Lower Bounds for Relaxation-Based Shortest Path Algorithms

By

Adrian Gushin
Bachelor of Science

Computer Science

Lower Bounds for Relaxation-Based Shortest Path Algorithms

By

Adrian Gushin

Advisor: Michelangelo Grigni

An abstract of
A thesis submitted to the Faculty of the
Emory College of Arts and Sciences of Emory University
in partial fulfillment of the requirements for the degree of
Bachelor of Science
in Computer Science
2024

Abstract

Lower Bounds for Relaxation-Based Shortest Path Algorithms
By Adrian Gushin

Computing the shortest path in a directed, weighted graph is a classical algorithms problem with applications to transportation, social networking, and many other fields. Although the recent years have seen the development of fast shortest path procedures, the traditional Bellman Ford algorithm and others like it remain the only shortest path algorithms that can operate on a general graph and therefore still see use in fields like packet routing.

This paper examines existing lower bounds for the performance of Bellman Ford-like shortest path algorithms. It moves beyond the non-adaptive setting that has received frequent attention to examine adaptive algorithms, which are attentive to information beyond the mere topology of the input graph. Specifically, I will expand some existing results about non-adaptive algorithms to a more general set of non-adaptive and weakly adaptive approaches. Additionally, I will produce new lower bounds for several Bellman Ford-like adaptive algorithms that show the inclusion of adaptive heuristics does not improve the minimum number of relaxation operations beyond $\Omega(n^3)$ on a weighted graph with n vertices.

Lower Bounds for Relaxation-Based Shortest Path Algorithms

By

Adrian Gushin

Advisor: Michelangelo Grigni

A thesis submitted to the Faculty of the
Emory College of Arts and Sciences of Emory University
in partial fulfillment of the requirements for the degree of
Bachelor of Science
in Computer Science
2024

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Computing the shortest path between two points in a network is a common graph theory problem with a litany of uses. An algorithm that quickly performs this task has applications in social networking, transportation, and packet routing, among other areas [13, 15, 19]. One of the first attempts to solve this problem is the Bellman-Ford algorithm, which accepts a graph and a start node as input and outputs a set of optimal paths to every other vertex in the graph. This algorithm works on any graph structure that does not contain negative cycles. However, it is quite slow, and on a graph with n vertices and m edges, it requires $\Omega(mn)$ operations in the worst case.

Two approaches have been taken to improve upon the Bellman-Ford algorithm. The first is the development of more efficient path-finding algorithms. Dijkstra's algorithm, one of the most commonly used improvements to Bellman-Ford, reduces the runtime to $\mathrm{O}((m + n) \cdot \log n)$. If a Fibonacci heap is used, Dijkstra's runtime decreases further to $\mathrm{O}(m+n \log n)$. Very recently, a new algorithm has been developed that can find shortest paths in near-linear time [2].

However, it is important to note that the development of these new algorithms has not rendered more classical approaches like Bellman-Ford obsolete. For example, Dijkstra's algorithm cannot handle negative edge weights and the near-linear time algorithm only accepts small integer weights. Because Bellman-Ford is more general than these newer approaches, it still has uses today in negative cycle detection and managing internet traffic [5, 11]. This commentary motivates the second approach to improving Bellman-Ford, which is to implement heuristics that allow the path-finding algorithm to terminate early, skip unnecessary steps, or otherwise speed up its performance without compromising its generalizability. When an algorithm is improved by these heuristics, it is called *adaptive*.

Much attention has been given to quantifying the performance of Bellman-Ford and other similar path-finding algorithms. In fact, recent papers have been able to get within a constant factor of the number of operations these algorithms require [3]. However, very little work has been done into assessing the performance of the adaptive variants of these algorithms. This paper will contribute to the current body of work by exploring the runtime of Bellman-Ford-like algorithms when they are improved by commonly discussed adaptive heuristics.

Chapter 2

Definitions

Consider a weighted directed graph G with an edge set E, vertex set V, and designated start vertex s, where $s \in V$. Define a cost function $c : E \to \mathbb{R}$ that maps an edge to its associated weight in the digraph. A path π of length l is a sequence of l edges that join $l + 1$ vertices. For all $e_i = (u_i, v_i) \in \pi$ where $1 \leq i < l$, $v_i = u_{i+1}$. This property requires that π's edges be consecutive; however, this behavior does not matter at the endpoints. π is considered a *shortest path* from s to a destination $t \in V \setminus \{s\}$ if the following criteria are met:

1. For all $1 \leq i \leq l$, $e_i \in E$.
2. Suppose $d^* = \sum_{i=1}^{l} c(e_i)$ is the total cost of traversing π. Then, d^* must be less than or equal to the cost of every other possible path from s to t.

The single-source shortest paths (SSSP) problem extends this idea to a larger graph structure. Solving SSSP requires finding the shortest path between s and every other vertex $v \in V$. Solving this problem is equivalent to computing a shortest-path tree rooted at s. A tree T with a vertex set V' is a shortest-path tree if it has the following properties:

1. V' equals the set of all vertices reachable from s.

2. If d' is the cost of the path from s to $t' \in V'$ in T, then d' must also be the cost of the shortest path from s to t' in G.

There are many modern algorithms that compute the shortest path efficiently, including the near-linear time approach that operates on all integer edge weights discussed in the introduction. However, this approach is not directly comparable to Bellman-Ford and other classical algorithms as it introduces an extra $O(\log W)$ factor, where "$W \geq 2$ is the minimum integer such that $[c(e)] \geq -W$ for all $e \in E$" [2].

A popular algorithmic technique, used by both Bellman-Ford and Dijkstra's algorithm, is *relaxation*. Algorithms based on this principle store a set of tentative distance labels from s to all other $v \in V$, denoted as $d(v)$. The initialization procedure for these distances takes G as input and is shown below.

Algorithm 1 Initialize(G) Procedure

```
for $v \in G.V$ do
    if $v == s$ then
        $d(v) \leftarrow 0$
    else
        $d(v) \leftarrow \infty$
    end if
end for
```

A popular approach to solving SSSP is the use of relaxation-based algorithms. A relax is an operation that accepts as input an edge $e = (u, v)$ and attempts to update $d(v)$ to a smaller value. The specifications for relaxation are shown in Algorithm 2.

Algorithm 2 Relax(u, v) Procedure

```
$d^* \leftarrow d(u) + c(u, v)$
if $d^* < d(v)$ then
    $d(v) \leftarrow d^*$
end if
```

Implementations of relaxation-based algorithms vary across adaptive and non-adaptive, sometimes called oblivious, approaches [3, 8, 12]. A non-adaptive implementation uses a predetermined relaxation sequence computed exclusively based upon information found in the topology of the graph, and it will not change or skip parts of its sequence mid-execution. The algorithm is therefore oblivious to information about the edge weights or whether or not the path it is taking is "smart." Algorithm 3 shows a typical implementation of non-adaptive Bellman-Ford.

Algorithm 3 Non-Adaptive Bellman-Ford

```
initialize $G$
for $i = 1$ to $|V| - 1$ do
    for $e \in E$ do
        relax($e$)
    end for
end for
```

In contrast, adaptive approaches can be attentive to information beyond the topology of the input graph. These modifications can allow the algorithm to skip unnecessary relaxations or alter its approach to minimize its computational workload. A simple and commonly used adaptive modification to Bellman-Ford is to terminate the algorithm after no distance label receives an update during a round. Dijkstra's algorithm uses a more complicated adaptive approach that uses a priority queue to greedily choose the next edge to relax. Because of the wide number of modifications that can be made to an algorithm, it is prudent to define classes of adaptivity that describe the level of awareness a procedure has to non-topological features. These definitions are summarized in Table 1. The table is organized in ascending order of awareness.

Class	**Characteristics**
Non-Adaptive	Algorithm is only aware of G's topological connectivity.
Weakly Adaptive	Algorithm knows when $d(v)$ is updated for all v.
Moderately Adaptive	Algorithm can compare $d(v)$ to $d(v')$ for all pairs of vertices.
Strongly Adaptive	Algorithm is aware of the value of $d(v)$ for all v.

Table 2.1: Descriptions of varying levels of adaptivity.

Adaptive algorithms use *edge eligibility* to leverage their extra information and make performance gains. An edge $e = (u, v)$ is *eligible* if $d(u)$, the distance label of its parent vertex, has been updated since the last time e was relaxed. This rule may also be referred to as the *parent-checking heuristic* [12]. The parent-checking heuristic can be implemented by modifying the relax function to take timestamps into account. Suppose $t : V \cup E \to \mathbb{R}$ is a function that accepts either a vertex or an edge as input and maps it to the time at which it was most recently updated. For a vertex v, an update occurs when its $d(v)$ is reduced. For an edge e, an update occurs when **relax**(e) is called, regardless of whether any distance labels are changed. Therefore, if $e = (u, v)$, then the comparison $t(v) > t(u)$ checks if u has received a distance label improvement since the last time e was relaxed, the core operation behind the parent-checking heuristic. Algorithm 4 uses this function to implement LazyRelax, an altered version of the relax procedure that utilizes this heuristic.

Algorithm 4 LazyRelax(e) Procedure

```
$e = (u, v)$
**if** $t(v) > t(u)$ **then**
    **relax**($e$)
**end if**
```

The precise interaction of an algorithm with its eligible edges varies based on implementation. An algorithm that uses a fixed relaxation sequence may simply skip over all non-eligible edges in its ordering. Dijkstra's algorithm, a more complicated approach, makes a greedy choice by always choosing to relax edges whose parent has the lowest known $d(v)$ amongst the vertices on its visited queue. To generalize, adaptive algorithms prefer to relax eligible edges over non-eligible ones, as doing so prevents wasted operations and speeds up the algorithm.

Much attention has already been given to the performance of non-adaptive Bellman-Ford algorithms [3, 7, 8]. This paper will emphasize weakly adaptive relaxation-based algorithms that use the parent-checking heuristic. I will show two results. Firstly, there exist classes of adaptive relaxation-based algorithms that use the parent-checking heuristic that cannot outperform, in the general case, the non-adaptive lower bound of $\Omega(mn)$ relaxations on an input graph with m edges and n vertices. Additionally, I show that introducing an element of stochastic choice to these algorithms reduces the average-case number of relaxations needed to $\Omega(ml \log c)$, where l and c are parameters to a class of graph that will be discussed later.

Chapter 3

Related Work

There are three areas of related works. The first is the supersequence problem: given a set, create a sequence that contains, as a subsequence, every possible permutation of the elements in the set. The second category of relevant problems concerns the calculation of a sequence of operations that minimize the number of relaxes necessary from relaxation-based SSSP algorithms like Bellman-Ford. In doing so, these procedures attempt to tighten the upper bounds on the number of relaxations required to converge to the shortest path solution. The final class of relevant literature presents already existing bounds on relaxation-based SSSP algorithms.

3.1 The Supersequence Problem

The shortest path problem can be solved by computing a supersequence of $G's$ vertices. Because the shortest path is guaranteed to be a subsequence in this ordering, a naive procedure that simply iterates through the supersequence and, for every entry v in the sequence, relaxes all of v's edges. The efficiency of this approach therefore depends greatly on the length of the supersequence. Historical results about this length include [6, 9, 14]. The most recent solution to the problem still cannot reduce the length of the supersequence from being quadratic in the number of set elements [16]. As a result, this naive algorithm cannot perform better than Bellman-Ford.

3.2 Existing Upper Bounds

3.2.1 Yen's Algorithm

Yen's algorithm, an old approach, remains the tightest upper bound on this problem in the deterministic case [18]. Suppose that a graph G is given as input, with a vertex set V, edge set E, and a start node s. The algorithm initializes by arbitrarily labelling every vertex $v \in V$ with a distinct, positive integer, starting with s. Yen forms two sub-graphs from E. G^+ contains every edge (u, v) where the arbitrary label assigned to u is lower than that assigned to v. Its complement, G^-, contains the rest of the edges, which are those where u has a greater label than v. These sub-graphs are DAGs and Yen's algorithm processes them in a topological fashion. For G^+, this ordering sorts the vertices by ascending label order. For G^-, the order is descending instead. After initializing the two DAGs, the algorithm executes a "round of relaxation," which is one iteration of the outer while loop shown in Algorithm 5, until the stop condition is reached. This condition is usually either a number of iterations or a point where no vertex receives a distance label update. The code in Algorithm 5 shows an example that halts when no updates have occurred.

Each round of relaxation has two main events. First, for each vertex $u \in G$ in the topological order of G^+, relax all the edges e outgoing from u such that $e \in E_{G^+}$. Then, repeat this step for all the edges in E_{G^-}. The following pseudocode shows [1]'s implementation of the procedure. Their version has some adaptive features, including the parent-checking heuristic and an early termination if no vertices have their distance labels updated.

For any input graph, there will exist some number of iterations t such that this procedure finds the shortest path. To gain an intuition for this result, note that if

Algorithm 5 Yen's Algorithm

```
number the vertices arbitrarily, starting with $s$
while LazyRelax succeeds on at least 1 vertex do
    for each vertex $u$ in ascending order do
        for $e^+ \in G^+$ do
            LazyRelax($e^+$)
        end for
    end for
    for each vertex $u$ in descending order do
        for $e^- \in G^-$ do
            LazyRelax($e^-$)
        end for
    end for
end while
```

multiple sequential edges are in the same DAG and their source has a correct distance label, the topological sorting of the DAG ensures that only one relaxation round is needed to properly update all of these sequential edges. If the edges alternate between the two DAGs, then one round will properly update two edges since both DAGs are relaxed. It follows that the absolute worst case for Yen's procedure is one in which the edges on a shortest path alternate between G^+ and G^-. Since every round of relaxation will properly update two edges, the algorithm will need $|V|/2$ rounds of relaxation, or $|E| \cdot |V|/2$ relaxation operations, to correctly solve the problem in the worst case. It is important to observe that although the implementation of Yen's algorithm shown in 5 and the subsequent modifications done in 6 incorporate some adaptive elements, these components do not ultimately improve the performance of the algorithm in the worst case, when one edge from each DAG is discovered at a time.

3.2.2 Randomized Approaches

Bannister and Eppstein improve Yen's upper bound, with high probability, by introducing randomization into the algorithm [1]. Their procedure initializes the vertex ordering as a random permutation of all possible sequences with the start node s first, in order to maintain the usefulness of the topological ordering described earlier. Their pseudocode is shown below in Algorithm 6. The core insight: given a triple of sequen-

Algorithm 6 Bannister and Eppstein's Algorithm

```
number the vertices randomly such that all permutations starting with $s$
are equally likely
while LazyRelax succeeds on at least 1 vertex do
    for each vertex $u$ in ascending order do
        for $e^+ \in G^+$ do
            LazyRelax($e^+$)
        end for
    end for
    for each vertex $u$ in descending order do
        for $e^- \in G^-$ do
            LazyRelax($e^-$)
        end for
    end for
end while
```

tial connected vertices, $\{a, b, c\}$, that are being labeled by the random permutation, there are $3! = 6$ possible labelings. If l_i denotes the label of vertex i, then of those 6 labelings, 4 of them are "good" for the algorithm, meaning they reduce the number of rounds of relaxations needed to achieve correctness. Specifically, any configuration in which l_b is not the minimum label is a good result for the algorithm. If l_a is the minimum, then edge (a, b) must be in G^+. If edge (b, c) is in G^+ as well, then the topological sort of the DAG ensures that (a, b) is relaxed before (b, c), maintaining the order. However, if it is in G^-, (b, c) will still be relaxed second because, as used in [1], one relaxation round goes through all the edges in G^+ followed by all the edges in G^-. Note that reversing the order of relaxations does not interfere with the result; instead, the "good" outcomes become all those in which l_b is not the maximum. To complete the argument, note that if l_c is the minimum, then (b, c) must be in G^-. If (a, b) is in G^-, it will be relaxed first because of the topological ordering; if not, it will still be relaxed first because edges in G^+ are relaxed first. The two cases where l_b is the minimum label correspond to behavior similar to that of the original

Yen's algorithm. This situation corresponds to the case in which $(a, b) \in G^-$ and $(b, c) \in G^+$. (a, b) needs to be relaxed before (b, c) but since G^+ is relaxed prior to G^-, the algorithm needs two rounds of relaxation to properly update both edges, which is the same performance as the worst case of Yen's algorithm. Note that because each triple minimum requires a round of relaxation to correctly set its distance label, the total number of relaxation rounds can be upper-bounded by the expected number of triple minima whose distance labels need to be corrected. Since the probability of a triple containing a minimum is 1/3 and there are $n - 2$ candidate minima–Bannister and Eppstein assume the shortest path tree is a line, so the start and end cannot be minima–then there are $(n - 2)/3$ expected minima. But, the start node already has its label set correctly, so there are in total $(n - 3)/3$ expected rounds of relaxation needed and therefore at most $mn/3$ total relaxations needed.

3.2.3 Related Problems

Another notable discovery, although less directly pertinent to the content of this paper, is that of the minimum violation permutation. Introduced by [10], the problem attempts to approximate the best possible vertex ordering for several inputted graphs. The authors produce a linear program, the solution to which approximates an ordering that minimizes the number of relaxations needed to correctly label every edge in all the inputted graphs. A restriction of this approach is its generalizability; [10]'s procedure requires that all input graphs be highly similar–such as several graphs with the same topology but different edge weight sets.

3.3 Existing Lower Bounds

3.3.1 Non-Adaptive Deterministic Algorithms

Most relevantly, Eppstein [3] shows that on complete graphs, the class of non-adaptive relaxation-based algorithms require at least $(\frac{1}{6} - o(1))n^3$ relaxations. Since these algorithms are all non-adaptive, they have some fixed and pre-determined relaxation sequence σ, where each entry in σ is an edge to be relaxed. For simplicity, Eppstein labels each path edge with weight 0 and all other edges with weight 1. The goal is to greedily select the position of the path edges in σ to maximize the number of wasted relax operations before a path edge is discovered; in other words, to maximize the distance between two consecutive path edges in σ. For simplicity, Eppstein only explicitly places the even-numbered path edges. This choice is equivalent to choosing every edge, as since the graph is complete, choosing an edge in position i and in position $i+2$ leaves only 1 choice to maintain the path in position $i+1$, provided that i is even. Therefore, if s_i is the position of the ith edge in the path in σ, Eppstein defines the following telescopic sum:

$$S = (s_2 - s_0) + (s_4 - s_2) + (s_6 - s_4) + \cdots$$

This sum represents the total distance between all the even-numbered path edges in σ. Therefore, $|\sigma| \geq S$, and by maximizing S, the algorithm will be forced to compute many relax operations before finding the path. To do so, note that for any edge i, the $i-1$ edges that were selected prior have already added $i-1$ vertices to the path (not i vertices, since the start node s is always trivially included in the path). If the graph has n total vertices, this statement implies the existence of $n-i+1$ remaining vertices that still need to be added to the path. Between steps s_{i-2} and s_i, the algorithm must first relax at least all the $n-i+1$ edges originating at vertex $i-2$ connecting to the $n-i+1$ remaining vertices. To connect vertex i to the path, all of the pairs between the remaining unmatched vertices must be explored. Therefore, an additional $2 \cdot \binom{n-i+1}{2}$ relaxations must be performed. The factor of 2 is included because an edge that connects vertex A and then vertex B is distinct from a path that connects vertex B and then vertex A. Note that the path edge must be the last edge to be relaxed amongst all of these edges, otherwise the greedy principle for assigning the path edges would be violated. Therefore, Eppstein bounds one term in the telescopic sum S as

$$s_i - s_{i-2} \geq (n-i+1) + 2 \cdot \binom{n-i+1}{2} = (n-i+1)^2$$

By induction, it follows that

$$\sum_{i=2,4,6...} S_i \geq \sum_{i=2,4,6...} (n-i+1)^2 = \frac{n^3-n}{6}$$

This statement completes the proof.

It is important to note that this proof requires assuming that the relaxation algorithm is completely non-adaptive. Consider instead a weakly adaptive algorithm that skips relaxations that violate the parent-checking heuristic. In order to keep σ fixed, an $e = (u, v)$ violating the parent-checking heuristic in position $\sigma[i]$ does not affect σ, but rather causes the algorithm to skip the relaxation of e and proceed to step $\sigma[i+1]$ in the sequence. Suppose G is complete and initialized using the steps in Algorithm 1. Path edges have weight 0; all other edges have weight 1. Eppstein tries to maximize

$$(s_2 - s_0) + (s_4 - s_2) + ...$$

where s_i is the position of the ith path edge in σ. Note that because G is complete and the path must start at s, it follows that only selecting even edges in the path creates a unique set of odd edges in the path as well. Therefore, maximizing the distance between the even path edges in σ maximizes the number of relaxation operations required. s_i and s_{i-2} are chosen greedily, such that the number of relaxes needed to properly label them in σ is maximized. This rule means that once the first path edge (s,u) has been discovered, the distance label to all vertices in $V \setminus \{s,u\}$ is now 1 because an edge from s to that vertex must have been relaxed. If this is not the case, then the greedy rule is violated as another non-path edge could have been relaxed before the discovery of a path edge. Since all of these vertices received a distance label update, none of them violate the parent-checking heuristic and their edges can be relaxed as the algorithm looks for the second path edge. However, since all the vertices not currently in the path have a distance label of 1, the only way they can receive an update is via the discovery of their path edge. By the time q, the second vertex on the path, is discovered, q will be the only eligible vertex. Therefore, only q's $n-1$ outgoing edges will be relaxed, at which point the process repeats itself for the next path vertex until the shortest path has been discovered. The result is a $\Omega(n^2)$ lower bound on the number of relaxes needed, as the $O(n)$ path edges after q has been discovered take $O(n)$ work to find, plus the $O(n^2)$ work that went into finding u and q. Therefore, Eppstein's original labelling scheme does not generalize to the weakly adaptive case.

3.3.2 Non-Adaptive Randomized Algorithms

Interestingly, Eppstein also shows that if σ is selected randomly from a distribution of possible relaxation sequences instead of being deterministically generated, then the lower bound remains $(\frac{1}{12} - o(1))n^3$. Eppstein approaches this proof by defining D as a random permutation of the vertices of a complete graph. He assigns a weight of 0 to all the edges that connect consecutive vertices in this ordering and a weight of 1 to every other edge, thus creating a unique shortest path. He maintains the sum S defined above, with one difference. Because the algorithm's procedure is now random, he introduces a conditional probability distribution C_i, which is defined as distribution for the remaining, unordered vertices in D conditioned on the i edges, and their associated $i+1$ vertices, that have already been added to the shortest path. There are therefore $n-i-1$ remaining vertices, and each permutation of these vertices has equal probability. As in the deterministic case, there are $2 \cdot \binom{n-i-1}{2}$ choices for the edge $i+2$. Therefore, $\mathrm{E}[s_{i+2} - s_i] \geq$ the average distance of these choices from s_i because the position of each edge is equally likely in the relaxation sequence. Note that in the best case, σ can minimize this average by having the next $2 \cdot \binom{n-i-1}{2}$ relaxes operate on a distinct edge. Therefore,

$$\mathrm{E}[s_{i+2} - s_i | C_i] \geq \binom{n-i-1}{2}$$

Eppstein then applies a corollary of Yao's principle [17] to the summation. The relevant definitions for the corollary are D_G, which is the set of all possible probability distributions for the assignment of weights to the edges of G, and Σ_G, the set of relaxation sequences that are guaranteed to produce a correct result if executed. If $\rho(\sigma, D)$ is a function that outputs the number of relaxations needed to correct all the distance labels for a sequence $\sigma \in \Sigma_G$ on weights drawn using distribution $D \in D_G$, then $\min_{\sigma \in \Sigma_G} \rho(\sigma, D)$ lower bounds the relaxation cost of all randomized non-adaptive relaxation algorithms. Performing the above sum and applying this corollary yields the desired result.

Kolliopoulos and Stein [8] find a more general statement of the above results in the non-adaptive case and claims that for any non-adaptive relaxation-based SSSP algorithm, there exists a graph that requires $\Omega(mn)$ relaxations to ensure a correct result.

3.3.3 Stochastic Algorithms

Meyer et al. [12] introduce an approach that handles cases where the edge weights are determined stochastically as opposed to designed adversarily. They also explore sparser graphs, distinct from the analysis on complete graphs performed by Eppstein. Figure 1 shows the gadget on which they perform their calculations.

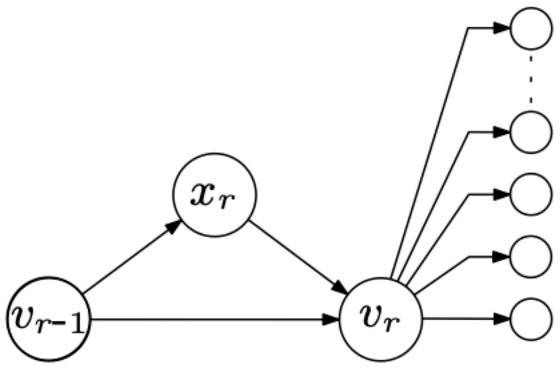

Figure 3.1: The gadget used by Meyer et al. to create their graph.

The graph in Figure 1 is composed of r of the triangular gadget between vertices v_{r-1}, x_r, and v_r. Connected to v_r is an additional component with $\Theta(r)$ vertices. This problem is stochastic because they assume all of the weight edges are randomly assigned in a uniform manner from the range [0,1]. They argue that the existence of this class of graph shows that there exist graphs with $O(n)$ nodes and random edge weights that force relaxation-based algorithms to use at least $\Theta(n^2)$ operations with high probability. They do this by defining a random variable Z_i, which is 1 if the cost of the path $v_{i-1} \to x_i \to v_i <$ the cost of the path $v_{i-1} \to v_i$, and 0 otherwise. They prove that $\mathrm{P}[Z_i = 1] = 1/6$ and argue that this implies $Z = Z_1 + \cdots + Z_r$ is binomial with $p = 1/6$. As a result of the Chernoff bound (see [4]) v_r is relaxed, with high probability, $\Theta(n)$ times. This step forces another $\Theta(n)$ relaxes in the group of $\Theta(n)$ nodes at the end of the graph, resulting in the promised $\Theta(n^2)$ bound.

Finally, [7] reduces the Bellman-Ford algorithm to two related problems, the APAS hypothesis and the Min-Plus Convolution hypothesis. The APAS hypothesis states: “there is neither a O($h^{1-\epsilon}m$) nor O($hm^{1-\epsilon}$) time algorithm for finding the length of a shortest h-hop-bound s-t path in undirected graphs with non-negative edge weights, for any $\epsilon > 0$.” The result holds for $n = \Theta(\sqrt{m}$ and $h = \Theta(m^{\eta})$ for $\eta \in (0, 1/2]$. The statement of the Min-Plus Convolution hypothesis is identical, except for that it holds for graphs with $n = \Theta(m^{\nu})$ and $h = \Theta(m^{\eta})$ with $\eta \in [1/2, 1]$. These lower bounds apply to all instances of relaxation-based algorithms, regardless of whether or not they are adaptive. Both of these hypotheses are currently unproven, and the paper assumes that they are correct in order to reduce hop-bounded Bellman-Ford. However, if the hypotheses are true, then it follows that the Bellman-Ford running time of O(hm) is optimal, where h is the number of edges permitted to be in the path.

Chapter 4

Our Results

4.1 Improving Other Results

4.1.1 Meyer et al.

Consider a queue-based relaxation algorithm similar to the one implemented by [12] operating on a graph G with V vertices and E edges. The algorithm stores vertices to be relaxed in a FIFO queue Q. While Q is non-empty, the algorithm removes the vertex at the head of Q and relaxes its associated edges. If, as a result of a relaxation operation, the distance label $d(v)$ of some vertex v is updated, then v is immediately pushed onto Q. In this way, the algorithm implements the parent-checking heuristic. However, to save time, the algorithm will not add v to Q if there is another copy of v in Q already waiting to be relaxed. The algorithm initializes by setting $d(v) = \infty$ for all $v \in V$ and then updates $d(s) \rightarrow 0$, where s is the start node. Q therefore starts with s enqueued as the only element. I will show that on a complete graph with $n = |V|$ vertices, this procedure requires $\Theta(n^3)$ relaxes to solve the SSSP problem.

Proof. Suppose $z = n - 1$ represents the number of outgoing edges associated with a vertex. Then, consider the the function $g : V \rightarrow R^z$, which maps the edges of a vertex to a vector of their weights. Let $W \sim n^2$ be a large, positive integer weight.

Additionally, if σ is the order that vertices will be added to Q in the first iteration of the algorithm, let σ_i represent the ith vertex in this ordering. Finally, the destination node t will be chosen such that it is always the first vertex the algorithm explores. In other words, $t = \sigma_1$. Using these definitions, g is implemented as follows:

1. If $v = s$, assign W to all edges.
2. Otherwise, $v = \sigma_i$, where $i > 0$ and represents v's position in the queue order.
3. Assign a weight of 0 to the one edge originating from v and connecting to s.
4. Assign a weight of $1 - i$ to all edges pointing to the vertices σ_j, where $0 < j < i$.
5. Assign a weight of W to the remaining vertices. These vertices are represented by σ_h, where $i < h < n$.

Figure 4.1 shows an example of this scheme applied to a graph of 3 vertices. Note that the shortest path is $s \rightarrow \sigma_2 \rightarrow \sigma_1$, demonstrating that σ is always in reverse-path order.

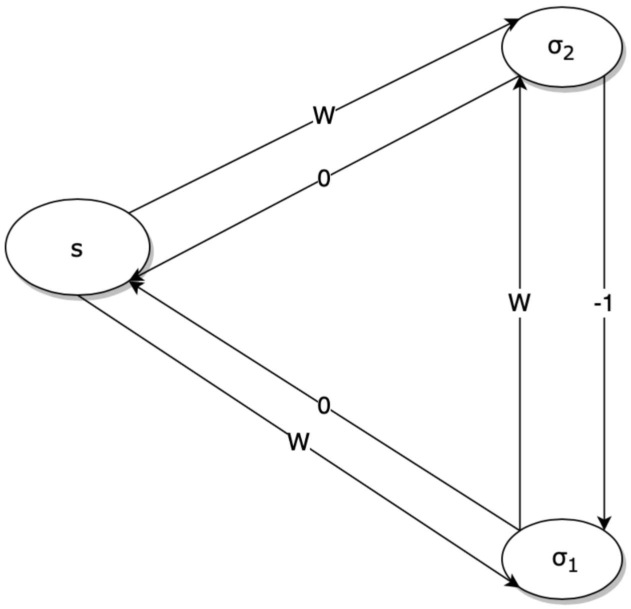

Figure 4.1: The queue labeling scheme applied to a graph of 3 vertices.

Lemma 4.1.1. *A digraph weighted using the above scheme will never produce any negative cycles.*

This scheme will create 1 vertex with $n - 1$ negative edges, 1 vertex with $n - 2$ negative edges, and so on. Vertex s will always have 0 negative edges. Suppose the vertex with only negative outgoing edges is vertex r. Then, the cycle containing the most negative edges starts from edge (s, r) and then takes every negative edge back to s. There are $n-1$ such edges, so this cycle has total cost $W-(n-2)-(n-3)\cdots-2-1$. W can always be set to some positive number such that the above sum is non-negative, preventing a negative cycle. Note that this cycle contains the most negative edges possible, so every other cycle in the graph must also have a total cost much larger than 0, proving that this scheme will never create negative cycles.

Lemma 4.1.2. *σ will continue to be in reverse path order between rounds of relaxation.*

In the first round of relaxation, s will be relaxed and every other vertex will be added to Q in the order σ. Then, after a vertex is processed, it will be removed from Q. Because of the structure of the graph, a negative edge will be relaxed to each vertex in positions 1 through $i - 1$ in σ. Since vertices will not be added to Q multiple times, the effect of this ordering is that relaxing a vertex in position i will append the vertex in position $i - 1$ to the end of Q. The exception is the final vertex in σ. This vertex will not receive any distance label updates, as no negative edges are directed towards it. It will therefore not be added back to Q. For example, after the first round of relaxation, σ will contain the vertices in positions 1 to $n - 2$ in reverse path order. This pattern will continue until the algorithm is complete.

Theorem 4.1.3. *This queue-based relaxation algorithm requires at least $\Omega(n^3)$ relaxations to correct every distance label.*

Because σ is guaranteed to be in reverse path order, each round of relaxation will only correct 1 distance label in the path (with the exception of the first round, in which 2 distance labels will be corrected). By construction, the shortest path must take the edge that connects s to vertex σ_{n-1} and then must take every negative edge that connects the vertices in reverse σ ordering. However, because the algorithm relaxes edges in σ order, it will always find the best path edge to relax last. Once it finds the path edge and its associated path vertex, every vertex that has an earlier position in σ will receive a distance label update and be forced to relax again. Ignore the first round of relaxation to make the computation simpler. After this point, the edges of s and σ_{n-1} have achieved their correct distance labels. At this point, $n-2$ vertices still need to have their edges correctly updated. Since 1 path edge is labelled correctly per round, $n-2$ rounds of relaxation are needed. Within each round, $k \cdot z$ edges need to be relaxed, where k is the number of vertices still on Q at the start of the round. Because the graph is complete, $z = \Theta(n)$. Therefore, the total number of relaxes needed can be lower-bounded by the following:

$$(n-2) \cdot \sum_{k=1}^{n-2} kn = \Omega(n^3)$$

This step completes the proof. □

Remark. The labeling scheme shown in Figure 4.1 forces poor behavior for a general relaxation-based algorithm, not just the queue-based one presented by Meyer et al. The only requirements for the lower bound to hold are that σ is fixed and that the algorithm relaxes every edge outgoing from a vertex before moving onto the next vertex in σ. If these conditions are met, then the above proof is written generally enough to show that a $\Theta(n^3)$ lower bound applies.

4.1.2 Eppstein

To adjust Eppstein's approach [3] to the weakly adaptive case, consider the following labelling scheme. Recall that for Eppstein, G is a complete directed graph.

1. Every vertex has a label ranging from n to 1. The label reflects its reversed order in the path. For instance, since s is the first vertex in the path, its label is n. The final vertex in the path would have label 1, and so on.

2. Each path edge has weight 0.

3. All other edges have weight equal to their parent vertex's label. For example, all non-path edges outgoing from s would have weight n.

Figure 4.2 below shows the result of this scheme applied to a graph where $|V| = 4$.

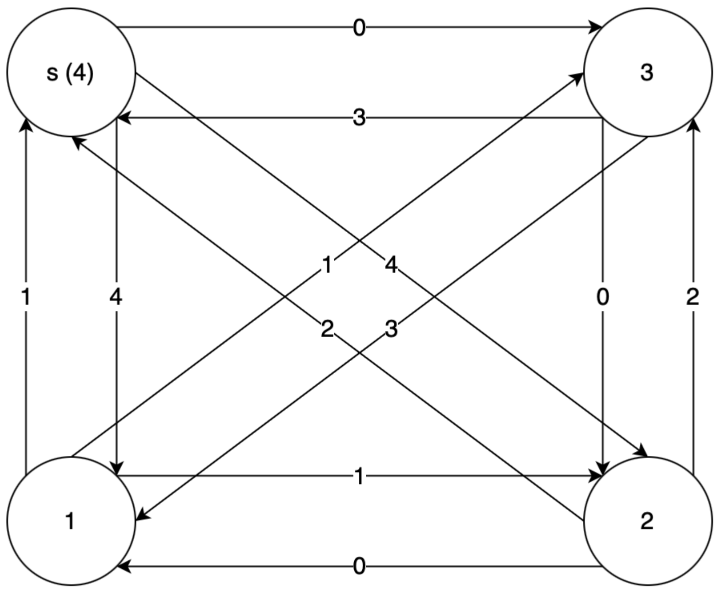

Figure 4.2: The relabeling scheme applied to a graph of 4 vertices.

Trivially, no negative cycles can be created as there are no negative weights. However, the gradual reduction in weight magnitude as vertices deeper into the path are discovered does correct the problem with Eppstein's original approach. Now, when a new path vertex q is discovered, relaxing its edges will always provide a distance label update to every other vertex later in the path and enable more relaxes. In the best case for the algorithm, G's other edges will be earlier in σ than q's edges. If this situation is the case, then the other vertices still will not have received distance label updates when their edges are being queried, and thus, all these edges will be skipped. However, this scheme still updates the labels so that relaxes are possible the next time these edges are visited. Note that Eppstein computes the number of relaxations between finding two path edges to be

$$s_i - s_{i-2} \geq (n - i + 1) + 2 \cdot \binom{n - i + 1}{2} = (n - i + 1)^2$$

This term is $\Theta(n^2)$. Therefore, although this labelling heuristic only enables every edge to be relaxed in between finding every two path edges in the worst case, requiring $\Theta(n^2) * n/2 = \Omega(n^3)$ relaxes to guarantee a correct solution is still sufficient to show that the addition of the parent-checking heuristic and weakly adaptive behavior does not improve the asymptotic performance of relaxation-based shortest path algorithms on complete graphs, further generalizing Eppstein's result.

4.2 New Results

4.2.1 Problem Statement

Suppose we are given the following gadget:

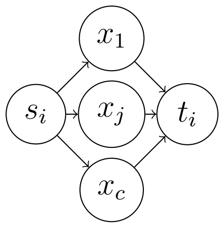

Figure 4.3: The primary gadget. The graphs in this section will be formed by repeating this structure l times.

The central column of vertices has c nodes. The node x_j represents an arbitrary vertex in the range $1 \leq j \leq c$. Each node has one incoming edge from vertex s and one outgoing edge to vertex t. Suppose we form a graph $G(V, E)$ by linking together l of these gadgets such that vertex t_{i-1} in gadget $i-1$ is vertex s_i in gadget i. Suppose that a relaxation-based shortest path algorithm is used to find the shortest path from s_1 to t_l and meets the following criteria:

1. The algorithm knows $G(V, E)$ and s_1, but may not directly observe any edge weights.

2. The algorithm does know whether or not a relaxation is successful, and will only perform a relaxation on an outgoing edge of vertex v if the distance label $d(v)$ was updated since the time of the last relaxation. If a vertex is in this state, its edges are called eligible.

3. The algorithm will relax every edge outgoing from a vertex v before moving on. For all v, the order of these relaxations will be fixed, i.e. if vertex s_i is relaxed at two separate points in the algorithm, its edges will be relaxed in the same order. The precise ordering is arbitrary.

4. If v_i is the current vertex being relaxed, then v_{i+1} will be set as the first eligible vertex encountered while relaxing the edges of v_i in accordance with the fixed sequence specified in assumption 3.

5. If no eligible vertices are reachable from the current vertex, the algorithm will return to the eligible vertex closest to s_i. This behavior actually ends up forcing the best-case scenario for this algorithm, as any other relaxes would be extra work since every distance label will be updated once the algorithm returns to s_1.

Lemma 4.2.1. *This greedy algorithm performs at least as well as Bellman-Ford on a G composed of several of the gadgets shown in Figure 4.3.*

Note that although this algorithm is not Bellman-Ford, it performs similarly to Bellman-Ford on the proposed gadget and therefore is an acceptable analogy. Observe that Bellman-Ford always finds the shortest path in $O(nm)$ time. This statement is true because Bellman-Ford relaxes every edge in the graph in one round of relaxation, which means that after every round, at least 1 edge in the shortest path must be correctly labelled. The greedy approach proposed above performs similarly on the gadget presented in Figure 1. In the context of this algorithm, a round of relaxation is defined as the time it takes for the algorithm to correctly label one gadget. Suppose the algorithm is currently in gadget i processing vertex s_i. Assuming the graph is labelled correctly behind gadget i, relaxing s_i will immediately find the edge in the shortest path, as all edges outgoing from s_i always have weight 0. To find the correct outgoing edge from column c_i, the algorithm will have to examine every vertex in c_i in the worst case. There are l gadgets in G, so at most lc rounds of relaxation are needed. Now, examine the behavior of the algorithm during one round of relaxation. In one gadget, the algorithm relaxes the c edges outgoing from s_i and then the single edge outgoing from x_j. There are at most l gadgets being examined in one round, which means at most $cl + l$ relaxes occur. Therefore, this algorithm requires $O((cl)^2)$ relaxes on G. If $n = |V|$ and $m = |E|$, then for this G, $n = cl + l + 1$ and $m = 2cl$. Applying the assumption that $c \sim l$, it follows that the algorithm requires $O(nm)$ relaxes, the same running time as Bellman-Ford.

The greedy algorithm I propose is therefore comparable to Bellman-Ford, at least for the G I have presented. I chose to use it because the structure of the algorithm makes computations easier than the original Bellman-Ford algorithm. Although the greedy approach performs like Bellman-Ford, it is important to note that neither the greedy algorithm nor Bellman-Ford are optimal for solving the SSSP problem in this G. If the algorithm knew the structure of G ahead of time, finding the shortest paths could be accomplished in linear time by computing a topological sort of the vertices because G is a DAG. However, considering this structure is still relevant in the general case where the topology of G is not directly known and therefore G is not guaranteed to be a DAG.

4.2.2 Deterministic Weakly Adaptive Algorithm

I will show that relaxation algorithms of this class must use at least $\Omega(mn)$ relaxation operations in order to guarantee a correct shortest path.

Lemma 4.2.2. *In a graph composed of l of the gadgets shown in Figure 1, the s node of each gadget will be relaxed c times for every s node that appears prior to it in the structure.*

Suppose all the outgoing edges from a node s_i have weight 1. After the algorithm has relaxed an s node, it will randomly choose an edge (s_i, x_j) to an eligible x_j to continue its relaxation. Within each gadget, the weights of the x_j nodes should be assigned adversarially such that the algorithm is forced to relax these nodes in descending order. In other words, in the gadget above, if $(x_2, s_2), (x_1, s_2), (x_3, s_2)$ is a sub-sequence of the algorithm's relaxation order, then $w(x_2, s_2) = -1$, $w(x_1, s_2) = -2$, and $w(x_3, s_2) = -3$. At the start of the algorithm, only vertex s_1 is eligible. Relaxing it causes vertices x_1 through x_c to become eligible. One of these vertices is picked at random, at the process continues forward until the end of the path is reached. The algorithm now returns to vertex s_1 to select one of its $c-1$ eligible neighbors to continue relaxation. Due to the invariant defined above, the weight of the edge connecting to s_2 will be one less than the previous edge, which causes the distance label at s_2 to be updated. Based on the topology of the graph, this cascades into the graph and causes s_2 and every vertex after it to become eligible again. Since there are c vertices in the central construct after s_1, this cascade will occur c times, forcing every s node after the first to be relaxed c times. Therefore, to generalize, each s node is relaxed c times for every other s node behind it.

Theorem 4.2.3. *There exist graphs such that relaxation-based algorithms that use* *the parent-checking heuristic must use $\Omega(mn)$ relaxes to ensure every distance label is correct.*

Thinking of the graph from the end first, the edges of s_l are relaxed $(l-1)c$ times because for each start node s_i where $i < l$, c eligible paths are explored until the algorithm moves on to the node s_{i+1} and there are $l-1$ s nodes behind s_l. Likewise, the nodes of s_{l-1} are relaxed $(l-2)c$ times, and so on. Every time an s node is fully relaxed, it means c relaxations have occurred. Note that the start node s_1 is unique since there are no s nodes before it. This node is only relaxed once, for a total of c relaxes. Therefore, if i is the number of s nodes preceding the current node and τ is the total number of relaxations used by the algorithm, then

$$\tau \geq c + c^2 \sum_{i=1}^{l-1} i = \Theta(c^2 l^2)$$

Now note that for the original graph, there are lc central nodes, l start nodes, and 1 destination node, so $n = lc + l + 1 = \Theta(lc)$. Additionally, $m = 2lc = \Theta(lc)$. By substitution, it follows that $\tau \geq \Theta(mn)$, so this algorithm will need at least $\Omega(mn)$ relaxations to solve this problem.

4.2.3 Removing Assumptions

Suppose we go back to the deterministic case and remove assumption 5. Now, when there are no eligible vertices in the neighborhood of the most recently relaxed vertex, the algorithm will randomly select any column with eligible vertices and will relax the next vertex using the sequencing requirement established in assumption 3. The requirement stipulates that if the algorithm returns to the same column several times, it will always test for eligible vertices in the same order. I will show that easing this assumption cannot improve the result found in Theorem 2.

Lemma 4.2.4. *Relaxing an s node causes every vertex deeper in the structure to* *become eligible, even when Assumption 5 is removed.*

If c_i is the column that would be selected by the approach used by assumption 5, suppose instead that the algorithm chooses some other column c_j with eligible vertices, where $j \geq i$. Since the eligible vertices of this column will always be explored in the same order, an adversary can assign edge weights such that every edge in the column causes an improvement to its associated s vertex, but the edges are discovered in the least-helpful order possible. To do this, assign a weight of -1 to the first edge explored by the vertex, -2 to the second, and so on. Therefore, the algorithm will need to go through all c edges in order to find the optimal distance label at s_{j+1}. This step ensures that there is no possible unlucky selection of c_j. Since the algorithm always explores eligible vertices within a column in the same order, exploring the columns in a random order still allows the adversary to optimize the edge weights to achieve worst-case performance. Therefore, no selection of c_j will allow the algorithm to skip edges or learn which edge is optimal. Now, note that by the topology of the graph, improving a distance label at any s vertex necessarily improves the distance labels at every vertex deeper in the graph. The consequence is that every vertex beyond the relaxed s node will become eligible again.

Corollary 4.2.5. *Assumption 5 only makes computations easier and is not required to show the $\Omega(mn)$ lower bound shown in Theorem 4.2.2.*

The intuition here is enough to show that the algorithm is either using the behavior established in assumption 5 or it is wasting time. If $j = i$, then the algorithm is relaxing eligible vertices in the deepest (closest to s_1) column, which is the behavior of assumption 5. If $j > i$, note that eventually, the algorithm will have to relax an edge in column c_i once it runs out of all other eligible vertices. Even if there is only 1 edge remaining in c_i, its weight can be assigned in an adversarial manner such that it is more negative than any other edge in that column. This step forces a distance label improvement at node s_{i+1}, which cascades into the graph and reduces every other distance label as well, resetting the problem and requiring that every edge relaxed by the algorithm when it was processing column c_j be relaxed again. Therefore, the assumption 5 behavior forces the best case scenario for this algorithm because it prevents it from wasting time relaxing edges that do not make permanent improvements to the distance labels.

4.2.4 Randomized Weakly Adaptive Algorithms

Similar to Eppstein, I will also test this graph structure on randomized algorithms. To make the computation simpler, assume all outgoing edges of any vertex s_i have weight 1. All incoming edges are randomly assigned a weight in the range $[-c, -1]$ without replacement instead of having the weights assigned adversarially as above. Suppose that instead of a fixed ordering of eligible vertex selection, the algorithm randomly and uniformly selects the next vertex from all the eligible neighbors.

Lemma 4.2.6. *When relaxing the vertices in the column of the gadget shown in Figure 2, the uniform selection scheme will relax, on average, the vertex at the midpoint of the eligible vertices in the column.*

Observe that the only time that this new property matters is when the algorithm is selecting an eligible vertex after relaxing the first previously eligible s node. That is because all distance label updates from a vertex v get propagated to every node deeper than v in the structure of the graph. The result is that the only time the new rule can cause the algorithm to skip relaxations is when it gets lucky on a s_i vertex such that all s_j where $1 < j \leq i - 1$ are not eligible. The way that the algorithm could get lucky is by skipping ahead and selecting a lower edge weight first. This would prevent the higher edge weights from triggering relaxation cascades via the adaptivity of the algorithm. Suppose s_i is the current s node and x is the eligible neighbor selected by the algorithm. If $w(u, v)$ represents the weight of edge (u, v), then $w(s_i, x)$ is a random variable with expected value $(-1/c) \sum_{j=1}^{c} j = -(c + 1)/2$ since every edge weight is equally likely.

Theorem 4.2.7. *Relaxation-based algorithms that use the uniform selection scheme require, on average, at least $\Omega(ml \log c)$ to guarantee all distance labels are correct.*

Since the expected value is the midpoint, on average half the neighbors of s_i will have an edge weight greater than the currently selected edge weight. The adaptive nature of the algorithm will then mark all of these vertices as ineligible because they are guaranteed to not be in the shortest path. From here, the process of selecting the next eligible vertex is identical, but the range of possible values of the random variable $w(s_i, x)$ is cut in half. Because the algorithm is dividing by 2 every time, there will be $\Theta(\log c)$ meaningful relaxes. Each of these relaxes triggers c relaxes from every s vertex ahead of s_i. Therefore,

$$\mathrm{E}[\tau] = c + c \log c \sum_{i=1}^{l-1} = \Theta(l^2 c \log c)$$

Using the same substitution as above, in the more stochastic case, the algorithm requires at least $\Omega(ml \log c)$ relaxations to achieve correctness on average.

Chapter 5

Conclusion

This paper contributes to the study of shortest path algorithms in two ways. Firstly, it generalizes Eppstein's [3] results to a weakly adaptive setting. Secondly, it introduces new lower bounds to other Bellman Ford-like algorithms that utilize the parent-checking heuristic. These additions to the field move beyond the study of non-adaptive algorithms.

However, there remain several open questions. Firstly, this paper only deals with the weakly adaptive case. Can these approaches be generalized to moderately or even strongly adaptive settings? Additionally, with the exception of the generalization of Eppstein's procedure, the lower bounds in this paper are computed with respect to specific algorithmic procedures as opposed to a general relaxation algorithm. Can new lower bounds be computed for a general relaxation algorithm implementing at least weakly adaptive heuristics?

Bibliography

[1] Michael J. Bannister and David Eppstein. Randomized speedup of the bellman-ford algorithm, 2011.

[2] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 600–611. IEEE, 2022.

[3] David Eppstein. Lower bounds for non-adaptive shortest path relaxation. In *Workshop on Algorithms and Data Structures*, 2023. URL `https://api.semanticscholar.org/CorpusID:258714937`.

[4] Torben Hagerup and Christine Rüb. A guided tour of chernoff bounds. *Information processing letters*, 33(6):305–308, 1990.

[5] Charles L Hedrick. Rfc1058: Routing information protocol, 1988.

[6] Daniel J Kleitman and David Joseph Kwiatkowski. A lower bound on the length of a sequence containing all permutations as subsequences. *Journal of Combinatorial Theory, Series A*, 21(2):129–136, 1976.

[7] Tomasz Kociumaka and Adam Polak. Bellman-ford is optimal for shortest hop-bounded paths. *arXiv preprint arXiv:2211.07325*, 2022. URL `https://arxiv.org/abs/2211.07325`.

[8] Stavros G Kolliopoulos and Clifford Stein. Finding real-valued single-source shortest paths in $o(n^3)$ expected time. *Journal of Algorithms*, 28(1):125–141, 1998.

[9] PJ Koutas and TC Hu. Shortest string containing all permutations. *Discrete Mathematics*, 11(2):125–132, 1975.

[10] Silvio Lattanzi, Ola Svensson, and Sergei Vassilvitskii. Speeding up bellman ford via minimum violation permutations. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[11] Stefan Lewandowski. Shortest paths and negative cycle detection in graphs with negative weights., 2010.

[12] Ulrich Meyer, Andrei Negoescu, and Volker Weichert. New bounds for old algorithms: On the average-case behavior of classic single-source shortest-paths approaches. In *International Conference on Theory and Practice of Algorithms in (Computer) Systems*, pages 217–228. Springer, 2011.

[13] Sven Peyer, Dieter Rautenbach, and Jens Vygen. A generalization of dijkstra's shortest path algorithm with applications to vlsi routing. *Journal of Discrete Algorithms*, 7(4):377–390, 2009.

[14] Sasa Radomirovic. A construction of short sequences containing all permutations of a set as subsequences. *the electronic journal of combinatorics*, 19(4):P31, 2012.

[15] AB Sadavare and RV Kulkarni. A review of application of graph theory for network. *International Journal of Computer Science and Information Technologies*, 3(6): 5296–5300, 2012.

[16] Oliver Tan. Skip letters for short supersequence of all permutations. *Discrete Mathematics*, 345(12):113070, 2022.

[17] Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 222–227. IEEE Computer Society, 1977.

[18] Jin Y Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of applied mathematics*, 27(4): 526–530, 1970.

[19] Xiaohan Zhao, Alessandra Sala, Haitao Zheng, and Ben Y Zhao. Efficient shortest paths on massive social graphs. In *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 77–86. IEEE, 2011.