Combining All Pairs Shortest Paths and All Pairs Bottleneck Paths Problems*

Tong-Wook Shinn and Tadao Takaoka

Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand

Abstract. We introduce a new problem that combines the well known All Pairs Shortest Paths (APSP) problem and the All Pairs Bottleneck Paths (APBP) problem to compute the shortest paths for all pairs of vertices for all possible flow amounts. We call this new problem the All Pairs Shortest Paths for All Flows (APSP-AF) problem. We firstly solve the APSP-AF problem on directed graphs with unit edge costs and real edge capacities in $\tilde{O}(\sqrt{t}n^{(\omega+9)/4}) = \tilde{O}(\sqrt{t}n^{2.843})$ time, where n is the number of vertices, t is the number of distinct edge capacities (flow amounts) and $O(n^{\omega}) < O(n^{2.373})$ is the time taken to multiply two n-by-n matrices over a ring. Secondly we extend the problem to graphs with positive integer edge costs and present an algorithm with $\tilde{O}(\sqrt{t}c^{(\omega+5)/4}n^{(\omega+9)/4}) = \tilde{O}(\sqrt{t}c^{1.843}n^{2.843})$ worst case time complexity, where c is the upper bound on edge costs.

1 Introduction

Finding the shortest paths between pairs of vertices in a graph is one of the most extensively studied problems in algorithms research. The shortest paths problem is often categorized into the Single Source Shortest Paths (SSSP) problem, which is to compute the shortest paths between one source vertex to all other vertices in the graph, and the All Pairs Shortest Paths (APSP) problem, which is to compute the shortest paths between all possible pairs of vertices on the graph.

Arguably the most famous algorithm for the APSP problem is Floyd's algorithm that runs in $O(n^3)$ time [5]. There have been many attempts at providing sub-cubic time bounds for solving the APSP problem on dense graphs with real edge costs [6,3,12,16,8,2,9], all achieving time improvements by logarithmic factors. The current best time bound is $O(n^3 \log\log n/\log^2 n)$ by Han and Takaoka [9]. If the graph has integer edge costs, faster matrix multiplication over a ring [10] can be utilized to achieve deeply sub-cubic time bounds. Alon, Galil and Margalit achieved $O(n^{(3+\omega)/2})$ time bound for solving the APSP problem on directed unweighted graphs, where $O(n^{\omega})$ is the time bound on multiplying two n-by-n matrices over a ring [1]. This time complexity translates to $O(n^{2.687})$

* This research was supported by the EU/NZ Joint Project, Optimization and its Applications in Learning and Industry (OptALI).

A. Pardo and A. Viola (Eds.): LATIN 2014, LNCS 8392, pp. 226–237, 2014.

with $\omega < 2.373$ [14]. The best time bound for this problem is currently $O(n^{2.530})$ by Zwick [15], thanks to Le Gall's recent achievement in rectangular matrix multiplication [7].

Another well studied problem in graph theory is finding the maximum bottleneck between pairs of vertices. The bottleneck of a path is the minimum capacity of all edges on the path. The bottleneck for the pair of vertices (u, v) is the maximum bottleneck of all paths from u to v. The problem of finding the paths that give the maximum bottlenecks for all pairs of vertices is formally known as the All Pairs Bottleneck Paths (APBP) problem. Vassilevska, Williams and Yuster achieved $O(n^{2+\omega/3}) = O(n^{2.791})$ time bound for solving the APBP problem on graphs with real edge capacities [13], and this has subsequently been improved to $O(n^{(\omega+3)/2}) = O(n^{2.687})$ by Duan and Pettie [4].

Let us consider the shortest path that gives us the bottleneck value of b from vertex u to vertex v. In other words, we can push flows of amounts up to b from u to v using this path. If the flow demand from u to v is less than b, however, there may be a shorter path. This information is useful if we wish to minimize the path cost (distance) for varying flow demands. Thus we combine the two well known APSP and APBP problems and compute the shortest paths for all pairs for all possible flow demands. We call this new problem the All Pairs Shortest Paths for All Flows (APSP-AF) problem. Note that this is different from the All Pairs Bottleneck Shortest Paths (APBSP) problem [13], which is to compute the bottlenecks of the shortest paths for all pairs. There are obvious practical applications for the APSP-AF problem in any form of network analysis, such as computer networks, transportation and logistics, etc.

In this paper we present two algorithms for solving the APSP-AF problem on directed graphs with positive integer edge costs and real edge capacities. Firstly we present an algorithm to solve the problem on graphs with unit edge costs in $O(\sqrt{t}n^{(\omega+9)/4}) = O(\sqrt{t}n^{2.843})$ time, where t is the number of distinct edge capacities. We then extend this algorithm to solve the problem on graphs with positive integer edge costs of at most c in $O(\sqrt{t}c^{(\omega+5)/4}n^{(\omega+9)/4}) = O(\sqrt{t}c^{1.843}n^{2.843})$ time, which is reduced to the complexity of the first algorithm when $c = 1$.

2 Preliminaries

Let $G = (V, E)$ be a directed graph with non-negative integer edge costs and real edge capacities. Let $n = |V|$ and $m = |E|$. Vertices (or nodes) are given by integers such that $\{1, 2, 3, ..., n\} \in V$. Let (i, j) denote the edge from vertex i to vertex j. Let $cost(i, j)$ denote the cost and $cap(i, j)$ denote the capacity of the edge (i, j). Let t be the number of distinct $cap(i, j)$, and let c be the upper bound on $cost(i, j)$. We define path *length* as the number of edges on the path, irrespective of their costs or capacities. We define path *cost* or *distance* as the sum of all edge costs on the path.

We represent G in a series of matrices. Let $R^\ell = \{r^\ell_{ij}\}$ be the reachability matrix, for $0 < \ell < n$, where $r^\ell_{ij} = 1$ if j is reachable from i via some path of length up to ℓ and $r^\ell_{ij} = 0$ otherwise. $r^\ell_{ii} = 1$ for all ℓ. $r^1_{ij} = 1$ if an edge

exists from i to j, and 0 otherwise. R^1 is called the adjacency matrix of G. Let $C^\ell = \{c^\ell_{ij}\}$ be the capacity matrix, where c^ℓ_{ij} represents the maximum possible capacity (or bottleneck) from i to j via any paths of lengths up to ℓ. $c^\ell_{ii} = \infty$ for all ℓ. $c^1_{ij} = cap(i,j)$ if there is an edge from i to j, and 0 otherwise. Let $D^\ell = \{d^\ell_{ij}\}$ be the distance matrix, where d^ℓ_{ij} represents the shortest possible distance from i to j via any paths of lengths up to ℓ. $d^\ell_{ii} = 0$ for all ℓ. $d^1_{ij} = cost(i,j)$ if there is an edge from i to j, and ∞ otherwise.

Let $X * Y$ denote the $(min, +)$-product and $X \star Y$ denote the (max, min)-product of the two matrices X and Y, where:

$$X * Y = \min_{k=1}^{n}\{x_{ik} + y_{kj}\} \quad X \star Y = \max_{k=1}^{n}\{\min\{x_{ik}, y_{kj}\}\}$$

Clearly the $(min, +)$-product is applicable to the distance matrix whereas the (max, min)-product is applicable to the capacity matrix.

3 Review of the Algorithm by Alon, Galil and Margalit

Our algorithm for solving the APSP-AF problem is largely based on the algorithm given by Alon *et al.* [1]. Therefore we provide a review of this algorithm using the same set of terminologies as an earlier review of the same algorithm by Takaoka [11]. The algorithm under review computes the All Pairs Shortest Distances (APSD) on directed graphs with unit edge costs. In summary this algorithm achieves sub-cubic time bound by utilizing faster matrix multiplication over a ring to perform Boolean matrix multiplication, and also using the novel idea of *Bridging Sets*.

Algorithm 1 consists of two phases. We refer to the first part of the algorithm as the *acceleration phase*, and the second part of the algorithm as the *cruising phase*. The acceleration phase repeatedly performs Boolean matrix multiplication with the adjacency matrix to compute APSD for all pairs with distances up to $\ell = r$, where r is a constant such that $1 < r < n$. Clearly this only works on graphs with unit edge costs where the path length and the path cost are equivalent. The algorithm then switches to the cruising phase where the ordinary multiplication method is used with the help of bridging sets, S_i, where S_i is a set of "via" vertices for all rows i of the distance matrix D. That is, when computing $d^\ell_{ik} + d^\ell_{kj}$ for the $(min, +)$-product, we inspect only the set of vertices in S_i for k rather than inspecting all $O(n)$ elements. Alon *et al.* have shown that with path lengths equal to r, the size of the bridging set S_i for each row i is bounded by $O(n/r)$ [1]. Hence we start the cruising phase with $|S_i| = O(n/r)$ for each row i.

The acceleration phase takes $O(rn^\omega)$ time, and the cruising phase performs repeated squaring of the distance matrix in $O(n^2 \cdot \frac{n}{r})$ time. Alon *et al.* chose to increase the path length by a factor of $\frac{3}{2}$ in each iteration of the cruising phase. This factor of $\frac{3}{2}$ is somewhat arbitrary, as any factor greater than 1 and less than 2 can be used. Because the size of the bridging set decreases by a constant factor in each iteration, we end up with a geometric series if we add up

Algorithm 1. Algorithm by Alon, Galil and Margalit

```
/* Acceleration Phase*/
for ℓ = 2 to r do
    R^ℓ ← R^{ℓ−1} × R^1 /* Boolean matrix multiplication */
    for i ← 1 to n; j ← 1 to n do
        if r^ℓ_ij = 1 and d^{ℓ−1}_ij = ∞ then d^ℓ_ij ← ℓ
        if d^{ℓ−1}_ij < ℓ then d^ℓ_ij ← d^{ℓ−1}_ij

/* Cruising Phase*/
while ℓ < n do
    ℓ' ← ⌈3ℓ/2⌉
    for i ← 1 to n do
        Scan i^th row of D^ℓ with j and find the smallest set of equal d^ℓ_ij such that
            ⌈ℓ/2⌉ ≤ d^ℓ_ij ≤ ℓ and let the set of corresponding j be S_i
    for i ← 1 to n; j ← 1 to n do
        m_ij ← min_{k∈S_i}{d^ℓ_ik + d^ℓ_kj} /* Squaring D^ℓ with S_i */
        if d^ℓ_ij ≤ ℓ then
            d^{ℓ'}_ij ← d^ℓ_ij
        else if m_ij ≤ ℓ' then
            d^{ℓ'}_ij ← m_ij
    ℓ ← ℓ'
```

the time complexities of each iteration, and hence the first squaring dominates the time complexity. The total time complexity of $O(n^{(3+\omega)/2}) = O(n^{2.687})$ of this algorithm comes from balancing the time complexities of the two phases to retrieve the best value for r, that is, setting $rn^\omega = n^2 \cdot \frac{n}{r}$ then solving for r.

4 APSP-AF on Graphs with Unit Edge Costs

We first consider solving the All Pairs Shortest Distances for All Flows (APSD-AF) problem on directed graphs with unit edge costs, that is, computing only the shortest distances rather than actual path. Path lengths and path distances are used interchangeably in this section. To re-iterate the APSD-AF problem, for each pair of vertices (i, j) for each possible flow amount, we want to compute the shortest distance. Thus our aim here is to obtain a set of (d, f) pairs for all pairs of vertices, where f is the maximum flow amount that can be pushed through the shortest path whose length (distance) is d. We refer to the distinct capacity values as *maximal flows*. i.e. there are t maximal flows. Assume that the maximal flows are sorted in increasing order. If we wish to push f such that $f_1 < f < f_2$ for consecutive maximal flows f_1 and f_2, then clearly f is represented by f_2.

Let U be a matrix such that u_{ij} is a set of (d, f) pairs as described above. Let both (d, f) and (d', f') be in u_{ij} such that $d < d'$. We keep (d', f') *iff* $f < f'$. In other words, a longer path is only useful to us if it can accommodate a greater flow. If $d = d'$, we keep the pair that provides the bigger flow. Since there can

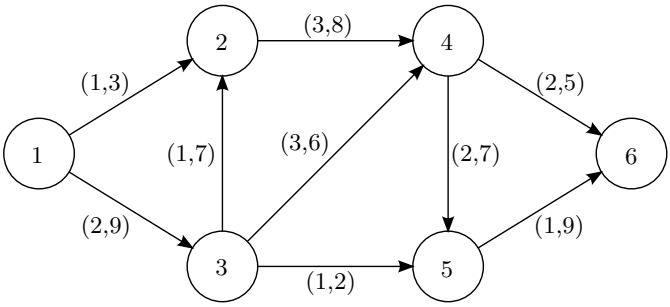

Fig. 1. An example graph with $n = 6$, $m = 9$, $t = 7$ and $c = 3$. Numbers in the parenthesis beside each edge shows the edge cost and capacity, respectively.

only be $n-1$ different values of d, each u_{ij} has at most $n-1$ pairs of (d, f). We assume the pairs are sorted in ascending order of d. We make an interesting observation here that once all (d, f) pairs for all u_{ij} are computed (i.e. the APSP-AF problem is solved), the first pairs for all u_{ij} is the solution to the APBSP problem, and the last pairs for all u_{ij} is the solution to the APBP problem.

Example 1. If the graph in Figure 1 had unit edge costs instead of the varying integer edge costs, solving APSD-AF on the graph would result in three (d, f) pairs from vertex 1 to vertex 6, that is, $u_{1,6} = \{(3, 5), (4, 6), (5, 7)\}$.

We now introduce Algorithm 2 to solve the APSD-AF problem on directed graphs with unit edge costs. Let P^f be the approximate distance matrix for shortest paths that can accommodate flows up to f. In the acceleration phase, we compute the maximum bottleneck values for all possible path lengths up to r for all pairs, where r is a constant such that $1 < r < n$. Then from the results gathered in the acceleration phase, we prepare a series of distance matrices, P^f, one for each maximal flow value f, and move onto the cruising phase where we compute the shortest distances for all pairs for all flows by repeatedly squaring each P^f.

Lemma 1. *Algorithm 2 correctly solves APSD-AF on directed graphs with unit edge costs.*

Proof. In the acceleration phase, instead of performing Boolean matrix multiplication as in Algorithm 1, we compute the (max, min)-product with the capacity matrices C^1 and $C^{\ell-1}$. After each matrix multiplication, if a path of greater capacity has been found for the vertex pair (i, j), we append the pair (ℓ, c_{ij}^{ℓ}) to u_{ij} since we have found a longer path that can accommodate a greater flow. Thus after the r^{th} iteration of the acceleration phase, all relevant (d, f) pairs for all u_{ij} are found such that $d \leq r$.

After the acceleration phase we initialize the approximate distance matrices P^f from U, one matrix for each maximal flow f, in preparation for the cruising

Algorithm 2. Solve APSD-AF on graphs with unit edge costs

```
/* Initialization for acceleration phase */
for i ← 1 to n; j ← 1 to n do
    u_ij ← φ /* φ is empty */

/* Acceleration phase */
for ℓ ← 2 to r do
    C^ℓ ← C^{ℓ−1} ⋆ C^1 /* (max, min) matrix multiplication */
    for i ← 1 to n; j ← 1 to n; i ≠ j do
        if c_ij^ℓ > c_ij^{ℓ−1} then
            Append (ℓ, c_ij^ℓ) to u_ij

/* Initialization for cruising phase */
P^f ← I for all maximal flows f /* I has 0 diagonal elements and ∞ for others */
for i ← 1 to n; j ← 1 to n; i ≠ j do
    Let u_ij = {(d_1, f_1), (d_2, f_2), ..., (d_s, f_s)} for some s /* We skip empty u_ij */
    k ← 1 /* k iterates from 1 to s */
    for all maximal flows f in increasing order do
        if f > f_k then
            k ← k + 1 /* The next d_k value is needed */
        if k > s then
            break /* We proceed to the next u_ij */
        p_ij^f ← d_k

/* Cruising phase */
for all maximal flows f do
    Perform cruising phase of Algorithm 1 on P^f

/* Finalization */
for i ← 1 to n; j ← 1 to n; i ≠ j do
    for all maximal flows f in increasing order do
        d ← p_ij^f
        Let the last pair of u_ij be x = (d', f') /* If u_ij is empty, x = φ */
        if x = φ or (f > f' and d < ∞) then
            if d = d' /* This condition is false if x = φ */ then
                Replace x with (d, f)
            else
                Append (d, f) to u_ij
```

phase. Note that if the (d, f) pair for a given flow value f does not exist in u_{ij}, we take the next pair (d', f') in u_{ij} (if one exists) and let $p_{ij}^f = d'$.

At this stage, if $p_{ij}^f < \infty$, p_{ij}^f is already the length of the shortest path from i to j that can push flow f. Thus the actual aim of the cruising phase of this algorithm is to compute the shortest distance for all other elements in P^f such that $p_{ij}^f = \infty$ at the start of the cruising phase. Note that unless G is strongly

connected, some elements of P^f will remain at ∞ until the end of the algorithm. The aim of the cruising phase is achieved by repeatedly squaring each P^f with the help of the bridging set, as proven in [1].

Retrieving sets of (d, f) pairs after the cruising phase from each resulting P^f is simply a reverse process of the initialization for the cruising phase, and thus our search for all sets of (d, f) pairs for all (i, j) is complete after finalization. □

Lemma 2. *Algorithm 2 runs in* $O(\sqrt{t}n^{(\omega+9)/4}) = O(\sqrt{t}n^{2.843})$ *worst case time.*

Proof. For the acceleration phase we use the the current best known algorithm to compute the (max, min)-product in each iteration, which gives us the time bound of $O(rn^{(3+\omega)/2})$ [4]. The time complexity for the cruising phase is $O(tn^3/r)$ since there are a total of t maximal flows, each taking $O(n^3/r)$ time to finish the computation of APSD. The time bound for the initialization for the cruising phase and the finalization is $O(tn^2)$, which is absorbed by $O(tn^3/r)$ since $n/r > 1$. We balance the time complexities of the acceleration phase and the cruising phase by setting $r = \sqrt{t}n^{(3-\omega)/4}$, and this gives us the total worst case time complexity of $O(\sqrt{t}n^{(\omega+9)/4})$. □

If $t = \Omega(n^{(\omega+1)/2})$, the value we choose for r may exceed n. In such a case, we simply stay in the acceleration phase until $r = n-1$. Thus a more accurate worst case time complexity of Algorithm 2 is actually $O(\min\{n^{(5+\omega)/2}, \sqrt{t}n^{(\omega+9)/4}\})$. For all subsequent time complexities, for simplicity, we only show the time bound for actually going into the cruising phase.

A straightforward method of solving the APSD-AF problem is to repeatedly compute APSD for each maximal flow value f using only edges that have capacities greater than or equal to f. This method is equivalent to starting the cruising phase at $r = 1$, giving us the time complexity of $O(tn^{2.530})$ if we use Zwick's algorithm to solve APSD for each maximal flow value [15]. For $t > n^{0.626}$, Algorithm 2 is faster. Note that a simple decremental algorithm where edges are removed in the reverse order of capacities while repeatedly solving APSD cannot be used to solve the APSD-AF problem because edges with larger capacities may later be required to provide shorter paths for a smaller maximal flow values.

Theorem 1. *There exists an algorithm that can solve APSP-AF on directed graphs with unit edge costs in* $\tilde{O}(\sqrt{t}n^{(\omega+9)/4})$ *worst case time.*

Proof. As noted earlier there can be $O(n)$ (d, f) pairs for each vertex pair (i, j). Since the lengths of each path can be $O(n)$, explicitly listing all paths takes $O(n^4)$ time. We get around this by modifying Algorithm 2 to extend the (d, f) pair to the (d, f, s) triplet, where s is the *successor* node, such that retrieving the actual path from (d, f, s) can be performed by simply following the successor nodes. In the acceleration phase witnesses for the (max, min)-product can be retrieved with an extra polylog factor [4], and the successor nodes can be computed from the witnesses in each iteration in $O(n^2)$ time [15]. In the cruising phase retrieving the witnesses, and hence the successor nodes, is a simple exercise since ordinary matrix multiplication is performed. Therefore extending (d, f) to (d, f, s) only takes an additional polylog factor.

The explicit path for a given flow demand from i to j can be generated in time linear to the path length as follows. Firstly we perform binary search for the triplet (d, f, s) in u_{ij} with f as the key to find the minimum distance d such that f is greater than or equal to the given flow requirement. We then traverse the successor nodes s one by one, using d to look up each subsequent successor node in $O(1)$ time. □

5 APSP-AF on Graphs with Integer Edge Costs

We now consider solving the APSD-AF problem on directed graphs with integer edge costs and real edge capacities, where the edge cost is bounded by c. Note that with integer edge costs we need to make a clear distinction between path lengths and distances. One approach for solving this problem is to use the method described in [1] to replace G with an expanded graph G' such that all edges in G' have unit edge costs, then applying the algorithm on G' to solve the problem on G. G' is created by attaching a chain of $c-1$ artificial vertices to each real vertex such that the artificial edges linking the artificial vertices in each chain have unit edge costs and capacities of ∞. We then replace each real edge (i, j) with an artificial edge with unit edge cost and capacity of $cap(i, j)$ by choosing one of the artificial vertices of i (or i itself) as the source vertex and the real vertex j as the destination, such that there exists a path from i to j with length equal to $cost(i, j)$. See Figure 2 for an illustration of how a graph is expanded. The expanded graph G' has $O(cn)$ vertices, and we can clearly solve APSD-AF on G by solving APSD-AF on G' in $O(\sqrt{t}(cn)^{(9+\omega)/4})$ time.

Example 2. Solving APSD-AF on the graph in Figure 1 results in a total of five (d, f) pairs from vertex 1 to 6, that is, $u_{(1)(6)} = \{(4, 2), (6, 3), (7, 5), (8, 6), (9, 7)\}$.

We can do better, however, with the key observation that only the acceleration phase of Algorithm 2 is restricted to graphs with unit edge costs. In other words, we can complete the acceleration phase on the expanded graph G', gather the intermediate results, and then finish off the remaining computation after contracting the graph back to G. We need care here, as the path lengths in G' are actually equivalent to the path costs in G, and the bridging sets in the cruising phase are determined from the path lengths rather than the path costs. Therefore we need to make substantial changes to Algorithm 2 to keep track of both the path lengths and the path costs of G in the acceleration phase, as well as modifying the cruising phase to correctly use the path lengths of G in determining the bridging sets.

Firstly we extend the pair (d, f) to the triplet (h, d, f), where h is the path length in G, d is the path cost in G (i.e. the path length in G') and f is the maximal flow. We introduce $U' = \{u'_{ij}\}$ where u'_{ij} is a set of triplets (h, d, f) for all pairs of vertices in G'. We omit the superscript ℓ that denotes the path length in the following matrix definitions. Let $C' = \{c'_{ij}\}$ be the capacity matrix of G' and let $W = \{w_{ij}\}$ be the witness matrix for the (max, min)-product. Let $Q^f = \{q^f_{ij}\}$ such that q^f_{ij} is the length of the path that gives the path cost

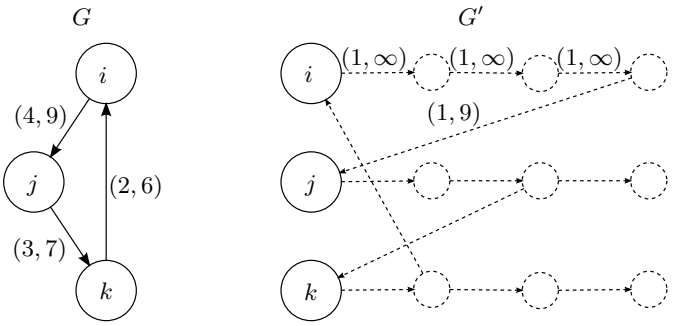

Fig. 2. Expanding G to G' with $c = 4$

(distance) of p_{ij}^f, where $P^f = \{p_{ij}^f\}$ is the distance matrix as defined in Section 4. That is, p_{ij}^f is the minimum path cost (distance) of all paths from i to j that can push flow of amount f. Note that h in the triplet (h, d, f) is no longer required once all Q^f are initialized before the start of the cruising phase.

Lemma 3. *Algorithm 3 correctly solves APSD-AF on directed graphs with non-negative integer edge costs.*

Proof. We start by creating G' from G then proceed to the acceleration phase. We only need to show that the actual path length h in the triplet (h, d, f) is correctly determined, since we have already discussed the (d, f) pairs in Section 4. What is effectively happening in the acceleration phase of Algorithm 3 is that the path length information is carried from one real vertex to the next real vertex by the artificial vertices in between. Since we are multiplying by $C'^{(1)}$ in each iteration, the witness w_{ij} will always be the vertex that comes straight before the destination vertex j on the path from i to j. That is, it is not possible for any vertices (real or artificial) to exist between $k = w_{ij}$ and j. Therefore we retrieve the last (h, d, f) triplet from u'_{ik} and increment the given h *iff* j is a real vertex. Thus the correct path length in G is given by h at the end of the acceleration phase since we are not counting the artificial vertices in the path.

The changes made to the cruising phase are to ensure that the bridging sets S_i is correctly determined from the path lengths rather than the path costs. Note that in Algorithm 2 this distinction was unnecessary because we were only considering graphs with unit edge costs. Clearly the correctness of the crusing phase remains intact by keeping $q_{ij}^{f,\ell}$ updated alongside $p_{ij}^{f,\ell}$. □

Lemma 4. *Algorithm 3 runs in* $\tilde{O}(\sqrt{t}c^{(\omega+5)/4}n^{(\omega+9)/4}) = \tilde{O}(\sqrt{t}c^{1.843}n^{2.843})$ *worst case time.*

Proof. The time complexity of the acceleration phase is $\tilde{O}(r(cn)^{(3+\omega)/2})$ since there are $O(cn)$ vertices in G'. After the r^{th} iteration in the acceleration phase, we have computed the bottleneck for all paths of lengths up to r, but this is path lengths in the expanded graph G', and not G. We divide r by c to retrieve the

Algorithm 3. Solve APSD-AF on directed graphs with non-negative integer edge costs

/* Initialization for acceleration phase */
Create G' from G /* G is expanded to G' */
for $i \leftarrow 1$ to cn; $j \leftarrow 1$ to cn **do**
  $u'_{ij} \leftarrow \phi$

/* Acceleration phase */
for $\ell \leftarrow 2$ to r **do**
  $C'^{(\ell)} \leftarrow C'^{(\ell-1)} \star C'^{1}$ /* Witnesses given as $W = \{w_{ij}\}$ */
  for $i \leftarrow 1$ to cn; $j \leftarrow 1$ to cn; $i \neq j$ **do**
    if $c'^{(\ell)}_{ij} > c'^{(\ell-1)}_{ij}$ **then**
      Let $k = w_{ij}$, and $(h, d, f) \leftarrow$ last triplet in u'_{ik} /* If empty, $h = 0$ */
      if $j \in G$ /* If j is a real vertex */ **then**
        Append $(h+1, \ell, c'^{(\ell)}_{ij})$ to u'_{ij}
      else
        Append $(h, \ell, c'^{(\ell)}_{ij})$ to u'_{ij}

/* Initialization for cruising phase, $\ell = r$ */
$U \leftarrow$rows and columns in U' for real vertices /* G' is contracted back to G */
$P^{f,\ell}, Q^{f,\ell} \leftarrow I$ for all maximal flows f
for $i \leftarrow 1$ to n; $j \leftarrow 1$ to n; $i \neq j$ **do**
  Let $u_{ij} = \{(h_1, d_1, f_1), ..., (h_s, d_s, f_s)\}$ for some s /* Skip empty u_{ij} */
  $k \leftarrow 1$ /* k iterates from 1 to s */
  for all maximal flows f in increasing order **do**
    if $f > f_k$ **then** $k \leftarrow k+1$ /* The next d_k value is needed */
    if $k > s$ **then** break /* We proceed to the next u_{ij} */
    $p^{f,\ell}_{ij} \leftarrow d_k$; $q^{f,\ell}_{ij} \leftarrow h_k$

/* Cruising phase */
for all maximal flows f **do**
  while $\ell < n$ **do**
    $\ell' \leftarrow \lceil \frac{3\ell}{2} \rceil$
    for $i \leftarrow 1$ to n **do**
      Scan i^{th} row of $Q^{f,\ell}$ with j to find the smallest set of equal $q^{f,\ell}_{ij}$ such that $\lceil \ell/2 \rceil \leq q^{f,\ell}_{ij} \leq \ell$ and let the set of corresponding j be S_i
    for $i \leftarrow 1$ to n; $j \leftarrow 1$ to n **do**
      $m_{ij} \leftarrow \min_{k \in S_i}\{p^{f,\ell}_{ik} + p^{f,\ell}_{kj}\}$
      $k \leftarrow$ the vertex that gives above m_{ij} such that $p^{f,\ell}_{ik} + p^{f,\ell}_{kj}$ is minimum
      if $m_{ij} < p^{f,\ell}_{ij}$ **then**
        $p^{f,\ell'}_{ij} \leftarrow m_{ij}; q^{f,\ell'}_{ij} \leftarrow q^{f,\ell}_{ik} + q^{f,\ell}_{kj}$
      else
        $p^{f,\ell'}_{ij} \leftarrow p^{f,\ell}_{ij}; q^{f,\ell'}_{ij} \leftarrow q^{f,\ell}_{ij}$
    $\ell \leftarrow \ell'$

/* Finalization - same as Algorithm 2 */

lower bound on the path lengths in the original graph G after the acceleration phase. Therefore the time complexity of the cruising phase is $O(tcn^3/r)$. Both the time complexities for initialization for cruising phase and finalization are again absorbed by the time complexity of the cruising phase. We balance the time complexities of the acceleration phase and the cruising phase by setting $r = \sqrt{t}c^{(-1-\omega)/4}n^{(3-\omega)/4}$, which gives us the total worst case time complexity of $\tilde{O}(\sqrt{t}c^{(\omega+5)/4}n^{(\omega+9)/4})$. □

Theorem 2. *There exists an algorithm to solve the APSP-AF problem on directed graphs with positive integer edge costs in* $\tilde{O}(\sqrt{t}c^{(\omega+5)/4}n^{(\omega+9)/4})$ *worst case time complexity.*

Proof. Clearly we can take a similar approach to the method described in the proof of Theorem 1. We can still use the path cost (distance) to look up each successor node in $O(1)$ time. We note that the witnesses in the acceleration phase can be artificial vertices, but the corresponding real vertices can be retrieved in $O(1)$ time simply by storing this information when G is expanded to G'. □

If $c = 1$, $\tilde{O}(\sqrt{t}c^{(\omega+5)/4}n^{(\omega+9)/4})$ becomes $\tilde{O}(\sqrt{t}n^{(\omega+9)/4})$, hence we have successfully generalized the APSP-AF problem from graphs with unit edge costs to graphs with integer edge costs. To compare with the straightforward method of repeatedly solving the APSP problem for each maximal flow value using Zwick's algorithm, we use the formula $\omega(1, r, 1) = 2 + (\omega - 2)(r - \alpha)/(1 - \alpha)$ where $O(n^{\omega(1,r,1)})$ is the time taken to multiply an n-by-n^r matrix with an n^r-by-n matrix, and α is a constant such that multiplying an n-by-n^α matrix with an n^α-by-n remains within the $\tilde{O}(n^2)$ time bound. We let $\omega = 2.376$ and $\alpha = 0.294$ in this comparison [15]. The time complexity of the straightforward method becomes $\tilde{O}(tn^{2+\mu})$, where $c = n^x$ such that the equation $\omega(1, \mu, 1) = 1 + 2\mu - x$ is satisfied. Clearly for relatively smaller values for c and larger values for t, Algorithm 3 is faster.

Finally we make a note that the idea of expanding the graph to G' for the acceleration phase then contracting it back to G for the cruising phase can retrospectively be applied to Algorithm 1 to give a sharper time bound than $O((cn)^{(3+\omega)/2})$, which is the time bound given by Alon *et al.* in their original paper [1]. The time bound of $O((cn)^{(3+\omega)/2})$ is sub-cubic for $c < n^{0.117}$. Using our new approach of contracting the graph after the acceleration phase, the time bound can be improved to $O(c^{(1+\omega)/2}n^{(3+\omega)/2})$, which is sub-cubic for $c < n^{0.186}$. For solving the same problem as Algorithm 1, however, other algorithms are already known that remain sub-cubic for larger values of c [11,15].

6 Concluding Remarks

The key achievements of this paper are: 1) the introduction of a new problem that clearly has numerous practical applications in network analysis involving both path costs and capacities, 2) non-trivial extension of Algorithm 1 to solve the new problem that is more complex than the APSP problem, and 3) a better method

to utilize the artificial graph for integer edge costs resulting in an improved time bound for not only our new algorithm, but also an existing algorithm for solving the APSP problem.

Solving the new APSP-AF problem on other types of graphs (e.g. undirected, real edge costs, etc) as well as finding efficient algorithms for the single source version of the problem remain on the agenda for future research.

References

1. Alon, N., Galil, Z., Margalit, O.: On the Exponent of the All Pairs Shortest Path Problem. In: Proc. 32nd FOCS, pp. 569–575 (1991)
2. Chan, T.: More algorithms for all-pairs shortest paths in weighted graphs. In: Proc. 39th STOC, pp. 590–598 (2007)
3. Dobosiewicz, W.: A more efficient algorithm for the min-plus multiplication. International Journal of Computer Mathematics 32, 49–60 (1990)
4. Duan, R., Pettie, S.: Fast Algorithms for (max,min)-matrix multiplication and bottleneck shortest paths. In: Proc. 19th SODA, pp. 384–391 (2009)
5. Floyd, R.: Algorithm 97: Shortest Path. Communications of the ACM 5, 345 (1962)
6. Fredman, M.: New bounds on the complexity of the shortest path problem. SIAM Journal on Computing 5, 83–89 (1976)
7. Le Gall, F.: Faster Algorithms for Rectangular Matrix Multiplication. In: Proc. 53rd FOCS, pp. 514–523 (2012)
8. Han, Y.: An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest paths. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 411–417. Springer, Heidelberg (2006)
9. Han, Y., Takaoka, T.: An $O(n^3 \log\log n/\log^2 n)$ Time Algorithm for All Pairs Shortest Paths. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. SWAT, pp. 131–141. Springer, Heidelberg (2012)
10. Schönhage, A., Strassen, V.: Schnelle Multiplikation Groβer Zahlen. Computing 7, 281–292 (1971)
11. Takaoka, T.: Sub-cubic Cost Algorithms for the All Pairs Shortest Path Problem. Algorithmica 20, 309–318 (1995)
12. Takaoka, T.: A faster algorithm for the all-pairs shortest path problem and its application. In: Chwa, K.-Y., Munro, J.I. (eds.) COCOON 2004. LNCS, vol. 3106, pp. 278–289. Springer, Heidelberg (2004)
13. Vassilevska, V., Williams, R., Yuster, R.: All Pairs Bottleneck Paths and Max-Min Matrix Products in Truly Subcubic Time. Journal of Theory of Computing 5, 173–189 (2009)
14. Williams, V.: Breaking the Coppersmith-Winograd barrier. In: Proc. 44th STOC (2012)
15. Zwick, U.: All Pairs Shortest Paths using Bridging Sets and Rectangular Matrix Multiplication. Journal of the ACM 49, 289–317 (2002)
16. Zwick, U.: A Slightly Improved Sub-Cubic Algorithm for the All Pairs Shortest Paths Problem with Real Edge Lengths. Algorithmica 46, 278–289 (2006)