2013 IEEE 54th Annual Symposium on Foundations of Computer Science

A forward-backward single-source shortest paths algorithm

David B. Wilson
Microsoft Research, Redmond
David.Wilson@microsoft.com

Uri Zwick
Tel Aviv University
zwick@tau.ac.il

Abstract*—We describe a new *forward-backward* variant of Dijkstra's and Spira's Single-Source Shortest Paths (SSSP) algorithms. While essentially all SSSP algorithm only scan edges *forward*, the new algorithm scans some edges *backward*. The new algorithm assumes that edges in the out-going and incoming adjacency lists of the vertices appear in non-decreasing order of weight. (Spira's algorithm makes the same assumption about the out-going adjacency lists, but does not use incoming adjacency lists.) The running time of the algorithm on a complete directed graph on n vertices with independent exponential edge weights is $O(n)$, with very high probability. This improves on the previously best result of $O(n \log n)$, which is best possible if only forward scans are allowed, exhibiting an interesting separation between forward-only and forward-backward SSSP algorithms. As a consequence, we also get a new all-pairs shortest paths algorithm. The expected running time of the algorithm on complete graphs with independent exponential edge weights is $O(n^2)$, matching a recent result of Peres *et al*. Furthermore, the probability that the new algorithm requires more than $O(n^2)$ time is *exponentially* small, improving on the *polynomially* small probability of Peres *et al.

***Keywords*-graph algorithms; shortest paths;**

I. INTRODUCTION

A. Shortest Paths

The *Single-Source Shortest Paths* (SSSP) problem, which calls for the computation of a tree of shortest paths from a given vertex in a directed or undirected graph with non-negative edge weights, is one of the most important and most studied algorithmic graph problems. The classical algorithm of Dijkstra [1], implemented with an appropriate priority queue data structure, e.g., Fibonacci heaps (Fredman and Tarjan [2]), solves the problem in $O(m + n \log n)$ time, where m is the number of edges and n is the number of vertices in the graph. For *undirected* graphs with non-negative *integer* edge weights, Thorup [3] obtained an $O(m+n)$-time algorithm.

The running time of Dijkstra's algorithm is almost linear in the size of the input graph, and the running time of Thorup's algorithm is linear. Can we hope for a *sublinear* time algorithm for the SSSP problem? In general the answer is of course "no", as an SSSP algorithm must examine essentially all the edges of the graph. There are, however, some interesting settings in which the input graph undergoes an initial *preprocessing* phase after which it may be possible to solve the SSSP problem in sublinear time. We consider here a particularly simple such preprocessing phase that sorts the edges in the adjacency lists of the vertices of the graph in non-decreasing order of weight.

Work of Uri Zwick supported by grant no. 2012338 of the United-States–Israel Binational Science Foundation (BSF).

The *All-Pairs Shortest Paths* (APSP) problem calls for the solution of the SSSP problem from every vertex of the input graph. It can clearly be solved in $O(mn + n^2 \log n)$ time by running Dijkstra's algorithm from every vertex. Pettie [4] improved this running time to $O(mn + n^2 \log \log n)$. The problem can be solved in $O(mn)$ time, if the graph is undirected and the edge weights are integral, by running Thorup's [3] algorithm from each vertex.

B. Average case results

Many authors considered the *average case* complexity of the SSSP and APSP problems. Perhaps the simplest setting for such studies is the case of a complete directed graph on n vertices in which the weight of each edge is an independent *exponential* random variable. We denote this probabilistic model by $\mathcal{K}_n(\text{EXP}(1))$. Hassin and Zemel [5] and Frieze and Grimmett [6] gave simple algorithms that solve the APSP problem, when the input graph is drawn from $\mathcal{K}_n(\text{EXP}(1))$, in $O(n^2 \log n)$ expected time.

Spira [7] initiated the study of the expected running time of SSSP and APSP algorithms in a much more general probabilistic model, now referred to as the *end-point independent* model. The input graph in this model is a complete directed graph on n vertices. Each vertex v has a (deterministic or stochastic) process that generates $n-1$ non-negative edge weights. These edge weights are randomly permuted and assigned to the out-going edges of v. The process associated with each vertex is *arbitrary*; different vertices may have different processes. Spira [7] gave an APSP algorithm whose expected running time in this model is $O(n^2 \log^2 n)$. Spira's algorithm first applies the sorting preprocessing step described above and then solves each SSSP problem in $O(n \log^2 n)$ expected time.

Spira's result was improved by several authors. Takaoka and Moffat [8] improved the running time to $O(n^2 \log n \log \log n)$. Bloniarz [9] improved it to $O(n^2 \log n \log^* n)$. Finally, Moffat and Takaoka [10] and Mehlhorn and Priebe [11] (see also recent simplifications by Takaoka and Hashim [13], [12]) improved the running time to $O(n^2 \log n)$. All these algorithms, like Dijkstra's and Spira's algorithms, use only the *out-going* adjacency lists of the graph. They all start by sorting the out-going adjacency lists and then running an SSSP algorithm from each vertex. The fastest algorithms above solve each SSSP problem in $O(n \log n)$ expected time. Mehlhorn and Priebe [11] showed that for the endpoint independent model, $\Omega(n \log n)$ expected time is best possible for algorithms that can only access the (sorted) out-going adjacency lists of the graph.

Peres *et al.* [14] recently revisited the more basic $\mathcal{K}_n(\text{EXP}(1))$ model, in which edge weights are independent exponential random variables, and obtained an APSP algorithm whose expected running in this setting is $O(n^2)$, which is clearly optimal. This algorithm is markedly different from all algorithms mentioned above as it does not sort the edge weights and then solve an SSSP problem from each vertex. Rather, it finds all distances in the graph "simultaneously", by running a static version of the dynamic APSP algorithm of Demetrescu and Italiano [15].

C. A new SSSP algorithm

Can the SSSP problem be solved in $O(n)$ time, assuming that the edge weights are independent exponential random variables and that the adjacency lists of the graph are given in sorted order? Adapting the argument of Mehlhorn and Priebe [11] we show that any SSSP algorithm that can only access the (sorted) out-going adjacency lists of the input graph, which is assumed to be drawn from $\mathcal{K}_n(\text{EXP}(1))$, must examine $\Omega(n \log n)$ edges, with high probability. As it is not clear how incoming adjacency lists could be used to speed up SSSP algorithms, this seems to give a negative answer to the question.

Surprisingly, we show however that the SSSP problem on $\mathcal{K}_n(\text{EXP}(1))$ *can* be solved in $O(n)$ time, with very high probability, beating the above $\Omega(n \log n)$ lower bound, by a *forward-backward* algorithm that uses both the out-going and incoming (sorted) adjacency lists of the input graph. Although we only analyze our new algorithm in an ideal probabilistic model, we believe that suitable variants of the new algorithm may be used to speed up SSSP computations in more realistic settings.

We develop the new $O(n)$ time SSSP algorithm in two steps. In the first step, which is by far the more challenging step, and where most of the novelty in this paper lies, we devise and analyze an SSSP algorithm that *scans* (or *examines*) only $O(n)$ edges of the graph, with very high probability. On average, the algorithm examines a constant number of edges incident to each vertex. As explained, it is crucial here that the algorithm is allowed to examine both the out-going and incoming adjacency lists of each vertex. In the second, and more standard step, we show that the algorithm can be implemented to run in $O(n)$ time. On average, the algorithm is only allowed to perform a constant number of operations per edge examined. As essentially all Dijkstra-like SSSP algorithms, our new algorithm uses a *priority-queue* data structure. To get an $O(n)$-time implementation we need to perform priority-queue operations in $O(1)$ expected amortized cost. We show that this is possible in our setting using relatively simple *bucket-based* priority queues.

D. A new APSP algorithm

One setting in which the sortedness assumption of the adjacency lists may be justified is that of solving the APSP problem. In the APSP setting, we can afford to spend $O(n^2)$ time on bucket-sorting the adjacency lists. We can then solve an SSSP problem from each vertex of the graph in $O(n)$ time, getting an $O(n^2)$-time algorithm for solving the APSP problem on $\mathcal{K}_n(\text{EXP}(1))$. This matches the recent result of Peres *et al.* [14]. The new $O(n^2)$-time APSP algorithm is very different from the algorithm of Peres *et al.* [14], which does not simply run an SSSP algorithm from each vertex. Furthermore, while the APSP algorithm of [14] runs in $O(n^2)$ expected time, it was only shown in [14] that the probability that it requires more than $O(n^2)$ time is $O(n^{-1/26})$. We show here, on the other hand, that the probability that our new algorithm requires more than $O(n^2)$-time is *exponentially* small, specifically it is at most $\exp(-\Theta(n/\log n))$.

E. On the probabilistic model

For simplicity and concreteness, we stated all our results in the $\mathcal{K}_n(\text{EXP}(1))$ probabilistic model. Exponential edge weights are convenient to work with due to their memoryless property. However, the assumption that edge weights are drawn from an exponential distribution can be greatly relaxed. In the full version of the paper we show that the expected number of edges scanned by the algorithm is still $O(n)$ when edge weights are drawn from a *uniform* distribution on $[0, 1]$ and when edge weights are *powers* of exponential random variables, i.e., of the form $\text{EXP}(1)^s$, where $0 < s \leq 1$. (We conjecture that the same is true also when $s > 1$.)

F. Related results

Bidirectional algorithms, which perform a forward search from a source vertex and a backward search from a target vertex, can be used to efficiently find a shortest path between a given pair of vertices (see, e.g., Nicholson [16] and Pohl [17]). Luby and Ragde [18] used such a bidirectional algorithm to show that a shortest path from a given source to a given target in $\mathcal{K}_n(\text{EXP}(1))$ can be found in $O(\sqrt{n} \log n)$ expected time. (They again assume, of course, that the adjacency lists are given in sorted order.) However, the bidirectional search technique does not seem to be applicable for the SSSP problem where distances to all vertices are sought. (Where do we start the backward search from?)

Our new *forward-backward* algorithm is *not* bidirectional. It is a Dijkstra-like unidirectional algorithm that uses some backward scans.

Meyer [19], Hagerup [20] and Goldberg [21] obtained SSSP algorithms with an expected running time of $O(m)$. The m-edge directed input graph may be arbitrary but its edge weights are assumed to be chosen at random from a common non-negative probability distribution. When the edge weights are independent, the running time of these algorithms is $O(m)$ with high probability. Our result differs considerably from these results. Our algorithm runs in $O(n)$ time, which is $o(m)$, on a complete graph with $m = \Omega(n^2)$ edges.

G. Organization of paper

The rest of this paper is organized as follows. In the next section we briefly review the classical algorithms of Dijkstra and Spira which form the basis of our new algorithms. Before presenting our improved algorithm for finding shortest paths, we present in Section III an improved algorithm for *verifying* that a given tree is indeed a tree of shortest paths. This helps us explain the ideas behind the improved algorithm for finding shortest paths in the simplest possible setting. In Section IV we then present our improved forward-backward shortest paths algorithm. The probabilistic analysis of the new algorithm is given in Section V. In Section VI we describe an efficient bucket-based implementation of the priority queues used by our new algorithm. Due to lack of space, many of the proofs are deferred to the full version of the paper.

II. The algorithms of Dijkstra and Spira

In this section we review the classical SSSP algorithms of Dijkstra and Spira and set the stage for the description of our new SSSP algorithm.

A. Dijkstra's algorithm

We start with a brief review Dijkstra's algorithm [1] for finding a tree of shortest paths from a given source vertex s in a directed graph $G = (V, E)$ with a non-negative weight (or cost) function $c : E \to \mathbb{R}^+$ defined on its edges.

Dijkstra's algorithm maintains for each vertex v a *tentative* distance $d[v]$, which is the length of the shortest path from s to v discovered so far. Initially $d[s] = 0$ while $d[v] = \infty$ for every $v \neq s$. It also maintains a set $S \subseteq V$ which contains vertices whose distance from s was already found. Initially $S = \varnothing$. Finally, it also maintains a *priority queue* P that holds all vertices in $V \setminus S$ whose tentative distance is finite. The key of each vertex in P is its tentative distance. The algorithm starts by inserting s into P.

In each iteration, Dijkstra's algorithm removes from P a vertex u with a smallest tentative distance. The tentative distance $d[u]$ of u is then guaranteed to be the distance from s to u in the graph, so u is added to S. In addition to that, all *out-going* edges of u are *relaxed*, i.e., for each out-going edge $(u, v) \in E$, the algorithm checks whether $d[u]+c[u, v] < d[v]$. If so, then a shorter path to v was found and the tentative distance of v is changed to $d[u] + c[u, v]$. If v is not already in P, it is inserted into P, with key $d[v]$. If v is already in P, then its key is decreased to $d[v]$.

Dijkstra's algorithm examines each edge of the graph at most once. It inserts at most n vertices into the priority queue P, and performs at most m *decrease-key* operations and at most n *extract-min* operations, where $n = |V|$ and $m = |E|$. With suitable data structures the running time of Dijkstra's algorithm is $O(m + n \log n)$ time.

B. Spira's algorithm

Spira's algorithm [7] attempts to improve on Dijkstra's algorithm when the out-going edges of each vertex u in the graph are given in *non-decreasing* order of weight. In such a setting, a tree of shortest paths may potentially be found without scanning all the edges of the graph.

When Dijkstra's algorithm finds the distance to a vertex u, it immediately scans (and relaxes) all its out-going edges. Spira's algorithm adopts a lazier approach. It scans the out-going edges of u one by one. The algorithm scans an out-going edge (u, v) only after it finds all vertices whose distance from s is smaller than $d[u] + c[u, v']$, where (u, v') is the edge preceding (u, v) in the adjacency list of u. To achieve that, the priority queue P used by Spira's algorithm holds *edges* rather than vertices. The key of an edge (u, v) in P is $d[u] + c[u, v]$. Also, if (u, v) is in P then $d[u]$ is already set to the correct distance from s to u.

Spira's algorithm again maintains a set $S \subseteq V$ that contains all vertices whose distance from s was already determined. Initially $S = \{s\}$. If $v \in S$, then $d[v]$ is the distance from s to v. If $v \neq S$, then $d[v] = \infty$. If $v \in S \backslash \{s\}$, then $(p[v], v)$ is the last edge on a path of length $d[v]$ from s to v. Initially $d[s] = 0$ while $d[v] = \infty$ for every $v \neq s$, and $p[v] = \texttt{nil}$, for every $v \in V$.

Spira's algorithm starts by scanning the *first* out-going edge (s, v) of s and inserting it into the priority queue P with key $d[s] + c[s, v] = c[s, v]$. In each iteration the algorithm extracts an edge (u, v) with the smallest key $d[u] + c[u, v]$ from P. If (u, v) is not the last out-going edge of u, then the edge (u, v') that follows it in the adjacency list of u is inserted into P with key $d[u] + c[u, v']$. Now, if $v \notin S$, then $d[u] + c[u, v]$, the key of (u, v), is guaranteed to be the distance to v. Thus, $d[v]$ is set to $d[u] + c[u, v]$, $p[v]$ is set to u, v is added to S, and the first out-going edge (v, w) of v, if there is one, is scanned and inserted into P.

Note that Spira's algorithm inserts an edge (u, v) into P even if $v \in S$ or if P already contains and edge (u', v) with $d[u'] + c[u', v] < d[u] + c[u, v]$. When (u, v) is extracted from P, the algorithm knows that it is time to scan the next out-going edge of u.

```
Algorithm Spira($G = (V, Out, c), s$)

$S \leftarrow \{s\}$ ; $P \leftarrow \varnothing$
foreach $v \in V$ do
    $d[v] \leftarrow \infty$
    $p[v] \leftarrow$ nil
    reset($Out[v]$)
$d[s] \leftarrow 0$
forward($s$)
while $S \neq V$ and $P \neq \varnothing$ do
    $(u, v) \leftarrow$ extract-min($P$)
    forward($u$)
    if $v \notin S$ then
        // New distance found
        $d[v] \leftarrow d[u] + c[u, v]$
        $p[v] \leftarrow u$
        $S \leftarrow S \cup \{v\}$
        forward($v$)
```

```
Function forward($u$)

$v \leftarrow$ next($Out[u]$)
if $v \neq$ nil then
    insert($P, (u, v), d[u] + c[u, v]$)
```

Figure 1. Spira's algorithm. forward(u) scans the next out-going edge of u.

Pseudo-code of Spira's algorithm is given in Figure 1. We discuss it in detail as most of it is reused by our improved algorithm. Each vertex $u \in V$ has an adjacency list $Out[u]$ of its out-going edges, sorted in non-decreasing order of weight. Although we view $Out[u]$ as a list of edges, each element in $Out[u]$ is a vertex, the other endpoint of the edge that leaves u. Each list $Out[u]$ has a pointer used to sequentially access its edges. reset($Out[u]$) makes this pointer point to the first edge of the list, if the list is non-empty. next($Out[u]$) returns the edge currently pointed to and advances the pointer to the next edge in the list, or past the end of the list. If the list is empty, or the pointer is past the end of the list, then next($Out[u]$) returns nil.

The implementation of Spira's algorithm uses a function forward(u) that finds the next out-going edge of u and inserts it, if it exists, into P with key $d[u] + c[u, v]$. The next out-going edge is found by calling next($Out[u]$).

Spira [7] analyzed his algorithm in the *end-point independent* model mentioned in Section I-B. Note that $\mathcal{K}_n(\text{EXP}(1))$ is clearly an end-point independent model.

Theorem II.1 ([7]). *Spira's algorithm correctly computes a tree of shortest paths from s in the input graph $G = (V, E)$. The expected number of edges scanned by the algorithm, when edge weights are generated using an end-point independent process, is at most $(1 + o(1))n \log n$.*

Proof: The correctness proof is a simple modification of the correctness proof of Dijkstra's algorithm: Whenever an edge (u, v) is extracted from the priority queue, unless $v \in S$ already, there is no cheaper path to v.

We next bound the expected number of edges examined by the algorithm when the edge weights are generated by an end-point independent process. We say that the algorithm is in stage k when $|S| = k$. When an edge (u, v) is extracted from P in stage k, the probability that $v \notin S$ is at least $(n - k)/n$. (More precisely, if (u, v) is the i-th out-going edge of u, then this probability is $(n - k)/(n - i)$.) Thus, the expected number of edges extracted in stage k is at most $n/(n-k)$, and the expected number of edges extracted during all stages is at most $\sum_{k=1}^{n-1} n/(n-k) = (1+o(1))n \log n$. Since the priority queue never has more than n edges in it, the expected number of edge insertions is also at most $(1 + o(1))n \log n$. ■

It is not difficult to show, using a slightly more careful analysis, that the expected number of edges scanned by Spira's algorithm, when run on $\mathcal{K}_n(\text{EXP}(1))$, is $(1 + o(1))n \log n$.

III. Verifying shortest paths trees

Before considering the problem of *finding* a shortest paths tree (SPT), let us consider the easier problem of *verifying* that a given tree is indeed a SPT.

A verification algorithm is an algorithm that receives a weighted directed graph $G = (V, E, c)$, where $c : E \to \mathbb{R}^+$, and a directed spanning tree T of G rooted at a source vertex s. The algorithm should check whether T is a SPT of G with source s.

We assume that each vertex u has an adjacency list $Out[u]$ containing the out-going edges of u and an adjacency list $In[u]$ containing the incoming edges of u. Furthermore, we assume that the edges appear in these adjacency lists in non-decreasing order of weight. The tree T is specified using an array p of parent pointers. If s is the root of the tree, then $p[s] =$ nil. If $u \neq s$, then $p[u] \neq$ nil and $(p[u], u)$ is the last edge in the path from s to u within T.

Given a tree T with root s, let $d[u]$ denote the length of the path from s to u within the tree. We have $d[s] = 0$, and $d[u] = d[p[u]] + c[p[u], u]$, for every $u \neq s$, where $c[u, v]$ is the weight of an edge (u, v). These formulas lead to an $O(n)$-time recursive procedure for computing the array d from the array p. By capping the recursion at depth n, we can detect any cycles that might exist in the graph defined by p, and thereby verify that the array p specifies a valid tree.

A tree T is a SPT if and only if $d[u] + c[u, v] \geq d[v]$, or equivalently $c[u, v] \geq d[v] - d[u]$, for every $(u, v) \in E$.

A. A forward-only verification algorithm

Let $D = \max\{d[u] : u \in V\}$ be the maximal distance in T. The most obvious verification algorithm simply scans the out-going adjacency list of each vertex u, verifying the condition $c[u, v] \geq d[v] - d[u]$, until it either exhausts the adjacency list of u, or encounters an edge (u, v) for which $c[u, v] \geq D - d[u]$. If (u, v') appears after (u, v) in $Out[u]$, then $c[u, v'] \geq c[u, v] \geq D - d[u] \geq d[v'] - d[u]$, so (u, v') satisfies the required condition. We refer to this algorithm as the *forward-only* verification algorithm.

It is not difficult to verify that the edges examined by this forward-only verification algorithm, when the given tree T is indeed a tree of shortest paths, are exactly the edges that Spira's algorithm inserts into its priority queue, though not necessarily in the same order. As an immediate corollary of the discussion following Theorem II.1, we thus get:

Theorem III.1. *The expected number of edges examined by the above forward-only verification algorithm, when run on a SPT of $\mathcal{K}_n(\text{EXP}(1))$, is $(1 + o(1))n \log n$.*

In the full version of the paper we show that any verification algorithm that only uses the out-going adjacency lists must inspect an expected number of at least $(1+o(1))n \log n$ edges when applied to $\mathcal{K}_n(\text{EXP}(1))$. A similar result, for a different randomly weighted graph, was obtained by Mehlhorn and Priebe [11].

B. A forward-backward verification algorithm

We next show that by using the incoming adjacency lists as well as the out-going adjacency lists we can obtain a verification algorithm that runs in $O(n)$ time, with high probability, when given a SPT of $\mathcal{K}_n(\text{EXP}(1))$. The forward-backward verification algorithm is based on the notion of *pertinent* edges.

Definition III.2 (Pertinent edges)**.** *An edge $(u, v) \in E$ is said to be* out-pertinent*, with respect to a given source vertex s and threshold M, if and only if $c[u, v] \leq 2(M - d[u])$. Edge (u, v) is said to be* in-pertinent *if and only if $c[u, v] < 2(d[v] - M)$. (Note that the first inequality is $\leq$ while the second is $<$.) An edge is said to be* pertinent *if it is either out-pertinent or in-pertinent. We let E^{out}_{per} denote the set of out-pertinent edges, E^{in}_{per} denote the set of in-pertinent edges, and $E_{per} = E^{out}_{per} \cup E^{in}_{per}$ denote the set of pertinent edges.*

Remark III.3. *For any weighted graph and any M, every edge in the shortest path tree is either out-pertinent or in-pertinent, and no edge is both.*

The forward-backward verification algorithm sets the threshold M to be the *median* distance of the vertices from the source, and then checks the condition $c[u, v] \geq d[v] - d[u]$ for all pertinent edges, *ignoring* all other edges. The median distance M can be computed in linear time. For every vertex u for which $d[u] \leq M$, the algorithm then checks all out-going edges of u of weight at most $2(M - d[u])$. This is the *forward* scan. For every vertex v for which $d[v] \geq M$, it then checks all incoming edges of v of weight less than $2(d[v] - M)$. This is the *backward* scan. If all conditions are satisfied, the verification algorithm accepts T. The correctness of the algorithm follows from the following lemma.

Lemma III.4. *If $c[u, v] \geq d[v] - d[u]$ for every pertinent edge (u, v), then $c[u, v] \geq d[v] - d[u]$ for every $(u, v) \in E$.*

Proof: If $(u, v) \notin E_{per}$, then $c[u, v] > 2\,(M - d[u])$ and $c[u, v] \geq 2\,(d[v] - M)$. Thus,

$$c[u, v] \;>\; \frac{1}{2}\Big(2\,(M - d[u]) + 2\,(d[v] - M)\Big) \;=\; d[v] - d[u] \,,$$

as required. ∎

The verification algorithm is correct with any choice of M. Letting M be the median distance minimizes the running time in many interesting cases. For $\mathcal{K}_n(\text{EXP}(1))$ we show in Section V that the number of pertinent edges with respect to the median distance is $\Theta(n)$ both in expectation and with high probability. As a consequence, we get:

Theorem III.5. *The running time of the forward-backward verification algorithm, when run on $\mathcal{K}_n(\text{EXP}(1))$ with sorted adjacency lists, is $\Theta(n)$, with very high probability. The probability that the running time exceeds $\Theta(n + \Delta)$ decays exponentially in Δ.*

Remark III.6. *For the purposes of verifying a SPT, it suffices to check the edges (u, v) for which (u, v) is in the tree, or $c[u, v] < 2(M - d[u])$, or $c[u, v] < 2(d[v] - M)$. However, for the purposes of finding the SPT, our algorithm will also look at edges for which $c[u, v] = 2(M - d[u])$, and up to n additional edges.*

IV. The forward-backward shortest paths algorithm

Our goal in this section is to develop a single-source shortest paths algorithm that matches the performance of the forward-backward verification algorithm of the previous section. To achieve that, almost all edges examined by the algorithm must be *pertinent*. The new algorithm is composed of two stages. In the first stage the algorithm finds distances to the closest $\lceil n/2 \rceil$ vertices, and hence also the median distance M. In its first stage, the algorithm behaves exactly like Spira's algorithm described in Section II-B. In its second stage, the algorithm finds the distances to the at most $\lfloor n/2 \rfloor$ remaining vertices.

In the second stage of the algorithm, the median distance M is known, and the algorithm starts to identify in-pertinent edges. When an in-pertinent edge (u, v) is found, using a backward scan from v, a *request* is issued for the forward scan of the edge (u, v), at the appropriate time.

```
Algorithm sssp($G = (V, In, Out, c), s$)

$S \leftarrow \{s\}$ ; $M \leftarrow \infty$ ; $n \leftarrow |V|$
$P \leftarrow \varnothing$ ; $Q \leftarrow \varnothing$
foreach $v \in V$ do
    $d[v] \leftarrow \infty$ ; $p[v] \leftarrow$ nil
    $out[u] \leftarrow$ true ; $Req[v] \leftarrow \varnothing$
    reset($Out[v]$) ; reset($Req[v]$)
    reset($In[v]$)
$d[s] \leftarrow 0$
forward($s$)
while $S \neq V$ and $P \neq \varnothing$ do
    $(u, v) \leftarrow$ extract-min($P$)
    forward($u$)
    if $v \notin S$ then
        // New distance found
        $d[v] \leftarrow d[u] + c[u, v]$
        $p[v] \leftarrow u$
        $S \leftarrow S \cup \{v\}$
        forward($v$)
        if $|S| = \lceil n/2 \rceil$ then
            $M \leftarrow d[v]$
            foreach $w \notin S$ do
                backward($w$)
    // Find more in-pertinent edges
    while $Q \neq \varnothing$ and
    min($Q$) $< 2\,($min($P$)$-M)$ do
        $(u, v) \leftarrow$ extract-min($Q$)
        if $v \notin S$ then
            backward($v$)
            request($u, v$)
```

```
Function forward($u$)

if $out[u]$ then
    // find next out-pertinent edge
    $v \leftarrow$ next($Out[u]$)
    if $v =$ nil or $c[u, v] > 2\,(M - d[u])$ then
        $out[u] \leftarrow$ false
if not $out[u]$ then
    // find next in-pertinent edge
    $v \leftarrow$ next($Req[u]$)
$active[u] \leftarrow v \neq$ nil
if $active[u]$ then
    insert($P, (u, v), d[u] + c[u, v]$)
```

```
Function backward($v$)

$u \leftarrow$ next($In[v]$)
if $u \neq$ nil then
    insert($Q, (u, v), c[u, v]$)
```

```
Function request($u, v$)

if $c[u, v] > 2(M - d[u])$ then
    append($Req[u], v$)
    if $u \in S$ and not $active[u]$ then
        // urgent request
        forward($u$)
```

Figure 2. The new forward-backward SSSP algorithm. The inner **while** loop is only executed when $|S| \geq \lceil n/2 \rceil$. forward($u$) scans the next out-going edge of u, and backward(v) scans the next incoming edge of v. request(u, v) requests a forward scan of edge (u, v); the grey condition is optional.

Usually, a requested edge (u, v) is simply appended to a list $Req[u]$ of requested edges. If u had exhausted all its out-going pertinent edges when a new in-pertinent edge (u, v) is discovered, the request (u, v) is considered to be *urgent*, and (u, v) is immediately scanned and added to P.

A. Description of the algorithm

The input to the algorithm is a weighted directed graph $G = (V, E, c)$, where $c : E \rightarrow \mathbb{R}^+$, and a source $s \in V$. Each vertex $u \in V$ has a list $Out[u]$ of its out-going edges and a list $In[u]$ of its incoming edges of u. Both lists are sorted in non-decreasing order of cost. Although we view $Out[u]$ and $In[u]$ as list of edges, each element in them is a vertex, the other endpoint of the edge that leaves or enters u. Every vertex $u \in V$ also has a second list $Req[u]$ of out-going edges. Initially this list is empty. $Req[u]$ contains edges whose scan was specifically *requested*. All requested edges are in-pertinent.

Each such In, Out or Req adjacency list L has a *pointer* used to sequentially access its edges. reset(L) makes this pointer point to the first edge of the list, if the list is non-empty. next(L) returns the edge currently pointed to and advances the pointer to the next edge in the list, or past the end of the list. If the list is empty, or the pointer is past the end of the list, then next(L) returns nil. If an edge is appended to L when the pointer is past the end of the list, the pointer is set to point to the newly added edge.

For each vertex u, the algorithm maintains a bit $out[u]$ which is set if $Out[u]$ may still contain unscanned out-pertinent edges. Initially $out[u]$ is set to true. When the next pertinent out-going edge of u is sought, and $out[u]$ is set, the algorithm looks at the next available edge from $Out[u]$. If this edge is out-pertinent, it is used. If it is not out-pertinent, then $out[u]$ is set to false. When $out[u]$ is false, the next edge from $Req[u]$, if there is one, is used; it is guaranteed to be in-pertinent.

The algorithm maintains a set $S \subseteq V$ that contains all the vertices v whose distance from s was already found. Initially $S = \{s\}$. If $v \in S$, then $d[v]$ is the distance from s to v in the graph. If $v \in S \setminus \{s\}$, then $(p[v], v)$ is the last edge on a shortest path from s to v in the graph. If $v \notin S$, then $d[v] = \infty$ and $p[v] =$ nil. We also have $p[s] =$ nil. The algorithm also maintains the size of the set S. (There is actually no need to explicitly maintain S. The condition $v \in S$, used below, can be replaced by the condition $d[v] < \infty$, and the condition $S \neq V$ can be replaced by $|S| \neq |V|$. However, it is useful to refer to the set S by name.)

The algorithm maintains two priority queues P and Q. The first priority queue P is analogous to the priority queue used by Spira's algorithm. The second priority queue Q is used to identify in-pertinent edges as explained above. At any stage during the operation of the algorithm, each vertex u has at most one out-going edge (u, v) in P. Essentially all these edges are pertinent. Similarly, each vertex v has at most one incoming edge (u, v) in Q. When (u, v) is inserted into Q we have $v \notin S$. However, v may be added to S before (u, v) is extracted from Q. All edges extracted from Q are in-pertinent edges.

Pseudo-code of the new forward-backward single-source shortest paths algorithm is given in Figure 2. It starts with straightforward initializations. In particular, M is initialized to ∞. The algorithm uses a function forward(u) that finds the next pertinent out-going edge (u, v) of u, if there is one, and inserts it into P. In the first stage of the algorithm all edges are assumed to be pertinent. The algorithm starts by calling forward(s) to insert the first out-going edge of s into P.

forward(u) works as follows. If $out[u]$ is true, it uses next$(Out[u])$ to obtain the next out-going edge (u, v) from $Out[u]$, if there is one. If (u, v) exists, it checks whether $c[u, v] \leq 2(M - d[u])$. If the algorithm is still in its first stage, then $M = \infty$ and the condition is automatically satisfied. If the algorithm is already in its second stage and the condition is satisfied, then (u, v) is out-pertinent. If (u, v) does not exist, or fails the condition, then $out[u]$ is set to false, as $Out[u]$ does not contain additional out-pertinent edges. If $out[u]$ is false, forward(u) uses next$(Req[u])$ to obtain the next edge (u, v) from $Req[u]$. If an appropriate edge (u, v) is found, from either $Out[u]$ or $Req[u]$, then u is said to be *active*, $active[u]$ is set to true, and (u, v) is inserted into P. If no next edge (u, v) is found, then u is said to be *inactive*, and $active[u]$ is set to false.

The operation of algorithm is composed of *iterations*. Each iteration starts by extracting an edge (u, v) of minimum key from P and by calling forward(u) to scan the next out-going edge of u, if any, and add it to P. If $v \notin S$, then, as we shall see, $d[u] + c[u, v]$ is the distance from s to v, so $d[v]$ is set to $d[u] + c[u, v]$, $p[v]$ is set to u, v is added to S and its first out-going edge is scanned by calling forward(v). This is all that is done in an iteration during the *first stage* of the algorithm, i.e., until the $\lceil n/2 \rceil$-th vertex is added to S. The behavior of the algorithm in the first stage is thus identical to the behavior of Spira's algorithm. If v is the $\lceil n/2 \rceil$-th vertex is added to S, then M is set to $d[v]$, which is the median distance, and the algorithm enters its second stage which lasts until distances to all vertices reachable from s are found.

The second stage starts by backward scanning the first incoming edge of each vertex which is not yet in S and inserting it into the priority queue Q. Backward scans are performed using backward(v).

Each iteration during the second stage of the algorithm ends with the execution of an inner while loops that identifies new in-pertinent edges. While $Q \neq \varnothing$ and $\min(Q) < 2(\min(P) - M)$, an edge (u, v) of minimum weight is extracted from Q. If $v \notin S$, then the next incoming edge of v is scanned and inserted into Q by calling backward(v). The forward scan of (u, v), which is guaranteed to be in-pertinent, is then requested by calling request(u, v). $\min(P)$ and $\min(Q)$ in the condition above are the minimum keys of elements contained in P and Q, respectively. (If P is empty, then $\min(P)$ is taken to be ∞.) If the request of an edge (u, v) is urgent, then this edge is immediately inserted into P, which may decrease $\min(P)$. The inner while loop is not executed during the first stage, as Q become non-empty only at the end of the first stage.

Requesting an edge (u, v) is done by calling request(u, v). We shall prove that every requested edge is in-pertinent, and therefore not out-pertinent. To simplify the correctness proof, we assume at first that request(u, v) explicitly checks that (u, v) is not out-pertinent. We later prove that this test is unnecessary, as it is always satisfied. (To indicate the fact that the test $c[u, v] > 2(M - d[u])$ could be omitted, it is shown in grey.) request(u, v) appends (u, v) to $Req[u]$. If $u \in S$ and $active[u]$ is false, then request is urgent, and forward(u) is called immediately to scan (u, v).

B. Correctness of the algorithm

We begin with some technical lemmas that play a central role in the correctness proof. Due to lack of space, the proofs of the lemmas are omitted. Let d_v be the distance from s to v in the input graph. Our goal is to show that when the algorithm terminates, $d[v] = d_v$, for every $v \in V$. The first (obvious) lemma claims that the algorithm never underestimates distances.

Lemma IV.1. *At any stage of the forward-backward algorithm, $d_v \leq d[v]$, for every $v \in V$.*

Let $key[u, v] = d[u] + c[u, v]$ be the key of an edge (u, v) when it is inserted into P. ($d[u]$ does not change after that moment.) Parts (i) and (iv) of the second (technical) lemma claim that P and Q are *monotone* priority queues, i.e., the keys of the successive edges extracted from them are monotonically non-decreasing.

Lemma IV.2. *For the forward-backward algorithm,*

- (i) *If (u, v) is extracted from Q before (u', v'), then $c[u, v] \leq c[u', v']$.*
- (ii) *If (u, v) is inserted into $Req[u]$ before (u, v'), then $c[u, v] \leq c[u, v']$.*
- (iii) *If (u, v) is inserted into P before (u, v'), then $c[u, v] \leq c[u, v']$.*
- (iv) *If (u, v) is extracted from P before (u', v'), then $key[u, v] \leq key[u', v']$.*

Lemma IV.3. *If $u \in S$ and (u, v) is an out-pertinent or a requested edge that was not extracted yet from P, then P must contain an edge (u, v') (possibly $v' = v$) with $c[u, v'] \leq c[u, v]$.*

Lemma IV.4. *When the forward-backward algorithm terminates, S is the set of vertices reachable from the source.*

Lemma IV.5. *When an edge (u, v) is extracted from priority queue P, all incoming in-pertinent edges of v have already been requested.*

We are now ready for the proof of the following theorem.

Theorem IV.6. *The forward-backward single-source shortest paths algorithm correctly finds a tree of shortest paths.*

Proof: Let $s = w_0, w_1, \ldots, w_k = v$ be a shortest path from the source s to a vertex v. We prove by induction on k that $d[w_k] = d_{w_k}$. As $d[s] = d_s = 0$, the claim is true for $k = 0$. Suppose $k > 0$ and that our induction hypothesis is true for $k - 1$. Since w_k is reachable from the source, by Lemma IV.4, it is adjoined to S at some iteration, and we let (u, w_k) denote the edge that is extracted from P during that iteration.

Suppose at first that $w_{k-1} \notin S$ when (u, w_k) is extracted from P. Then by Lemma IV.2(iv),

$$d[w_k] \leq d[w_{k-1}] = d_{w_{k-1}} = d_{w_k} - c[w_{k-1}, w_k],$$

which combined with Lemma IV.1 implies $d[w_k] = d_{w_k}$ (and $c[w_{k-1}, w_k] = 0$).

Suppose instead that $w_{k-1} \in S$ when (u, w_k) is extracted from P. Since (w_{k-1}, w_k) is a shortest path edge, it is either out-pertinent or in-pertinent. If it is in-pertinent, by Lemma IV.5 it was requested by the end of the previous iteration. In either case, by the end of the previous iteration, edge (w_{k-1}, w_k) is an out-pertinent or requested edge that has not been extracted from P, and $w_{k-1} \in S$. By Lemma IV.3, there must be some edge (w_{k-1}, x) in queue P for which

$$\begin{aligned} key[w_{k-1}, x] &= d[w_{k-1}] + c[w_{k-1}, x] \\ &\leq d[w_{k-1}] + c[w_{k-1}, w_k] \\ &= d_{w_{k-1}} + c[w_{k-1}, w_k] = d_{w_k} \\ &\leq d[u] + c[u, w_k] = key[u, w_k] = d[w_k]. \end{aligned}$$

Since it was edge (u, w_k) that was extracted from P, $key[u, w_k] \leq key[w_{k-1}, x]$, so the inequalities $\leq$ are actually equalities, and $d[w_k] = d_{w_k}$.

Thus by induction each vertex v reachable from the source satisfies $d[v] = d_v$, and so the algorithm computes a shortest path tree. ■

C. Complexity of the algorithm

The running time of the algorithm is clearly dominated by the priority queue operations. The following two lemmas show that the number of priority queue operations performed by the algorithm is $O(|E_{per}|)$. The proofs are again omitted due to lack of space.

Lemma IV.7. *For the forward-backward algorithm, all requested edges are in-pertinent.*

Remark IV.8. *Since in-pertinent edges are not out-pertinent, one consequence of Lemma IV.7 is that the grey if statement in the request function is unnecessary.*

Lemma IV.9. *For the forward-backward algorithm,*

- (i) *The edges inserted into Q are all the in-pertinent edges, together with a lightest incoming non-in-pertinent edge (if any) to each vertex found after the $\lceil n/2 \rceil$th vertex.*
- (ii) *All edges inserted into P, except possibly one out-going edge for each vertex, are pertinent edges.*

Theorem IV.10. *The running time of the forward-backward single-source shortest paths algorithm, when run on $\mathcal{K}_n(\text{EXP}(1))$ with sorted adjacency lists, is $O(n)$, with very high probability.*

V. Performance with exponential edge costs

A. Shortest path tree for randomly weighted graphs

For the complete graph with i.i.d. $\text{EXP}(1)$ edge weights, Davis and Prieditis [22] and Janson [23] gave an elegant characterization of the set of all distances from a given source vertex s, which we now recall. Let $v_1, \ldots, v_n$ denote the vertices arranged in increasing order of distance from the source s (in particular $v_1 = s$). Let d_u denote the distance to vertex u from the source. For $k = 2, \dots, n$, let p_k denote the index of v_k's parent in the shortest-path tree, i.e., (v_{p_k}, v_k) is an edge of the shortest path tree. Because of the memoryless property of the exponential distribution, and because there are $k(n-k)$ edges from $v_1, \dots, v_k$ to the remaining $n-k$ vertices, it follows that $d_{v_{k+1}} - d_{v_k}$ is an exponential random variable with mean $1/(k(n-k))$, independent of the previous distances, v_{k+1} is a uniformly random vertex from the remaining vertices, and p_{k+1} is a uniformly random choice from $1, \dots, k$. The quantities v_{k+1}, p_{k+1}, and $d_{v_{k+1}} - d_{v_k}$ are mutually independent, and the only dependence that they have upon the values of $v_1, \dots, v_k$, $p_2, \dots, p_k$, and $d_{v_1}, \dots, d_{v_k}$ is that v_{k+1} is distinct from $v_1, \dots, v_k$ and that p_{k+1} is one of $v_1, \dots, v_k$.

Define $X_k \sim \text{EXP}[1/(k(n-k))]$ and $d_{v_k} = \sum_{i=1}^{k-1} X_i$, where all the X_i's are independent of each other. The weight of any edge (v_{p_k}, v_k) of the shortest-path tree is

$$c(v_{p_k}, v_k) = d_{v_k} - d_{v_{p_k}}.$$

For any edge (v_j, v_k) not in the shortest path tree, we have

$$c(v_j, v_k) = \begin{cases} \text{EXP}(1) & k < j \\ d_{v_k} - d_{v_j} + \text{EXP}(1) & j < k, \end{cases}$$

where all these exponential random variables are mutually independent and also independent of $v_1, \dots, v_n$, $d_{v_1}, \dots, d_{v_n}$, and $p_2, \dots, p_n$. (Here there is a slight difference between the cases of directed graphs and undirected graphs. In the case of undirected graphs, the formula is $|d_{v_k} - d_{v_j}| + \text{EXP}(1)$ regardless of the order of j and k. Otherwise, the characterizations of $v_1, \dots, v_n$, $d_{v_1}, \dots, d_{v_n}$, $p_2, \dots, p_n$ and $c(u, v)$ are the same for directed and undirected graphs.)

B. Comparison of pertinent edges to a Poisson process

The above characterization of the shortest path tree is particularly convenient for the purposes of comparing the distance to a vertex to the median distance, allowing us to estimate the number of pertinent edges.

Theorem V.1. *Let Λ denote the random variable*

$$\Lambda = 2(n-1) \sum_{k=\lceil n/2 \rceil}^{n-1} \frac{1}{k} \text{EXP}(1),$$

where the $\text{EXP}(1)$*'s are i.i.d. Then the number of out-pertinent edges that are not shortest paths edges is stochastically dominated by* $\text{POISSON}(\Lambda)$*, and similarly for the number of in-pertinent edges that are not SPT edges.*

Proof: For each edge we can associate an independent Poisson point process on $\mathbb{R}^+$, and that edge's associated exponential random variable in its weight is then just the first point in the point process. The indicator variable for the edge being lighter than a certain threshold is dominated by the number of points of the point process in that interval. Thus the number of out-pertinant edges which are not shortest-path tree edges is dominated by a Poisson random variable with a random rate Λ_{out}, and similarly for the in-pertinent edges which are not SPT edges. The variable Λ_{in} for in-pertinent edges is given by

$$\begin{aligned} \Lambda_{\text{in}} &= (n-1) \sum_{j=\lceil n/2 \rceil+1}^{n} 2(d_{v_j} - M) \\ &= 2(n-1) \sum_{j=\lceil n/2 \rceil+1}^{n} \sum_{k=\lceil n/2 \rceil}^{j-1} X_k \\ &= 2(n-1) \sum_{k=\lceil n/2 \rceil}^{n-1} (n-k) X_k \\ &= 2(n-1) \sum_{k=\lceil n/2 \rceil}^{n-1} \frac{\text{EXP}(1)}{k}, \end{aligned}$$

where the $\text{EXP}(1)$'s are i.i.d. A similar calculation of out-pertinent edges gives

$$\Lambda_{\text{out}} = (n-1) \sum_{j=1}^{\lceil n/2 \rceil-1} 2(M - d_{v_j}) = 2(n-1) \sum_{k=\lfloor n/2 \rfloor+1}^{n-1} \frac{\text{EXP}(1)}{k}.$$

When n is odd, Λ_{out} and Λ_{in} are equal in distribution, while if n is even, Λ_{in} has one more term. In either case, the theorem follows. ■

C. First moment estimate

Theorem V.2. *For Λ defined in Theorem V.1, we have*

$$\mathbb{E}[\text{POISSON}(\Lambda)] < (\log 4)n + 1.$$

Proof: As $\mathbb{E}[\text{POISSON}(\Lambda)] = \mathbb{E}[\Lambda]$, we have

$$\mathbb{E}[\Lambda] = 2(n-1) \sum_{k=\lceil n/2 \rceil}^{n-1} \frac{1}{k}.$$

For even n the sum is at most

$$\frac{1}{n-1} + \int_{(n-2)/2}^{n-2} \frac{dk}{k} = \frac{1}{n-1} + \log 2,$$

and thus $\mathbb{E}[\Lambda] \le (\log 4)(n-1) + 2 < (\log 4)n + 1$. When n is odd, a slightly better bound may be obtained. ■

Thus the expected number of in-pertinent edges not in the shortest path tree is less than $(\log 4)n + 1$, and similarly for out-pertinent edges not in the shortest path tree. In particular, the expected number of pertinent edges is less than $(1 + 4 \log 2)n + 1 < 3.7726n + 1$.

VI. Efficient priority queues

In this section we briefly sketch the two-level bucket-based monotone priority queues used to implement the new forward-backward SSSP algorithm in $O(n)$ time, with high probability. We use $B = \Theta(n)$ high-level buckets, each of width $W = \Theta(1/(n \log n))$. The i-th bucket, where $0 \leq i < B$, contains items whose keys are in the interval $[iW, (i+1)W)$. Items with key $\geq BW$ are placed in the last bucket. The items in each high-level bucket are stored in a linked list. When a bucket becomes *active*, it is split into k equal-width low-level buckets, where k is the number of items contained in the high-level bucket when it becomes active. We do not resplit or rebalance low-level buckets when new items are added to them. The items in each low-level bucket are stored in a standard binary heap (Williams [24]).

Theorem VI.1. *Suppose the forward-backward algorithm's priority queues P and Q are implemented as a two-level bucket monotone priority queues with sub-buckets based on binary heaps, as described above. If $W = \Theta(1/(n \log n))$ and $B = \Theta(n)$, then when the algorithm is run on $\mathcal{K}_n(\text{EXP}(1))$, the expected running time for priority queues P and Q is $\Theta(n)$, and the running time is $\Theta(n)$ except with probability $\exp(-\Theta(n/\log n))$.*

References

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, pp. 269–271, 1959.

[2] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. Assoc. Comput. Mach.*, vol. 34, no. 3, pp. 596–615, 1987.

[3] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *J. ACM*, vol. 46, no. 3, pp. 362–394, 1999.

[4] S. Pettie, "A new approach to all-pairs shortest paths on real-weighted graphs," *Theoret. Comput. Sci.*, vol. 312, no. 1, pp. 47–74, 2004.

[5] R. Hassin and E. Zemel, "On shortest paths in graphs with random weights," *Math. Oper. Res.*, vol. 10, no. 4, pp. 557–564, 1985.

[6] A. M. Frieze and G. R. Grimmett, "The shortest-path problem for graphs with random arc-lengths," *Discrete Appl. Math.*, vol. 10, no. 1, pp. 57–77, 1985.

[7] P. M. Spira, "A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$," *SIAM J. Comput.*, vol. 2, pp. 28–32, 1973.

[8] T. Takaoka and A. Moffat, "An $O(n^2 \log n \log \log n)$ expected time algorithm for the all shortest distance problem," in *Mathematical foundations of computer science, 1980 (Proc. Ninth Sympos., Rydzyna, 1980)*, ser. Lecture Notes in Comput. Sci. Berlin: Springer, 1980, vol. 88, pp. 643–655.

[9] P. A. Bloniarz, "A shortest-path algorithm with expected time $O(n^2 \log n \log^* n)$," *SIAM J. Comput.*, vol. 12, no. 3, pp. 588–600, 1983.

[10] A. Moffat and T. Takaoka, "An all pairs shortest path algorithm with expected time $O(n^2 \log n)$," *SIAM J. Comput.*, vol. 16, no. 6, pp. 1023–1031, 1987.

[11] K. Mehlhorn and V. Priebe, "On the all-pairs shortest-path algorithm of Moffat and Takaoka," *Random Structures Algorithms*, vol. 10, no. 1-2, pp. 205–220, 1997, average-case analysis of algorithms (Dagstuhl, 1995).

[12] T. Takaoka, "A simplified algorithm for the all pairs shortest path problem with $O(n^2 \log n)$ expected time," *J. Comb. Optim.*, vol. 25, no. 2, pp. 326–337, 2013.

[13] T. Takaoka and M. Hashim, "A simpler algorithm for the all pairs shortest path problem with $O(n^2 \log n)$ expected time," in *Combinatorial optimization and applications. Part II*, ser. Lecture Notes in Comput. Sci. Berlin: Springer, 2010, vol. 6509, pp. 195–206.

[14] Y. Peres, D. Sotnikov, B. Sudakov, and U. Zwick, "All-pairs shortest paths in $O(n^2)$ time with high probability," in *Proc. of 51st FOCS*, 2010, pp. 663–672.

[15] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths," *J. ACM*, vol. 51, no. 6, pp. 968–992 (electronic), 2004.

[16] T. Nicholson, "Finding the shortest route between two points in a network," *The Computer Journal*, vol. 9, no. 3, pp. 275–280, 1966.

[17] I. Pohl, "Bi-directional search," in *Machine Intelligence, 6*. Edinburgh University Press, 1971, pp. 127–140.

[18] M. Luby and P. Ragde, "A bidirectional shortest-path algorithm with good average-case behavior," *Algorithmica*, vol. 4, no. 4, pp. 551–567, 1989.

[19] U. Meyer, "Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds," *J. Algorithms*, vol. 48, no. 1, pp. 91–134, 2003.

[20] T. Hagerup, "Simpler computation of single-source shortest paths in linear average time," *Theory Comput. Syst.*, vol. 39, no. 1, pp. 113–120, 2006.

[21] A. V. Goldberg, "A practical shortest path algorithm with linear expected time," *SIAM J. Comput.*, vol. 37, no. 5, pp. 1637–1655, 2008.

[22] R. Davis and A. Prieditis, "The expected length of a shortest path," *Inform. Process. Lett.*, vol. 46, no. 3, pp. 135–141, 1993.

[23] S. Janson, "One, two and three times $\log n/n$ for paths in a complete graph with random weights," *Combin. Probab. Comput.*, vol. 8, no. 4, pp. 347–361, 1999.

[24] J. Williams, "Algorithm 232: Heapsort," vol. 7, pp. 347–348, 1964.