Combining the Shortest Paths and the Bottleneck Paths Problems

Tong-Wook Shinn\textsuperscript{1} \quad Tadao Takaoka\textsuperscript{2}

Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
\textsuperscript{1} Email: tad@cosc.canterbury.ac.nz
\textsuperscript{2} Email: tong-wook.shinn@pg.canterbury.ac.nz

Abstract

We combine the well known Shortest Paths (SP) problem and the Bottleneck Paths (BP) problem to introduce a new problem called the Shortest Paths for All Flows (SP-AF) problem that has relevance in real life applications. We first solve the Single Source Shortest Paths for All Flows (SSSP-AF) problem on directed graphs with unit edge costs in $O(nm)$ worst case time bound. We then present two algorithms to solve SSSP-AF on directed graphs with integer edge costs bounded by $c$ in $O(m^2 + mn \log \left(\frac{m}{n}\right))$ time bounds. Finally we extend our algorithms for the SSSP-AF problem to solve the All Pairs Shortest Paths for All Flows (APSP-AF) problem in $O(m^2n + nc)$ and $O(m^2n + mn^2 \log \left(\frac{m}{n}\right))$ time bounds. All algorithms presented in this paper are practical for implementation.

Keywords: Shortest Paths, SP, Bottleneck Paths, BP, Single Source Shortest Paths, SSSP, All Pairs Shortest Paths, APSP

1 Introduction

The problem of finding the shortest paths between pairs of vertices on a graph is one of the most extensively studied problems in algorithm research. This problem is formally known as the Shortest Paths (SP) problem and is often categorized into the Single Source Shortest Paths (SSSP) problem and the All Pairs Shortest Paths (APSP) problem. As the names suggest, the SSSP problem is to compute the shortest paths from one single source vertex to all other vertices on the graph, and the APSP problem is to compute the shortest paths between all possible pairs on the graph. The most well known algorithm for solving the SSSP problem is the algorithm by Dijkstra\textsuperscript{3} that runs in $O(n^2)$ time, where $n$ is the number of vertices in a graph. This algorithm can be enhanced with a priority queue, and if the Fibonacci heap\textsuperscript{5} is used to implement the priority queue then the time complexity becomes $O(m + n \log n)$ where $m$ is the number of edges in the graph. If edge costs are integers bounded by $c$, then the SSSP problem can be solved in $O(n + n \log c)$ time using a complex priority queue for integers. For solving the APSP problem, the $O(n^3)$ algorithm by Floyd\textsuperscript{4} is the most well known.

If edges have capacities, the bottleneck of a path is the minimum capacity out of all edge capacities on the path. In other words, the bottleneck of a path from vertex $u$ to vertex $v$ is the maximum amount of flow that can be pushed from $u$ to $v$ down the path. Finding the paths that give the maximum bottlenecks between pairs of vertices is also a well studied problem and is formally known as the Bottleneck Paths (BP) problem. The Single Source Bottleneck Paths (SSBP) problem can be solved with a simple modification to the algorithm by Dijkstra\textsuperscript{3}. For an undirected graph the All Pairs Bottleneck Paths (APBP) problem can be solved in $O(n^2)$, which is optimal\textsuperscript{7}.

The SP and BP problems are concerned with finding the minimum or the maximum possible values. However, the shortest path may not give the biggest bottleneck, and the path that gives the maximum bottleneck may not be the shortest path. If the flow demand from a vertex to another vertex is known, then it is clearly beneficial to find the shortest path that can fully accommodate that flow demand. Thus we combine the SP and BP problems to compute the shortest paths for all possible flow amounts. We call this problem the Shortest Paths for All Flows (SP-AF) problem. As is common in graph paths problems, we divide the SP-AF problem into the Single Source Shortest Paths for All Flows (SSSP-AF) problem, and the All Pairs Shortest Paths for All Flows (APSP-AF) problem.

There are many obvious real life applications for this new problem, such as routing in computer networks, transportation, logistics, and even planning for emergency evacuations. The city of Christchurch has recently been hit by a series of strong earthquakes, mostly notably in September 2010 and in February 2011. If we consider hundreds of people evacuating from a building in such emergencies, if the amount of people (flow) can be predetermined, we can find the shortest route for all people in various locations around the building to their respective evacuation points such that the flow of people can be fully accommodated. If the flow amounts (of people) are not considered in calculating the evacuation routes, congestion may occur at various points in the building that could lead to serious accidents.

In this paper we present algorithms for solving the SSSP-AF and APSP-AF problems on directed graphs with non-negative integer edge costs and real edge capacities that are faster than the straightforward methods. We first give an algorithm to solve SSSP-AF...
on graphs with unit edge costs in $O(mn)$ time. We then present two algorithms for solving SSSP-AF on graphs with non-negative integer edge costs of at most $c$ in $O(m^2 + nc)$ and $O(m^2 + mn \log \left( \frac{n}{m} \right))$ time bounds. Finally we show that the main concepts behind the SSSP-AF algorithms can be extended to solve APSP-AF in $O(m^2 n + nc)$ and $O(m^2 n + mn^2 \log \left( \frac{m}{mn} \right))$ time bounds.

2 Preliminaries

Let $G = (V, E)$ be a directed graph with non-negative integer edge costs bounded by $c$ and non-negative real capacities. Let $n = |V|$ and $m = |E|$. Vertices are given by integers such that $\{1, 2, 3, ..., n\} \in V$. Let $(i, j)$ be the edge from vertex $i$ to vertex $j$. Let $\text{cost}(i, j)$ denote the edge cost (or distance) and let $\text{cap}(i, j)$ denote the edge capacity. Let $\text{OUT}(v)$ be the set of vertices that are directly reachable from $v$, and $\text{IN}(v)$ be the set of vertices that can directly reach $v$. There can be up to $m$ distinct capacities if all edge capacities are unique. We refer to the distinct capacities as maximal flows. Then the SP-AF problem is to solve the SP problem for all maximal flows.

For all our algorithms only comparison operations are performed on the maximal flow values. Therefore the real values of maximal flows can be mapped to integer values without any loss of generality by first sorting the maximal flows in increasing order then assigning incremental integer values starting from 1. This allows us to use maximal flow values as indexes of arrays in our algorithms.

We use the computational model that allows comparison-addition operations and random access with $O(\log n)$ bits to be performed in $O(1)$ time.

3 Single Source Shortest Paths for All Flows

We first consider solving the SP-AF problem from a source vertex $s$ to all other vertices in $G$. Initially we compute just the distances rather than actual paths then later show that the paths information can be computed in the same time bound. That is, we first solve the Single Source Shortest Distances for All Flows (SSSD-AF) problem, then show that the algorithm can be extended to solve the SSSP-AF problem with no increase in the worst case time complexity.

The SSSD-AF problem can be defined as the problem of computing the set of all $(d, f)$ pairs for each destination vertex, where $d$ is the shortest distance from $s$ and $f$ is the maximal flow value. Let $S[v]$ be the set of $(d, f)$ pairs for the destination vertex $v$. Suppose $(d, f)$ and $(d', f')$ both exist in $S[v]$ such that $d < d'$. Then we keep $(d', f')$ iff $f < f'$, that is, a longer path is only useful if it can accommodate a greater flow. If $d = d'$, we keep the pair that gives us the greater flow.

Example Solving SSSP-AF on the example graph in Figure 1 with $s = 1$ would result in $S[6] = \{(4, 2), (5, 3), (6, 5), (8, 6)\}$.

The straightforward method for solving the SSSP-AF problem is to iterate through each maximal flow $f_i$ and solving the SSSP problem for the sub-graph that only have edges with capacities $f_i$ or greater. On graphs with unit edge costs SSSP can be solved in $O(m)$ time with a simple breadth-first-search (BFS), resulting in $O(m^2)$ time bound for solving the SSSP-AF problem. On graphs with integer edge costs bounded by $c$, we can use the algorithm by Thorup [9] to solve SSSP-AF in $O(m^2 + mn \log \log c)$ time.

Note that SSSP-AF cannot be solved with a simple decremental algorithm, where edges are removed one by one in decreasing order of capacity then the connectivity of all affected vertices are checked. This method fails because edges with larger capacities may later be required to provide shorter paths for smaller maximal flows.

3.1 Unit edge costs

Algorithm 1 solves SSSP-AF in $O(mn)$ time by utilizing the fact that even though there are $O(mn)$ maximal flows, there can only be $O(n)$ paths with unique path costs for each destination vertex. As noted earlier, if multiple paths from $s$ to $v$ exists with equal path costs, we only need to keep the path that can accommodate the biggest maximal flow out of those paths. Thus for graphs with unit edge costs, even with $m$ maximal flows, the size of $S[v]$ is $O(n)$ for each $v$.

Let $B[v]$ be the bottleneck of a path from $s$ to vertex $v$. Let $D[v]$ be a possible distance from $s$ to $v$. Let $Q[i]$ be a set of vertices that may be added to Spanning Tree (SPT) at distance $i$, such that $1 \leq i \leq n - 1$, i.e. one set of vertices exists for each possible distance from $s$.

The algorithm starts off with just $s$ in the SPT and makes incremental changes to the SPT as we iterate through the maximal flow values in increasing order. The SPT is a persistent data structure and we do not build it up from scratch in each iteration. In summary, as we iterate through each maximal flow we cut nodes from the SPT that cannot accommodate the maximal flow and add the nodes back to the SPT at the shortest possible distance from $s$ (the root).

Lemma 3.1 Algorithm 1 correctly solves SSSD-AF on directed graphs with unit edge costs.

Proof By iterating from $Q[1]$ to $Q[n - 1]$ for each maximal flow, we ensure that all vertices are added to the SPT at the minimum possible distance from $s$ for the maximal flow value in the current iteration. When the minimum possible distance is found for a vertex $v$ to be added to the SPT (line 14), all potential parent nodes, $\text{IN}(v)$, are inspected to ensure that $v$ is added to the SPT such that the bottleneck from $s$ to $v$ is maximized for the given distance (line 16). Since $D[v]$ is monotonically increasing (lines 8 and 22), $v$ cannot be added to the SPT multiple times at the same distance. Thus any time a vertex $v$ is added to the SPT, $(D[v], B[v])$ can be appended to $S[v]$ as

Figure 1: An example of a directed graph with $n = 6$, $m = 9$ and $c = 3$. The first number in the parenthesis is the edge cost and the second number is the edge capacity.
Algorithm 1 Solve SSSD-AF on graphs with unit edge costs in $O(mn)$ time

1: for all $v \in V$ do
2: $B[v] \leftarrow 0, D[v] \leftarrow 0$
3: $B[\infty] \leftarrow \infty, SPT \leftarrow \emptyset$
4: for all maximal flows $f$ in increasing order do
5: for all $v \in V$ such that $B[v] < f$ do
6: if $v$ is in SPT then
7: Cut $v$ from SPT
8: $D[v] \leftarrow D[v] + 1$
9: Add $v$ to $Q[D[v]]$
10: for $i \leftarrow 1$ to $n-1$ do
11: while $Q[i]$ is not empty do
12: Remove $v$ from $Q[i]$ for all $IN(v)$ as $u$ do
13: if $D[u] = D[v] - 1$ then
14: $b \leftarrow \min(\text{cap}(u, v), B[u])$
15: if $b > B[v]$ then
16: $B[v] \leftarrow b$
17: $\text{Add } v \text{ to } SPT, u \text{ as parent}$
18: if $v$ is in SPT then
19: Append $(D[v], B[v])$ to $S[v]$
20: else
21: $D[v] \leftarrow D[v] + 1$
22: Add $v$ to $Q[D[v]]$

a $(d, f)$ pair. It follows that once we iterate through all maximal flows we have retrieved all relevant $(d, f)$ pairs.

Lemma 3.2 Algorithm 1 runs in $O(mn)$ worst case time.

Proof We perform lifetime analysis to determine the upper bound of Algorithm 1. Each vertex $v$ can be cut from the SPT and re-added to the SPT $O(n)$ times, once per each possible distance from $s$. Cutting/adding $v$ from/to the SPT takes $O(1)$ time, achieved by setting the parent of $v$ to either NULL or $u$, respectively. Therefore the total time complexity of all operations involving the SPT is $O(n^2)$. Also there are a total of $O(n^2)$ $(d, f)$ pairs. Before each vertex $v$ is added to the SPT all incoming edges $(u, v)$ are inspected. This results in $O(m)$ edges being inspected in total for the entire duration of the algorithm for each possible distance from $s$. Since there are $O(n)$ possible distances from $s$, the total time taken for edge inspection is $O(mn)$. Thus the total worst case time complexity becomes $O(n^2 + mn) = O(mn)$.

Theorem 3.3 There exists an algorithm to solve SSSP-AF on directed graphs with unit edge costs in $O(mn)$ time.

Proof There are $O(n)$ destination vertices, $O(m)$ maximal flows, and the length of each path is $O(n)$. Therefore storing all explicit paths as a solution to the SSSP-AF problem takes $O(mn^2)$, which is too expensive. As is common in graph path algorithms, we work around this problem by storing just the predecessor vertex for storing the path information. The predecessor vertex for a path from $s$ to $v$ is the vertex that comes immediately before $v$ on the path.

We extend Algorithm 1 to store the parent vertex, $u$, along the $(d, f)$ pair in line 20, i.e. we can extend the $(d, f)$ pair to the $(d, f, u)$ triplet. Then $u$ is the predecessor vertex for the shortest path from $s$ to $v$ that can accommodate flow up to $f$. By using $d$ and $u$, any explicit path can be retrieved by recursively following the predecessor vertices in time linear to the path length. Clearly the additional storage of the predecessor vertex does not increase the worst case time complexity of the algorithm.

3.2 Cascading Bucket System

Before we move onto solving the SSSP-AF problem on graphs with integer edge costs, we review the $k$-level cascading bucket system (CBS) [1] [2]. A detailed review of this data structure has also been provided by Takaoka [9].

In the $k$-level CBS the key value $d$ is given by:

$$d = x_{k-1}p^{k-1} + x_{k-2}p^{k-2} + \ldots + xp^1 + x_0$$

where $p$ is the number of buckets (or length) of each level.

Let $i$ be the largest index such that $x_i$ is non-zero. Then an element with key of $d$ is inserted into the $x_i$th bucket at level $i$. The values of $x_i$ for all $0 \leq i < k$ are calculated only once when an element is inserted, and each insertion takes $O(k)$ time.

The decrease-key operation can be performed by removing the element from the CBS in $O(1)$ time, updating the key value, then re-inserting in the same level in $O(1)$ time, or re-inserting at a lower level in $O(l)$ time where $l$ is the difference between the initial level and the new level.

The delete-min operation is more involved than the insert or the decrease-key operations. We maintain an active pointer at each level, at level $a_i$, for all $0 \leq i < k$, such that $a_i$ is the minimum index of the non-empty bucket at level $i$. $a_i = p$ means level $i$ is empty. To perform the delete-min operation, if level $0$ is not empty, we simply pick up the minimum non-empty bucket pointed to by $a_0$. If level $0$ is empty, then we find the lowest non-empty level, $j$, and re-distribute the elements in the $a_j$th bucket into level $j - 1$, then re-distribute the elements in the $a_{j-1}$th bucket into level $j - 2$, and so on, until level $0$ is non-empty. This process of repeated re-distribution from a higher level down to lower levels is referred to as cascading, hence the name for the data structure. Each delete-min takes $O(k+p)$ time if $j < k - 1$, and $O(k + M/p^{k-1})$ if $j = k - 1$, where $M$ is the maximum key value that the CBS supports.

Example Figure 2 shows an example of a 3-level CBS that can support key values up to 1399. $p = 10$ was chosen to make the example easier to understand, since it becomes straightforward to determine the correct bucket for base 10 numbers. If the element with key equal to 19 is removed from the CBS, $a_0$ becomes 10, and the next delete-min operation will trigger the cascading operation, resulting in elements in $a_1$ to be re-distributed into level 0.

3.3 Integer edge costs

In Section 3.1 we gave an algorithm to solve SSSP-AF on directed graphs with unit edge costs in $O(mn)$ time, that is faster than the straightforward method of $O(n^2)$. In this section we present two algorithms to solve SSSP-AF on directed graphs with non-negative edge costs in $O(m^2 + nc)$ and $O(m^2 + mn \log \frac{1}{\epsilon})$ time bounds. Both time bounds are faster than the $O(m^2 + mn \log \log c)$ time bound of the straightforward method for a wide range of values for $c$, $m$ and $n$, and have the added benefit of not relying on a complex data structure that is difficult to implement in real life situations. We note that Algorithm 1 can also be used to solve SSSP-AF in $O(mnc)$ time since the
maximum distance from \( s \) for any vertex for any maximal flow value is \( O(nc) \). \( O(mnc) \) is a comparatively efficient time bound for dense graphs with small \( c \).

The two time bounds of \( O(mn + nc) \) and \( O(m^2 + mn \log (\frac{c}{m})) \) actually both come from the same algorithm, Algorithm 2, but using different data structures to implement the priority queue. Algorithm 2 is a natural extension to the well-known algorithm by Dijkstra (3). For this algorithm we define the triplet \((v, d, f)\), where \( v \) is the destination vertex and \( d \) and \( f \) are equivalent to the \((d, f)\) pair as defined in Section 3. We let \( D^i[v] \) be the current shortest distance from \( s \) to \( v \) for the maximal flow value of \( f \). We let \( Q \) be the priority queue for the \((v, d, f)\) triplets with \( d \) as the key where the operations performed on \( Q \) are \text{insert}, \text{decrease-key} and \text{delete-min}. In summary, all \( O(mn) \) \((v, d, f)\) triplets are added to the priority queue, \( Q \), then as they are removed one by one, the \((d, f)\) pair is appended to \( S[v] \) if the \((d, f)\) pair is useful i.e. \( f \) is greater than any flow in existing pairs in \( S[v] \).

**Definition** \((d, f)\) in \( S[v] \) is correct if \( d \) is the shortest distance of a path that can push flows up to \( f \) from \( s \) to \( v \).

**Lemma 3.4** Algorithm 2 correctly solves SSSD-AF on directed graphs with integer edge costs.

**Proof** We provide a formal proof by induction. Let \( S \) be the set of \((v, d, f)\) such that \( S[v] \) contains the pair \((d, f)\), for all \( v \), for all \((d, f)\) pairs. Then in the beginning of each iteration:

1. The set of \((d, f)\) pairs in each \( S[v] \) are all correct i.e. all \((v, d, f)\) triplets in \( S \) are correct.
2. For any \((v, d, f)\) in \( Q \), \( d \) is the distance of the shortest path from \( s \) to \( v \) that can push \( f \) only through the path that lies in \( S \) except for the end point \( v \).

**Basis** Before the while-loop begins (line 9) all are correct, and we suppose the theorem is correct at the beginning of some iteration. Then:

**Algorithm 2** Solve SSSD-AF on graphs with integer edge costs.

1. for all \( v \neq s \in V \) do
2. for all maximal flows \( f \) do
3. \( D^f[v] \leftarrow \infty \)
4. \( Q \leftarrow \) empty
5. Add \((s, 0, \infty)\) to \( Q \)
6. for all \( v \neq s \in V \) do
7. for all maximal flow \( f \) do
8. Add \((v, \infty, f)\) to \( Q \)
9. while \( Q \) is not empty do
10. Delete \((v, d, f)\) with minimum \( d \) from \( Q \)
11. for all \( w \neq s \in OUT(v) \) do
12. \( f' \leftarrow \text{MIN}(f, \text{cap}(v, w)) \)
13. \( d' \leftarrow d + \text{cost}(v, w) \)
14. if \( d' < D^f[w] \) then
15. Let \( x = (w, D^f[w], f') \) in \( Q \)
16. \( D^f[w] \leftarrow d' \)
17. Put \( x \) in the correct position in \( Q \)
18. if \( S[v] \) is empty then
19. Append \((d, f)\) to \( S[v] \)
20. else
21. Let \((d_0, f_0)\) - last pair in \( S[v] \)
22. if \( f_0 < f \) then
23. if \( d_0 = d \) then
24. Delete \((d_0, f_0)\) from \( S[v] \)
25. Append \((d, f)\) to \( S[v] \)

1. Let \((d_0, f_0)\) be the last pair in \( S[v] \). Suppose \((d, f)\) is appended to \( S[v] \) at the end of the loop. Note that \( d \) values are sorted in increasing order in \( S[v] \). Since \( f \) is appended only when \( f \) > \( f_0 \), there can be no shorter path in \( S[v] \) that can push \( f \). Thus \((d, f)\) is appended as a correct pair.

2. We remove \((v, d, f)\) from \( Q \) in line 10. The new distance \( d' \) and flow \( f' \) from \( v \) to \( w \) are computed for all possible \( w \). If \( d' < D^f[w] \), \((v, d, f)\) now lies in the path from \( s \) to \( w \), and \((w, d', f')\) is added to \( Q \). Since \((v, d, f)\) is added to \( S[v] \) at the end of the loop, \( d' \) is the distance of the shortest path from \( s \) to \( w \) that can push \( f \) only through the path that lies in \( S \) except for the end point \( v \).

**Lemma 3.5** Algorithm 3 can run in \( O(mn) \) worst case time.

**Proof** We use a one dimensional bucket system to implement \( Q \). Insert and decrease-key operations can be performed in \( O(1) \), resulting in \( O(mn) \) time bound for both operations for the whole algorithm. The delete-min operation is performed simply by scanning through \( Q \) from \( i = 0 \) to \( nc \) one by one, where \( i \) is the distance from \( s \). We only have to scan through the distances once, and therefore the time complexity of the delete-min operation for the whole duration of the algorithm is \( O(nc) \). Each vertex can be inspected exactly once at each maximal flow value. This means \( O(mn) \) edge inspections are performed per maximal flow value, resulting in \( O(m^2) \) for the whole algorithm. Thus we have \( O(m^2 + mn + nc) = O(m^2 + nc) \) as the total worst case time complexity of Algorithm 2 using the one dimensional bucket system to implement the priority queue.

**Lemma 3.6** Algorithm 3 can run in \( O(m^2 + mn \log (\frac{m}{n})) \) worst case time.
Proof We use k-level CBS to implement Q. Since there are a total of $O(mn^2)$ triplets to be inserted into $Q$, the total time complexity for operations performed on $Q$ are: $O(kmn)$ for insert, $O(kmn)$ for decrease-key, and $O(kmn + pmn + cn/pk^{k-1})$ for delete-min, where $p$ is the length of each level. We choose $p = (\frac{n}{m})^{1/k}$ and $k = \log (\frac{n}{m})$ to implement our k-level CBS. Then the term $O(kmn)$ dominates and the total time complexity for all the three operations involving $Q$ becomes $O(mn \log (\frac{n}{m}))$. As shown in the proof of Lemma 3.6 the total time taken for edge inspection is $O(m^2)$, resulting in the total worst case time complexity of $O(m^2 + mn \log (\frac{n}{m}))$ for Algorithm 2 using the k-level CBS to implement the priority queue.

Theorem 3.7 There exists algorithms to solve the SSSP-AF problem on directed graphs with integer edge costs in $O(m^2 + nc)$ or $O(m^2 + mn \log (\frac{n}{m}))$ time bounds.

Proof We take the same approach as discussed in the proof of Theorem 3.3. In line 17 of Algorithm 2 we store $v$ as the predecessor vertex alongside the $(w, d', f')$ triplet.

Note that in the proof of Lemma 3.6 we could have chosen $k = O(\log (\frac{n}{m})/\log \log (\frac{n}{m}))$ to speed up the algorithm by a polylog factor. We also note that if $c = 1$ then Algorithm 2 has the time complexity of $O(m^2)$, thus Algorithm 1 has not been made redundant by Algorithm 2.

4 All Pairs Shortest Paths for All Flows

The key achievement in this paper is not to come up with an original data structure but to devise algorithms to successfully utilize existing well known data structures, based on the observation that the maximum distance of any simple path on a graph with integer edge costs bounded by $c$ is $O(nc)$. From this observation what we have effectively achieved is to find a method to share resources. That is, instead of having to repeatedly scan over $O(nc)$ distances for solving SSSP for each maximal flow value, we solve SSSP for all maximal flows at the same time while sharing the common resource, $Q$, thereby allowing us to scan $O(nc)$ only once. Takahashi used a similar idea to achieve $O(mn + n^2 \log (\frac{n}{m}))$ for the APSP problem. In this section we further extend the idea of sharing common resources to solve the problem of APSD-AF.

We let $D^f[u][v]$ be the currently known shortest distance from vertex $u$ to vertex $v$ for maximal flow $f$. We extend the $(v, d, f)$ triplet that was defined in Section 3.3 to the quadruple $(u, v, d, f)$, where $u$ and $v$ are the starting and the ending vertices of a possible path, respectively. Then we can extend Algorithm 2 to solve the APSD-AF problem, as shown in Algorithm 3.

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

1. for all $(u, v) \in V \times V$ do
2. for all maximal flows $f$ do
3. if $u = v$ then
4. $D^f[u][v] \leftarrow 0$
5. else
6. $D^f[u][v] \leftarrow \infty$
7. $Q \leftarrow$ empty
8. for all $v \in V$ do
9. Add $(v, v, 0, \infty)$ to $Q$
10. for all $(u, v) \in V \times V$ such that $u \neq v$ do
11. for all maximal flow $f$ do
12. Add $(u, v, \infty, f)$ to $Q$
13. while $Q$ is not empty do
14. Delete $(u, v, d, f)$ with minimum $d$ from $Q$
15. for all $w \neq u \in OUT(v)$ do
16. $f' \leftarrow \min(f, \text{cap}(v, w))$
17. $d' \leftarrow d + \text{cost}(v, w)$
18. if $d' < D^f'[u][w]$ then
19. Let $x = (u, w, D^f'[u][w], f')$ in $Q$
20. $D^f'[u][w] \leftarrow d'$
21. Put $x$ in the correct position in $Q$
22. if $S[u][v]$ is empty then
23. Append $(d, f)$ to $S[u][v]$
24. else
25. Let $(d_0, f_0) \leftarrow$ last pair in $S[u][v]$
26. if $f_0 < f$ then
27. if $d_0 = d$ then
28. Delete $(d_0, f_0)$ from $S[u][v]$
29. Append $(d, f)$ to $S[u][v]$

Lemma 4.2 Algorithm 3 can run in $O(m^2n + nc)$ worst case time.

Proof There are $O(n^2)$ pairs of vertices resulting in $O(mn^2)$ $(u, v, d, f)$ quadruples. Each vertex pair can be observed exactly once at each maximal flow value hence the number of edge inspections that occur at one maximal flow value is $O(mn)$, resulting in the total time bound of $O(m^2n)$ for edge inspections. Using the one dimensional bucket system to implement $Q$, we have $O(m^2n + mn^2 + nc) = O(m^2n + nc)$ as the worst case time complexity.

Lemma 4.3 Algorithm 3 can run in $O(m^2n + mn^2 \log (\frac{n}{m}))$ worst case time.

Proof We use the k-level cascading bucket system to implement $Q$, where the delete-min operation now takes $O(kmn^2 + pmn^2 + cn/pk^{k-1})$ time. We choose $p = (\frac{c}{mn})^{1/k}$ and $k = \log (\frac{c}{mn})$ for the total time complexity of the delete-min operation to become $O(mn^2 \log (\frac{n}{m}))$. Thus the total worst case time complexity using the cascading bucket system is $O(m^2n + mn^2 \log (\frac{n}{m}))$.

Theorem 4.4 There exists algorithms to solve the APSD-AF problem on directed graphs with integer edge costs in $O(m^2n + nc)$ or $O(m^2n + mn^2 \log (\frac{c}{mn}))$ time bounds.

Proof We take the same approach as before and modify Algorithm 3 to store $v$ as the predecessor vertex alongside the $(u, w, d', f')$ quadruple in line 21.
We have introduced a new graph path problem and provided non-trivial algorithms to solve the new problem that are both practical and faster than the straightforward methods.

The example of evacuation planning has been used in the introduction of the paper to show the relevance of the SP-AF problem in real life. Another possible application of the SSSP-AF problem is in computer networking, as a more sophisticated dynamic routing protocol than the currently commonly used protocols such as RIP and OSPF. And with the introduction of Software Defined Networking (SDN) (10), we can also propose APSP-AF as a possible algorithm to calculate the routes in the entire network.

Trivial lower bounds of $O(mn)$ and $O(mn^2)$ exist for the SSSP-AF and APSP-AF problems, respectively, on weighted digraphs. This paper has investigated only graphs with integer edge costs. Can we provide a better time bound than the straightforward $O(m^2 + mn \log n)$ for the SSSP-AF problem on directed graphs with real edge costs? Is there a faster algorithm on undirected graphs? How close can we get to the lower bounds of the SP-AF problems? We conclude the paper with these open questions and look forward to further research that may address these open problems.

References

[1] Ahuja, K., Melhorn, K., Orlin, J. B., & Tarjan, R. E. (1990), Faster algorithms for the shortest path problem, in ‘Journal of ACM’, Vol. 37, pp. 213-223
[2] Denardo, E. V., & Fox, B. L. (1979), Shortest-route methods: I. Reaching, pruning, and buckets, in ‘Operations Research’, Vol. 27, pp. 161–186
[3] Dijkstra, E. (1959), A note on two problems in connexion with graphs, in ‘Numerische Mathematik’, Vol. 1, pp. 269–271
[4] Floyd, R. (1962), Algorithm 97: Shortest Path, in ‘Communications of the ACM’, Vol. 5, pp. 345
[5] Fredman, M. (1976), New bounds on the complexity of the shortest path problem, in ‘SIAM Journal on Computing’, Vol. 5, pp. 83–89
[6] Fredman, M. & Tarjan, R. (1987), Fibonacci heaps and their uses in improved network optimization algorithms, in ‘Journal of the Association for Computing Machinery’, Vol. 34, pp. 596–615
[7] Hu, T. C. (1961), The maximum capacity route problem, in ‘Operations Research’, Vol. 9, pp. 898–900
[8] Takaoka, T. (2012), Efficient Algorithms for the All Pairs Shortest Path Problem with Limited Edge Costs, in ‘Proceeding of 18th CATS’, pp. 21–26
[9] Thorup, M. (2003), Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem, in ‘Proceeding of 35th STOC’, pp. 149-158
[10] 2012ONF Open Networking Foundation (2012), Software-Defined Networking: The New Norm for Networks, in ONF White Paper