Some Extensions of the All Pairs Bottleneck Paths Problem*

Tong-Wook Shinn and Tadao Takaoka

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

Abstract. We extend the well known bottleneck paths problem in two directions for directed unweighted (unit edge cost) graphs with positive real edge capacities. Firstly we narrow the problem domain and compute the bottleneck of the entire network in $O(n^\omega \log n)$ time, where $O(n^\omega)$ is the time taken to multiply two n-by-n matrices over ring. Secondly we enlarge the domain and compute the shortest paths for all possible flow amounts. We present a combinatorial algorithm to solve the Single Source Shortest Paths for All Flows (SSSP-AF) problem in $O(mn)$ worst case time, followed by an algorithm to solve the All Pairs Shortest Paths for All Flows (APSP-AF) problem in $O(\sqrt{d}n^{(\omega+9)/4})$ time, where d is the number of distinct edge capacities. We also discuss real life applications for these new problems.

1 Introduction

The bottleneck (capacity) of a path is the minimum capacity of all edges on the path. Thus the bottleneck is the maximum flow that can be pushed through this path. The bottleneck (capacity) of a pair of vertices (i, j) is the maximum of all bottleneck values of all paths from i to j. The bottleneck paths problems are important in various areas, such as logistics and computer networks. In this paper we consider two extensions to this well known problem on directed unweighted graphs with positive real edge capacities.

The bottleneck of the (entire) network is the minimum bottleneck out of all bottlenecks for all pairs (i, j). In this paper we introduce a simple algorithm based on binary search to show that we can compute the bottleneck of the network faster than computing the all pairs bottleneck paths. The method is based on the transitive closure of a Boolean matrix and the time complexity of the algorithm is $O(n^\omega \log n)$ where $\omega = 2.373$ [16]. This algorithm is simple but effective, and provides a good starting point for this paper.

Consider the shortest path from vertex s to vertex t that can push a flow of amount up to b. If the flow demand from s to t is less than b, however, there may be a shorter route, which is useful if one wishes to minimize the distance

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

for a given amount of flow. Thus we compute the shortest path for each possible bottleneck value. We call this problem Shortest Paths for All Flows (SP-AF). We present a non-trivial $O(mn)$ algorithm to solve the Single Source Shortest Paths for All Flows (SSSP-AF) problem, that is, computing the shortest path for all flows from one source vertex to all other vertices in the graph.

Naturally, we move onto the All Pairs Shortest Paths for All Flows (APSP-AF) problem, where we compute the shortest distances for all flows for all pairs of vertices in the graph. Note that this new problem is different from the All Pairs Bottleneck Shortest Paths (APBSP) problem [15], which is to compute the bottlenecks for all pairs shortest paths. Applying our algorithm for SSSP-AF n times gives us $O(mn^2)$. If the graph is dense, however, $m = O(n^2)$, and the time complexity becomes $O(n^4)$. We can utilize faster matrix multiplication over ring to achieve a sub-quartic time bound for dense graphs. We present an algorithm that runs in $O(\sqrt{m}n^{(\omega+9)/4})$ time. If the edge capacities are integers bounded by c, the time complexity of our algorithm becomes $O(\sqrt{min\{c,m\}}n^{(\omega+9)/4})$, and if we introduce the parameter d as the number of distinct edge capacities, we have $O(\sqrt{d}n^{(\omega+9)/4})$.

The algorithms are presented in the order of increasing complexity. Section 3 details the algorithm for solving the bottleneck of the network. In Section 4 and Section 5 we present the algorithms for solving SSSP-AF and APSP-AF, respectively. In Section 6 we further analyze our new algorithm for APSP-AF. Finally we describe some real life applications of the SP-AF problems in Section 7 before concluding the paper.

2 Preliminaries

Let $G = \{V, E\}$ be a directed unweighted graph with edge capacities of positive real numbers. Let $n = |V|$ and $m = |E|$. Vertices (or nodes) are given by integers such that $\{1, 2, 3, ..., n\} \in V$. Let $e(i, j) \in E$ denote the edge from vertex i to vertex j. Let $cap(i, j)$ denote the capacity of the edge $e(i, j)$.

We call $C = \{c_{ij}\}$, where c_{ij} represents a capacity from i to j, a capacity matrix. Let $C^\ell = c_{ij}^\ell$ be the maximum bottleneck for all paths of lengths up to ℓ from i to j. Clearly $c_{ij}^1 = cap(i, j)$ if $e(i, j) \in E$, and 0 otherwise. Let c_{ij}^* be the maximum bottleneck for all paths from i to j. We call $C^* = \{c_{ij}^*\}$ the closure of C and also refer to it as the bottleneck matrix. The problem of computing C^* is formally known as the All Pairs Bottleneck Paths (APBP) problem. For graphs with unit edge costs, the APBP problem is well studied in [15] and [6]. The complexities of algorithms given by the two papers are $O(n^{2+\omega/3}) = O(n^{2.791})$ and $O(n^{(\omega+3)/2}) = O(n^{2.687})$, respectively.

Let $Q = A \star B$ denote the (max, min)-product of capacity matrices A and B, where $Q = \{q_{ij}\}$ is given by:

$$q_{ij} = \max_{1 \le k \le n} \{\min\{a_{ik}, b_{kj}\}\}$$

Note that if all elements in A and B are either 0 or 1, this becomes Boolean matrix multiplication. If we interpret "max" as addition and "min" as multi-

plication, the set of non-negative numbers forms a closed semi-ring. Similarly, the set of matrices where the product is defined as the (max, min)-product and the sum is defined as a component-wise "max" operation also forms a closed semi-ring. Then the bottleneck matrix is given by the closure of the capacity matrix, where the closure of matrix A is defined by:

$$A^* = I + A + A^2 + A^3 + ...$$

and I is the identity matrix with diagonal elements of ∞ and non-diagonal elements of 0. Although A^* is defined by an infinite series we can stop at $n-1$. The computational complexity of computing A^* is asymptotically equal to that of the matrix product in the more general setting of closed semi-ring [1].

Similarly to the capacity matrix, we can define the distance matrix, where each element represents the distance from i to j. The problem of computing the closure of the distance matrix is formally known as the All Pairs Shortest Paths (APSP) problem. Zwick achieved $O(n^{2.575})$ time for solving APSP on directed graphs with unit edge costs [17], which has recently been improved to $O(n^{2.53})$ thanks to Le Gall's new algorithm for rectangular matrix multiplication [8].

Let $Q = A * B$ denote the $(min, +)$-product, or the distance product, of distance matrices A and B, where $Q = \{q_{ij}\}$ is given by:

$$q_{ij} = \min_{1 \le k \le n} \{a_{ik} + b_{kj}\}$$

3 The bottleneck problem of the entire network

Let *bottleneck* be the bottleneck value of the entire network. See Example 1 for an illustration. Let the capacity matrix C be defined by $c_{ij} = cap(i, j)$. One straightforward method to compute *bottleneck* would be to compute $C^* = \{c^*_{ij}\}$ for all pairs (i, j) and find the minimum among them. We can solve the problem more efficiently by a simple but effective binary search as shown in Algorithm 1.

We begin by assuming that the edge capacities are integers bounded by c. Let the threshold value t be initialized to $c/2$. Let $B = \{b_{ij}\}$ be a Boolean matrix such that $b_{ij} = 1$ if $cap(i, j) \ge t$, and 0 otherwise. Let us compute the transitive closure, B^*, of B. Then, from the equation:

$$b^*_{ij} = \Sigma\{b_{ik_1} b_{k_1 k_2} ... b_{k_r j} \mid \text{all possible paths } e(i, k_1), e(k_1, k_2), ..., e(k_r, j)\}$$

we observe that $b^*_{ij} = 1$ if and only if $b_{ik_1} = 1$, $b_{k_1 k_2} = 1$, ..., $b_{k_r j} = 1$ for some path. From this we derive that *bottleneck* $\ge t$ *iff* $b^*_{ij} > 0$ for all pairs (i, j). We repeatedly halve the possible range $[\alpha, \beta]$ for *bottleneck* by adjusting the threshold, t, through binary search.

Obviously the iteration over the while loop is performed $O(\log c)$ times. Thus the total time becomes $O(B(n) \log c)$, where $B(n)$ is the time for multiplying two n-by-n Boolean matrices. If c is large, say $O(2^n)$, the algorithm is not very efficient, taking $O(n)$ halvings of the possible ranges of *bottleneck*.

Algorithm 1 Solve bottleneck problem of the entire network

$\alpha \leftarrow 0$
$\beta \leftarrow c$
while $\beta - \alpha > 0$ **do**
    $t \leftarrow (\alpha + \beta)/2$
    for $i \leftarrow 1$ to n, $j \leftarrow 1$ to n **do**
        if $c_{ij} \geq t$ **then**
            $b_{ij} \leftarrow 1$
        else
            $b_{ij} \leftarrow 0$
    Compute B^*
    if $b^*_{ij} > 0$ for all i, j **then**
        $\alpha \leftarrow t$
    else
        $\beta \leftarrow t$
bottleneck $\leftarrow \alpha$

In this case, we sort edges in ascending order. Since there are at most m possible values of capacities, doing binary search over the sorted edges gives us $O(n^\omega \log m) = O(n^\omega \log n)$. Obviously this method also works for edge capacities of real numbers. We note that the actual bottleneck path can be obtained using the witness technique in [2] with an extra polylog factor.

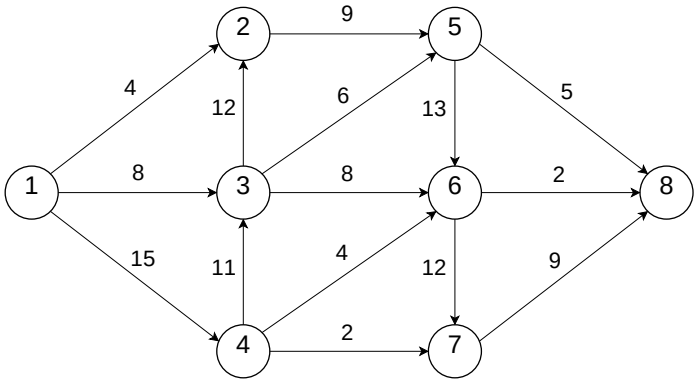

Fig. 1. An example of directed unweighted graph with edge capacities.

Example 1. The *bottleneck* of the graph in Figure 1 is 9, which is the capacity of edges $e(2, 5)$ and $e(7, 8)$. This example illustrates that the *bottleneck* of an entire network, even for simple graphs, may not be immediately obvious.

4 Single source shortest paths for all flows problem

From a source vertex s to all other vertices $v \in V$, we want to find the shortest paths for each flow value. The shortest path from s to v for a given flow value f allows us to push flows up to f as quickly as possible. For some $f' < f$, however, there may be a shorter path. Thus if we find the shortest path for all possible flows, we can respond to queries of flow demands from s to v with the quickest paths that can accommodate the flows. We observe that there can be up to m different values of f, which we refer to as the maximal flows from s to v.

A straightforward method of solving SSSP-AF is solving SSSP for each maximal flow f, that is, we repeatedly solve SSSP using only $e(u, v) \in E$ such that $cap(u, v) \geq f$, for all f. SSSP can be solved by a simple breadth-first-search (BFS) on graphs with unit edge costs, hence this approach takes $O(m^2)$ time. Each BFS will result in a shortest path spanning tree (SPT) with s as the root. Explicit paths can be retrieved by traversing up the SPTs.

One may be led to think that SSSP-AF can be solved with a simple decremental algorithm, that is, repeatedly removing edges in decreasing order of capacity, and checking for connectivity of vertices. This method, however, gives incorrect results because edges with larger capacities may later be required to provide shorter paths for smaller flows. The SP-AF problem requires solving the shortest paths problem and the bottleneck paths problem at the same time. This is not a trivial matter, as operations required to solve the two problems generally take us in opposite directions; maximizing bottlenecks comes at the cost of increased distances and minimizing distances comes at the expense of decreased bottlenecks. We have achieved $O(mn)$ worst case time for solving SSSP-AF by fully exploiting the fact that all edges have unit costs. The resulting algorithm was surprisingly simple, as shown in Algorithm 2.

Let $B[v]$ be the bottleneck of a path from s to vertex v, $L[v]$ be the possible length of the path from s to v, and let T represent the SPT. T is a kind of persistent data structure, that is, we do not compute T from scratch for each maximal flow. Let $Q[i]$ be a set of vertices that may be added to T at distance i, such that $1 \leq i \leq n-1$, i.e. one set for each possible path length from s. We iterate through each maximal flow f in increasing order. At each iteration, all $v \in V$ such that $B[v] < f$ is cut from T and added to $Q[L[v]+1]$. The key observation here is that when v gets cut from T, it is only possible for v to be re-added to T at a greater distance from s than the previous distance. For all vertices that have been cut, we attempt to add each back to T at the minimum possible path length from s for current f by emptying Q from $Q[1]$ to $Q[n-1]$. If there are many potential parent nodes at a given path length, we choose the parent that gives us the maximum bottleneck. T is organized as a linked structure with detailed operations being omitted. At each iteration we are effectively solving SSSP for the given value of f by performing incremental updates to the SPT. Therefore at the end of each iteration $L[*]$ contains the shortest distances for all destination vertices for maximal flow f, and the path can be retrieved by traversing up the SPT.

Algorithm 2 Solve single source shortest paths for all flows problem

```
1: for i ← 1 to n do
2:     B[i] ← 0, L[i] ← 0
3: B[s] ← ∞, T ← s /* T is for SPT, initially only root s */
4: for all maximal flows f in increasing order do
5:     for all v ∈ V such that B[v] < f do
6:         if v is in T then
7:             Cut v from T
8:         L[v] ← L[v] + 1
9:         Push v to Q[L[v]] /* v to be processed later */
10:    for ℓ ← 1 to n − 1 do
11:        while Q[ℓ] is not empty do
12:            Pop v from Q[ℓ]
13:            for all e(u, v) ∈ E do
14:                if L[u] = L[v] − 1 then
15:                    if MIN(cap(u, v), B[u]) > B[v] then
16:                        B[v] ← MIN(cap(u, v), B[u]) /* B[v] increased */
17:                        Add v to T with u as the parent
18:            if v is not in T then
19:                L[v] ← L[v] + 1
20:                Push v to Q[L[v]] /* v to be processed later */
```

We perform lifetime analysis of vertices in the data structure $Q[*]$ for the worst case time complexity of Algorithm 2. Each vertex v can be cut from T and be re-added to T $O(n)$ times, once per each possible path length from s. Cutting/adding v from/to T 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 T is $O(n^2)$. Before each vertex v is added to T all incoming edges $e(u, v)$ are inspected. This results in $O(m)$ edges being inspected in total for the entire duration of the algorithm for each possible path length from s. Since there are $O(n)$ possible path lengths, the total time taken for edge inspection is $O(mn)$. Even though we iterate up to $O(m)$ times, we are bounded by the fact that each vertex can only be observed at each possible path length exactly once. Therefore the total time complexity of the algorithm is $O(mn)$. Obviously APSP-AF can be solved in $O(mn^2)$ by running this algorithm n times.

Example 2. Figure 2 shows Algorithm 2 being applied on the graph in Figure 1 with $s = 1$. Initially T is created with all edges. At iteration $f = 4$, $e(4, 7)$ is cut, causing vertex 7 to be reattached to T under vertex 6. $L[7]$ is increased from 2 to 3 and $B[7]$ is increased from 2 to 8. Vertices in T will never reoccupy the shaded region, which will grow larger to the right as f increases.

5 All pairs shortest paths for all flows problem

For each pair of vertices (i, j) for each maximal flow, we want to compute the shortest path. Thus our aim here is to obtain tuples of pairs (ℓ, f) for all (i, j),

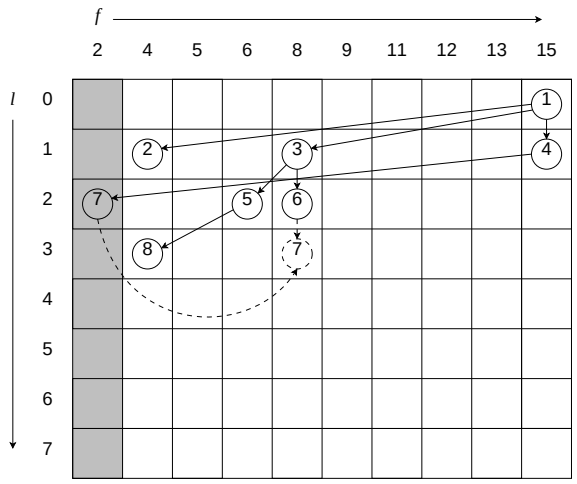

Fig. 2. Changes to the SPT at iteration $f = 4$.

where f is the maximum flow that can be pushed through a shortest path whose length is $\ell < n$. We can assume that the values of ℓ are all distinct.

Let C be the capacity matrix and let D^f be the approximate distance matrix for paths that can accommodate flows up to f. A more detailed description of D^f follows shortly in the main description of our algorithm. Let T be a matrix such that t_{ij} is a tuple of pairs (ℓ, f) as described above. Let both (ℓ, f) and (ℓ', f') be in t_{ij} such that $\ell < \ell'$. We keep (ℓ', f') *iff* $f < f'$ i.e. a longer path is only relevant if it can accommodate a greater flow. Each t_{ij} has at most $n - 1$ pairs of (ℓ, f). We assume the pairs are sorted in ascending order of ℓ. We make an interesting observation here that the set of first pairs for all t_{ij} is the solution to APBSP, and the set of last pairs for t_{ij} is the solution to APBP. For APSP-AF, all pairs (ℓ, f) for all t_{ij} are computed.

Example 3. Solving APSP-AF on the graph given in Figure 1 results in a tuple of four pairs of (ℓ, f) from vertex 4 to vertex 7, that is, $t_{47} = ((1, 2), (2, 4), (3, 8), (5, 9))$.

Algorithm 3 is largely based on the method given by Alon, Galil and Margalit in [2], which is commonly used to solve various all pairs path problems [9,13,17,15]. This method has been reviewed in [13] and we use the same set of terminologies as the review. The algorithm consists of two phases; the *acceleration* phase and the *cruising* phase. Simply speaking, we run the algorithm by Alon *et al.* for all f in parallel with a modified acceleration phase.

We compute the (max, min)-products in the acceleration phase, multiplying the capacity matrix C one by one. The ℓ^{th} iteration of the acceleration phase, therefore, finds the maximum bottleneck for all paths of lengths up to ℓ.

After the acceleration phase we initialize distance matrices D^f from T, one matrix for each maximal flow f, in preparation for the cruising phase. At this stage, d^f_{ij} is the length of the shortest path that can push flow f, if the path

Algorithm 3 Solve all pairs shortest paths for all flows problem

```
    /* initialization for acceleration phase */
 1: C^0 ← I
 2: for i ← 1 to n; j ← 1 to n do
 3:     t_ij ← φ /* φ is empty */
    /* acceleration phase */
 4: for ℓ ← 1 to r do
 5:     C^ℓ ← C^(ℓ−1) ⋆ C
 6:     for i ← 1 to n; j ← 1 to n such that i ≠ j do
 7:         f ← c_ij^ℓ
 8:         if f > c_ij^(ℓ−1) then
 9:             t_ij ← t_ij||(ℓ, f) /* append (ℓ, f) to t_ij */
    /* initialization for cruising phase */
10: for i ← 1 to n; j ← 1 to n such that i ≠ j do
11:     for all x in T[i, j] do
12:         if x ≠ φ then
13:             let x = (ℓ, f)
14:             d_ij^f ← ℓ
15:         else
16:             d_ij^f ← ∞
    /* cruising phase */
17: ℓ ← r
18: while ℓ < n do
19:     ℓ_1 ← ⌈ℓ * 3/2⌉
20:     for all maximal flow f do
21:         for i ← 1 to n do
22:             Scan i^th row of D^f with j and find the smallest set of equal d_ij^f
23:                 such that ⌈ℓ/2⌉ ≤ d_ij^f ≤ ℓ and let the set of corresponding j be S_i
24:         for i ← 1 to n; j ← 1 to n such that i ≠ j do
25:             m_ij ← min_{k∈S_i}{d_ik^f + d_kj^f}
26:             if m_ij ≤ ℓ_1 then
27:                 d_ij^f ← m_ij
28:     ℓ ← ℓ_1
    /* finalization */
29: for i ← 1 to n; j ← 1 to n such that i ≠ j do
30:     for all maximal flow f in increasing order do
31:         ℓ ← d_ij^f
32:         Let the last pair of t_ij be x = (ℓ′, f′)
33:         if x = φ or (f > f′ and ℓ < ∞) then
34:             if ℓ = ℓ′ then
35:                 Replace x with (ℓ, f)
36:             else
37:                 t_ij ← t_ij||(ℓ, f) /* append (ℓ, f) to t_ij */
```

length is r or less. In the cruising phase, we perform repeated squaring on the distance matrices with the help of the bridging set S_i. At the end of the cruising phase we thus have the shortest paths for all flows for all (i, j). Retrieving tuples of (ℓ, f) from the resulting matrix D^f is a reverse process of the initialization for the cruising phase.

Now we analyze the worst case time complexity of Algorithm 3. For the acceleration phase we use the the current best known algorithm given by Duan and Pettie [6] to compute the (max, min)-product in each iteration, giving us $O(rn^{(3+\omega)/2})$. The time complexity for the cruising phase is $O(mn^3/r)$. This is because $|S_i|$ is $O(n/r)$ as proven in [2], and no logarithmic factor is required for repeated squaring because the path length ℓ increases by a factor of $\frac{3}{2}$ in each iteration and hence the first squaring dominates the complexity. The time complexity for the initialization for the cruising phase and the finalization is $O(mn^2)$, which is absorbed by $O(mn^3/r)$ since $n/r > 1$. We balance the time complexities of the two phases by setting $r = \sqrt{m}n^{(3-\omega)/4}$, which gives us the total time complexity of $O(\sqrt{m}n^{(\omega+9)/4})$. If capacities are integers bounded by c, we only have to iterate c times in line 20, giving us $O(\sqrt{min\{c, m\}}n^{(\omega+9)/4})$.

As noted earlier there can be up to $n - 1$ (ℓ, f) pairs for each vertex pair (i, j). Since the lengths of each path can be $O(n)$, explicitly listing all paths could take $O(n^4)$ time. We get around this by extending the pair (ℓ, f) to the triplet (ℓ, f, u), where u is the successor node. In the acceleration phase witnesses can be retrieved with an extra polylog factor [6], and the successor nodes can be computed from the witnesses at each iteration in $O(n^2)$ time [17]. In the cruising phase retrieving u is a trivial exercise since ordinary matrix multiplication is performed. We can generate explicit paths in time linear to the path length by using ℓ as the index for looking up subsequent successor nodes. That is, we can still retrieve each successor node in $O(1)$ time even with $O(n)$ triplets (ℓ, f, u) for all pairs (i, j) because we know that the path length decrements by 1 as we step through each successor node.

6 Distinct edge capacities – parameter d

So far we have been assuming that the number of distinct edge capacities, d, is bounded by the number of edges, m, and used only m and n as complexity parameters. Hence the worst case time complexity of Algorithm 3 was given to be $O(\sqrt{m}n^{(\omega+9)/4})$. If we compare this with $O(mn^2)$ given by Algorithm 2, and $O(n^{(5+\omega)/2})$ given by simply staying in the acceleration phase of Algorithm 3 until $r = n$, then we observe that for dense graphs Algorithm 3 is faster than $O(mn^2)$, and for sparse graphs Algorithm 3 is faster than $O(n^{(5+\omega)/2})$.

As we will discuss further in Section 7, d may not be related to m, especially for dense graphs where $m = O(n^2)$. Therefore we incorporate d directly into the time complexity of Algorithm 3 to give $O(\sqrt{d}n^{(\omega+9)/4})$ and the merit of Algorithm 3 emerges with dense graphs having relatively small number of maximal flows. Another straightforward method of solving APSP-AF on graphs with d distinct edge capacities is to compute the $(min, +)$-closure d times. Us-

ing Zwick's algorithm in [17], the time complexity of this method is $O(dn^{2.53})$. Clearly, for most values of d, Algorithm 3 is faster.

It is well known that algorithms that utilize faster matrix multiplication over ring is not practical to be used on modern day computers [10] and are mostly of theoretical importance. We therefore highlight that Algorithm 3 can be easily turned into a practical algorithm by computing (max, min)-product in the acceleration phase using the naive approach. This results in the time complexity of $O(\sqrt{d}n^3)$, which is still very much useful for dense graphs alongside our combinatorial algorithm of $O(mn^2)$ that is better suited for sparse graphs.

7 Real life applications of APSP-AF

Computer networks can be accurately modeled by unweighted directed graphs with edge capacities, by representing each hop (e.g. router) as a vertex, each network link as an edge, and the bandwidth of each link as edge capacities. However, routing protocols that are commonly used today are based on less accurate models. For example, the Routing Information Protocol (RIP) computes routes based solely on the hop counts, while the Open Shortest Path First (OSPF) protocol, by default, computes routes based solely on the bandwidths.

In today's computer networks each router is autonomous, and therefore each router computes SSSP. RIP is often implemented with Bellman-Ford algorithm [7,3] and OSPF is often implemented with Dijkstra's algorithm [5]. We present SSSP-AF as a better solution that uses both the hop count and the bandwidth at the same time. Advanced routers are able to gather information such as the current flow amount from one IP subnet to another. With SSSP-AF, a router can make a better routing decision for a given flow based on the flow amount by choosing a route that minimizes the latency without causing congestion.

Furthermore, we introduce APSP-AF as a potential routing algorithm for Software Defined Networking (SDN) [18]. SDN is a new paradigm in computer networking where routers are no longer autonomous and the whole network can be controlled at a centralized location. The central controller has in-depth knowledge of the network and as a result SDN can benefit from more sophisticated routing algorithms. By solving APSP-AF for the whole network, the fastest routes can be determined for all flow requirements for all sources and destinations. As noted in Section 6, Algorithm 3 can be easily turned into a practical $O(\sqrt{d}n^3)$ algorithm. This is very much relevant in real life computer networks where distinct bandwidth values are defined (e.g. 100Mbps, 1Gbps).

8 Concluding remarks

We have extended the well known bottleneck paths problems, introducing new problems that have real life applications. We provided non-trivial algorithms to solve the problems more efficiently than straightforward methods.

This paper only considered directed unweighted graphs. In enterprise computer networks, most links are bi-directional, meaning undirected graphs are

adequate for modeling those networks. Also for computer networks involving low latency switches and long cables with repeaters, introducing edge costs may enable more accurate modeling of the networks. Hence solving the SP-AF problem on other types of graphs would not only be a natural extension to this paper, but also have relevance in real life.

Trivial lower bounds of $\Omega(n^2)$ and $\Omega(n^3)$ exist for SSSP-AF and APSP-AF, respectively. Most current researches in all pairs paths problems focus on breaking the cubic barrier of $O(n^3)$ to get closer to the trivial lower bound of $\Omega(n^2)$. With the APSP-AF problem we have effectively shifted the focus in time complexities from "cubic-to-quadratic" to "quartic-to-cubic". We have thus opened up a new area of research, where we hope many new contributions would occur in the future.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. N. Alon, Z. Galil and O. Margalit: On the Exponent of the All Pairs Shortest Path Problem. Proc. 32nd IEEE FOCS (1991) pp. 569–575
3. R. Bellman: On a Routing Problem. Quart. Appl. Math. 16 (1958) pp. 87–90
4. D. Coppersmith and S. Winograd: Matrix Multiplication Via Arithmetic Progressions. Journal of Symbolic Computation 9 (1990) pp. 289–317
5. E. Dijkstra: A Note on Two Problems in Connexion With Graphs. Numerische Mathematik 1 (1959) pp. 269–271
6. R. Duan and S. Pettie: Fast Algorithms for (max,min)-matrix multiplication and bottleneck shortest paths. Proc. 19th SODA (2009) pp. 384–391
7. L. Ford: Network Flow Theory. RAND Paper (1956) pp. 923
8. F. Le Gall: Faster Algorithms for Rectangular Matrix Multiplication. Proc. 53^{rd} FOCS (2012) pp. 514–523
9. Z. Galil and O. Margalit: All Pairs Shortest Paths for Graphs with Small Integer Length Edges. Journal of Computer and System Sciences 54 (1997) pp. 243–254
10. S. Robinson: Toward an Optimal Algorithm for Matrix Multiplication. SIAM News 38, 9 (2005)
11. A. Schönhage and V. Strassen: Schnelle Multiplikation Groβer Zahlen. Computing 7 (1971) pp. 281–292
12. R. Seidel: On the all-pairs-shortest-path problem. Proc. 24th ACM STOC (1990) pp. 213–223
13. T. Takaoka: Sub-cubic Cost Algorithms for the All Pairs Shortest Path Problem. Algorithmica 20 (1995) pp. 309–318
14. T. Takaoka: Efficient Algorithms for the 2-Center Problems. ICCSA 2 (2010) pp. 519–532
15. V. Vassilevska, R. Williams, R. Yuster: All Pairs Bottleneck Paths and Max-Min Matrix Products in Truly Subcubic Time. Journal of Theory of Computing 5 (2009) pp. 173–189
16. V. Williams: Breaking the Coppersmith-Winograd barrier. STOC (2012)
17. U. Zwick: All Pairs Shortest Paths using Bridging Sets and Rectangular Matrix Multiplication. Journal of the ACM 49 (2002) pp. 289–317
18. Open Networking Foundation: Software-Defined Networking: The New Norm for Networks ONF White Paper (2012)