Directed Single–Source Shortest–Paths in Linear Average–Case Time

Ulrich Meyer

MPI–I–2001–1-002 May 2001

Author's Address

Ulrich Meyer
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
umeyer@mpi-sb.mpg.de
www.uli-meyer.de

Acknowledgements

Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

Abstract

The quest for a linear-time single-source shortest-path (SSSP) algorithm on directed graphs with positive edge weights is an ongoing hot research topic. While Thorup recently found an $\mathcal{O}(n+m)$ time RAM algorithm for undirected graphs with n nodes, m edges and integer edge weights in $\{0,\ldots,2^w-1\}$ where w denotes the word length, the currently best time bound for directed sparse graphs on a RAM is $\mathcal{O}(n+m\cdot\log\log n)$.

In the present paper we study the average-case complexity of SSSP. We give simple label-setting and label-correcting algorithms for arbitrary directed graphs with random real edge weights uniformly distributed in $[0,1]$ and show that they need linear time $\mathcal{O}(n+m)$ with high probability. A variant of the label-correcting approach also supports parallelization. Furthermore, we propose a general method to construct graphs with random edge weights which incur large non-linear expected running times on many traditional shortest-path algorithms.

Keywords

Graph Algorithms, Combinatorial Optimization, Shortest Path, Average-Case Analysis.

1 Introduction

The *single-source shortest-path problem* (SSSP) is a fundamental and well-studied combinatorial optimization problem with many practical and theoretical applications [1]. Let $G = (V, E)$ be a directed graph, $|V| = n$, $|E| = m$, let s be a distinguished vertex of the graph, and c be a function assigning a non-negative real *weight* to each edge of G. The objective of the SSSP is to compute, for each vertex v reachable from s, the weight of a minimum-weight ("shortest") path from s to v, denoted by $\mathrm{dist}(v)$; the weight of a path is the sum of the weights of its edges.

Shortest-paths algorithms are typically classified into two groups: *label-setting* and *label-correcting*. The basic label-setting approach is Dijkstra's algorithm [16]. It maintains a partition of V into *settled*, *queued*, and *unreached* nodes, and for each node v a *tentative distance* $\mathrm{tent}(v)$; $\mathrm{tent}(v)$ is always the weight of some path from s to v and hence an upper bound on $\mathrm{dist}(v)$. For unreached nodes v, $\mathrm{tent}(v) = \infty$. Initially, s is queued, $\mathrm{tent}(s) = 0$, and all other nodes are unreached. In each iteration, the queued node v with smallest tentative distance is selected and declared settled, and all edges (v, w) are *relaxed*, i.e., $\mathrm{tent}(w)$ is set to $\min\{\mathrm{tent}(w), \mathrm{tent}(v) + c(v, w)\}$. If w was unreached, it is now queued. It is well known that $\mathrm{tent}(v) = \mathrm{dist}(v)$, when v is selected from the queue.

Label-correcting SSSP algorithms may remove nodes from the queue for which $\mathrm{tent}(v) > \mathrm{dist}(v)$ and hence have to *re-insert* those nodes until they are finally settled ($\mathrm{dist}(v) = \mathrm{tent}(v)$). In spite of poor worst-case running times, implementations of label-correcting approaches frequently outperform label-setting algorithms [9, 55].

1.1 Previous Work

Using Fibonacci heaps [23] Dijkstra's algorithm can be implemented to run in $\mathcal{O}(n \log n + m)$ time for positive edge weights. A number of faster algorithms have been developed on the more powerful RAM (random access machine) model which basically reflects what one can use in a programming language such as `C`. Nearly all of these algorithms are based on Dijkstra's algorithm and improve the priority queue data structure (see [52] for an overview).

Thorup [52] has given the first $\mathcal{O}(n + m)$ time RAM algorithm for *undirected* graphs with integer edge weights in $\{0, \ldots, 2^w - 1\}$ for word length w. His approach applies label-setting but significantly deviates from Dijkstra's algorithm in that it does not visit the nodes in order of increasing distance from s but traverses a so-called *component tree*. Hagerup [29] recently generalized Thorup's approach to directed graphs, the time complexity, however, remains super-linear $\mathcal{O}(n + m \cdot \log w)$.

The currently fastest RAM algorithm for sparse directed graphs is due to Thorup [53] and needs $\mathcal{O}(n + m \cdot \log \log n)$ time. Faster approaches for somewhat denser graphs have been proposed by Raman [47, 48]: they require $\mathcal{O}(m + n \cdot \sqrt{\log n \cdot \log \log n})$ and $\mathcal{O}(m + n \cdot (w \cdot \log n)^{1/3})$ time, respectively. Further results have been obtained for bounded integer edge weights, see [48] for a recent overview. In particular, there is constant-time integer priority queue [7] that yields a linear-time SSSP algorithm for directed graphs; however, the priority queue requires a special memory architecture with "byte overlap" which is currently not supported by existing hardware.

The classic label-correcting algorithm of Bellman–Ford [4, 21] and all of its improved derivatives (see [9] for an overview) need $\Omega(n \cdot m)$ time in the worst case. Some of these algorithms, however, show very good practical behavior for SSSP on sparse graphs [9, 55]. Nevertheless, average-case analysis of shortest paths algorithms mainly focused on the *All–Pairs Shortest Paths* (APSP) problem on the *complete* graph with random edge weights [12, 24, 32, 43, 51]. Mehlhorn and Priebe [39] proved that for the *complete* graph with random edge weights, every SSSP algorithm has to check at least $\Omega(n \log n)$ edges with high probability. Noshita [44] and Goldberg and Tarjan [27] analyzed the expected number of `decreaseKey` operations in Dijkstra's algorithm; the time bound, however, does not improve over the worst-case complexity of the algorithm. Meyer and Sanders [41] gave a label-correcting algorithm for random edge weights in $[0, 1]$ that runs in average-case time $\mathcal{O}(n + m + d \cdot \mathcal{L})$ where d denotes the maximum node degree in the graph and $\mathcal{L}$ denotes the maximum weight of a shortest path to a node reachable from s, i.e., $\mathcal{L} = \max_{v \in G, \mathrm{dist}(v) < \infty} \mathrm{dist}(v)$. However, there are graphs where $m = \mathcal{O}(n)$ but $d \cdot \mathrm{E}[\mathcal{L}] = \Omega(n^2)$, thus incurring $\Omega(n^2)$ time on average.

1.2 New Results

In Section 2 we present conceptually simple label-setting and label-correcting SSSP algorithms with adaptive bucket splitting. To the best of our knowledge these are the first algorithms which provably achieve linear average-case time on *arbitrary directed graphs* with independent random edge weights uniformly distributed in $[0, 1]$. In Sections 3 and 4 we show that the results are also obtained with high probability (whp)[1]. Section 5 provides some extensions in order to establish the same bounds for a larger class of random edge weights and improve the worst-case running times. We also sketch a parallelization for the label-correcting variant. In Section 6 we propose a general method to construct graphs with random edge weights which incur large non-linear running times on many traditional label-correcting algorithms whp. While this problem is interesting in its own right it also stresses the asymptotic superiority of our new algorithms.

Preliminary accounts of the results of the present paper have been published in [40]. Subsequently, Goldberg obtained the same average-case bound with an alternative algorithm [26] based on radix heaps. For integer edge weights it achieves improved worst-case running times.

2 Adaptive Bucket-Splitting

In this section we present our new algorithms. We first review the bucket concept for priority queues in SSSP algorithms (Section 2.1) and then show how it can be made adaptive in order to yield efficient algorithms (Section 2.2). Section 2.3 sheds light on two algorithmic details which are important to achieve the linear average-case bounds. Finally, Section 2.4 provides the worst-case bounds.

2.1 Preliminaries

Our algorithms are based on keeping nodes in *buckets*. This technique has already been used in Dial's implementation [15] of Dijkstra's algorithm for integer weights in $\{0, \ldots, C\}$: a queued

[1]For a problem of size n, we say that an event occurs *with high probability* (whp) if it occurs with probability at least $1 - \mathcal{O}(n^{-c})$ for an arbitrary but fixed constant $c \geq 1$.

node v is stored in the bucket B_i with index $i = \mathrm{tent}(v)$. In each iteration Dial's algorithm removes a node v from the first nonempty bucket and relaxes the outgoing edges of v. In the following we will also use the term *current bucket*, B_{cur}, for the first nonempty bucket. Buckets are implemented as doubly linked lists so that inserting or deleting a node, finding a bucket for a given tentative distance and skipping an empty bucket can be done in constant time. However, in the worst case, Dial's implementation has to traverse $C \cdot n$ buckets.

Alternatively, a whole interval of tentative distances can be mapped to a single bucket: v is kept in the bucket with index $\lfloor \mathrm{tent}(v)/\Delta \rfloor$. The parameter Δ is called the *bucket width*. Let λ denote the smallest edge weight in the graph. Dinitz [17] and Denardo and Fox [14] observed that taking $\Delta \leq \lambda$, Dijkstra's algorithm correctly settles *any* node from B_{cur}. Choosing $\Delta > \lambda$ either requires to repeatedly find a node with smallest tentative distance in B_{cur} or results in a label-correcting algorithm which may *re-insert* nodes into B_{cur} again in case they have been previously removed with non-final distance value: e.g., the algorithm of [41] uses buckets of width $1/\text{max\_node\_degree}$ in order to limit the number of re-insertions. However, on sparse graphs with few high-degree nodes, a large number of buckets is needed.

2.2 The Algorithms

In this section we present our new algorithms, called SP-S for the label-setting version and SP-C for the label-correcting version. For ease of exposition we assume real edge weights from the interval $[0, 1]$ (any input with non-negative edge weights meets this requirement after proper scaling). The algorithm begins with n buckets at level L_0, $B_{0,0}, \ldots, B_{0,n-1}$, each of which has width $\Delta_0 = 2^{-0} = 1$, and $s \in B_{0,0}$ with $\mathrm{tent}(s) = 0$. Bucket $B_{0,j}$, $0 \leq j < n$, represents tentative distances in the range $[j, j+1)$. A current bucket can be *split*, i.e., a new level of buckets is created, and the nodes of B_{cur} are *redistributed*. After that, the current bucket is reassigned to be the leftmost nonempty bucket in the highest level of the *bucket hierarchy* $\mathcal{B}$.

The two algorithms SP-S and SP-C only differ in the bucket splitting criterion. The label-setting version SP-S aims to have a single node in the current bucket whereas adaptive splitting in SP-C is applied to achieve a compromise between either scanning too many narrow buckets or incurring too many *re-insertions* due to broad buckets. Note the difference between *redistributing* a node (into a new bucket of higher level because of a bucket split) and *re-inserting* a node (into a bucket of the hierarchy after it has already been deleted from a current bucket before with a larger distance value; the number of re-insertions of a node equals the number of times it is removed from current buckets while having a non-final distance value). We first describe the label-correcting approach and then give the modification to obtain the label-setting variant.

SP-C works in *phases*: first, it checks the maximum node degree d^* (sum of in-degree plus out-degree) among all nodes in the current bucket $B_{\text{cur}} = B_{i,j}$ of width Δ_i at level L_i.

If $d^* \leq 1/\Delta_i$, then the algorithm removes *all* nodes from the current bucket B_{cur} and relaxes their outgoing edges. This may result in new nodes entering B_{cur} which are removed in the next phase. Furthermore, nodes previously removed from B_{cur} may re-enter it again in case their distances values have been improved by the relaxations. A new phase starts.

Otherwise, i.e., $d^* > 1/\Delta_i$, the algorithm first splits B_{cur} into $\kappa = \Delta_i/\Delta_{i+1} \geq 2$ new buckets having width $\Delta_{i+1} = 2^{-\lceil \log_2 d^* \rceil}$ each. If B_{cur} represents a distance range $[a, a + \Delta_i)$, then $B_{i+1,k}$, $0 \leq k < \kappa$, represents tentative distances in the range $[a + k \cdot \Delta_{i+1}, a + (k+1) \cdot \Delta_{i+1})$,

```
SP-C                                                              (* SP-S *)
Create level L_0 and insert s into B_{0,0}
i := 0,  j := 0,  Δ_i := 1
while i ≥ 0 do
  while nonempty bucket exists in L_i do
    repeat
      go_on := true
      j := min {k ≥ 0 | B_{i,k} ≠ ∅}
      d* := max {degree(v) | v ∈ B_{i,j}}                  (* b* := # nodes in B_{i,j} *)
      if d* > 1/Δ_i then                                   (* if b* > 1 then *)
        Create level L_{i+1} with Δ_{i+1} := 2^{-⌈log_2 d*⌉}   (* with Δ_{i+1} := Δ_i · 2^{-⌈log_2 b*⌉} *)
        U_1 := {v ∈ B_{i,j} | ∀u ∈ B_{i,j} : tent(v) ≤ tent(u)}
        U_2 := {v ∈ B_{i,j} | ∀(u,v) ∈ E : c(u,v) ≥ Δ_i}
        Remove U_1 ∪ U_2 from B_{i,j} and
        relax all outgoing edges from nodes in U_1 ∪ U_2
        Redistribute all remaining nodes from B_{i,j} to L_{i+1}
        go_on := false,  i := i + 1,  j := 0
    until go_on = true
    Remove all nodes of B_{i,j} and relax their outgoing edges
  Remove level L_i,  i := i − 1
```

Figure 1: Pseudo-code for SP-C. Modifications for SP-S are given in brackets.

see Figure 2. Those nodes from $B_{\mathrm{i,j}}$ which have surely reached their final distance values are removed, and their outgoing edges are relaxed (details follow in Section 2.3 and Figure 1). All remaining nodes from $B_{\mathrm{i,j}}$ are redistributed into the new buckets of L_{i+1} according to their tentative distances, i.e., a node v with tentative distance $\mathrm{tent}(v)$, $a \le \mathrm{tent}(v) < a + \Delta_{\mathrm{i}}$, is moved up to $B_{\mathrm{i+1,k}}$ with $k = \lfloor(\mathrm{tent}(v) - a)/\Delta_{\mathrm{i+1}}\rfloor$. The first nonempty bucket of L_{i+1} becomes the new current bucket B'_{cur}, and SP-C checks again the maximum node degree d^* in B'_{cur}. A new phase starts if $d^* \le 1/\Delta_{\mathrm{i+1}}$; otherwise, a new level L_{i+2} is created, the nodes of B'_{cur} are redistributed, and so on.

The algorithm stays with the current bucket until it is split or finally remains empty after a phase. Due to the non-negativity of the edge weights, once a current bucket in charge of distance range $[a, a + \Delta_{\mathrm{cur}})$ remains empty, it will never be filled again; at that time all nodes with distances less than $a + \Delta_{\mathrm{cur}}$ are settled. After all buckets of the highest level have been emptied, SP-C deletes this level and continues with the next nonempty bucket of the previous level. Thus, whenever the SP-C starts removing all the nodes from B_{cur}, then the following invariant holds:

Invariant 1 *For SP-C, let d denote the maximum node degree in the current bucket $B_{cur} = B_{i,j}$ at level L_i after splitting. Then the bucket width Δ_i of B_{cur} is at most $2^{-\lceil \log_2 d \rceil} \le 2^{-i}$. Furthermore, the total level width of L_i, Δ_{i-1}, is at most 2^{-i+1} for $i \ge 1$.*

The label-setting version SP-S differs from SP-C in the following aspect: before removing nodes from B_{cur}, SP-S determines the number of nodes b^* in the current bucket (instead of

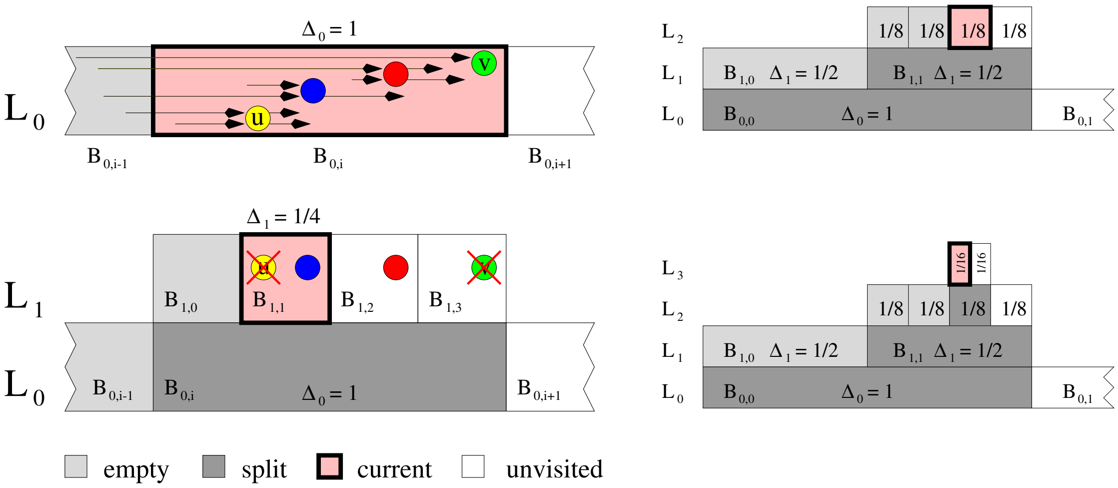

Figure 2: (Left) Basic case in the label-correcting approach: detecting a node with degree 4 in $B_{0,j}$ of L_0 forces a split into buckets of widths $2^{-\lceil \log_2 4 \rceil} = 1/4$. Nodes u and v are surely settled (u has smallest tentative distance in B_{cur}, all incoming edges of v have weight larger than B_{cur}), they are not redistributed. (Right) General situation: after a bucket split, the first nonempty bucket of the highest level becomes the new current bucket.

the maximum node degree) and splits B_{cur} into $2^{\lceil \log_2 b^* \rceil}$ new buckets in case $b^* > 1$, compare Figure 1. Hence, SP-S keeps on splitting buckets (while removing surely settled nodes and relaxing their outgoing edges) until the new current bucket only stores one node v. As all edge weights are non-negative, by then v has reached its final distance as well and can be safely removed. Since each bucket split of SP-S creates at least two new buckets for the next level, we have the following invariant:

Invariant 2 *For SP-S, buckets at level L_i have width $\Delta_i \leq 2^{-i}$. The total level width of L_i, Δ_{i-1}, is at most 2^{-i+1} for $i \geq 1$.*

2.3 Early Removals and Target Bucket Searches

During a bucket split both approaches do not redistribute nodes which have surely reached their final distance values. This helps to reduce the overhead for transferring nodes to higher and higher levels. Clearly, the node u with minimum $\text{tent}(u)$ in $B_{\text{cur}} = [a, a + \Delta_{\text{cur}})$ is settled: all edges out of nodes v with $\text{dist}(v) < a$ have already been relaxed with final distance values. Furthermore, as all edge weights are non-negative, $\text{tent}(u)$ cannot be decreased by relaxations of edges out of other nodes v in B_{cur} having $\text{tent}(v) \geq \text{tent}(u)$; compare node u in Figure 2. Additionally, a node is surely settled[2] if all of its incoming edges have weight at least Δ_{cur}, e.g., node v in Figure 2.

Lemma 1 *Let v be a node in the current bucket B_{cur} of width Δ_{cur}. The tentative distance $\text{tent}(v)$ is final if the lightest incoming edge of v has weight at least Δ_{cur}.*

[2]This observation was first used in [17] and [14]. Subsequently, it has also been applied in Thorup's algorithm [52], and a similar criterion for parallel node removal in heaps has been presented in [13]. Goldberg's recent paper [26] for SSSP in linear average-case time also relies on this observation.

Proof: Let $[a, a + \Delta_{\text{cur}})$ denote the interval of tentative distances which belong to B_{cur}. The outgoing edges of all nodes with final distances at most a were already relaxed before, and the correct distance values were used for these nodes. Hence, the value $\text{tent}(v)$ can only be reduced via a path $\langle u, \ldots, v\rangle$ with total weight less than $\text{tent}(v) - a < \Delta_{\text{cur}}$ and $u \in B_{\text{cur}}$. However, since all incoming edges of v have weight at least Δ_{cur} such a path cannot exist. □

For the relaxation of an edge $e = (u, v)$ from a node $u \in B_{\text{cur}}$ which improves $\text{tent}(v)$, it is necessary to find the appropriate target bucket which is in charge of $\text{tent}(v)$: either B_{cur} or a so far unvisited non-split bucket. The target bucket search can be done by a simple bottom-up search: for $t = \text{tent}(v)$, the search starts at level L_0 and checks whether the bucket $B^0 = [b_0, b_0 + \Delta_0) = [\lfloor t \rfloor, \lfloor t \rfloor + 1)$ has already been split or not. In the latter case, v is moved into B^0, otherwise the search continues in level L_1 and checks $B^1 = [b_1, b_1 + \Delta_1)$ where $b_1 = \lfloor (t - b_0)/\Delta_1 \rfloor$. Generally, if the target bucket $B^i = [b_i, b_i + \Delta_{\text{i}})$ at level L_i has been split then the search proceeds with bucket $B^{i+1} = [b_{i+1}, b_{i+1} + \Delta_{\text{i+1}})$ where $b_{i+1} = \lfloor (t - b_i)/\Delta_{\text{i+1}} \rfloor$. Each level can be checked in constant time. The maximum number of levels to be checked depends on the splitting criterion:

Each splitting reduces Δ_{cur} by at least a factor of two. For SP-C, buckets are only split in the presence of a node with degree bigger than $1/\Delta_{\text{cur}}$. Since the maximum node degree in G is at most $2 \cdot n$, the search for SP-C will be successful after at most $\lceil \log_2 2n \rceil$ levels. In contrast, all nodes redistributed by a split of SP-S may belong to the same new bucket, causing repeated splits. However, each split settles at least one node thus limiting the height of the bucket hierarchy to n and hence the search time for relaxations to $\mathcal{O}(n)$.

2.4 Worst-Case Bounds

In this section we consider the worst-case performance of our approaches.

Theorem 1 *Using the simple bottom-up target search routine for relaxations, the worst-case time of SP-C is $\mathcal{O}(n \cdot m \cdot \log n)$, SP-S requires $\mathcal{O}(n \cdot m)$ time.*

Proof: Initializing global arrays for tentative distances, level zero buckets, pointers to the queued nodes, and storing the lightest incoming edge for each node can clearly be done in linear time for both algorithms.

During the execution of SP-C, if a node $u \in B_{\text{cur}}$ is found that has degree $d > 1/\Delta_{\text{cur}}$, then B_{cur} is split into at most $\kappa \leq 2 \cdot (d - 1)$ new buckets. Each node node u can cause at most one splitting since u is redistributed to buckets having width at most $1/d$, and u never moves back to buckets of larger width. Hence, the costs for the following actions can be charged on the degree of the node u that caused the splitting: setting up, visiting and finally removing the at most κ buckets of the new level. For all splittings this amounts to $\mathcal{O}(m)$ operations in total. Furthermore, each phase or splitting settles at least one node. Thus, there are at most n phases or splittings, each of which transfers at most $\min\{n, m\}$ nodes during a splitting. For SP-C, each phase or splitting relaxes at most m edges; since the target buckets can be found in $\mathcal{O}(\log n)$ time, over all phases and splittings this accounts for $\mathcal{O}(n \cdot m \cdot \log n)$ operations.

The argumentation for SP-S is slightly different: again there are at most n splittings but each of them creates at most $\min\{2 \cdot (n - 1), m\}$ new buckets (because B_{cur} cannot keep more than $\min\{n, m\}$ nodes and the number of new buckets is less than twice the number of nodes

B_{cur}). This adds up to $\mathcal{O}(\min\{n^2, n\cdot m\})$ operations for setting up, visiting and finally removing all buckets and redistributing nodes to higher levels. As SP-S is a label-setting algorithm each edge is relaxed only once. Having a maximum hierarchy height of n this accounts for another $\mathcal{O}(n \cdot m)$ operations. $\square$

In Section 5 (Lemma 18) we will give an extension that reduces the worst-case time for a target bucket search (and hence for a (re-)relaxation) to $\mathcal{O}(1)$. After this modification, the asymptotic worst-case times of SP-S and SP-C will be $\mathcal{O}(\min\{n^2, n \cdot m\})$ and $\mathcal{O}(n \cdot m)$, respectively.

3 Average–Case Complexity

Now we show that both SP-C and SP-S achieve linear average-case time under the assumption of *random* real edge weights which are independent and uniformly distributed in the interval $[0, 1]$. It turns out that SP-S also succeeds in linear time with high probability whereas SP-C requires some modifications in order to reach the same high probability bound.

3.1 Redistributions

First we will bound the average-case overhead due to node redistributions during bucket splits of the label correcting approach SP-C.

Lemma 2 *For any node v, the average number of redistributions of v in SP-C is bounded from above by a constant.*

Proof: Let v be an arbitrary node with degree d and B_{cur} the first current bucket that is split with v being involved. Let $\Delta_{\text{cur, 0}}$ denote the width of B_{cur}, and let $d^* > 1/\Delta_{\text{cur, 0}}$ denote the maximum node degree in B_{cur}, i.e., a redistribution is required. If all incoming edges of v have weight at least $\Delta_{\text{cur, 0}}$, then v can be removed according to the criterion of Lemma 1 (and never re-enters a bucket)[3]. Otherwise, after the first redistribution, v is stored in a bucket of width $1/d^*$ or less. After the i-th time v was moved to a higher level, it is reconsidered in a current bucket $B_{\text{cur, i}}$ of width $\Delta_{\text{cur, i}} \leq 1/(2^{i-1} \cdot d^*)$.

For the analysis we use the principle of deferred decisions: in order to decide whether v needs to be redistributed from the current bucket of width Δ_{cur}, we query the incoming edges of v one by one. However, instead of checking their concrete weights, we just reveal whether they are smaller than Δ_{cur} or not. We stop after detecting the first edge $e^* = (u, v)$ with weight less than Δ_{cur}. The only knowledge we gained is that $c(e^*)$ is uniformly distributed in $[0, \Delta_{\text{cur}})$. Possible further redistributions of v are due to splits of buckets with even smaller widths. Thus, the edges into v which were tested prior to e^* (and therefore have weight at least Δ_{cur}) do not need to be queried again later. The weights of the (at most $d - 1$) remaining edges that have not yet been tested are uniformly distributed in $[0, 1]$.

Now assume that v was already redistributed i times and that the current bucket $B_{\text{cur, i}}$ it resides in is split: then $B_{\text{cur, i}}$ has width $\Delta_{\text{cur, i}} \leq 1/(2^{i-1} \cdot d^*)$. Furthermore, $\mathrm{P}[c(e^*) < \Delta_{\text{cur, i}}] <$

[3]Note that this criterion is independent of $\text{tent}(v)$ within B_{cur}.

$1/2$ since $\Delta_{\text{cur, i}} \leq 1/2 \cdot \Delta_{\text{cur, i-1}}$, and $c(e^*)$ is uniformly distributed in $[0, \Delta_{\text{cur, i-1}})$. Hence, the probability that v is not redistributed any more is at least

$$\frac{1}{2} \cdot (1 - \Delta_{\text{cur, i}})^{d-1} > \frac{1}{2} \cdot (1 - d \cdot \Delta_{\text{cur, i}}) > \frac{1}{2} - \frac{1}{2^i} .$$

Therefore, the expected number of redistributions for v can be bounded by

$$1 + \sum_{j=1}^{\infty} \prod_{i=1}^{j} (1 - (\frac{1}{2} - \frac{1}{2^i})) < 1 + \sum_{j=0}^{\infty} (\frac{3}{4})^j = 5 .$$

□

Similarly, we consider the average-case overhead for node redistributions during bucket splits of the label-setting variant SP-S:

Lemma 3 *For any node v with degree d, the average number of redistributions of v in SP-S is bounded from above by $\mathcal{O}(1 + \log d)$.*

Proof: Every split reduces the current bucket width by at least a factor of two. Hence, if a node v with degree d takes part in $i + \lceil \log d \rceil$ splittings for SP-S then we are in a similar situation as in the proof of Lemma 2 for SP-C after i splittings: v is reconsidered in a current bucket of width $\Delta_{\text{cur, i}} \leq 1/(2^{i-1} \cdot d)$. Therefore, after at most $\lceil \log d \rceil$ initial splittings the analysis for SP-C completely carries over to SP-S: the redistribution of v stops after another $\mathcal{O}(1)$ splittings on the average. □

We combine the results of the previous two lemmas with observations made in Section 2 in order to obtain the following partial result:

Lemma 4 *All bucket splittings of SP-C and SP-S can be done in average-case time $\mathcal{O}(n + m)$ in total.*

Proof: For SP-C, as already discussed for the worst-case bound, setting up, visiting and finally removing the buckets of the new level can be charged on the degree of the node which caused the splitting. As each node can cause at most one splitting this sums up to $\mathcal{O}(m)$ operations. The remaining operations involved in splittings are due to node redistributions; by Lemma 2 is amounts to $\mathcal{O}(1)$ for each node on the average, hence all splittings of SP-C can be done in $\mathcal{O}(n + m)$ average-case total time.

The number of new buckets created by a bucket splitting of SP-S is proportional to the number of nodes in the split bucket. Therefore, the total average-case costs for all split operations of SP-S are proportional to the total number of redistributed nodes, and this number can be bounded by $\mathcal{O}(n + \sum_{v \in G} \log \text{degree}(v)) = \mathcal{O}(n + m)$ using Lemma 3. □

3.2 Target Bucket Search

Now we address the complexity to find the appropriate target bucket for node (re-)insertions or (re-)relaxations. A modification presented in Section 5 reduces the worst-case search time from $\mathcal{O}$(max. hierarchy height) to $\mathcal{O}(1)$, thus immediately yielding a linear time average-case bound for SP-S (there are no node re-insertions and each edge is relaxed at most once). However, for random edge weights, the simple bottom-up search strategy of Section 2.3 is equally suited for both SP-C and SP-S.

Lemma 5 *The target bucket for an (re-)insertion or (re-)relaxation via an edge $e = (u, v)$ can be found in constant time on average.*

Proof: The weight of e is uniformly distributed in $[0, 1]$. Therefore, looking at a specific level L_i, the probability that $\text{tent}(v) = \text{tent}(u) + c(u, v)$ with $u \in B_{\text{cur}}$ belongs to L_i is at most the total width of L_i, $\Delta_{\text{i-1}}$. By Invariants 1 and 2, $\Delta_{\text{i-1}} \leq 2^{-i+1}$ for $i \geq 1$. Thus, for maximum hierarchy height h, the expected search time when checking levels in bottom-up fashion is bounded by

$$\mathcal{O}(1 + \sum_{i=1}^{h} i \cdot 2^{-i+1}) = \mathcal{O}(1 + \sum_{i=1}^{\infty} i \cdot 2^{-i+1}) = \mathcal{O}(1).$$

The average-case search time is maximal if the bucket hierarchy has been build by recursively splitting the first bucket of the topmost level into two new buckets and node u represents the smallest possible distance in the topmost level, see Figure 3. □

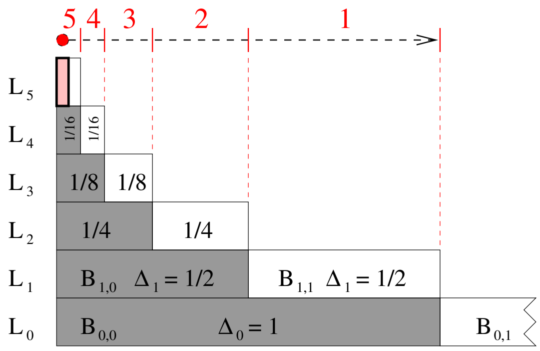

Figure 3: A worst-case setting for edge relaxations with bottom-up search.

3.3 The Label-Setting Bounds

We are ready to state the first result for SP-S:

Theorem 2 *SP-S runs on arbitrary directed graphs with random edge weights that are independent and uniformly drawn from $[0, 1]$ in $\mathcal{O}(n + m)$ average-case time.*

Proof: As shown in Section 2, the initialization for SP-S can be done in $\mathcal{O}(n + m)$ time. By Lemma 4 all bucket splits together require $\mathcal{O}(n + m)$ time on average. Due to the label-setting approach each edge is relaxed at most once: this accounts for $\mathcal{O}(m)$ target bucket searches each of which takes $\mathcal{O}(1)$ time on average by Lemma 5. □

It is surprisingly simple to show that the average-case time bound of Theorem 2 for SP-S also holds with high probability. Our basic tool is a variant of the well-known Chernoff-Hoeffding bounds:

Lemma 6 ([19, 33]) *Let the random variables $X_1, \ldots, X_n$ be independent, with $a_k \leq X_k \leq b_k$ for each k, for some arbitrary reals a_k, b_k. Let $X = \sum_k X_k$. Then for any $t \geq 0$,*

$$\mathrm{P}\left[|X - \mathrm{E}\left[X\right]| \geq t\right] \quad \leq \quad 2 \cdot \exp\left(\frac{-2 \cdot t^2}{\sum_k (b_k - a_k)^2}\right). \tag{1}$$

We have to reconsider the total number of nodes redistributions (Lemmas 3 and 4) and the total number of operations for target bucket searches (Lemma 5):

Lemma 7 *SP-S incurs $\mathcal{O}(n + m)$ node redistributions with high probability.*

Proof: By Lemma 3 (and Lemma 2) the probability that a node v with degree d is redistributed during bucket splittings on levels $L_{\lceil \log d \rceil + j}$ or higher is bounded from above by $\sum_j^\infty (3/4)^j$. Hence, we can choose $j = \Theta(\log n)$ to see that v is redistributed at most $r(v) \leq \lceil \log d \rceil + j = \Theta(\log n) = r$ times with probability at least $1 - n^{-c-1}$ for an arbitrary constant $c > 0$. Furthermore, by Boole's inequality, no node is redistributed more than r times with probability at least $1 - n^{-c}$. Note that the argumentation above also bounds the maximum height of the bucket hierarchy by $r = \Theta(\log n)$ whp.

Let X_v be the random variable denoting the upper bound of redistributions for node v as given in Lemma 3. Recall that only weights of edges entering node v have been used to derive X_v, i.e., these random variables are independent. Let $X = \sum_v X_v$. By Lemma 3 we have $\mathrm{E}[X] = \mathcal{O}(n + m)$. We define $X'_v = \min\{X_v, r\}$ and $X' = \sum_v X'_v$. Hence, $\mathrm{E}[X'] \leq \mathrm{E}[X]$. Now we can apply the Chernoff-Hoeffding bound (1) from Lemma 6 for X':

$$\begin{aligned}\mathrm{P}[X \geq \mathrm{E}[X] + t] \quad &\leq \quad \mathrm{P}[X' \geq \mathrm{E}[X] + t] + n^{-c} \leq \mathrm{P}[X' \geq \mathrm{E}[X'] + t] + n^{-c} \\ &\leq \quad 2 \cdot \exp\left(\frac{-t^2}{n \cdot r}\right) + n^{-c} \leq 2 \cdot n^{-c} \text{ for } t = n\,.\end{aligned}$$

□

Lemma 8 *All target bucket searches of SP-S can be performed in total $\mathcal{O}(n + m)$ time with high probability.*

Proof: Let X_e be the random variable denoting the upper bound on the level checks needed to perform a target bucket search for edge e as given in Lemma 5. Recall that X_e has been derived independently from all other edge weights, $\mathrm{E}[X_e] = \mathcal{O}(1)$, and that X_e takes values from one up to the maximum hierarchy height which in turn is bounded by $r = \Theta(\log n)$ whp according to Lemma 7. In complete analogy to the proof of that lemma we bound $\mathrm{P}[\sum_e X_e \geq \mathrm{E}[\sum_e X_e] + n]$ from above by $\mathcal{O}(n^{-c})$, $c \geq 1$, using Boole's inequality and the Chernoff-Hoeffding bound for random variables $X'_e = \min\{X_e, r\}$. □

The linear time high probability bound for SP-S immediately follows from Theorem 2 and the two last lemmas. The worst-case time $\Theta(n \cdot m)$ can be trivially avoided by monitoring the actual resource usage of SP-S and starting the computation from scratch with Dijkstra's algorithm after SP-S has consumed $\Theta(n \cdot \log n + m)$ operations. Similarly, SP-S can be combined with other algorithms in order to obtain improved worst-case bounds for non-negative integer edge weights [48].

Theorem 3 *Using the label-setting approach SP-S, single-source shortest-paths on arbitrary directed graphs with random independent edge weights uniformly drawn from* $[0, 1]$ *can be solved in* $\mathcal{O}(n + m)$ *time with high probability while preserving an* $\mathcal{O}(n \cdot \log n + m)$ *worst-case time bound.*

3.4 Average-Case Complexity of the Label-Correcting Approach

For the label-correcting version SP-C we still have to bound the overhead due to node re-insertions. Each node v reachable from the single source s will be inserted into the bucket structure $\mathcal{B}$ and removed from a current bucket at least once. A re-insertion of v into $\mathcal{B}$ occurs if v had already been deleted from some current bucket with non-final distance value $\mathrm{tent}(v)$ before, and the relaxation of an edge into v improves $\mathrm{tent}(v)$. A re-insertion of v subsequently triggers another deletion of v from a current bucket involving re-relaxation of v's outgoing edges. Hence, a re-insertion of v has to be clearly distinguished from a *redistribution* of v to a higher level bucket because of a bucket split. Furthermore, if v is already present in a so far unvisited bucket and v is moved to another bucket because of a relaxation, this is also *not* considered a node re-insertion of v. The two latter operations do not incur additional relaxations of v's outgoing edges.

We will map node re-insertions on paths between nodes (Lemma 10) using the observation that long paths with small total weight are unlikely for random edge weights (Lemma 9). In the following, a path without repeated edges and total weight at most Δ will be called a Δ-*path*.

Lemma 9 ([41]) *Given a path* $\mathcal{P}$ *of* l *non-repeated edges with independent and uniformly distributed edge weights in* $[0, 1]$*. The probability that* $\mathcal{P}$ *is a* Δ*-path is bounded by* $\Delta^l / l!$ *for* $\Delta \leq 1$.

Proof: Let X_i denote the weight of the i-th edge on the path. The total weight of the path is then $\sum_{i=1}^{l} X_i$. We prove by induction over l that $\mathbf{P}\left[\sum_{i=1}^{l} X_i \leq \Delta\right] = \Delta^l / l!$ for $\Delta \leq 1$: if $l = 1$ then due to the uniform distribution the probability that a single edge weight is at most $\Delta \leq 1$ is given by Δ itself: $\mathbf{P}[X_1 \leq \Delta] = \Delta$. Now we assume that $\mathbf{P}\left[\sum_{i=1}^{l} X_i \leq \Delta\right] = \Delta^l / l!$ for $\Delta \leq 1$ is true for some $l \geq 1$. In order to prove the result for a path of $l + 1$ edges, we split the path into a first part of l edges and a second part of one edge. For a total path weight at most Δ we have to consider all combinations for $0 \leq z \leq \Delta \leq 1$ such that the first part of l edges has weight at most $\Delta - z$ and the second part (one edge) has weight z. Thus,
$\mathbf{P}\left[\sum_{i=1}^{l+1} X_i \leq \Delta\right] = \int_0^{\Delta} \mathbf{P}\left[\sum_{i=1}^{l} X_i \leq \Delta - x\right] dx = \int_0^{\Delta} \frac{(\Delta - x)^l}{l!} \, dx = \frac{\Delta^{l+1}}{(l+1)!}$ $\Box$

After having formalized the observation that long paths with small total weight are unlikely we now map re-insertions on such paths. In the following, we will distinguish *local* re-insertions where the re-inserted node is put to the current bucket and *far* re-insertions where the target bucket is not the current bucket.

Lemma 10 *For any node v, SP-C incurs at most* 10 *re-insertions on average.*

Proof: Consider an arbitrary reachable node v with degree d. We first discuss re-insertions into a current bucket due to node removals from this very bucket (local re-insertions). According to Invariant 1, SP-C tries to settle v for the first time (say in phase t) by removing it from a current bucket B_{i_0,j_0} of level L_{i_0} having width $\Delta_{i_0} = 2^{-I_0} \leq 2^{-\lceil \log_2 d \rceil} \leq 2^{-i_0}$. The node v can only be re-inserted for phase $t+1$ in the following situation: v has an incoming edge $e = (u, v)$ of length at most Δ_{i_0}, u is removed from B_{i_0,j_0} in phase t and the relaxation of e gives an improvement for $\mathrm{tent}(v)$; compare Figure 4. Another re-insertion for the phase $t+2$ needs an incoming path $\langle u_1, u_2, v \rangle$ of two edges with sufficiently small total weight (at most Δ_{i_0}) such that u_1 is removed from B_{i_0,j_0} in phase t and u_2 is removed from B_{i_0,j_0} in phase $t+1$ and $\mathrm{tent}(v)$ improves. And so on. We define G_i to be the subgraph of G which is induced by all vertices with degree at most 2^i. All nodes in B_{i_0,j_0} have degree at most 2^{I_0}, hence the paths required for re-insertions of v in B_{i_0,j_0} are simple (2^{-I_0})-paths into v in G_{I_0}.

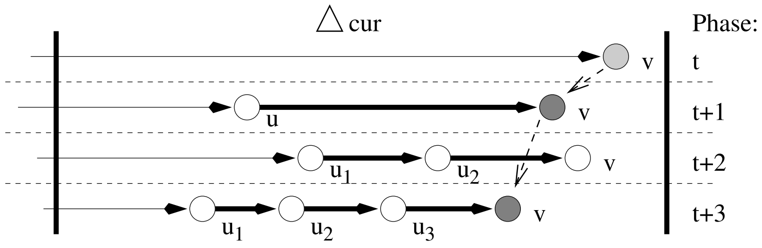

Figure 4: Local re-insertions of node v in phases $t+1$ and $t+3$. There is no re-insertion of v in phase $t+2$ because $\mathrm{tent}(v)$ is not improved.

If B_{i_0,j_0} is split and v is not filtered out and settled by an early removal, then v may be removed again from current buckets $B_{i_1,j_1}, \ldots, B_{i_k,j_k}$ of levels $i_1 < \ldots < i_k$ which cover smaller and smaller parts of B_{i_0,j_0}. As above, all nodes in B_{i_x,j_y} have degree at most 2^{I_x} Thus, after the k-th phase of emptying B_{i_x,j_y}, v can (re-)enter the current bucket B_{i_x,j_y} for the next phase via a (2^{-I_x})-path of k edges into v in G_{I_x} provided that its total path weight is sufficiently small.

Therefore, an upper bound on the total number of paths identified above (which in turn bound the number of local re-insertions of v) is given by[4]

$$\sum_{i=\lceil \log_2 d \rceil}^{\lceil \log_2 2n \rceil - 1} \text{\# simple } (2^{-i})\text{-paths into } v \text{ in } G_i = \sum_{i=0}^{\lceil \log_2 2n \rceil - 1} \text{\# simple } (2^{-i})\text{-paths into } v \text{ in } G_i \quad (2)$$

Since all nodes in G_i have degree at most 2^i, no more than $d \cdot (2^i)^{(l-1)}$ simple paths of l edges each enter node v in G_i. Taking into account the probability that these paths are (2^{-i})-paths

[4]Note that the two sums are equivalent since v is not part of the subgraphs $G_0, \ldots, G_{\lceil \log_2 d \rceil - 1}$.

(Lemma 9), the average-case number of these paths into v over all levels is bounded from above by

$$\sum_{i=k}^{\lceil \log_2 2n \rceil} \sum_{l=1}^{\infty} d \cdot 2^{i\cdot(l-1)} \cdot 2^{-i\cdot l}/l! = \sum_{i=k}^{\lceil \log_2 2n \rceil} \frac{d}{2^i} \cdot \left(\sum_{l=1}^{\infty} \frac{1}{l!} \right) \leq \sum_{i=k}^{\lceil \log_2 2n \rceil} 2^{-i+k} \cdot e \leq 2 \cdot e \,.$$

We still have to deal with those re-insertions of v into the bucket hierarchy $\mathcal{B}$ where the target bucket is *not* the current bucket (far re-insertions). Note that such a re-insertion of v will take place in a bucket $B_{k,l}$ which covers a fraction from the distance range of that current bucket $B_{i,j}$, $i < k$, from which v was deleted for the last time. The node v will be eventually encountered: either after the algorithm changed the current bucket or – after an update of $\mathrm{tent}(v)$ – node v moved into the current bucket from where it will be deleted in the next phase. After that, further re-insertions of v will take place in the current bucket as local re-insertions, until either v is settled or the current bucket is split again. Therefore, a far re-insertion of v can happen at most once per level.

However, after the very first removal of v at all, from a bucket B_{i_0,j_0}, $\mathrm{tent}(v)$ is already correct up to at most $\Delta_{i_0} = 2^{-I_0} \leq 2^{-\lceil \log_2 d \rceil}$. The first far re-insertion of v may only happen for some level L_{i_1}, $i_1 > i_0$, having total level width at most $\Delta_{i_0} = 2^{-I_0} \leq 2^{-\lceil \log_2 d \rceil}$: a far re-insertion of v into B_{i_1,j_1} requires a node u which is removed from some current bucket $B_{x,y}$, $x > j_0$ and an edge (u,v) with weight less than the total width of L_{i_1}. Hence, the probability that the weight of (u,v) is sufficiently small is bounded from above by $\Delta_{i_0} \leq 2^{-\lceil \log_2 d \rceil}$. The widths of higher levels decrease geometrically, $\Delta_{i_k} \leq 2^{-\lceil \log_2 d \rceil - k}$. See Figure 5 for an example. Once all incoming edges of v have weight larger than the level width under consideration, no

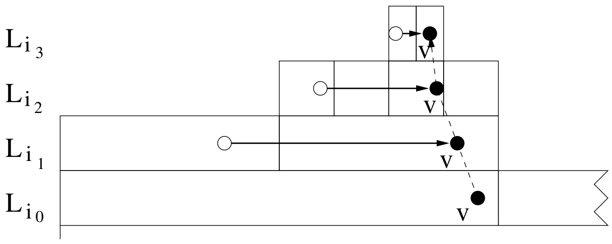

Figure 5: Far re-insertions for node v at higher levels require smaller edge weights.

further far re-insertions of v can happen for higher levels. Therefore, in complete analogy to the number of redistributions of a node v during bucket splits (Lemma 2) we can give an upper bound on the average number of far re-insertions of v by just examining the weights of incoming edges of v in piecemeal fashion and exploit that after each level there is a constant probability that no further far re-insertions of v will happen; we conclude that the average number of far re-insertions of v is bounded by

$$\sum_{j=1}^{\infty} \prod_{i=1}^{j} (1 - (\frac{1}{2} - \frac{1}{2^i})) < \sum_{j=0}^{\infty} (\frac{3}{4})^j = 4 \,.$$

Summing up the two bounds for local and far re-insertions we find that v is re-inserted at most $2 \cdot e + 4 < 10$ times on average. □

Corollary 1 *The expected number of re-insertions of SP-C is bounded by* $\mathrm{E}[n_+] = \mathcal{O}(n)$ *and the expected number of re-relaxations is bounded by*

$$\mathrm{E}[m_+] \leq 10 \cdot \sum_{v \in G} \textit{out-degree}(v) = \mathcal{O}(m)\,.$$

Theorem 4 *SP-C runs on arbitrary directed graphs with random edge weights that are independent and uniformly drawn from* $[0,1]$ *in* $\mathcal{O}(n+m)$ *average-case time.*

Proof: Similar to SP-S, the initialization of SP-C can be done in $\mathcal{O}(n+m)$ time. By Lemma 4, all bucket splits require accumulated $\mathcal{O}(n+m)$ time on average. According to Corollary 1 there are $\mathcal{O}(n+m)$ (re-)insertions and (re-)relaxations on average each of them involving a target bucket search. The proof of Lemma 5 upper-bounds the average-case search time for an edge (v,w) independent of the placement of v in the bucket hierarchy and the number of times v is re-inserted. On the other hand, the weights of edges leaving v are not used to bound the number of re-insertions for v in the proof of Lemma 10. Hence, we can combine Corollary 1 and Lemma 5 to obtain the average-case linear time bound. □

4 A High–Probability Bound for SP-C

In order to prove a linear time bound that holds with high probability, the label correcting approach SP-C has to be changed in some aspects. In its current form – even though the total number of re-insertions is bounded by $\mathcal{O}(n)$ on average – some nodes may be re-inserted a logarithmic number of times with non-negligible probability. If such a node has out-degree $\Theta(n)$ then SP-C incurs a super-linear number of re-relaxations since each time all outgoing edges are relaxed.

In Section 4.1 we give the modifications for SP-C* to cope with that problem; Section 4.2 provides the adapted analysis for the high probability bounds.

4.1 Modified Algorithm

Our refined version which we will call SP-C* applies deferred edge relaxations: Only edges of weight at most the current bucket width are immediately relaxed. Due to the correlation of bucket widths and maximum node degrees this means that for each deleted node only a constant number of edges will be immediately relaxed in expectation and not much more with high probability. Correctness is preserved since larger edge weights cannot influence the tentative distances in the current bucket. Edges with larger weight are relaxed (with proper distance values) once their source node is guaranteed to be settled. We first give the appropriate modifications of SP-C:

1. For each node u create a list O_u, that keeps the outgoing edges of u in *semi-sorted order*[5] according to their weights, i.e., O_u consists of at most $\lceil \log_2 2n \rceil + 1$ groups where group g_i,

[5] The semi-sorted order can be obtained in linear time by integer sorting for small values; edge weights $c(e_k) > 2^{-\lceil \log_2 2n \rceil}$ are mapped to $\lfloor \log_2 1/c(e_k) \rfloor$ thus creating integers from 0 to $\lceil \log_2 2n \rceil - 1$, smaller edge weights are mapped to the integer $\lceil \log_2 2n \rceil$. After the sorting, the order is reversed such that large integers (which account for small edge weights) appear first.

$0 \le i < \lceil \log_2 2n \rceil$ holds all outgoing edges e_k with $2^{-i-1} < c(e_k) \le 2^{-i}$ in arbitrary order, $g_{\lceil \log_2 2n \rceil}$ keeps all edges with weight at most $2^{-\lceil \log_2 2n \rceil}$ in arbitrary order, and g_i appears before g_j for all $j < i$.

2. Let u be a node which is removed from B_{cur}. Unless u is removed from B_{cur} during a bucket split (i.e., $\mathrm{tent}(u)$ is final by Lemma 1), only edges (u, v) with weight at most Δ_{cur} are relaxed. They always constitute the first groups in O_u due to the semi-sorting. The remaining edges are relaxed once and for all after B_{cur} and its refining buckets (if present) finally remain empty. By then, all involved nodes are surely settled.

In order to remember those edges which still have to be relaxed, every bucket $B_{\text{i,j}}$ is equipped with an additional edge list $E_{\text{i,j}}$. Whenever a node u is deleted from $B_{\text{cur}} = B_{\text{i,j}}$ (either by removing or redistributing) we move all remaining edges e_k satisfying $c(e_k) > \Delta_{\text{cur}}$ from O_u to $E_{\text{i,j}}$. Note that these edges constitute the last groups in O_u. Hence, if u is re-inserted into B_{cur} no transferable edges are left in O_u, and this can be checked in constant time. All edges in $E_{\text{i,j}}$ are relaxed once after $B_{\text{i,j}}$ finally remains empty or – in case $E_{\text{i,j}}$ has been split – after its refining level L_{i+1} is removed. It is easy to check by induction that this schedule of deferred edge relaxations does not affect the correctness of the algorithm since all potentially relevant edges are relaxed in time.

4.2 Modified Average-Case Analysis

In the following we will give separate high probability bounds on the total number of local re-insertions (re-relaxations), n'_+ (m'_+), and the total number of far re-insertions (re-relaxations), n''_+ (m''_+). By Boole's inequality, the combined bounds, $n_+ = n'_+ + n''_+$ and $m_+ = m'_+ + m''_+$ also hold with high probability.

In Section 4.2.1 we start with an alternative proof of $\mathrm{E}[n'_+] = \mathcal{O}(n)$ that is applicable to both SP-C and SP-C* (Lemma 11 and Corollary 2). We extend the idea of this proof to show that the modifications of SP-C* reduce $\mathrm{E}[m'_+]$ from $\mathcal{O}(m)$ to $\mathcal{O}(n)$ as well (Corollary 3). Section 4.2.2 provides some observations concerning the number of edges on paths with small total weight (Lemma 12) and the number of nodes reachable via such paths (Lemma 13). Using these observations together with the alternative proof scheme of Lemma 11 we can apply Azuma's Martingale tail inequality [38] to yield whp-bounds on n'_+ and m'_+ for local re-insertions and their associated edge re-relaxations in Section 4.2.3. Equivalent whp-bounds for far re-insertions and their re-relaxations are given in Section 4.2.4. Finally, re-considering the overhead for node re-distributions during bucket splits and target bucket searches (Section 4.2.5) completes the proof for the whp-bound of SP-C*.

4.2.1 Mapping Local Re-Insertions to Connection Pairs

Our average-case analysis of Section 3.4 applied a pretty intuitive mapping of local node re-insertions to paths of small weight in induced subgraphs. For the whp-bound we want to use Azuma's Martingale tail inequality [38] in order to show that a modification of edge weights out of some arbitrary node will not dramatically change the total number of local re-insertions. However, – given our crude estimates on the maximum number of paths with small total weight, Lemma 10, (2) – a modification of an edge weight could affect too many of these paths. Therefore, we will map the local re-insertions on *connection pairs* between nodes deleted from the same bucket. This mapping is also valid for the basic algorithm SP-C. Some parts of the following analysis extend proofs from the full version[6] of [41].

As before, a path with total weight at most Δ which does not follow any edge more than once (a path "without edge repetition") will be called a *Δ-path*. We define G_i to be the subgraph of G which is induced by all vertices with degree at most 2^i. Let C_i be the set of all node pairs $[u, v]_i$ connected by some (2^{-i})-path $\langle u, \ldots, v\rangle$ in G_i, hence all nodes on the path $\langle u, \ldots, v\rangle$ have degree at most 2^i. Furthermore, let C_i^v be the set of those pairs in C_i whose second component is v. We give an injective mapping from the set of local re-insertions of v into the set of connection pairs $C^v = \cup_i C_i^v$.

Lemma 11 *For both SP-C and SP-C*, the total number of local re-insertions of a node v is bounded by $\sum_{i=\lceil \log_2 \text{degree}(v)\rceil}^{\lceil \log_2 2n\rceil -1} |C_i^v|$.*

Proof: Consider a local node re-insertion of v into a fixed current bucket $B_{\text{cur}} = [a, a + \Delta_{\text{cur}})$ for phase t. Hence, all nodes with final distance less than a have already been assigned their correct label, they are settled; $\text{tent}(v)$ is correct up to at most Δ_{cur}. Due to Invariant 1, $\Delta_{\text{cur}} = 2^{-j} \leq 2^{-k}$, for some integer $j \geq k$, where $k = \lceil \log_2 \text{degree}(v)\rceil$. Unless B_{cur} is split or finally remains empty, both SP-C and SP-C* explore paths within G_j.

Let us fix a (constrained) *shortest path* $A(v, B_{\text{cur}}) = \langle s, \ldots, v_0, v_1, \ldots, v_l, v\rangle$ from s to v such that all nodes $s, \ldots, v_0$ are reached in optimal shortest path distance less than a (they are settled in G), and $v_1, \ldots, v_l$ with distance at least a have degree at most 2^j. Hence, $\langle v_1, \ldots, v_l, v\rangle$ is a shortest path from v_1 to v in G_j. Unless B_{cur} is split, the node v_i will be removed from B_{cur} when for the i-th time all nodes are removed from this bucket; after that v_i will never re-enter B_{cur}. However, note that in case B_{cur} is split and paths from $G_{j'}$, $j' > j$ are considered in the suffix of the shortest path from s to v, the distance of v_i may improve. In that case v_i will appear again in some higher level bucket of width $2^{-j'}$. Therefore, after the i-th phase of emptying B_{cur}, v_i is only *locally* settled whereas all nodes on the sub-path $\langle s, \ldots v_0\rangle$ are already *globally* settled.

Since we are considering a local re-insertion of v there must be a most recent phase $t' < t$ when v was removed from this particular B_{cur} and no split happened since then. Let v' be the first locally unsettled node on $A(v, B_{\text{cur}})$ immediately before phase t'. Hence, according to the constraints given above, v' has reached its local shortest path distance for G_j after the edge relaxations of phase $t' - 1$. It will be removed from B_{cur} in phase t', locally settled, and will never re-enter this current bucket. By definition, v is removed from B_{cur} in phase t' as well, but it will re-enter B_{cur} in phase t, therefore $v' \neq v$. The sub-path $A'(v', v) = \langle v', \ldots, v\rangle$ of $A(v, B_{\text{cur}})$ is a shortest path from v' to v in G_j. Since both v' and v are removed together from B_{cur} with bucket width 2^{-j} in phase t', $A'(v', v)$ is also a (2^{-j})-path, i.e., $[v', v]_j \in C_j^v$. As v' becomes locally settled for bucket width (2^{-j}), the re-insertion of v in phase t can be uniquely mapped to $[v', v]_j$. $\square$

Corollary 2 *For both SP-C and SP-C*, the total number of local re-insertions is bounded by $n'_+ \leq \sum_{i=0}^{\lceil \log_2 2n\rceil -1} |C_i|$, and $|C_i|$ is bounded by the number of simple (2^{-i})-paths in G_i. Hence, by Lemma 10, $\mathrm{E}[n'_+] = \mathcal{O}(n)$.*

[6] http://www.mpi-sb.mpg.de/~umeyer/dps/delta-long.ps.gz

According to our modifications listed above, re-relaxations of $e = (v, w)$ for $v \in B_{\text{cur}}$ can only occur if $c(e) \leq \Delta_{\text{cur}}$. Therefore, arguing along the lines of Lemma 11, re-relaxations due to local re-insertions (in the following called *locally caused re-relaxations*) can be uniquely mapped to the set $C^* = \cup_i C_i^*$ where C_i^* holds triples $[u, v, w]_i$ with $[u, v]_i \in C_i$ and $e = (v, w)$ having weight at most 2^{-i}. Thus, an upper bound on the total number of locally caused re-relaxations, m'_+ is given by $m'_+ \leq |C^*|$. Similarly to Lemma 10, the number of locally caused re-relaxation for edges out of node v where $k_v = \lceil \log_2 \text{degree}(v) \rceil$ can be bounded by

$$\sum_{i=k_v}^{\lceil \log_2 2n \rceil - 1} \left(\text{\# simple}(2^{-i})\text{-paths into } v \text{ within } G_i \cdot \text{ \# edges } e = (v, w) \text{ with } c(e) \leq 2^{-i} \right) \quad (3)$$

Furthermore, m'_+, does not exceed

$$\sum_{v \in G} \sum_{i=k_v}^{\lceil \log_2 2n \rceil - 1} \text{\# } (2^{-i+1})\text{-paths without repeated edges into } v \text{ in } G_i \quad (4)$$

since any path $\langle u, \ldots v, w \rangle$ in G_i with non-repeated edges where $\langle u, \ldots v \rangle$ is a (2^{-i})-path and the edge (v, w) has weight at most 2^{-i} is also a (2^{-i+1})-path in G_i, but not the other way round.

Corollary 3 *By Lemma 9, Formula (4) and the observation that there are at most $2^{i \cdot l}$ paths with non-repeated edges of length l into each node of G_i, it follows that the total expected number of locally caused re-relaxations in SP-C* is bounded by* $\mathrm{E}[m'_+] = \mathcal{O}(n)$.

4.2.2 Some Observations for Random Edge Weights

The actual values of our bounds n'_+ and m'_+ can be regarded as functions of the n independent random variables $X_v := \{(v, w, c(v, w)) : (v, w) \in E)\}$, i.e., X_v describes all edges leaving node v including their weight. On the other hand, n'_+ and m'_+ are bounded by the sets $|C|$ and $|C^*|$, respectively, and the sizes of these sets in turn are bounded by certain numbers of paths (Formulae (2) and (4)).

In the following we will show that all these paths consist of $\mathcal{O}(\log n / \log \log n)$ edges whp (Lemma 12) and that number of nodes reached from some arbitrary node v via such paths is rather limited whp (Lemma 13). Therefore, the modification of a single random variable X_v can only affect a limited number of entries of the sets C and C^*. These observations will be used in Section 4.2.3. to yield the desired high-probability bound on n'_+ and m'_+ using Azuma's Martingale inequality.

Lemma 12 *Each (2^{-i+1})-path without repeated edges in G_i contains no more than $l^* = \mathcal{O}(\frac{\log n}{\log \log n})$ edges for all $0 \leq i < \lceil \log_2 2n \rceil$ whp.*

Proof: There are at most $2^{i \cdot l}$ paths without edge repetitions of length l into each node of G_i. By Lemma 9, the probability for the presence of any (2^{-i+1})-path in G_i with l edges can be bounded by $n \cdot (2^{-i+1} \cdot 2^i)^l / l! = n \cdot 2^l / l!$. Hence, by Boole's inequality there are no (2^{-i+1})-paths of $l \geq l^*$ edges in G_i for all $0 \leq i < \lceil \log_2 2n \rceil$ with probability at least

$$\begin{aligned} & 1 - \sum_{l \geq l^*} n \cdot \lceil \log_2 2n \rceil \cdot 2^l / l! \geq 1 - n \cdot \lceil \log_2 2n \rceil \cdot 2^{l^*} / l^*! \cdot \sum_{l \geq 0} 2^l / l! \\ \geq \quad & 1 - n \cdot \lceil \log_2 2n \rceil \cdot 2^{l^*} / \Theta((l^*/e)^{l^*} \cdot \sqrt{l^*}) \cdot e^2 \geq 1 - \mathcal{O}(n \cdot \log n) \cdot (\frac{2 \cdot e}{l^*})^{l^*} \geq 1 - n^{-c} \end{aligned}$$

for some arbitrary positive constant c and $l^* = \Theta(\frac{\log n}{\log \log n})$. □

Clearly, the bound of Lemma 12 also holds for (2^{-i})-paths without edge repetitions in G_i. Now we bound the number of nodes that are either reached from a given node v via (2^{-i})-paths without repeated edges in G_i or reach node v via such paths. Similarly, we consider reachability in G_i via (2^{-i+1})-paths that consist of a (2^{-i})-path plus a terminal edge of weight at most 2^{-i}:

Lemma 13 *For any node v,*

$$|\{u : [v,u]_i \in C \ \vee \ [u,v]_i \in C,\ 0 \le i < \lceil \log_2 2n \rceil\}| \le n^{\mathcal{O}(1/\log\log n)}$$

and

$$|\{u : [v,w,u]_i \in C^* \ \vee \ [u,w,v]_i \in C^*,\ 0 \le i < \lceil \log_2 2n \rceil\}| \le n^{\mathcal{O}(1/\log\log n)}$$

with high probability.

Proof: Consider the nodes reached from v via (2^{-i+1})-paths in G_i. We argue that the number of nodes reachable by connections in C_i of length l can be bounded by the offspring in the l-th generation of the following branching process ([3, 31] provide an introduction to branching processes): An individual (a node) u has its offspring defined by the number Y_u of edges with weight at most 2^{-i+1} leaving it. Let Z_l denote the total offspring after l generations when branching from node v (i.e., $Z_1 = Y_v$). The *branching factor* of a branching process is given by $\rho = \mathbf{E}[Z_1]$. Furthermore, for branching processes with identical and independent probabilities for the generation of new nodes, $\mathbf{E}[Z_l] = \rho^l$. Additionally, it is shown in [2] (as cited in [35, Theorem 1]) that $Z_l = \mathrm{O}\big(\rho^l \log n\big)$ whp.

As long as new edges are encountered when following paths out of v, the offspring of the branching process is an exact model for the number of paths leaving v which use only edges with weight at most 2^{-i+1}. After a node has been reached from multiple paths, the events on those paths are no longer independent. However, all but one of the multiple paths produce only duplicate entries into C_i. The additional paths can therefore be discarded. All remaining events are independent.

For independent edge weights uniformly distributed in $[0,1]$, we have $\mathrm{E}[Y_u] \le \text{degree}(u) \cdot 2^{-i+1} \le 2$ and by the discussion above, $\mathrm{E}[Z_l] \le (\text{max-degree}(G_i) \cdot 2^{-i+1})^l \le 2^l$ and

$$Z_l = \mathcal{O}((\text{max-degree}(G_i) \cdot 2^{-i+1})^l \cdot \log n) = \mathcal{O}(2^l \cdot \log n) \text{ whp.}$$

In order to asymptotically bound the sum $\sum_{j \le l} Z_j$ we can concentrate on term Z_l since a growing exponential sum is dominated by its last summand: $\sum_{j \le l} Z_j = \mathcal{O}(2^l \cdot \log n)$ whp. By Lemma 12 we have $l \le l^* = \mathcal{O}(\frac{\log n}{\log\log n})$ whp. Therefore, the number of nodes reached from v via (2^{-i+1})-paths in G_i is at most $\mathcal{O}(2^{l^*} \cdot \log n)$ whp. Since the number of nodes reached from v via (2^{-i})-paths in G_i can only be smaller we obtain the bound $|\{u : [v,u]_i \in C_i\}| = \mathcal{O}(2^{l^*} \cdot \log n)$ whp. Similarly, we obtain $|\{u : [u,v]_i \in C_i\}| = \mathcal{O}(2^{l^*} \cdot \log n)$ whp by just traversing the edges during the branching process in opposite direction. Finally, using Boole's inequality we get

$$\begin{aligned} |\{u : [v,u]_i \in C \vee u : [u,v]_i \in C,\ 0 \le i < \lceil \log_2 2n \rceil\}| &= \mathcal{O}\left(2 \cdot \lceil \log_2 2n \rceil \cdot 2^{\mathcal{O}(\frac{\log n}{\log\log n})} \cdot \log n\right) \\ &= n^{\mathcal{O}(\frac{1}{\log\log n})} \text{ whp} \end{aligned}$$

for any node v. For the elements of C_i^* note that $|\{u : [v,w,u]_i \in C_i^*\}|$ is bounded by the set of nodes reached from v via (2^{-i+1})-paths in G_i which in turn we have shown to be bounded by $\mathcal{O}(2^{l^*} \cdot \log n)$ whp. Repeating the argument above yields the statement for elements in C^*. □

4.2.3 Local Re-Insertions

Equipped with the observations from Section 4.2.2 we are now ready to show

Lemma 14 *SP-C* incurs at most $n'_+ = \mathcal{O}(n)$ local re-insertions and $m'_+ = \mathcal{O}(n)$ locally caused re-relaxations with high probability.*

Proof: Let Q stand for the event that the whp-bounds of Lemma 13 hold for all nodes v and that at most $a \cdot \log n$ edges with weight at most (2^{-i+1}) leave any node v in G_i for all $i \le \lceil 2 \cdot \log n \rceil$. By Boole's inequality and Chernoff bounds [38], Q holds whp for some constant a and sufficiently large n. If Q does not hold then $n'_+ \le n^2 \cdot \log n$ and $m'_+ \le n^3 \cdot \log n$ by the definitions of the sets C and C^*. Therefore, $\mathrm{E}[n'_+ \,|\, Q] \le \mathrm{E}[n'_+] + o(n) = \mathcal{O}(n)$ and $\mathrm{E}[m'_+ \,|\, Q] \le \mathrm{E}[m'_+] + o(n) = \mathcal{O}(n)$. It suffices to bound $\mathrm{P}[n'_+ > \mathrm{E}[n'_+ \,|\, Q] + t \,|\, Q]$ and $\mathrm{P}[m'_+ > \mathrm{E}[m'_+ \,|\, Q] + t \,|\, Q]$ for some $t = \mathcal{O}(n)$.

By our definition of the n independent random variables $X_v := \{(v, w, c(v, w)) : (v, w) \in E)\}$ we can apply Azuma's Martingale tail inequality [38] to show

$$\mathrm{P}[n'_+ > \mathrm{E}[n'_+ \,|\, Q] + t \,|\, Q] \le \exp\left(\frac{-t^2}{2 \cdot \sum_{v \in G} c_v^2}\right)$$

where c_v is defined such that (given Q) changing the value of variable X_v (while maintaining Q) could change $\sum_j |C_j| \ge n'_+$ by at most c_v.

Now we determine c_v for the bound on n'_+: as Q holds, there are at most $n^{a/\log\log n}$ connection pairs $[u, v]_i \in C_i$ with second component v. Furthermore, the number of edges with weight at most 2^{-i} leaving v is at most $a \cdot \log n$, and their target nodes in turn reach at most $n^{a/\log\log n}$ nodes each via (2^{-i})-paths in G_i. Therefore, multiplying these three quantities gives an upper bound for the number of elements affected in C_i by a modification of X_v while maintaining Q. Since the argument is valid for all subgraphs G_i, we have $c_v \le \lceil \log_2 2n \rceil \cdot (n^{a/\log\log n})^2 \cdot a \cdot \log n = n^{\mathcal{O}(1/\log\log n)}$. Applying this argument to all nodes we get $2 \cdot \sum_{v \in G} c_v^2 \le 2 \cdot n \cdot (n^{\mathcal{O}(1/\log\log n)})^2 = n \cdot n^{\mathcal{O}(1/\log\log n)}$ and

$$\mathrm{P}[n'_+ > \mathrm{E}[n'_+ \,|\, Q] + n \,|\, Q] \le e^{-n^{\left(1 - \mathcal{O}(\frac{1}{\log\log n})\right)}}.$$

Since $\mathrm{E}[n'_+ \,|\, Q] \le \mathrm{E}[n'_+] + o(n) = \mathcal{O}(n)$ because Q holds whp, the last bound can be simplified to $\mathrm{P}[n'_+ > \mathcal{O}(n)] \le n^{-c}$ for any positive constant $c > 1$. Analogously,

$$\mathrm{P}[m'_+ > \mathrm{E}[m'_+ \,|\, Q] + t \,|\, Q] \le \exp\left(\frac{-t^2}{2 \cdot \sum_{v \in G} \tilde{c_v}^2}\right)$$

where $\tilde{c_v}$ is defined such that changing the value of variable X_v respecting Q could influence $\sum_j |C^*_j| \ge m'_+$ by at most $\tilde{c_v}$. As shown above, such a modification of X_v changes $\sum_j |C_j|$ by at most $n^{\mathcal{O}(1/\log\log n)}$. By Q, there are at most $a \cdot \log n$ edges with weight 2^{-i} out of node w for each affected node pair $[u, w]_i \in C_i$. Hence, at most $n^{\mathcal{O}(1/\log\log n)} \cdot a \cdot \log n = n^{\mathcal{O}(1/\log\log n)}$ entries of $\sum_j |C^*_j|$ are affected, i.e., $\tilde{c_v} \le n^{\mathcal{O}(1/\log\log n)}$. The rest of the argument for the whp-bound on m'_+ is the same as for the bound on n'_+.

□

4.2.4 Far Re-Insertions and Redistributions

So far we have shown that the bounds on the total number of local reinsertions, n'_+, and the total number of locally caused re-relaxations, m'_+, do not significantly deviate from their expected values whp. Now we do the same for the total number of far re-insertions, n''_+, and the total number of re-relaxations caused by them, m''_+:

Lemma 15 *SP-C* incurs at most $n''_+ = \mathcal{O}(n)$ far re-insertions and $m''_+ = \mathcal{O}(n)$ far caused re-relaxations with high probability.*

Proof: We define the n independent random variables $X'_w := \{(v, w, c(v, w)) : (v, w) \in E)\}$ as sets of edge weights *into* node w. Lemma 10 derives an upper bound $f(X'_v)$ for the number of far re-insertions for any node v by exclusively taking into consideration the weights of the edges entering v. In particular, $\mathrm{E}[f(X'_v)] = \mathcal{O}(1)$ and $\mathrm{E}[n''_+] = \mathcal{O}(n)$. There is at most one far re-insertion of v for each level, and the number of levels is bounded by $\mathcal{O}(\log n)$. Hence, we can give a high probability bound on the sum of all far re-insertions, using the Chernoff-Hoeffding bound (Fomula (1)) with $a_k = 0$, $b_k = \mathcal{O}(\log n)$ and $t = n$ in order to show $n''_+ = \sum_v f(X'_v) \leq \mathrm{E}[n''_+] + \mathcal{O}(n) = \mathcal{O}(n)$ whp.

For the number of far caused re-relaxations we use a similar argument as in the proof of Lemma 14: let Q' be the event that in G_i at most $a \cdot \log n$ edges with weight at most 2^{-i} leave any node. Given Q', the far re-insertion of a node v into a bucket of width 2^{-i} triggers at most $a \cdot \log n$ re-relaxations when v is deleted (recall that the algorithm relaxes edges (v, w) with weight larger than the current bucket width only after v is surely settled).

Q' holds whp for an appropriate constant a and sufficiently large n (Chernoff bounds). If Q' is not true (this happens with probability less than n^{-c} for some arbitrarily large positive constant $c \geq 3$) each node can take part in at most $\mathcal{O}(\log n)$ far re-insertions each of which relaxes at most n edges. Therefore, $\mathrm{E}[m''_+ \,|\, Q] \leq \mathrm{E}[m''_+] + o(n)$ and it suffices to bound $\mathrm{P}[m''_+ > \mathrm{E}[m''_+ \,|\, Q'] + t \,|\, Q']$ for some $t = \mathcal{O}(n)$. Again we apply Azuma's Martingale tail inequality to show

$$\mathrm{P}[m''_+ > \mathrm{E}[m''_+ \,|\, Q'] + t \,|\, Q'] \leq \exp\left(\frac{-t^2}{2 \cdot \sum_{v \in G} c'^2_v}\right)$$

where changing the value of variable X'_v could change m''_+ by at most c'_v (given Q'). By the discussion above, modifying X'_v will change n''_+ by at most $\mathcal{O}(\log n)$, and each such far re-insertion triggers at most $\mathcal{O}(\log n)$ edge re-relaxations (given Q'). Therefore $c'_v = \mathcal{O}(\log^2 n)$, i.e., $\mathrm{P}[m''_+ > \mathrm{E}[m'_+ \,|\, Q] + n \,|\, Q] \leq e^{-n/\mathcal{O}(log^4 n)}$. Using $\mathrm{E}[m''_+ \,|\, Q'] = \mathrm{E}[m''_+] + \mathcal{O}(n) = \mathcal{O}(\mathrm{E}[n''_+]) + \mathcal{O}(n) = \mathcal{O}(n)$ we get $\mathrm{P}[m''_+ > \mathcal{O}(n)] \leq n^{-c}$ for any positive constant $c > 1$. □

By now we have obtained whp-bounds on the total number of both types of node re-insertions and edge re-relaxations. We still have to check the overhead due to node redistributions during bucket splits:

Lemma 16 *All node redistributions can be done in total $\mathcal{O}(n)$ time whp.*

Proof: As the number of node redistributions is bounded in the same way as the number of far node re-insertions (compare Lemma 10), the proof is equivalent to the proof for the high probability bound of n''_+ in Lemma 15. □

4.2.5 Target Bucket Searches

Finally, we will show a that the target bucket searches involved in all (re-) insertions and (re-) relaxations can be done with high probability using a linear number operations in total:

Lemma 17 *SP-C* performs all target bucket searches in total $\mathcal{O}(n+m)$ time whp.*

Proof: Let X_i be the weight of the i-th edge in G. Let $r(X)$ denote an upper bound on the total time for all target bucket searches by assuming for each source node a worst-case position in the bucket hierarchy as presented in Lemma 5. We have $\mathrm{E}[r] = \mathcal{O}(\mathrm{E}[m_+] + m) = \mathcal{O}(n+m)$ since the expected time for a single target bucket search with worst-case source node position is $\mathcal{O}(1)$.

Let R be the proposition that the statements of Lemma 12, Lemma 14, and Lemma 15 hold (this happens whp). Otherwise there are at most $\mathcal{O}(m \cdot n)$ relaxations each of which takes $\mathcal{O}(\log n)$ time. Hence, $\mathrm{E}[r \mid R] \leq \mathrm{E}[r] + o(n+m) = \mathcal{O}(n+m)$.

In the best case an edge does not have to be relaxed at all. On the other hand by Lemma 12 and proposition R, each edge can be relaxed at most $\mathcal{O}(\log n / \log\log n)$ times per level, thus altogether $\mathcal{O}(\log^2 n)$ times, and each relaxation takes at most $\mathcal{O}(\log n)$ operations. By the definition of $r(X)$ the changes of node placements induced by a modification of e_i will not affect $r(X)$ for edges $e_j \neq e_i$. Thus, modifying X_i will change $r(X)$ by at most $\mathcal{O}(\log^3 n)$, given R. We apply Azuma's inequality and get

$$\mathrm{P}[r(X) > \mathrm{E}[r \mid R] + n \mid R] \leq e^{-\frac{n}{\mathcal{O}(\log^6 n)}}.$$

Finally, using $\mathrm{E}[r \mid R] \leq \mathrm{E}[r] + o(n+m) = \mathcal{O}(n+m)$ we obtain $\mathrm{P}[r(X) > \mathcal{O}(n+m)] \leq n^{-c}$ for any positive constant $c > 1$. □

Now we are ready to state the final result for SP-C*. Combining Lemmas 14 to 17 immediately yields:

Theorem 5 *SSSP on arbitrary directed graphs with random edge weights which are independent and uniformly drawn from $[0,1]$ can be solved in $\mathcal{O}(n+m)$ time with high probability.*

5 Extensions

In this section we sketch a few extensions: we consider a larger class of random edge weights, show ways to improve the worst-case performance and investigate a simple parallelization.

5.1 Other Edge Weight Distributions

The uniform distribution used in the previous sections is just a special case of the following much more general class: let us consider independent random values in $[0,1]$ having a common distribution function F with the properties that $F(0) = 0$ and that $F'(0) > 0$ exists. These properties of F imply that the distribution in the neighborhood of 0 can be approximated by a uniform distribution. Hence, our key property – long paths of small total weight are unlikely – is still valid. Lemma 9 can be restated [12] as follows:

There are constants $\alpha \le 1$, $\beta \ge 1$ such that the probability that $\mathcal{P}$ is a Δ-path for $\Delta \le \alpha$ is bounded by $(\beta \cdot \Delta)^l / l!$. Shrinking all bucket widths of SP-C and SP-C* by the constant factor β/α, the average number of re-insertions and re-relaxations still remains linear. The same holds true for the bounds on redistributions and target bucket searches. Therefore, incurring some constant factor loss, both the label-setting and the label-correcting version achieve linear time average-case complexity on the extended edge weight class, too.

5.2 Worst-Case Performance

In the following we describe an alternative for the simple bottom-up search in order to find target buckets in worst-case constant time. Using this guided search routine improves the total worst-case time for SP-S and SP-C (or SP-C*) without automatic fall-back to Dijkstra's algorithm to $\mathcal{O}(\min\{n^2, n \cdot m\})$ and $\mathcal{O}(n \cdot m)$, respectively.

Lemma 18 *The target bucket search for an edge relaxation can be performed in worst-case constant time.*

Proof: For the relaxation of an edge $e = (u, v)$ where u is in the current bucket $B_{\text{cur}} = B_{\text{i,j}}$ with bucket width Δ_{i} of top-level L_i we have to find the highest level L_r, $r \le i$, that can contain the un-split target bucket for v. We distinguish two cases: $c(e) \le \Delta_{\text{i-1}}$ and $c(e) > \Delta_{\text{i-1}}$.

Firstly, if $c(e) \le \Delta_{\text{i-1}}$ (recall that $\Delta_{\text{i-1}}$ is the total level width of L_i) then the target bucket of v is either located in L_i, hence $r = i$, or it is the immediate successor bucket $\vec{B}_{\text{i}}$ of the rightmost bucket in L_i. Note that $\vec{B}_{\text{i}}$ is located in some level below L_i and has width at least $\Delta_{\text{i-1}}$. Upon creation of the new top-level L_i, a pointer to $\vec{B}_{\text{i}}$ can be maintained as follows: if L_i is a refinement of a bucket $B_{\text{i-1,j}}$ which is not the rightmost bucket in L_{i-1}, then $\vec{B}_{\text{i}} = B_{\text{i-1,j+1}}$, otherwise $\vec{B}_{\text{i}} = \vec{B}_{\text{i-1}}$. See Figure 6 for an example.

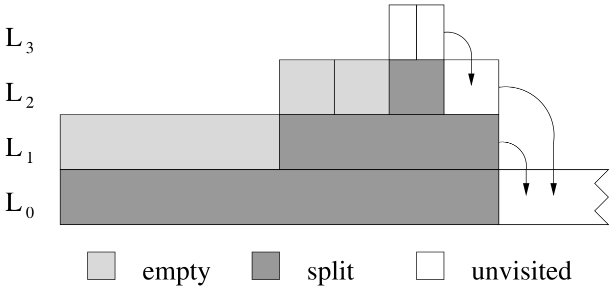

Figure 6: Example for pointers to the immediate successor buckets.

Secondly, consider the case $c(e) > \Delta_{\text{i-1}}$ where the target bucket is definitely located in a level below L_i. Let $x_e = \lfloor \log_2 1/c(e) \rfloor$. For each level L_k, $k \ge 1$, we maintain an additional array $S_k[\cdot]$ with $\log_2 1/\Delta_{k-1}$ entries: $S_i[x_e] = r < i$ denotes the highest level L_r possibly containing the target bucket of v for a relaxation of $e = (u, v)$ with $u \in B_{\text{cur}} = B_{\text{i,j}}$. As before, if v does not belong to L_r, then it will fit into the immediate successor bucket $\vec{B}_{\text{r}}$ of L_r. The correctness follows inductively from the construction:

$S_1[\cdot]$ is an empty array. $S_i[\cdot]$, $i \ge 2$, can be easily created from $S_{i-1}[\cdot]$: the highest possible level for a target bucket of an edge $e = (u, v)$ with $c(e) \ge \Delta_{\text{i-2}}$ will be in a level L_r below L_{i-1}

independent of the position of u in L_{i-1}. Again, this is an immediate consequence of the fact that the total width of L_{i-1} is given by $\Delta_{\text{i-2}} \leq c(e)$. The proper search level L_r can be identified in $\mathcal{O}(1)$ time by looking up $S_{i-1}[\lfloor \log_2 1/c(e) \rfloor] = r$. As L_i is just a refinement of some bucket from L_{i-1} and because for $c(e) > \Delta_{\text{i-2}}$ all targets are located in a level below L_{i-1} we have $S_i[k] = S_{i-1}[k]$ for $0 \leq k < \log_2 1/\Delta_{i-2}$. Hence, these entries can simply be copied from $S_{i-1}[\cdot]$ to $S_i[\cdot]$.

New entries are needed for $S_i[\log_2 1/\Delta_{\text{i-2}}], \ldots, S_i[\log_2 1/\Delta_{\text{i-1}} - 1]$, accounting for edge weights $\Delta_{\text{i-1}} \leq c(e) < \Delta_{\text{i-2}}$. These weights are too large for a target within L_i (which has total level width $\Delta_{\text{i-1}}$) but the target may be located in L_{i-1}, therefore $S_i[\log_2 1/\Delta_{\text{i-2}}] = i-1$, $\ldots$, $S_i[\log_2 1/\Delta_{\text{i-1}} - 1] = i-1$. On the other hand, if the target does not lie in L_{i-1} then the target bucket is definitely given by the immediate successor bucket of L_{i-1}, $\vec{B}_{\text{i-1}}$, because $\vec{B}_{\text{i-1}}$ has width at least $\Delta_{\text{i-2}} > c(e)$.

Hence, all required lookups in the additional data structures for a relaxation of e can be done in constant time. For SP-C and SP-C*, the costs to build $S_i[\cdot]$ and the pointer to $\tilde{B}_{\text{cur}}$ can be charged on the degree of the node v that caused the creation of L_i by a bucket split. For SP-S* the costs are charged on the number of nodes in the split bucket. □

As already mentioned in Section 2.4 for SP-S, both the label-correcting and the label-setting version can keep track of the actual resource usage and start the computation from scratch with Dijkstra's algorithm after $\Theta(n \cdot \log n + m)$ operations, thus limiting the worst-case total execution time to $\mathcal{O}(n \log n + m)$.

5.3 Parallelization

In the following we sketch the result of a simple parallelization for SP-C and SP-C* on the parallel random access machine (PRAM) [22, 34]. This is one of the most widely studied abstract models of a parallel computer. A PRAM consists of p independent processors (processing units, PUs) and a shared memory, which these processors can synchronously access in unit time. We assume the *arbitrary* CRCW (concurrent read concurrent write) PRAM, i.e., in case of conflicting write accesses to the same memory cell, an adversary can choose which access is successful. A fast and efficient parallel algorithm minimizes both *time* and *work* (product of time and number of processors). Ideally, the work bound matches the complexity of the best (known) sequential algorithm.

5.3.1 Previous Work on Parallel SSSP

So far, there is no parallel $\mathcal{O}(n \cdot \log n + m)$ work PRAM SSSP algorithm with sublinear running time for arbitrary digraphs with non-negative edge weights. The $\mathcal{O}(n \cdot \log n + m)$ work solution by Driscoll et. al. [18] (refining a result of Paige and Kruskal [45]) has running time $\mathcal{O}(n \cdot \log n)$. An $\mathcal{O}(n)$ time algorithm requiring $\mathcal{O}(m \cdot \log n)$ work was presented by Brodal et. al. [6]. These algorithms settle the nodes one by one in the order of Dijkstra's algorithm and only perform edge relaxations in parallel. All known faster algorithms require more work, some are tuned for special graph classes, see for example [6, 8, 11, 25, 30, 37, 36, 50, 54].

Parallel SSSP on *random graphs* [5, 20] has been considered by several authors [10, 13, 25, 28, 41, 42, 49]. Under the assumption of independent random edge weights uniformly distributed in the interval $[0, 1]$, the fastest work-efficient label-correcting approach [41, 42] requires $\mathcal{O}(\log^2 n)$ time and linear work on average; furthermore, $\mathcal{O}(d \cdot \mathcal{L} \cdot \log n + \log^2 n)$ time and $\mathcal{O}(n + m + d \cdot \mathcal{L} \cdot \log n)$ work on average is achieved for arbitrary graphs with random edge weights where d denotes the maximum node degree in the graph and $\mathcal{L}$ denotes the maximum weight of a shortest path to a node reachable from s, i.e., $\mathcal{L} = \max_{v \in G, \mathrm{dist}(v) < \infty} \mathrm{dist}(v)$. The algorithm fails if the product $d \cdot \mathcal{L}$ is large.

5.3.2 Improved Results for Moderate Diameter and Maximum Node Degree

Similar to the algorithm of [42] each phase of SP-C and SP-C* can be parallelized by concurrently removing the nodes from the current bucket and relaxing their outgoing edges with several PUs in parallel.

Theorem 6 *Assuming independent random edge weights uniformly distributed in* $[0, 1]$*, parallel SSSP on a CRCW PRAM for graphs with maximum shortest path weight* $\mathcal{L}$ *can be solved in average-case time*

$$T \;=\; \mathcal{O}\left(\min_i \left\{ 2^i \cdot \mathcal{L} + \sum_{v \in G, \mathrm{degree}(v) > 2^i} \mathrm{degree}(v) \right\} \cdot \mathrm{polylog}\, n \right) \qquad (5)$$

using $\mathcal{O}(n + m + T)$ *work.*

Let us first demonstrate the improvement over [42] with an example: consider a graph G with $\mathrm{E}[\mathcal{L}] = \Theta(\sqrt{n})$ where all nodes have degree at most $c = \mathcal{O}(1)$ with the exception of one node that has degree $d = \Theta(\sqrt{n})$, hence $m = \mathcal{O}(n)$. The algorithm of [42] requires $\Omega(d \cdot \mathcal{L}) = \Omega(n)$ average-case time, i.e., no speedup can be obtained, whereas our new approach needs $\mathcal{O}(\sqrt{n} \cdot \mathrm{polylog}\, n)$ time on average using linear work (the minimum in (5) is obtained for $i = \lceil \log_2 c \rceil$). Hence, the existence of a few high-degree nodes will not necessarily slow down the whole computation since those nodes do not impose a *globally* fixed small step width (as opposed to the Δ-stepping of [41, 42]).

We sketch the proof for the parallel time bound (5): Our parallelization mainly follows the concepts presented in [41, 42]. However, we do not spread each bucket over p local buckets (one for each PU) but implement each bucket by a single array with dynamic space adjustment. Concurrent access to the arrays is realized by semi-sorting of the requests as in [42], the processors are scheduled using standard prefix-sum computations. A phase can be performed in polylogarithmic time. As each of the paths followed by the algorithm in a non-split bucket consists of at most $\mathcal{O}(\log n / \log \log n)$ edges whp (Lemma 12), each bucket of the hierarchy is emptied during at most $\mathcal{O}(\log n / \log \log n)$ phases whp. Then the algorithm steps forward to the next nonempty bucket and stops after encountering an empty bucket of L_0. For maximum shortest path weight $\mathcal{L}$ it visits at most $\sum_{j=0}^{i} 2^j \cdot \mathcal{L} = \mathcal{O}(2^i \cdot \mathcal{L})$ buckets in charge of nodes with degree at most 2^i. Each node v with $\mathrm{degree}(v) > 2^i$ can create at most $\mathcal{O}(\mathrm{degree}(v))$ extra buckets if v causes a bucket split. Therefore, altogether there are at most $x = \mathcal{O}(\min_i \{2^i \cdot \mathcal{L} + \sum_{v \in G, \mathrm{degree}(v) > 2^i} \mathrm{degree}(v)\})$ buckets to be visited. Hence, there are at most $\mathcal{O}(x \cdot \log n / \log \log n)$ phases whp, and each of them can be performed in polylogarithmic time.

6 Constructing Difficult Inputs with Random Edge Weights

In this section we demonstrate that random edge weights do not automatically result in good algorithmic performance.

Deterministic worst-case inputs for label-correcting algorithms are usually based on the principle that paths with few edges are found first but longer paths have smaller total weights and hence lead to improvements on the tentative distances. Each such improvement triggers re-insertions which eventually make the computation expensive.

Figure 7 provides an example for the Bellman–Ford algorithm [4, 21] with a FIFO queue: the shortest path between s and q follows the node sequence a, b, c. The algorithm improves $\mathrm{tent}(q)$ for each node on this path, re-inserts q and re-relaxes the outgoing edges of q. In general, let the path from s to q consist of $\Theta(n)$ nodes and let q have out-degree $\Theta(n)$. Then the Bellman–Ford algorithm requires $\Theta(n^2)$ time on this graph class with $m = \mathcal{O}(n)$ edges.

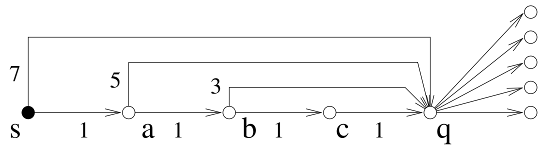

Figure 7: Bad input graph for the Bellman–Ford algorithm.

For random edge weights it is unlikely that a given long path has a small path weight, compare Lemma 9; the expected path weight is linear in the number of edges. In fact, if we replace the fixed edge weights in the above graph class by random edge weights then the expected number of re-insertions for q is constant and therefore the expected time for Bellman-Ford is linear in that case.

6.1 Emulating Fixed Edge Weights

Our main idea for the construction of difficult graphs with random edge weights is to emulate *single edges* e_i with fixed weight by *whole subgraphs* S_i with random edge weights. Each S_i contains exactly one source s_i and one sink t_i. Furthermore, the subgraphs are pairwise edge-disjoint and can only share sources and sinks. Each S_i is built by a chain of $l = \Omega(\log n)$ so-called (u, v, k)-*gadgets*. An (u, v, k)-gadget consists of $k + 2$ nodes u, v, $w_1, \ldots, w_k$ and the $2 \cdot k$ edges (u, w_i) and (w_i, v). The parameter k is called the *blow-up factor* of a gadget. As before, we will assume that random edge weights are independent and uniformly drawn from $[0, 1]$.

Lemma 19 *The expected shortest path weight between u and v in a $(u, v, 1)$-gadget is 1, in a $(u, v, 2)$-gadget it is $23/30$.*

Proof: For $k = 1$, there is only one u-v path in the $(u, v, 1)$-gadget and its expected total weight is clearly $2 \cdot 1/2 = 1$.

For $k = 2$, let X_1 and X_2 be the random variables which denote the weight of the paths $\langle u, w_1, v\rangle$ and $\langle u, w_2, v\rangle$, respectively. The density function for X_i is given by $f(x) = x$, if $0 \leq x \leq 1$, and $f(x) = 2 - x$, $1 < x \leq 2$. The distribution function for the shortest path weight in a $(u, v, 2)$-gadget is

$$W(z) = 1 - \left(1 - \int_0^z f(x)\, dx\right)^2 .$$

The expected value of the shortest path is calculated via the derivative of $W(z)$ as $\int_0^2 x \cdot w(x)\, dx = 23/30$. □

The largest fixed edge weight is emulated by a chain of l gadgets each of which has blow-up factor one. Shorter fixed weights are emulated by choosing higher blow-up factors for some or all of the l gadgets in their respective subgraphs. If we take the parameter l large enough then the actual shortest path weights in S_i will just slightly deviate from their expected values with high probability. In case the gradations between these expected values are much higher than the deviations then the emulated behavior will be as desired.

6.2 Example for Bellman–Ford

Now we provide a concrete conversion example for the Bellman-Ford algorithm with a FIFO queue. As a basis we use the input class of Figure 7. The fixed weight edges entering the node q are replaced by $r = \Theta(n^{1/3-\alpha})$ subgraphs S_i, each of which consists of a chain with $l = \Theta(n^{2/3+\alpha})$ gadgets for some arbitrary constant $0 < \alpha < 1/3$. More specifically, S_i includes $(i \cdot n^{1/3+\alpha})$ $(\cdot, \cdot, 2)$-gadgets, the remaining gadgets have blow-up factor one. Figure 8 shows the principle.

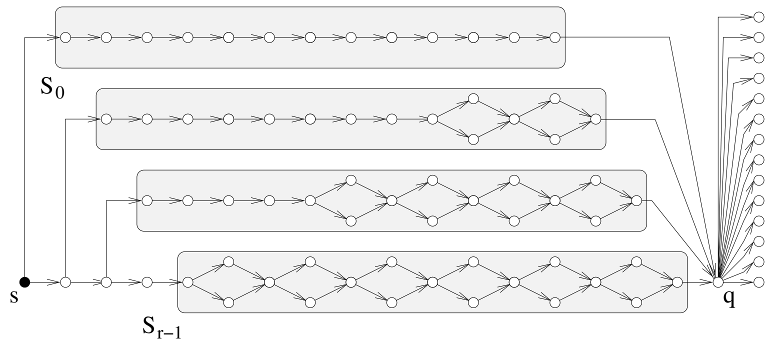

Figure 8: Modified bad input graph with random edge weights.

The shortest path $\mathcal{P}_i$ from s to q via S_i has expected total weight

$$E[w_i] = l - i \cdot 7/30 \cdot n^{1/3+\alpha} + \mathcal{O}(n^{1/3-\alpha}).$$

Using Chernoff bounds [38] it turns out that each w_i deviates from its expected value by at most

$$\mathcal{O}(n^{1/3+\alpha/2} \cdot \log n) = o(n^{1/3+\alpha}) \text{ whp.}$$

Hence, $w_0 > w_1 > \ldots > w_{r-1}$ whp. On the other hand the number of edges on the paths $\mathcal{P}_i$ increases in i. Consequently, the node q is re-inserted r times whp. If q has out-degree $\Theta(n)$ this means Bellman-Ford requires non-linear $\Theta(n^{4/3-\alpha})$ operations on these inputs whp.

Lemma 20 *There are input graphs with n nodes and $m = \mathcal{O}(n)$ edges and random edge weights such that the Bellman-Ford algorithm with a FIFO queue requires $\Theta(n^{4/3-\alpha})$ operations whp for any constant $0 < \alpha < 1/3$.*

The above input class also yields poor performance on other SSSP label-correcting approaches like Pallottino's algorithm [46] which has worst-case execution time $\mathcal{O}(n^2 \cdot m)$ but performs very well on many practical inputs [9, 55]. Pallottino's algorithm maintains two FIFO queues Q_1 and Q_2. The next node to be removed is taken from the head of Q_1 if this queue is not empty and from the head of Q_2 otherwise. A node with improved distance value is added to the tail of Q_1 if it had been queued before, or to the tail of Q_2 otherwise.

Due to the structure of our graph class the only nodes which can ever appear in queue Q_1 are the node q and its immediate successors nodes. Similar to the Bellman-Ford algorithm the FIFO queues in Pallottino's algorithm enforce that q is reached via paths $\mathcal{P}_i$ in order of increasing i. Since $w_0 > w_1 > \ldots > w_{r-1}$ whp, Pallottino's algorithm frequently puts q into Q_1 and thus relaxes the $\Theta(n)$ outgoing edges of q before the computation carries on. Altogether the algorithm incurs $\Theta(n^{4/3-\alpha})$ operations whp.

Conclusions

We have presented SSSP algorithms which require linear time whp on arbitrary directed graphs with random edge weights. To the best of our knowledge these are the first algorithms which provably achieve this time bound. Worst-case time $\Theta(n \log n + m)$ can still be guaranteed by monitoring the actual time usage and switching back to Dijkstra's algorithm if required. The proofs for the label-setting version turned out to be significantly easier, especially for the high probability bound. However, the label-correcting scheme also supports parallelization. Furthermore, we have shown how to construct difficult non-deterministic input graphs for many traditional label-correcting algorithms.

Acknowledgements

The author would like to thank Torben Hagerup and Volker Priebe for valuable discussions and comments. After reading a preliminary version of this paper for the label-correcting approach, Torben Hagerup also suggested to examine a label-setting variant that splits the current bucket according to the number of nodes instead of node degrees.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows : Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] K. B. Athreya. Large deviation rates for branching processes - I, single type case. *Annals of Applied Probability*, 4(3):779–790, 1994.

[3] K. B. Athreya and P. Ney. *Branching Processes*. Springer, 1972.

[4] R. Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[5] B. Bollobás. *Random Graphs*. Academic Press, 1985.

[6] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998.

[7] A. Brodnik, S. Carlson, J. Karlsson, and J. I. Munro. Worst case constant time priority queues. In *Proc. 12th Annual Symposium on Discrete Algorithms*, pages 523–528. ACM–SIAM, 2001.

[8] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. Part II : Optimal parallel algorithms. *Theoretical Computer Science*, 203(2):205–223, 1998.

[9] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest path algorithms: Theory and experimental evaluation. *Math. Programming*, 73:129–174, 1996.

[10] A. Clementi, J. Rolim, and E. Urland. Randomized parallel algorithms. In *Solving Combinatorial Problems in Parallel*, volume 1054 of *LNCS*, pages 25–50, 1996.

[11] E. Cohen. Using selective path-doubling for parallel shortest-path computations. *Journal of Algorithms*, 22(1):30–56, January 1997.

[12] C. Cooper, A. Frieze, K. Mehlhorn, and V. Priebe. Average-case complexity of shortest-paths problems in the vertex-potential model. *Random Structures & Algorithms*, 16:33–46, 2000.

[13] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *23rd Symp. on Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*, pages 722–731. Springer, 1998.

[14] E. V. Denardo and B. L. Fox. Shortest route methods: 1. reaching pruning and buckets. *Operations Research*, 27:161–186, 1979.

[15] R. B. Dial. Algorithm 360: Shortest-path forest with topological ordering. *Communications of the ACM*, 12(11):632–633, 1969.

[16] E. W. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.

[17] E. A. Dinic. Economical algorithms for finding shortest paths in a network. In *Transportation Modeling Systems*, pages 36–44, 1978.

[18] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31, 1988.

[19] D. P. Dubhashi and A. Panconesi. Concentration of measure for the analysis of randomized algorithms. Draft Manuscript, `http://www.brics.dk/~ale/papers.html`, October 1998.

[20] P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci.*, 5(A):17–61, 1960.

[21] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1963.

[22] S. Fortune and J. Wyllie. Parallelism in random access memories. In *Proc. 10th Symp. on the Theory of Computing*, pages 114–118. ACM, 1978.

[23] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[24] A. M. Frieze and G. R. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10:57–77, 1985.

[25] A. M. Frieze and L. Rudolph. A parallel algorithm for all-pairs shortest paths in a random graph. In *Proc. 22nd Allerton Conference on Communication, Control and Computing*, pages 663–670, 1985.

[26] A. V. Goldberg. A simple shortest path algorithm with linear average time. Technical report STAR-TR-01-03, InterTrust Technologies Corporation, Santa Clara, CA, 2001.

[27] A. V. Goldberg and R. E. Tarjan. Expected performance of Dijkstra's shortest path algorithm. Technical Report TR-96-062, NEC Research, 1996.

[28] Q. P. Gu and T. Takaoka. A sharper analysis of a parallel algorithm for the all pairs shortest path problem. *Parallel Computing*, 16(1):61–67, 1990.

[29] T. Hagerup. Improved shortest paths on the word RAM. In *27th Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of *LNCS*, pages 61–72. Springer, 2000.

[30] Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. *Algorithmica*, 17(4):399–415, 1997.

[31] T. Harris. *The Theory of Branching Processes*. Springer, 1963.

[32] R. Hassin and E. Zemel. On shortest paths in graphs with random weights. *Math. Oper. Res.*, 10(4):557–564, 1985.

[33] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1964.

[34] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, 1992.

[35] R. M. Karp and Y. Zhang. Bounded branching process and AND/OR tree evaluation. *Random Structures & Algorithms*, 7(2):97–116, 1995.

[36] P. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, November 1997.

[37] P. N. Klein and S. Subramanian. A linear-processor polylog-time algorithm for shortest paths in planar graphs. In *Proc. 34th Annual Symposium on Foundations of Comptuer Science*, pages 259–270. IEEE, 1993.

[38] C. McDiarmid. Concentration. In Michel Habib, Colin McDiarmid, Jorge Ramirez-Alfonsin, and Bruce Reed, editors, *Probabilistic Methods for Algorithmic Discrete Mathematics*, volume 16 of *Algorithms Combin.*, pages 195–248. Springer, 1998.

[39] K. Mehlhorn and V. Priebe. On the all-pairs shortest-path algorithm of Moffat and Takaoka. *Random Structures & Algorithms*, 10:205–220, 1997.

[40] U. Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proc. 12th Annual Symposium on Discrete Algorithms*, pages 797–806. ACM–SIAM, 2001.

[41] U. Meyer and P. Sanders. Δ-stepping: A parallel shortest path algorithm. In *6th European Symposium on Algorithms (ESA)*, volume 1461 of *LNCS*, pages 393–404. Springer, 1998.

[42] U. Meyer and P. Sanders. Parallel shortest path for arbitrary graphs. In *Proc. Euro-Par 2000 Parallel Processing*, volume 1900 of *LNCS*, pages 461–470. Springer, 2000.

[43] A. Moffat and T. Takaoka. An all pairs shortest path algorithm with expected time $O(n^2 \log n)$. *SIAM Journal on Computing*, 16:1023–1031, 1987.

[44] K. Noshita. A theorem on the expected complexity of Dijkstra's shortest path algorithm. *Journal of Algorithms*, 6:400–408, 1985.

[45] R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In *International Conference on Parallel Processing*, pages 14–20. IEEE Computer Society Press, 1985.

[46] S. Pallottino. Shortest-path methods: Complexity, interrelations and new propositions. *Networks*, 14:257–267, 1984.

[47] R. Raman. Priority queues: Small, monotone and trans-dichotomous. In *4th Annual European Symposium on Algorithms (ESA)*, volume 1136 of *LNCS*, pages 121–137. Springer, 1996.

[48] R. Raman. Recent results on the single-source shortest paths problem. *ACM SIGACT News*, 28(2):81–87, June 1997.

[49] J. Reif and P. Spirakis. Expected parallel time and sequential space complexity of graph and digraph problems. *Algorithmica*, 7:597–630, 1992.

[50] H. Shi and T. H. Spencer. Time–work tradeoffs of the single-source shortest paths problem. *Journal of Algorithms*, 30(1):19–32, 1999.

[51] P. M. Spira. A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$. *SIAM Journal on Computing*, 2:28–32, 1973.

[52] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

[53] M. Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30:86–109, 2000.

[54] J. L. Träff and C. D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. *Journal of Parallel and Distributed Computing*, 60(9):1103–1124, 2000.

[55] F. B. Zhan and C. E. Noon. Shortest path algorithms: An evaluation using real road networks. *Transportation Science*, 32:65–73, 1998.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments
MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA
MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?
MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable “open-world” desing in Java
MPI-I-1999-1-003	P. Sanders, S. Egner, J. Korst	Fast Concurrent Access to Parallel Disks
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid *E*-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-2-002	F. Jacquemard, C. Meyer, C. Weidenbach	Unification in Extensions of Shallow Equational Theories
MPI-I-98-1-031	G.W. Klau, P. Mutzel	Optimal Compaction of Orthogonal Grid Drawings
MPI-I-98-1-030	H. Brönniman, L. Kettner, S. Schirra, R. Veltkamp	Applications of the Generic Programming Paradigm in the Design of CGAL
MPI-I-98-1-029	P. Mutzel, R. Weiskircher	Optimizing Over All Combinatorial Embeddings of a Planar Graph
MPI-I-98-1-028	A. Crauser, K. Mehlhorn, E. Althaus, K. Brengel, T. Buchheit, J. Keller, H. Krone, O. Lambert, R. Schulte, S. Thiel, M. Westphal, R. Wirth	On the performance of LEDA-SM