Single Source Shortest Paths for All Flows with Integer Costs*

Tadao Takaoka

Department of Computer Science, University of Canterbury
Christchurch, New Zealand
`tad@cosc.canterbury.ac.nz`

Abstract

We consider a shortest path problem for a directed graph with edges labeled with a cost and a capacity. The problem is to push an unsplittable flow f from a specified source to all other vertices with the minimum cost for all f values. Let $G = (V, E)$ with $|V| = n$ and $|E| = m$. If there are t different capacity values, we can solve the single source shortest path problem t times for all f in $O(tm + tn \log n)$ time, which is $O(m^2)$ when $t = m$. We improve this time to $O(\min(t, cn)m + cn^2)$, which is less than $O(cmn)$ if edge costs are non-negative integers bounded by c. Our algorithm performs better for denser graphs.

1998 ACM Subject Classification E.1 Data Structures, F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases information sharing, shortest path problem for all flows, priority queue, limited edge cost, transportation network

Digital Object Identifier 10.4230/OASIcs.ATMOS.2015.56

1 Introduction

We consider a network optimization problem such that each edge has two quantities associated, that is, cost and capacity. We want to maximize a flow from a specified source vertex s to a destination vertex v with minimum cost. Here we have two objectives; flow amount and path cost. Both cannot be optimized at the same time. Let us call the minimum cost path the shortest path. We need to compute the shortest path for the given flow value f for all possible f. An example is the routing problem in a train network. Suppose f passengers want to travel together in a group from a station specified as the source vertex s to the destination station expressed by vertex v. On the way they may need to change trains at several stations. The capacity of an edge corresponds to the remaining number of seats on the train and the cost corresponds to the fare. Let d be the cost of a path from s to v and f be the flow (unsplittable) from s to v. The pair (d, f) is called a *df*-pair.

Another example is a computer network. Here vertices correspond to hub computers and edges correspond to the links. Capacities are band-widths and flows are packet sizes to be sent. It is regarded as better if packets are transmitted together to prevent packet loss and recovery.

If $d \le d'$ and $f \ge f'$, (d, f) is better than or equal to (d', f'), the latter being redundant and represented by the former. Otherwise, excluding the opposite case, they are incomparable. We only need to compute incomparable *df*-pairs.

* This research was partially supported by Optali : Optimization and its Applications in Learning and Industry, funded by EU and New Zealand Government.

© Tadao Takaoka;
licensed under Creative Commons License CC-BY
15th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'15).
Editors: Giuseppe F. Italiano and Marie Schmidt; pp. 56–67

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Similar problems in the literature are the multi (bi)-objective shortest path problem [11] and the minimum cost flow problem [1]. In the former, the two objectives are similar; additive costs over paths. In our problem, they are cost and capacity. In the latter, the flow can be split over several paths to minimize the cost. In our model, a flow cannot be divided. Unsplittable flow is studied in a few papers such as [3] and [9], in which flow amounts are considered and costs are not.

The network optimization model in this paper is simple enough to be used by network practitioners, but to the author's knowledge there is no algorithmic or theoretical analysis on this model in the literature apart from the recent [15], [16] and [14]. [15] improves the second term in the complexity of $O(tm + tn \log n)$, the first remaining $O(tm)$. This paper improves the first term. [16] and [14] deal with the all pairs shortest path for all flows (APSP-AF) problem. The complexity in this paper is better in certain combinations of t and c.

The algorithmic technique is viewed as information sharing described in [18], which solves the all pairs shortest path problem efficiently. More specifically, for a graph with n vertices, the single source shortest path problem is solved n times by changing the source n times, where they share common resources obtained in advance as preprocessing or during the course of computation. In our problem, we solve the single source shortest path problem for all flow amounts simultaneously utilizing some common data structures.

To prepare for the later development, we describe the single source shortest path problem for all flows (SSSP-AF) in the following. Let $G = (V, E)$ be a directed graph where $V = \{v_1, \cdots, v_n\}$ and $E \subseteq V \times V$. Let $|E| = m$. The cost and capacity of edge (u, v) is a non-negative real number denoted by $cost(u, v)$ and a positive real number $cap(u, v)$ respectively. We specify a vertex, s, as the source. A shortest path from s to vertex v is a path such that the sum of edge costs of this path is the minimum among all paths from s to v. The minimum cost is also called the shortest distance. The single source shortest path problem (SSSP) is to compute shortest paths from s to all other vertices.

The bottleneck (value) of a path is the minimum capacity of all edges on the path. The bottleneck of the pair of vertices (u, v) is the maximum bottleneck of all paths from u to v. Such a path is called the bottleneck path from u to v. The single source bottleneck path (SSBP) problem is to compute the bottleneck paths from s to all vertices v. The bottleneck from s to v is the maximum flow value of a simple path from s to v. Those two problems are well studied. For the bottleneck path problem the readers are referred to [13] for single source and [10] for all pairs.

If we send a smaller unsplittable flow from s to v, there may be a shorter path from s to v. Thus it makes sense to compute the shortest paths from s to v for all possible flows for all vertices v, which is called SSSP-AF. We compute a tuple of pairs (d, f), called a *df*-pair, for each v where d is the shortest distance of a path that can push f to v.

▶ **Example 1.** An example graph with solutions is given in Figure 1 with v_1 as the source. A pair of cost and capacity is attached to each edge. Also *df*-pairs are attached to each vertex as worked solutions with d's and f's in increasing order. For example, we have (3, 1)(6, 3)(7, 4) at vertex v_4. This means if we want to carry the flow amount of 4, we need to take the path of (v_1, v_3, v_4) at the cost of 7. If the flow is 3, we have a cheaper path of (v_1, v_2, v_3, v_4). If we push just 1, the path (v_1, v_2, v_4) costs us 3. If we want to push 2, this flow amount is missing from the *df*-pairs at v_4. The *df*-pair (6, 3) covers this case, meaning there is no cheaper route than for flow 3. *df*-pairs are worked out for other vertices as well. The *df*-pair (6, 3) at v_4 is obtained from (4, 3) at v_3 and the label on edge (v_3, v_4), which is $(2, 5)$, that is, 6=4+2 and 3=min{3, 5}.

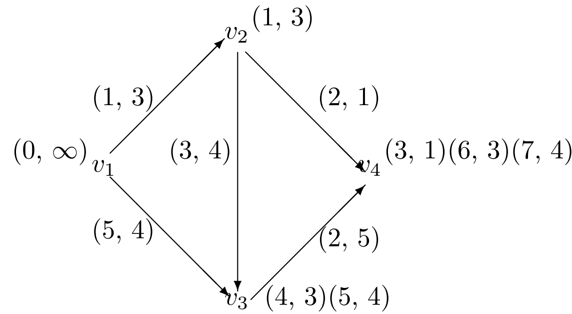

Figure 1 An example graph with incomparable *df*-pairs.

The problem can be solved by removing edges one by one. Suppose there are t different capacity values $cap_1, \ldots, cap_t$ in this order. The simplest algorithm looks like:

```
for i = 1 to t do begin
  Remove edges whose capacity is less than cap_i
  For the flow f such that cap_i ≤ f < cap_{i+1},
    solve the single source shortest path problem for the resulting graph.
end
```

If we use a Fibonacci heap [12] for Dijkstra's algorithm [8], the complexity of this algorithm becomes $O(tm + tn \log n)$, including the time for sorting capacities. This trivial upper bound is our starting point. The above algorithm works for edge costs of non-negative real numbers. In this paper, we improve the complexity when edge costs are non-negative integers bounded by a small positive constant c, achieving $O(\min\{t, cn\}m + cn^2) \leq O(cmn)$. The trivial complexity above is strongly polynomial, while our complexity is pseudo-polynomial. The point here is that we have a speed-up when c is small. When $t = O(m)$ and $m = O(n^2)$, the trivial complexity hits $O(n^4)$, called quartic, while our complexity $O(cn^3)$ can stay sub-quartic when $c = o(n)$. Similar studies are done on the all pairs shortest path problem (APSP) with integer edge costs such as [2], [17] and [20], who investigated up to what value of c we can stay in sub-cubic for the APSP complexity. The best bound for such c is $O(n^{0.624})$ if we use the Coppersmith-Winograd matrix multiplication algorithm. Recent studies improve this bound slightly.

The rest of the paper is as follows: In Sections 2 and 3, SSSP and SSBP are described in a pedagogical way so that we can see how they can be combined to solve the SSSP-AF problem. In Section 4, SSSP-AF is solved with the data structure of a one dimensional bucket system. The computational complexity of the approach described in this section is already known [15]. In Section 5, we improve the complexity in Section 4 by introducing another data structure and enhancing the one-dimensional bucket system. This section is the major contribution of the paper. In Section 6, we define the single source bottleneck path for all costs (SSBP-AC) problem. Although we can design an algorithm for this problem on its own, we show the problem can be solved as a by-product of the algorithm in Section 5. Section 7 concludes the paper. We use up-right fonts for some long names of variables and functions for readability.

2 Single source shortest path problem

We describe Dijkstra's algorithm [8] below in our style. The set S, called the solution set, is the set of vertices to which the shortest distances have been finalized by the algorithm. The

set F, called the frontier set, is the set of vertices which is outside S and can be reached from S by a single edge. We note that the distances to vertices in F can be limited to a small band when edge costs are bounded by a small integer.

Let $OUT(v) = \{w|(v, w) \in E\}$. The solution (the shortest distances from s) is in the array d at the end of the computation. To simplify presentation, only the shortest path distances are calculated, not the shortest paths. We assume all vertices are reachable from the source. Paths are given by a sequence of vertices such that for two successive vertices u and v, there is an edge (u, v). We list two invariants maintained by Algorithm 1 below.

(1) S is the set of vertices v to which shortest distances are worked out in $dist[v]$.
(2) If v is in F, $dist[v]$ is the distance of the shortest path that lies in S except for the end point v itself.

▶ **Lemma 2.** *The invariants* (1) *and* (2) *are kept through Algorithm 1.*

Proof. Lemma is true before while. Suppose (1) is true immediately after line 4. If there is a shorter path to v after line 5 via another vertex, say u, in F, which must exist to reach v, then $dist[u]$ is shorter than $dist[v]$, which is a contradiction. After v is included in S, all w in F or in $V - S - F$ are updated with the smallest possible $dist[w]$. Thus (2) is preserved. ◀

Throughout the paper, comments are given in the pseudo codes of the algorithms by the double slash for readability.

Algorithm 1

```
1.  S = ∅
2.  dist[s] = 0; dist[v] = ∞ for all v ≠ s
3.  F = {s}
4.  while F is not empty do begin
5.    v =delete-min(F) // with key dist[v]
6.    S = S ∪ {v}
7.    for w ∈ OUT(v) do
8.      if w ∉ S then
9.        if w ∈ F then dist[w] = min{dist[w], dist[v] + cost(v, w)} //decrease-key
10.         else begin dist[w] = dist[v] + cost(v, w); F = F ∪ {w} end // insert
11. end
```

At the end of computation F becomes empty and S becomes V, giving the solution in $dist$. We use a simple data structure of one-dimensional bucket system with array Q. $Q[i]$ is a list of items whose key value is i. Items in our case are vertices. We observe delete-min or delete-max operations can be done in $O(cn)$ time in total where cn is the size of Q, and decrease-key or increase-key, and insert can be done in $O(1)$ time per operation.

▶ **Theorem 3.** *Algorithm 1 solves the SSSP in $O(m + cn)$ time* [7]*.*

Proof. We use a one-dimensional bucket system for the priority queue following Dial's idea [7], where total delete-min takes $O(cn)$ time and each insertion and decrease-key takes $O(1)$ time. Suppose there are m_i edges from vertex v_i. Summation of $m_iO(1)$ gives the result, where $m = m_1 + \ldots + m_n$. ◀

3 Single source bottleneck path problem

We modify Algorithm 1 slightly for the single source bottleneck path problem. Note that we can push flow f from s to v through a path whose bottleneck value is f. The bottleneck path is sometimes called the widest path, where the capacity of an edge is viewed as the width. The solution set S and frontier set F are similarly defined.

(1) S is the set of vertices v to which maximum flows are worked out in $flow[v]$.
(2) If v is in F, $flow[v]$ is the flow of the path with the maximum flow to v that lies in S except for the end point v itself.

We use array "$flow$" instead of "$dist$" in the following. Capacities, which are non-negative real numbers, are sorted and normalized to integers $1, \dots, t, t+1$, where there are t different capacity values in the graph and $t+1$ represents ∞. Note that $t \le m$.

Algorithm 2

```
1.  S = ∅
2.  flow[s] = t + 1; flow[v] = 0 for all v ≠ s
3.  F = {s}
4.  while F is not empty do begin
5.    v =delete-max(F) // with key flow[v]
6.    S = S ∪ {v}
7.    for w in OUT(v) do
8.      if w ∉ S then
9.        if w ∈ F then flow[w] = max{flow[w], min{flow[v], cap(v, w)}} //increase-key
10.         else begin flow[w] = min{flow[v], cap(v, w)}; F = F ∪ {w} end // insert
11. end
```

▶ **Lemma 4.** *Invariants* (1) *and* (2) *are kept through the iteration in the while loop.*

Proof. Omitted. ◀

▶ **Theorem 5.** *After normalization of capacities, Algorithm 2 solves the SSBP in* $O(m+t) = O(m)$ *time.*

Proof. Omitted. ◀

4 Single source shortest paths for all flows

We parameterize Dijkstra's algorithm with the flow value f. Array $dist[v]$ is extended to $dist[v, f]$ whose intuitive meaning is the distance of the shortest path that can push f to v. The solution set S is extended to $S(f)$, meaning the solution set for the SSSP for the flow value f. Data structure Q is used for F such that items (v, f) are kept in the list at $Q[dist[v, f]]$, that is, $dist[v, f]$ is the key. The idea is to solve $t + 1$ SSSP's in parallel with the shared data structure Q.

Algorithm 3

Main data structures

$dist[v, f]$: currently shortest distance of path from source s to vertex v that can push flow f.

Q : a one-dimensional array of lists of items (v, f). If $Q[d]$ includes (v, f), $dist[v, f]$ is d. In each list the same vertex may appear more than once with different f. This part will be improved in Algorithm 4.

Pointer array indexed by (v, f) : $pointer[v, f]$ points to item (v, f) in Q. Capacities are normalized to $1, \ldots, t, t + 1$.

1. $S[f] = \emptyset$ for $f = 1, \ldots, t, t + 1$ // t+1 is for infinity
2. $dist[s, f] = 0$ for $f = 1, \ldots, t, t + 1$; Other $dist$ are initialized to ∞
3. $Q[0] = \{(s, t + 1)\}$
4. while Q is not empty do begin
5. (v, f) =delete-min(Q) // with key $dist[v, f]$
6. $S[f] = S[f] \cup \{v\}$
7. for w in $OUT(v)$ do begin
8. $d^* = dist[v, f] + cost(v, w)$ // candidate distance for w
9. $f^* = \min\{f, cap(v, w)\}$ // candidate flow for w
10. if w is not in $S[f^*]$ then
11. if $(w, f^*) \in Q$ then $dist[w, f^*] = \min\{dist[w, f^*], d^*\}$ //decrease-key
12. else begin $dist[w, f^*] = d^*; Q = Q \cup \{(w, f^*)\}$ end // insert
13. end
14. end

The following lemma is obvious.

▶ **Lemma 6.** *In Algorithm 3, we have* $f \leq f' \Rightarrow S(f) \supseteq S(f')$

We establish two assertions similar to those in the previous sections.

(1) For all v in $S[f]$, $dist[v, f]$ is the distance of the shortest path from s to v that can push flow f.

(2) For all (v, f) in Q, $dist[v, f]$ is the distance of the shortest path from s to v that can push flow f whose vertices are in $S[f]$ except for the end point v.

▶ **Lemma 7.** *The above invariants* (1) *and* (2) *are kept through iterations by the while-loop.*

Proof. The proof is based on induction overthe while-loop. Before the while-loop (1) and (2) are obviously true. Suppose there is a shorter path to v via u in Q that can push flow f at line 5. This means $dist[u, f] < dist[v, f]$, which is a contradiction to line 5 that chose $dist[v, f]$ as minimum. To push flow f^* via v after v is included in $S[f]$, we update all (w, f^*) in Q with possible shorter distances via v, or include (w, f^*) if it was outside Q with the new distance via v. Note that $S(f^*) \supseteq S(f)$ from Lemma 6, meaning the path in $S(f)$ for (w, f^*) is included in $S(f^*)$ except for w. Thus at the end of one iteration (2) is preserved. ◀

▶ **Theorem 8.** *Algorithm 3 solves SSSP-AF in* $O(tm + cn)$ *time* [15].

Proof. The correctness is seen from the fact that at the end of the algorithm the set $S(f)$ includes all v to which flow f can be pushed. The time is analysed from delete-min and decrease-key/insert. The former takes $O(cn)$. The latter takes $O(tm)$, because each vertex v_i joins $S(f)$'s at most t times and decrease-key/insert takes $O(tm_i)$ for each v_i, where

$|OUT(v_i)| = m_i$, resulting in $O(tm)$ over summation on i. Note that all (v, f) in Q are distinct so that we have at most t such (v, f)'s in Q for each v. ◀

The following monotone property is obvious and can be used for obtaining the solution, i.e., incomparable *df*-pairs in increasing order for each vertex.

▶ **Lemma 9.** *It holds for finalized distances that* $f \leq f' \Rightarrow dist[v, f] \leq dist[v, f']$.

From this lemma, we can list up incomparable *df*-pairs in increasing order for each v.

// $L[v]$ is the container of the solution for v. "||" is to append a pair to the list.
$L[v] = \phi; dist[v, 0] = \infty$ for all v
for each v do
  for $f = 1$ to t do
    if $dist[v, f] < \infty$ and $dist[v, f] \neq dist[v, f-1]$ then $L[v] = L[v]||(dist[v, f], f)$

In [4] and [6], the simple one-dimensional bucket system is generalized to the k-level cascading bucket system for SSSP. The following is a very brief sketch of the data structure. Readers interested in algorithm structures may skip to the end of this section.

▶ **Example 10.** An example of a 3-level radix-10 bucket system is given below. The initial list of keys is (7, 31, 34, 38, 56, 78, 113, 456, 477, 812, 1256, 1279).
$Base = 0, a_0 = 7, Q[0] = (\phi, \phi, \phi, \phi, \phi, \phi, \phi, (7), \phi, \phi)$
$Base = 0, a_1 = 3, Q[1] = (\phi, \phi, \phi, (31, 34, 38), \phi, (56), \phi, (78), \phi, \phi)$
$Base = 0, a_2 = 1, Q[2] = (\phi, (113), \phi, \phi, (456, 477), \phi, \phi, \phi, (812), \phi, \phi, \phi, (1256, 1279))$
After delete-min, key 7 at level 0 is deleted and for the next next delete-min, list (31, 34, 38) is re-distributed to level 0, resulting in
$Base = 30, a_0 = 1, Q[0] = (\phi, (31), \phi, \phi, (34), \phi, \phi, \phi(38), \phi)$
$Base = 0, \ \ a_1 = 5, Q[1] = (\phi, \phi, \phi, \phi, \phi, (56), \phi, (78), \phi, \phi)$
$Base = 0, \ \ a_2 = 1, Q[2] = (\phi, (113), \phi, \phi, (456, 477), \phi, \phi, \phi, (812), \phi, \phi, \phi, (1256, 1279))$
After deleting 31 for delete-min suppose we decrease key 477 to 59, that will go out of level 2 and join key 56 at level 1. Due to the nature of Dijkstra's algorithm, it will not go to a lower level than $Q[0]$.

Now the initial key value $d[v] = cost(s, v)$ is given like a radix-p number, where only x_{k-1} may exceed $p - 1$.

$$d[v] = x_{k-1}p^{k-1} + \ldots + x_1 p + x_0 \quad (0 \leq x_0, x_1, \ldots, x_{k-2} \leq p-1, \\ 0 \leq x_{k-1} \leq \lceil c/p^{k-1} \rceil - 1) \text{ for some } k.$$

The data structure has k levels of buckets. At the i-th level for each i, there are p buckets. Let i be the largest index of non-zero x_i. Item v is inserted into the x_i-th bucket at level i for all v in the frontier. During the computation, we maintain the items in the appropriate buckets based on the current value of $d[v]$. Let a_i, called the active pointer, be the smallest index of a non-empty bucket in level i. The role of a_i is to skip many empty buckets at level i. The base for level i, B_i, and the range for the j-th bucket at level i, R_j, are defined by

$$B_i = a_{k-1}p^{k-1} + \ldots + a_{i+1}p^{i+1}, R_j = [B_i + jp^i, B_i + (j+1)p^i - 1]$$

If item v is in level i for $d[v] = x_{k-1}p^{k-1} + \ldots + x_1 p + x_0$, i is the largest index such that $a_{k-1} = x_{k-1}, \ldots, a_{i+1} = x_{i+1}$ and $a_i \neq x_i$. Items move from a higher level to a lower

level and from a higher bucket to a lower bucket in the same level. The minimum can be found by scanning for a non-empty level and then the first non-empty bucket. For delete-min, the keys in this bucket are re-distributedto lower levels, finally creating a non-empty bucket at level 0.. Decrease-key can be done by moving the item in the data structure. Insert can be done by putting the item at the largest level and follow decrease-key. Suppose we solve t SSSP's. In [18], it is shown that t SSSP's can be solved in $O(tm + tn\log(c/t))$ time with this data structure. In [18], the data structure is used for the all pairs shortest path problem, where $t = n$, achieving the complexity of $O(mn + n^2\log(c/n))$. We can use the data structure for the SSSP-AF problem where t SSSP's are solved in $O(tm + tn\log(c/t))$ time. This complexity is good when c is large with a better second term, but when t is large, the first term of $O(tm)$ is outstanding. The same thing can be said of Thorup's data structure [19] that spends $O(tm + tn\log\log c)$ time when applied to our problem. We try to improve the first term in the next section.

We note at this stage that in the list $Q[d]$ for some d in the one-dimensional system, there might be items (v, f) and (v, f') such that $f \neq f'$ for some v. The following section is to prevent this duplication of items for the same v with more formalism.

5 A faster algorithm for SSSP-AF

▶ **Definition 11.** Natural order $\leq_n$ is defined on *df*-pairs, (d, f) and (d', f'), by

$$(d, f) \leq_n (d', f') \Rightarrow d \leq d' \wedge f \leq f'$$

Merit order $\leq_m$ is defined on (d, f) and (d', f') by

$$(d, f) \leq_m (d', f') \Rightarrow d' \leq d \wedge f \leq f'$$

The natural order represents a numerical order while the merit order specifies which is better for our objective. Note that both are partial orders.

▶ **Definition 12.** For v in $S[f]$, pair $(dist[v, f], f)$ is said to be Pareto optimal at v if there is no pair $(dist[v, f'], f')$ such that v is in $S(f')$ and $(dist[v, f'], f') >_m (dist[v, f], f)$. In other words, $(dist[v, f], f)$ is Pareto optimal if there is no better *df*-pair so far at v.

The priority queue Q is augmented by array $flow$, which is initialized to all 0. $flow[v, d]$ is the maximum flow so far from s to v with cost d. We maintain each list $Q[d]$ such that each v appears at most once in the list. If (v, f) is to be inserted to list $Q[d]$, where $d = dist[v, f]$, $flow[v, d]$ is consulted. If $f \leq flow[v, d]$, this insertion is ignored. If not, $(v, flow[v, d])$ is deleted from Q, (v, f) is inserted and $flow[v, d]$ is updated to f. Decrease-key(v, f) is to perform $delete(v, f)$ and $insert(v, f)$ with the new distance. We maintain pointers for each pair (v, f) to locate (v, f) in Q in $O(1)$ time. Delete-min takes $O(cn)$ time in total.

Algorithm 4

Main data structures

$dist[v, f]$: same as Algorithm 3

Q : a one-dimensional array of lists (buckets) of items (v, f). If $Q[d]$ includes (v, f), $dist[v, f]$ is d. In each list every vertex appears at most once.

$flow$: $flow[v, d]$ gives the maximum flow that can be pushed from s to v through a path in $S(f)$ except v with cost d. The size of array $flow$ is $O(cn^2)$.

Pointer array indexed by (v, f) : same as Algorithm 3

0. $dist[v, f]$ are initialized to ∞ for all $v \neq s$ and f
1. $S[f] = \emptyset$ for $f = 1, \ldots, t, t+1$; // $t+1$ is for infinity
2. $dist[s, f] = 0$ for $f = 1, \ldots, t, t+1$; $flow[v, d] = 0$ for all v and d
3. $Q[0] = \{(s, t+1)\}$
4. while Q is not empty do begin
5. (v, f) =delete-min(Q) // with key $dist[v, f]$
6. $S[f] = S[f] \cup \{v\}$
7. for w in $OUT(v)$ do begin
8. $d^* = dist[v, f] + cost(v, w)$ // candidate distance for w via v
9. $f^* = \min\{f, cap(v, w)\}$ // candidate flow for w via v
10. if w is not in $S[f^*]$ then
11. if (w, f^*) is in Q then begin
12. $dist[w, f^*] = \min\{dist[w, f^*], d^*\}$
13. decrease-key(w, f^*)
14. $flow[w, d^*] = \max\{flow[w, d^*], f^*\}$
15. end
16. else begin
17. $dist[w, f^*] = d^*$
18. insert(w, f^*)
19. $flow[w, d^*] = f^*$
20. end // if-else
21. end // for
22. end // while
23. procedure insert(w, f^*)
24. begin
25. if $f^* > flow[w, d^*]$ then begin
26. if $(w, flow[w, d^*])$ is in Q then delete$(w, flow[w, d^*])$
27. $Q = Q \cup (w, f^*)$ // insert with key $dist[w, f^*]$
28. end
29. end
30. procedure decrease-key(w, f^*)
31. begin delete(w, f^*); insert(w, f^*) end

The loop invariants (1) and (2) in the previous section hold for Algorithm 4 as well. In addition we have the following lemma, which is similar to (2).

► **Lemma 13.** *For all v and d, let $f = flow[v, d]$. If (v, f) is in Q, f is the maximum flow of the path with cost d that can push f from s to v whose vertices are in $S[f]$ except for the end point v.*

Proof. Suppose this invariant holds at the beginning of the while loop. After v is included

in $S[f]$, $flow[w, d^*]$ is updated at lines 14 and 19. Note that we have $f^* \leq f$ and thus $S[f^*] \supseteq S[f]$ from Lemma 6. Thus the path is in $S(f^*)$ except for the end point and the lemma holds for $f^* = flow[w, d^*]$ as well. ◀

▶ **Lemma 14.** *At each iteration of while loop, pair $(dist[v, f], f)$ is Pareto optimal for any $(v, f) \in S[f]$ at line 5.*

Proof. Suppose the statement is true at the beginning of each iteration. We perform one more iteration. Suppose $(dist[v, f], f)$ is not Pareto optimal for some v and f at the end of the iteration. Then for some $(dist[v, f'], f')$ we have $(dist[v, f'], f') >_m (dist[v, f], f)$, which means.

$$(dist[v, f'] < dist[v, f] \wedge f' \geq f) \vee (dist[v, f'] \leq dist[v, f] \wedge f' > f).$$

By a simple calculation, this is equivalent to

$$(dist[v, f'] < dist[v, f] \wedge f' \geq f) \vee (dist[v, f'] = dist[v, f] \wedge f' > f).$$

This contradicts the fact that $dist[v, f]$ is the distance of the shortest path that can push f to v, or the fact that (v, f) is updated (in the form of (w, f^*)) with the maximum possible f by consulting $flow[v, dist[v, f]]$. Lemma 13 guarantees f is the maximum flow to v with cost $dist[v, f]$. ◀

In the following lemma we abbreviate $(dist[v, f], f)$ as (d, f).

▶ **Lemma 15.** *All Pareto optimal df-pairs at any v can be sorted in increasing natural order. Furthermore if $(d, f) \leq_n (d', f')$, we have $d < d'$ and $f < f'$.*

Proof. If not sorted in natural order, there must be (d, f) and (d', f') at v such that $d > d'$ and $f \leq f'$ or $d \leq d'$ and $f > f'$. Then $(d, f) <_m (d', f')$ or $(d, f) >_m (d', f')$, a contradiction to Pareto optimal. The latter half can be seen as follows: Suppose there are (d, f) and (d', f') at v such that $d = d'$ or $f = f'$, which is a contradiction to Pareto optimal. ◀

▶ **Theorem 16.** *Algorithm 4 solves SSSP-AF in $O(min\{t, cn\}m + cn^2)$ time.*

Proof. Correctness is similar to that of Theorem 8. We measure the complexity by the number of accesses to major data structures. If a one-dimensional bucket system is used for Q, the total time for scanning the array for delete-min is $O(cn)$. For edge inspection at line 7, we observe pair (v, f) at line 5 brings Pareto optimal $(dist[v, f], f)$ at v. The size of the Pareto optimal solution at each v is bounded by $\min\{t, cn\}$ from the previous lemma. Thus the number of edge inspections for decrease-key and insert at line 7 is bounded by $\min\{t, cn\}m_i$ for vertex v_i. Summation over i can give us the time for decrease-key and insert being $O(\min\{t, cn\}m)$. The initialization for array $dist$, array $flow$ and Boolean arrays for membership of $S[f^*]$ and Q used at lines 10 and 11 takes $O(cn^2 + tn)$. Thus the total time is given by $O(\min\{t, cn\}m + cn + (cn + t)n) = O(\min\{t, cn\}m + (cn + t)n) = O(\min\{t, cn\}m + cn^2)$. ◀

▶ **Corollary 17.** *The SSSP-AF problem with edge costs bounded by c can be solved in $O(cmn)$ time. If the cost is a unit, it can be solved in $O(mn)$ time.*

Note. We could use the cascading bucket system to improve delete-min operations to $O(tn \log(c/t))$, but cannot improve the complexity of $O(cn^2)$ for the initialization of $flow$. It is open whether we can improve this time for initialization. In a way we improved the complexity of the first term at the higher cost of the second term, resulting in a better overall complexity for $c = o(n)$.

6 Single source bottleneck paths for all costs problem (SSBP-AC)

For every given cost d, we work out maximum flow $flow[v,d]$ that can be pushed from s to v for all v and d. Although we can design an algorithm for this problem by swapping the roles of *distance* and *flow* in Algorithm 4, Algorithm 4 already solves this problem in array $flow$. Following the monotone property of $flow$ similar to Lemma 9, the solution can be obtained by

$L[v] = \phi; flow[v,0] = 0$ for all v
for each v do
  for $d = 1$ to cn do if $flow[v,d] > 0$ then
    if $flow[v,d] \neq flow[v,d-1]$ then $L[v] = L[v]||(d, flow[v,d])$

7 Concluding remarks

Let us analyze our complexity as compared with the best known result in [15], which is $O(m^2 + mn\log(c/m))$. If $t = m$, we can say the degree of variety of the graph is high, that is, every edge has a distinct capacity. We take this case of $t = m$ following [15]. Let us define the density of the graph by $e = m/(n(n-1))$ [5], which is approximated by $e = m/n^2$. The complexity in [15] becomes $O(m^2) = O((en^2)^2)$ as the second term of the complexity becomes minor. Our complexity becomes $O(cen^3)$. Thus our complexity is better when $c \leq en$. We can say our algorithm performs well for denser graphs, i.e., with e close to 1.

When $c = O(n)$ or costs are real numbers for a dense graph, i.e., $m = O(n^2)$, the complexity is standing at $O(n^4)$ with only n as a complexity parameter. It is open whether sub-quartic is possible. There are some possibilities to extend our idea to improve time complexities for the all pairs shortest paths for all flows (APSP-AF) problem.

Acknowledgments. The author is thankful to the referees, whose careful reading and constructive comments greatly improved the quality of the paper. He also acknowledges he was greatly inspired by Tong-Wook Shinn on the subject of the research.

References

1 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.
2 N. Alon, Z. Galil, and O. Margalit. *On the exponent of the all pairs shortest path problems*. JCSS 54, 255-262, 1997.
3 Amit Chakrabarti, Chandra Chekuri, Anupam Gupta, and Amit Kumar. Approximation algorithms for the unsplittable flow problem. *Algorithmica 47(1): 53-78*, 2007.
4 B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming 73, 129-174*, 1996.
5 Thomas F. Coleman and Jorge J. More. Estimation of sparse jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis 20 (1): 187-209*, 1983.
6 E. V. Denardo and B. L. Fox. Shortest-route methods: I. reaching, pruning, and buckets. *Operations Research 27, 161-186*, 1979.
7 R. B. Dial. Algorithm 360: Shortest path forest with topological ordering. *CACM 12, 632-633*, 1969.

8 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math. 1, 269-271*, 1959.

9 Y. N. Dinits, N. Garg Y, and N. Goemans. On the single-source unsplittable flow problem. *Combinatorica, Springer, 19(1), 17-41*, 1999.

10 Ran Duan and Seth Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09), pp. 384–391*, 2009.

11 Matthias Ehrgott. *Multicriteria Optimization.* Springer-Verlag, 2005.

12 M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Jour. ACM 34, 596-615*, 1987.

13 Harold N. Gabow and Robert E. Tarjan. Algorithms for two bottleneck optimization problems. *Journal of Algorithms 9 (3): 411–417*, 1988.

14 Tong-Wook Shinn and Tadao Takaoka. Combining all pairs shortest paths and all pairs bottleneck paths problems. *LATIN 2014: 226-237*, 2014.

15 Tong-Wook Shinn and Tadao Takaoka. Combining the shortest paths and the bottleneck paths problems. *ACSC 2014: 13-18*, 2014.

16 Tong-Wook Shinn and Tadao Takaoka. Some extensions of the bottleneck paths problem. *WALCOM 2014: 176-187*, 2014.

17 Tadao Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica 20(3): 309-318*, 1998.

18 Tadao Takaoka. Sharing information for the all pairs shortest path problem. *Theor. Comput. Sci. 520: 43-50*, 2014.

19 M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *STOC03, 149-158*, 2003.

20 U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Jour. ACM, 49, 3, 289–317*, 2002.