Hybrid Bellman-Ford-Dijkstra Algorithm

Yefim Dinitz and Rotem Itzhak
Department of Computer Science
Ben-Gurion University of the Negev
POB 653, Beer-Sheva 84105, Israel
E-mail: {dinitz,rotemitz}@cs.bgu.ac.il

Technical Report CS-10-04

June 6, 2010

Abstract

Consider the single-source cheapest paths problem in a digraph with negative edge costs allowed. A hybrid of Bellman-Ford and Dijkstra algorithms is suggested, improving the running time bound upon Bellman-Ford for graphs with a sparse distribution of negative cost edges. The algorithm iterates Dijkstra several times without re-initializing values $d(v)$ at vertices. At most $k+2$ executions of Dijkstra solve the problem, if for any vertex reachable from the source, there exists a cheapest path to it with at most k *negative cost* edges.

In addition, a new, straightforward proof is suggested that the Bellman-Ford algorithm results in a cheapest path tree from the source.

1 Introduction

Several basic graph algorithms are widely known from the 50th-60th of the past century. Among them are single-source cheapest path algorithms and the PERT algorithm for finding the early project schedule in a DAG (directed acyclic graph) via finding the longest paths in it (see, e.g., [3, Chapter 25.4]). Though the basic graph algorithms have being massively taught at all CS departments all over the world since then, it turns out that new aspects can be revealed. We present two such results on cheapest paths algorithms: a new algorithm and a new proof.

In what follows, we say "graph" meaning a digraph with edge costs. The input graph is denoted by $G = (V, E, c)$. An edge or cycle of a negative cost are called "negative". When relating to classic results, we usually do not provide references; the reader can refer to any textbook in algorithms, e.g., [3]. The main goal of this paper is *worst case* running time bounds, for algorithms.

A special kind of innovations is hybrid algorithms: a combination of known algorithms for solving either a new problem or an old problem with a new time bound. For example, the algorithm of [7, 6] combines PERT and Dijkstra algorithms for finding the early schedule in a *general* graph with precedence relations of AND or OR type at nodes (PERT and the single-source cheapest paths problems are boundary cases of this problem). Surprisingly, though these algorithms seem quite different, the hybrid algorithm combining them is natural, and its running time is the sum of running times of PERT and Dijkstra.[1]

This paper presents a new, hybrid algorithm for finding cheapest paths from a source s in a graph G with *general* edge costs. It combines Bellman-Ford and Dijkstra algorithms, and will be called Bellman-Ford-Dijkstra (BFD). Recall that both former algorithms pass over the graph edges, executing the update (called also " relabeling" or "relaxation") of the tentative function d on vertices. Dijkstra works for the non-negative edge costs case. It is a search type algorithm: it makes just a single pass on all vertices and edges reachable from s, in a greedy order, while never recomputing the d values at scanned vertices.

For graphs with negative edges, the Bellman-Ford algorithm is used. In

[1] A similar algorithm for some grammar problem was suggested in [10], without paying attention to its hybrid nature.

contrast, it works in rounds, each consisting of a simple loop of relaxations on the graph edges, in an arbitrary order. The known bound on the number of its rounds is $r+1$, *where r is the minimal integer such that for any vertex reachable from s, there exists a cheapest path from s to it containing at most r edges*. If the input graph contains a negative cycle reachable from s, the cheapest paths do not exist, and Bellman-Ford detects that in $|V|$ rounds. Otherwise, r as above is at most $|V|-1$. Since each round costs $O(|E|)$, the implied running time bound is $O(|V||E|)$.[2] Notice that this is by an order of magnitude more than the Dijkstra bound $O(|E|+|V|\log|V|)$.

Our goal is to decrease the (worst case) round number bound of Bellman-Ford. We achieve it by using a *smart* loop of relaxations at each round. The idea of BFD is to iterate Dijkstra at Bellman-Ford rounds, without re-initializing values of d at vertices. We show that this works, despite the common knowledge announced everywhere: "Dijkstra's algorithm cannot handle graphs with negative edge costs". Our bound for the number of BFD rounds is $k+2$, *where k is the minimal integer such that for any vertex reachable from s, there exists a cheapest path from s to it containing at most k* ***negative*** *edges.* This is an essential improvement over the Bellman-Ford bound for graphs with a sparse dispersion of negative edges. (It should be mentioned that the running time of a BFD round increases from $O(|E|)$ to $O(|E|+|V|\log|V|)$, which is slightly worse for sparse graphs.)

That is, BFD is faster than Bellman-Ford for in-between cases, when there are either just a few negative edges in the graph, or many such edges but sparsely dispersed in the graph. A motivation of such cases comes naturally from classic, seemingly purely non-negative problem settings. For example, consider a GPS problem of finding a shortest route in a road map. Suppose that a driver chooses a few objects of interest, such as a beautiful view or a good restaurant, and assigns to each of them a route cost reduction. Then, the road map can acquire several negative edges.

It should be mentioned that there is a wide study on *practical* cheapest paths algorithms, whose running time for known benchmarks is drastically lower than that defined by known worst case bounds. For example, see [1, 2] and references therein. In particular, in [2, Section 5.3] a variant of the Bellman-Ford algorithm is mentioned, where at each round the edges are passed in the order of the values of function d at their initial vertices, which

[2] With the additional assumption that edge lengths are integers bounded below by $-N \le -2$, Goldberg [8] suggests an algorithm with the $O(\log N\sqrt{|V|}|E|)$ bound.

is the same order as that of BFD. However, this variant was rejected there from consideration, for practical running time reasons.

The correctness proof of BFD is a generalization of a usual such proof for Bellman-Ford. Hence, BFD may become an instructive supplement for a university course in algorithms.

In addition, we suggest a new proof of a classic property of the Bellman-Ford algorithm. Proofs that Bellman-Ford computes a cheapest paths tree from s remained not simple for decades. For example, in both editions of a popular textbook [3], it is about three pages long. Other, much shorter proofs appear in [12, 9]. However, they are indirect: the key lemma proves that back-pointers π never form a cycle, and it implies the statement. In contrast, the proof of a similar statement for the Dijkstra algorithm is straightforward: By the algorithm, beginning from the initial tree consisting of root s only, each iteration adds a new leaf edge to the back-pointer graph; clearly, its property of being a tree rooted at s is maintained. We present a similar proof for Bellman-Ford.

2 Hybrid Bellman-Ford-Dijkstra Algorithm

2.1 Algorithm

Recall the basic Bellman-Ford (BF) and Dijkstra algorithms.

Initialize
    $d(v) \leftarrow \infty$, for all $v \in V$
    $\pi(v) \leftarrow nil$, for all $v \in V$
    $d(s) \leftarrow 0$

***Relax*(u, v)**
    if $d(v) > d(u) + c(u, v)$
        $d(v) \leftarrow d(u) + c(u, v)$
        $\pi(v) \leftarrow u$

Plain_scan
    for each edge $(u, v) \in E$
        ***Relax*(u, v)**

Dijkstra_scan
    $S \leftarrow \emptyset$
    while (there is a vertex in $V \setminus S$ with $d < \infty$) **do**
        find vertex u in $V \setminus S$ with the minimal value of d
        $S \leftarrow S \cup \{u\}$
        for each edge $(u, v) \in E$ /* scanning of u */
            ***Relax*(u, v)**

***Dijkstra*(G, s)**
    Initialize
    Dijkstra_scan
    return(d, π)

***Bellman-Ford*(G, s)**
    Initialize
    $i \leftarrow 1$
    do
        Plain_scan
        i++
    until ((there was no change of d at ***Plain_scan***) **or** $(i = |V|)$)
    if $(i < |V|)$ **return**(d, π)
    else return("There exists a negative cycle reachable from s.")

Algorithm Bellman-Ford-Dijkstra (BFD) is as follows:

***Bellman-Ford-Dijkstra*(G, s)**
    Initialize
    $i \leftarrow 1$
    do
        Dijkstra_scan
        i++
    until ((there was no change of d at ***Dijkstra_scan***) **or** $(i = |V| - 1)$)
    if $(i < |V| - 1)$ **return**(d, π)
    else return("There exists a negative cycle reachable from s.")

Notice that BFD may be considered a particular version of BF, since at each round, *Relax* is executed on all edges reachable from s.

Although this paper concentrates on worst case time bounds, we consider also the basic practical variants of BF and of BFD, for completeness. We denote them by BF-p and BFD-p, respectively. They group $Relax(u, v)$ executions to bunches with the same vertex u, called "scanning u" (in $Dijkstra\_scan$ this comes automatically). At each round, for each vertex u the value of $d(u)$ at the time of scanning u at that round is stored. At the next round, vertices u are scanned *only if* their current d values are strictly less than their d values stored at the previous round (since otherwise, their scanning is useless). Though the worst case bound of BF-p remains the same as that of BF, it is known that in practice, this modification decreases the running time significantly. Various choices of the vertex scanning order in BF are used; see, e.g., [2] for their comparison.

2.2 Analysis

Recall known properties of *Relax*-based algorithms. We denote by $opt(v)$ the cost of a cheapest path from s to v in G. We set $opt(v) = \infty$ if v is not reachable from s.

Fact 2.1 *Consider an arbitrary (properly initialized) sequence of Relax executions.*

1. *At any moment and for any vertex v, holds $d(v) \geq opt(v)$.*
2. *Values of d may only decrease. Therefore after $d(v)$ reaches $opt(v)$, neither $d(v)$ nor $\pi(v)$ do not change.*
3. *If there is a negative cycle reachable from s, then at any moment there exists an edge (u, v) with $d(u) + c(u, v) < d(v)$.*

For a path P, let us define $neg(P)$ be the number of negative edges in P, not including its first and last edges, if negative. For a vertex v reachable from s, we define $neg(v)$ be the minimal value of $neg(P)$ over all cheapest paths from s to v. We call the path providing that minimum *neg-optimal* for v. We formally set $neg(s) = -1$, and $neg(v) = \infty$ if v is not reachable from s. We define $neg(G, s)$ to be the maximal *finite* value of $neg(v)$. It is known that if there exists a cheapest path from s to v, then there exists a simple such path. Therefore, neither $neg(v)$ nor $neg(G, s)$ can exceed $(|V| - 1) + 2 = |V| - 3$.

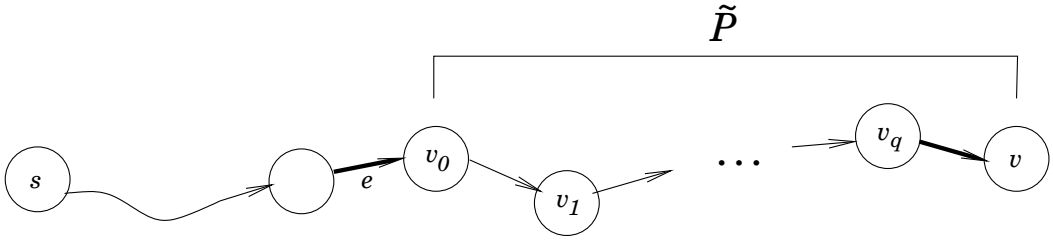

Figure 1: Path P and its sub-path $\tilde{P}$. Negative edges are shown as thick lines. The last edge could be not negative. The part of P before $\tilde{P}$ could be empty ($s = v_0$), and then (v_0, v_1) could be negative.

Proposition 2.2 *If there exists a cheapest path from s to v, after $neg(v)+1$ BFD rounds, holds $d(v) = opt(v)$. This holds for BFD-p as well.*

Proof: We prove for BFD by induction on $neg(v)$. Clearly, the statement is correct for the basis case $neg(v) = -1$ (that is, for $v = s$).

We now assume that the statement is correct for all v', $neg(v') < k$, $k \geq 0$, and will prove it for v, $neg(v) = k$. Let P be a neg-optimal path to v. Let e be the last negative edge along P, which is neither its first nor its last edge, if it exists (that is, if $k \geq 1$). We denote by $\tilde{P}$ the final part of P after e, if it exists, and set $\tilde{P} = P$ otherwise. Let $\tilde{P} = (v_0, v_1, \ldots, v_q, v)$ and denote $\tilde{V} = \{v_0, v_1, \ldots, v_q\}$. Notice that in any case, $neg(v_0) = k - 1$, by definition. Hence, $d(v_0) = opt(v_0)$ after k rounds, by the induction assumption. For illustration, see Figure 1.

Consider the $(k+1)$th round of BFD. At first, we prove for the case, when all non-negative edges of P have a *positive* cost. Let us prove by induction that *the vertices $v_i \in \tilde{V}$ enter S with $d(v_i) = opt(v_i)$ in the increasing order of i*. As the basis, we prove that v_0, with $d(v_0) = opt(v_0)$, is scanned first in $\tilde{V}$. If $k = 0$, then $v_0 = s$, while s is always scanned first in the first *Dijkstra_scan*, by initializing.

Assume now $k \geq 1$. Then, by the definition of $\tilde{P}$ and the case assumption, all edges in $\tilde{P}$, except maybe the last one, have a positive cost. Since any prefix of P is an optimal path to its end-vertex, function *opt strictly grows* along $\tilde{P}$ on $\tilde{V}$. Therefore, by Fact 2.1(1), for any v_i, v_j, $i < j$, always holds $opt(v_i) < opt(v_j) \leq d(v_j)$. In particular, during the $(k+1)$th round, $d(v_0) = opt(v_0)$ always is the *unique* minimal value of d on $\tilde{V}$. Hence by the Dijkstra rule, v_0 enters S first among the vertices in $\tilde{V}$. This is the end of the induction basis.

At the induction step, we assume that v_i, $i < q$, entered S with $d(v_i) = opt(v_i)$ before v_{i+1}, and prove that v_{i+1} enters S next in $\tilde{V}$ with $d(v_{i+1}) = opt(v_{i+1})$. When v_i enters S, it is immediately scanned; during its scan, $d(v_{i+1})$ gets its final value $opt(v_{i+1})$ via the relaxation on edge (v_i, v_{i+1}), if $d(v_{i+1})$ was not equal it before that. After that at the $(k+1)$th round, $d(v_{i+1})$ is always minimal among the values of d on $\{v_{i+1}, \ldots, v_q\}$, by the reasons similar to those in the basis proof for $k \geq 1$. Therefore, v_{i+1} enters S first among the vertices in $\{v_{i+1}, \ldots, v_q\}$. This is the end of the inner induction proof.

Observe that while scanning v_q at the $(k+1)$th round of BFD, also $d(v)$ gets its final value $opt(v)$ via the relaxation on edge (v_q, v), as required in the step of the outer induction. This is the end of the outer induction proof.

Now, let edge costs on $\tilde{P}$ be general. This case differs in that there may be sub-paths of $\tilde{P}$ consisting of edges of zero cost only, so that *opt* is a constant at each of them, and those constants strictly grow along $\tilde{P}$. The only difference from the analysis of the previous case is that the vertices v_i, $i \geq 1$, of such a sub-path may enter S in *any* order, provided $d(v_i) = opt(v_i)$ at the moment of entering S. Indeed, suppose that vertex v_j enters S, while the first vertex on $\tilde{P}$ not in S is v_{i+1}, $i+1 < j$, and v_i is in S with $d(v_i) = opt(v_i)$. By the analysis as above, we have $d(v_{i+1}) = opt(v_{i+1})$ at that moment. By the Dijkstra rule, $d(v_j) \leq d(v_{i+1}) = opt(v_{i+1})$, and thus $d(v_j) \leq opt(v_{i+1}) \leq opt(v_j) \Rightarrow d(v_j) = opt(v_{i+1}) = opt(v_j)$. This closes the gap between the restricted and the general cases.

For BFD-p, the difference is that there is no guarantee that all vertices are scanned at each round. The corresponding change in the proof that $d(v)$ converges to $opt(v)$ is as follows: Let us consider the first round, k', where some vertex v_m on $\tilde{P}$ acquires $d(v_m) = opt(v_m)$. By the reasons as above, holds $k' \leq k$. The inner induction in the proof is then applied to round $k'+1$ instead of round $k+1$, and its base is v_m instead of v_0. At the induction step, we show, in a similar way, that vertices v_i, $m+1 \leq i \leq q$, subsequently acquire values $opt(v_i)$. By the choice of v_m, this means that their d values really decrease. Therefore, they *should* be scanned at the same round, as required. □

Theorem 2.3 *If there is no negative cycle reachable from s in G, BFD correctly computes $opt(v)$ for all $v \in V$ and the cheapest path tree, in at most $neg(G, s) + 2$ rounds. Otherwise, it reports on the existence of such a cycle. This holds for BFD-p as well.*

Proof: By Proposition 2.2, if there is no negative cycle reachable from s, after $neg(G,s)+1$ BFD rounds, d values equal *opt* values at all vertices. By Fact 2.1(2), at the round numbered at most $(neg(G,s)+2)$ there is no change of d values, and BFD stops. Since $neg(G,s) \leq |V|-3$, this should happen not later than after round $|V|-1$.

Since *Dijkstra_scan* executes *Relax* on all edges reachable from s, it may be considered a specific implementation of *Plain_scan*. Therefore, BFD may be considered a specific implementation of the generic BF. Since BF is known to produce the cheapest path tree from s, this holds for BFD as well.

If there exists a negative cycle reachable from s, then by Fact 2.1(3), even at round $|V|-1$ there would exist an edge (u,v) with $d(u)+c(u,v)<d(v)$. Since *Dijkstra_scan* executes *Relax* on all edges reachable from s, there would be a change of d at round $|V|-1$. BFD will then stop and report accordingly.

The proof for BFD-p is similar. □

2.3 Running Time

Clearly, the running time of BFD is defined by that of *Dijkstra_scan*. It is easy to see that the implementations of Dijkstra supported by a priority queue are robust to negative edge costs. Therefore, the implementation based on Fibonacci heap provides the following bound:

Theorem 2.4 *There exists an implementation of BFD running in time* $O(neg(G,s)\cdot(|E|+|V|\log|V|))$ *for graphs* $G=(V,E)$ *with no negative cycle reachable from* s.

For a graph G *with* negative cycles reachable from s, the round number bound of BFD is about $|V|$, the same as that of BF.[3] Let us try to reduce this bound, for specific cases. The following statement is straightforward.

Observation 2.5 *For any known bound* B *for* $neg(G,s)$, *the stopping condition* $(i=|V|-1)$ *of BFD may be replaced by* $(i=B+2)$, *thus bounding the number of BFD rounds by* $B+2$.

[3] A wide analysis of practical negative cycle detection may be found in [2].

Let $\#neg(G)$ denote the minimum of the number of vertices with outgoing negative edges (excluding s) and the number of vertices with incoming negative edges, in G. Clearly, $\#neg(G)$ cannot exceed the total number of negative edges in G. By definition, we have $neg(G,s) \leq \#neg(G)$.

Corollary 2.6 *The stopping condition ($i = |V| - 1$) of BFD may be replaced by ($i = \#neg(G) + 2$), thus bounding the number of BFD rounds by $\#neg(G) + 2$.*

Remark: The arc-fixing algorithm suggested in [2, Section 5.5] iterates Dijkstra executions on modified versions of G. It is stated there (with no proof) that it solves the problem in at most $\#neg(G)$ Dijkstra runs.

Let us describe, for a not strongly connected graph G, the following bound $B^{SCC}(G)$ for $neg(G,s)$, which can be computed by a linear time preprocessing. We begin with computing DAG G^{SCC} of strongly connected components (SCCs) of G (e.g., by Kosaraju-Sharir algorithm); the edge between two SCCs is called negative if there exists a negative edge between their vertices in G. We give to each SCC C weight $\#neg(G(C))$, where $G(C)$ is the sub-graph induced by C. We define the weight of a path in G^{SCC} be the sum of the weights of its vertices plus the number of negative edges in it. We compute the maximal weight of a path from the SCC of s in G^{SCC} (it is well known how to do this using a topological sort). We denote that weight by $B^{SCC}(G,s)$. It is easy to see that $neg(G,s) \leq B^{SCC}(G,s)$, thus implying:

Corollary 2.7 *For any graph G, the value of $B^{SCC}(G,s)$ can be computed in a linear time. Then, the stopping condition ($i = |V| - 1$) of BFD may be replaced by ($i = B^{SCC}(G,s) + 2$), thus bounding the number of BFD rounds by $B^{SCC}(G,s) + 2$.*

Let us return to the running time of a BFD round. Observe that many implementations of Dijkstra are known. They provide either better worst case bounds for particular graph classes, or better constant factors in running time bounds for the general case. An implementation with a lower constant factor may be preferred, in practice, to those with a better $O(\cdot)$ bound.

Note that some implementations of Dijkstra might rely on non-negativity of edge costs, either explicitly or implicitly, and hence might work improperly if there are negative edges in G. For example, the implementation of Wagner [14] assumes edge costs to be not so large non-negative integers, and uses an

integer priority queue. It runs in time $O(|E| + opt_{max})$, with a low constant factor, where opt_{max} denotes the maximal value of a cheapest path from s. Hence, it may be preferred when $opt_{max} = O(|E|)$.

Observe that when Dijkstra scans u, if $c(u, v) < 0$ and $v \in V \setminus S$, $Relax(u, v)$ decreases $d(v)$ below $d(u)$. Since Wagner's implementation relies on the known property of Dijkstra to insert vertices to S in a non-decreasing order of their d values, it will simply *skip* scanning vertices v as above, thus implementing Dijkstra improperly.

Let us describe a robust version BFD-r of BFD, which runs $Dijkstra\_scan$ on a graph with non-negative edges only. The changes in BFD-r w.r.t. BFD as above are as follows. Denote the set of negative edges in G by E^-.

- Routines $Dijkstra\_scan$ and $Plain\_scan$ acquire an additional parameter E, thus getting the edge set to scan via this parameter, instead of using the globally defined set E.
- The call to $Dijkstra\_scan$ in the $do < \ldots > until$ loop is replaced by two consequent calls: $Dijkstra\_scan(E \setminus E^-)$ and $Plain\_scan(E^-)$.

Theorem 2.8 *The BFD-r version of BFD is correct, keeping the round number bounds, whichever implementation of Dijkstra on a graph with non-negative edge costs is used in Dijkstra_scan.*

Proof: Let us show that the proof of Proposition 2.2 (and thus of Theorem 2.3) remains valid for BFD-r. It is easy to check that the proof part analyzing the insertion of vertices in $\tilde{V}$ into S remains fully valid, being now related to the execution of $Dijkstra\_scan(E \setminus E^-)$. Consider now the concluding remark on ensuring the equality $d(v) = opt(v)$ via executing $Relax(v_q, v)$ after inserting v_q into S. It remains valid either by executing it in $Dijkstra\_scan$, if $c(v_q, v) \geq 0$, or by executing it in $Plain\_scan(E^-)$, otherwise. □

In addition, there exist implementations of *Dijkstra-like* algorithms, working with special time bounds for specific graph classes. For example, algorithm [5] works when edge costs are *positive* real numbers. It generalizes Dijkstra by relaxing the choice condition of the next vertex to scan: from $d(v)$ being minimal in $V \setminus S$ to $\lfloor d(v)/c_{min} \rfloor$ being minimal in $V \setminus S$, where c_{min} is the minimal edge cost. In accordance, it uses the integer priority

queue with key $\lfloor d(v)/c_{min} \rfloor$. Its running time bound is *scalable* w.r.t. edge costs: $O(|E| + \frac{opt_{max}}{c_{min}})$.

Let us describe yet another version BFD-r$^+$. Let E^+ be the set of edges with positive costs in G, and c^+_{min} be the minimal *positive* edge cost (that is, it is c_{min} of graph (V, E^+)). BFD-r$^+$ differs from BFD-r in that the two calls replacing $Dijkstra\_scan$ are: $Dijkstra\_scan(E^+)$ and $Plain\_scan(E \setminus E^+)$.

Accordingly, we change the measure of the BFD complexity: Let $non\_pos(G, s)$ be defined similarly to $neg(G, s)$, but with respect to edges with non-positive costs.

Proposition 2.9 *If there exists a cheapest path from s to v, after $non\_pos(v)+1$ rounds of BFD-r$^+$ holds $d(v) = opt(v)$.*

Proof: Let us show that the proof of Proposition 2.2 remains valid, with minor changes, while notion $neg(G, s)$ changes to $non\_pos(G, s)$ and each reference to a negative edge changes to that to a non-positive edge. Note that only the restricted case is needed in the analysis. Since now $c(v_i, v_{i+1}) \geq c^+_{min}$, $1 \leq i \leq q-1$, the values of $\lfloor opt(v_i)/c^+_{min} \rfloor$ strictly grow along P. Using this change, the property of vertices in $\tilde{V}$ being scanned in their order on $\tilde{P}$, with $d(v_i) = opt(v_i)$, is proved in the same way. □

The proof of Theorem 2.3 remains valid for the following theorem, now based on Proposition 2.9 instead of Proposition 2.2.

Theorem 2.10 *The BFD-r$^+$ version of BFD is correct when the Dijkstra-like algorithm and its implementation of [5] are used in $Dijkstra\_scan$, while the round number bound changes to $non\_pos(G, s) + 2$.*

Also the analogs of Corollaries 2.6 and 2.7 and their proofs remain correct, with the notation change as above.

It is easy to check that all results of this section hold for BFD-p and its respective versions – BFD-pr and BFD-pr+.

Open question: The Dijkstra choice rule relaxation of [5] was substantially extended for undirected graphs in Dijkstra-like algorithms [13] and [11]. It would be interesting to check whether these algorithms could be used in some version of BFD.

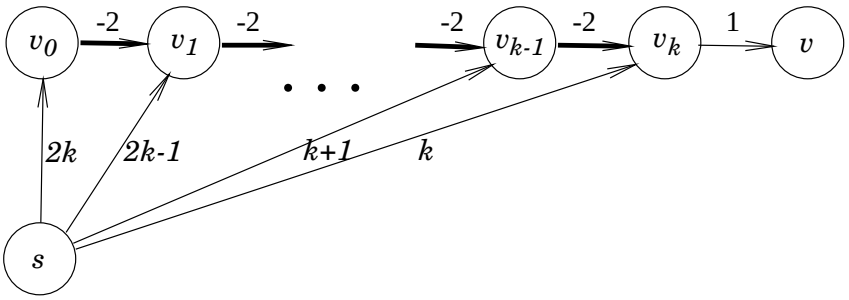

Figure 2: Tightness example.

2.4 Tightness of Bounds and a Speeding Up Idea

Consider the graph G presented in Figure 2. The cheapest path from s to v is $P = (s, v_0, v_1, \ldots, v_k, v)$, and its cost is 1. It contains all k negative edges in G as its inner edges. Hence, $neg(G, s) = k = |V| - 3$.

Let us run BFD from s. Consider the first $Dijkstra\_scan$. After scanning s, the values of d get: $d(v_i) \leftarrow 2k - i$, $i = 0, 1, \ldots, k$. Then, the vertices v_i are scanned in the order of d, that is in the inverse order of indices (from here on, scans of s and v are not mentioned, since they change no value of d). While scanning v_k, the value of $d(v)$ becomes $k + 1$, and all $d(v_i)$ decrease by 1, $i = 1, \ldots, k$. At the second $Dijstra\_scan$, the order of scanning is the same, so that $d(v)$ and $d(v_i)$, $i = 2, \ldots, k$, decrease by 1. The jth $Dijstra\_scan$, $j = 3, \ldots, k + 1$, is similar, reducing the d values by 1 for the $k + 2 - j$ last vertices of P. As a result, all d values become equal to the *opt* values. At the round $k + 2$, there is no change of d, and hence BFD finishes.

The total number of rounds is $k + 2 = neg(G, s) + 2 = |V| - 1$. This proves the tightness of the Theorem 2.3 bound. Note that also the average number of updates of d per edge is high in this example: about $\frac{1}{2}|V|$.

It is interesting that there may be a way to solve such a hard example by BFD with a small number of $Dijkstra\_scan$ executions, using the reweighting technique as in the Johnson algorithm (see, e.g., [3, Chapter 26.3]; also the entire approach of [2] is based on a similar *potential* technique).

Here is an example (for illustration see Figure 3). Let us add to the above graph G a "copy" s' of s, with edge (s', s), $c(s', s) = 0$, and edges to all v_i with costs as denoted in the figure. Notice that $neg(G, s)$ remains equal to k, and BFD running from s works exactly as above, so this example is as hard as the

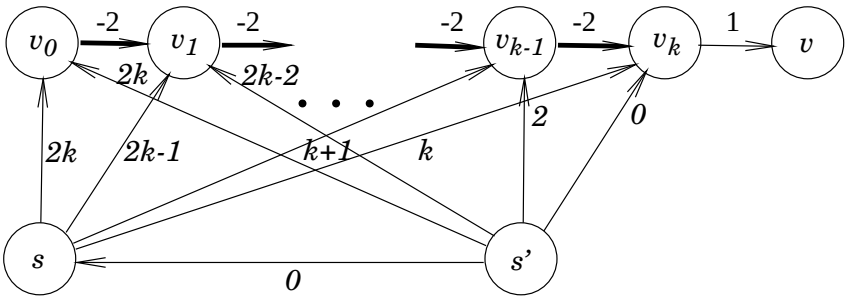

Figure 3: Example of the power of Johnson reweighting for BFD.

one above. Let us now run BFD *from* s'. Pay attention that each edge (s', v_i) is an optimal path from s to v_i. Hence $neg(G, s') = 0$, and thus running BFD from s' would require just two rounds. Note that all vertices reachable from s are reachable from s' in the new graph. Using Johnson's reweighting, we arrive at a graph with non-negative costs of all edges reachable from s. Hence, an execution of Dijkstra from s on that graph will provide a solution for the original graph. Summarizing, only three Dijkstra/$Dijkstra\_scan$ executions would suffice in total, instead of $k+2$ executions. This shows how much help can be obtained by choosing an appropriate source for an auxiliary BFD run, with subsequent reweighting.

A motivation for existence of such problem instances may continue that mentioned in Section 1. Naturally, all negative edges created by the driver would go in the direction of his motion. In such a case, probably, there would be no optimal path from the driver's target or from some far away vertex approximately equidistant from the source and the target, containing many negative edges. Then, such a choice of an auxiliary source as above will help.

Note that in general, finding such a useful auxiliary source, if exists, might be hard.

3 Shortest path tree proof for Bellman-Ford

Let us prove the following simple property:

Lemma 3.1 *Consider any (properly initialized) Relax-based algorithm. If a relaxation on edge (u, v) decreases $d(v)$ to $d(u) + c(u, v) = opt(v)$, at that moment $d(u)$ already equals $opt(u)$.*

Proof: Indeed, assume to the contrary that the optimal path P from s to u is cheaper than $d(u)$ at that moment. Then, the path $P||(u,v)$ is cheaper than $d(u) + c(u,v) = opt(v)$, a contradiction. □

At any moment of BF, denote by $\bar{V}$ the (dynamic) vertex set $\{v \in V : d(v) = opt(v) < \infty\}$.

Proposition 3.2 *At any moment of Bellman-Ford, the graph T formed by the vertices in $\bar{V}$ and the back-pointers from them is a cheapest path tree from s to the vertices in $\bar{V}$.*

Proof: We prove by induction on the size of $\bar{V}$. The basis case $\bar{V} = \{s\}$ is trivial.

Assume T is a cheapest path tree to $\bar{V}$, when the relaxation on edge (u,v) decreases $d(v)$ to $d(u) + c(u,v) = opt(v)$. By Lemma 3.1, u is in T. By the induction assumption, the path from s to u in T, P_u, costs $opt(u)$. As a result of the relaxation, the new vertex v and the leaf edge from u to v are added to T, keeping it be a tree rooted at s. The path $P_u||(u,v)$ to v in T costs $opt(u) + c(u,v) = d(u) + c(u,v) = opt(v)$. Hence, the new T is a cheapest path tree from s to the new $\bar{V}$. By Fact 2.1(2), back-pointers from vertices currently in $\bar{V}$ will never change after that. This is the end of the induction step. □

Clearly, this Proposition implies straightforwardly that Bellman-Ford builds the cheapest paths tree, when exists.

Acknowledgment

The authors thank Boris Cherkassky and Andrew Goldberg for their help with relating the paper to contemporary practical methods and more.

References

[1] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics* **15** (2010).

[2] B.V. Cherkassky, L. Georgiadis, A.V. Goldberg, R.E. Tarjan, and Renato F. Werneck. Shortest Path Feasibility Algorithms: an Experimental Evaluation. In *Proc. of 6th International Workshop on Algorithm Engineering and Experiments*, SIAM, 2008.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*, McGraw-Hill, 2001.

[4] E.A. Dinic. An algorithm for the solution of the max-flow problem with the polynomial estimation. *Doklady Akademii Nauk SSSR* **194** (1970), no. 4 (in Russian; English transl.: *Soviet Mathematics Doklady* **11** (1970), 1277–1280).

[5] E.A. Dinic. Economical algorithms for finding shortest paths in a network, in: *Transportation Systems. Models, Algorithms, Software, Analysis*, Yu.S. Popkov and B.L. Shmulyian eds., Moscow, 1978, 36–44 (in Russian).

[6] E.A. Dinic. The fastest algorithm for the PERT problem with AND- and OR-vertices (the new-product-new technology problem). In *Proc. of the Math. Progr. Soc. Conference on Integer Programming and Combinatorial Optimization (IPCO'90), Waterloo, Canada*, R. Kannan and W. R. Pulleyblank eds., Univ. of Waterloo Press, 1990, pp. 185-187.

[7] E.A. Dinic, A.B. Merkov, and I.A. Tejman. Coordination analysis and computing of early periods of launching for a set of new technologies, in: *Models and Methods for Forecast of the Science-Technology Progress*, V. V. Tokarev ed., Moscow, 1984, 125–131 (in Russian).

[8] A.V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.* **24**, 1995, 494–504.

[9] J. Kleinberg and E. Tardós. *Algorithm Design*. Pearson, Addison Wesley, 2006.

[10] D. E. Knuth. A generalization of Dijkstra's algorithm. *Information Processing Letters* **6** (1977), no.1, pp.1–5.

[11] S. Pettie and V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. SIAM J. Comp. **34** (2005), 1398-1431.

[12] R.E. Tarjan, *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1983.

[13] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM* **46** (1999), 362-394.

[14] R.A. Wagner. A shortest path algorithm for edge-sparce graphs. *J. ACM* **23** (1976), 50–57.