graph G': $\mathbb{M}_{G'}$

function APSP_MODIFY_EDGE(G, $\mathbb{M}_G$, e_n, e_w, δ)
  $i \leftarrow e_n[0]$
  $j \leftarrow e_n[1]$
  $Cost\_i \leftarrow cal\_cost\_of\_remove(G, i)$
  $Cost\_j \leftarrow cal\_cost\_of\_remove(G, j)$
  if $Cost\_i < Cost\_j$ **then**
    $G_k \leftarrow i$
  else
    $G_k \leftarrow j$
  end if
  //Calculate APSP matrix after removing node.
  $\mathbb{M}_{G-G_k} \leftarrow APSP\_remove\_node(G, \mathbb{M}_G, G_k, \delta)$
  //Add the node back, update the edge, then use warm-start of Algorithm 1 (MMJ distance by recursion) to calculate the new APSP matrix.
  $G' \leftarrow cal\_updated\_graph(G, e_n, e_w)$
  $\mathbb{M}_{G'} \leftarrow APSP\_add\_node(G', \mathbb{M}_{G-G_k})$
  return $\mathbb{M}_{G'}$
end function

4.1 Correctness proof of Algorithm 7

The correctness of Algorithm 7 follows the conclusion of Theorem 4.1.

THEOREM 4.1. *The small graph which is composed of nodes in candidate_node_list in Algorithm 7 contains all the shortest paths from node i to j on graph G.*

PROOF. We can divide nodes in graph G into two sets: nodes in *candidate_node_list*, noted G_c; nodes not in *candidate_node_list*, noted G_r. Suppose there exists a shortest path from node i to j on graph G involves a node in G_r, the involved node is noted ξ. The path is noted $p(i,\ \xi) + p(\xi,\ j)$. The APSP matrix of G is $\mathbb{M}_G$. Since the length of path $p(i,\ \xi) + p(\xi,\ j)$ is great than or equal to $\mathbb{M}_G[i,\ \xi] + \mathbb{M}_G[\xi,\ j]$, which is contradict to Step 9 of Algorithm 7, which says the shortest path distance from node i to j on graph G is less than $\mathbb{M}_G[i,\ \xi] + \mathbb{M}_G[\xi,\ j]$. So, a shortest path from node i to j on graph G cannot involve a node in G_r, the correctness of Theorem 4.1 is proved. □

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 the shortest paths between two nodes on graph G. Algorithm 8 is devised for warm-start calculation of all the shortest paths between two nodes, based on the APSP matrix and conclusion of Theorem 4.1.

We can even enumerate all the paths from node i to j to check if it is a shortest path, since the graph decided by *small_matrix* is small.

Algorithm 7 warm-start calculation of shortest path

Input: APSP of G: $\mathbb{M}_G$, Adjacency matrix: $\mathbb{A}_G$, start node: i, end node: j
Output: Shortest path from i to j: $path(i,\ j)$

function WARM_CAL_SHORTEST_PATH($\mathbb{M}_G$, $\mathbb{A}_G$, i, j)
  if $i == j$ **then**
    return $[i]$
  end if
  $remaining\_node\_list \leftarrow G - i - j$
  $candidate\_node\_list \leftarrow empty\_list$
  $candidate\_node\_list.append(i)$
  for t *in* $remaining\_node\_list$ **do**
    if $\mathbb{M}_G[i,\ j] < \mathbb{M}_G[i,\ t] + \mathbb{M}_G[t,\ j]$ **then**
      pass
    else
      $candidate\_node\_list.append(t)$
    end if
  end for
  $candidate\_node\_list.append(j)$
  $K \leftarrow len(candidate\_node\_list)$
  $small\_matrix \leftarrow zeros((K, K))$
  for $i,\ m$ *in* $enumerate(candidate\_node\_list)$ **do**
    for $j,\ n$ *in* $enumerate(candidate\_node\_list)$ **do**
      $small\_matrix[i,\ j] \leftarrow \mathbb{A}_G[m,\ n]$
    end for
  end for
  //Use Dijkstra's algorithm to calculate the path from node 0 to node $K-1$, on the graph defined by $small\_matrix$.
  $path \leftarrow cal\_path\_by\_dijkstra(small\_matrix, 0, K-1)$
  //Translate the path into original node index.
  $path(i,\ j) \leftarrow [candidate\_node\_list[i]\ for\ i\ in\ path]$
  return $path(i,\ j)$
end function

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 calculate the APSP matrix firstly.

4.4 Maintaining a key_node_list

When all shortest paths from node i to j is known, we can calculate a key_node_list for node pair $(i,\ j)$, which collects all the essential nodes to form a shortest path from node i to j. When needing to remove a node, we can just check each pair of nodes' key_node_list to decide if the shortest path is affected. Algorithm 10 is a variant of Algorithm 5, which 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 $O(n^2)$.

Step 2 to 8 of Algorithm 10 can be calculated in advance of knowing which node is about to be removed. Algorithm 10 works

Algorithm 8 warm-start calculation of all shortest paths

Input: $small\_matrix$: $\mathbb{M}_s$, $candidate\_node\_list$: $\mathbb{C}_l$, start node: i, end node: j
Output: All shortest paths from i to j on graph G: $\mathbb{P}(i,\ j)$

function WARM_CAL_ALL_SHORTEST_PATHS($\mathbb{M}_s$, $\mathbb{C}_l$, i, j)
  if $i == j$ **then**
    return $[[i]]$
  end if
  $\mathbb{P}(i,\ j) \leftarrow empty\_list$
  Use Dijkstra's algorithm to calculate a shortest path from node i to j, on the graph defined by $\mathbb{M}_s$, noted $\Psi_{(i,j)}$;
  Append $\Psi_{(i,j)}$ to $\mathbb{P}(i,\ j)$;
  Divide nodes in $\mathbb{C}_l$ into two sets: nodes in $\{i,\ j\}$, noted Φ_p; nodes not in $\{i,\ j\}$, noted Φ_r;
  for t *in* Φ_r **do**
    Calculate the shortest path from node i to t, and t to j, link the two paths into a new path P_new;
    Check if P_new is already in $\mathbb{P}(i,\ j)$, if yes, *continue*;
    Append P_new to $\mathbb{P}(i,\ j)$;
  end for
  return $\mathbb{P}(i,\ j)$
end function

Algorithm 9 Cold-start calculation of all shortest paths on undirected graph

Input: Adjacency matrix: $\mathbb{A}_G$, start node: i, end node: j
Output: All shortest paths from i to j on graph G: $\mathbb{P}(i,\ j)$

function ALL_SHORTEST_PATHS_UNDIRECTED_GRAPH($\mathbb{A}_G$, i, j)
  if $i == j$ **then**
    return $[[i]]$
  end if
  $remaining\_node\_list \leftarrow G - i - j$
  $candidate\_node\_list \leftarrow empty\_list$
  $candidate\_node\_list.append(i)$
  Use Dijkstra's algorithm to calculate shortest path distances from node i to all nodes on graph G, noted $\mathbb{V}_i$;
  Use Dijkstra's algorithm to calculate shortest path distances from node j to all nodes on graph G, noted $\mathbb{V}_j$;
  for t *in* $remaining\_node\_list$ **do**
    if $\mathbb{V}_i[j] < \mathbb{V}_i[t] + \mathbb{V}_j[t]$ **then**
      pass
    else
      $candidate\_node\_list.append(t)$
    end if
  end for
  $candidate\_node\_list.append(j)$
  Calculate $small\_matrix$ $\mathbb{M}_s$ with $candidate\_node\_list$ and $\mathbb{A}_G$;
  Use Algorithm 8 to calculate all shortest paths between i and j on graph G, noted $\mathbb{P}(i,\ j)$;
  return $\mathbb{P}(i,\ j)$
end function

even when all shortest paths calculated in Step 4 is not complete (e.g., we have missed some shortest paths during Step 4).

Algorithm 10 warm-start calculation of APSP by *key_node_list*

Input: G, APSP of G: $\mathbb{M}_G$, node to be removed: G_k, hyper-parameter: δ
Output: $APSP\ of\ G - G_k : \mathbb{M}_{G-G_k}$

function APSP_BY_KEY_NODE_LIST($G, \mathbb{M}_G, G_k, \delta$)
  $key\_node\_list\_all \leftarrow empty\_list$
  for Each pair of node $(i,\ j)$ **do**
    Use Algorithm 8 to calculate all the shortest paths from node i to j on graph G;
    Calculate the intersection of all shortest paths, noted $\mathbb{L}_k$, $\mathbb{L}_k$ collects all the essential nodes to form a shortest path from node i to j;
    Remove node i and j from $\mathbb{L}_k$;
    $key\_node\_list\_all.append(\mathbb{L}_k)$
  end for
  Calculate the *need_update_list* of Algorithm 5 with *key_node_list_all*, by checking each pair of nodes' $\mathbb{L}_k$ to decide if the shortest path is affected when removing G_k;
  Use Step 4 to 20 of Algorithm 5 to calculate $\mathbb{M}_{G-G_k}$;
  return $\mathbb{M}_{G-G_k}$
end function

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_all* in Algorithm 10 is very expensive to calculate, the time complexity is at least $O(n^3)$. Therefore, we devise another variant of Algorithm 5, which uses the conclusion of Corollary 3.7 and the technique used in Algorithm 10 to calculate the *key_node_list*, for one pair of nodes. Then check if the being removed node is in the *key_node_list* from node i to j. To save some time, we replace $\{i,\ j\}$ with $\Psi_{(i,j)}$ in Step 8 of Algorithm 8.

The new algorithm is referred to as Algorithm 11. Further experiment in Section 5 shows Algorithm 11 performs better than Algorithm 5.

5 TESTING OF THE ALGORITHMS

We tested the algorithms for warm-start calculation of the new APSP matrix after a minor update of a dense graph, e.g., removing a node, or modifying an edge. All the code in the experiments is implemented with Python. To compare the algorithms more reliably, we convert the Python code into C++ code.

5.1 Experiment I

In Experiment I, we test warm-start calculation of the new APSP matrix after removing a node, and compare with cold-start calculation of the Floyd-Warshall algorithm. In the experiment, a random node is removed from a complete graph, then record the time spent for calculating the new APSP matrix, by warm-start calculation of Algorithm 5 and cold-start of Floyd-Warshall algorithm. The ratio of the used time is calculate with Equation 27. Different sizes of complete graphs are tested, from 1,000 nodes to 5,000 nodes. For each size of graph less than 5,000 nodes, we repeat the experiment 20 times and calculate the average and standard deviation (SD) of the ratios; for graph of size of 5,000 nodes, the experiment is repeated five times.

$$r = \frac{APSP_{warm}(G^{'})}{APSP_{cold}(G^{'})} \tag{27}$$

As shown in Table 2, a warm-start calculation only needs 0.57 of the time, to a cold-start calculation with the Floyd-Warshall algorithm on average. That means we can save 43% of calculation time if using warm-start calculation.

Algorithm 11 APSP by *key_node_list* and Corollary 3.7

Input: Adjacency matrix: $\mathbb{A}_G$, APSP of G: $\mathbb{M}_G$, node to be removed: G_k, hyper-parameter: δ
Output: $APSP\ of\ G - G_k : \mathbb{M}_{G-G_k}$

function APSP_BY_KEY_NODE_LIST($G, \mathbb{M}_G, G_k, \delta$)
  $remaining\_node\_list \leftarrow G - G_k$
  for i *in remaining_node_list* **do**
    for j *in remaining_node_list* **do**
      if $\mathbb{M}_G[i,\ j] >= \mathbb{M}_G[i,\ k] + \mathbb{M}_G[k,\ j]$ **then**
        Calculate the *key_node_list* from i to j with the technique used in Algorithm 10;
        if G_k in *key_node_list* **then**
          $need\_update\_list[i].append(j)$
        end if
      end if
    end for
  end for
  Use Step 4 to 20 of Algorithm 5 to calculate $\mathbb{M}_{G-G_k}$;
  return $\mathbb{M}_{G-G_k}$
end function

Table 2: Warm-start vs. cold-start calculation of the APSP matrix, after removing a node.

	N = 1000	N = 2000	N = 3000	N = 5000	Avg
Ratio	0.58±0.32	0.48±0.22	0.61±0.19	0.6±0.13	0.57

Table 3: Warm-start vs. cold-start calculation of the APSP matrix, after modifying an edge.

	N = 1000	N = 2000	N = 3000	N = 5000	Avg
Ratio	0.46±0.24	0.46±0.21	0.58±0.23	0.51±0.11	0.5

5.2 Experiment II

The setting of Experiment II is similar to Experiment I, except that we are testing modifying an edge, not removing a node. As shown in Table 3, we can save 50% of calculation time if using warm-start calculation, when compared with the Floyd-Warshall algorithm.

Table 4: Warm-start vs. cold-start calculation of shortest path.

	N = 1000	N = 2000	N = 3000	N = 5000	Avg
Ratio	0.02±0.00	0.02±0.01	0.01±0.00	0.007±0.00	0.01

Table 5: Warm-start vs. cold-start calculation of the APSP matrix, after removing a node, by Algorithm 11.

	N = 1000	N = 2000	N = 3000	N = 5000	Avg
Ratio	0.45±0.31	0.34±0.22	0.24±0.13	0.33±0.16	0.34

5.3 Experiment III

In Experiment III, we test warm-start calculation of the shortest path between two nodes, based on the known APSP matrix and Theorem 3.6. And compared with cold-start calculation of the shortest path by Dijkstra's algorithm.

Other settings of the experiment are similar to Experiment I and II. In the experiment, we test calculating the shortest path between two nodes, on complete graphs of different sizes, by warm-start and cold-start calculation separately. Each method is repeated 1,000 times. The result shows a warm-start calculation only needs 0.01 of the time of a cold-start calculation on average. That means we can save 99% of calculation time if using warm-start calculation.

5.4 Experiment IV

The setting of Experiment IV is similar to Experiment I, except that we are using 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 a large dense graph.

7 CONCLUSION

We propose two algorithms for warm-start calculation of 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, and try to warm-start from the known APSP matrix to reach the new APSP matrix. A cold-start calculation of the APSP matrix for the updated graph needs $O(n^3)$ time. It is a very expensive time cost for a large dense graph. We are trying to utilize the already calculated APSP matrix to make calculation of the new APSP matrix less expensive. 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 large portion of calculation time when compared with the Floyd-Warshall algorithm. Moreover, we proposed another algorithm for warm-start computing of the shortest path between two nodes, and tested it. Result shows warm-start computing can save 99% of time, compared with cold-start computing of the shortest path by 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), 213–223.

[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), 345–345.

[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), 507–532.

[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).