10-1-1995

Finding Real-Valued Single-Source Shortest Paths in o(n^3) Expected Time

Stavros G. Kolliopoulos
Dartmouth College

Clifford Stein
Dartmouth College

FINDING REAL-VALUED SINGLE-SOURCE SHORTEST PATHS IN $o(n^3)$ EXPECTED TIME

Stavros G. Kolliopoulos
Clifford Stein

Technical Report PCS-TR95-272

10/95

Finding Real-Valued Single-Source Shortest Paths in $o(n^3)$ Expected Time

Stavros G. Kolliopoulos Clifford Stein*

Department of Computer Science
Dartmouth College†

Abstract

Given an n-vertex directed network G with real costs on the edges and a designated source vertex s, we give a new algorithm to compute shortest paths from s. Our algorithm is a simple deterministic algorithm with $O(n^2 \log n)$ expected running time over a large class of input distributions.

The shortest path problem is an old and fundamental problem with a host of applications. Our algorithm is the first strongly-polynomial algorithm in over 35 years to improve upon some aspect of the running time of the celebrated Bellman-Ford algorithm for arbitrary networks, with any type of cost assignments.

1 Introduction

Given a directed network $G = (V, E, c)$ where c is a real-valued function mapping edges to costs, the single-source shortest path problem (abbreviated SSSP) consists of finding, for each vertex $v \in V$, the simple path of minimum total cost from a designated source vertex s. This is an old and fundamental problem in network optimization with a plethora of applications in operations research, see for example [AMO93]. It also arises as a subproblem in other optimization problems such as network flows.

The classic Bellman-Ford algorithm solves the SSSP problem in an n-vertex m-edge network in $O(nm)$ time [B58, FF62]. This simple algorithm has been widely used and studied for over 35 years, however, in all that time, no progress has been made in improving the worst case bound for arbitrary real-valued shortest path problems.

We cannot improve on the Bellman-Ford algorithm in the worst case for all networks. However, in this paper, we will show that on any network with real edge costs chosen from a large class of input distributions, we can provide a deterministic algorithm for SSSP that runs in $O(n^2 \log n)$ expected time. Our algorithm is faster than Bellman-Ford in the case when $m = \omega(n \log n)$, and, under the similarity assumption [AMO93], faster than the recent scaling algorithm of Goldberg [G95], when $m = \omega(n^{1.5} \log n)$. To our knowledge, this is the first strongly polynomial algorithm that solves SSSP in $o(n^3)$ time for networks with arbitrary numbers of edges, arbitrary topology, and real-valued costs.

Our model of random edge costs is Bloniarz's endpoint independent model [B83]. This model is the most general studied in the shortest path literature [S73, B83, MT87, TM80]. It includes the common case of all edge costs drawn from the uniform distribution, and more generally any network in which the only restriction is that all costs of edges emanating from a particular vertex v are chosen from the same distribution. Our method uses ideas from Bellman-Ford and from an algorithm of Moffat and Takaoka

†Department of Computer Science, Dartmouth College 6211, Hanover, NH 03755-3510.
E-mail: {stavros, cliff}@cs.dartmouth.edu.

*Research partly supported by NSF Award CCR-9308701, a Walter Burke Research Initiation Award and a Dartmouth College Research Initiation Award.

[MT87] originally intended for nonnegative cost assignments, and turns out to be quite simple drawing on the simplicity of these two algorithms.

We also show that in a restricted model of computation, the Bellman-Ford algorithm is the best possible. We consider an *oblivious* model of computation, in which the decisions of which edges to relax are made in advance, before seeing the input. We show that in this case, any algorithm whose basic operation is edge relaxation has to perform $\Omega(nm)$ edge relaxations in the worst case in order to correctly compute shortest paths. The Bellman-Ford algorithm fits into this model.

Previous and Related Work. For arbitrary real costs the existence of negative cycles, i.e. paths with negative cost where every vertex has degree 2, makes the SSSP problem NP-hard [GJ79]. In the absence of negative cycles the fastest SSSP algorithm, as mentioned above, is attributed to Bellman and Ford [B58, FF62] and can be implemented to run in $O(nm)$ time, worst case. This is $O(n^3)$ for dense graphs. Until recently all alternative implementations of Bellman-Ford first solve an assignment problem to find vertex potentials which allows reweighting of edges so that all edge costs are nonnegative. Then Dijkstra's algorithm [D59] is applied to the reweighted network. The bottleneck in this approach is the solution of the assignment problem. The first and fastest strongly polynomial time algorithm for the assignment problem is Kuhn's Hungarian algorithm [K55]. Implemented with Fibonacci heaps [FT87], this algorithm runs in $O(nm + n^2 \log n)$ time. Gabow and Tarjan [GT87] have a scaling algorithm for the assignment problem that runs in $O(\sqrt{n}m \log(nC))$ time, where C is the absolute value of the most negative edge cost. Recently Goldberg [G95] has given a scaling algorithm which finds shortest paths without solving an assignment problem first; this algorithm has running time $O(\sqrt{n}m \log C)$. All these algorithms detect the existence of a negative cycle. We also note that if the costs are nonnegative, faster algorithms are possible, as Dijkstra's algorithm [D59] implemented with Fibonacci heaps [FT87] runs in $O(n \log n + m)$ time.

We are not aware of any work on the average case complexity of the SSSP problem for real-valued edge costs, the problem with nonnegative edge costs is well studied. In the all pairs shortest path problem, (abbreviated APSP) we are interested in computing the shortest paths between all $n(n-1)$ pairs of vertices. All previous work on the analysis of algorithms for networks with random nonnegative edge costs has been for the APSP problem. An APSP algorithm with expected running time $O(n^2 \log^2 n)$ on networks with random edge costs was presented in a classical paper by Spira [S73]. This result was later refined ([CL77], [BMF79]) and improved in [TM80] where an $O(n^2 \log n \log \log n)$ time algorithm was given. Refinements included taking into account non-unique edge costs. Bloniarz [B83] achieved an expected running time of $O(n^2 \log n \log^* n)$ and relaxed Spira's initial assumption that edge costs are drawn independently from any single but arbitrary distribution. He introduced the *endpoint independent* randomness model which is the most general in the shortest path literature. Frieze and Grimmet [FG85] gave an $(O(n(m + n \log n))$ expected time algorithm, an improvement over Bloniarz's when m is small enough. They also gave an $O(n^2 \log n)$ expected time algorithm for the case where edge costs are identically and independently distributed with distribution function F, where $F(0) = 0$ and F is differentiable at 0. The fastest algorithm so far under the endpoint independent model is due to Moffat and Takaoka [MT87] and runs in $O(n^2 \log n)$ time. Although these papers are concerned with the APSP problem, they all run an SSSP routine n times, using each vertex as a source in turn. Because they all require an $O(n^2 \log n)$ time preprocessing sorting phase, the running times are only reported for the APSP problem, but these algorithms do give ideas for the SSSP problem. Recently some research has been done on randomized algorithms that use ideas from matrix multiplication [AGM91, S92] but, for arbitrary cost assignments, only pseudopolynomial algorithms exist. The extent to which randomization can be used for faster algorithms, as was the case with e.g. minimum cut [KS93] and minimum spanning tree [KKT], is an open question.

The outline of the paper is as follows. In Section 2, we start with a high level description of the new algorithm. Subsequently we present the randomness model, and give an implementation with fast average case. In Section 3, we present the lower bound for oblivious algorithms. In Section 4, we conclude with open questions.

```
procedure FAST_SSSP(Network G, source s)
        for all v ∈ V do
                sort the list of edges out of v;
                d^0(v):= 0;
        for i= 1 to n-1 do
                d^i =PASS(G, d^{i-1});
        for all edges (u,v) ∈ E do
                if d^{n-1}(u) + c(u,v) < d^{n-1}(v) then
                   report a negative cycle and break;
```

Figure 1: Algorithm FAST_SSSP

2 The Algorithm

In this section we give an algorithm for SSSP with average case running time $O(n^2 \log n)$ on a broad class of networks with random edge costs. We will give the algorithm in two parts. In Section 2.1 we give a modified version of the Bellman-Ford [B58, FF62] algorithm that reduces solving a shortest path problem to a sequence of n shortest path problems in a simpler network. Then in Section 2.2 we show how to solve this simpler shortest path problem in $O(n \log n)$ time, on average.

We use n to denote $|V|$ and m to denote $|E|$ for the network of interest. The *length* of path $p = (v_1, v_2, \ldots, v_k)$ is $k-1$ while the cost is defined as $c(p) = \sum_{i=1}^{k-1} c(v_i, v_{i+1})$. The shortest path between a pair of vertices u, v is a minimum cost simple path joining the two vertices. This minimum cost is called the *distance*, denoted $d(u,v)$. Our algorithms will concentrate on finding the distance from the source s to each vertex, and we will use $d(v)$ to denote $d(s,v)$. We concentrate on finding the distances but our method can easily be modified to find the actual paths without asymptotic overhead.

2.1 Skeleton of the new algorithm

We begin by reviewing and modifying the Bellman-Ford algorithm. On a network where negative costs are present Bellman-Ford performs $n-1$ passes. Let $d^i(v)$ be the distance of the shortest path from s to v that has been found during the first i passes. During pass i, all edges (u,v) are relaxed in an arbitrary order. More precisely, pass i is implemented by first assigning $d^i(v) = d^{i-1}(v)$ for all v and then, for all edges (u,v), if

$$d^i(u) + c(u,v) < d^i(v), \tag{1}$$

then $d^i(v)$ is updated to $d^i(u) + c(u,v)$. This step is called *relaxing* edge (u,v). An extra n-th pass will reveal the existence of a negative cycle since any possible vertex cost decrease will be due to a non-simple path. Adding a simple preprocessing phase to Bellman-Ford, in which we sort the edge lists, we get the algorithm FAST_SSSP, which appears in Figure 2.1.

It is easy to show by induction that after the i-th pass all shortest paths of length *at most* i have certainly been discovered, and possibly others. More precisely, let $\hat{d}^i(v)$ be the distance of the shortest path from s to v of length at most i. Then it for all i and v, Bellman-Ford maintains the invariant that at the end of pass i, $d^i(v) \leq \hat{d}^i(v)$.

In our statement of FAST_SSSP in Figure 2.1, all the work of the algorithm is done by procedure PASS, which updates the distance labels. The standard Bellman Ford algorithm implements PASS by relaxing (applying equation (1)) to all m edges yielding a total running time of $O(nm)$.

Implementing PASS. Our improvements will come from implementing carefully procedure PASS, to ultimately run in $O(n \log n)$ expected time. We begin by observing that the correctness of Bellman-Ford does not require that we compute the $d^i(v)$'s during iteration i , but rather, we only need compute the $\hat{d}^i(v)$'s, for after $n-1$ iterations, we will then have computed $\hat{d}^{n-1}(v)$, which is exactly what we want.

This allows us to make the following modifications:

Modification 1. *We can relax all edges in "parallel".* Typically, a sequential implementation of Bellman-Ford does not do this, as it only slows the algorithm down. However, we can accomplish this by replacing condition (1) with the following condition:

$$d^{i-1}(u) + c(u,v) < d^i(v). \tag{2}$$

If we apply this rule instead, it will clearly be the case that after pass i, $d^i(v) = \min_{u \in V}\{d^{i-1}(u) + c(u,v)\}$. However, since we are only allowing the length of a path to grow by at most one edge in each pass, we can show inductively that we are actually computing $\hat{d}^i(v)$ in iteration i.

Modification 2. *During pass i only vertices already at length exactly $i-1$ from the source need to have their outgoing edges relaxed.* Consider a vertex v for which $d^{i-1}(v) = d^{i-2}(v)$, i.e. a vertex that is not at length exactly $i-1$ from the source. Then this vertex had its outgoing edges relaxed at some prior phase $j \leq i-1$, and hence for all neighbors w,

$$d^i(w) \leq d^j(w) \leq d^j(v) + c(v,w) = d^{i-1}(v) + c(v,w).$$

Thus, for all outgoing edges (v,w), $d^i(w) \leq d^{i-1}(v) + c(v,w)$, and applying (2) to v's outgoing edges will not cause any vertex labels to be updated.

In light of the two observations above, we observe that during pass i, relaxing all edges in the set I_v of incoming edges to vertex v may be too much. Of all the edges in I_v only one is crucial, namely the one that will give the minimum value for $d^i(v)$. Of course, we don't know which edge this is *a priori*, but we will capture a set of edges that will contain it. To do this, we recast the implementation of pass i as a modified SSSP problem in an auxiliary network. Let a *single-source, two-level network* be a network $G_\varepsilon = (V_\varepsilon, E_\varepsilon, c_\varepsilon)$ where $V_\varepsilon = s \cup V_1 \cup V_2$ and edges connect only the source to vertices in V_1 and the vertices in V_1 to vertices in V_2. The cost function c ranges over the reals. We show how an auxiliary single source, two-level network can be used in the implementation of PASS.

We define now a particular G_ε in terms of the input to PASS, namely $G = (V, E, c)$ and $d^{i-1}(.)$. Let V' be the set of vertices in G whose current shortest path estimate derives from a path of length exactly $i-1$, i.e. $d^{i-1}(v) < d^{i-2}(v)$. Then V_1 contains a copy of every vertex in V'. V_2 contains a copy of every vertex $v \in V - s$. We introduce two types of edges in E_ε. First, for each pair $(x,y) \in V_1 \times V_2$, where x is a copy of u and y is a copy of v we add an edge (x,y) with $c_\varepsilon(x,y) = c(u,v)$. Second for all $x \in V_1$, x a copy of v, we add an edge (s,x) with $c_\varepsilon(s,x) = d^{i-1}(v)$.

We use these definitions in the upcoming lemma.

Lemma 2.1 *Let $T = T(n,m)$ be the time to solve a* SSSP *problem in a single-source, two-level network with $\Theta(n)$ vertices and $O(m)$ edges. Then, routine* PASS(G, d^{i-1}) *can be implemented to compute $d^i(.)$ in $O(T+n)$ time. Hence,* FAST_SSSP *is correct and has time complexity $O(n^2 + nT + m \log n)$.*

Proof. The implementation of pass i reduces to solving SSSP in the auxiliary network G_ε followed by a finishing step. Let SSSP_AUX(G_{aux}, S) be any routine that solves SSSP on a network G_{aux} in time T where S is the set of vertices in G_{aux} with known distances. We implement PASS as follows:

```
procedure PASS(G, d^{i-1})
        d_ε = SSSP_AUX(G_ε, s ∪ V_1);
        d^i(v) = min{d_ε(v), d^{i-1}(v)};
```

By the construction of G_ε, we can know that the value of $d_\varepsilon(v)$ for $v \in V_1$ is equal to $d^{i-1}(v)$. Thus computing distances $d_\varepsilon(v)$ in G_ε for the vertices in V_2 is equivalent to computing

$$\min_u \{d^{i-1}(u) + c(u,v)\}, \ \forall v$$

in G. Note that this is the same as if (2) were applied to all edges in G. Thus in the last step of Pass, when $d^i(v)$ is calculated as shown, it is equivalent to the process described above where d^i is first initialized to d^{i-1} and then (2) is applied to every edge. Inductively if the given $d^{i-1}(.)$ is correct, $d^i(.)$ as output by PASS captures the shortest paths of length at most i in G and thus routine PASS is correct.

For the running time, we notice that G_ε does not need to be explicitly constructed. All that is required is to identify the set of vertices whose current shortest path length is exactly $i-1$, and hence setting up the modified SSSP computation can easily be done in $O(n)$ time. The last step requires an additional $O(n)$ time, so the total time required for PASS is $O(T+n)$ and the lemma follows. ■

2.2 An implementation with fast average case

Our goal now is to implement the SSSP_AUX routine to run in $o(n^2)$ time on the particular network G_ε. We are unaware of a method achieving such a worst case bound but we show how to do it in $O(n \log n)$ expected time on networks with random edge costs. In this section we define the class of probability measures for which our analysis holds and then present an algorithm by Moffat and Takaoka [MT87] and show that it can be used to efficiently find shortest paths in G_ε.

We define first the randomness model used for the analysis. The definition follows [B83] except that we allow negative costs as well. Let $\mathcal{G}_n$ be the set of all n vertex directed networks and suppose P is a probability measure on $\mathcal{G}_n$. We may identify $\mathcal{G}_n$ with the set of all $n \times n$ matrices with entries in $(-\infty, +\infty)$. P is uniquely characterized by its distribution function $F_P : \mathcal{G}_n \to [0, 1]$ defined by

$$F_P(G) = P\{G' \in \mathcal{G}_n | c_{G'}(i,j) \leq c_G(i,j) \text{ for } 1 \leq i,j \leq n\}.$$

We say that P is *endpoint-independent* if, for $1 \leq i,j,k \leq n$ and $G \in \mathcal{G}_n$, we have that

$$F_P(G) = F_P(G'),$$

where G' is obtained from G by interchanging the costs of edges (i,j) and (i,k). Intuitively exchanging endpoints of edges emanating from any fixed vertex does not affect the probability.

Several natural edge cost assignments are endpoint-independent, including the one used by Spira [S73] where edge costs are independently, identically distributed random variables. Another probability measure that meets the endpoint independence criterion is when each source vertex i has a probability measure P_i associated with it and the entries $c_G(i,j)$ of the matrix are drawn independently according to distribution P_i. The reader is referred to [B83] for further examples.

We proceed with a high level description of the Moffat-Takaoka method. Moffat and Takaoka give a SSSP algorithm with expected running time $O(n \log n)$ under the above input model for a nonnegative cost assignment, assuming all of the edge lists are sorted. Their algorithm is similar to Dijkstra's [D59] and exploits the fact that vertices extracted in increasing cost order from a priority queue cannot have their cost decreased later on; once a vertex v is removed from the priority queue, its distance estimate is the cost of the actual shortest path from the source to v. Of course this is not true, in general, when edges with negative costs are present.

We now review the algorithm of Moffat and Takaoka. For concreteness, we will refer to the algorithm as MT and assume that is takes, as input, a network G, and a set S of vertices whose shortest path distances are already computed. Recall that it relies on all edges having non-negative costs. The set S of *labeled* vertices is maintained throughout. These are the vertices for which the shortest paths are known. For every element v in S a *candidate* edge (v,t) is maintained, known to be the least cost unexamined edge out

of v. Every candidate edge gives rise to a candidate path, there are at most $|S|$ of them at any one time. The candidate paths are maintained in a priority queue. At every step we seek to expand S by picking the least cost candidate path p. If p with final edge (v,t) leads to an unlabeled vertex t, it is deemed to be *useful*, and t is added to S. Otherwise, the path is ignored, but in both cases the candidate path (v,t) is replaced by a different path ending in (v,t') with $c(v,t') \geq c(v,t)$. There are two extremes in the policy for picking this next candidate (v,t'): either select simply the next largest edge out of v or scan the sorted adjacency list until a useful edge (v,t') is found. Based on the policy selection we obtain Spira's [S73] and Dantzig's [D60] methods respectively. The Moffat-Takaoka routine, MT, uses Dantzig's algorithm up to a critical point with respect to the size of S and then switches to Spira's. Using standard binary heaps we can obtain the following theorem.

Lemma 2.2 *Let G be a network of n vertices with nonnegative edge costs drawn from an endpoint independent distribution and S a set of vertices of G whose shortest path distances have been computed. Then* MT(G, S) *solves* SSSP *on G in $O(n \log n)$ expected time given that the edge lists are presorted by cost.*

Proof. See Theorem 1 in [MT87]. ■

We note that, in general, the nonnegativity condition is necessary for correctness as e.g. in Dijkstra. However, we will now show that for a a single-source, two-level network with real costs, such as G_ε, MT computes correctly shortest paths.

Lemma 2.3 *Let $G_\varepsilon = (s \cup V_1 \cup V_2, E_\varepsilon, c_\varepsilon)$ be the single-source, two-level network defined above. Then* MT*(G_ε, $s \cup V_1$) solves* SSSP *on G_ε in $O(n \log n)$ expected time given that the edge lists are presorted by cost, and the edge costs are drawn from an endpoint independent distribution.*

Proof (Sketch). MT will induce a total order on the candidate paths to vertices in V_2. Since the length of any path in G_ε cannot exceed 2, this total order computes correctly the shortest paths. For the running time, it suffices to notice that the analysis in Theorem 1 of [MT87] relies on the following fact. In an endpoint independent distribution when $|S| = j$ each candidate leads to each of the $n - j$ unlabeled vertices with equal probability. This fact is not harmed by the negativity of a candidate edge. Another way to look at this is the following. A sufficiently large constant C may be added to all edge costs in G_ε to make them nonnegative. This reweighting increases the costs of all paths to $v \in V_2$ by $2C$, so MT can be used to compute the $d_\varepsilon(v)$'s correctly. In the context of FAST_SSSP where $n-1$ different single-source, two-level networks are processed we notice that reweighting the $O(m)$ edges from $V_1 \times V_2$ can be done once for all the $n-1$ auxiliary networks. Thus by Lemma 2.2 MT on G_ε has $O(n \log n)$ expected time. ■

By Lemmata 2.1, 2.3 we obtain the following result.

Theorem 2.1 *Let G be a network of n vertices with real edge costs drawn from an endpoint independent distribution. Algorithm* FAST_SSSP *solves* SSSP *on G in $O(n^2 \log n)$ expected time if no negative cycle exists. Otherwise it reports the existence of a negative cycle in the same time bound.*

The worst case complexity of the Moffat-Takaoka subroutine is $O(n^2)$ therefore the worst case running time of our algorithm is $O(n^3)$.

3 A Lower Bound

For over thirty years Bellman-Ford has been the fastest algorithm for the general SSSP problem with an $O(nm)$ worst case running time. It is thus natural to wonder whether one can show a lower bound on the running time of SSSP in networks with real-valued edge costs. We do not know how to do this, but we can show a lower bound in a restricted model of computation that captures Bellman-Ford-like algorithms.

Observe that the Bellman-Ford algorithm disregards any information about the network except the number of edges, and performs a fixed sequence of edge relaxations until no vertex label can be decreased.

Motivated by this we define an *oblivious* algorithms for SSSP to be one that, given as input a network G with m edges numbered $1, \ldots, m$, decides on a sequence $S_m = e_{i_1} e_{i_2} \ldots e_{i_k}$ of k edge relaxations, $1 \leq e_{i_j} \leq m$ on the basis of m alone. An oblivious algorithm performs only the chosen relaxations and terminates when no distance estimate can be decreased. We charge unit cost for every relaxation. Obviously Bellman-Ford falls into the oblivious class, as do generalizations that do not restrict the algorithm to operate in phases. In the following theorem we prove that the Bellman-Ford algorithm is asymptotically optimal within this model. We call an edge *optimal* if it belongs to a shortest path.

Theorem 3.1 *Let $\mathcal{A}$ be any oblivious* SSSP *algorithm. There exists a network G with with n vertices and m edges on which $\mathcal{A}$ requires $\Omega(nm)$ worst case time.*

Proof. We restrict our attention to networks G, with $m > n$, in which the maximum length of a shortest path is exactly $n-1$, i.e. the shortest path tree is a single path. Let k be the length of the relaxation sequence $S_m = e_{i_1} e_{i_2} \ldots e_{i_k}$ that $\mathcal{A}$ performs on input G. A *subsequence* σ of S_m is any sequence of the form $\sigma = e_{i_{j_1}} e_{i_{j_2}} \ldots e_{i_{j_l}}$ with $1 \leq j_1 < j_2 < \ldots j_l \leq k$. The correctness of any relaxation-based algorithm comes by guaranteeing that the edges in the shortest path tree are relaxed in an order that is a topological sort of the edges of the tree. In our case, in which the shortest path tree is a path, this means that the edges of the path must be relaxed in the linear order specified by the path. Before seeing the input, it is possible that any sequence s_n of $n-1$ out of the m edges can be the chain of optimal edges, and there are $\frac{m!}{(m-(n-1))!}$ such chains. $\mathcal{A}$ will correctly compute the shortest path distances only if s_n is a subsequence of S_m. The relaxation sequence of length k has exactly $\binom{k}{n-1}$ subsequences of length $n-1$. In order for the algorithm $\mathcal{A}$ to be correct on all networks, it must be that

$$\binom{k}{n-1} \geq \frac{m!}{(m-(n-1))!} \tag{3}$$

For simplicity of presentation we replace $n-1$ by n, this only affects the calculations below by an $O(1)$ factor. We want to compute k_0, the smallest possible value of k, for which (3) holds. Increasing the left hand side and decreasing the right hand side can only make this lower bound smaller. Let $K(r(k))$ denote the minimum k such that $r(k)$ holds. We assume, wlog, that $0 \leq n \leq k$. We can upper bound the left-hand side of (3) by using the identity [CLR90]

$$\frac{k^k}{n^n (k-n)^{k-n}} \geq \binom{k}{n}. \tag{4}$$

We can lower bound the right-hand side of (3) by

$$\frac{m!}{(m-n)!} = m(m-1)\ldots(m-n+1)) \geq (m-n)^n. \tag{5}$$

By (4) and (5), we get

$$k_0 = K\left(\binom{k}{n} \geq \frac{m!}{(m-n)!}\right) \geq K\left(\frac{k^k}{n^n (k-n)^{k-n}} \geq (m-n)^n\right) = k_1$$

But $k^{k-n}/(k-n)^{k-n} \leq e^n$ thus we have that

$$\begin{aligned} k_1 &\geq K\left(\frac{k^n e^n}{n^n} \geq (m-n)^n\right) \\ &= K\left(k^n \geq \left(\frac{n(m-n)}{e}\right)^n\right) \\ &= K\left(k \geq \frac{n(m-n)}{e}\right). \end{aligned}$$

It follows that $k_0 = \Omega(nm)$. ■

4 Conclusion and Open Problems

In this paper we provided the first algorithm in over 30 years that achieves $o(n^3)$ running time for SSSP with arbitrary real costs, although not worst case. The complexity gap between $O(nm)$ time for for dense graphs and $O(n^2 \log n)$ time is considerable. This raises the natural question of whether an $o(nm)$ worst case time algorithm for SSSP exists. If the answer is no, can one reason about a lower bound in a non-oblivious computational model.

Acknowledgement. The first author wishes to thank posthumously Donald Johnson for starting him working on shortest paths and all the valuable discussions.

References

[AMO93] R. K. Ahuja, T. L. Magnanti and J. B. Orlin. *Network Flows,* Prentice Hall, Englewood Cliffs, NJ, 1993.

[AGM91] N. Alon, Z. Galil and O. Margalit. *On the exponent of the all pairs shortest path problem,* Proc. 32nd IEEE Symp. Found. Comp. Sci., 1991, 569-575.

[B58] R. Bellman. *On a routing problem,* Quarterly of Appl. Math., 16 (1958), 87-90.

[BMF79] P. A. Bloniarz, A. Meyer and M. Fischer. *Some observations on Spira's shortest path algorithm,* Tech. Rept. 79-6, Comp. Sci. Dept., State University of New York at Albany, Albany, NY, 1979.

[B83] P. A. Bloniarz. *A shortest path algorithm with expected time* $O(n^2 \log n \log^* n)$, SIAM J. Comput., 12 (1983), 588-600.

[CL77] J. S. Carson and A. M. Law. *A note on Spira's algorithm for the all-pairs shortest path problem,* SIAM J. Comput., 6 (1977), 696-699.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[D60] G. B. Dantzig. *On the shortest route through a network,* Management Sci., 6 (1960), 187-190.

[D59] E. W. Dijkstra. *A note on two problems in connection with graphs,* Numerische Mathematics, 1 (1959), 269-271.

[FF62] L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks,* Princeton University Press, Princeton, NJ, 1962.

[FT87] M. L. Fredman and R. E. Tarjan *Fibonacci heaps and their uses in improved network optimization problems,* J. ACM, 34 (1987), 596-615.

[FG85] A. M. Frieze and G. R. Grimmet. *The shortest-path problem for graphs with random arc-lengths,* Discrete Applied Mathematics, 10 (1985), 57-77.

[GT87] H. N. Gabow and R. E. Tarjan.*Faster scaling algorithms for network problems,* SIAM Journal on Computing, 18 (1989), 1013-1036.

[GJ79] M. S. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness,* W. H. Freeman, San Francisco, CA, 1979.

[G95] A. V. Goldberg. *Scaling algorithms for the shortest path problem,* SIAM J. Comput., 24 (1995), 494-504. Preliminary version in Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms, 1993, 222-231.

[KKT] D. R. Karger, P. N. Klein and R. E. Tarjan. *A randomized linear-time algorithm to find minimum spanning trees,* J. of ACM, to appear.

[KS93] D. R. Karger and C. Stein. *An $\tilde{O}(n^2)$ algorithm for minimum cuts,* Proc. 25th ACM Symp. on Theory of Computing, 1993, 757-765. To appear in J. ACM.

[K55] H. W. Kuhn. *The hungarian method for the assignment problem,* Naval Research Logistics Quarterly, 2 (1955) ,83-97.

[MT87] A. Moffat and T. Takaoka. *An all pairs shortest path algorithm with expected time $O(n^2 \log n)$,* SIAM J. Comput., 16 (1987), 1023-1031.

[S92] R. Seidel. *On the all-pairs-shortest-path problem,* Proc. 24th ACM Symp. on Theory of Computing, 1992, 745-749.

[S73] 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., 2 (1973), 28-32.

[TM80] T. Takaoka and A. M. 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, Lecture Notes in Computer Science, Vol. 88, P. Dembinski, ed., Springer Verlag, Berlin, 1980, 643-655.