Hybrid Bellman-Ford-Dijkstra Algorithm

Yefim Dinitz [*]   Rotem Itzhak [†]

Abstract

We 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 the tentative value $d(v)$ 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. Though those algorithms have being massively taught at all CS departments all over the world since then, it turns out that their new aspects can be revealed. We present two such results on cheapest paths algorithms: a new, combined algorithm and a new proof.[1]

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

[*]Department of Computer Science, Ben-Gurion University of the Negev, POB 653, Beer-Sheva 84105, Israel. E-mail: `dinitz@cs.bgu.ac.il` .

[†]B.Sc. student there. E-mail: `rotemitz@cs.bgu.ac.il` .

[1]Usually, they say "shortest path". However, in the presence of negative values assigned to edges, we prefer avoiding the "negative length" notion.

are called "negative". When relating to classic results, we usually do not provide a reference; the reader can refer to any textbook on algorithms, e.g., [4]. In this paper, the main goal is finding algorithms with better *worst case* running time bounds.

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, Dijkstra algorithm solves the single-source *shortest* path problem. The PERT chart algorithm finds the early time-schedule in a DAG (directed acyclic graph), whose nodes are events and the edges represent the precedence relation between them, via finding *longest* paths in it [13] (see also [4, Chapter 24.2]). The algorithm of [9, 8] combines PERT and Dijkstra algorithms for finding the early time-schedule in a *general* graph with precedence relations of AND or OR type at nodes; the PERT chart 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.[2]

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 original algorithms pass over the graph edges, executing the update (called also "relaxation" or "relabeling") 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.

For graphs with negative edges, Bellman-Ford algorithm is used. In contrast, the basic Bellman-Ford works in rounds, each being a simple loop of relaxations on the graph edges, in any order. It finds the cheapest path costs to each vertex and the cheapest paths tree in at most $r + 1$ *rounds, 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* (in other words, r is the minimal possible depth of the cheapest paths tree). 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 a single round cost is $O(|E|)$, the implied running time

[2]A similar algorithm for some grammar problem was suggested in [12], without paying attention to its hybrid nature.

bound is $O(|V||E|)$. Notice that this is by an order of magnitude more than the Dijkstra time bound $O(|E| + |V| \log |V|)$.[3]

Our goal is decreasing the (worst case) round number bound of Bellman-Ford. We achieve it at BFD by running Dijkstra at each Bellman-Ford round, without re-initializing values of d at vertices. This is legal, since Dijkstra is just a *smart loop* of relaxations. 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 in 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, where there are a few *special edges* in the network. 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 a route cost reduction to visiting each one of them. 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 reasonable benchmarks is drastically lower than that defined by the known worst case bounds. For example, see [2, 3] and references therein. In [3, 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 is the same order as that of BFD. However, that variant was rejected there from consideration, for practical running time reasons.

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

[3]With the additional assumption that the edge lengths are integers bounded below by $-N \leq -2$, Goldberg [10] suggests an algorithm with the $O(\log N \sqrt{|V|}|E|)$ time bound, for this problem.

The paper is composed as follows. Section 2 defines and analyzes the Bellman-Ford-Dijkstra algorithm. In Section 3, we suggest a possible definition of the new algorithm class *Iterated Greedy*, including BFD.

In Appendix (Section 4), we present an improvement of the classic analysis of relaxation-based algorithms. Proofs of the fact that they compute a cheapest paths tree from s remained not simple for decades. For example, in both editions of a popular textbook [4], this proof is about three pages long. Other, much shorter proofs appear in [15, 11]. However, they are indirect: the key lemma proves that back-pointers π computed by the algorithm never form a cycle, and this lemma 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 suggest a similar proof for the general relaxation-based algorithm.

2 Hybrid Bellman-Ford-Dijkstra Algorithm

2.1 Algorithm

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

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

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

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 $u$ */
            Relax(u, v)

Dijkstra(G, s)
    Initialization
    Dijkstra_scan
    return($d, \pi$)

Bellman-Ford(G, s)
    Initialization
    $i \leftarrow 0$
    do
        $i$++
        Plain_scan
    until ((there was no change of $d$ at Plain_scan) or ($i = |V|$))
    if ($i < |V|$) return($d, \pi$)
    else return("There exists a negative cycle reachable from $s$.")
```

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

```
Bellman-Ford-Dijkstra(G, s)
    Initialization
    $i \leftarrow 0$
    do
        $i$++
        Dijkstra_scan
    until ((there was no change of $d$ at Dijkstra_scan) or ($i = |V| - 1$))
    if ($i < |V| - 1$) return($d, \pi$)
    else return("There exists a negative cycle reachable from $s$.")
```

Notice that BFD may be considered as 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 variant 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 $d(u)$ while scanning u 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 is the same as that of BF, it is known that in practice, this modification significantly decreases the running time.

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 $\pi(v)$ 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 $(|V| - 1) - 2 = |V| - 3$.

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

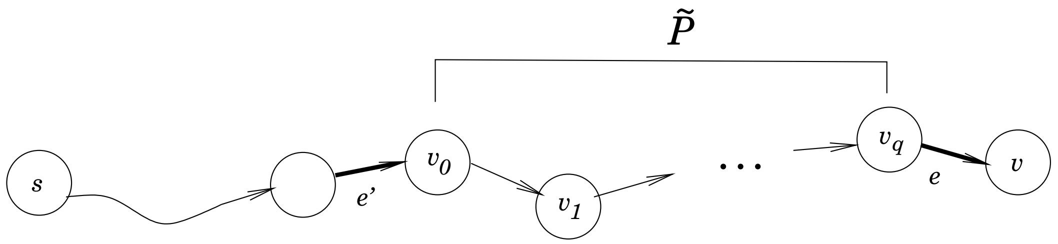

Figure 1: Path P and its sub-path $\tilde{P}$. Negative edges are drawn thick. Edge e may be not negative. The part of P before $\tilde{P}$ could be empty ($s = v_0$), and then edge (v_0, v_1) could be negative.

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, and e be its last edge. 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 part of P between e', if exists, and e, excluding; we set $\tilde{P}$ be P without e otherwise. Let $\tilde{P} = (v_0, v_1, \ldots, v_q)$. Observe that in any case, $neg(v_0) = k - 1$. Hence, $d(v_0) = opt(v_0)$ after k rounds, by the induction assumption. For illustration, see Figure 1.

Consider the first round, k', where v_0 is scanned with $d(v_0) = opt(v_0)$. We will prove that $d(v) = opt(v)$ at the end of that round. Since k' is at most $k + 1$, this will suffice.

At first, we prove for the case, when all edges of $\tilde{P}$, except maybe its first edge if $v_0 = s$, have a *positive* cost. Let us prove by induction the auxiliary statement: *At the k'th round, the vertices $v_i \in \tilde{P}$ are scanned 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 on $\tilde{P}$. 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 $\tilde{P}$ have a positive cost. Since any prefix of P is an optimal path to its end-vertex, function *opt strictly grows* along $\tilde{P}$. 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'th round, $d(v_0) = opt(v_0)$ always is the *unique* minimal value of d on $\tilde{P}$. Hence by the Dijkstra rule, v_0 enters S and is scanned first among the vertices on $\tilde{P}$.

At the induction step, we assume that v_i, $i < q$, is scanned with $d(v_i) = \mathit{opt}(v_i)$ before v_{i+1}, and prove that v_{i+1} is scanned next on $\tilde{P}$ with $d(v_{i+1}) = \mathit{opt}(v_{i+1})$. When v_i is scanned, $d(v_{i+1})$ gets its final value $\mathit{opt}(v_{i+1})$ 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'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 and is scanned first among the vertices in $\{v_{i+1}, \ldots, v_q\}$. This is the end of the auxiliary statement proof.

By the above statement, v_q is scanned at the k'th round with $d(v_q) = \mathit{opt}(v_q)$. In particular, after the relaxation on edge (v_q, v), holds $d(v) \leq d(v_q) + c(v_q, v) = \mathit{opt}(v_q) + c(v_q, v) = \mathit{opt}(v)$. Hence, $d(v) = \mathit{opt}(v)$ after the k'th round, as required in the induction step.

Now, let edge costs on $\tilde{P}$ be general. This case differs in that there may be sub-paths of $\tilde{P}$ consisting of zero cost edges only. Notice that *opt* is a constant at each such sub-path, 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 be scanned in *any* order between them, with $d(v_i) = \mathit{opt}(v_i)$ at the moment of their scanning. Indeed, suppose that vertex v_j enters S and is scanned, while the first vertex on $\tilde{P}$ not in S is v_{i+1}, $i + 1 < j$, and v_i is in S with $d(v_i) = \mathit{opt}(v_i)$. By the analysis as above, we have $d(v_{i+1}) = \mathit{opt}(v_{i+1})$ at that moment. By the Dijkstra rule, $d(v_j) \leq d(v_{i+1}) = \mathit{opt}(v_{i+1})$, and thus $d(v_j) \leq \mathit{opt}(v_{i+1}) \leq \mathit{opt}(v_j) \Rightarrow d(v_j) = \mathit{opt}(v_{i+1}) = \mathit{opt}(v_j)$. 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 $\mathit{opt}(v)$ is as follows: We consider now the first round, k', where *some vertex* v_m on $\tilde{P}$ is scanned with $d(v_m) = \mathit{opt}(v_m)$. By the induction assumption, v_0 has this property after k rounds, hence $k' \leq k + 1$. The base of the induction in the proof of the auxiliary statement is now 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 $\mathit{opt}(v_i)$. By the choice of v_m, this means that their d values strictly decrease. Hence, by the scanning rule of BFD-p, all of them are really scanned at that round. $\square$

Theorem 2.3 *If there is no negative cycle reachable from s in G, BFD correctly computes the cheapest path value 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, d values equal *opt* values at all vertices after $neg(G,s)+1$ rounds of BFD. 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) reachable from s 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. $\square$

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, its 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.

Similarly to BF, BFD detects the improper input—a graph G with a negative cycle reachable from s—in $|V|-1$ rounds.[4] Let us reduce this bound for some specific cases. The following statement is straightforward.

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

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

For example, the total number of negative edges in G is such a bound.

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. 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 $< i = \#neg(G) + 2 >$.*

Remark: The arc-fixing algorithm suggested in [3, 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 the strongly connected components (SCCs) of G (e.g., by the 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 $< i = B^{SCC}(G, s) + 2 >$.*

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, there are implementations passing on an integer priority queue in the key increasing order (e.g. [17]). They run in time $O(|E| + \ldots)$, with a low constant factor. 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 the considered implementations rely on the known property of Dijkstra to insert vertices to S in a non-decreasing order of their d values, they 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. $\square$

In addition, there exist implementations of *Dijkstra-like* algorithms working with special time bounds for specific graph classes. For example, algorithm [7] works when edge costs are *positive* real numbers. It generalizes Dijkstra by *relaxing its 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: $O(|E| + \frac{opt_{max}}{c_{min}})$, where opt_{max} denotes the maximal cost of a cheapest path from s.

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, $d(v)$ equals $opt(v)$ after $non\_pos(v) + 1$ BFD rounds.*

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. $\square$

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 *Consider the BFD-r$^+$ version of BFD, where the Dijkstra-like algorithm and its implementation [7] are used in $Dijkstra\_scan$. If there is no negative cycle reachable from s in G, it correctly computes $opt(v)$ for all $v \in V$ and the cheapest path tree, in at most $non\_pos(G, s) + 2$ rounds. Otherwise, it reports on the existence of such a cycle.*

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$^+$.

Remark: When the work [7] became known at the West in 90th, the Dijkstra choice rule relaxation of [7] was substantially extended in fast Dijkstra-like algorithms [16] and [14]. However, they work for undirected graphs only. Since any negative edge taken in both directions forms a negative cycle, these algorithms may not be used in BFD.

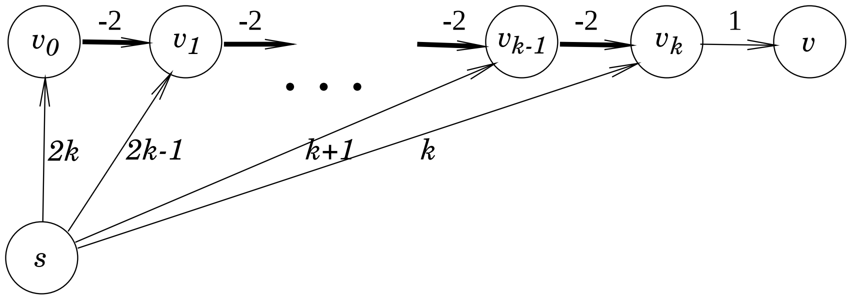

Figure 2: Tightness example.

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 (here and 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. After $(k+1)$ rounds, all d values on P 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., [4, Chapter 26.3]; also the entire approach of [3] 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 from s 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

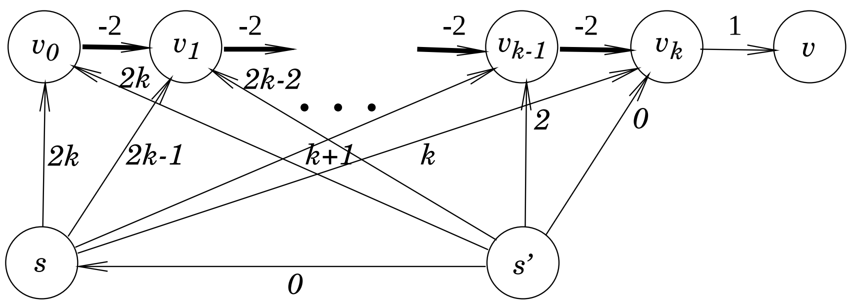

Figure 3: Example of the power of Johnson reweighting for BFD.

this example is as hard as the above one. 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_scan* /Dijkstra executions would suffice in total, instead of $k + 2$ executions. This shows how much help can be obtained sometimes by choosing an appropriate source for an auxiliary BFD run, with subsequent reweighting.

A motivation for existence of such problem instances may continue the GPS-related one 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 containing many negative edges from the driver's target or from some far away vertex approximately equidistant from the source and the target. Then, the choice of such a vertex as an auxiliary source as above will help.

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

3 Discussion: Iterated Greedy Algorithms

After acquainting Bellman-Ford-Dijkstra algorithm, Allan Borodin paid attention that BFD *iterates* the *greedy* algorithm Dijkstra, in order to extend its applicability. Generalizing the situation, he asked what can we *prove* for a general iterated greedy algorithm.[5] For greedy algorithms, some general lower bounds were proved in [1, 5]. For iterated greedy algorithms, he conjectured that even the analysis of the two iterations case would be difficult.

Let us try defining the *Iterated Greedy* algorithm class, aiming in just a classification tool, in the meanwhile. An iterated greedy algorithm consists of consecutive phases, so that:

- *an initial solution is given;*
- *all phases run the* same *greedy algorithm, which improves its initial solution to its output one;*
- *the phases are fully idependent, except that the output solution of each phase is passed to the next phase as its initial one.*

An analysis of such an algorithm may be based on its following property, if holds: *Each phase advances by increasing by at least one the value of some parameter of the current solution, while the maximal value of that parameter is bounded by some B.* As a result, the number of phases is bounded by B.

By its essence, Bellman-Ford-Dijkstra is an iterated greedy algorithm. By Proposition 2.2, each its *Dijkstra_scan* phase increases by one the bound for $neg(v)$ that guarantees the provable optimality of the current $d(v)$ value. Also Bellman-Ford may be considered as an iterated greedy algorithm, with a degenerate greedy sub-algorithm *Plain_scan*. Similarly to BFD, each its phase increases by one the bound for the minimal length of an optimal path from s to v that guarantees the provable optimality of the $d(v)$ value (the standard correctness proof of BF is based on this property). Also Dinitz (Dinic) network flow algorithm [6] may be seen as an iterated greedy algorithm. Each its phase builds the layered sub-network L of the residual network, finds a blocking flow in L using a *greedy* flow algorithm, and updates its initial flow by that blocking flow. As a result, the distance from the flow source to the flow sink in the residual network increases by at least one.

Seemingly, the above list may be extended by other interesting algorithms.

[5]Personal communication.

4 APPPENDIX: Shortest path tree proof

Consider any (properly initialized) relaxation based algorithm $\mathcal{A}$ for the single-source cheapest paths finding (in particular, Bellman-Ford). Let us prove the following simple property:

Lemma 4.1 *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 $P||(u, v)$ costs less than $d(u) + c(u, v) = opt(v)$, a contradiction. □

At any moment of an execution of $\mathcal{A}$, denote by $\bar{V}$ the (growing) vertex set $\{v \in V : d(v) = opt(v) < \infty\}$.

Proposition 4.2 *At any moment, 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 that 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 4.1, 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 Proposition implies straightforwardly that $\mathcal{A}$ builds the cheapest paths tree from s to all vertices that it eventually provides the optimal value of d to.

Acknowledgment

The authors thank Boris Cherkassky and Andrew Goldberg for their help with relating the paper to contemporary practical methods, and for their comments. They are also grateful to Seth Pettie for his useful remarks.

References

[1] A. Borodin, M.N. Nielsen, and C. Rackoff. (Incremental) Priority Algorithms. *Algorithmica* **37** (2003), no. 4, 295–326.

[2] 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).

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

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

[5] S. Davis and R. Impagliazzo. Models of Greedy Algorithms for Graph Problems. *Algorithmica* **54** (2009), no. 3, 269–317.

[6] 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).

[7] 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).

[8] 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.

[9] 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).

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

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

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

[13] D.G. Malcolm, J.H. Roseboom, C.E. Clark, W. Fazar. Application of a Technique for Research and Development Program Evaluation. *Operations Research* **7** (1959), no.5, 646-669.

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

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

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

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