Design and Analysis of Sequential and Parallel

Single–Source Shortest–Paths Algorithms

Ulrich Meyer

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
Universität des Saarlandes

Saarbrücken, 2002

Datum des Kolloquiums:	21.10.2002
Dekan:	Prof. Dr. Philipp Slusallek
Gutachter:	Prof. Dr. Kurt Mehlhorn
	Prof. Dr. Jop F. Sibeyn

Abstract. We study the performance of algorithms for the **S**ingle-**S**ource **S**hortest-**P**aths (SSSP) problem on graphs with n nodes and m edges with nonnegative random weights. All previously known SSSP algorithms for directed graphs required superlinear time. We give the first SSSP algorithms that provably achieve linear $\mathcal{O}(n+m)$ average-case execution time on arbitrary directed graphs with random edge weights. For independent edge weights, the linear-time bound holds with high probability, too. Additionally, our result implies improved average-case bounds for the **A**ll-**P**airs **S**hortest-**P**aths (APSP) problem on sparse graphs, and it yields the first theoretical average-case analysis for the "**A**pproximate **B**ucket **I**mplementation" of Dijkstra's SSSP algorithm (ABI–Dijkstra). Furthermore, we give constructive proofs for the existence of graph classes with random edge weights on which ABI–Dijkstra and several other well-known SSSP algorithms require superlinear average-case time. Besides the classical sequential (single processor) model of computation we also consider *parallel* computing: we give the currently fastest average-case linear-work parallel SSSP algorithms for large graph classes with random edge weights, e.g., sparse random graphs and graphs modeling the WWW, telephone calls or social networks.

Kurzzusammenfassung. In dieser Arbeit untersuchen wir die Laufzeiten von Algorithmen f¨ur das K¨urzeste-Wege Problem (*Single-Source Shortest-Paths*, SSSP) auf Graphen mit n Knoten, m Kanten und nichtnegativen zuf¨alligen Kantengewichten. Alle bisherigen SSSP Algorithmen ben¨otigten auf gerichteten Graphen superlineare Zeit. Wir stellen den ersten SSSP Algorithmus vor, der auf beliebigen gerichteten Graphen mit zuf¨alligen Kantengewichten eine beweisbar lineare *average-case*-Komplexit¨at $\mathcal{O}(n + m)$ aufweist. Sind die Kantengewichte unabh¨angig, so wird die lineare Zeitschranke auch mit hoher Wahrscheinlichkeit eingehalten. Außerdem impliziert unser Ergebnis verbesserte *average-case*-Schranken f¨ur das *All-Pairs Shortest-Paths* (APSP) Problem auf d¨unnen Graphen und liefert die erste theoretische *average-case*-Analyse f¨ur die *"Approximate Bucket Implementierung"* von Dijkstras SSSP Algorithmus (ABI-Dijkstra). Weiterhin f¨uhren wir konstruktive Existenzbeweise f¨ur Graphklassen mit zuf¨alligen Kantengewichten, auf denen ABI-Dijkstra und mehrere andere bekannte SSSP Algorithmen durchschnittlich superlineare Zeit ben¨otigen. Neben dem klassischen seriellen (Ein-Prozessor) Berechnungsmodell betrachten wir auch Parallelverarbeitung; f¨ur umfangreiche Graphklassen mit zuf¨alligen Kantengewichten wie z.B. d¨unne Zufallsgraphen oder Modelle f¨ur das WWW, Telefonanrufe oder soziale Netzwerke stellen wir die derzeit schnellsten parallelen SSSP Algorithmen mit durchschnittlich linearer Arbeit vor.

Acknowledgements

First of all, I would like to thank my *Doktorvater*, Kurt Mehlhorn, for his constant support, patience and generosity. He provided a very good balance between scientific freedom and scientific guidance. The same holds true for my co-advisor, Jop F. Sibeyn. I could be sure to fall on sympathetic ears with them whenever I wanted to discuss some new ideas.

Some of the material presented in this thesis has grown out of joint work with Andreas Crauser, Kurt Mehlhorn, and Peter Sanders. Working with them was fun and and taught me a lot. I am also indebted to Lars Arge, Paolo Ferragina, Michael Kaufmann, Jop F. Sibeyn, Laura Toma, and Norbert Zeh for sharing their insights and enthusiasm with me while performing joint research on non-SSSP subjects.

The work on this thesis was financially supported through a Graduiertenkolleg graduate fellowship, granted by the Deutsche Forschungsgemeinschaft, and through a Ph. D. position at the Max-Planck Institut f¨ur Informatik at Saarbr¨ucken. I consider it an honor and privilege to have had the possibility to work at such a stimulating place like MPI. The members and guests of the algorithm and complexity group, many of them friends by now, definitely played an important and pleasant role in my work at MPI. Thanks to all of you.

I would also like to acknowledge the hospitality and warm spirit during many short visits and one longer stay with the research group of Lajos Rónyai at the Informatics Laboratory of the Hungarian Academy of Sciences. Furthermore, many other people outside MPI and the scientific circle have helped, taught, encouraged, guided, and advised me in several ways while I was working on this thesis. I wish to express my gratitude to all of them, especially to my parents.

Of all sentences in this thesis, however, none was easier to write than this one: To my wife, Annamária Kovács, who did a great job in checking and enforcing mathematical correctness, and to my little daughter, Emilia, who doesn't care about shortest-paths at all, this thesis is dedicated with love.

Contents

Chapter 1

Introduction

This thesis deals with a basic combinatorial-optimization problem: computing shortest paths on directed graphs with weighted edges. We focus on the single-source shortest-paths (SSSP) version that asks for minimum weight paths from a designated source node of a graph to all other nodes; the weight of a path is given by the sum of the weights of its edges. We consider SSSP algorithms under the classical sequential (single processor) model and for parallel processing, that is, having several processors working in concert.

Computing SSSP on a parallel computer may serve two purposes: solving the problem faster than on a sequential machine and/or taking advantage of the aggregated memory in order to avoid slow external memory computing. Currently, however, parallel and external memory SSSP algorithms still constitute major performance bottlenecks. In contrast, internal memory sequential SSSP for graphs with nonnegative edge weights is quite well understood: numerous SSSP algorithms have been developed, achieving better and better asymptotic worst-case running times. On the other hand, many sequential SSSP algorithms with less attractive worst-case behavior perform very well in practice but there are hardly any theoretical explanations for this phenomenon.

We address these deficits, by providing the first sequential SSSP algorithms that provably achieve optimal performance on the average. Despite intensive research during the last decades, a comparable worst-case result for directed graphs has not yet been obtained. We also prove that a number of previous sequential SSSP algorithms have non-optimal average-case running times. Various extensions are given for parallel computing.

In Section 1.1 of this introduction, we will first motivate shortest-paths problems. Then we compare average-case analysis with worst-case analysis in Section 1.2. Subsequently, we sketch the different models of computation considered in this thesis (Section 1.3). Then, in Section 1.4, we give a first overview of our new results. Finally, Section 1.5 outlines the organization of the rest of this thesis.

1.1 Motivation

Shortest-paths problems are among the most fundamental and also the most commonly encountered graph problems, both in themselves and as subproblems in more complex settings [3]. Besides obvious applications like preparing travel time and distance charts [71], shortest-paths computations are frequently needed in telecommunications and transporta-

tion industries [128], where messages or vehicles must be sent between two geographical locations as quickly or as cheaply as possible. Other examples are complex traffic flow simulations and planning tools [71], which rely on a large number of individual shortest-paths problems. Further applications include many practical integer programming problems. Shortest-paths computations are used as subroutines in solution procedures for computational biology (DNA sequence alignment [143]), VLSI design [31], knapsack packing problems [56], and traveling salesman problems [83] and for many other problems.

A diverse set of shortest-paths models and algorithms have been developed to accommodate these various applications [37]. The most commonly encountered subtypes are: *One-Pair Shortest-Paths* (OPSP), *Single-Source Shortest-Paths* (SSSP), and *All-Pairs Shortest-Paths* (APSP). The OPSP problem asks to find a shortest path from one specified source node to one specified destination node. SSSP requires the computation of a shortest path from one specified source node to every other node in the graph. Finally, the APSP problem is that of finding shortest paths between all pairs of nodes. Frequently, it is not required to compute the set of shortest paths itself but just the distances to the nodes; once the distances are known the paths can be easily derived. Other subtypes deal with modified constraints on the paths, e.g., what is the shortest-path weight from node a to node b through a node c, what is the k-th shortest path from node a to node b, and so on. Further classifications concern the input graph itself. Exploiting known structural properties of the input graphs may result in simpler and/or more efficient algorithms.

Throughout the thesis, we will consider graphs $G = (V, E)$ with $|V| = n$ nodes, $|E| = m$ edges (Section 2.1 provides a short summary of the basic terminology for shortest-path problems on graphs). We will mainly deal with the SSSP problem on directed graphs with nonnegative edge weights. We will restrict ourselves to the computation of the shortest-paths weight labels without reporting the paths themselves; see Figure 1.1 for an example.

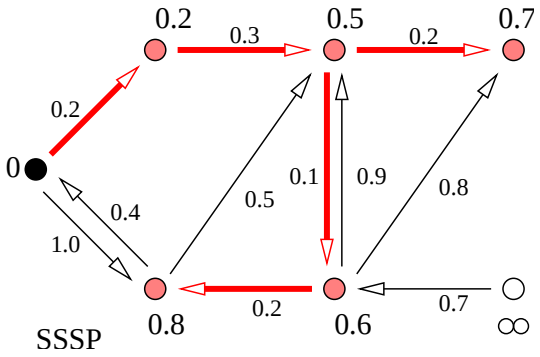


Figure 1.1: Node labels of a solution for SSSP. The source node is marked in black.

1.2 Worst-Case versus Average-Case Analysis

The efficiency of an algorithm for a certain machine model is often stated using *worst-case analysis*: an upper bound on the running time is given that holds true for any input of a certain class. For sequential SSSP, the worst-case running time can be given by a formula $f(n, m)$ depending exclusively on the number of nodes, n, and the number of edges, m.

Additional restrictions on edge weights, graph structures etc. may result in sharper upper bounds.

Shortest-paths algorithms commonly apply iterative labeling methods, of which two major types are *label-setting* and *label-correcting* (we will formally introduce these methods in Section 2.2). Some sequential label-correcting algorithms have polynomial-time upper-bounds, others even require exponential time in the worst case. In either case, the best sequential label-setting algorithms have better worst-case bounds than that of any label-correcting algorithm. Hence, at first glance, label-setting algorithms seem to be the better choice. However, several independent experimental studies [25, 36, 39, 60, 64, 87, 111, 145] showed that SSSP implementations of label-correcting approaches frequently outperform label-setting algorithms. Thus, worst-case analysis sometimes fails to bring out the advantages of algorithms that perform well in practice.

Evaluating the performance of shortest-paths algorithms on the basis of real-world data is both desirable and problematic: clearly, testing and refining an algorithm based on concrete and practically relevant instances (like road maps for SSSP) will be helpful to improve the algorithmic performance on these very instances. However, benchmarks of real-life inputs are usually restricted to some fixed-size instances, thus making it difficult to predict the scalability of an algorithm. Furthermore, the actual running time may crucially depend on structural input properties that may or may not be represented by the benchmarks. Consequently, experimental evaluation frequently relies on synthetic input data that can be generated in varying sizes and that are designed to more or less model real-word data.

Many input generators produce input instances at random according to a certain probability distribution on the set of possible inputs of a certain size. For this input model one can study the *average-case performance*, that is, the *expected* running time of the algorithm averaged according to the applied probability distribution for the input instances (Section 2.4 will supply some basic facts and pointers concerning probability theory). It nearly goes without saying that the choice of the probability distribution on the set of possible instances may crucially influence the resulting average-case bounds. A useful choice establishes a reasonable compromise between being a good model for real-world data (i.e., producing "practically relevant" instances with sufficiently high probability) and still being mathematically analyzable.

Frequently used input models for the *experimental* performance evaluation of SSSP algorithms are random or grid-like graphs with independent random edge weights. The resulting inputs exhibit certain structural properties with high probability (whp)[1], for example concerning the maximum shortest-path weight or connectivity. However, for some of these properties it is doubtful whether they reflect real-life features or should rather be considered as artifacts of the model, which possibly misdirect the quest for practically relevant algorithms.

Mathematical average-case analysis for shortest-paths algorithms has focused on the APSP problem for a simple graph model, namely the complete graph with random edge weights. One of the main contributions of this thesis is a thorough mathematical average-case analysis of sequential SSSP algorithms on *arbitrary* directed graphs with random edge

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

weights.

1.3 Models of Computation

The analysis of an algorithm must take into account the properties and restrictions of the underlying machine model. In this section we sketch the models of computation used in the thesis.

1.3.1 Sequential Computing

The standard "von Neumann" model of computation (see e.g., [2]) assumes some uniform cost for any basic operation like accessing a memory cell, assigning, adding or comparing two values and so on.

There are further model distinctions concerning the set of supported basic operations: advanced algorithms often rely on additional constant time operations provided on the "RAM (**R**andom **A**ccess **M**achine) with word size w" [2]. This model basically reflects what one can use in a programming language such as `C` and what is supported by current hardware. In particular, it allows direct and indirect addressing, bitwise logical operations, arbitrary bit shifts and arithmetic operations on $\mathcal{O}(w)$-bit operands in constant time. Frequently, the values of variables may also take real numbers (standard RAM model without explicit word size).

In contrast to the RAM, there is the pointer-machine model: it disallows memory-address arithmetic. Therefore, bucketing, which is essential to some of our algorithms, is impossible in this model. Also, there is the comparison based model, where weights may only be compared[2].

The algorithms proposed in this thesis do not require any new machine model properties that would not have been used before for other SSSP algorithms as well; the standard RAM model is sufficient. Some of our algorithms even work in the weaker models. By way of contrast, a recent paper of Brodnik et.al. [21] strengthens the machine model in order to obtain a linear-time SSSP algorithm for directed graphs; its priority queue requires a special memory-architecture model with "byte overlap", which is currently not supported by any existing hardware.

1.3.2 Parallel Computing

Large input sizes require algorithms that efficiently support parallel computing, both in order to achieve fast execution and to take advantage of the aggregate memory of the parallel system. The parallel random access machine (PRAM) [52, 72, 88] 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. Most of our parallel algorithms 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. The strict PRAM

[2](However, in order to facilitate any meaningful SSSP computation, at least addition of weights must be allowed.)

model is mainly implemented on experimental parallel machines like the SB-PRAM [51]. Still, it is valuable to highlight the main ideas of a parallel algorithm without tedious details of a particular architecture.

The performance of PRAM algorithms for input graphs with n nodes and m edges is usually described by the two parameters *time* $T^*(n,m)$ (assuming an unlimited number of available PUs) and *work* $W(n,m)$ (the total number of operations needed). Let $C(n,m)$ be the execution time for some fixed sequential SSSP algorithm. The obtainable *Speed-up* $S(n,m,p)$ over this sequential algorithm due to parallelization is bounded by $\mathcal{O}\left(\frac{C(n,m)}{T^*(n,m)+W(n,m)/p}\right)$. Therefore, a fast and efficient parallel algorithm minimizes both $T^*(n,m)$ and $W(n,m)$; ideally $W(n,m)$ is asymptotic to the sequential complexity of the problem. A number of SSSP PRAM algorithms has been invented to fit the needs of parallel computing. Unfortunately, most of them require significantly more work than their sequential counterparts.

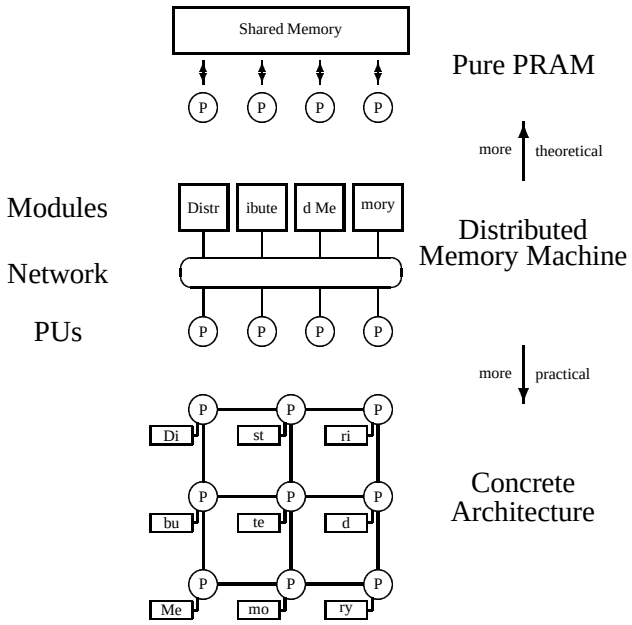


Figure 1.2: Different models of parallel computation.

Other models like BSP [142] and LogP [34] view a parallel computer as a so called *distributed memory machine (DMM)*, i.e., a collection of sequential processors, each one having its own local memory. The PUs are interconnected by a network, which allows them to communicate by sending and receiving messages. Communication constraints are imposed by global parameters like latency, limited network bandwidth and synchronization delays. Clearly, worst-case efficient DMM algorithms are at least as hard to obtain as their PRAM equivalents. Even more detailed models of a parallel computer are obtained by additionally taking into consideration the concrete network architecture used to connect the PUs, thus allowing more fine-grained performance predictions of the message passing procedures. Figure 1.2 depicts the relation between parallel models of computation.

1.4 New Results in a Nutshell

This section provides a very brief overview of our new results. More comprehensive listings including pointers to previous and related work are given in the beginnings of the respective chapters.

1.4.1 Sequential Algorithms (Chapter 3)

For arbitrary *undirected* networks with nonnegative edge costs, it is known that Single-Source Shortest-Paths problems can be solved in linear time in the worst case [136]. It is unknown, however, whether this can also be achieved for *directed* networks. We prove that on average, a similar result indeed holds. Our problem instances are arbitrary directed networks on n nodes and m edges whose edge weights are randomly chosen according to the uniform distribution on $[0, 1]$, independently of each other. We present both label-setting and label-correcting algorithms that solve the SSSP problem on such instances in time $O(n + m)$ on the average. The time bound can also be obtained with high probability.

Only very little is known about the average-case performance of previous SSSP algorithms. Our research yields the first theoretical average-case analysis for the "**A**pproximate **B**ucket **I**mplementation" [25] of Dijkstra's algorithm [42] (ABI–Dijkstra): for random edge weights and either random graphs or graphs with constant maximum node degree we show how the bucket width must be chosen in order to achieve linear $\mathcal{O}(n + m)$ average-case execution time. Furthermore, we give constructive existence proofs for graph classes with random edge weights on which ABI–Dijkstra and several other well-known SSSP algorithms are forced to run in superlinear time on average. While this is interesting in its own right it also stresses the advantages of our new algorithms.

1.4.2 Parallel Algorithms (Chapter 4)

Besides the classical sequential (single processor) model of computation we also consider parallel processing. Unfortunately, for general graphs with nonnegative edge weights, no fast and work-efficient parallel SSSP algorithms are known.

We present new average-case results for a number of important graph classes; for example, we provide the first work-optimal PRAM algorithms that require sublinear average-case time for sparse random graphs, and graphs modeling the WWW, telephone calls or social networks. Most of our algorithms are derived from the new sequential label-correcting approach exploiting the fact that certain operations can be performed independently on different processors or disks. The algorithms are analyzed in terms of quite general graph properties like (expected) diameter, maximum shortest-path weight or node degree sequences. For certain parameter ranges, already very simple extensions provably do the job; other parameters require more involved data structures and algorithms. Sometimes, our methods do not lead to improved algorithms at all, e.g., on graphs with linear diameter. However, such inputs are quite atypical.

Preliminary accounts of the results covered in this thesis have been presented in [33, 103, 104, 105, 106, 107]. The follow-up paper of Goldberg [67] on sequential SSSP was helpful

to streamline some proofs for our sequential label-setting algorithm.

1.5 Organization of the Thesis

The rest of the thesis is composed as follows: Chapter 2 provides the definitions for shortest-paths problems on graphs and reviews the basic solution strategies. In particular, it discusses simple criteria to verify that some tentative distance value is final. These criteria play a crucial role in our SSSP algorithms – independent of the machine model.

Throughout the thesis we will use probabilistic arguments; Chapter 2 also provides a few very basic facts about probability theory and offers references for further reading. Advanced probabilistic methods will be presented when needed.

Subsequently, there is a division into the parts sequential SSSP (Chapter 3) and parallel SSSP (Chapter 4). Each of these chapters starts with an overview of previous work and then presents our contributions. At the end of the chapters we will provide a few concluding remarks and highlights some open problems. Many concepts developed in Chapter 3 will be reused for the parallel algorithms.

Chapter 2

Definitions and Basic Concepts

In this chapter we will provide a short summary of the basic terminology (Section 2.1) and solution approaches (Section 2.2) for shortest-path problems on graphs. Readers familiar with the SSSP problem may choose to skip these sections and refer to them when necessary. In Section 2.3 we present advanced strategies that turn out to be essential for our new algorithms. Finally, Section 2.4 provides some basic facts about probability theory.

2.1 Definitions

A graph $G = (V, E)$ consists of a set V of nodes (or vertices) and a set E of edges (or arcs). We let $|V| = n$ denote the number of nodes in G, $|E| = m$ represents the number of edges.

Directed Graphs

The edge set E of a *directed* graph consists of ordered pairs of nodes: an edge e from node u to node v is denoted by $e = (u, v)$. Here u is also called the *source*, v the *target*, and both nodes are called *endpoints* of (u, v). Furthermore, (u, v) is referred to as one of u's *outgoing* edges or one of v's *incoming* edges, as an edge *leaving* u or an edge *entering* v. The number of edges leaving (entering) a node is called the *out-degree* (*in-degree*) of this node. The *degree* of a node is the sum of its in-degree and out-degree. The *adjacency-list* of a node u consists of all nodes v such that $(u, v) \in E$. Depending on the application, a corresponding member of the adjacency-list of node u may be interpreted as either the target node v or the edge (u, v). The adjacency-list of node u is also often called the *forward star* of u, $FS(u)$ for short.

Undirected Graphs

Undirected graphs are defined in the same manner as directed graphs except that edges are unordered pairs of nodes, i.e., $\{u, v\}$ stands for an undirected edge that connects the nodes u and v. Hence, in undirected graphs an edge can be imagined as a "two-way" connection, whereas in directed graphs it is just "one-way". Consequently, there is no distinction between incoming and outgoing edges. Two undirected edges are adjacent if they share a common endpoint.

Networks

(Un)directed graphs whose edges and/or nodes have associated numerical values (e.g., costs, capacities, etc.) are called *(un)directed networks*. We shall often not formally distinguish between graphs and networks; for example, when we consider the unweighted breadth-first search version of the shortest-path problem. However, our model for shortest-paths problems are usually networks $(V, E, c(\cdot))$ in which a function $c(\cdot)$ assigns independent random costs, *weights*, to the edges of E. The weight of the edge (u, v) is denoted by $c(u, v)$.

Subgraphs

We say that a graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given a set $V' \subseteq V$, the subgraph of G *induced* by V' is the graph $G' = (V', E')$, where $E' = \{(u, v) \in E : u, v \in V'\}$. Furthermore, we shall often define subgraphs based on a threshold c^* on the weights of the edges in E: the subset of edges is given by $E' = \{(u, v) \in E : c(u, v) \leq c^*\}$, and only nodes being either source or target of at least one edge in E' are retained in V'.

Paths and Cycles

A *path* P from u to w in a directed graph G is a node sequence $\langle v_0, v_1, \ldots, v_k \rangle$ for some $k \geq 1$, such that the edges (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k) are part of E, $v_0 = u$, and $v_k = w$. The nodes v_0 and v_k are called the starting point and endpoint of P, respectively. If all nodes v_i on P are pairwise distinct then we say that the path is *simple*. *Cycles* are those paths where the starting point and the endpoint are identical. Paths and cycles of undirected graphs are defined equivalently on undirected edges. A graph is called *acyclic* if it does not contain any cycle. We say that a node v is *reachable* from a node u in G if there is a path $\langle u, \ldots, v \rangle$ in G. An undirected (directed) graph is called (*strongly*) *connected* if it contains a path from u to v for each pair $(u, v) \in V \times V$. The *weight* of a path $P = \langle v_0, \ldots, v_k \rangle$ with edges (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k) in $(V, E, c(\cdot))$ is defined to be $c(P) := \sum_{i=0}^{k-1} c(v_i, v_{i+1})$. In contrast, the *size* of a path denotes the number of edges on the path.

Using the notation above we can formally state the kind of SSSP problems we are interested in:

Definition 1 *Consider a network $(G = (V, E), c)$ with a distinguished vertex s ("source") and a function c assigning a nonnegative real-valued* weight *to each edge of G. The objective of the SSSP is to compute, for each vertex v reachable from s, the minimum weight* $\mathrm{dist}(s, v)$*, abbreviated* $\mathrm{dist}(v)$*, among all paths from s to v. We set* $\mathrm{dist}(s) := 0$*, and* $\mathrm{dist}(s, v) := \infty$ *if v is not reachable from s. We call a path of minimum weight from s to v a* shortest path from s to v.

A valid solution for the SSSP problem implies certain properties for the underlying shortest paths:

Property 1 *If the path $\langle s = v_0, v_1, \ldots, v_{k-1}, v_k \rangle$ is a shortest path from node s to node v_k, then for every i, $1 \leq i \leq k-1$, the sub-path $\langle s = v_0, \ldots, v_i \rangle$ is a shortest path from node s to node v_i.*

Property 2 *A directed path $P = \langle s = v_0, v_1, \ldots, v_{k-1}, v_k \rangle$ from the source node s to node v_k is a shortest path if and only if* $\mathrm{dist}(v_{i+1}) = \mathrm{dist}(v_i) + c(v_i, v_{i+1})$ *for every edge* $(v_i, v_{i+1}) \in P$*. Furthermore, the numbers* $\mathrm{dist}(\cdot)$ *represent proper shortest-paths distances if and only if they satisfy the following* optimality condition*:*

$$\mathrm{dist}(s) = 0 \;\; \textit{and} \;\; \mathrm{dist}(v) = \min_{(u,v)\in E} \mathrm{dist}(u) + c(u,v) \quad \forall v \in V \setminus \{s\}.$$

We need to define two characteristic measures for weighted graphs, which play a crucial role in the analysis of our parallel SSSP algorithms:

Definition 2 *The* maximum shortest-path weight for the source node s *is defined as* $\mathcal{L}(s) := \max\{\mathrm{dist}(s,v) \mid \mathrm{dist}(s,v) < \infty\}$*, abbreviated* $\mathcal{L}$*.*

On graphs with edge weights in $[0, 1]$, the value of $\mathcal{L}$ is bounded from above by the *diameter* $\mathcal{D}$:

Definition 3 *In a graph $G = (V, E)$, let* $\mathrm{minsize}(u, v)$ *denote the minimum number of edges (i.e., the size) needed among all paths from u to v if any, $-\infty$ otherwise; then $\mathcal{D} := \max_{u,v\in V}\{\mathrm{minsize}(u, v), 1\}$ is called the* diameter $\mathcal{D}$ *of G.*

The running times of our parallel SSSP algorithms are explicitly stated in dependence of the value $\mathcal{L}$. Part of our research is concerned with the problem to find good upper bounds on $\mathcal{L}$ for certain graph classes: in Section 3.9.3 we will derive stronger bounds for $\mathcal{L}$ on random graphs by using known results on the diameter of appropriate subgraphs with bounded edge weights.

2.2 Basic Labeling Methods

Shortest-paths algorithms are usually based on iterative *labeling methods*. For each node v in the graph they maintain a tentative distance label $\mathrm{tent}(v)$; $\mathrm{tent}(v)$ is an upper bound on $\mathrm{dist}(v)$. The value of $\mathrm{tent}(v)$ refers to the weight of the lightest path from s to v found so far (if any). Initially, the methods set $\mathrm{tent}(s) := 0$, and $\mathrm{tent}(v) := \infty$ for all other nodes $v \neq s$.

The generic SSSP labeling approach repeatedly selects an arbitrary edge (u, v) where $\mathrm{tent}(u) + c(u, v) < \mathrm{tent}(v)$, and resets $\mathrm{tent}(v) := \mathrm{tent}(u) + c(u, v)$. Identifying such an edge (u, v) can be done with $\mathcal{O}(m)$ operations. The method stops if all edges satisfy

$$\mathrm{tent}(v) \leq \mathrm{tent}(u) + c(u,v) \quad \forall (u,v) \in E. \tag{2.1}$$

By then, $\mathrm{dist}(v) = \mathrm{tent}(v)$ for all nodes v; compare Property 2. If $\mathrm{dist}(v) = \mathrm{tent}(v)$ then the label $\mathrm{tent}(v)$ is said to be *permanent (or final)*; the node v is said to be *settled* in that case.

The total number of operations needed until the labeling approach terminates depends on the order of edge selections; in the worst case it is pseudo-polynomial for integer weights, $\Theta(n^2 \cdot m \cdot C)$ and $\Theta(m \cdot 2^n)$ otherwise [3]. Therefore, improved labeling algorithms perform the selection in a more structured way: *they select nodes rather than edges*. In order to do so they keep a *candidate node set* Q of "promising" nodes. We require Q to contain the starting node u of any edge (u, v) that violates the optimality condition (2.1):

Requirement 1 $Q \supseteq \{u \in V : \exists (u,v) \in E \textit{ with } \mathrm{tent}(u) + c(u,v) < \mathrm{tent}(v)\}$

The labeling methods based on a candidate set Q of nodes repeatedly select a node $u \in Q$ and apply the SCAN operation (Figure 2.1) to it until Q finally becomes empty. Figure 2.2 depicts the resulting generic SSSP algorithm.

```
procedure SCAN(u)
  Q := Q \ {u}
  for all (u,v) ∈ E do
    if tent(u) + c(u,v) < tent(v) then
      tent(v) := tent(u) + c(u,v)
      if v ∉ Q then
        Q := Q ∪ {v}
```

Figure 2.1: Pseudo code for the SCAN operation.

```
algorithm GENERIC SSSP
  for all v ∈ V do
    tent(v) := ∞
  tent(s) := 0
  Q := {s}
  while Q ≠ ∅ do
    select a node u ∈ Q
    SCAN(u)
```

Figure 2.2: Pseudo code for the generic SSSP algorithm with a candidate node set.

SCAN(u) applied to a node $u \in Q$ first removes u from Q and then *relaxes* all[1] outgoing edges of u, that is, the procedure sets

$$\mathrm{tent}(v_i) := \min\{\mathrm{tent}(v_i), \mathrm{tent}(u) + c(u,v_i)\} \quad \forall (u,v_i) \in FS(u).$$

If the relaxation of an edge (u,v_i) reduces $\mathrm{tent}(v_i)$ where $v_i \notin Q$, then v_i is also inserted into Q.

During the execution of the labeling approaches with a node candidate set Q the nodes may be in different *states*: a node v never inserted into Q so far (i.e., $\mathrm{tent}(v) = \infty$) is said to be *unreached*, whereas it is said to be a *candidate* (or *labeled* or *queued*) while it belongs to Q. A node v selected and removed from Q whose outgoing edges have been relaxed is said to be *scanned* as long as v remains outside of Q.

The pseudo code of the SCAN procedure (Figure 2.1) immediately implies the following:

Observation 1 *For any node* v, $\mathrm{tent}(v)$ *never increases during the labeling process.*

[1]There are a few algorithms that deviate from this scheme in that they sometimes only consider a subset of the outgoing edges. We shall use such strategies later on as well.

Furthermore, the SCAN procedure maintains Q as desired:

Lemma 1 *The* SCAN *operation ensures Requirement 1 for the candidate set Q.*

Proof: Q is required to contain the starting node u of any edge $e = (u, v)$ that violates the optimality condition (2.1). Due to Observation 1, the optimality condition (2.1) can only become violated when $\mathrm{tent}(u)$ is decreased, which, in turn, can only happen if some edge (w, u) into u is relaxed during a $\mathrm{SCAN}(w)$ operation. However, if $\mathrm{SCAN}(w)$ reduces $\mathrm{tent}(u)$ then the procedure also makes sure to insert u into Q in case u is not yet contained in it. Furthermore, at the moment when u is removed from Q the condition (2.1) will not become violated because of the relaxation of $FS(u)$.

□

In the following we state the so-called monotonicity property. It proves helpful to obtain better data structures for maintaining the candidate set Q.

Lemma 2 (Monotonicity) *For nonnegative edge weights, the smallest tentative distance among all nodes in Q never decreases during the labeling process.*

Proof: Let $M(Q) := \min\{\mathrm{tent}(v) : v \in Q\}$. $M(Q)$ will not decrease if a node is removed from Q. The tentative distance of a node v can only decrease due to a $\mathrm{SCAN}(u)$ operation where $u \in Q$, $(u, v) \in E$, and $\mathrm{tent}(u) + c(u, v) < \mathrm{tent}(v)$; if v is not queued at this time, then a reduction of $\mathrm{tent}(v)$ will result in v entering Q. However, as all edge weights are nonnegative, $\mathrm{SCAN}(u)$ updates $\mathrm{tent}(v) := \mathrm{tent}(u) + c(u, v) \geq M(Q)$. □

So far, we have not specified how the labeling methods select the next node to be scanned. The labeling methods can be subdivided into two major classes: *label-setting* approaches and *label-correcting* approaches. Label-setting methods exclusively select nodes v with final distance value, i.e., $\mathrm{tent}(v) = \mathrm{dist}(v)$. Label-correcting methods may select nodes with non-final tentative distances, as well.

In the following we show that whenever Q is nonempty then it always contains a node v with final distance value. In other words, there is always a proper choice for the next node to be scanned according to the label-setting paradigm.

Lemma 3 (Existence of an optimal choice.) *Assume $c(e) \geq 0$ for all $e \in E$.*
(a) After a node v is scanned with $\mathrm{tent}(v) = \mathrm{dist}(v)$, it is never added to Q again.
(b) For any node v reachable from the source node s with $\mathrm{tent}(v) > \mathrm{dist}(v)$ there is a node $u \in Q$ with $\mathrm{tent}(u) = \mathrm{dist}(u)$, where u lies on a shortest path from s to v.

Proof: (a) The labeling method ensures $\mathrm{tent}(v) \geq \mathrm{dist}(v)$ at any time. Also, when v is added to Q, its tentative distance value $\mathrm{tent}(v)$ has just been decreased. Thus, if a node v is scanned from Q with $\mathrm{tent}(v) = \mathrm{dist}(v)$, it will never be added to Q again later.
(b) Let $\langle s = v_0, v_1, \ldots v_k = v\rangle$ be a shortest path from s to v. Then $\mathrm{tent}(s) = \mathrm{dist}(s) = 0$ and $\mathrm{tent}(v_k) > \mathrm{dist}(v_k)$. Let i be minimal such that $\mathrm{tent}(v_i) > \mathrm{dist}(v_i)$. Then $i > 0$, $\mathrm{tent}(v_{i-1}) = \mathrm{dist}(v_{i-1})$ and

$$\mathrm{tent}(v_i) > \mathrm{dist}(v_i) = \mathrm{dist}(v_{i-1}) + c(v_{i-1}, v_i) = \mathrm{tent}(v_{i-1}) + c(v_{i-1}, v_i).$$

Thus, by Requirement 1 for the candidate set, the node v_{i-1} is contained in Q. □

The basic label-setting approach for nonnegative edge weights is Dijkstra's method [42]; it selects a candidate node with minimum tentative distance as the next node to be scanned. Figure 2.3 demonstrates an iteration of Dijkstra's method.

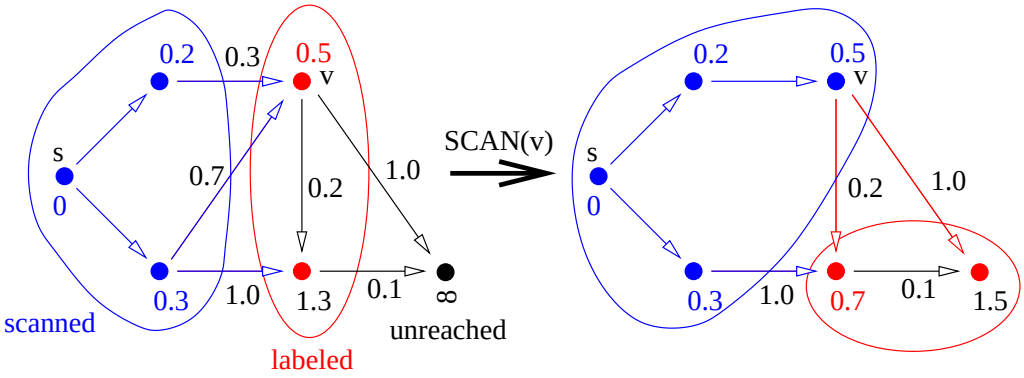


Figure 2.3: SCAN step in Dijkstra's algorithm.

The following lemma shows that Dijkstra's selection method indeed implements the label-setting paradigm.

Lemma 4 (Dijkstra's selection rule) *If* $c(e) \geq 0$ *for all* $e \in E$ *then* $\text{tent}(v) = \text{dist}(v)$ *for any node* $v \in Q$ *with minimal* $\text{tent}(v)$.

Proof: Assume otherwise, i.e., $\text{tent}(v) > \text{dist}(v)$ for some node $v \in Q$ with minimal tentative distance. By Lemma 3, there is a node $z \in Q$ lying on a shortest path from s to v with $\text{tent}(z) = \text{dist}(z)$. Due to nonnegative edge weights we have $\text{dist}(z) \leq \text{dist}(v)$. However, that implies $\text{tent}(z) < \text{tent}(v)$, a contradiction to the choice of v. □

Hence, by Lemma 3, label-setting algorithms have a bounded number of iterations (proportional to the number of reachable nodes), but the amount of time required by each iteration depends on the data structures used to implement the selection rule. For example, in the case of Dijkstra's method the data structure must support efficient minimum and decrease_key operations.

Label-correcting algorithms may have to rescan some nodes several times until their distance labels eventually become permanent; Figure 2.4 depicts the difference to label-setting approaches. Label-correcting algorithms may vary largely in the number of iterations needed to complete the computation. However, their selection rules are often very simple; they frequently allow implementations where each selection runs in constant time. For example, the Bellman-Ford method [15, 50] processes the candidate set Q in simple FIFO (*First-In First-Out*) order.

Our new SSSP algorithms either follow the strict label-setting paradigm or they apply label-correcting with clearly defined intermediate phases of label-setting steps.

2.3 Advanced Label-Setting Methods

In this section we deal with a crucial problem for label-setting SSSP approaches: identifying candidate nodes that have already reached their final distance values. Actually, several

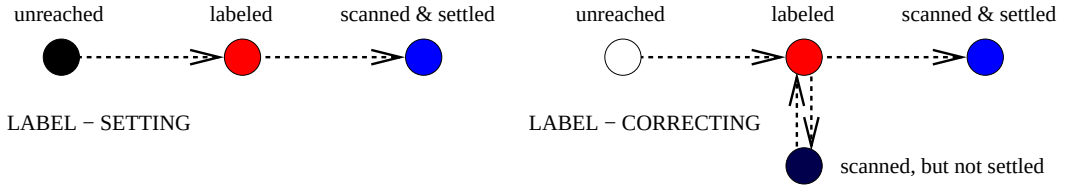


Figure 2.4: States for a reachable non-source node using a label-setting approach (left) and label-correcting approach (right).

mutually dependent problems have to be solved: first of all, a *detection criterion* is needed in order to deduce that a tentative distance of a node is final. Secondly, the data structure that maintains the set Q of candidate nodes must support efficient *evaluation* of the criterion and *manipulation* of the candidate set, e.g. inserting and removing nodes. For each iteration of the labeling process, the criterion must identify at least one node with final distance value; according to Lemma 3 such a node always exists. However, the criterion may even detect a whole subset $R \subseteq Q$ of candidate nodes each of which could be selected. We call R the *yield* of a criterion.

Large yields may be advantageous in several ways: being allowed to select an arbitrary node out of a big subset R could simplify the data structures needed to maintain the candidate set. Even more obviously, large yields facilitate concurrent node scans in parallel SSSP algorithms, thus reducing the parallel execution time. On the other hand, it is likely that striving for larger yields will make the detection criteria more complicated. This may result in higher evaluation times. Furthermore, there are graphs where at each iteration of the labeling process the tentative distance of only one single node in Q is final - even if Q contains many nodes; see Figure 2.5.

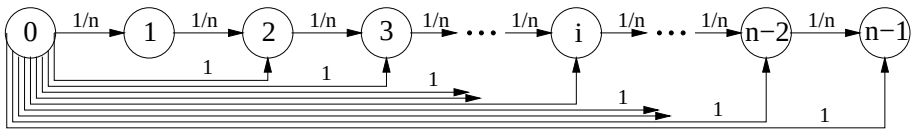


Figure 2.5: Input graph for which the candidate set Q contains only one entry with final distance value: after settling nodes 0, 1, ..., $i-1$ the queue holds node i with (actual) distance i/n, and all other $n-i-1$ queued nodes have tentative distance 1.

In the following we will present the label-setting criteria used in this thesis. We have already seen in Lemma 4 that Dijkstra's criterion [42], i.e., selecting a labeled node with minimum tentative distance as the next node to be scanned, in fact implements the label-setting paradigm. However, it also implicitly sorts the nodes according to their final distances. This is more than the SSSP problem asks for. Therefore, subsequent approaches have been designed to avoid the sorting complexity; they identified label-setting criteria that allow to scan nodes in non-sorted order. Dinitz [43] and Denardo and Fox [36] observed the following:

Criterion 1 (GLOB-criterion) *Let* $M \leq \min\{\mathrm{tent}(v) : v \in Q\}$. *Furthermore, let* $\lambda := \min\{c(e) : e \in E\}$. *Then* $\mathrm{tent}(v) = \mathrm{dist}(v)$ *for any node* $v \in Q$ *having* $\mathrm{tent}(v) \leq M + \lambda$.

The GLOB-criterion of Dinitz and Denardo/Fox is *global* in the sense that it applies uniformly to any labeled node whose tentative distance is at most $M + \lambda$. However, if $\lambda = c(u', v')$ happens to be small[2], then the criterion is restrictive for all nodes $v \in V$, even though its restriction may just be needed as long as the nodes u' and v' are part of the candidate set. Therefore, it is more promising to apply *local* criteria: for each node $v \in Q$ they only take into account a subset of the edge weights, e.g. the weights of the incoming edges of v. The following criterion is frequently used in our algorithms:

Lemma 5 (IN-criterion) *Let $M \leq \min\{\mathrm{tent}(v) : v \in Q\}$. For nonnegative edge weights, the labeled nodes of the following sets have reached their final distance values:*

$\mathcal{U}_1 := \{v \in Q \mid \forall u \in Q : \mathrm{tent}(v) \leq \mathrm{tent}(u)\}$

$\mathcal{U}_2 := \{v \in Q \mid \forall (u, v) \in E : \mathrm{tent}(v) \leq M + c(u, v)\}$

Proof: The claim for the set $\mathcal{U}_1$ was established in Lemma 4 of Section 2.2. The proof for the set $\mathcal{U}_2$ follows the same ideas; assume $\mathrm{tent}(v) > \mathrm{dist}(v)$ for some node $v \in \mathcal{U}_2$. By Lemma 3 there is a node $z \in Q$, $z \neq v$, lying on a shortest path from s to v with $\mathrm{tent}(z) = \mathrm{dist}(z) \geq M$. However, since all edges into v have weight at least $\mathrm{tent}(v) - M$, this implies $\mathrm{dist}(v) \geq \mathrm{dist}(z) + \mathrm{tent}(v) - M \geq \mathrm{tent}(v)$, a contradiction. □

Lemma 5 was already implicit in [36, 43]. However, for a long time, the IN-criterion has not been exploited in its full strength to derive better SSSP algorithms. Only recently, Thorup [136] used it to yield the first linear-time algorithm for undirected graphs. Our algorithms for directed graphs also rely on it. Furthermore, it is used in the latest SSSP algorithm of Goldberg [67], as well.

The IN-criterion for node v is concerned with the incoming edges of v. In previous work [33], we also identified an alternative version, the *OUT-criterion*.

Lemma 6 (OUT-criterion) *Let*

$$\begin{aligned} M &\leq \min\{\mathrm{tent}(v) : v \in Q\} \leq M', \\ \mathcal{U} &:= \{v \in Q : \mathrm{tent}(v) \leq M'\}, \\ \gamma(\mathcal{U}) &:= \min\{+\infty, \{\min\{c(v, w) : v \in \mathcal{U}, (v, w) \in E\}\}. \end{aligned}$$

If $\gamma(\mathcal{U}) \geq M' - M$ and $c(e) \geq 0$ for all $e \in E$ then $\mathrm{tent}(v) = \mathrm{dist}(v)$ for all $v \in \mathcal{U}$.

Proof: In a similar way to the proof of Lemma 5, we assume $\mathrm{tent}(v) > \mathrm{dist}(v)$ for some node $v \in \mathcal{U}$. Again, by Lemma 3, there will be a node $z \in Q$, $z \neq v$, lying on a shortest path from s to v with $\mathrm{tent}(z) = \mathrm{dist}(z) \geq M$. Due to nonnegative edge weights, $\mathrm{dist}(z) \leq \mathrm{tent}(v) \leq M'$. Consequently, $z \in \mathcal{U}$. However, this implies $\mathrm{dist}(v) \geq \mathrm{dist}(z) + \gamma(\mathcal{U})$. If $\gamma(\mathcal{U}) \geq M' - M$ then $\mathrm{dist}(v) \geq M'$, a contradiction. □

Applications of the OUT-criterion are not only limited to SSSP; it is also used for conservative parallel event-simulation [99].

So far we have provided some criteria to detect nodes with final distance values. In the following chapters, we will identify data structures that efficiently support the IN-criterion.

[2]This is very likely in the case of independent random edge weights uniformly distributed in $[0, 1]$ as assumed for our average-case analyses.

2.4 Basic Probability Theory

In this section we review a few basic definitions and facts for the probabilistic analysis of algorithms. More comprehensive presentations are given in [48, 49, 74, 112, 129].

Probability

For the average-case analyses of our SSSP algorithms we assume that the input graphs of a certain size are generated according to some probability distribution on the set of all possible inputs of that size, the so-called *sample space* Ω. A subset $\mathcal{E} \subseteq \Omega$ is called an *event*. A *probability measure* $\mathbf{P}$ is a function that satisfies the following three conditions: $0 \leq \mathbf{P}[\mathcal{E}] \leq 1$ for each $\mathcal{E} \subseteq \Omega$, $\mathbf{P}[\Omega] = 1$, and $\mathbf{P}[\cup_i \mathcal{E}_i] = \sum_i \mathbf{P}[\mathcal{E}_i]$ for pairwise disjoint events $\mathcal{E}_i$. A sample space together with its probability measure build a *probability space*.

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

The *conditional probability* $\mathbf{P}[\mathcal{E}_1 | \mathcal{E}_2] = \frac{\mathbf{P}[\mathcal{E}_1 \cap \mathcal{E}_2]}{\mathbf{P}[\mathcal{E}_2]}$ refers to the probability that an experiment has an outcome in the set $\mathcal{E}_1$ when we already know that it is in the set $\mathcal{E}_2$. Two events $\mathcal{E}_1$ and $\mathcal{E}_2$ are called *independent* if $\mathbf{P}[\mathcal{E}_1 | \mathcal{E}_2] = \mathbf{P}[\mathcal{E}_1]$.

Boole's inequality often proves helpful for dependent events: let $\mathcal{E}_1, \ldots \mathcal{E}_n$ be any collection of events, then $\mathbf{P}\left[\cup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \mathbf{P}\left[\mathcal{E}_i\right]$.

Random Variables

Any real valued numerical function $X = X(\omega)$ defined on a sample space Ω may be called a *random variable*. If X maps elements in Ω to $\mathbb{R}_+ \cup \{0\}$ then it is called a *nonnegative* random variable. A *discrete* random variable is supposed to take only isolated values with nonzero probability. Typical representatives for *discrete* random variables are *binary* random variables, which map elements in Ω to $\{0, 1\}$. For any random variable X and any real number $x \in \mathbb{R}$ we define $\left[X \begin{matrix} = \\ \leq \\ \geq \end{matrix} x\right] = \left\{\omega \in \Omega : X(\omega) \begin{matrix} = \\ \leq \\ \geq \end{matrix} x\right\}$. The *distribution function* F_X of a random variable X is given by $F_X(x) = \mathbf{P}[X \leq x]$. A *continuous* random variable X is one for which $F_X(x)$ can be expressed as $F_X(x) = \int_{-\infty}^{x} p_X(x)\, dx$ where $p_X(x)$ is the so-called *density function*. For example if a random variable X is *uniformly distributed in* $[0, 1]$ (that is the case for the edge weights in our shortest-path problems) then $p_X(x) = 1$ for $0 \leq x \leq 1$ and $p_X(x) = 0$ otherwise.

For any two random variables X and Y, X is said to *(stochastically) dominate* Y (denoted by $X \succeq Y$) if $\mathbf{P}[X \geq k] \geq \mathbf{P}[Y \geq k]$ for all $k \in \mathbb{R}$. Two random variables X and Y are called *independent* if, for all $x, y \in \mathbb{R}$, $\mathbf{P}[X = x \mid Y = y] = \mathbf{P}[X = x]$.

Expectation

The *expectation* of a discrete random variable X is given by $\mathbf{E}[X] = \sum_{x \in \mathbb{R}} x \cdot \mathbf{P}[X = x]$. For a continuous random variable X we have $\mathbf{E}[X] = \int_{-\infty}^{\infty} x \cdot p_X(x)\, dx$. Here are a few important properties of the expectation for arbitrary random variables X and Y:

- If X is nonnegative, then $\mathbf{E}[X] \geq 0$.

- $|\,\mathbf{E}[X]\,| \leq \mathbf{E}[\,|X|\,]$.
- $\mathbf{E}[c \cdot X] = c \cdot \mathbf{E}[X|$ for any $c \in \mathbb{R}$.
- $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$ *(linearity of expectation)*.
- If X and Y are independent, then $\mathbf{E}[X \cdot Y] = \mathbf{E}[X] \cdot \mathbf{E}[Y]$

The *conditional expectation* of a random variable X with respect to an event $\mathcal{E}$ is defined by $\mathbf{E}[X|\mathcal{E}] = \sum_{x \in \mathbb{R}} x \cdot \mathbf{P}[X = x \mid \mathcal{E}]$. An important property of the conditional expectation is that $\mathbf{E}[\,\mathbf{E}[Y|X]\,] = \mathbf{E}[Y]$ for any two random variables X and Y.

Tail Estimates

Frequently, we are interested in the probability that random variables do not deviate too much from their expected values. The *Markov Inequality* for an arbitrary nonnegative random variable X states that $\mathbf{P}[X \geq k] \leq \frac{\mathbf{E}[X]}{k}$ for any $k > 0$. More powerful tail estimates exist for the sum of independent random variables. Here is one version of the well-known *Chernoff bound*:

Lemma 7 (Chernoff bound [26, 77]) *Let $X_1, \ldots, X_k$ be independent binary random variables. Let $\mu = \mathbf{E}[\sum_{j=1}^{k} X_j]$. Then it holds for all $\delta > 0$ that*

$$\mathbf{P}\left[\sum_{j=1}^{k} X_j \geq (1+\delta) \cdot \mu\right] \leq e^{-\min\{\delta^2, \delta\} \cdot \mu/3}. \tag{2.2}$$

Furthermore, it holds for all $0 < \delta < 1$ that

$$\mathbf{P}\left[\sum_{j=1}^{k} X_j \leq (1-\delta) \cdot \mu\right] \leq e^{-\delta^2 \cdot \mu/2}. \tag{2.3}$$

We shall introduce further bounds in the subsequent chapters whenever the need arises. More material on tail estimates can be found for example in [41, 77, 100, 110, 127].

Chapter 3

Sequential SSSP Algorithms

This chapter deals with single-source shortest-paths algorithms for the sequential model of computation. However, many of the concepts presented in here will be reused for the parallel and external-memory SSSP algorithms, as well.

The chapter is structured as follows: first of all, Section 3.1 sketches previous and related work for the sequential machine model. An overview of our new contributions is given in Section 3.2. In Section 3.3 we review some simple bucket-based SSSP algorithms. Then we present our new algorithms SP-S and SP-C (Section 3.4). Both algorithms run in linear time on average. Even though the two approaches are very similar, the analysis of SP-S (Section 3.5) is significantly simpler than that of SP-C (Sections 3.6 and 3.8); in poetic justice, SP-C is better suited for parallelizations (Chapter 4). Furthermore, once analysis of SP-C is established, it easily implies bounds for other label-correcting algorithms (Section 3.9). In Section 3.10, we demonstrate a general method to construct graphs with random edge weights, that cause superlinear average-case running-times with many traditional label-correcting algorithms. Finally, Section 3.11 provides some concluding remarks and presents a few open problems.

3.1 Previous and Related Work

In the following we will list some previous shortest-paths results that are related to our research. Naturally, due to the importance of shortest-paths problems and the intensive research on them, our list cannot (and is not intended to) provide a survey of the whole field. Appropriate overview papers for classical and recent sequential shortest-paths results are, e.g., [25, 37, 114, 124, 136, 146].

3.1.1 Sequential Label-Setting Algorithms

A large fraction of previous work is based on Dijkstra's method [42], which we have sketched in Section 2.2. The original implementation identifies the next node to scan by linear search. For graphs with n nodes, m edges and nonnegative edge weights, it runs in $\mathcal{O}(n^2)$ time, which is optimal for fully dense networks. On sparse graphs, it is more efficient to use a priority queue that supports extracting a node with smallest tentative distance and reducing tentative distances for arbitrary queued nodes. After Dijkstra's result, most subse-

quent theoretical developments in SSSP for general graphs have focused on improving the performance of the priority queue: Applying William's heap [144] results in a running time of $\mathcal{O}(m \cdot \log n)$. Taking Fibonacci heaps [53] or similar data structures [19, 44, 135], Dijkstra's algorithm can be implemented to run in $\mathcal{O}(n \cdot \log n + m)$ time. In fact, if one sticks to Dijkstra's method, thus considering the nodes in non-decreasing order of distances, then $\mathcal{O}(n \cdot \log n + m)$ is even the best possible time bound for the comparison-addition model: any $o(n \cdot \log n)$-time algorithm would contradict the $\Omega(n \cdot \log n)$-time lower-bound for comparison-based sorting.

A number of faster algorithms have been developed for the more powerful RAM model. Nearly all of these algorithms are still closely related to Dijkstra's algorithm; they mainly strive for an improved priority queue data structure using the additional features of the RAM model (see [124, 136] for an overview). Fredman and Willard first achieved $\mathcal{O}(m \cdot \sqrt{\log n})$ expected time with randomized fusion trees [54]; the result holds for arbitrary graphs with arbitrary nonnegative edge weights. Later they obtained the deterministic $\mathcal{O}(m + n \cdot \log n / \log\log n)$-time bound by using atomic heaps [55].

The ultimate goal of a worst-case linear time SSSP algorithm has been partially reached: Thorup [136, 137] gave the first $\mathcal{O}(n+m)$-time RAM algorithm for *undirected* graphs with nonnegative floating-point or integer weights fitting into words of length w. His approach applies label-setting, too, but deviates significantly 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*. Unfortunately, Thorup's algorithm requires the atomic heaps [55] mentioned above, which are only defined for $n \geq 2^{12^{20}}$. Hagerup [76] generalized Thorup's approach to directed graphs. The time complexity, however, becomes superlinear $\mathcal{O}(n + m \cdot \log w)$. The currently fastest RAM algorithm for sparse directed graphs is due to Thorup [138]; it needs $\mathcal{O}(n + m \cdot \log\log n)$ time. Alternative approaches for somewhat denser graphs have been proposed by Raman [123, 124]: 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. Using an adaptation of Thorup's component tree approach, Pettie and Ramanchandran [119] recently obtained improved SSSP algorithms for the pointer machine model. Still, the worst-case complexity for SSSP on sparse directed graphs remains superlinear.

In Section 3.3 we will review some basic implementations of Dijkstra's algorithm with *bucket* based priority queues [3, 36, 38, 43]. Alternative bucket approaches include nested (multiple levels) buckets and/or buckets of different widths [4, 36]. So far, the best bound for SSSP on arbitrary *directed* graphs with nonnegative integer edge-weights in $\{1, \ldots, C\}$ is $\mathcal{O}(m + n \cdot (\log C)^{1/4+\epsilon})$ expected time for any fixed $\epsilon > 0$ [124].

3.1.2 Sequential Label-Correcting Algorithms

The classic label-correcting SSSP approach is the Bellman–Ford algorithm [15, 50]. It implements the set of candidate nodes Q as a FIFO-Queue and achieves running time $\mathcal{O}(n\cdot m)$. There are many more ways to maintain Q and select nodes from it (see [25, 59] for an overview). For example, the algorithms of Pallottino [117], Goldberg and Radzik [69], and Glover et al. [64, 65, 66] subdivide Q into two sets Q_1 and Q_2 each of which is implemented as a list. Intuitively, Q_1 represents the "more promising" candidate nodes. The

algorithms always scan nodes from Q_1. According to varying rules, Q_1 is frequently refilled with nodes from Q_2. These approaches terminate in worst-case polynomial time. However, none of the alternative label-correcting algorithms succeeded to asymptotically improve on the $\mathcal{O}(n \cdot m)$-time worst-case bound of the simple Bellman-Ford approach. Still, a number of experimental studies [25, 36, 39, 60, 64, 87, 111, 145] showed that some recent label-correcting approaches run considerably faster than the original Bellman–Ford algorithm and even outperform label-setting algorithms on certain data sets. So far, no profound average-case analysis has been given to explain the observed effects. A striking example in this respect is the shortest-paths algorithm of Pape [118]; in spite of exponential worst-case time it performs very well on real-world data like road graphs.

We would like to note that faster sequential SSSP algorithms exist for special graph classes with arbitrary nonnegative edge weights, e.g., there is a linear-time approach for planar graphs [81]. The algorithm uses graph decompositions based on separators that may have size up to $\mathcal{O}(n^{1-\epsilon})$. Hence, it may in principle be applied to a much broader class of graphs than planar graphs if just a suitable decomposition can be found in linear time. The algorithm does not require the bit manipulating features of the RAM model and works for directed graphs, thus it remains appealing even after Thorup's linear-time RAM algorithm for arbitrary undirected graphs. Another example for a "well-behaved" input class are graphs with constant tree width; they allow for a linear-time SSSP algorithm as well [24].

3.1.3 Random Edge Weights

Average-case analysis of shortest-paths algorithms mainly focused on the *All-Pairs Shortest-Paths* (APSP) problem on either the complete graph or random graphs with $\Omega(n \cdot \log n)$ edges and random edge weights [32, 57, 80, 109, 133]. Average-case running times of $\mathcal{O}(n^2 \cdot \log n)$ as compared to worst-case cubic bounds are obtained by virtue of an initial pruning step: if $\mathcal{L}$ denotes a bound on the maximum shortest-path weight, then the algorithms discard *insignificant edges* of weight larger than $\mathcal{L}$; they will not be part of the final solution. Subsequently, the APSP is solved on the reduced graph. For the inputs considered above, the reduced graph contains $\mathcal{O}(n \cdot \log n)$ edges on the average. This pruning idea does not work on sparse random graphs, let alone arbitrary graphs with random edge weights.

Another pruning method was explored by Sedgewick and Vitter [130] for the average-case analysis of the *One-Pair Shortest-Paths* (OPSP) problem on *random geometric graphs* $G_n^d(r)$. Graphs of the class $G_n^d(r)$ are constructed as follows: n nodes are randomly placed in a d-dimensional unit cube, and each edge weight equals the Euclidean distance between the two involved nodes. An edge (u, v) is included if the Euclidean distance between u and v does not exceed the parameter $r \in [0, 1]$. Random geometric graphs have been intensively studied since they are considered to be a relevant abstraction for many real world situations [40, 130]. Assuming that the source node s and target node t are positioned in opposite corners of the cube, Sedgewick and Vitter showed that the OPSP algorithm can restrict itself to nodes and edges being "close" to the diagonal between s and t, thus obtaining average-case running-time $\mathcal{O}(n)$.

Mehlhorn and Priebe [102] proved that for the *complete* graph with random edge weights every SSSP algorithm has to check at least $\Omega(n \cdot \log n)$ edges with high probability. Noshita

[115] and Goldberg and Tarjan [70] analyzed the expected number of *decreaseKey* operations in Dijkstra's algorithm; for the asymptotically fastest priority queues, however, the resulting total average-case time of the algorithm does not improve on its worst-case complexity.

3.2 Our Contribution

We develop a new sequential SSSP approach, which adaptively splits buckets of an approximate priority-queue data-structure, thus building a bucket hierarchy. The new SSSP approach comes in two versions, **SP-S** and **SP-C**, following either the label-setting (**SP-S**) or the label-correcting (**SP-C**) paradigm. Our method is the first that provably achieves *linear* $\mathcal{O}(n+m)$ average-case execution time on *arbitrary directed* graphs.

In order to facilitate easy exposition we assume random edge weights chosen according to the uniform distribution on $[0,1]$, independently of each other. In fact, the result can be shown for much more general random edge weights in case the distribution of edge weights "is linear" in a neighborhood of zero. Furthermore, the proof for the average-case time-bound of **SP-S** does not require the edge weights to be independent. If, however, independence is given, then the linear-time bound for **SP-S** even holds with high probability. After some adaptations, the label-correcting version **SP-C** is equally reliable.

The worst-case times of the basic algorithms **SP-S** and **SP-C** are $\mathcal{O}(n \cdot m)$ and $\mathcal{O}(n \cdot m \cdot \log n)$, respectively. **SP-C** can be modified to run in $\mathcal{O}(n \cdot m)$ worst-case time as well. Furthermore, running any other SSSP algorithm with worst-case time $T(n,m)$ "in parallel" with either **SP-S** or **SP-C**, we can always obtain a combined approach featuring linear average-case time and $\mathcal{O}(T(n,m))$ worst-case time.

Our result immediately yields an $\mathcal{O}(n^2 + n \cdot m)$ average-case time algorithm for APSP, thus improving upon the best previous bounds on sparse directed graphs.

Furthermore, our analysis for **SP-C** implies the first theoretical average-case analysis for the "**A**pproximate **B**ucket **I**mplementation" [25] of Dijkstra's SSSP algorithm (ABI–Dijkstra): assuming either random graphs or arbitrary graphs with constant maximum node degree we show how the bucket widths must be chosen in order to achieve linear $\mathcal{O}(n+m)$ average-case execution time for random edge weights. The same results are obtained with high-probability for a slight modification of ABI–Dijkstra, the so-called sequential "Δ-Stepping" implementation.

Finally, we present a general method to construct sparse input graphs with random edge weights for which several label-correcting SSSP algorithms require superlinear average-case running-time: besides ABI–Dijkstra and Δ-Stepping we consider the "Bellman–Ford Algorithm" [15, 50], "Pallottino's Incremental Graph Algorithm" [117], the "Threshold Approach" by Glover et al. [64, 65, 66], the basic version of the "Topological Ordering SSSP Algorithm" by Goldberg and Radzik [69]. The obtained lower bounds are summarized in Table 3.1. It is worth mentioning that the constructed graphs contain only $\mathcal{O}(n)$ edges, thus maximizing the performance gap as compared to our new approaches with linear average-case time.

Preliminary accounts of our results on sequential SSSP have appeared in [104, 106]. Sub-

SSSP Algorithm	Average-Case Time
Bellman–Ford Alg. [15, 50]	$\Omega(n^{4/3-\epsilon})$
Pallottino's Incremental Graph Alg. [117]	$\Omega(n^{4/3-\epsilon})$
Basic Topological Ordering Alg. [69]	$\Omega(n^{4/3-\epsilon})$
Threshold Alg. [64, 65, 66]	$\Omega(n \cdot \log n / \log\log n)$
ABI–Dijkstra [25]	$\Omega(n \cdot \log n / \log\log n)$
Δ-Stepping [106] (Chap. 3.9.1)	$\Omega(n \cdot \sqrt{\log n / \log\log n})$

Table 3.1: Proved lower bounds on the average-case running times of some label-correcting algorithms on difficult input classes with $m = \mathcal{O}(n)$ edges and random edge weights.

sequently, Goldberg [68, 67] obtained the linear-time average-case bound for arbitrary directed graphs as well. He proposes a quite simple alternative algorithm based on radix heaps. For integer edge weights in $\{1, \ldots, M\}$, where M is reasonably small, it achieves improved worst-case running-time $\mathcal{O}(n \cdot \log C + m)$, where $C \leq M$ denotes the ratio between the largest and the smallest edge weight in G. However, for real weights in $[0, 1]$ as used in our analysis, the value of C may be arbitrarily large. Furthermore, compared to our label-correcting approaches, Goldberg's algorithm exhibits less potential for parallelizations. We comment on this in Section 4.8.

Even though Goldberg's algorithm is different from our methods, some underlying ideas are the same. Inspired by his paper we managed to streamline some proofs for **SP-S**, thus simplifying its analysis and making it more appealing. However, more involved proofs as those in [104, 106] are still necessary for the analysis of the label-correcting version **SP-C**.

3.3 Simple Bucket Structures

3.3.1 Dial's Implementation

Many SSSP labeling algorithms – including our new approaches – are based on keeping the set of candidate nodes Q in a data structure with *buckets*. This technique was already used in Dial's implementation [38] of Dijkstra's algorithm for integer edge weights in $\{0, \ldots, C\}$: a labeled node v is stored in the bucket $B[i]$ with index $i = \mathrm{tent}(v)$. In each iteration Dial's algorithm scans a node v from the first nonempty bucket, that is the bucket $B[k]$, where k is minimal with $B[k] \neq \emptyset$. In the following we will also use the term *current bucket*, B_{cur}, for the first nonempty bucket. Once $B_{\text{cur}} = B[k]$ becomes empty, the algorithm has to change the current bucket. As shown in Lemma 2, the smallest tentative distance among all labeled nodes in Q never decreases in the case of nonnegative edge weights. Therefore, the new current bucket can be identified by sequentially checking $B[k+1], B[k+2], \ldots$ until the next nonempty bucket is found.

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. Still, in the worst case, Dial's implementation has to traverse $C \cdot (n-1) + 1$ buckets. However, by reusing empty buckets cyclically, space for only $C + 1$ buckets is needed. In that case $B[i]$ is in charge of nodes v with $\mathrm{tent}(v) \bmod (C+1) = i$. That is, as

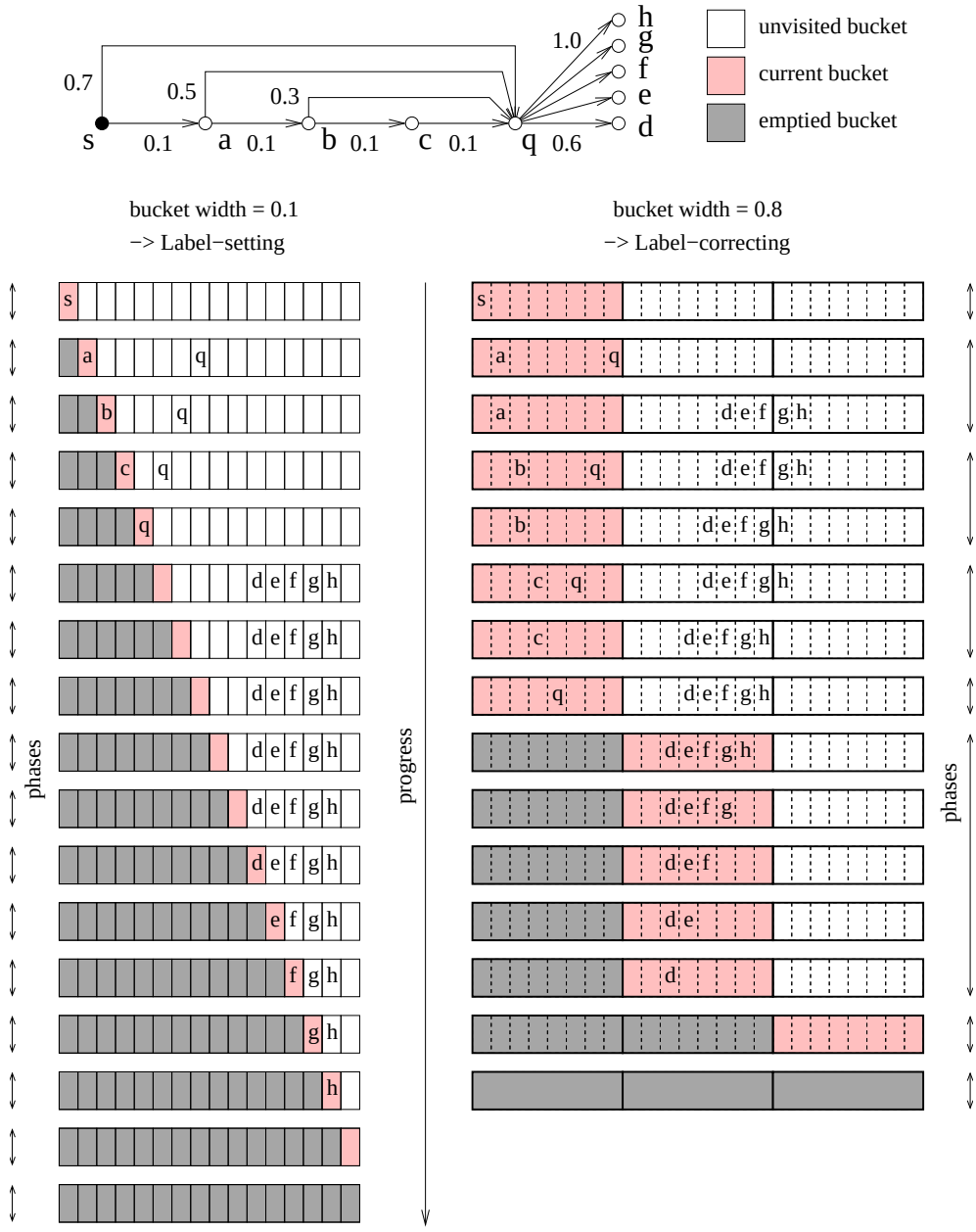


Figure 3.1: Impact of the bucket width Δ. The drawing shows the contents of the bucket structure for ABI-Dijkstra running on a little sample graph. If Δ is chosen too small then the algorithm spends many phases with traversing empty buckets. On the other hand, taking Δ too large causes overhead due to node rescans: in our example, the node q is rescanned several times. Choosing $\Delta = 0.8$ gives rise to 27 edge relaxations, whereas taking $\Delta = 0.1$ results in just 12 edge relaxations but more phases.

the algorithm proceeds, $B[i]$ hosts nodes with larger and larger tentative distances. In each iteration, however, the tentative distances of all nodes currently kept in $B[i]$ have the same value. This is an easy consequence of the fact that the maximum edge weight is bounded by C.

3.3.2 Buckets of Fixed Width

Dial's implementation associates one concrete tentative distance with each bucket. Alternatively, a whole interval of tentative distances may be mapped onto a single bucket: node $v \in Q$ 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 [43] and Denardo/Fox [36] demonstrated that a label-setting algorithm is easily obtained if $\lambda > 0$: taking $\Delta \leq \lambda$, all nodes in B_{cur} have reached their final distance; this is an immediate consequence of either Criterion 1 or Lemma 5 using the lower bound $M := j \cdot \Delta$ if $B_{\text{cur}} = B[j]$. Therefore, these nodes can be scanned in arbitrary order.

Choosing $\Delta > \lambda$ either requires to repeatedly find a node with smallest tentative distance in B_{cur} or results in a label-correcting algorithm: in the latter case it may be needed to *rescan* nodes from B_{cur} if they have been previously scanned with non-final distance value. This variant is also known as the "Approximate Bucket Implementation of Dijkstra's algorithm" [25] (ABI-Dijkstra) where the nodes of the current bucket are scanned in FIFO order. The choice of the bucket width has a crucial impact on the overall performance of ABI-Dijkstra. On the one hand, the bucket width should be small in order to limit the number of node rescans. On the other hand, setting Δ very small may result in too many buckets. Figure 3.1 depicts the tradeoff between these two parameters.

Sometimes, there is no good compromise for Δ: in Section 3.10.3 we will provide a graph class with $\mathcal{O}(n)$ edges and random edge weights, where each fixed choice of Δ forces ABI-Dijkstra into superlinear average-case running time. Therefore, our new SSSP approaches change the bucket width adaptively.

3.4 The New Algorithms

3.4.1 Preliminaries

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 nonnegative edge weights meets this requirement after proper scaling. The algorithms keep the candidate node set Q in a *bucket hierarchy* $\mathcal{B}$: they begin with an array L_0 at level 0. L_0 contains n buckets, $B_{0,0}, \ldots, B_{0,n-1}$, each of which has width $\Delta_0 = 2^{-0} = 1$. The starting node s is put into the bucket $B_{0,0}$ with $\mathrm{tent}(s) = 0$. Bucket $B_{0,j}$, $0 \leq j < n$, represents tentative distances in the range $[j, j + 1)$.

Our algorithms may extend the initial bucket structure by creating new levels (and later removing them again). Thus, beside the initial n buckets of the first level, they may create new buckets. All buckets of level i have equal width Δ_i, where $\Delta_i = 2^{-l}$ for some integer $l \geq i$. The array L_{i+1} at level $i + 1$ refines a single bucket $B_{i,k}$ of the array L_i at level i. This means that the range of tentative distances $[x, x + \Delta_i)$ associated with $B_{i,k}$ is divided over the buckets of L_{i+1} as follows: let L_{i+1} contain $\kappa_{i+1} = \Delta_i/\Delta_{i+1} \geq 2$ buckets; then $B_{i+1,j}$, $0 \leq j < \kappa_{i+1}$ keeps nodes with tentative distances in the range $[x + j \cdot \Delta_{i+1}, x + (j + 1) \cdot \Delta_{i+1})$. Since level $i + 1$ covers the range of some single bucket of level i we also say that level $i + 1$ has *total level width* Δ_i.

The level with the largest index i is also referred to as the *topmost* or *highest* level. The *height* of the bucket hierarchy denotes the number of levels, i.e., initially it has height one. The *first* or *leftmost* nonempty bucket of an array L_i denotes the bucket $B_{i,k}$ where k is minimal with $B_{i,k} \neq \emptyset$. Within the bucket hierarchy $\mathcal{B}$, the *current bucket* B_{cur} always refers to the leftmost nonempty bucket in the highest level.

Our algorithms generate a new topmost level $i+1$ by *splitting* the current bucket $B_{\text{cur}} = B_{i,k}$ of width Δ_i at level i. When a node v contained in $B_{i,k}$ is moved to its respective bucket in the new level then we say that v is *lifted*. After all nodes have been removed from $B_{i,k}$, the current bucket is reassigned to be the leftmost nonempty bucket in the new topmost level.

3.4.2 The Common Framework

After initializing the buckets of level zero, setting $B_{\text{cur}} := B_{0,0}$, and inserting s into B_{cur}, our algorithms **SP-S** and **SP-C** work in *phases*. It turns out that each phase will settle at least one node, i.e., it scans at least one node with final distance value. A phase first identifies the new current bucket B_{cur}. Then it inspects the nodes in it, and takes a preliminary decision whether B_{cur} should be split or not.

If not, the algorithms scan *all* nodes from B_{cur} simultaneously[1]; we will denote this operation by $\text{SCAN\_ALL}(B_{\text{cur}})$. We call this step of the algorithm *regular node scanning*. As a result, B_{cur} is first emptied but it may be refilled due to the edge relaxations of $\text{SCAN\_ALL}(B_{\text{cur}})$. This marks the *regular* end of the phase. If after a regular phase all buckets of the topmost level are empty, then this level is removed. The algorithms stop after level zero is removed.

Otherwise, i.e., if B_{cur} should be split, then the algorithms first identify a node set $U_1 \cup U_2 \subseteq B_{\text{cur}}$ (see Figure 3.2) whose labels are known to have reached their final distance values; note that in general $U_1 \cup U_2 \subsetneq B_{\text{cur}}$. Then all these nodes are scanned, denoted by $\text{SCAN\_ALL}(U_1 \cup U_2)$. This step of the algorithm is called *early node scanning*. It removes $U_1 \cup U_2$ from B_{cur} but may also insert additional nodes into it due to edge relaxations. If B_{cur} remains nonempty after $\text{SCAN\_ALL}(U_1 \cup U_2)$ then the new level is actually created and the remaining nodes of B_{cur} are lifted to their respective buckets of the newly created level. The phase is over; in that case we say that the phase found an *early end*. If, however, the new level was not created after all (because B_{cur} became empty after $\text{SCAN\_ALL}(U_1 \cup U_2)$) then the phase still ended regularly.

3.4.3 Different Splitting Criteria

The label-setting version **SP-S** and the label-correcting version **SP-C** only differ in the bucket splitting criterion. Consider a current bucket B_{cur}. **SP-S** splits B_{cur} until it contains a single node v; by then $\text{tent}(v) = \text{dist}(v)$. If there is more than one node in B_{cur} then the current bucket is split into two new buckets; compare Figure 3.2.

In contrast, adaptive splitting in **SP-C** is applied to achieve a compromise between either scanning too many narrow buckets or incurring too many node rescans due to wide buckets:

[1] Actually, the nodes of the current bucket could also be extracted one-by-one in FIFO order. However, in view of our parallelizations (Chapter 4), it proves advantageous to consider them in phases.

SP-S **(* SP-C *)**

Create L_0, initialize $\mathrm{tent}(\cdot)$, and insert s into $B_{0,0}$
$i := 0$, $\Delta_i := 1$
while $i \geq 0$ **do**
  while nonempty bucket exists in L_i **do**
    repeat
      $\mathit{regular}$:= true /* Begin of a new phase */
      $j := \min\{k \geq 0 \mid B_{i,k} \neq \emptyset\}$
      $B_{\text{cur}} := B_{i,j}$, $\Delta_{\text{cur}} := \Delta_i$
      $b^* :=$ # nodes in B_{cur} **(*** $d^* := \max\{\text{degree}(v) \mid v \in B_{\text{cur}}\}$ ***)**
      if $b^* > 1$ **then** **(* if** $d^* > 1/\Delta_{\text{cur}}$ **then *)**
        $U_1 := \{v \in B_{\text{cur}} \mid \forall u \in B_{\text{cur}} : \mathrm{tent}(v) \leq \mathrm{tent}(u)\}$
        $U_2 := \{v \in B_{\text{cur}} \mid \forall (u,v) \in E : c(u,v) \geq \Delta_{\text{cur}}\}$
        SCAN_ALL$(U_1 \cup U_2)$ /* Early scanning */
        if $B_{\text{cur}} \neq \emptyset$
          Create L_{i+1} with $\Delta_{i+1} := \Delta_{\text{cur}}/2$ **(*** with $\Delta_{i+1} := 2^{-\lceil \log_2 d^* \rceil}$ ***)**
          Lift all remaining nodes from B_{cur} to L_{i+1}
          $\mathit{regular}$:= false, $i := i + 1$ /* Early end of a phase */
    until $\mathit{regular}$ = true
    SCAN_ALL(B_{cur}) /* Regular scanning / end of a phase */
  Remove L_i, $i := i - 1$

Figure 3.2: Pseudo code for **SP-S**. Modifications for **SP-C** are given in (* comments *). SCAN_ALL denotes the extension of the SCAN operation (Figure 2.1) to sets of nodes.

let d^* denote the maximum node degree (sum of in-degree plus out-degree) among all nodes in B_{cur} at the topmost level i. If $d^* > 1/\Delta_i$, then **SP-C** splits B_{cur} into $\kappa_i = \Delta_i/\Delta_{i+1} \geq 2$ new buckets having width $\Delta_{i+1} = 2^{-\lceil \log_2 d^* \rceil}$ each; compare Figure 3.2 and Figure 3.3.

Both splitting criteria imply a simple upper bound on the bucket widths in the hierarchy: initially, buckets of level 0 have width $\Delta_0 \leq 2^{-0} = 1$. Each subsequent splitting of $B_{\text{cur}} = B_{i,j}$ at level i creates a new level $i+1$ with at least two buckets of width $\Delta_{i+1} \leq \Delta_i/2$. Therefore, we find by induction:

Invariant 1 *Buckets of level i have width $\Delta_i \leq 2^{-i}$.*

For the label-correcting version **SP-C**, the invariant can be refined as follows:

Invariant 2 *Let d denote the maximum degree among all nodes in the current bucket $B_{cur} = B_{i,j}$ at level i when the regular node scanning of* **SP-C** *starts. Then the bucket width Δ_i of B_{cur} is at most $\min\{2^{-\lceil \log_2 d \rceil}, 2^{-i}\}$.*

In the following we will collect some more observations concerning the current bucket:

3.4.4 The Current Bucket

The algorithms stay with the same current bucket $B_{\text{cur}} = B_{i,j}$ until it is split or finally becomes empty after a phase. Then a new current bucket must be found: the leftmost

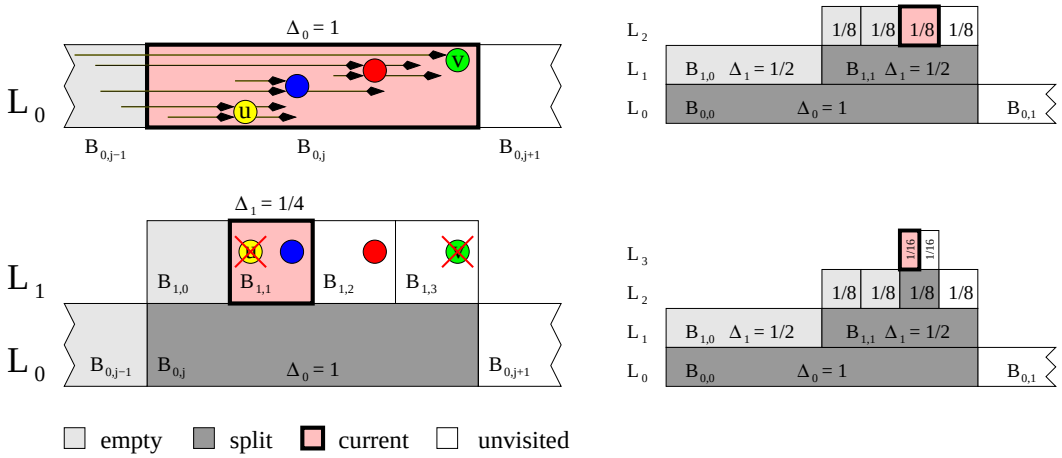


Figure 3.3: (Left) Basic case in the label-correcting approach: detecting a node with degree 4 in $B_{0,j}$ of L_0 forces a split into buckets of widths $2^{-\lceil \log_2 4 \rceil} = 1/4$. Nodes u and v are not lifted but selected for early node scans (u has smallest tentative distance in B_{cur}, all incoming edges of v have weight larger than Δ_{cur}). (Right) General situation: after a bucket split, the first nonempty bucket of the highest level becomes the new current bucket.

nonempty bucket of the topmost level. Thus, B_{cur} is reassigned in a way that maintains the following invariant:

Invariant 3 *When a phase of* **SP-S** *or* **SP-C** *starts then the current bucket B_{cur} contains a candidate node with smallest tentative distance. Furthermore, if some node $v \in Q$ is contained in B_{cur}, then any node $u \in Q$ having $\text{tent}(u) \leq \text{tent}(v)$ is contained in B_{cur} as well.*

This is easily seen by induction: initially, the nodes with the smallest tentative distances reside in the leftmost nonempty bucket of level zero; i.e., in B_{cur}. By the monotonicity property (Lemma 2) the smallest tentative distance in Q never decreases. Therefore, if $B_{\text{cur}} = B_{0,j}$ becomes empty after a phase, then the next nonempty bucket can be found by linear search to the right, i.e., testing $B_{0,j+1}, B_{0,j+2}, \ldots$. When level $i+1$ comes into existence then it inherits the tentative distance range and the non-scanned nodes from the single bucket B_{cur} at level i. After lifting, the nodes with smallest tentative distances are found in the leftmost nonempty bucket of level $i+1$, thus maintaining the invariant.

Reassigning B_{cur} at the new level $i+1$ is done by linear search again, this time starting from the leftmost bucket of array L_{i+1}, $B_{i+1,0}$. Subsequent reassignments of B_{cur} on the topmost level $i+1$ continue the search from the previous current bucket of level $i+1$. In case there is no further nonempty bucket on the topmost level then this level is removed and the search continues at the first non-split bucket of the previous level, if any.

Remark 1 *Note that the algorithms can keep counters for the number of nodes currently stored in each level. Thus, if the counter for the topmost level is zero at the beginning of a phase, this level can be instantly removed without further sequential search for another nonempty bucket, which does not exist*[2].

[2]The usage of these counters does not influence the sequential complexity of the algorithms, but they will

Altogether we can summarize the crucial points about reassigning the current bucket as follows:

Observation 2 *Let t be the total number of newly created buckets. Keeping track of the current bucket can be done in $\mathcal{O}(n+t)$ time.*

3.4.5 Progress of the Algorithms

We continue with some observations concerning the efficiency of the scan operations:

Lemma 8 *The tentative distances of all nodes in the set $U_1 \cup U_2$ are final.*

Proof: Let B_{cur} be in charge of the distance range $[M, M+\Delta_{\text{cur}})$. Hence, $\text{tent}(v) < M + \Delta_{\text{cur}}$ for any node $v \in U_1 \cup U_2$. By Invariant 3, $M \leq \min\{\text{tent}(v) : v \in Q\}$. Therefore, we can apply the IN-criterion of Lemma 5 where $U_1 = \mathcal{U}_1$ and $U_2 \subseteq \mathcal{U}_2$. □

Observe that whenever $\text{SCAN\_ALL}(B_{\text{cur}})$ is executed in **SP-S** then B_{cur} contains at most one node. Together with Invariant 3 and Lemmas 4, 5 and 8 this implies

Corollary 1 **SP-S** *implements the label-setting paradigm.*

A further remark is in order concerning the IN-criterion:

Remark 2 *Our algorithms may lift a node v several times. In that situation, a naive method to check whether v belongs to the set U_2 might repeatedly read the weights of all edges into v and recompute the minimum. This can result in a considerable overhead if the in-degree of v is large and v is lifted many times. Therefore, it is better to determine the smallest incoming edge weight of each node once and for all during the initialization. This preprocessing takes $\mathcal{O}(n+m)$ time; the result can be stored in an extra array. Afterwards, each check whether a node v belongs to U_2 or not can be done in constant time by a lookup in the extra array.*

In the following we turn to the number of phases:

Lemma 9 *Each phase of* **SP-S** *or* **SP-C** *settles at least one node.*

Proof: By Invariant 3, B_{cur} contains a candidate node v with smallest tentative distance. Furthermore, according to Lemma 4, $\text{tent}(v) = \text{dist}(v)$. A phase with regular end scans all nodes from B_{cur}, hence v will be settled. In case of an early phase end, v belongs to the set U_1. Therefore, it is scanned in that case, too. □

At most $\min\{n, m\}$ nodes are reachable from the source node s. Consequently, the algorithms require at most $\min\{n, m\}$ phases. Each phase causes at most one splitting. As for **SP-S**, each splitting creates at most two new buckets. Hence, we immediately obtain the following simple upper bounds:

Corollary 2 *At any time the bucket hierarchy for* **SP-S** *contains at most $\min\{n, m\}$ levels.* **SP-S** *creates at most $2 \cdot \min\{n, m\}$ new buckets.*

prove useful for subsequent parallelizations.

The splitting criterion of **SP-C** implies better bounds on the maximum height of the hierarchy:

Lemma 10 *At any time the bucket hierarchy for* **SP-C** *contains at most* $\lceil \log_2 n \rceil + 2$ *levels.* **SP-C** *creates at most* $4 \cdot m$ *new buckets.*

Proof: A current bucket on level i having width Δ_i is only split if it contains a node with degree $d > 1/\Delta_i$. By Invariant 1, $\Delta_i \leq 2^{-i}$. Furthermore, $d \leq 2 \cdot n$ (recall that d is the sum of in-degree and out-degree). Therefore, $2 \cdot n > 1/\Delta_i \geq 2^i$ implies that splittings may only happen on level i, $i \geq 0$, if $i < \log_2 n + 1 \leq \lceil \log_2 n \rceil + 1$. In other words, there can be at most $\lceil \log_2 n \rceil + 2$ levels.

During the execution of **SP-C**, if a node $v \in B_{\text{cur}}$ is found that has degree $d > 1/\Delta_{\text{cur}}$ and all other nodes in B_{cur} have degrees at most d, then v causes a splitting of B_{cur} into at most $2 \cdot (d-1)$ new buckets. After node v caused a splitting it is either settled by the subsequent early scan operations or it is lifted into buckets of width at most $1/d$. Observe that v never falls back to buckets of larger width. Therefore, each node can cause at most one splitting. Consequently, **SP-C** creates at most $\sum_{v \in V} 2 \cdot (\text{degree}(v) - 1) \leq 4 \cdot m$ new buckets. □

The observations on the maximum hierarchy heights given above naturally bound the number of lift operations for each node $v \in V$. However, we can provide sharper limits based on the weights of the edges into v:

Lemma 11 *Consider a reachable node* $v \in V$ *with in-degree* d'*. Let* $e_1 = (u_1, v), \ldots, e_{d'} = (u_{d'}, v)$ *denote its incoming edges. Furthermore, let* h *be an upper bound on the maximum height of the bucket hierarchy. Node* v *is lifted at most*

$$L(v) := \min \left\{ h - 1, \max_{1 \leq j \leq d'} \left\lceil \log_2 \frac{1}{c(e_j)} \right\rceil \right\} \text{ times.}$$

Proof: Within the bucket hierarchy, nodes can only be moved upwards, i.e. to levels with higher indices. This is obvious for **SP-S**; in the case of **SP-C**, note that a rescanned node can only reappear in the same B_{cur} it was previously scanned from, or in one of its refining buckets on higher levels. Thus, each node can be lifted at most $h - 1$ times. Let $c_v^* := \min_{1 \leq j \leq d'} c(e_j)$ be the weight of the lightest edge into node v. If v is lifted fewer than $\left\lceil \log_2 \frac{1}{c_v^*} \right\rceil$ times or if $\left\lceil \log_2 \frac{1}{c_v^*} \right\rceil > h$, then the claim holds. Otherwise, after $\left\lceil \log_2 \frac{1}{c_v^*} \right\rceil$ lift operations, v resides in a current B_{cur} of width at most $2^{-\left\lceil \log_2 \frac{1}{c_v^*} \right\rceil} \leq c_v^*$ (by Invariant 1). If B_{cur} is split then $v \in U_1 \cup U_2$. Hence, v will be settled by early node scanning of $U_1 \cup U_2$ (Lemma 8); on the other hand, if B_{cur} is not split, then v is eventually settled by regularly scanning the nodes in B_{cur}. In both cases, v will not be lifted any more. □

Using $\max_{1 \leq j \leq d'} \left\lceil \log_2 \frac{1}{c(e_j)} \right\rceil \leq \sum_{1 \leq j \leq d'} \left\lceil \log_2 \frac{1}{c(e_j)} \right\rceil$, Lemma 11 yields the following simple inequality, which in general is far from being sharp:

Corollary 3 **SP-S** *and* **SP-C** *perform at most* $\sum_{e \in E} \left\lceil \log_2 \frac{1}{c(e)} \right\rceil$ *lift operations.*

3.4.6 Target-Bucket Searches

We still have to specify how the SCAN_ALL procedure works in the bucket hierarchy. For the relaxation of an edge $e = (u, v)$ it is necessary to find the appropriate target bucket that is in charge of the decreased $\mathrm{tent}(v)$: 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 $t = \mathrm{tent}(v)$, the search starts in the array 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 the array L_1 and checks $B^1 = [b_1, b_1 + \Delta_1)$ where $b_1 = b_0 + \lfloor (t - b_0)/\Delta_1 \rfloor \cdot \Delta_1$. Generally, if the target bucket $B^i = [b_i, b_i + \Delta_i)$ at level i has been split then the search proceeds with bucket $B^{i+1} = [b_{i+1}, b_{i+1} + \Delta_{i+1})$ where $b_{i+1} = b_i + \lfloor (t - b_i)/\Delta_{i+1} \rfloor \cdot \Delta_{i+1}$.

Each level can be checked in constant time. In the worst case all levels of the current bucket hierarchy must be examined, i.e., at most n levels for **SP-S** (Corollary 2), and at most $\lceil \log_2 n \rceil + 2$ levels for **SP-C** (Lemma 10); see also Figure 3.4. Better bounds can be obtained if we include the weight of the relaxed edge into our considerations (Lemma 12). Furthermore, in Section 3.7.1 (Lemma 19) we will give an extension that reduces the worst-case time for a target-bucket search to $\mathcal{O}(1)$.

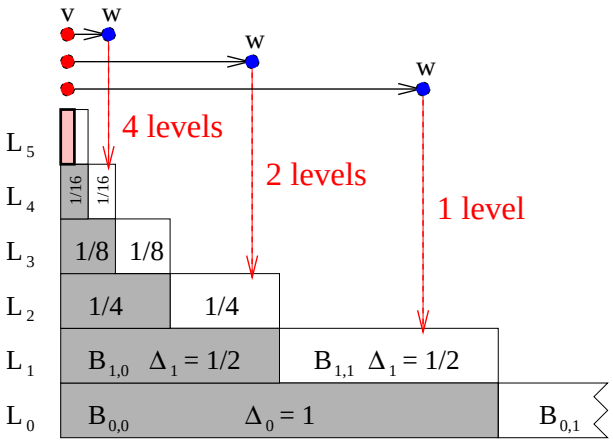


Figure 3.4: A worst-case setting for relaxations of the edge (v, w) with bottom-up search for the target bucket. The smaller the edge weight the more levels have to be checked.

Lemma 12 *Let h be an upper bound on the maximum height of the bucket hierarchy. The bottom-up target-bucket search for the relaxation of an edge $e = (v, w)$ with weight $c(e)$ checks at most $R(v, w) := \min \left\{ h, \left\lceil \log_2 \frac{1}{c(e)} \right\rceil + 1 \right\}$ levels.*

Proof: When the edge $e = (v, w)$ from the forward star $FS(v)$ is relaxed, then v is scanned from a current bucket $B_{\text{cur}} := B_{k,l}$ at the topmost level k. Hence, the target-bucket search for w examines at most $k + 1 \leq h$ levels. The claim trivially holds if the target bucket for w belongs to level 0. Therefore, let us assume that the proper target bucket lies in a level $i^* \geq 1$. Let $R_i = [r_i, r_i + \Delta_{i-1})$ be the range of tentative distances level $i \geq 1$ is in charge of; recall that the whole range of level i represents the range of a single split bucket at level $i - 1$. Due to the way buckets are split, $R_k \subseteq R_i$ for $1 \leq i \leq k$. Therefore, when e is relaxed we have $\mathrm{tent}(v) \geq r_k \geq r_i$. The level i^* of the target bucket must be in

charge of the new tentative distance $\mathrm{tent}(v) + c(e) \geq r_{i^*} + c(e)$. Hence, $c(e) < \Delta_{i^*-1}$. By Invariant 1, $\Delta_{i^*-1} \leq 2^{-i^*+1}$. After transformations we obtain $i^* < \left\lceil \log_2 \frac{1}{c(e)} \right\rceil + 1$. □

Remark 3 *For non-negative edge weights the algorithms never relax self-loop edges* (v, v). *Hence, the total costs to scan a node* v *including edge relaxations is bounded by* $\mathcal{O}\left(1 + \sum_{w \in FS(v), w \neq v} R(v, w)\right)$.

This concludes our collection of basic observations. In the next sections we will combine these partial results in order to obtain performance bounds for **SP-S** and **SP-C**.

3.5 Performance of the Label-Setting Version SP-S

In this section we will first prove that **SP-S** has worst-case time $\mathcal{O}(n \cdot m)$. Then we show that it runs in linear $\mathcal{O}(n + m)$ time on the average. Finally, we prove that this bound also holds with high probability.

Initializing global arrays for tentative distances, level zero buckets, and pointers to the queued nodes and storing the lightest incoming edge for each node (Remark 2) can clearly be done in $\mathcal{O}(n + m)$ time for both algorithms **SP-S** and **SP-C**. By Lemma 9, there are at most n phases, each of which requires constant time for setting control variables, etc. The remaining costs of **SP-S** account for the following operations:

(a) Scanning nodes.

(b) Generating, traversing and removing new buckets.

(c) Lifting nodes in the bucket hierarchy.

As for (a), we have seen before that **SP-S** performs label-setting (Corollary 1); each node is scanned at most once. Consequently, each edge is relaxed at most once, as well. By Corollary 2, the bucket hierarchy for **SP-S** contains at most n levels. Therefore, Lemma 12 implies that the total number of levels checked during all relaxations is bounded by

$$\min\left\{ n \cdot m \ , \ \sum_{e \in E} \left(\left\lceil \log_2 \frac{1}{c(e)} \right\rceil + 1 \right) \right\}. \tag{3.1}$$

Concerning (b), we know from Corollary 2 that the algorithm creates at most

$$2 \cdot \min\{n, m\} \tag{3.2}$$

new buckets. Traversing and finally removing them can be done in time linear in the number of buckets; also compare Observation 2. Identifying the sets U_1 and U_2 during a split takes time linear in the number of nodes in the split bucket. Each of these nodes is either early scanned or lifted right after the split. Hence, the constant time share to check whether such a node belongs to $U_1 \cup U_2$ can be added to its respective scan or lift operation.

Finally, the upper bound for (c) follows from Lemma 11 and Corollary 3 where $h \leq n$, and at most $\min\{n, m\}$ nodes are reachable from s: **SP-S** performs at most

$$\min\left\{ \min\{n, m\} \cdot n \ , \ \sum_{e \in E} \left\lceil \log_2 \frac{1}{c(e)} \right\rceil \right\} \text{ lift operations.} \tag{3.3}$$

Combining the bounds above immediately yields the following result:

Theorem 1 **SP-S** *requires* $\mathcal{O}\left(\min\left\{n \cdot m\,,\; n + m + \sum_{e\in E}\left\lceil \log_2 \frac{1}{c(e)}\right\rceil\right\}\right)$ *time.*

In the following we will consider the average-case behavior of **SP-S**:

3.5.1 Average-Case Complexity of SP-S

Theorem 2 *On arbitrary directed networks with random edge weights that are uniformly drawn from* $[0,1]$, **SP-S** *runs in* $\mathcal{O}(n+m)$ *average-case time; independence of the random edge weights is* not *needed.*

Proof: Define $X_e := \left\lceil \log_2 \frac{1}{c(e)}\right\rceil$. By Theorem 1, **SP-S** runs in $\mathcal{O}\left(n + m + \sum_{e\in E} X_e\right)$ time. Hence, it is sufficient to bound the expected value of $\sum_{e\in E} X_e$. Due to the linearity of expectation we have $\mathbf{E}\left[\sum_{e\in E} X_e\right] = \sum_{e\in E}\mathbf{E}\left[X_e\right]$. By the uniform distribution,

$$\mathbf{E}\left[X_e\right] \;=\; \sum_{i=1}^{\infty} i\cdot \mathbf{P}\left[X_e = i\right] = \sum_{i=1}^{\infty} i\cdot 2^{-i} = 2 \tag{3.4}$$

for any edge $e \in E$. Therefore, $\mathbf{E}\left[\sum_{e\in E} X_e\right] = \mathcal{O}(m)$. □

Note once more that Theorem 2 does not require the random edge weights to be independent. However, if they are independent then the result even holds with high probability. This can be shown with a concentration result for the sum of independent, geometrically distributed random variables:

Lemma 13 ([127], Theorem 3.38) *Let* $X_1,\ldots,X_k$ *be independent, geometrically distributed random variables with parameters* $p_1,\ldots,p_k \in (0,1)$. *Let* $\mu = \mathbf{E}[\sum_{i=1}^{k} X_i]$, *and let* $p = \min_{1\le i\le k} p_i$. *Then for all* $\delta > 0$ *it holds that*

$$\mathbf{P}\left[\sum_{i=1}^{k} X_i \ge (1+\delta)\cdot\mu\right] \le \left(1 + \frac{\delta\cdot\mu\cdot p}{k}\right)^{k}\cdot e^{-\delta\cdot\mu\cdot p}\,. \tag{3.5}$$

Proof: In [127] it is shown how (3.5) can be derived from the Chernoff bounds. □

Theorem 3 **SP-S** *requires* $\mathcal{O}(n+m)$ *time with high probability on arbitrary directed networks with random independent edge weights uniformly drawn from* $[0,1]$.

Proof: Define $X_e := \left\lceil \log_2 \frac{1}{c(e)}\right\rceil$. As in the case of Theorem 2 we have to bound $\sum_{e\in E} X_e$. Note that $\mathbf{P}\left[X_e = i\right] = (1/2)^i$, i.e., X_e is a geometrically distributed random variable with parameter $p_e = 1/2$. Furthermore, $\mathbf{E}[X_e] = 2$. Hence, we can use Formula (3.5) with $\delta = 1$, $p = 1/2$, and $\mu = 2\cdot m$ to see

$$\mathbf{P}\left[\sum_{e\in E} X_e \ge 4\cdot m\right] \le (2/e)^m\,.$$

□

3.5.2 Immediate Extensions

This section deals with a few simple extensions for **SP-S**: in particular, we consider a larger class of distribution functions for the random edge weights, sketch how to improve the worst-case performance, and identify implications for the All-Pairs Shortest-Paths problem.

Other Edge Weight Distributions

Theorem 2 does not only hold for random edge weights uniformly distributed in $[0,1]$: from its proof it is obvious that any distribution function for the random edge weights satisfying

$$\mathbf{E}\left[\left\lceil \log_2 \frac{1}{c(e)} \right\rceil\right] = \mathcal{O}(1)$$

is sufficient. The edge weights may be dependent, and even different distribution functions for different edge weights are allowed.

The uniform distribution in $[0,1]$ is just a special case of the following much more general class: let us assume $0 \leq c(e) \leq 1$ has a distribution function F_e with the properties that $F_e(0) = 0$ and that $F'_e(0)$ is bounded from above by a positive constant. These properties imply that F_e can be bounded from above as follows: there is an integer constant $\alpha \geq 1$ so that for all $0 \leq x \leq 2^{-\alpha}$, $F_e(x) \leq 2^{\alpha} \cdot x$. Let Y_e be a random variable that is uniformly distributed in $[0, 2^{-\alpha}]$. For all $0 \leq x \leq 1$, $\mathbf{P}[\, c(e) \leq x \,] \leq \mathbf{P}[\, Y_e \leq x \,] = \min\{2^{\alpha} \cdot x, 1\}$. Consequently, for all integers $i \geq 1$,

$$\mathbf{P}\left[\left\lceil \log_2 \frac{1}{c(e)} \right\rceil \geq i\right] \leq \mathbf{P}\left[\left\lceil \log_2 \frac{1}{Y_e} \right\rceil \geq i\right] = \min\{2^{\alpha-i}, 1\}.$$

Thus,

$$\mathbf{E}\left[\left\lceil \log_2 \frac{1}{c(e)} \right\rceil\right] \leq \sum_{i=1}^{\infty} i \cdot \min\{2^{\alpha-i}, 1\} \leq \alpha + \sum_{i=1}^{\infty} (i+\alpha) \cdot 2^{-i} = 4 \cdot \alpha.$$

Other Splitting Criteria

If the current bucket B_{cur} contains $b^* > 1$ nodes at the beginning of a phase then **SP-S** splits B_{cur} into two new buckets. Actually, **SP-S** still runs in linear average-case time if B_{cur} is split into $\mathcal{O}(b^*)$ buckets: in that case the total number of newly generated buckets over the whole execution of **SP-S** is linear in the total number of nodes plus the total number of lift operations, i.e., $\mathcal{O}(n+m)$ on the average.

Improved Worst-Case Bounds

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 nonnegative integer edge weights [124].

All-Pairs Shortest-Paths

Solving SSSP for each source node separately is the most obvious All-Pairs Shortest-Paths algorithm. For sparse graphs, this stragegy is very efficient. In fact, **SP-S** implies that the APSP problem on arbitrary directed graphs can be solved in $\mathcal{O}(n^2 + n \cdot m)$ time on the average. This is optimal if $m = \mathcal{O}(n)$. All previously known average-case bounds for APSP were in $\omega(n^2)$.

3.6 Performance of the Label-Correcting Version SP-C

In analogy to Section 3.5 we first consider the worst-case complexity of **SP-C** (Figure 3.5). We give a simple proof for the $\mathcal{O}(n \cdot m \cdot \log n)$-bound and then continue with a more detailed treatment of the number of node scans. Subsequently, we turn to the average-case running time. For the high-probability bound, however, **SP-C** needs some modifications; we will deal with that in Section 3.7.

SP-C

Create L_0, initialize $\mathrm{tent}(\cdot)$, and insert s into $B_{0,0}$
$i := 0$, $\Delta_i := 1$
while $i \geq 0$ **do**
  while nonempty bucket exists in L_i **do**
    repeat
      regular := true /* Begin of a new phase */
      $j := \min\{k \geq 0 \mid B_{i,k} \neq \emptyset\}$
      $B_{\text{cur}} := B_{i,j}$, $\Delta_{\text{cur}} := \Delta_i$
      $d^* := \max\{\text{degree}(v) \mid v \in B_{\text{cur}}\}$
      if $d^* > 1/\Delta_{\text{cur}}$ **then**
        $U_1 := \{v \in B_{\text{cur}} \mid \forall u \in B_{\text{cur}} : \mathrm{tent}(v) \leq \mathrm{tent}(u)\}$
        $U_2 := \{v \in B_{\text{cur}} \mid \forall (u,v) \in E : c(u,v) \geq \Delta_{\text{cur}}\}$
        SCAN_ALL($U_1 \cup U_2$) /* Early scanning */
        if $B_{\text{cur}} \neq \emptyset$
          Create L_{i+1} with $\Delta_{i+1} := 2^{-\lceil \log_2 d^* \rceil}$
          Lift all remaining nodes from B_{cur} to L_{i+1}
          regular := false, $i := i + 1$ /* Early end of a phase */
    until *regular* = true
    SCAN_ALL(B_{cur}) /* Regular scanning / end of a phase */
  Remove L_i, $i := i - 1$

Figure 3.5: Pseudo code for **SP-C**.

Theorem 4 **SP-C** *requires* $\mathcal{O}(n \cdot m \cdot \log n)$ *time in the worst case.*

Proof: Similar to **SP-S**, the initialization of the data structures for **SP-C** can be done in $\mathcal{O}(n + m)$ time. By Lemma 10, **SP-C** creates at most $4 \cdot m$ new buckets. Hence, creating,

traversing and removing all buckets can be done in $\mathcal{O}(n+m)$ time (see also Observation 2). The bucket hierarchy for **SP-C** contains at most $h = \mathcal{O}(\log n)$ levels (Lemma 10). Consequently, **SP-C** performs at most

$$\min\left\{ n \cdot h \ , \ \sum_{e \in E} \left\lceil \log_2 \frac{1}{c(e)} \right\rceil \right\} = \mathcal{O}(n \cdot \log n)$$

lift operations (Lemma 11 and Corollary 3). By Lemma 9, there are at most n phases, each of which settles at least one node. Therefore, even though nodes may be rescanned, no node is scanned more than n times. That implies that altogether at most $n \cdot m$ edges are relaxed. Due to $h = \mathcal{O}(\log n)$, the target-bucket search for each relaxation requires at most $\mathcal{O}(\log n)$ time. Hence, **SP-C** runs in $\mathcal{O}(n \cdot m \cdot \log n)$ time in the worst-case. □

In Section 3.7.1 (Lemma 19) we will give an extension that reduces the worst-case time for a target-bucket search to $\mathcal{O}(1)$. After this modification, the asymptotic worst-case time for **SP-C** will be $\mathcal{O}(n \cdot m)$. In the following we will have a closer look at the number of scan operations.

3.6.1 The Number of Node Scans

Each node v that is reachable from the source node s will be inserted into the bucket structure $\mathcal{B}$ and scanned from some current bucket at least once. Recall from the description of the algorithms in Section 3.4.2 that there are *regular* node scans and *early* node scans; for an early scan of v, the IN-criterion (Lemma 5) ensures $\mathrm{tent}(v) = \mathrm{dist}(v)$. A *re-insertion* of v into $\mathcal{B}$ occurs if v was previously scanned with non-final distance value $\mathrm{tent}(v)$, and now the relaxation of an edge into v reduces $\mathrm{tent}(v)$. A re-insertion of v later triggers a *rescan* of v from some current bucket involving re-relaxations of v's outgoing edges. We distinguish *local rescans* of v, i.e., v is rescanned from the same current bucket from which it was scanned before, and *nonlocal rescans* of v, where v is rescanned from a different current bucket.

Lemma 14 (Number of Scan Operations) *Let $v \in V$ be an arbitrary node with degree $d \geq 1$ and in-degree d', $1 \leq d' \leq d$. Define $c_v^* := \min_{1 \leq j \leq d'} c(e_j)$ to be the weight of the lightest edge into node v. Let G_i be the subgraph of G that is induced by all vertices with degree at most 2^i. For each $v \in G_i$, let C_i^v be the set of nodes $u \in G_i$ that are connected to v by a simple directed path $\langle u, \ldots, v \rangle$ in G_i of total weight at most 2^{-i}. Finally, let $D(G_i)$ denote the maximum size (i.e., number of edges) of any simple path in G_i with total weight at most 2^{-i}.* **SP-C** *performs at most*

$$S_N(v) := \min\left\{ \lceil \log_2 n \rceil + 1, \min_{k \geq 0}\left\{ k : c_v^* \geq \frac{1}{2^k \cdot d} \right\} \right\} \text{ nonlocal rescans of node } v \tag{3.6}$$

and at most

$$S_L(v) := \sum_{i=\lceil \log_2 d \rceil}^{\lceil \log_2 n \rceil + 1} |C_i^v| \text{ local rescans of node } v. \tag{3.7}$$

For each current bucket of width 2^{-i} there are at most $D(G_i) + 1$ regular scan phases.

Proof: By Invariant 2, the very first regular scan of node v (if any) takes place when v resides in a current bucket B^{i_0} having bucket width $\Delta_{i_0} \leq 2^{-\lceil \log_2 d \rceil}$. We first discuss nonlocal rescans:

Node v can only be rescanned from another current bucket after v moved upwards in the bucket hierarchy. As there are at most $\lceil \log_2 n \rceil + 2$ levels, this can happen at most $\lceil \log_2 n \rceil + 1$ times. The new current buckets cover smaller and smaller ranges of tentative distances: the k-th nonlocal rescan of v implies that v was previously (re-) scanned from some current bucket $B^{i_{k-1}}$ having bucket width $\Delta_{i_{k-1}} \leq 2^{-\lceil \log_2 d \rceil - k + 1}$ whereas it is now rescanned from a new current bucket $B^{i_k} = [b_{i_k}, b_{i_k} + \Delta_{i_k}) \subset B^{i_{k-1}}$ of width $\Delta_{i_k} \leq 2^{-\lceil \log_2 d \rceil - k}$ at some higher level $i_k > i_{k-1}$. As soon as v enters the current bucket B^{i_k} we have $\mathrm{tent}(v) < b_{i_k} + 2^{-\lceil \log_2 d \rceil - k}$. By then, the smallest tentative distance among all queued candidate nodes is at least b_{i_k} (Invariant 3). Therefore, if

$$c_v^* \geq \frac{1}{2^k \cdot d} \geq 2^{-\lceil \log_2 d \rceil - k} \quad \text{for some } k \geq 0 \tag{3.8}$$

then $\mathrm{tent}(v) = \mathrm{dist}(v)$ according to the IN-criterion (Lemma 5); in that case, the k-th nonlocal rescan of v will be the last time that v is scanned. The bound (3.6) of the lemma follows by choosing the smallest nonnegative k satisfying the inequality (3.8).

Now we turn to the local rescans of v. Consider a current bucket $B_{\text{cur}} = [b, b + \Delta_{\text{cur}})$ from which v is rescanned. By Invariant 2, B_{cur} has bucket width $\Delta_{\text{cur}} = 2^{-i}$ for some $i \geq \lceil \log_2 d \rceil$. We shall use the notation $\mathrm{tent}^j(v)$ to denote the tentative distance of v at the beginning of phase j. Let t denote the first regular scan phase for B_{cur}. Let t' be the last phase where v is rescanned from this B_{cur}. Hence, $z := \mathrm{tent}^{t'}(v)$ constitutes the smallest tentative distance found for v before B_{cur} is split or finally becomes empty. It follows that there are at most $t' - t$ local rescans of v from B_{cur}. Furthermore, $b \leq \mathrm{dist}(v) \leq z < b + \Delta_{\text{cur}}$, and $\mathrm{tent}^j(v) > \mathrm{tent}^{t'}(v)$ for all $j < t'$.

Let $V_S \subseteq V$ denote the set of nodes that were scanned and settled before phase t; note that V_S contains all nodes with $\mathrm{dist}(\cdot) < b$. Finally, let $V_B \subseteq V \setminus V_S$ be the set of nodes contained in B_{cur} when the phase t starts. All nodes in V_B have degree at most 2^i, i.e., they belong to G_i. Furthermore, as long as the algorithm scans nodes and B_{cur} is neither split nor emptied, the algorithm keeps exploring paths in G_i, starting from the nodes in V_B.

Let us fix a path $P = \langle v_t, v_{t+1}, \ldots, v_{t'} = v \rangle$ whose exploration causes the last rescan of v from B_{cur} in phase t' with tentative distance z: node v_j, $t \leq j \leq t'$, is scanned in phase j; therefore, $\mathrm{tent}^{j+1}(v_{j+1}) \leq \mathrm{tent}^j(v_j) + c(v_j, v_{j+1})$. On the other hand, $\mathrm{tent}^{j+1}(v_{j+1}) < \mathrm{tent}^j(v_j) + c(v_j, v_{j+1})$ would imply that there is another path from V_B to v in G_i that results in $\mathrm{tent}^{t'}(v) < z$; this contradicts the choice of P. Thus, $\mathrm{tent}^{j+1}(v_{j+1}) = \mathrm{tent}^j(v_j) + c(v_j, v_{j+1})$. We conclude $c(P) = \mathrm{tent}^{t'}(v) - \mathrm{tent}^t(v_t) < b + \Delta_{\text{cur}} - b = 2^{-i}$. In fact, P must be a simple path in G_i; otherwise – due to nonnegative edge weights – v would already have been scanned in some phase $j < t'$ with $\mathrm{tent}^j(v) = z$, thus contradicting our observation that $\mathrm{tent}^j(v) > \mathrm{tent}^{t'}(v)$. Therefore, v can be reached from the nodes $v_t, \ldots, v_{t'-1}$ in G_i along paths of weight less than 2^{-i}, i.e., $v_t, \ldots v_{t'-1} \in C_i^v$. Consequently, $|C_i^v| \geq t' - t$. Since there are at most $t' - t$ local rescans of v from B_{cur} their number is bounded by $|C_i^v|$.

The discussion above easily implies that k subsequent regular scan phases for B_{cur} require at least one simple path in G_i having $k-1$ edges and total weight at most 2^{-i}. Hence, there are at most $D(G_i)+1$ regular scan phases for this current bucket.

After having dealt with local rescans of node v from one particular current bucket of width 2^{-i}, we can easily bound the total number of local rescans of node v: the first local rescan of v (if any) takes place in a current bucket of width 2^{-i_0}, where $i_0 \geq \lceil \log_2 d \rceil$. Further local rescans may happen for other current buckets of width 2^{-i_1}, 2^{-i_2}, ..., where $i_0 < i_1 < i_2 < \ldots$; hence, there are at most $\sum_{i=\lceil \log_2 d \rceil}^{\lceil \log_2 n \rceil + 1} |C_i^v|$ local rescans of v in total. □

Using Lemma 11 (for the number of lift operations), Lemma 12 and Remark 3 (for the costs of scan operations), and Lemma 14 (for the number of scan operations), we can restate the worst-case running time of **SP-C** as follows:

Corollary 4 *Let $L(v)$, $R(v,w)$, $S_N(v)$, and $S_L(v)$ be defined as in Lemmas 11, 12, 14 and 14, respectively.* **SP-C** *runs in time*

$$\mathcal{O}\left(n + m + \sum_{v \in V}\left(L(v) + \Big(1 + S_N(v) + S_L(v)\Big) \cdot \Big(1 + \sum_{w \in FS(v), w \neq v} R(v,w)\Big)\right)\right). \tag{3.9}$$

3.6.2 Average–Case Complexity of SP-C

This section serves to establish an average-case bound on the running time of **SP-C** under the assumption of independent random edge weights that are uniformly drawn from $[0,1]$. According to Corollary 4 it is sufficient to find an upper bound for

$$\mathbf{E}\left[n + m + \sum_{v \in V}\left(L(v) + \Big(1 + S_N(v) + S_L(v)\Big) \cdot \Big(1 + \sum_{w \in FS(v), w \neq v} R(v,w)\Big)\right)\right].$$

Due to the linearity of expectation we can concentrate on the terms $\mathbf{E}[L(v)]$ and

$$\mathbf{E}\left[\Big(1 + S_N(v) + S_L(v)\Big) \cdot \Big(1 + \sum_{w \in FS(v), w \neq v} R(v,w)\Big)\right]. \tag{3.10}$$

Recall from Lemma 14 that the values of $S_N(v)$ and $S_L(v)$ depend on the weights of edges and simple paths into node v; they do *not* depend on the weights of any edge (v,w), $w \neq v$, in the forward star of v. On the other hand, $R(v,w)$ solely depends on the weight of the edge (v,w); see Lemma 12. Hence, as all edge weights are independent, the random variables $(1 + S_N(v) + S_L(v))$ and $\Big(1 + \sum_{w \in FS(v), w \neq v} R(v,w)\Big)$ are independent, too. Consequently, the average-case running time for **SP-C** is bounded from above by

$$\mathcal{O}\left(n + m + \sum_{v \in V}\left(\mathbf{E}[L(v)] + \Big(1 + \mathbf{E}[S_N(v)] + \mathbf{E}[S_L(v)]\Big) \cdot \Big(1 + \sum_{w \in FS(v), w \neq v} \mathbf{E}[R(v,w)]\Big)\right)\right) \tag{3.11}$$

Now we will show the following inequalities: For any node $v \in V$ and any edge $(v,w) \in E$ we have

$$\begin{array}{lcll} \mathbf{E}[L(v)] & \leq & 2 \cdot \text{degree}(v) & \text{(Lemma 15)} \\ \mathbf{E}[R(v,w)] & \leq & 3 & \text{(Lemma 15)} \\ \mathbf{E}[S_N(v)] & \leq & 4 & \text{(Lemma 16)} \\ \mathbf{E}[S_L(v)] & \leq & 2 \cdot e & \text{(Lemma 18).} \end{array}$$

After inserting these inequalities in (3.11) we immediately obtain

Theorem 5 *On arbitrary directed networks with random edge weights that are independent and uniformly drawn from* $[0,1]$, **SP-C** *runs in* $\mathcal{O}(n+m)$ *average-case time.*

In the remainder we prove the inequalities mentioned above.

Lemma 15 *Let* $L(v)$ *and* $R(v,w)$ *be defined as in Lemmas 11 and 12, respectively. For any node* $v \in V$ *and any edge* $(v,w) \in E$ *we have* $\mathbf{E}[L(v)] \leq 2 \cdot \text{degree}(v)$ *and* $\mathbf{E}[R(v,w)] \leq 3$.

Proof: For $L(v)$, consider a reachable node $v \in V$ with in-degree d', $1 \leq d' \leq \text{degree}(v)$. Let $e_1 = (u_1, v), \ldots, e_{d'} = (u_{d'}, v)$ denote its d' incoming edges. $L(v)$ is bounded from above by $\sum_{1 \leq j \leq d'} \left\lceil \log_2 \frac{1}{c(e_j)} \right\rceil$. By Inequality (3.4), we have

$$\mathbf{E}\left[\left\lceil \log_2 \frac{1}{c(e)} \right\rceil\right] = \sum_{i=1}^{\infty} i \cdot \mathbf{P}\left[\left\lceil \log_2 \frac{1}{c(e)} \right\rceil = i\right] = \sum_{i=1}^{\infty} i \cdot 2^{-i} = 2$$

for any edge $e \in E$. Hence, $\mathbf{E}[L(v)] \leq 2 \cdot d' \leq 2 \cdot \text{degree}(v)$. Similarly, $R(v,w) \leq \left\lceil \log_2 \frac{1}{c(v,w)} \right\rceil + 1$ for any edge $(v,w) \in E$. Thus, $\mathbf{E}[R(v,w)] \leq 2 + 1 = 3$. □

Lemma 16 *For any node* $v \in V$*, let* $S_N(v)$ *be defined as in Lemma 14; then* $\mathbf{E}[S_N(v)] \leq 4$.

Proof: Let v have degree $d \geq 1$ and in-degree d', $1 \leq d' \leq d$. Let $c_v^* := \min_{1 \leq j \leq d'} c(e_j)$ be the weight of the lightest edge into node v. From the definition of $S_N(v)$ we easily derive

$$S_N(v) \leq \min_{k \geq 0} \left\{ k : c_v^* \geq \frac{1}{2^k \cdot d} \right\}.$$

Due to the uniform edge weight distribution we have

$$\mathbf{P}\left[c_v^* \geq \frac{1}{2^k \cdot d}\right] \geq 1 - \sum_{j=1}^{d'} \mathbf{P}\left[c(e_j) < \frac{1}{2^k \cdot d}\right] = 1 - d' \cdot \frac{1}{2^k \cdot d} \geq 1 - 2^{-k}.$$

Thus, $\mathbf{P}[S_N(v) \leq k] \geq 1 - 2^{-k}$ and $\mathbf{P}[S_N(v) \geq k+1] \leq 2^{-k}$. Therefore,

$$\begin{aligned} \mathbf{E}[S_N(v)] &\leq \sum_{i=1}^{\infty} \mathbf{P}[S_N(v) = i] \cdot i \leq \sum_{i=1}^{\infty} \mathbf{P}[S_N(v) \geq i] \cdot i \\ &\leq \sum_{i=1}^{\infty} 2^{-i+1} \cdot i = 2 \cdot \sum_{i=1}^{\infty} 2^{-i} \cdot i = 4. \end{aligned}$$

□

Before we show the last missing inequality, $\mathbf{E}[S_L(v)] \leq 2 \cdot e$, we first prove that long paths with small total weight are unlikely for random edge weights. This observation will be used to bound $\mathbf{E}[S_L(v)]$ in Lemma 18. In the following, a path without repeated edges and total weight at most Δ will be called a Δ-*path*.

Lemma 17 *Let $\mathcal{P}$ be a path of l non-repeated edges with independent and uniformly distributed edge weights in $[0,1]$. The probability that $\mathcal{P}$ is a Δ-path equals $\Delta^l/l!$ 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: $\mathbf{P}\left[X_1 \leq \Delta\right] = \Delta$. 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 of at most Δ, we have to consider all combinations for $0 \leq z \leq \Delta \leq 1$ so that the first part of l edges has weight at most $\Delta - z$ and the second part (one edge) has weight z. Thus,
$\mathbf{P}\left[\sum_{i=1}^{l+1} X_i \leq \Delta\right] = \int_0^{\Delta} \mathbf{P}\left[\sum_{i=1}^{l} X_i \leq \Delta - z\right] dz = \int_0^{\Delta} \frac{(\Delta-z)^l}{l!}\, dz = \frac{\Delta^{l+1}}{(l+1)!}$ □

Lemma 18 *Let $S_L(v)$ be defined as in Lemma 14. For any node $v \in V$, $\mathbf{E}[S_L(v)] \leq 2 \cdot e$.*

Proof: Recall that G_i denotes the subgraph of G that is induced by all vertices with degree at most 2^i. Furthermore, for any $v \in G_i$, C_i^v denotes the set of nodes $u \in G_i$ that are connected to v by a simple directed path $\langle u, \ldots, v\rangle$ in G_i of total weight at most 2^{-i}. For any node v with degree d, $S_L(v) := \sum_{i=\lceil \log_2 d\rceil}^{\lceil \log_2 n\rceil+1} |C_i^v|$.

Let us define $\mathcal{P}_i^v$ to be the set of all simple (2^{-i})-paths into v in G_i. Obviously, $|C_i^v| \leq |\mathcal{P}_i^v|$. Hence,

$$\mathbf{E}[S_L(v)] \leq \sum_{i=\lceil \log_2 d\rceil}^{\lceil \log_2 n\rceil+1} \mathbf{E}\left[|\mathcal{P}_i^v|\right].$$

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 (Lemma 17), we find

$$\begin{aligned}
\mathbf{E}[S_L(v)] &\leq \sum_{i=\lceil \log_2 d\rceil}^{\lceil \log_2 n\rceil+1} \sum_{l=1}^{\infty} d \cdot 2^{i\cdot(l-1)} \cdot 2^{-i\cdot l}/l! \\
&= \sum_{i=\lceil \log_2 d\rceil}^{\lceil \log_2 n\rceil+1} \frac{d}{2^i} \cdot \left(\sum_{l=1}^{\infty} \frac{1}{l!}\right) \leq \sum_{i=0}^{\infty} 2^{-i} \cdot e = 2 \cdot e\,. \qquad (3.12)
\end{aligned}$$

□

Hence, by now we have provided the missing pieces in order to show that **SP-C** runs in linear time on the average (Theorem 5).

As already mentioned in Section 3.5.2 for **SP-S**, **SP-C** can keep track of the actual resource usage: after $\Theta(n \cdot \log n + m)$ operations it may abort and start the computation from scratch with Dijkstra's algorithm. Thus, the worst-case total execution time is limited by $\mathcal{O}(n \log n + m)$ while the linear average-case bound is still preserved.

Unfortunately, **SP-C** is less reliable than its label-setting counterpart **SP-S**; the linear-time bound is not obtained with high probability. Why? In its current form – even though the total number of node scans is bounded by $(1 + 4 + 2 \cdot e) \cdot n < 11 \cdot n$ on average (Lemmas 16 and 18) – **SP-C** may scan some nodes many more than 11 times with non-negligible probability. If such a node v has out-degree $\Theta(n)$ then **SP-C** may perform a superlinear number of operations: each time v is scanned, all its outgoing edges are relaxed. In Section 3.7 we give the modifications for **SP-C** to cope with that problem; Section 3.8 provides the adapted analysis for the high-probability bounds.

3.7 Making SP-C More Stable

Our modified version of **SP-C**, which we will call **SP-C***, applies deferred edge relaxations: when a node v is scanned from a current bucket B_{cur} of width Δ_{cur}, then only those edges out of v that have weight at most Δ_{cur} are relaxed immediately. The remaining edges of larger weight are relaxed once and for all after $\text{tent}(v)$ is guaranteed to be final.

The rationale behind this modification is the following: due to the correlation of bucket widths and maximum node degrees it turns out that as long as $\text{tent}(v)$ is not yet final, each (re-)scan of v in **SP-C*** relaxes at most a constant number of v's outgoing edges in expectation and not much more with high probability. Therefore, even if some nodes may be rescanned much more often than the average, it becomes very unlikely that these nodes cause a superlinear number of edge relaxations.

Furthermore, we describe an alternative for the simple bottom-up search in order to find target buckets in constant worst-case time. The fast relaxation procedure is not crucial to obtain the high-probability bound for **SP-C***. However, being allowed to assume that any (re-)relaxation can be done in constant worst-case time somewhat simplifies our analysis. Subsequently, we describe our modifications in more detail:

1. For each node $v \in V$, **SP-C*** creates a list $\tilde{O}_v$ that initially keeps all outgoing edges of v in *grouped order* according to their weights as follows:

   - $\tilde{O}_v$ consists of at most $\lceil \log_2 2n \rceil + 1$ groups;
   - group g_i, $0 \leq i < \lceil \log_2 2n \rceil$, in $\tilde{O}_v$ holds all outgoing edges (v, w) with $2^{-i-1} < c(v, w) \leq 2^{-i}$ in arbitrary order;
   - $g_{\lceil \log_2 2n \rceil}$ keeps all edges of $\tilde{O}_v$ with weight at most $2^{-\lceil \log_2 2n \rceil}$;
   - for all $j < i$, g_i appears before g_j in $\tilde{O}_v$.

   The grouped order can be established in linear time by integer-sorting for small values: edge weights $c(e) > 2^{-\lceil \log_2 2n \rceil}$ are mapped onto $\lfloor \log_2 1/c(e) \rfloor$, thus creating

integers from 0 to $\lceil \log_2 2n \rceil - 1$; smaller edge weights are mapped onto the integer $\lceil \log_2 2n \rceil$. After an integer-sorting by non-increasing order the largest integers (which account for the smallest edge weights) appear first.

2. Each bucket $B_{i,j}$ is equipped with an additional edge list $E_{i,j}$. Whenever a node v is deleted from $B_{\text{cur}} = B_{i,j}$ (by either scanning or lifting), **SP-C*** moves all remaining edges $e \in \tilde{O}_v$ satisfying $c(e) > \Delta_{\text{cur}}$ from $\tilde{O}_v$ to $E_{i,j}$. Note that these edges constitute the last groups in $\tilde{O}_v$. Hence, if v is *rescanned* from B_{cur} no transferable edges are left in $\tilde{O}_v$; this can be checked in constant time.

3. All edges in $E_{i,j}$ are relaxed as soon as $B_{\text{cur}} = B_{i,j}$ is empty after a phase or – in case $B_{i,j}$ has been split – after its refining level $i+1$ is removed. This is sufficiently early: we have $c(e) \geq \Delta_i$ for any edge $e = (v, w) \in E_{i,j}$; hence, if $\text{dist}(w)$ belongs to the range of $B_{i,j}$, then $\text{dist}(v) + c(v, w) > \text{dist}(w)$ since $\text{dist}(v)$ lies in the range of $B_{i,j}$ as well; in that case a relaxation of (v, w) is not needed at all. On the other hand, if $\text{dist}(w)$ is too large to be covered by $B_{i,j}$, then $(v, w) \in E_{i,j}$ is relaxed before w will be scanned in its respective bucket. By the time $(v, w) \in E_{i,j}$ is relaxed, we have $\text{tent}(v) = \text{dist}(v)$ (this follows from the monotonicity property).

The total number of operations needed to perform the initial grouping and to maintain the additional edge lists is bounded by $\mathcal{O}(n + m)$.

3.7.1 Performing Relaxations in Constant Time

In the following we describe a method to perform edge relaxations in constant worst-case time. Using this guided search routine instead of the simple bottom-up search improves the total worst-case time of **SP-C** and **SP-C*** to $\mathcal{O}(n \cdot m)$. Additionally, it simplifies our analysis in Section 3.8. Therefore, from now on we will assume that **SP-C*** applies this worst-case efficient method.

Lemma 19 *The target bucket search for an edge relaxation can be performed in constant worst-case time.*

Proof: For the relaxation of an edge $e = (u, v)$, where u is in the current bucket $B_{\text{cur}} = B_{i,j}$ with bucket width Δ_i of the topmost level i, the algorithm has to find the highest level $r \leq i$ that can potentially contain the non-split target bucket for v. We distinguish two cases: $c(e) \leq \Delta_{i-1}$ and $c(e) > \Delta_{i-1}$.

Firstly, if $c(e) \leq \Delta_{i-1}$ (recall that Δ_{i-1} is the total level width of level i) then the target bucket of v is either located in level i, hence $r = i$, or it is the immediate successor bucket $\vec{B}_i$ of the rightmost bucket in level i. Note that $\vec{B}_i$ is located in some level below i having total level width at least Δ_{i-1}. Upon creation of the new topmost level i, a pointer to $\vec{B}_i$ can be maintained as follows: if level i is a refinement of a bucket $B_{i-1,j}$ that is not the rightmost bucket in level $i-1$, then $\vec{B}_i = B_{i-1,j+1}$, otherwise $\vec{B}_i = \vec{B}_{i-1}$; see Figure 3.6 for an example.

Secondly, consider the case $c(e) > \Delta_{i-1}$ where the target bucket is definitely located in some array below L_i. Let $x_e = \lfloor \log_2 1/c(e) \rfloor$. For each level k, $k \geq 1$, 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

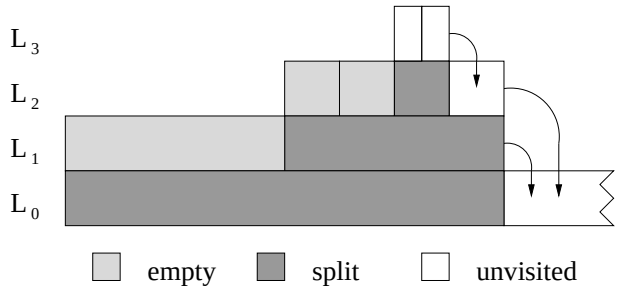


Figure 3.6: Example for pointers to the immediate successor buckets.

level r potentially containing the target bucket of v for a relaxation of $e = (u, v)$ with $u \in B_{\text{cur}} = B_{i,j}$. As before, if v does not belong to level r, then it will fit into the immediate successor bucket $\vec{B}_r$ of level r. The correctness follows inductively from the construction:

$S_1[\cdot]$ is an empty array. Now we show how $S_i[\cdot]$, $i \geq 2$, can be created from $S_{i-1}[\cdot]$. The highest possible level for a target bucket of an edge $e = (u, v)$ with $c(e) \geq \Delta_{i-2}$ is located in some level $r < i - 1$, independent of the position of u in level $i - 1$. As before, this is an immediate consequence of the fact that the total width of level $i - 1$ is given by $\Delta_{i-2} \leq c(e)$. By induction, the proper search level r can be identified in $\mathcal{O}(1)$ time by looking up $S_{i-1}[\lfloor \log_2 1/c(e) \rfloor] = r$. The new level i is just a refinement of some bucket from level $i - 1$. Hence, for $c(e) > \Delta_{i-2}$, all targets are still located in the same level $r < i - 1$ as before. Consequently, we have $S_i[k] = S_{i-1}[k]$ for $0 \leq k < \log_2 1/\Delta_{i-2}$. In other words, these entries can simply be copied from $S_{i-1}[\cdot]$ to $S_i[\cdot]$.

New entries are needed for $S_i[\log_2 1/\Delta_{i-2}]$, ..., $S_i[\log_2 1/\Delta_{i-1} - 1]$, accounting for edge weights $\Delta_{i-1} < c(e) \leq \Delta_{i-2}$. These weights are too large for a target within level i (which has total level width Δ_{i-1}) but the target may be located in the array L_{i-1}; therefore, $S_i[\log_2 1/\Delta_{i-2}] = i - 1, \ldots, S_i[\log_2 1/\Delta_{i-1} - 1] = i - 1$. 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}, $\vec{B}_{i-1}$, because $\vec{B}_{i-1}$ has width at least $\Delta_{i-2} \geq c(e)$.

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 to set the pointer to $\vec{B}_{\text{cur}}$ are bounded by $\mathcal{O}(\log(\text{degree}(v))$, where v is a node that caused the creation of level i by a bucket splitting. As each node is responsible for at most one splitting, the worst-case total costs to maintain the extra data structures are bounded by $\mathcal{O}(n + m)$. □

3.8 A High–Probability Bound for SP-C*

In this section we first give a revised worst-case bound for the running time of **SP-C***. Then we prove linear average-case time for **SP-C*** using this revised bound. After some technical lemmas we can identify an event $\mathcal{E}(\kappa)$, which holds with high probability. The event $\mathcal{E}(\kappa)$ is used to show that the revised worst-case bound for the running time of **SP-C*** satisfies the so-called *Bounded Martingale Difference Condition* with suitable parameters for nearly all instantiations of the random edge weights. At that point the high-probability bound for the running time of **SP-C*** follows easily from a generalization of the Chernoff bounds.

3.8.1 A Revised Worst-Case Bound for SP-C*

Lemma 20 *Let $L(v)$, $S_N(v)$, and $S_L(v)$ be defined as in Lemmas 11, and 14, respectively. Furthermore, for every $v \in V$ define*

$$R^*(v) := \left|\left\{(v,w) \in E \;:\; v \neq w,\; c(v,w) \leq 2^{-\lceil \log_2 \text{degree}(v) \rceil}\right\}\right|.$$

SP-C* *runs in time*

$$\mathcal{O}\left(n + m + \sum_{v \in V}\left(L(v) + \Big(S_N(v) + S_L(v)\Big) \cdot \Big(1 + R^*(v)\Big)\right)\right). \tag{3.13}$$

Proof: We only elucidate the differences as compared to **SP-C** (Corollary 4). By Section 3.7.1, each relaxation can be performed in constant time. Hence, scanning each node and relaxing *all* its outgoing edges *once* can be done in $\mathcal{O}(n+m)$ time in total. As long as $\text{tent}(v) > \text{dist}(v)$, each scan of v relaxes at most $R^*(v)$ edges out of v. This can happen at most $(S_N(v) + S_L(v))$ times. Each such non-final scan of node v requires at most $\mathcal{O}(1 + R^*(v))$ operations. The total number of additional operations needed to maintain the new data structures is bounded from above by $\mathcal{O}(n+m)$. □

Based on (3.13) we will prove the average-case running time of **SP-C***:

Lemma 21 *On arbitrary directed networks with random edge weights that are independent and uniformly drawn from $[0,1]$,* **SP-C*** *runs in $\mathcal{O}(n+m)$ average-case time.*

Proof: According to Lemma 20 we can concentrate on the expected value of the function

$$f := \sum_{v \in V}\left(L(v) + \Big(S_N(v) + S_L(v)\Big) \cdot \Big(1 + R^*(v)\Big)\right). \tag{3.14}$$

As already discussed in Section 3.6.2, the random variable $(S_N(v)+S_L(v))$ does not depend on the weights of non-self-loop edges out of node v. On the other hand, $R^*(v)$ only depends on these edge weights. Hence, $(S_N(v) + S_L(v))$ and $R^*(v)$ are independent. Therefore,

$$\mathbf{E}[f] = \sum_{v \in V}\left(\mathbf{E}[L(v)] + \Big(\mathbf{E}[S_N(v)] + \mathbf{E}[S_L(v)]\Big) \cdot \Big(1 + \mathbf{E}[R^*(v)]\Big)\right).$$

We already showed before that $\mathbf{E}[L(v)] \leq 2 \cdot \text{degree}(v)$, $\mathbf{E}[S_N(v)] \leq 4$, and $E[S_L(v)] \leq 2 \cdot e$; see Lemmas 15, 16, and 18, respectively. Finally,

$$\mathbf{E}[R^*(v)] \leq \text{degree}(v) \cdot 2^{-\lceil \log_2 \text{degree}(v) \rceil} \leq 1$$

for any $v \in V$. Hence,

$$\mathbf{E}[f] \leq \sum_{v \in V}\Big(2 \cdot \text{degree}(v) + (4 + 2 \cdot e) \cdot 2\Big) \leq 4 \cdot m + 20 \cdot n. \tag{3.15}$$

□

Later on, we will also need a simple upper bound for the function f of (3.14). Using

$$\begin{array}{lcll} L(v) & \leq & \lceil \log_2 n \rceil + 1 & \text{(by Lemmas 10 and 11),} \\ S_N(v) & \leq & \lceil \log_2 n \rceil + 1 & \text{(by Lemma 14),} \\ S_L(v) & \leq & (\lceil \log_2 n \rceil + 2) \cdot n & \text{(by Lemma 14),} \\ R^*(v) & \leq & n & \text{(by Lemma 20)} \end{array}$$

we obtain,

$$f = \sum_{v \in V} \Big(L(v) + \Big(S_N(v) + S_L(v) \Big) \cdot \Big(1 + R^*(v) \Big) \Big) \leq n^4 \quad \text{for } n \geq 32. \tag{3.16}$$

3.8.2 Some Observations for Random Edge Weights

In the following we will define some event $\mathcal{Q}(\kappa)$ and provide a lower bound on its probability.

Lemma 22 *Let $\kappa \geq 1$ be an arbitrary constant. Let $\mathcal{Q}(\kappa)$ stand for the event that for all $0 \leq i \leq \lceil \log_2 n \rceil + 1$ it is true that*

(a) for each node in G_i the number of incoming and outgoing edges with weight at most 2^{-i} is bounded by $d^*_\kappa = (3 \cdot \kappa + 7) \cdot \ln n$.

(b) each simple (2^{-i})-path in G_i contains fewer than $l^*_\kappa = \frac{2 \cdot e \cdot (\kappa+2) \cdot \ln n}{\ln \ln n}$ edges.

$\mathcal{Q}(\kappa)$ holds with probability at least $1 - 1/2 \cdot n^{-\kappa}$ if $n \geq 32$.

Proof: Let us begin with (a): consider a node $v \in G_i$ with degree k, $1 \leq k \leq 2^i$. Let X_j denote the binary random variable that is one if the j-th edge of v has weight at most 2^{-i}, and zero otherwise. We have $\mu := \mathbf{E}[\sum_j X_j] = k \cdot 2^{-i} \leq 1$. Due to independent edge weights, we can use the Chernoff bound (2.2) of Lemma 7 with $\delta := \frac{(3 \cdot \kappa + 6) \cdot \ln n}{\mu}$ in order to show

$$\mathbf{P} \left[\sum_{j=1}^{k} X_j \geq (3 \cdot \kappa + 7) \cdot \ln n \right] \leq e^{-\delta \cdot \mu / 3} \leq n^{-\kappa - 2} .$$

By Boole's inequality, (a) holds with probability at least

$$1 - n \cdot (\lceil \log_2 n \rceil + 2) \cdot n^{-\kappa - 2} \geq 1 - 1/4 \cdot n^{-\kappa} \text{ for } n \geq 32.$$

Now we turn to part (b): there are at most $2^{i \cdot l}$ simple paths of size l into each node of G_i. By Lemma 17, the probability that a given path of l independent edges has total weight at most 2^{-i} is bounded by $\frac{(2^{-i})^l}{l!}$. Hence, there is at least one simple (2^{-i})-path with l edges in G_i with probability at most $n \cdot 2^{i \cdot l} \cdot \frac{2^{-i \cdot l}}{l!} = \frac{n}{l!}$. Therefore, by Boole's inequality there is at least one simple (2^{-i})-path of $l \geq l^*_\kappa$ edges in any G_i, $0 \leq i \leq \lceil \log_2 n \rceil + 1$, with probability at most

$$\sum_{l \geq l^*_\kappa} \frac{n \cdot (\lceil \log_2 n \rceil + 2)}{l!} \quad \leq \quad \frac{n \cdot (\lceil \log_2 n \rceil + 2)}{l^*_\kappa !} \cdot \sum_{l \geq 0} \frac{1}{l!}$$

$$\begin{aligned}
&\leq \frac{n \cdot (\lceil \log_2 n \rceil + 2)}{(\frac{l^*_\kappa}{e})^{l^*_\kappa} \cdot \sqrt{2 \cdot \pi \cdot l^*_\kappa}} \cdot e \quad \text{by the Stirling Approximation} \\
&\leq \frac{n^2}{4 \cdot (\frac{l^*_\kappa}{e})^{l^*_\kappa / e}} \quad \text{for } n \geq 32 \\
&\leq \frac{n^{-\kappa}}{4} \, .
\end{aligned}$$

For the last inequality consider an upper bound for l^*_κ in

$$x^x = y \quad \Leftrightarrow \quad x \cdot \ln x = \ln y \quad \text{where } y := n^{\kappa+2} \geq e^e \text{ and } x := \tfrac{l^*_\kappa}{e} : \tag{3.17}$$

Since $x \geq e$ we have $\ln x \geq 1$ and therefore $x \leq \ln y$. Reinserting the last inequality in (3.17) yields the lower bound $x \geq \frac{\ln y}{\ln \ln y}$. If we repeat the reinsertion trick once more using the lower bound just obtained, then we find $x \leq \frac{\ln y}{\ln \ln y - \ln \ln \ln y} \leq \frac{2 \cdot \ln y}{\ln \ln y} \leq \frac{2 \cdot (\kappa+2) \ln n}{\ln \ln n}$. Hence, the choice of $l^*_\kappa = \frac{2 \cdot e \cdot (\kappa+2) \cdot \ln n}{\ln \ln n}$ ensures $x^x \geq n^{\kappa+2}$. Therefore, (b) holds with probability at least $1 - 1/4 \cdot n^{-\kappa}$.

Combining the bounds for (a) and (b) by Boole's inequality we find that $\mathcal{Q}(\kappa)$ holds with probability at least $1 - 2 \cdot 1/4 \cdot n^{-\kappa} = 1 - 1/2 \cdot n^{-\kappa}$. □

Using $\mathcal{Q}(\kappa)$, we can also prove high-probability bounds on the sizes of the sets C_i^v (which, in turn, define $S_L(v)$; see Lemma 14).

Lemma 23 *Let v be an arbitrary node in G_i. Let C_i^v be defined as in Lemma 14. Let $\widehat{C}_i^v$ be the set of nodes $u \in G_i$ that are connected to v by a simple directed path $\langle v, \ldots, u \rangle$ in G_i of total weight at most 2^{-i}. Let $\kappa, \epsilon \geq 1$ be arbitrary constants. If $\mathcal{Q}(\kappa)$ holds then there exist positive constants $n_0(\epsilon)$ and $\psi(\epsilon)$ so that $|C_i^v| \leq n^{\kappa \cdot \psi(\epsilon) / \ln \ln n}$ and $|\widehat{C}_i^v| \leq n^{\kappa \cdot \psi(\epsilon) / \ln \ln n}$ with probability at least $1 - n^{-\epsilon}$ for $n \geq n_0(\epsilon)$.*

Proof: Let us start with $\widehat{C}_i^v$. We argue that the number of those nodes that are reached from v via paths of length l can be bounded by the offspring in the l-th generation of the following branching process[3] $\mathbf{Z}$: An individual (a node) u has its offspring defined by the number Y_u of edges leaving u that have weight at most 2^{-i}. For independent edge weights uniformly distributed in $[0, 1]$, we find $\mathbf{E}[Y_u] \leq \text{degree}(u) \cdot 2^{-i} \leq 2^i \cdot 2^{-i} \leq 1$ since each node in G_i has degree at most 2^i. Let Z_l denote the total offspring after l generations when branching from node v (i.e., $Z_1 = Y_v$).

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 emanating from v that use only edges with weight at most 2^{-i}. After a node u is found to be the target of multiple paths out of v, the events on those paths would no longer be independent. However, all but one of the multiple paths would produce only duplicate entries into $\widehat{C}_i^v$. The additional paths are therefore discarded from the branching process. All remaining events are independent.

In the following, we consider another branching process $\mathbf{Z}'$ with identical and independent probabilities for the generation of new nodes: $Z'_0 = 1$, Z'_1 is a nonnegative integer-valued random variable, and $Z'_j = \sum_{i=1}^{Z'_{j-1}} Y'_i$ where the random variables Y'_i are distributed

[3] An introduction to branching processes can be found in [11, 79], for example.

like Z_1. Furthermore, the random variables Y_i' are independent of each other and independent of Z_{j-1}. The *branching factor* of this branching process is given by $\rho = \mathbf{E}[Z_1']$. For branching processes of this kind we have $\mathbf{E}[Z_l'] = \rho^l$. Additionally, it is shown in [91, Theorem 1] that for $\rho > 1$ and any constant $\epsilon > 1$, $Z_l' = \mathcal{O}(\rho^l \cdot \log n)$ with probability at least $1 - 1/2 \cdot n^{-\epsilon}$; the involved constants depend on ϵ. Taking $\rho = e > 1$, the random variables Z_j of the first branching process $\mathbf{Z}$ are stochastically dominated by the respective random variables Z_j' of $\mathbf{Z}'$.

In order to asymptotically bound the sum $\sum_{j \le l} Z_j'$ we can concentrate on the term Z_l' because a growing exponential sum is dominated by its last summand: $\sum_{j \le l} Z_j' = \mathcal{O}(e^l \cdot \log n)$ whp. If $\mathcal{Q}(\kappa)$ holds in addition, we have $l < l_\kappa^* = \frac{2 \cdot e \cdot (\kappa+2) \cdot \ln n}{\ln \ln n}$. In that case, the number of nodes that are reached from v via simple (2^{-i})-paths in G_i is at most $\mathcal{O}(e^{l_\kappa^*} \cdot \log n) = \mathcal{O}(n^{\frac{2 \cdot e \cdot (\kappa+2)}{\ln \ln n}} \cdot \log n)$ with probability at least $1 - 1/2 \cdot n^{-\epsilon}$.

Now we turn to C_i^v, which is symmetric to $\widehat{C}_i^v$ in the sense that it considers paths *entering* v instead of paths *leaving* v. Hence, the argument simply follows by traversing the edges during the branching process in the opposite direction. Therefore, if $\mathcal{Q}(\kappa)$ holds, there are positive constants $n_0(\epsilon)$ and $\psi(\epsilon)$ so that $|C_i^v| \le n^{\kappa \cdot \psi(\epsilon)/\ln \ln n}$ and $|\widehat{C}_i^v| \le n^{\kappa \cdot \psi(\epsilon)/\ln \ln n}$ with probability at least $1 - n^{-\epsilon}$ for $n \ge n_0(\epsilon)$. □

3.8.3 The Event $\mathcal{E}(\kappa)$ and the Method of Bounded Differences

In the next lemma we introduce the event $\mathcal{E}(\kappa)$ and give a lower bound on its probability:

Lemma 24 *Let $\kappa \ge 1$ be an arbitrary constant. Let $n_0(\kappa+2)$ and $\psi(\kappa+2)$ be the constants of Lemma 23 with $\epsilon = \kappa + 2$. We define $\mathcal{E}(\kappa)$ to be the event that*

(a) $\mathcal{Q}(\kappa)$ holds.

(b) $|C_i^v| \le n^{\kappa \cdot \psi(\kappa+2)/\ln \ln n}$ and $|\widehat{C}_i^v| \le n^{\kappa \cdot \psi(\kappa+2)/\ln \ln n}$ for all $0 \le i \le \lceil \log_2 n \rceil + 1$ and all $v \in G_i$.

Then $\mathcal{E}(\kappa)$ holds with probability at least $1 - n^{-\kappa}$ for any $n \ge \max\{n_0(\kappa+2),\ 32\}$.

Proof: By Lemma 22, (a) holds with probability at least $1 - 1/2 \cdot n^{-\kappa}$ for $n \ge 32$. Using Boole's inequality, (a) and (b) hold with probability at least $1 - 1/2 \cdot n^{-\kappa} - n \cdot (\lceil \log_2 n \rceil + 2) \cdot n^{-\kappa-2} \ge 1 - n^{-\kappa}$ for $n \ge \max\{n_0(\kappa+2),\ 32\}$. □

For technical reasons we will also need the events $\mathcal{E}_j(\kappa)$ and $\mathcal{E}^{m-j}(\kappa)$: let $X_l \in [0,1]$ denote the random variable associated with the weight of the l-th edge of the input graph for $1 \le l \le m$; then $\mathcal{E}_j(\kappa)$ stands for the event that $\mathcal{E}(\kappa)$ holds if we just consider the edge weights associated with $X_1, \ldots X_j$. Equivalently, $\mathcal{E}^{m-j}(\kappa)$ denotes the event that $\mathcal{E}(\kappa)$ holds if we restrict attention to the $m - j$ random variables $X_{j+1}, \ldots X_m$. Clearly, $\mathcal{E}(\kappa)$ implies $\mathcal{E}_j(\kappa) \wedge \mathcal{E}^{m-j}(\kappa)$ for all $1 \le j \le m$. Therefore, $\mathbf{P}[\mathcal{E}_j(\kappa)], \mathbf{P}[\mathcal{E}_{m-j}(\kappa)] \ge \mathbf{P}[\mathcal{E}(\kappa)] \ge 1 - n^{-\kappa}$. On the other hand, it does not necessarily follow from $\mathcal{E}_j(\kappa) \wedge \mathcal{E}^{m-j}(\kappa)$ that $\mathcal{E}(\kappa)$ holds as well. Still, $\mathcal{E}_j(\kappa) \wedge \mathcal{E}^{m-j}(\kappa)$ implies that some event $\mathcal{E}^*(\kappa)$ similar to $\mathcal{E}(\kappa)$ holds; it is easy to check that due to the contributions from two independent edge sets, the bounds of $\mathcal{Q}(\kappa)$ are at most doubled and as a consequence the bounds on $|C_i^v|$

and $|\widehat{C}_i^v|$ are at most squared. So, more formally, $\mathcal{E}^*(\kappa)$ stands for the event that for all $0 \leq i \leq \lceil \log_2 n \rceil + 1$ it is true that

(a) for each node in G_i the sum of incoming and outgoing edges with weight at most 2^{-i} is bounded by $2 \cdot d_\kappa^* = (6 \cdot \kappa + 14) \cdot \ln n$.

(b) each simple (2^{-i})-path in G_i contains fewer than $2 \cdot l_\kappa^* = \frac{4 \cdot e \cdot (\kappa+2) \cdot \ln n}{\ln \ln n}$ edges.

(c) $|C_i^v| \leq n^{2 \cdot \kappa \cdot \psi(\kappa+2)/\ln \ln n}$ and $|\widehat{C}_i^v| \leq n^{2 \cdot \kappa \cdot \psi(\kappa+2)/\ln \ln n}$ for all $v \in G_i$.

Similarly, $\mathcal{E}_{j_1}(\kappa) \wedge \mathcal{E}^{m-j_2}(\kappa)$ with $1 \leq j_1 < j_2 \leq m$ implies $\mathcal{E}^*(\kappa)$ if we only take into consideration the edge weights associated with $X_1, \ldots X_{j_1}$ and $X_{j_2+1}, \ldots X_m$. The events $\mathcal{E}(\kappa)$ and $\mathcal{E}^*(\kappa)$ will be used later in connection with the following tail estimate:

Lemma 25 (Method of Bounded Martingale Differences [45, 100]) *Let $\mathbf{X} = (X_1, \ldots, X_m)$ be a family of random variables with X_l taking values in the set A_l. Let $\mathbf{X}_i = \mathbf{a}_i$ be the typographical shortcut for $X_1 = a_1, \ldots, X_i = a_i$. Let $f(\mathbf{X})$ be a function that satisfies the following condition concerning $X_1, \ldots, X_m$ with parameters $t_1, \ldots t_m$: for any $\mathbf{a} = (a_1, \ldots, a_m) \in \prod_l A_l$ and any $b_i \in A_i$,*

$$g_i(\mathbf{a}) := \mid \mathbf{E}[f \mid \mathbf{X}_{i-1} = \mathbf{a}_{i-1}, X_i = b_i] - \mathbf{E}[f \mid \mathbf{X}_{i-1} = \mathbf{a}_{i-1}] \mid \leq t_i \tag{3.18}$$

for all $1 \leq i \leq m$. Then

$$\mathbf{P}[f(\mathbf{X}) > \mathbf{E}[f(\mathbf{X})] + t] \leq 2^{-\frac{t^2}{2 \cdot \sum_l t_l^2}}.$$

Moreover, let $\mathcal{Z}$ be any 'bad' subset of $\prod_l A_l$, such that $g_i(\mathbf{a}) \leq t_i$, $1 \leq i \leq m$ for each $\mathbf{a} \notin \mathcal{Z}$. Then

$$\mathbf{P}[f(\mathbf{X}) > \mathbf{E}[f(\mathbf{X})] + t] \leq 2^{-\frac{t^2}{2 \cdot \sum_l t_l^2}} + \mathbf{P}[\mathbf{X} \in \mathcal{Z}]. \tag{3.19}$$

Proof: See [100]. □

The function $g_i(\mathbf{a})$ of (3.18) measures how much the expected value of $f(\mathbf{X})$ changes if it is revealed that X_i takes the value b_i, conditioned on the fact that $X_1 = a_1, \ldots, X_{i-1} = a_{i-1}$. If the random variables $X_1, \ldots, X_m$ are *dependent* then fixing $X_i = b_i$ may influence the distribution of $X_{i+1}, \ldots, X_m$. However, if $X_1, \ldots, X_m$ are *independent* then the distribution of $X_{i+1}, \ldots, X_m$ remains the same. In that case, $g_i(\mathbf{a})$ can be rewritten as

$$\mid \mathbf{E}[f(a_1, \ldots, a_{i-1}, b_i, X_{i+1}, \ldots, X_m)] - \mathbf{E}[f(a_1, \ldots, a_{i-1}, X_i, X_{i+1}, \ldots, X_m)] \mid .$$

Hence, for independent random variables, $g_i(\mathbf{a})$ can be obtained by comparing $f(a_1, \ldots, a_{i-1}, b_i, x_{i+1}, \ldots, x_m)$ with $\mathbf{E}[f(a_1, \ldots, a_{i-1}, X_i, x_{i+1}, \ldots, x_m)]$ for each combination $(x_{i+1}, \ldots, x_m) \in \prod_{l=i+1}^m A_l$, weighted with the respective probabilities for $(x_{i+1}, \ldots, x_m)$ to occur. If we are just interested in upper bounds on $g_i(\mathbf{a})$, then it is even sufficient to bound

$$\max_{b_i, b_i' \in A_i} \left| f(a_1, \ldots, a_{i-1}, b_i, x_{i+1}, \ldots, x_m) - f(a_1, \ldots, a_{i-1}, b_i', x_{i+1}, \ldots, x_m) \right| \tag{3.20}$$

for each $(x_{i+1},\ldots,x_m) \in \prod_{l=i+1}^{m} A_l$, and then build the weighted sum over all combinations.

We shall use Lemma 25 as follows: $X_l \in [0,1]$ denotes the weight of the l-th edge of the input graph for $1 \le l \le m$; hence, the random variables $X_1,\ldots,X_m$ are independent. For f we substitute the function (3.14) concerning the running time of **SP-C***. The 'bad' subset $\mathcal{Z}$ contains those edge-weight vectors for which the event $\mathcal{E}(\kappa)$ does not hold. By Lemma 24, $\mathbf{P}[\mathcal{E}(\kappa)] \ge 1 - n^{-\kappa}$ for $n \ge \max\{n_0(\kappa+2),\ 32\}$. Thus, $\mathbf{P}[\mathbf{X} \in \mathcal{Z}] \le n^{-\kappa}$.

It remains to bound $g_i(\mathbf{a})$ for all $\mathbf{a} \notin \mathcal{Z}$ using (3.20): we know that $\mathcal{E}_{i-1}(\kappa)$ holds for the prefix $(a_1,\ldots,a_{i-1})$; if $\mathcal{E}^{m-i}(\kappa)$ also holds for the suffix $(x_i,\ldots,x_m)$ then this implies $\mathcal{E}^*(\kappa)$ concerning the first $i-1$ edges and the last $m-i$ edges of the graph. In that case (which happens for the overwhelming number of suffixes since $\mathbf{P}[\mathcal{E}^{m-i}(\kappa)] \ge 1-n^{-\kappa}$) we can use the knowledge that $\mathcal{E}^*(\kappa)$ holds in order to bound the maximum effect of arbitrarily changing the value of the i-th coordinate in (3.20). In fact, for that case we will show in the proof of Theorem 6 that (3.20) can be bounded from above by $2 \cdot n^{\beta(\kappa)/\ln\ln n}$ where $\beta(\kappa) \ge 1$ is some constant, and $n \ge \max\{n_0(\kappa+2), 32\}$.

Otherwise, i.e., in the unlikely case that $\mathcal{E}^{m-i}(\kappa)$ does not hold, then (3.20) can be trivially bounded by n^4 according to (3.16). It follows that for any $\mathbf{a} \notin \mathcal{Z}$ and all $1 \le i \le m$,

$$\begin{aligned} g_i(\mathbf{a}) &\le (1-n^{-\kappa}) \cdot 2 \cdot n^{\beta(\kappa)/\ln\ln n} + n^{4-\kappa} \\ &\le 3 \cdot n^{\beta(\kappa)/\ln\ln n} =: t_i \quad \text{for } \kappa \ge 4 \text{ and } n \ge \max\{n_0(\kappa+2),\ 32\}. \end{aligned}$$

Using Lemma 25 we find

$$\mathbf{P}[f > \mathbf{E}[f] + 5\cdot(n+m)] \le 2^{-\frac{25\cdot(n+m)^2}{18\cdot m\cdot n^{2\cdot\beta(\kappa)/\ln\ln n}}} + n^{-\kappa} \le 2^{-\frac{n}{n^{2\cdot\beta(\kappa)/\ln\ln n}}} + n^{-\kappa} \le 2\cdot n^{-\kappa}$$

if $n \ge e^{e^{4\cdot\beta(\kappa)}}$. Hence,

$$\mathbf{P}[f \le \mathbf{E}[f] + 5\cdot(n+m)] \ge 1 - 2\cdot n^{-\kappa}$$

for any constant $\kappa \ge 4$ and any $n \ge \max\{n_0(\kappa+2),\ e^{e^{4\cdot\beta(\kappa)}}\}$.

Theorem 6 *On arbitrary directed networks with random edge weights that are independent and uniformly drawn from* $[0,1]$, **SP-C*** *runs in* $\mathcal{O}(n+m)$ *time with high probability.*

Proof: According to the discussion above it is sufficient to show that (3.20) can be bounded from above by $2 \cdot n^{\beta(\kappa)/\ln\ln n}$ if the event $\mathcal{E}^*(\kappa)$ holds for all edge weights excluding $c(v',w')$, the weight of the i-th edge in the input graph. We rewrite f in the form

$$f = \sum_{v\in V} L(v) + \sum_{v\in V}\Big(S_N(v)\cdot\Big(1+R^*(v)\Big)\Big) + \sum_{v\in V}\Big(S_L(v)\cdot\Big(1+R^*(v)\Big)\Big)\,.$$

In the following, let $\Delta(Y)$ denote the maximum change of some random variable Y after a modification of $c(v',w')$.

By the definition of $L(v)$, as given in Lemma 11, modifying $c(v',w')$ does not influence $L(v)$ unless $v = w'$. Moreover, $L(w')$ is bounded by the maximum height of the bucket hierarchy. Hence, after an alteration of $c(v',w')$, the value $L(w')$ may change by at most $\lceil\log_2 n\rceil+1$. In other words, $\Delta\left(L(w')\right) \le \lceil\log_2 n\rceil+1$ and $\Delta\left(\sum_{v\in V} L(v)\right) \le \lceil\log_2 n\rceil+1$,

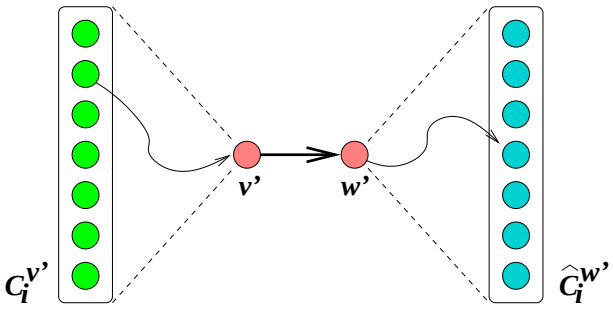
too. The arguments for $L(v)$ transfer to $S_N(v)$, which was defined in Lemma 14: $S_N(v) \leq \lceil \log_2 n \rceil + 1$; therefore, $\Delta\left(S_N(w')\right) \leq \lceil \log_2 n \rceil + 1$ whereas $\Delta\left(S_N(v)\right) = 0$ for any other node $v \neq w'$.

Let us turn to the random variable $R^*(v)$ from Lemma 20. Recall that $R^*(v)$ denotes the number of non-self-loop edges out of v that have weight at most $2^{-\lceil \log_2 \text{degree}(v) \rceil}$. As $\mathcal{E}^*(\kappa)$ holds for all edge weights excluding $c(v', w')$, we have $R^*(v) \leq 2 * d^*_\kappa = (6 \cdot \kappa + 14) \cdot \ln n$ for any $v \in V \setminus \{v'\}$ if $n \geq \max\{n_0(\kappa + 2),\ 32\}$. We find $\Delta\left(R^*(v')\right) \leq 1$: depending on $c(v', w')$, the edge (v', w') either belongs to the set of light edges leaving v' or not. Furthermore, $\Delta\left(R^*(v)\right) = 0$ for any $v \neq v'$. Using the results above we deduce

$$\begin{aligned} & \Delta\left(\sum_{v \in V}\Big(S_N(v) \cdot \Big(1 + R^*(v)\Big)\Big)\right) \\ \leq\ & S_N(v') \cdot \Delta\left(R^*(v')\right) + \Delta\left(S_N(w')\right) \cdot (1 + 2 \cdot d^*_\kappa) \\ \leq\ & (\lceil \log_2 n \rceil + 1) + (\lceil \log_2 n \rceil + 1) \cdot (1 + (6 \cdot \kappa + 14)) \cdot \ln n \leq \alpha(\kappa) \cdot \ln^2 n \end{aligned}$$

for some positive constant $\alpha(\kappa)$.

Finally, we will examine how much $\sum_{v \in V}\Big(S_L(v) \cdot (1 + R^*(v))\Big)$ may change if $c(v', w')$ is modified. Recall from Lemma 14 that $S_L(v)$ is defined to be $\sum_{i=\lceil \log_2 d \rceil}^{\lceil \log_2 n \rceil + 1} |C_i^v|$. For the edge (v', w') we consider node pairs from the sets $C_i^{v'}$ and $\widehat{C}_i^{w'}$ where i ranges from $\max\{\ \lceil \log_2\ \text{degree}(v') \rceil\ , \lceil \log_2\ \text{degree}(w') \rceil\ \}$ to $\lceil \log_2 n \rceil + 1$. The drawing below provides an example.



Any node $w \in \widehat{C}_i^{w'}$ can be reached from w' via a simple (2^{-i})-path in G_i. Put differently, $w' \in C_i^w$ for every $w \in \widehat{C}_i^{w'}$. If $c(v', w')$ is decreased, then the node v' and further nodes of $C_i^{v'}$ may enter C_i^w (if they did not belong to it before). Concretely speaking, $|C_i^w|$ may increase by at most $1 + |C_i^{v'}|$. On the other hand, if $c(v', w')$ grows then $|C_i^w|$ may shrink by at most $1 + |C_i^{v'}|$. Since $\mathcal{E}^*(\kappa)$ holds for all edges excluding (v', w'), we have $|C_i^v| \leq n^{2 \cdot \kappa \cdot \psi(\kappa+2)/\ln\ln n}$ for any $v \in G_i \setminus \{v'\}$, $i \leq \lceil \log_2 n \rceil + 1$ and $n \geq \max\{n_0(\kappa + 2),\ 32\}$. Hence,

$$\Delta\left(S_L(w)\right) \leq (\lceil \log_2 n \rceil + 2) \cdot (n^{2 \cdot \kappa \cdot \psi(\kappa+2)/\ln\ln n} + 1) \quad \text{if } w \in \widehat{C}_i^{w'} \text{ for some } G_i$$

and $\Delta\left(S_L(w)\right) = 0$ otherwise. The event $\mathcal{E}^*(\kappa)$ also implies $|\widehat{C}_i^{w'}| \leq n^{2 \cdot \kappa \cdot \psi(\kappa+2)/\ln\ln n}$. Thus,

$$\Delta\left(\sum_{v \in V} S_L(v)\right) \leq (\lceil \log_2 n \rceil + 2)^2 \cdot (n^{2 \cdot \kappa \cdot \psi(\kappa+2)/\ln\ln n} + 1)^2\ .$$

It remains to incorporate the factor $(1 + R^*(v))$:

$$\begin{aligned}
& \Delta\left(\sum_{v\in V}\left(S_L(v)\cdot\left(1+R^*(v)\right)\right)\right) \\
\leq\ & S_L(v')\cdot\Delta\left(R^*(v')\right) + \Delta\left(\sum_{v\in V} S_L(v)\right)\cdot(1+2\cdot d^*_\kappa) \\
\leq\ & (\lceil\log_2 n\rceil+2)\cdot n^{2\cdot\kappa\cdot\psi(\kappa+2)/\ln\ln n} + \\
& (\lceil\log_2 n\rceil+2)^2\cdot(n^{2\cdot\kappa\cdot\psi(\kappa+2)/\ln\ln n}+1)^2\cdot(1+(6\cdot\kappa+14))\cdot\ln n \\
\leq\ & n^{\beta(\kappa)/\ln\ln n} \quad \text{for some positive constant } \beta(\kappa)\ .
\end{aligned}$$

In fact, $\beta(\kappa)$ can be chosen in a way that

$$\begin{aligned}
\Delta(f) \ \leq\ & \Delta\left(\sum_{v\in V} L(v)\right) + \Delta\left(\sum_{v\in V}\left(S_N(v)\cdot\left(1+R^*(v)\right)\right)\right) + \\
& \Delta\left(\sum_{v\in V}\left(S_L(v)\cdot\left(1+R^*(v)\right)\right)\right) \\
\leq\ & \lceil\log_2 n\rceil + 1 + \alpha(\kappa)\cdot\ln^2 n + n^{\beta(\kappa)/\ln\ln n} \leq 2\cdot n^{\beta(\kappa)/\ln\ln n}
\end{aligned}$$

if $n \geq \max\{n_0(\kappa+2),\ 32\}$. □

3.9 Implications for the Analysis of other SSSP Algorithms

In this section we will re-use the analysis for the bucket-splitting algorithms **SP-C** and **SP-C*** (Sections 3.6 and 3.8) in order to yield average-case results for simpler SSSP algorithms. We consider the "Approximate Bucket Implementation of Dijkstra's algorithm" (ABI-Dijkstra) [25], which we already sketched in Section 3.3, and our refinement, the sequential Δ-Stepping.

3.9.1 ABI-Dijkstra and the Sequential Δ-Stepping

The Δ-Stepping algorithm maintains a one-dimensional array B of buckets where $B[i]$ stores the set $\{v \in V : v \text{ is queued and } \mathrm{tent}(v) \in [i\cdot\Delta, (i+1)\cdot\Delta)\}$. As before, the parameter Δ is a positive real number, which is also called the "bucket width". Let $c_{\max} := \max_{e\in E} c(e)$ denote the maximum edge weight. For maximum shortest-path weight $\mathcal{L} = \max\{\mathrm{dist}(s,v) \mid \mathrm{dist}(s,v) < \infty\}$, the algorithm traverses $\lceil\frac{\mathcal{L}}{\Delta}\rceil$ buckets. However, by cyclically reusing empty buckets, only space for $b_{\max} = \lceil\frac{c_{\max}}{\Delta}\rceil + 1$ buckets is needed. In that case, a node v with tentative distance $\mathrm{tent}(v)$ is kept in bucket $B[\ \lfloor\mathrm{tent}(v)/\Delta\rfloor \bmod b_{\max}\ ]$.

The Δ-Stepping approach distinguishes *light* edges and *heavy* edges: a light edge has weight at most Δ, the weight of a heavy edge is larger than Δ. In each *phase*, the algorithm scans *all* [4] nodes from the first nonempty bucket (current bucket) and relaxes all light edges

[4] Similar to the bucket-splitting approaches, the nodes of the current bucket could also be scanned one-by-one in FIFO order. However, in view of the parallelizations in Chapter 4 we consider them in phases already now.

out of these nodes. The relaxation of heavy edges is not needed at this time since they can only result in tentative distances outside of the scope of the current bucket, i.e., they will not insert nodes into the current bucket. Once the current bucket finally remains empty after a phase, all nodes in its distance range have been assigned their final distance values during the previous phase(s). Subsequently, all heavy edges emanating from these nodes are relaxed once and for all. Then the algorithm sequentially searches for the next nonempty bucket. As in the case of **SP-C** and **SP-C***, single buckets of the array can be implemented as doubly linked lists; inserting or deleting a node, finding a bucket for a given tentative distance and skipping an empty bucket can be done in constant time. A simple preprocessing can restructure the adjacency lists in order to support efficient access to the subsets of light edges.

The ABI-Dijkstra algorithm can essentially be seen as the sequential Δ-Stepping *without* distinction between light and heavy edges, i.e., all outgoing edges of a scanned node are relaxed immediately.

In the following we will transfer our analysis on the average-case number of node rescans from **SP-C** and **SP-C*** to ABI-Dijkstra and Δ-Stepping, respectively:

Theorem 7 *Consider directed graphs with n nodes, m edges and independent random edge weights uniformly distributed in $[0,1]$. Let d' and $\mathcal{L}'$ denote upper bounds[5] on the maximum node degree d and the maximum shortest-path weight $\mathcal{L}$, respectively. Choosing the bucket width as $\Delta := 2^{-\lceil \log_2 d \rceil}$, ABI-Dijkstra and Δ-Stepping run in $\mathcal{O}(n+m+\mathbf{E}[d \cdot \mathcal{L}])$ time on the average. Additionally, for $\Delta := 2^{-\lceil \log_2 d' \rceil}$, the Δ-Stepping algorithm requires $\mathcal{O}(n + m + d' \cdot \mathcal{L}')$ time with high probability. Both algorithms need $\mathcal{O}(n \cdot m)$ time in the worst case.*

Proof: We examine a reformulation of the algorithms **SP-C** and **SP-C*** with $2^{\lceil \log_2 d \rceil} + 1 \leq 2 \cdot d + 1$ initial buckets at level 0, each of them having width $\Delta_0 := 2^{-\lceil \log_2 d \rceil}$. Scanning low-degree nodes from buckets of reduced width will clearly not increase the probability of rescans. Furthermore, due to the revised choice of Δ_0 none of the initial buckets will ever be split. Hence, the bucket structure remains one-dimensional, and target-bucket searches always succeed in constant time. In other words, for the chosen bucket widths, the reformulated algorithms **SP-C** and **SP-C*** coincide with ABI-Dijkstra and Δ-Stepping, respectively.

As compared to the standard versions of **SP-C** and **SP-C***, at most $\left\lceil \frac{\mathcal{L}}{2^{-\lceil \log_2 d \rceil}} \right\rceil \leq 2 \cdot d \cdot \mathcal{L}$ extra buckets may have to be traversed. Therefore, disregarding $\mathcal{O}(\mathbf{E}[d \cdot \mathcal{L}])$ additional operations, the asymptotic average-case performances of ABI-Dijkstra and Δ-Stepping reduce to those of the standard versions **SP-C** and **SP-C***, respectively. The average-case bounds follow from Theorem 5 and Lemma 21. If $d \leq d'$ and $\mathcal{L} \leq \mathcal{L}'$ then the high-probability bound for the Δ-Stepping is inherited from Theorem 6.

Using $d \cdot \mathcal{L} = \mathcal{O}(n \cdot m)$ for arbitrary edge weights in $[0,1]$ the worst-case running-times are easily derived from the proofs of Theorem 4 and Lemma 19. □

[5] Actually, it is suffi cient if these bounds hold with high probability.

3.9.2 Graphs with Constant Maximum Node-Degree

By Theorem 7, both ABI-Dijkstra and the sequential Δ-Stepping run in linear average-case time on graphs with independent random edge weights uniformly distributed in $[0,1]$ where the term $\mathbf{E}[d \cdot \mathcal{L}]$ concerning the chosen source s is bounded by $\mathcal{O}(n+m)$. For the edge weights under consideration, we have the trivial bound $\mathcal{L} \leq n$ that is valid for arbitrary directed graphs and any choice of the source node. Therefore:

Corollary 5 *ABI-Dijkstra has linear average-case complexity on arbitrary directed graphs with random edge weights and constant maximum node degree. For the same graph class, the sequential Δ-Stepping algorithm runs in linear time with high probability.*

Important graph families with small constant degree – thus implying linear average-case time – frequently arise for transportation and communication problems. Furthermore, grid-graphs of small dimensions where a certain fraction of edges is missing frequently occur in applications of *percolation research* [73].

Another important input family that can be efficiently tackled by ABI-Dijkstra and the Δ-Stepping is the class of *random graphs* with random edge weights, which we will consider in the next subsection.

3.9.3 Random Graphs

Over the last forty years, the theory of random graphs [17, 46] has developed into an independent and fast-growing branch of mathematics with applications to reliability of transportation and communication networks or natural and social sciences. Frequently, random graphs are chosen to average the performance of an algorithm over all possible structural inputs.

We use the random digraph model $D(n, \bar{d}/n)$, which was introduced by Angluin and Valiant [9]. An instance of $D(n, \bar{d}/n)$ is a directed graph with n nodes where each edge is present with probability $\bar{d}/n$, independently of the presence or absence of other edges. Hence, each node has expected out-degree (and in-degree) $\bar{d}$.

Let us turn to the maximum node degree d of a random graph. There are deep results on this topic partially taking into account the whole degree sequence of random graphs, e.g., see [16, 101, 126]. However, for our purposes, a very simple proof yields a sufficient bound:

Lemma 26 *Let $G = (V,E)$ be a random graph from $D(n, \bar{d}/n)$. The maximum node degree d satisfies $d = \mathcal{O}(\bar{d} + \log n)$ whp.*

Proof: Define $X_{v,w}$ to be the binary random variable that is one if $(v,w) \in E$, and zero otherwise. Let O_v (I_v) denote the number of outgoing (incoming) edges of node v. Hence, $\mathbf{E}[O_v] = \sum_{w \in V} \mathbf{E}[X_{v,w}] = \bar{d}$ and $\mathbf{E}[I_v] = \sum_{w \in V} \mathbf{E}[X_{w,v}] = \bar{d}$. Using the formula of the Chernoff bound from Lemma 7 we find

$$\mathbf{P}[O_v > (1+\delta) \cdot \bar{d}] = \mathbf{P}[I_v > (1+\delta) \cdot \bar{d}] \leq e^{-\min\{\delta^2, \delta\} \cdot \bar{d}/3}.$$

Note the dependence between O_v and I_w caused by the edge (v,w). However, using Boole's inequality, each in-degree and each out-degree of G is at most $(1+\delta) \cdot \bar{d}$ with probability at least $1 - 2 \cdot n \cdot e^{-\min\{\delta^2, \delta\} \cdot \bar{d}/3}$. Choosing $\delta = \max\{(3/\bar{d}) \cdot (c+2) \cdot \ln n,\ 1\}$ for

some arbitrary constant $c \geq 1$, we find that the sum of maximum in-degree and maximum out-degree exceeds $2 \cdot \max\{2 \cdot \bar{d},\ \bar{d} + 3 \cdot (c+2) \cdot \ln n\}$ with probability at most n^{-c}. □

Next we examine the maximum shortest-path weight $\mathcal{L}$ of random graphs with independent random edge weights uniformly distributed in $[0,1]$. Again, we can rely on intensive previous work: in order to bound $\mathcal{L}$ for sparse random graphs with random edge weights we can use a known result on the their *diameter*. Let $\text{minsize}(u,v)$ denote the minimum number of edges needed among all paths from u to v in a graph $G = (V,E)$ if any, $-\infty$ otherwise. Then the diameter of G is defined to be $\mathcal{D} := \max_{u,v \in V}\{\text{minsize}(u,v), 1\}$. Reif and Spirakis [125] gave the following bound on the diameter of random graphs:

Lemma 27 ([125]) *The diameter of a random directed graph from $D(n, \bar{d}/n)$ is bounded by $\mathcal{O}(\log n)$ with probability at least $1 - n^{-c}$ for some arbitrary constant $c \geq 1$ and for all $\bar{d}/n \in [0,1] - [c^*/n, 2 \cdot c^*/n]$ where $c^* > 1$ is some constant that depends on c.*

Since each edge has weight at most one, $\mathcal{L} = \mathcal{O}(\log n)$ whp for nearly all choices of $\bar{d}$. However, the more random edges are added to the graph, the smaller the expected maximum shortest-path weight: based on a careful review of [57, 80], Priebe [120] shows $\mathcal{L} = \mathcal{O}(\frac{\log n}{\bar{d}})$ whp for $\bar{d} \geq C \cdot \log n$ where C is some sufficiently large constant. Combining these observations we can immediately deduce the following simple bound (we will show a stronger result in Chapter 4):

Corollary 6 *For a directed random graph from $D(n, \bar{d}/n)$ with independent random edge weights uniformly distributed in $[0,1]$, the maximum shortest-path weight $\mathcal{L}$ is bounded by $\mathcal{O}(\frac{\log^2 n}{\bar{d}})$ whp for all $\bar{d} \geq 2 \cdot c^*$ where $c^* > 1$ is the constant from Lemma 27.*

Consequently, by Lemma 26 and Corollary 6, $d \cdot \mathcal{L} = \mathcal{O}((\bar{d} + \log n) \cdot \frac{\log^2 n}{\bar{d}}) = \mathcal{O}(\log^3 n)$ with probability at least $1 - 2 \cdot n^{-c}$ for some arbitrary constant $c > 1$. Hence, taking $c \geq 2$ and recalling the worst-case bound $d \cdot \mathcal{L} = \mathcal{O}(n^2)$, this implies $\mathbf{E}[d \cdot \mathcal{L}] \leq \mathcal{O}(\log^3 n) + 2 \cdot n^{-c} \cdot \mathcal{O}(n^2) = \mathcal{O}(\log^3 n)$. Therefore, we conclude the following from Theorem 7:

Corollary 7 *Assuming independent random edge weights uniformly distributed in $[0,1]$, ABI-Dijkstra and Δ-Stepping run in linear average-case time on directed random graphs from $D(n, \bar{d}/n)$ where $\bar{d} \geq 2 \cdot c^*$ for the constant c^* of Lemma 27. On this graph class, the Δ-Stepping even runs in linear time with high probability.*

So far we have given upper bounds on the average-case complexity of some sequential SSSP algorithms. In the following section we will provide superlinear lower bounds on the average-case running time of certain SSSP algorithms.

3.10 Lower Bounds

Looking at the average-case performance of the sequential SSSP approaches considered so far, one might conclude that random edge weights automatically result in good algorithmic performance. Limitations for simple algorithms like ABI-Dijkstra as seen for graph classes with high maximum degree d might be artifacts of a poor analysis. For example, up to

now we have not excluded that taking a bucket width $\Omega(1/d)$ for ABI-Dijkstra would still reasonably bound the overhead for node rescans while the resulting reduction of the number of buckets to be traversed might facilitate linear average-case time. In this section we tackle questions of that kind. We provide graph classes with random edge weights that force a number of well-known label-correcting SSSP algorithms into superlinear average-case time.

Worst-case inputs for label-correcting algorithms are usually based on the following principle: Paths with a few edges are found earlier but longer paths have smaller total weights and hence lead to improvements on the tentative distances. Each such improvement triggers a node rescan (and potentially many edge re-relaxations), which eventually make the computation expensive. We shall elucidate this strategy for the Bellman–Ford algorithm.

The shortest-paths algorithm of Bellman–Ford [15, 50], **BF** for short, is the classical label-correcting approach. It maintains the set of labeled nodes in a FIFO queue Q. The next node v to be scanned is removed from the head of the queue; a node $w \notin Q$ whose tentative distance is reduced after the relaxation of the edge (v, w) is appended to the tail of the queue; if w already belongs to Q then it will not be appended. We define a *round* of **BF** by induction: the initialization, during which the source node is added to Q, is round zero. For $i > 0$, round i scans the nodes that were added to the queue during round $i - 1$.

Fixed Edge Weights

Now we explicitly construct a difficult input graph class with fixed edge weights for **BF**; see Figure 3.7 for a concrete instance. Let us call the class $G_{BF}(n, r)$. The shortest path from the single source $s = v_0$ to the node $q = v_r$ is given by $\mathcal{P} = \langle v_0, v_1, \ldots, v_{r-1}, v_r \rangle$. Each edge on $\mathcal{P}$ has weight one. Furthermore, there are edges (v_i, q), $0 \leq i < r$, having weight $c(v_i, q) = 2 \cdot (r - i) - 1$. Let us assume that these edges appear first in the adjacency lists of their respective source nodes. Finally, q has $n - r - 1$ outgoing edges to nodes that are not part of $\mathcal{P}$. Hence, for any choice of r, $G_{BF}(n, r)$ consists of at most $2 \cdot n$ edges.

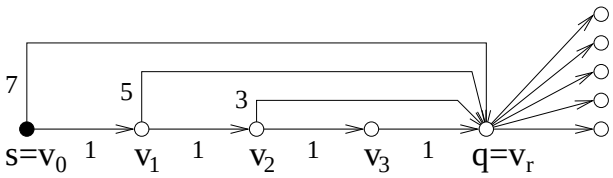


Figure 3.7: Difficult input graph $G_{BF}(10, 4)$ with fixed edge weights. It causes the Bellman–Ford algorithm to scan node q four times.

For this graph class, **BF** shows the following behavior: at the beginning of round i, $1 \leq i \leq r - 1$, the nodes v_i and q are in Q with $\mathrm{tent}(q) = 2 \cdot r - i$ and $\mathrm{tent}(v_i) = i$, respectively. Due to the order of the adjacency lists and the FIFO processing, q will be scanned before v_i in round i. Scanning v_i first reduces $\mathrm{tent}(q)$ to $i + 2 \cdot (r - i) - 1 = 2 \cdot r - i - 1$, thus adding q to Q again; then the node scan of v_i also relaxes the edge (v_i, v_{i+1}), hence reducing $\mathrm{tent}(v_{i+1})$ to $i + 1$. However, since the edge (v_i, v_{i+1}) is relaxed after the edge (v_i, q), the node v_{i+1} is appended to Q after q is already in Q again. This maintains the malicious setting for the next round. Altogether, q is scanned r times. Each scan of q

relaxes all its $n - r - 1$ outgoing edges. If we choose $r = n/2$, then **BF** performs $\Omega(n^2)$ operations for this graph with $\mathcal{O}(n)$ edges with fixed weights.

For random edge weights it is unlikely that a given long path has a small total path weight (e.g., compare Lemma 17 for paths of total weight at most 1). Moreover, the expected path weight is linear in the number of edges. In fact, if we replace the fixed edge weights in the graph class above by random edge weights then the expected number of rescans of q is constant and therefore the expected time for **BF** is linear in that case.

3.10.1 Emulating Fixed Edge Weights

Our main idea for the construction of difficult graphs with random edge weights is to emulate *single edges* e_i having 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 subgraph S_i is built by a chain of so-called (u, v, k)*-gadgets*:

Definition 4 *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 28 *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 Y_1 and Y_2 be the random variables that denote the weight of the paths $\langle u, w_1, v\rangle$ and $\langle u, w_2, v\rangle$, respectively. Let $f_e(x)$ denote the density function for the weight of the single edge e:

$$f_e(x) = \begin{cases} 1 & \text{for } 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

Using the definition above, the density function for Y_i is given by

$$\begin{aligned} f_{Y_i}(x) &= \int_{-\infty}^{+\infty} f_{(u,w_i)}(y) \cdot f_{(w_i,v)}(x-y)\ dy \\ &= \int_{\max\{0,x-1\}}^{\min\{1,x\}} 1 \cdot 1\ dy = \begin{cases} x & \text{for } 0 \le x \le 1 \\ 2 - x & \text{for } 1 < x \le 2 \end{cases} \end{aligned}$$

The distribution function for the shortest path weight in a $(u, v, 2)$-gadget is given by

$$\begin{aligned} W(z) &= 1 - \left(1 - \int_0^z f_{Y_1}(x)\ dx\right) \cdot \left(1 - \int_0^z f_{Y_2}(x)\ dx\right) \\ &= \begin{cases} \frac{-z^4}{4} + z^2 & \text{for } 0 \le z \le 1 \\ \frac{-z^4}{4} + 2 \cdot z^3 - 6 \cdot z^2 + 8 \cdot z - 3 & \text{for } 1 < z \le 2. \end{cases} \end{aligned}$$

Finally,

$$\mathbf{E}[\ \min\{Y_1, Y_2\}\ ] = \int_0^2 x \cdot w(x)\ dx$$

where

$$w(z) = \begin{cases} -z^3 + 2 \cdot z & \text{for } 0 \le z \le 1 \\ -z^3 + 6 \cdot z^2 - 12 \cdot z + 8 & \text{for } 1 < z \le 2 \end{cases}$$

denotes the derivative of $W(z)$. A simple calculation yields

$$\begin{aligned} \mathbf{E}[\,\min\{Y_1, Y_2\}\,] &= \int_0^1 -x^4 + 2 \cdot x^2 \;\; dx + \int_1^2 -x^4 + 6 \cdot x^3 - 12 \cdot x^2 + 8 \cdot x \;\; dx \\ &= \frac{7}{15} + \frac{3}{10} = \frac{23}{30}\,. \end{aligned}$$

□

Large fixed edge weights are emulated by chains of $l \ge 1$ gadgets, each of which has blow-up factor one. Smaller fixed weights are emulated using either fewer gadgets or higher blow-up factors for a fraction of the gadgets in their respective subgraphs. If we take the parameter l large enough then the actual shortest path weights in the chain subgraphs of gadgets 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.

3.10.2 Inputs for Algorithms of the List Class

Now we provide a concrete conversion example, which works for several SSSP label-correcting algorithms that apply simple list data-structures (like FIFO queues), among them the Bellman–Ford algorithm. As a basis we use the input class $G_{BF}(n, r)$ with fixed edge weights of Figure 3.7. Let $0 < \epsilon < 1/3$ be some arbitrary constant. The new input class with random edge weights – called $G_L(n, \epsilon)$ – is derived as follows: the fixed-weight edges entering the node q in $G_{BF}(n, r)$ are replaced by $r = n^{1/3-\epsilon}$ subgraphs S_i, $0 \le i < r$, each of which consists of a chain with $l = n^{2/3}$ gadgets. More specifically, S_i contains $(i \cdot n^{1/3+\epsilon})$ $(\cdot, \cdot, 2)$-gadgets whereas the remaining $l - i \cdot n^{1/3+\epsilon}$ gadgets in S_i have blow-up factor one. Altogether this accounts for $n^{1-\epsilon}$ gadgets, that is at most $4 \cdot n^{1-\epsilon}$ nodes and edges. Each S_i is reachable from s along a chain of $i + 1$ auxiliary edges; one further edge connects each S_i to q. Thus, the shortest path $\mathcal{P}_i$ from s to q through S_i comprises $2 \cdot n^{2/3} + i + 2$ edges. Figure 3.8 shows the principle. All these auxiliary edges account for at most another $3 \cdot n^{1/3-\epsilon}$ nodes and edges in the graph. Finally, q has outgoing edges to the remaining $\Theta(n)$ nodes of the graph. Hence, $G_L(n, \epsilon)$ consists of $\Theta(n)$ edges. Similarly to $G_{BF}(n, r)$, the adjacency lists of the nodes $s = v_0, v_1, \ldots, v_{r-1}$ are ordered in a way that the edge (v_i, S_i) is relaxed first when v_i is scanned.

Let W_i be the random variable denoting the weight of the shortest path $\mathcal{P}_i$ from s to q through S_i in $G_L(n, \epsilon)$. Due to the auxiliary edges that connect S_i with s, we find

$$E[W_i] = l - i \cdot 7/30 \cdot n^{1/3+\epsilon} + (i + 2)/2.$$

Hence, $E[W_i] - E[W_{i+1}] = 7/30 \cdot n^{1/3+\epsilon} - 1/2$. We will make use of yet another variant of the Chernoff bounds in order to show that $W_i > W_{i+1}$ with high probability:

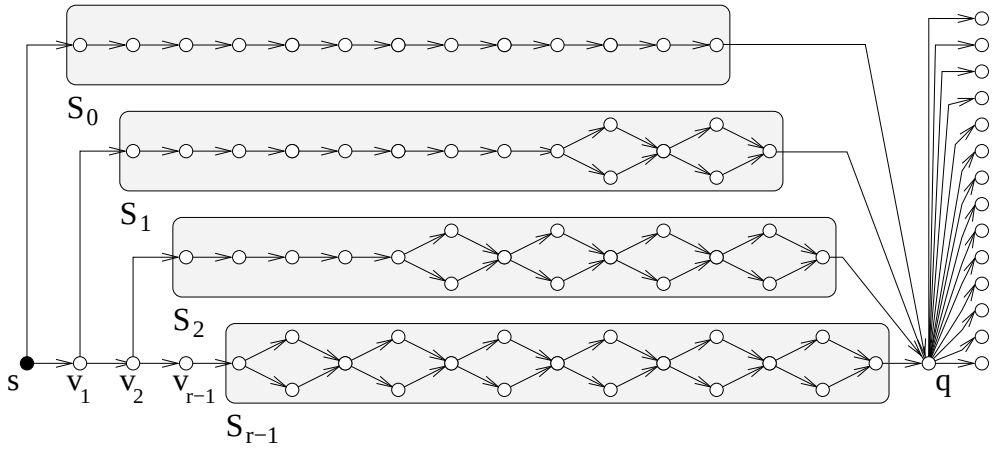


Figure 3.8: An instance of the graph class $G_L(n, \epsilon)$ with random edge weights; it is designed to cause large average-case running-times for label-correcting SSSP algorithms of the list class.

Lemma 29 ([45, 82]) *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 $t \geq 0$,*

$$\mathbf{P}\left[\mid X - \mathbf{E}[X] \mid \geq t\right] \quad \leq \quad 2 \cdot \exp\left(\frac{-2 \cdot t^2}{\sum_k (b_k - a_k)^2}\right) . \tag{3.21}$$

Lemma 30 *Let $\mathcal{W}$ be the event that the random variables W_i for the weight of the shortest path $\mathcal{P}_i$ from s to q through S_i in $G_L(n, \epsilon)$ satisfy*

$$W_0 > W_1 > \ldots > W_{n^{1/3-\epsilon}-1}.$$

For every constant $0 < \epsilon < 1/3$ and sufficiently large n, $\mathcal{W}$ holds with probability at least $1 - e^{-n^\epsilon}$.

Proof: Lemma 29 is applied as follows: for $n \geq 8$, W_i is the sum of $n^{2/3} + i + 2 \leq 2 \cdot n^{2/3}$ independent random variables: $n^{2/3}$ random variables for the shortest-path distances of the respective $n^{2/3}$ gadgets in S_i, and $i + 2$ random variables for the weights of the external edges that connect S_i with s and q. Each of these random variables takes values in the interval $[0, 2]$. Thus,

$$\mathbf{P}\left[\mid W_i - \mathbf{E}[W_i] \mid \geq 4 \cdot n^{1/3+\epsilon/2}\right] \leq 2 \cdot \exp\left(\frac{-2 \cdot 16 \cdot n^{2/3+\epsilon}}{2 \cdot n^{2/3} \cdot 4}\right) = 2 \cdot e^{-4 \cdot n^\epsilon}.$$

By Boole's inequality, $|W_i - \mathbf{E}[W_i]| < 4 \cdot n^{1/3+\epsilon/2}$ for all i, $0 \leq i < n^{1/3-\epsilon}$, with probability at least $1 - 2 \cdot n^{1/3-\epsilon} \cdot e^{-4 \cdot n^\epsilon} \geq 1 - e^{-n^\epsilon}$ for $n \to \infty$. We still have to check that $|W_i - \mathbf{E}[W_i]| \leq 4 \cdot n^{1/3+\epsilon/2}$ and $|W_{i+1} - \mathbf{E}[W_{i+1}]| \leq 4 \cdot n^{1/3+\epsilon/2}$ together imply $W_i > W_{i+1}$:

$$\begin{aligned} W_i - W_{i+1} \quad &\geq \quad \mathbf{E}[W_i] - 4 \cdot n^{1/3+\epsilon/2} - \left(\mathbf{E}[W_{i+1}] + 4 \cdot n^{1/3+\epsilon/2}\right) \\ &= \quad 7/30 \cdot n^{1/3+\epsilon} - 1/2 - 8 \cdot n^{1/3+\epsilon/2} > 0 \ \text{ for } n \to \infty . \end{aligned}$$

□

If the event $\mathcal{W}$ holds then we have achieved our main emulation goal: paths from s to q of larger size (i.e., more edges) have smaller total weight. In the following we study how some label-correcting algorithms perform on graphs from $G_L(n, \epsilon)$ if $\mathcal{W}$ holds.

The Bellman–Ford Algorithm

The graph class $G_L(n, \epsilon)$ was derived from the input class $G_{BF}(n, r)$, which had been tuned to the Bellman–Ford algorithm. Therefore, most of the subsequent observations are simple consequences of the discussion in Section 3.10. If the event $\mathcal{W}$ of Lemma 30 holds then the following actions take place: The node q is first reached via the shortest path $\mathcal{P}_0$ through the subgraph S_0; q will be queued. However, q will have been removed from the queue before the outgoing edge of the last node in S_1 will be relaxed. The relaxation via $\mathcal{P}_1$ improves $\mathrm{tent}(q)$; therefore, q is queued again. After the removal of q, $\mathrm{tent}(q)$ is improved via $\mathcal{P}_2$, etc. Consequently, the node q is scanned r times. Each time q is scanned, its $\Theta(n)$ outgoing edges are relaxed. Thus, if $\mathcal{W}$ holds, then **BF** requires non-linear $\Theta(n^{4/3-\epsilon})$ operations on graphs from $G_L(n, \epsilon)$.

Lemma 31 *There are input graphs with $\mathcal{O}(n)$ nodes and edges and random edge weights such that the Bellman–Ford algorithm with a FIFO queue requires $\Theta(n^{4/3-\epsilon})$ operations whp for any constant $0 < \epsilon < 1/3$.*

Proof: Follows immediately from the discussion above. □

Implementations of the **BF** algorithm often apply the so-called *parent-checking heuristic*: the outgoing edges of a node v in the queue are only relaxed if v's parent, u, concerning the current shortest path tree is not in the queue as well; otherwise $\mathrm{tent}(u)$ was recently improved and $\mathrm{tent}(v)$ is surely not final. Consequently, v is discarded from the queue.

The heuristic does not improve the performance of the Bellman-Ford algorithm on graphs from $G_L(n, \epsilon)$: the only nodes affected by this heuristic are the successor nodes of q when $\mathrm{tent}(q)$ is improved. However, none of these nodes has outgoing edges; removing them from the queue is asymptotically as costly as scanning them.

The input class $G_L(n, \epsilon)$ also yields poor performance on other SSSP label-correcting approaches belonging to the category of list algorithms. In the following we briefly consider two examples: the incremental graph algorithm of Pallottino [117] and the topological ordering algorithm of Goldberg and Radzik [69].

The Algorithms of Pallottino and Goldberg–Radzik

Pallottino's algorithm [117] – **PAL** for short – maintains two FIFO queues Q_1 and Q_2. Labeled nodes that have been scanned at least once are stored in Q_1 whereas labeled nodes that have never been scanned are maintained in Q_2. At any time, each node is in at most one queue. The next node to be scanned is removed from the head of Q_1 if this queue is not empty and from the head of Q_2 otherwise. Initially, Q_1 is empty and Q_2 holds the source node s. If $\mathrm{tent}(v)$ is reduced due a relaxation of the edge (u, v) while scanning u then v is only added to either Q_1 or Q_2 if it is currently not stored in any of them: if v was scanned before then it will be added to the tail of Q_1, or to the tail of Q_2 otherwise. This approach

has worst-case execution time $\mathcal{O}(n^2 \cdot m)$ but performs very well on many practical inputs [25, 145].

Due to the structure of our graph class $G_L(n, \epsilon)$ the only nodes that can ever appear in queue Q_1 are the node q and its immediate successors. Similarly to the **BF** algorithm the FIFO queues in **PAL** 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, **PAL** frequently puts q into Q_1 and thus relaxes the $\Theta(n)$ outgoing edges of q before the computation carries on. Altogether **PAL** needs $\Theta(n^{4/3-\epsilon})$ operations whp.

The algorithm of Goldberg and Radzik [69], abbreviated **GOR**, maintains two sets of nodes, Q_1 and Q_2, as well. Initially, Q_1 is empty and Q_2 contains the starting node. At the beginning of each round, the basic version of **GOR** builds Q_1 based on Q_2 and makes Q_2 the empty set. Q_1 is linearly ordered. During a round, the nodes are scanned from Q_1 according to this order; target nodes whose tentative distances are reduced by the scanning will be put into Q_2. After Q_1 is empty it is refilled as follows: let Q' be the set of nodes reachable from Q_2 via edges (u, v) in the subgraph G' having reduced costs $\mathrm{tent}(u)+c(u, v)-\mathrm{tent}(v) \leq 0$. Using depth first search, Q_1 is assigned the topologically sorted[6] set Q' such that for every pair of nodes u and v in Q' with $(u, v) \in G'$, u precedes v in Q_1. Thus, (u, v) will be relaxed before v is scanned from Q_1.

An edge $(u, v) \in E$ with $\mathrm{tent}(u) = \infty$ and $\mathrm{tent}(v) = \infty$ only belongs to G' if $c(u, v) = 0$. Let $\mathcal{W}'$ be the event that the input graph $G_L(n, \epsilon)$ contains no edges with weight zero. For independent random edge weights uniformly drawn from $[0, 1]$, $\mathcal{W}'$ holds with probability 1. Given $\mathcal{W}'$, it is easily shown by induction that (1) in the k-th round, **GOR** scans nodes that are reachable from s via paths of at most $2 \cdot k$ edges; (2) the DFS search in G' at the end of round k only finds nodes that can be reached from s in $G_L(n, \epsilon)$ via paths of at most $2 \cdot k + 2$ edges.

If the event $\mathcal{W}$ of Lemma 30 and $\mathcal{W}'$ hold then the node q of $G_L(n, \epsilon)$ is scanned for the first time in round $n^{2/3}+1$, based on a path from s to q via S_0 having $2 \cdot n^{2/3}+2$ edges. The next better paths via S_1 and S_2 comprise $2 \cdot n^{2/3}+3$ edges and $2 \cdot n^{2/3}+4$ edges, respectively; thus, they are not discovered by the DFS search at the end of round $n^{2/3}$. Therefore, the respective last nodes of S_1 and S_2 are not scanned before q in round $n^{2/3} + 1$. Similarly, a rescan of q during round $n^{2/3} + 1 + i$ is due to a path from s to q though $S_{2 \cdot i}$; none of the better paths via $S_{2 \cdot i+1}, \ldots, S_{r-1}$ has been found by the previous DFS searches. Thus, once q has been reached for the first time in round $n^{2/3} + 1$, its $\Theta(n)$ outgoing edges will be re-relaxed during each of the next $\lfloor (r - 1)/2 \rfloor$ phases again. We conclude that **GOR** requires $\Omega(n^{4/3-\epsilon})$ operations if $\mathcal{W}$ and $\mathcal{W}'$ hold, which happens with high probability.

3.10.3 Examples for Algorithms with Approximate Priority Queues

After having discussed difficult input graphs with random edge weights for some algorithms with simple list data structures in Section 3.10.2, we now turn to SSSP algorithms which apply some simple "approximate" priority queues.

[6] Topological sorting of G' is possible for nonnegative edge weights, since there are no cycles with negative total weight in G'; cycles of weight zero are either contracted or the respective back edges discovered during the DFS computation are deleted from G'.

Simple Bucket Approaches

Let us reconsider the sequential Δ-Stepping approach of Section 3.9.1 that applies an array $B[\cdot]$ with buckets of width Δ. For maximum node degree d and maximum shortest-path weight $\mathcal{L}$, our analysis showed that it is sufficient to take buckets of width $\Delta = \Theta(1/d)$ in order to limit the average-case overhead for node rescans and edge re-relaxations by $\mathcal{O}(n+m)$. However, $\lceil \mathcal{L}/\Delta \rceil = \Theta(\mathcal{L} \cdot d)$ buckets have to be traversed for that choice of Δ. The resulting total average-case time $\mathcal{O}(\mathbf{E}[\mathcal{L} \cdot d] + n + m)$ may be largely dominated by the term $\mathbf{E}[\mathcal{L} \cdot d]$. To which extent can Δ be increased in order to find an optimal compromise between the number of traversed buckets and the overhead due to node rescans?

In the following we will show that there are graphs with $\mathcal{O}(n)$ edges where any fixed bucket width Δ results in $\Omega(n \cdot \sqrt{\log n / \log\log n})$ operations on the average. This provides another motivation for the need of improved algorithms like the adaptive bucket splitting approaches **SP-C** and **SP-S**.

In principle, we would like to re-use the graph class $G_L(n, \epsilon)$ of Section 3.10.2 in order to construct difficult inputs for the Δ-Stepping. But there, in order to deal with random deviations, single fixed edge weights had been emulated by long chains of $n^{2/3}$ gadgets with random edge weights; each chain accounted for a total path-weight of $\Theta(n^{2/3})$ on the average. Our analysis for **SP-C** (and the Δ-Stepping), however, pointed out that node rescans can be attributed to paths of small total weight (at most $\Delta \leq 1$). Therefore, we will design another input class for the Δ-Stepping, featuring short chains and gadgets with rapidly increasing blow-up factors. The following lemma bounds the probability that the shortest path weight of a single (u, v, k)-gadget deviates too much from its expected value:

Lemma 32 *The shortest path weight between the nodes u and v in a (u, v, k)-gadget, $k \geq \ln^2 n$, lies in the interval $\left[\frac{1}{\sqrt{k} \cdot \ln n}, \frac{\ln n}{\sqrt{k}}\right]$ with probability at least $1 - \frac{2}{\ln^2 n}$ for $n \geq 20$.*

Proof: As in the proof of Lemma 28, let Y_i be the random variable denoting the weight of the path $\langle u, w_i, v \rangle$ via the intermediate node w_i within the gadget. Furthermore, let $Z = \min_i Y_i$ be the weight of the shortest path between u and v. The density function for Y_i is given by $f_{Y_i}(x) = x$ if $0 \leq x \leq 1$ (compare the proof of Lemma 28), hence $\mathbf{P}[Y_i \leq x] = 1/2 \cdot x^2$ for $0 \leq x \leq 1$. Therefore,

$$\mathbf{P}\left[Z \geq \frac{1}{\sqrt{k} \cdot \ln n}\right] = \left(1 - \frac{1}{2} \cdot \frac{1}{k \cdot \ln^2 n}\right)^k \geq 1 - \frac{1}{2} \cdot \frac{1}{k \cdot \ln^2 n} \cdot k > 1 - \frac{1}{\ln^2 n}.$$

In order to show $\mathbf{P}\left[Z \leq \frac{\ln n}{\sqrt{k}}\right] = 1 - \mathcal{O}\left(\frac{1}{\ln^2 n}\right)$, let X_i be a binary random variable such that $X_i = 1$ if both $c(u, w_i) \leq \frac{\ln n}{2 \cdot \sqrt{k}}$ and $c(w_i, v) \leq \frac{\ln n}{2 \cdot \sqrt{k}}$, and $X_i = 0$ otherwise. Due to independent random edge weights, $\mathbf{P}[X_i = 0] = 1 - \frac{\ln^2 n}{4 \cdot k}$. Clearly, $Z \leq \frac{\ln n}{\sqrt{k}}$ if $\sum_i X_i \geq 1$. Hence,

$$\mathbf{P}\left[Z \leq \frac{\ln n}{\sqrt{k}}\right] \geq \mathbf{P}\left[\sum_i X_i \geq 1\right] = 1 - \left(1 - \frac{\ln^2 n}{4 \cdot k}\right)^k \geq 1 - e^{-\frac{\ln^2 n}{4}} \geq 1 - \frac{1}{\ln^2 n}$$

for $n \geq 20$. Therefore, by Boole's inequality, $\mathbf{P}\left[\frac{1}{\sqrt{k} \cdot \ln n} \leq Z \leq \frac{\ln n}{\sqrt{k}}\right] \geq 1 - \frac{2}{\ln^2 n}$. □

Lemma 32 is used in the construction of a difficult graph class for the Δ-Stepping algorithm:

Lemma 33 *There are input graphs with $\mathcal{O}(n)$ nodes and edges and random edge weights such that the Δ-Stepping algorithm requires $\Omega(n \cdot \sqrt{\log n / \log\log n})$ operations on the average, no matter how Δ is chosen.*

Proof: Consider the following graph class: from the starting node s to some node q with $n/3$ outgoing edges there are r chains $C_1, \ldots, C_r$ of gadgets. Node s is the entering node of the first gadget of each chain. Similarly, q is the leaving node of the last gadget of each chain. Chain C_i, $1 \leq i \leq r$, consists of i gadgets G_i with blow-up factor $k_i = \ln^{6 \cdot i} n$ each. Finally, a separate chain $\tilde{C}$ of $n/3$ edges branches from node s as well; compare Figure 3.9.

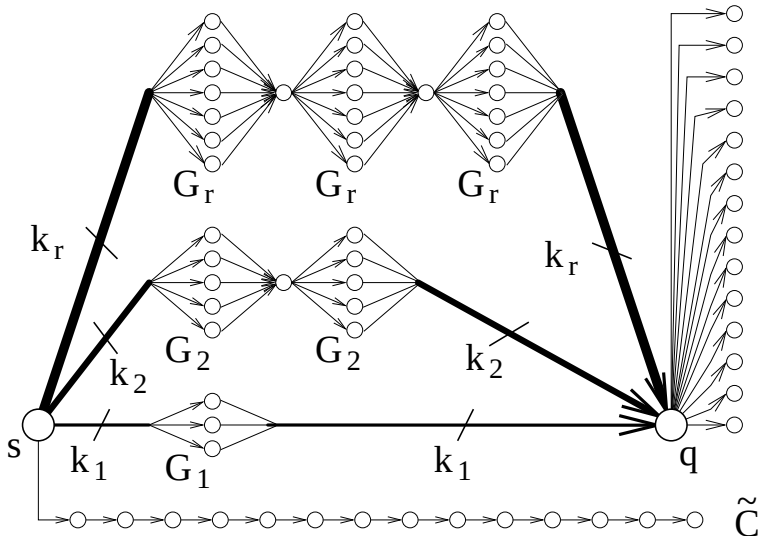


Figure 3.9: Difficult input class for the Δ-Stepping algorithm. Chain C_i consists of i gadgets G_i with blow-up factor $k_i = \ln^{6 \cdot i} n$ each. An extra chain $\tilde{C}$ causes a high maximum shortest-path weight on the average.

Due to $\tilde{C}$, the average-case maximum shortest-path weight in the graph satisfies $\mathbf{E}[\mathcal{L}] \geq n/6$. Thus, if the bucket width is chosen as $\Delta \leq 6 \cdot \sqrt{\frac{\ln\ln n}{\ln n}}$ (≤ 1 for $n \to \infty$) then the Δ-Stepping algorithm has to traverse at least $\mathbf{E}[\mathcal{L}]/\Delta = \Omega\left(n \cdot \sqrt{\frac{\ln n}{\ln\ln n}}\right)$ buckets on the average. Henceforth we will assume $1 \geq \Delta > 6 \cdot \sqrt{\frac{\ln\ln n}{\ln n}}$.

Let X_q denote the number of light edges ($c(e) \leq \Delta$) emanating from node q. Unless $\mathrm{tent}(q) = \mathrm{dist}(q)$, the Δ-Stepping algorithm only relaxes light edges out of q. We find $\mathbf{E}[X_q] = n/3 \cdot \Delta \geq 2 \cdot n \cdot \sqrt{\frac{\ln\ln n}{\ln n}}$. By Chernoff bounds (Lemma 7),

$$\mathbf{P}[X_q \leq \frac{\mathbf{E}[X_q]}{2}] \leq e^{-\mathbf{E}[X_q]/8} \leq e^{-\frac{n}{4} \cdot \sqrt{\frac{\ln\ln n}{\ln n}}} \leq \frac{1}{20} \text{ for } n \geq 20 \,.$$

Having r chains $C_1, \ldots, C_r$ accounts for a total of $\sum_{1 \leq i \leq r} i < r^2$ gadgets. We aim to choose r large under the condition that all these gadgets contribute at most $n/3$ nodes and $2 \cdot n/3$ edges in total. It is sufficient to consider the inequality $r^2 \cdot (1 + \ln^{6 \cdot r} n) \leq n/3$; after some simple transformations it turns out that $r = \frac{\ln n}{20 \cdot \ln\ln n}$ is a feasible choice for $n \geq 20$.

By Lemma 32, the weight of the shortest path within each single gadget of C_i exceeds the interval $\left[\frac{1}{\sqrt{k_i}\cdot\ln n}, \frac{\ln n}{\sqrt{k_i}}\right]$ with probability at most $\frac{2}{\ln^2 n}$ if $n \geq 20$.

Let $\mathcal{E}$ be the event that the shortest-path weights of all gadgets are bounded as stated above, and that node q has at least $\mathbf{E}[X_q]/2$ outgoing light edges. Hence,

$$\mathbf{P}[\mathcal{E}] \geq 1 - \frac{1}{20} - r^2 \cdot \frac{2}{\ln^2 n} = 1 - \frac{1}{20} - \frac{\ln^2 n}{400 \cdot (\ln\ln n)^2} \cdot \frac{2}{\ln^2 n} \geq 1 - \frac{1}{20} - \frac{1}{200} \geq \frac{9}{10} \,.$$

Let W_i be the shortest-path weight of chain C_i. If $\mathcal{E}$ holds, then

$$W_i \geq \frac{i}{\sqrt{k_i}\cdot\ln n} > \frac{(i+1)\cdot\ln n}{\sqrt{k_i}\cdot\ln^3 n} = \frac{(i+1)\cdot\ln n}{\sqrt{k_{i+1}}} \geq W_{i+1} \,, \tag{3.22}$$

and therefore $W_1 > W_2 > \ldots > W_r$ with probability at least $\frac{9}{10}$. In other words, given $\mathcal{E}$, we managed to produce an input graph where paths from s to q of larger size (i.e., more edges) have smaller total weight. How often q is scanned and how many outgoing (light) edges of q are relaxed during these scans depends on the chosen bucket width Δ: however, if $\mathcal{E}$ holds and $\Delta > 6\cdot\sqrt{\frac{\ln\ln n}{\ln n}}$, already the shortest-path from s to q through C_1, $\langle s, v_{1,1}, q\rangle$, has total weight at most $\frac{\ln n}{\sqrt{\ln^6 n}} = \frac{1}{\ln^2 n} < \Delta$. Therefore, s, $v_{1,1}$, and q are scanned from the current bucket $B[0]$ in phases 1, 2, and 3, respectively. In phase 5, the node q is scanned from $B[0]$ again since the Δ-Stepping finds a better path $\langle s, v_{2,1}, v_{2,2}, v_{2,3}, q\rangle$ through C_2. Altogether, if $\mathcal{E}$ holds and $\Delta > 6\cdot\sqrt{\frac{\ln\ln n}{\ln n}}$ then q is scanned $r \geq \frac{\ln n}{20\cdot\ln\ln n}$ times. Each such scan involves the relaxation of at least $\mathbf{E}[X_q]/2 \geq n\cdot\sqrt{\frac{\ln\ln n}{\ln n}}$ light edges leaving q. Hence – given $\mathcal{E}$ – the Δ-Stepping performs at least $\frac{n}{20}\cdot\sqrt{\frac{\ln n}{\ln\ln n}}$ edge relaxations. The claim follows from the fact that $\mathcal{E}$ holds with probability at least $9/10$. □

The "Approximate Bucket Implementation" [25] of Dijkstra's algorithm (ABI-Dijkstra) essentially boils down to the sequential Δ-Stepping *without* distinction between light and heavy edges, i.e., all outgoing edges of a removed node are relaxed. Hence, repeating the two parts of the proof above for either $\Delta \geq \ln\ln n/\ln n$ or $\Delta < \ln\ln n/\ln n$ when all edges out of node q are relaxed after each removal of q yields:

Corollary 8 *There are graphs with $\mathcal{O}(n)$ nodes and edges and random edge weights such that ABI-Dijkstra requires $\Omega(n\cdot\log n/\log\log n)$ operations on the average.*

A comparison between Lemma 33 and Corollary 8 seems to reveal an advantage of the Δ-Stepping over ABI-Dijkstra due to the distinction of light and heavy edges: for the underlying graph class and a proper choice of Δ, the Δ-Stepping approach computes SSSP in $\Theta(n\cdot\sqrt{\log n/\log\log n})$ time on the average, whereas ABI-Dijkstra requires $\Omega(n\cdot\log n/\log\log n)$ operations on the average. However, it might still be possible to construct graphs with $\mathcal{O}(n)$ nodes and edges and random edge weights where both approaches provably need $\Omega(n\cdot\log n/\log\log n)$ operations on the average.

Threshold Algorithm

Glover et al. suggested several variants [64, 65, 66] of a method which combines ideas lying behind the algorithms of Bellman–Ford and Dijkstra: the set of queued nodes is partitioned

into two subsets, NOW and NEXT. These sets are implemented by FIFO queues. The algorithms operate in phases; at the beginning of each phase, NOW is empty. Furthermore, the methods maintain a threshold parameter t whose value is computed as an average (weighted by constant factors) between the smallest and the average tentative distance among all nodes in NEXT. During a phase, the algorithms append nodes v from NEXT having $\mathrm{tent}(v) \leq t$ to the FIFO queue of NOW and scan nodes from NOW. The algorithmic variants differ in the concrete formulae for the computation of the threshold value t; there are other options as well, e.g., whether nodes v in NEXT having $\mathrm{tent}(v) \leq t$ are immediately appended to NOW or just after NOW remains empty.

The general strategy of the threshold method is potentially much more powerful than the Δ-Stepping with fixed bucket width: an appropriate re-selection of the threshold value t after each phase might avoid superfluous edge relaxations. Furthermore, no operations are wasted for the traversal of a large number of empty buckets. However, subsequently we will demonstrate that very similar graph classes as those for the Δ-Stepping also cause superlinear average-case time for the threshold algorithms with FIFO queues.

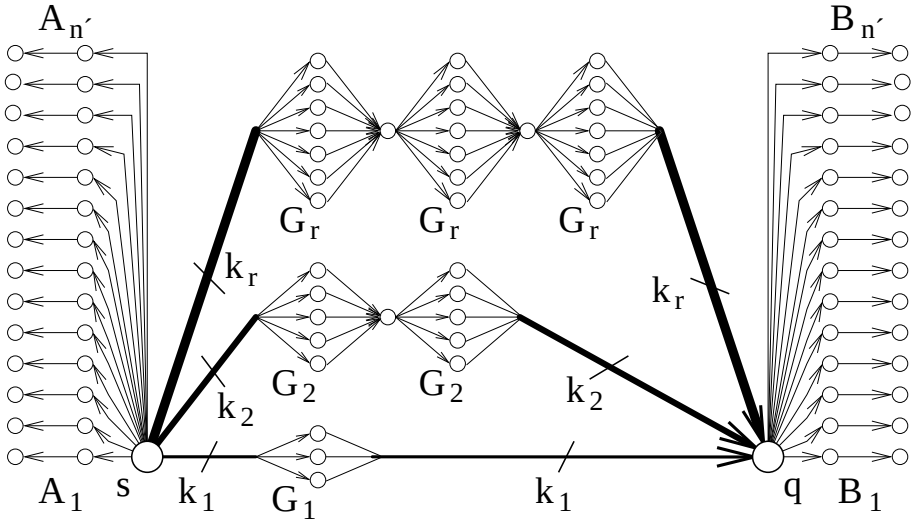


Figure 3.10: Modified input class for the threshold approach.

Lemma 34 *There are input graphs with n nodes, $m = \mathcal{O}(n)$ edges and random edge weights such that the standard threshold method with two FIFO queues requires $\Omega(n \cdot \log n / \log\log n)$ operations on the average.*

Proof: In the following we assume that nodes from NEXT are only transferred to NOW after NOW becomes empty. We re-use the notation and graph class developed in the proof of Lemma 33 with slight modifications, compare Figures 3.9 and 3.10: the separate long chain $\tilde{C}$ is omitted, and the $\Theta(n)$ edges out of the node q are extended to $n' = \Theta(n)$ independent paths $B_i = \langle q, b_i, b_i' \rangle$ of two edges each. Similarly, n' independent paths $A_i = \langle s, a_i, a_i' \rangle$ of length two each branch from the node s.

With at least constant probability it is true that (1) the event $\mathcal{E}$ from the proof of Lemma 33 holds, (2) $\Theta(n)$ edges from both A_i and B_i have weight at least $1/2$. We

will assume that these conditions hold. The basic rationale behind the chains A_i and B_i is to keep the set NEXT well filled with nodes having tentative distances at least $1/2$, thus maintaining a large threshold value.

After the edges of s are relaxed, NEXT contains the inner nodes of the first gadget from each chain C_i and also $\Theta(n)$ nodes a_i from the chains A_i. As a consequence, given the conditions on the edge weights as stated above, the threshold value will be $t = \Theta(1)$. Hence, a constant fraction of the nodes in NEXT will be moved to NOW, and – as $\mathcal{E}$ holds – the nodes on the shortest paths of the chains C_i are among them. The relaxation of all edges out of nodes in NOW will put another $\Theta(n)$ nodes a'_i from A_i into NEXT, thus keeping the threshold value large ($t = \Theta(1)$) whereas $\mathrm{tent}(q) \leq 1/\ln^2 n$ due to $\mathcal{E}$ and a path $\langle s, v', q\rangle$ where $v' \in C_1$.

Subsequently, one observes $r - 1 = \Theta(\log n/\log\log n)$ cycles of the same pattern: scanning q from NOW will cause NEXT to store $\Theta(n)$ nodes b_i or b'_i from B_i for the next two phases. Therefore, the threshold value remains sufficiently high to ensure that those nodes of C_i that are required for the next improvement of q are transferred to NOW. As a consequence, q is re-inserted into NOW with improved distance value, and the next cycle starts. Note that subsequent updates for $\mathrm{tent}(q)$ are separated by a phase where q is not part of NOW; however, as the queues operate in FIFO mode, the order of these updates will be maintained. Hence, with at least constant probability there are $\Theta(\log n/\log\log n)$ cycles, each of which requires $\Theta(n)$ operations. □

3.10.4 Summary Diffi cult Input Graphs

We have given provably difficult graph classes for a number of label-correcting algorithms, demonstrating that random edge weights do not automatically ensure good average-case performance. The proved lower bound results for graphs with $m = \mathcal{O}(n)$ edges are summarized in Table 3.2.

Algorithm	Running Time
Bellman–Ford Alg.	$\Omega(n^{4/3-\epsilon})$
Pallottino's Incremental Graph Alg.	$\Omega(n^{4/3-\epsilon})$
Basic Topological Ordering Alg.	$\Omega(n^{4/3-\epsilon})$
Threshold Alg.	$\Omega(n \cdot \log n/\log\log n)$
ABI–Dijkstra	$\Omega(n \cdot \log n/\log\log n)$
Δ-Stepping	$\Omega(n \cdot \sqrt{\log n/\log\log n})$

Table 3.2: Average-case running times for difficult input classes with $m = \mathcal{O}(n)$ edges and random edge weights.

3.11 Conclusions Sequential SSSP

We have presented the first SSSP algorithms that run in linear average-case time on arbitrary directed graphs with random edge weights. 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. However, as we shall see in Chapter 4, the label-correcting scheme has the advantage to support parallelization. Besides implications of our results for the analysis of simpler SSSP algorithms, we have shown how to construct difficult input graphs with random edge weights for many traditional label-correcting algorithms.

3.11.1 Open Problems

As for sequential SSSP, maybe the most interesting and most difficult open question is whether one can devise a worst-case linear time algorithm for arbitrary directed graphs with non-negative edge weights. If such an algorithm exists at all, then it seemingly requires completely new ideas.

In the following we will sketch some problems that are closer connected to the sequential SSSP algorithms presented in this thesis.

Dependent Random Edge Weights

Our analyses to obtain the high-probability bounds on the running time of our sequential SSSP algorithms crucially depend on the assumption of *independent* random edge weights. However, there are input classes with *dependent* random edge weights where the whp-bounds still hold: for example, consider *random geometric graphs* [40, 130] from the class $G_n^2(r)$ where n nodes are randomly placed in a unit square, and each edge weight equals the Euclidean distance between the two involved nodes. An edge (u, v) is included in the graph if the Euclidean distance between u and v does not exceed the parameter $r \in [0, 1]$. Taking $r = \Theta(\sqrt{\log(n)/n})$ results in a connected graph with maximum shortest-path weight $\mathcal{L} = \mathcal{O}(1)$ whp. Even though the nodes are placed independently and uniformly at random, the resulting random edge weights are dependent, see Figure 3.11. Still, choosing the bucket width for ABI-Dijkstra or the sequential Δ-Stepping as $\Delta := r$, both algorithms will run in linear time with high probability; each node is scanned from the current bucket at most twice, and therefore each edge is re-relaxed at most once. This is easily understood:

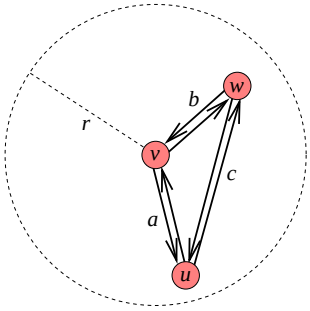


Figure 3.11: Dependent edge weights for random geometric graphs from $G_n^2(r)$: $a \leq b + c$.

Consider a node v with final distance $\mathrm{dist}(v)$ in the range $[M, M + \Delta)$ of the current bucket B_{cur}. Let v be scanned from B_{cur} for the first time in phase t. As seen in the proof of Lemma 14 for the local rescans of **SP-C**, a rescan of v from B_{cur} in phase $t + k$, $k \geq 2$,

requires a path $\mathcal{P} = \langle u = v_0, \ldots, v_k = v \rangle$ of total weight $c(\mathcal{P}) < \Delta$ such that node v_i is scanned from B_{cur} in phase $t + i$, and $\text{tent}(u) + c(\mathcal{P}) < \text{tent}(v)$. However, due to the definition of $G_n^2(r)$, if such a path exists in G, then the graph also contains the edge (u, v) having weight $c(u, v) \leq c(\mathcal{P})$. Hence, the relaxation of the edge (u, v) after scanning u in phase t will immediately reduce $\text{tent}(v)$ so much that no further improvement can be achieved via the path $\mathcal{P}$. Therefore, if at all, v is re-inserted at most once more for phase $t + 1$.

Random geometric graphs have a very special structure, and their edge weights obey the triangle-inequality. It would be interesting to identify weaker conditions under which our algorithms reach the stated high-probability bounds even though the random edge weights are dependent.

In order to show the linear average-case complexity of **SP-C** with improved edge relaxations (Section 3.7) we used independence only in the proof of Lemma 17. It is unclear whether the same result can be shown for dependent random edge weights.

Negative Edge Weights

Shortest-paths instances with (some) negative edge weights are apparently much more difficult to solve than instances where all weights are non-negative: the best known worst-case bound for sequential SSSP with arbitrary edge weights, $\mathcal{O}(n \cdot m)$, is due to the classic Bellman-Ford algorithm [15, 50]; only recently Fakcharoenphol and Rao [47] presented an $\mathcal{O}(n \cdot \log^3 n)$ algorithm for planar graphs. The best known average-case bound was given by Cooper *et al.* [32]; they show $\mathcal{O}(n^2)$ expected time for the *complete graph* with random edge weights generated according to the vertex-potential model [25]. Their algorithm exploits two structural properties that are exhibited on complete graphs with high probability. Firstly, the number of edges can be reduced significantly without changing the shortest-path weights. Secondly, each shortest path consists of at most $\mathcal{O}(\log n)$ edges.

However, for arbitrary graphs with random real edge weights, these conditions will usually not hold. It would be interesting to see whether our methods of adaptive bucket-splitting can result in any improvement for the average-case complexity of SSSP on such graphs.

Chapter 4

Parallel Algorithms

Having dealt with sequential single-source shortest-paths algorithms in the previous chapter we now turn to parallel SSSP. Some of our results are straightforward parallelizations of the sequential label-correcting approaches with buckets of Chapter 3; we show how the operations of a phase for the current bucket can be done in parallel and investigate how many phases are needed. For certain graph classes this simple strategy already yields good results. However, with an alternative scanning rule the number of phases can sometimes be reduced significantly without increasing the total number of operations. In particular, we give the first algorithms that achieve both sublinear average-case time and linear average-case work for random graphs, and graphs modeling the WWW, telephone calls or social networks. In order to facilitate easy exposition we focus on CRCW PRAM algorithms and only sketch extensions to DMMs.

This chapter is organized as follows: Section 4.1 provides an overview of previous and related work. Then we give a list of our contributions (Section 4.2). Before we discuss further details of our new algorithms we review some basic PRAM results and techniques in Section 4.3. Sections 4.4 and 4.5 deal with parallelizations of the sequential label-correcting algorithms from Chapter 3. Further improvements for random graphs are discussed in Section 4.6. Eventually, Section 4.7 provides our most efficient parallel SSSP algorithm, which uses different step widths on disjoint node sets at the same time. Some concluding remarks (also concerning potential parallelizations of Goldberg's new algorithm [67]) are given in Section 4.8.

4.1 Previous and Related Work

4.1.1 PRAM Algorithms (Worst-Case Analysis)

The parallel SSSP problem has so far resisted solutions that are fast and work-efficient at the same time: no PRAM algorithm is known that terminates with $\mathcal{O}(n \cdot \log n + m)$ work and sublinear running time for arbitrary digraphs with nonnegative edge weights. The $\mathcal{O}(n \cdot \log n + m)$-work solution by Driscoll *et al.* [44] (refining a result of Paige and Kruskal [116]) 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.* [20]. The algorithms above scan the nodes one by one in the order of Dijkstra's algorithm; only edge relaxations are performed in parallel. Hence, using

this method there is no possibility to break the worst-case time-bound of $\Omega(n)$. All other known SSSP algorithms for arbitrary graphs trade running time against efficiency.

The algorithm by Han *et al.* [78] (based on [35]) implicitly solves the APSP problem by reducing the shortest-paths computation to matrix multiplications over semirings: it needs $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n^3 \cdot (\log\log n/\log n)^{1/3})$ work. Applying randomized minimum computations [58] on a CRCW PRAM, the algorithm can also be implemented to run in $\mathcal{O}(\log n)$ time using $\mathcal{O}(n^3 \cdot \log n)$ work. Deterministically, it is possible to achieve $\mathcal{O}(\epsilon^{-1} \cdot \log n)$ time using $\mathcal{O}(n^{3+\epsilon} \cdot \log n)$ work for an arbitrary constant $\epsilon > 0$. Furthermore, there is a randomized algorithm [92] for SSSP on sparse graphs with integral nonnegative edge weights summing to W. It requires $\mathcal{O}(\text{polylog}(m+W))$ time and $\mathcal{O}((n+m)^2 \cdot \text{polylog}(m+W))$ work. Recently, Mulmuley and Shah [113] gave a lower bound of $\Omega(\log n)$ execution time for SSSP on PRAMs without bit operations using a polynomial number of processors. The lower bound even holds when the bit lengths of the edge weights are restricted to be of size $\mathcal{O}(\log^3 n)$.

Several parallel SSSP algorithms are based on the randomized parallel breadth-first search (BFS) algorithm of Ullman and Yannakakis [141]. In its simplest form, the BFS algorithm first performs $(\sqrt{n} \cdot \log n)$-limited searches from $\mathcal{O}(\sqrt{n})$ randomly chosen distinguished nodes in parallel. Then it builds an auxiliary graph of the distinguished nodes with edge weights derived from the limited searches and solves an APSP problem on this auxiliary graph. Finally, the distance values of non-distinguished nodes are updated. This simple BFS algorithm takes $\mathcal{O}(\sqrt{n} \cdot \text{polylog}(n))$ time using $\mathcal{O}(\sqrt{n} \cdot m \cdot \text{polylog}(n))$ work with high probability. A more involved version achieves $\mathcal{O}(t \cdot \text{polylog}(n))$ time using $\mathcal{O}((\sqrt{n} \cdot m + n \cdot m/t + n^3/t^4) \cdot \text{polylog}(n))$ work for any $t \leq \sqrt{n}$ whp.

Klein and Subramanian [93] extended the BFS idea of Ullman and Yannakakis to weighted graphs. They gave a parallel randomized approximation scheme for $(1+\epsilon)$-approximate single-source shortest-paths computations that runs in $\mathcal{O}(\sqrt{n} \cdot \epsilon^{-1} \cdot \log n \cdot \log^* n)$ time using $\mathcal{O}(\sqrt{n} \cdot m \cdot \log n)$ work. Furthermore, they showed how to use the result above to compute exact single-source shortest-paths with maximum path weight L by solving a series of $\mathcal{O}(\log L)$ sub-instances. The algorithm takes $\mathcal{O}(\sqrt{n} \cdot \log L \cdot \log n \cdot \log^* n)$ time and $\mathcal{O}(\sqrt{n} \cdot m \cdot \log L \cdot \log n)$ work.

Similar results have been obtained by Cohen [28], and Shi and Spencer [131]. Recently, Cohen [29] gave an $(1+\epsilon)$-approximation algorithm for undirected graphs that runs in polylogarithmic time and takes near linear work. Unfortunately, there seems to be no way to use it for exact computations by repeated approximations. Cohen also proposed a SSSP algorithm for graphs with a given $\mathcal{O}(n^\mu)$-separator decomposition that takes polylogarithmic time and $\mathcal{O}((n + n^{3\cdot\mu}) \cdot \text{polylog}(n))$ work.

More efficient parallel SSSP algorithms have been designed for special graph classes. Here are some examples: Combining the data structure of [20] with the ideas from [140] gives an algorithm which solves the SSSP problem on planar digraphs with arbitrary nonnegative edge weights in $\mathcal{O}(n^{2\cdot\epsilon} + n^{1-\epsilon})$ time and $\mathcal{O}(n^{1+\epsilon})$ work on a CREW PRAM. In contrast, the randomized algorithm of [92] requires planar graphs and integral edge weights summing to W. It runs in $\mathcal{O}(\text{polylog}(n+W))$ time using $\mathcal{O}(n \cdot \text{polylog}(n+W))$ work. Work-efficient SSSP algorithms for planar layered graphs have been proposed by Subramanian *et al.* [134] and Atallah *et al.* [10]. Furthermore, there is an $\mathcal{O}(\log^2 n)$-time linear-work

EREW PRAM algorithm for graphs with constant tree width [23].

4.1.2 PRAM Algorithms (Average-Case Analysis)

Random graphs [17, 46] with *unit weight* edges have been considered by Clementi *et al.* [27]: their solution is restricted to edge probabilities $\Theta(1)$ or $\Theta(\log^k n/n)$ for $k > 1$. In the latter case $\mathcal{O}(\log^{k+1} n)$ time and optimal $\mathcal{O}(n \cdot \log^k n)$ work is needed on the average.

Reif and Spirakis [125] bounded the expected diameter of the giant component of sparse random graphs with unit weights by $\mathcal{O}(\log n)$. Their result implies that the matrix based APSP algorithm needs $\mathcal{O}(\log \log n)$ iterations on average provided that the edge weights are nonnegative and satisfy the triangle inequality.

Frieze and Rudolph [58], and Gu and Takaoka [75] considered the APSP problem with random edge weights and showed that the standard matrix product algorithm can be implemented in $\mathcal{O}(\log \log n)$ time and $\mathcal{O}(n^3 \cdot \log \log n)$ work on average.

Crauser *et al.* [33] gave the first parallel label-setting algorithms that solve SSSP for random graphs with random edge weights in sublinear time using $\mathcal{O}(n \cdot \log n + m)$ work on the average. Their algorithms maintain the set of candidate nodes in a parallel priority queue, which is implemented by a number of sequential relaxed heaps [44] with random node assignment. The algorithms operate in phases: each phase first identifies a set of candidate nodes with final tentative distances; then these nodes are scanned in parallel. The *OUT-approach* applies the OUT-criterion of Lemma 6 in order to find scan-eligible nodes. It needs $\Theta(\sqrt{n})$ phases on the average. Similar performance on random graphs is obtained if the OUT-criterion is replaced by the IN-criterion from Lemma 5 (*IN-approach*). Changing between IN- and OUT-criterion after each phase (*INOUT-approach*), may speed-up the computation; for random graphs, however, $\Omega(n^{1/3})$ phases are still needed on the average.

The analysis for the OUT-approach was generalized in [103] to arbitrary graphs with random edge weights and maximum shortest-path weight $\mathcal{L} = \max\{\text{dist}(v) \mid \text{dist}(v) < \infty\}$: the algorithm achieves $T_H = \mathcal{O}(\mathbf{E}[\sqrt{\mathcal{L} \cdot m}] \cdot \log^3 n)$ average-case time using $\mathcal{O}(n \cdot \log n + m + T_H)$ work.

4.1.3 Algorithms for Distributed Memory Machines

PRAM algorithms can be emulated on distributed memory machines. The loss factors depend on the concrete parameters of the models; e.g. see [61] for emulation results on the BSP model. However, existing implementations [1, 22, 84, 85, 86, 139] on parallel computers with distributed memory usually avoid such emulations; they rather apply some kind of graph partitioning, where each processor runs a *sequential* label-correcting algorithm on its subgraph(s). Heuristics are used for the frequency of the inter-processor data-exchange concerning tentative distances, load-balancing, and termination detection. Depending on the input classes and parameter choices, some of these implementations perform fairly well, even though no speed-up can be achieved in the worst case. However, no theoretical average-case analysis has been given.

4.2 Our Contribution

We present new results on the average-case complexity of parallel SSSP assuming independent random edge weights uniformly drawn from $[0, 1]$. The average-case performance of our algorithms is expressed in terms of rather general properties of the input graph classes like the maximum shortest-path weight or the node degree sequence. For the purpose of an easy exposition we concentrate on the CRCW PRAM (**C**oncurrent-**R**ead **C**oncurrent-**W**rite **P**arallel **R**andom-**A**ccess **M**achine) [52, 72, 88] model of computation. However, we shall sketch extensions to distributed memory machines when this is appropriate.

Straightforward Parallelizations

In Section 4.4 we prove that already a simple parallelization of the **Δ-Stepping** algorithm from Section 3.9.1 performs reasonably well on a large class of graphs: for maximum shortest-path weight $\mathcal{L}$ (see Definition 2) and maximum node degree d, the parallelization takes $\mathcal{O}(\mathbf{E}[d \cdot \lceil d \cdot \mathcal{L} \rceil] \cdot \log^2 n)$ time using $\mathcal{O}(n + m + \mathbf{E}[d \cdot \lceil d \cdot \mathcal{L} \rceil] \cdot \log^2 n)$ work on the average. If $\mathcal{L}' \geq \mathcal{L}$ and $d' \geq d$ denote high-probability bounds on $\mathcal{L}$ and d, then an improved version of the parallel Δ-Stepping presented in Section 4.5 achieves $\mathcal{O}(\lceil d' \cdot \mathcal{L}' \rceil \cdot \log^2 n)$ time using $\mathcal{O}(n + m + \lceil d' \cdot \mathcal{L}' \rceil \cdot \log^2 n)$ work whp.

For several important graph classes with random edge weights, $\mathcal{L}$ is sufficiently small, i.e., $\lceil d \cdot \mathcal{L} \rceil \cdot \log^2 n = o(n)$ with high probability; in that case the improved parallel Δ-Stepping runs in sublinear time and linear work with high probability. A typical example are random graphs from the class $D(n, \bar{d}/n)$ as defined in Section 3.9.3: by Lemma 26 and Corollary 6 we already know $d = \mathcal{O}(\bar{d} + \log n)$ and $\mathcal{L} = \mathcal{O}(\frac{\log^2 n}{\bar{d}})$ with high probability. This implies that SSSP on random graphs from $D(n, \bar{d}/n)$ with random edge weights can be solved in $\mathcal{O}(\log^4 n + (1/\bar{d}) \cdot \log^5 n)$ time using linear work with high probability.

Improvements for Random Graphs

In Section 4.6 we reconsider random graphs with random edge weights. We will improve the high-probability bound on $\mathcal{L}$ from Corollary 6 to $\mathcal{O}((1/\bar{d}) \cdot \log n)$. For random graphs from $D(n, \bar{d}/n)$, the step width for the advanced parallel Δ-Stepping algorithm may be chosen as big as $\Delta = \Theta(1/\bar{d})$ instead of $\Theta(1/(\bar{d} + \log n))$. Together with an appropriate preprocessing of the input graph this yields the following result: SSSP on random graphs with random edge weights can be solved in $\mathcal{O}(\log^2 n)$ time using linear work on the average.

Improved Parallelizations for Graphs with Unbalanced Node Degrees

For arbitrary graphs, the simple parallelizations above require $\Omega(d \cdot \mathcal{L})$ phases. This is often too much if d and $\mathcal{L}$ happen to be large. In Section 4.7 we consider parallel SSSP for graphs with *unbalanced node degrees*, that is, input graphs, where the maximum node degree d is significantly larger than the average node degree $\bar{d}$. Typical examples are degree sequences that follow a *power law*: the number of nodes, y, of a given in-degree x is proportional to $x^{-\beta}$ for some constant parameter $\beta > 0$. For most massive graphs, $2 < \beta \leq 4$: for example, Kumar *et al.* [95] and Barabasi *et al.* [13] independently reported $\beta \approx 2.1$ for the in-degrees of the WWW graph, and the same value was estimated for telephone call

graphs [5]. In spite of constant average in-degree $\bar{d}$, one expects to observe at least one node with in-degree $\Omega(n^{1/(\beta-1)})$, i.e., $d = \Omega(n^{0.9})$ for $\beta = 2.1$. The observed diameters are usually very small; however, $\mathcal{L} = \Omega(1)$.

Unfortunately, if $\mathcal{L} = \Omega(1)$, all parallelizations introduced above require $\Omega(d)$ time. This drawback is partially removed by our new parallel label-correcting method, the **Parallel Individual Step-Widths SSSP (PIS-SP)** algorithm: it utilizes a number of different approximate priority-queue data structures in order to maintain nodes of similar degrees in the same queue. The algorithm applies different step widths for different queues at the same time. The approach can be easily implemented with a collection of sequential relaxed heaps. In that case it requires

$$T_H = \mathcal{O}\left(\log^3 n \cdot \mathbf{E}\left[\min_i \left\{\lceil 2^i \cdot \mathcal{L}\rceil + |v \in G, \text{in-degree}(v) > 2^i|\right\}\right]\right) \text{ time}$$

using $\mathcal{O}(n \cdot \log n + m + T_H)$ work on average. By virtue of a split-free bucket structure and an improved node selection strategy, another logarithmic factor can be gained on both running time and work. The resulting algorithm is the first to achieve $o(n^{1/2})$ average-case time and $\mathcal{O}(n + m)$ work for graphs modeling the WWW or telephone calls.

Comparison

Table 4.1 provides an overview of our new parallel SSSP algorithms. For each algorithm we list average-case time and work as functions of node degrees and maximum shortest-path weight. We have suitably instantiated these general formulae for three sample graph classes in order to demonstrate the performance differences. As already discussed for random graphs, even better results can be obtained if additional knowledge concerning the input graphs is used to optimize the parameter setting of the algorithms. For comparison we also listed the respective numbers of the best previous $\mathcal{O}(n \cdot \log n + m)$-work algorithm, the label-setting OUT-approach of [33].

It turns out that for all graph classes listed in Table 4.1 at least one of the new algorithms is both faster and requires less work than the OUT-approach. In particular, the **PIS-SP** algorithm is uniformly better. For random graphs, the performance gain is significant. In fact, the average-case execution-time of the two algorithms comes closest for graphs with $\Theta(\mathbf{E}[\sqrt{\mathcal{L} \cdot m}])$ nodes of degree $\Theta(m/\mathbf{E}[\sqrt{\mathcal{L} \cdot m}])$ each: in that case the OUT-approach requires $T_1 := \mathcal{O}(\mathbf{E}[\sqrt{\mathcal{L} \cdot m}] \cdot \log^3 n)$ time on average using $\mathcal{O}(n \cdot \log n + m + T_1)$ work whereas **PIS-SP** succeeds after $T_2 := \mathcal{O}(\mathbf{E}[\sqrt{\mathcal{L} \cdot m}] \cdot \log^2 n)$ time with $\mathcal{O}(n + m + T_2)$ work on the average.

In fairness, however, it should be mentioned that the OUT-approach may be augmented by intermediate applications of the IN-criterion (as it was done in [33] for random graphs). Still, the resulting average-case running times of the augmented algorithm will be in $\Omega(m^{1/3})$ for the general case, even if d and $\mathcal{L}$ are small. Furthermore, on sparse graphs, the augmented algorithm will require superlinear average-case time. On the other hand, it is an interesting open problem, whether the OUT-criterion (or some modification of it) can be profitably included into our linear work algorithms based on buckets.

Early results on the average-case analysis of our parallel SSSP algorithms on random graphs

Algorithm	Average-Case Performance
Simple Parallel Δ-Stepping (Section 4.4)	$T = \mathcal{O}(\mathbf{E}[d \cdot \lceil d \cdot \mathcal{L} \rceil] \cdot \log^2 n)$ $W = \mathcal{O}(n + m + T)$ RAND: $T = \mathcal{O}(\log^5 n)$ POW: $T = \Omega(n^{2/(\beta-1)})$ WWW: $T = \Omega(n)$ ART: $T = \Omega(n)$
Advanced Parallel Δ-Stepping (Section 4.5)	$T = \mathcal{O}(\mathbf{E}[\lceil d \cdot \mathcal{L} \rceil] \cdot \log^2 n)$ $W = \mathcal{O}(n + m + T)$ RAND: $T = \mathcal{O}(\log^4 n)$ POW: $T = \Omega(n^{1/(\beta-1)})$ WWW: $T = \Omega(n^{0.9})$ ART: $T = \mathcal{O}(n^{3/4} \cdot \log^2 n)$
Parallel Individual Step-Widths SSSP **PIS-SP** (Section 4.7)	$T = \mathcal{O}(\log^2 n \cdot \mathbf{E}[\min_i\{\lceil 2^i \cdot \mathcal{L} \rceil + \lvert v \in G, \text{in-degree}(v) > 2^i \rvert\})$ $W = \mathcal{O}(n + m + T)$ RAND: $T = \mathcal{O}(\log^4 n)$ POW: $T = \mathcal{O}(n^{1/\beta} \cdot \log^3 n)$ WWW: $T = o(n^{0.48})$ ART: $T = \mathcal{O}(n^{1/4} \cdot \log^2 n)$
Best Previous: Label-Setting OUT-Approach [33, 103]	$T = \mathcal{O}(\mathbf{E}[(\mathcal{L} \cdot m)^{1/2}] \cdot \log^3 n)$ $W = \mathcal{O}(n \cdot \log n + m + T)$ RAND: $T = \mathcal{O}(n^{1/2} \cdot \log^{3.5} n)$ POW: $T = \mathcal{O}(n^{1/2} \cdot \log^{3.5} n)$ WWW: $T = \mathcal{O}(n^{1/2} \cdot \log^{3.5} n)$ ART: $T = \mathcal{O}(n^{5/8} \cdot \log^3 n)$

Table 4.1: Average-case performance of our parallel SSSP algorithms on graphs with maximum node degree d and maximum shortest-path weight $\mathcal{L}$. For each algorithm, T and W denote average-case parallel time and work, respectively. We give concrete instantiations of T for three graph classes with $\mathcal{O}(n)$ edges and random edge weights: RAND denotes sparse random graphs, POW refers to graphs with $\mathbf{E}[\mathcal{L}] = \mathcal{O}(\log n)$, where the node in-degrees follow a power law with parameter β. WWW is a subclass of these graphs having $\beta = 2.1$, i.e., $d = \Omega(n^{0.9})$ and $o(n^{0.48})$ nodes of in-degree $\Omega(n^{0.48})$ in expectation. ART denotes an artificial graph class with $\mathbf{E}[\mathcal{L}] = \Theta(n^{1/4})$, where all but $\Theta(\log n)$ nodes have constant degree, the remaining nodes have degree $\Theta(n^{1/2})$.

have been published in [33, 106]; a version of the advanced parallel Δ-Stepping for non-random graphs appeared in [107]. Parallelizations of the adaptive bucket-splitting algorithm have been sketched in [104]. A precursor of the **PIS-SP** algorithm [105] has been presented in [103].

4.3 Basic Facts and Techniques

In this section we list a few facts and methods for PRAM algorithms, which we will use later on. It is *not* meant to be a tutorial on parallel algorithms; appropriate textbooks are, e.g., [6, 62, 63, 88, 96]. Readers with some background-knowledge in parallel algorithms may choose to skip this section.

Our PRAM algorithms are usually described in the *work-time framework*, a generalization of *Brent's scheduling principle* [18]: if a parallel algorithm does q operations (work) in time t, then on P processors, the same algorithm can be done in time t_P where $t_P = \mathcal{O}(t + \frac{q}{P})$. This "slow-down" feature facilitates the following accounting scheme: if an algorithm A consists of sub-algorithms $A_1, \ldots, A_k$, where A_i can be solved in t_i time using q_i work for $i = 1, \ldots, k$, then A can be executed in time $\mathcal{O}(\sum_{i=1}^{k} t_i)$ using $\mathcal{O}(\sum_{i=1}^{k} q_i)$ work. In the following we will state the time bounds of a few basic work-optimal PRAM sub-algorithms; see also Table 4.2.

Algorithm	Time	Work	Model
(Segmented) Prefix/Suffix Sums [97]	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	EREW
Fast Min/Max [132]	$\mathcal{O}(\log\log n)$	$\mathcal{O}(n)$	CRCW
Randomized Min/Max [58]	$\mathcal{O}(1)$ whp	$\mathcal{O}(n)$ whp	CRCW
Sorting n integers from $\{1, \ldots, n\}$ [122]	$\mathcal{O}(\log n)$ whp	$\mathcal{O}(n)$ whp	CRCW

Table 4.2: Performances for some basic PRAM algorithms

Many parallel algorithms use *prefix* and *suffix sums* in order to assign sub-problems to processors: consider a sequence $X = (x_1, x_2, \ldots, x_n)$ of n elements from some set S together with a binary *associative* operation $*$, e.g., sum, product, minimum or maximum; the *prefix sums* of X are the n expressions $s_i = x_1 * x_2 * \ldots * x_i$, where $1 \leq i \leq n$. Equivalently, the *suffix sums* are defined by $s'_i = x_i * x_{i+1} * \ldots * x_n$, for $1 \leq i \leq n$. The *segmented* version of prefix sums assumes an additional Boolean array Y of length n such that $y_1 = y_n = 1$; for each $i_1 < i_2$ where $y_{i_1} = y_{i_2} = 1$ and $y_j = 0$ for all $i_1 < j < i_2$, we wish to compute the prefix sums of the sub-array $(x_{i_1+1}, \ldots, x_{i_2})$ of X. For the segmented suffix sums we are interested in the sub-arrays $(x_{i_1}, \ldots, x_{i_2-1})$. Using balanced binary trees, all these prefix/suffix computations can be done in $\mathcal{O}(\log n)$ time with $\mathcal{O}(n)$ work [97].

Frequently, (small) sets of data must be copied to (many) other memory cells in parallel: let $X = (x_1, x_2, \ldots, x_n)$ be an array of n elements, and let $j_1 = 1 < j_2 < \ldots < j_s = n$ be a set of indices. The *segmented broadcasting problem* is that of computing the array $Y = (y_1, y_2, \ldots, y_n)$ such that $y_l = x_{j_i}$, for $j_{i-1} < l \leq j_i$ and $2 \leq i \leq s$. It can be solved with the same resource usage as the prefix/suffix problems.

Computing a single minimum or maximum is a special case of the prefix sum problem. On a CRCW PRAM, it can be done in $\mathcal{O}(\log\log n)$ time and linear work [132]. The algorithm uses the method of accelerated cascading [30] together with a constant-time algorithm that requires quadratic work. Based on this worst-case constant-time approach, Frieze and Rudolph [58] developed a randomized CRCW algorithm that runs in $\mathcal{O}(1)$ time while requiring only linear work with high probability.

Integer sorting is a simple way to reorganize data items. In particular, it is useful to bundle distributed data items that belong together: it is sufficient to assign common integers to the items of the same sub-groups and then sort the items according to the associated integers. In [122] it is shown how n integers with values in $\{1, \ldots, n\}$ can be sorted in $\mathcal{O}(\log n)$ time using a linear number of operations with high probability.

4.4 Simple Parallel Δ-Stepping

After having listed a few general-purpose PRAM routines in Section 4.3 we now turn to our first parallelization of an SSSP algorithm. Concretely speaking, we will provide a simple parallel version of ABI-Dijkstra and the sequential Δ-Stepping from Section 3.9. Recall that these algorithms keep the candidate nodes in an approximate priority-queue $B[\cdot]$ with buckets of width $\Delta = 2^{-\lceil \log d \rceil}$, where d denotes the maximum node degree in the input graph. That is, a queued node v having tentative distance $\mathrm{tent}(v)$ is stored in bucket $B[\lfloor \mathrm{tent}(v)/\Delta \rfloor]$. The current bucket B_{cur} denotes the first nonempty bucket of $B[\cdot]$. In a *phase*, all nodes in the current bucket are scanned (where the Δ-Stepping initially only relaxes light edges); after the current bucket becomes empty (and the Δ-Stepping has relaxed the associated heavy edges as well), the algorithms sequentially search for the next nonempty bucket. The simple parallelizations of this section will be restricted to the operations within a phase; searching for the next non-empty bucket is still done sequentially (in Section 4.7 we propose improved algorithms that support parallel search for the next non-empty buckets).

As already seen in Section 3.9, for maximum shortest-path weight $\mathcal{L}$, the algorithms traverse $\lceil \mathcal{L}/\Delta \rceil = \mathcal{O}(\lceil d \cdot \mathcal{L} \rceil)$ buckets; also compare Remark 1 in Section 3.4.4. The number of phases, however, may be bigger since scanned nodes can be re-inserted into the current bucket. Hence, we are left with two problems:

(a) Providing efficient parallelizations for a phase.

(b) Bounding the number of phases.

The total work over all phases should be $\mathcal{O}(n+m)$ on the average. Disregarding some initialization procedures, the total parallel time of the resulting algorithms will be given by the sum of the times for the phases. An upper bound on the number of phases is easily obtained: from Lemma 14 and Lemma 22 we can conclude that in the case of random edge weights there are at most $\mathcal{O}(\frac{\log n}{\log \log n})$ phases for each current bucket with high probability. As the current bucket is advanced at most $\mathcal{O}(\lceil d \cdot \mathcal{L} \rceil)$ times we find:

Corollary 9 *For graphs with random independent edge weights uniformly distributed in* $[0,1]$*, maximum shortest-path weight* $\mathcal{L}$ *and maximum node degree* d*, both ABI-Dijkstra and* Δ*-Stepping require* $\mathcal{O}(\mathbf{E}[\lceil d \cdot \mathcal{L} \rceil] \cdot \frac{\log n}{\log \log n})$ *phases on the average. If* $\mathcal{L}' \geq \mathcal{L}$ *and* $d' \geq d$ *denote upper bounds (that hold with high probability) then the number of phases is* $\mathcal{O}(\lceil d' \cdot \mathcal{L}' \rceil \cdot \frac{\log n}{\log \log n})$ *with high probability, too.*

4.4.1 Parallelizing a Phase via Randomized Node Assignment

If it is possible to scan several nodes during a phase in parallel then this work has to be distributed among the available $p \leq n$ processors in a load-balanced way. Furthermore, finding an appropriate distribution itself should not consume too much time and work. A simple yet efficient load-balancing method is to apply an initial *random* assignment of graph nodes to processors. Using an array $\mathrm{ind}[\cdot]$ of independent random PU indices uniformly distributed in $\{0,\ldots,p-1\}$, entry $\mathrm{ind}[v]$ gives the PU responsible for node v. The bucket structure is distributed over the processors, too. In a phase, each PU does the work for the

nodes randomly assigned to its own structure. That is, for $0 \leq i < p$, PU P_i is in charge of the bucket array $B_i[\cdot]$, and a queued node v will be kept in $B_{\text{ind}[v]}[\lfloor \text{tent}(v)/\Delta \rfloor]$. A phase of the simple parallelization comprises the following basic steps:

Step 1: *Identifying the global value* $\lfloor M/\Delta \rfloor$*, where* $M := \min\{\text{tent}(v) : v \text{ queued}\}$.

Step 2: *Node removal from the respective buckets* $B_i[\lfloor M/\Delta \rfloor]$.

Step 3: *Generating requests for edge relaxations.*

Step 4: *Assigning requests to their responsible PUs.*

Step 5: *Performing the relaxations according to the assigned requests.*

Step 1: Finding $\lfloor M/\Delta \rfloor$. The key feature of ABI-Dijkstra and Δ-Stepping is to restrict node scanning to those queued nodes v having $\text{tent}(v) < (\lfloor M/\Delta \rfloor + 1) \cdot \Delta$, where M denotes the smallest tentative distance among all queued nodes. In the sequential version (without cyclical bucket re-usage), $\lfloor M/\Delta \rfloor$ is simply the index of the first non-empty bucket $B[\lfloor M/\Delta \rfloor]$. Now that in the parallel version $B[\cdot]$ is distributed over p arrays $B_i[\cdot]$, the algorithm needs to find the globally smallest index j $(= \lfloor M/\Delta \rfloor)$ among all non-empty buckets $B_i[j]$ and all $0 \leq i < p$.

Let M' denote the smallest tentative distance among all queued nodes at the beginning of the previous phase. Starting from $B_i[\lfloor M'/\Delta \rfloor]$, for $0 \leq i < p$, PU P_i sequentially searches for the first non-empty bucket in $B_i[\cdot]$. Skipping k empty buckets can be done in time $\mathcal{O}(k + \log n)$: each PU can traverse empty buckets of its dedicated bucket structure in constant time per bucket; every $\Theta(\log n)$ iterations it is checked whether any PU has found a nonempty bucket and, if so, the globally smallest index with a nonempty bucket is found. The latter check can be done with the basic gather- and broadcasting algorithms of Section 4.3 in $\mathcal{O}(\log p) = \mathcal{O}(\log n)$ time.

Steps 2 and 3: Dequeuing nodes and generating requests. Once the value of $\lfloor M/\Delta \rfloor$ has been detected, PU P_i removes all nodes from $B_i[\lfloor M/\Delta \rfloor]$ in Step 2. Let us denote the resulting node set by R_i. Afterwards PU P_i is in charge of generating requests $(w, \text{tent}(v) + c(v, w))$ for edges $(v, w) \in E$ emanating from nodes $v \in R_i$. Depending on whether we are parallelizing ABI-Dijkstra or the Δ-Stepping, requests are created for either all outgoing edges or (as long as the current bucket does not change) just for the light outgoing edges, which have weight at most Δ. Let $Req(v)$ be the set of requests generated for node v. The work of an arbitrary PU P_i for a node $v \in R_i$ during steps 2 and 3 can a seen as job of size $1 + |Req(v)| = \mathcal{O}(d)$ whose execution takes time linear in its size.

Due to the random assignment of nodes to processors, the work performed during steps 2 and 3 can be analyzed using classical results of the occupancy ("balls into bins") problem [89, 94, 112, 129]. In particular, arguing along the lines of [7, Lemma 2] one finds:

Lemma 35 *Consider any number of subproblems of size in* $[1, k]$*. Let* K *denote the sum of all subproblem sizes. If the subproblems are allocated uniformly and independently at random to* p *PUs, then the maximum load of any PU will be bounded by* $\mathcal{O}(K/p + k \cdot \log(n + p))$ *with high probability.*[1]

[1] The seemingly unrelated parameter n comes into play since we based our definition of "whp" on it.

Proof: Consider an arbitrary but fixed PU P_i. It suffices to show that PU P_i receives load exceeding $\alpha \cdot (K/p + k \cdot \ln(n+p))$ with probability at most $n^{-\beta}/p$ for any $\beta > 0$ and an appropriately chosen α depending on β only. Let there be r subproblems, where $k_0,\ldots,$ k_{r-1} denote the respective sizes of the these subproblems. Define the 0-1 random variable X_j to be one if and only if subproblem j is assigned to P_i. Let $Y := \sum_j k_j \cdot X_j/k$ denote the "normalized load" received by PU P_i. Note that $\mathbf{E}[Y] = K/(p \cdot k)$. Since the X_j are independent and the weights k_j/k are in $(0,1]$, we can apply another version of the Chernoff bound [121, Theorem 1] which states that for any $\gamma > 1$,

$$\mathbf{P}\left[Y > \gamma \cdot \mathbf{E}[Y]\right] \leq \left(\frac{e^{\gamma-1}}{\gamma^\gamma}\right)^{\mathbf{E}[Y]} .$$

First we rewrite $\frac{e^{\gamma-1}}{\gamma^\gamma} = \frac{e^{\gamma-1}}{e^{\gamma\cdot\ln\gamma}} = e^{(1-\frac{1}{\gamma}-\ln\gamma)\cdot\gamma}$. Then by setting $\gamma \cdot \mathbf{E}[Y] = \alpha \cdot (K/p + k \cdot \ln(n+p))$ we get $\gamma = \alpha \cdot (k + p \cdot k^2 \cdot \ln(n+p)/K) \geq \alpha$. Now we can start to simplify:

$$\begin{aligned}
\mathbf{P}\left[Y > \gamma \cdot \mathbf{E}[Y]\right] &\leq e^{(1-\frac{1}{\gamma}-\ln\gamma)\cdot\gamma\cdot\mathbf{E}[Y]} = e^{(1-\frac{1}{\gamma}-\ln\gamma)\cdot\alpha\cdot(K/p+k\cdot\ln(n+p))} \\
&\leq e^{(1-\ln\alpha)\cdot\alpha\cdot\ln(n+p)} \leq e^{-\alpha\cdot\ln(n+p)} \text{ for } \alpha > e^2 \\
&\leq e^{-(\alpha-1)\cdot\ln n} \cdot e^{-\ln p} = n^{-(\alpha-1)}/p \ .
\end{aligned}$$

Setting $\alpha = \max\left\{\beta+1, e^2\right\}$ concludes the proof. □

Thus, if a parallel phase of ABI-Dijkstra or Δ-Stepping with $p \leq n$ PUs removes $|R|$ nodes and generates $|Req|$ requests, then Steps 2 and 3 take $\mathcal{O}\left((|R| + |Req|)/p + d \cdot \log n\right)$ time with high probability according to Lemma 35.

Steps 4 and 5: Assigning and performing relaxation requests. Let Req_i be the set of requests generated by PU P_i. Any request $(w,x) \in Req_i$ must now be transferred to the bucket structure associated with PU $P_{\text{ind}[w]}$. Using Lemma 35, it can be seen that due to the random indexing, each PU receives $\mathcal{O}\left(|\bigcup Req_j|/p + d \cdot \log n\right)$ requests whp. The value of $|\bigcup Req_j|$ can be obtained and broadcast in $\mathcal{O}(\log n)$ time. Each PU sets up an empty request buffer which is a constant factor larger than needed to accommodate the requests directed to it whp.

The requests are placed by "randomized dart throwing" [108] where PU P_i tries to write $(w,x) \in Req_i$ to a random position of the target buffer of PU $P_{\text{ind}[w]}$ (see Figure 4.1). Several PUs may try to write to the same memory location. This is the only step of the parallelization that needs the CRCW PRAM. Due to the choice of the buffer sizes each single placement succeeds with constant probability. Using Chernoff bounds it is straightforward to see that the dart throwing terminates in time proportional to the buffer size, $\mathcal{O}\left(|\bigcup Req_j|/p + d \cdot \log n\right)$ whp. For the unlikely case that a buffer is too small, correctness can be preserved by checking periodically whether the dart throwing has terminated and increasing the buffer sizes if necessary.

In step 5, each PU examines all slots of its request buffer and performs the associated relaxations in the order they are found. Since no other PUs work on its nodes the relaxations will be atomic.

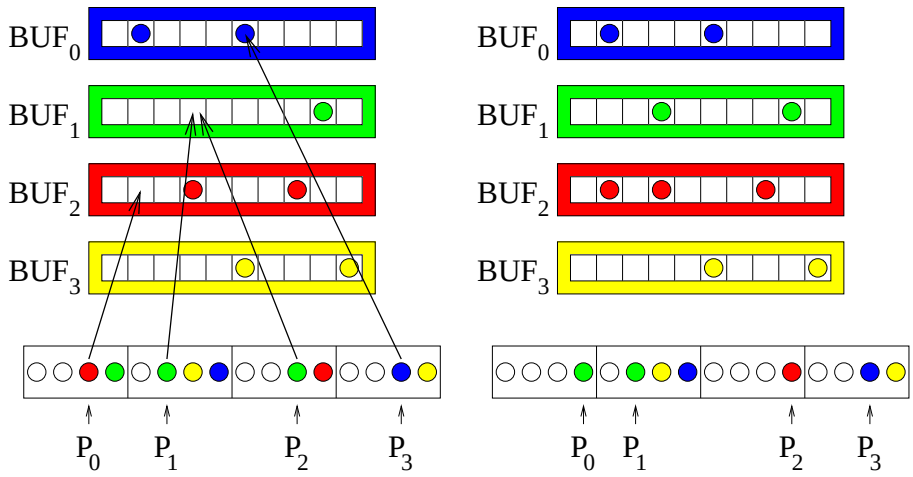


Figure 4.1: Placing requests into buffers by randomized dart-throwing. Processor P_0 succeeds in placing its currently treated request in BUF_2 and can turn to its next request. P_1 and P_2 contend for a free slot in BUF_1. P_2 wins. P_1 has to probe another position in BUF_1. P_3 fails because the chosen position is already occupied.

Theorem 8 (Simple Parallelization.) *Consider directed graphs with n nodes, m edges and independent random edge weights uniformly distributed in $[0,1]$. Let d' and $\mathcal{L}'$ denote upper bounds*[2] *on the maximum node degree d and the maximum shortest-path weight $\mathcal{L}$, respectively. On a CRCW PRAM, the simple parallelization of ABI-Dijkstra and Δ-Stepping with $\Delta = 2^{-\lceil \log_2 d \rceil}$ runs in $T = \mathcal{O}(d \cdot \mathbf{E}[\lceil d \cdot \mathcal{L} \rceil] \cdot \log^2 n)$ time using $\mathcal{O}(n+m+T)$ work on the average. Additionally, for $\Delta := 2^{-\lceil \log_2 d' \rceil}$, the simple parallel Δ-Stepping algorithm requires $T' = \mathcal{O}(d' \cdot \lceil d' \cdot \mathcal{L}' \rceil \cdot \log^2 n)$ time and $\mathcal{O}(n+m+T')$ work with high probability.*

Proof: Using the basic algorithms of Section 4.3, the initialization of the data structures and the reordering of the adjacency lists concerning light and heavy edges can be done in $\mathcal{O}((n+m)/p + \log(n))$ time. By Corollary 9, there are $\mathcal{O}(\mathbf{E}[\lceil d \cdot \mathcal{L} \rceil] \cdot \log n)$ phases on the average and $\mathcal{O}(\lceil d' \cdot \mathcal{L}' \rceil \cdot \log n)$ phases with high probability. If the five steps of the i-th parallel phase remove $|R^{(i)}|$ nodes and generate $|Req^{(i)}|$ requests then this phase takes $\mathcal{O}\left((|R^{(i)}| + |Req^{(i)}|)/p + d \cdot \log n\right)$ time with high probability. From the discussion of the sequential ABI-Dijkstra and Δ-Stepping in Section 3.9 we deduce $\mathbf{E}[\sum_i |R^{(i)}|] = \mathcal{O}(n)$ and $\mathbf{E}[\sum_i |Req^{(i)}|] = \mathcal{O}(n+m)$; for the Δ-Stepping these bounds even hold with high probability. Thus, the total time is bounded by $\mathcal{O}\left(\frac{n+m}{p} + d \cdot \mathbf{E}[\lceil d \cdot \mathcal{L} \rceil] \cdot \log^2 n\right)$ on the average and $\mathcal{O}\left(\frac{n+m}{p} + d' \cdot \lceil d' \cdot \mathcal{L}' \rceil \cdot \log^2 n\right)$ with high probability. The theorem follows by choosing $p = \lceil \frac{n+m}{d \cdot \mathbf{E}[\lceil d \cdot \mathcal{L} \rceil] \cdot \log^2 n} \rceil < n$ PUs for the average-case bound and $p = \lceil \frac{n+m}{d' \cdot \lceil d' \cdot \mathcal{L}' \rceil \cdot \log^2 n} \rceil$ PUs for the high-probability bound. □

[2] Actually, it is suffi cient if these bounds hold with high probability.

4.5 Advanced Parallelizations

The simple parallelization of the phases as shown above is already quite good for sparse graphs like road-maps, where the maximum node degree d is not much larger than m/n. However, if $d \gg m/n$ then the running time becomes less attractive. With an alternative load-balancing, one can save a factor of $\Theta(d)$:

Theorem 9 *With the parameters and machine model defined as in Theorem 8, parallel SSSP can be solved in $T = \mathcal{O}(\mathbf{E}[\lceil d \cdot \mathcal{L} \rceil] \cdot \log^2 n)$ time using $\mathcal{O}(n + m + T)$ work on the average. Additionally, the advanced parallel Δ-Stepping algorithm can be implemented to run in $T' = \mathcal{O}(d' \cdot \lceil d' \cdot \mathcal{L}' \rceil \cdot \log^2 n)$ time using $\mathcal{O}(n + m + T')$ work with high probability.*

In the following we describe the necessary modifications:

4.5.1 Improved Request Generation

The new feature we add is to explicitly organize the generation of requests. Instead of generating the set of requests derived from the bucket structure of PU P_i exclusively by PU P_i, now all PUs cooperate to build the total set of requests. This can be done by computing a prefix sum over the adjacency list sizes of the nodes in the request set first and then assign consecutive groups of nodes with about equal number of edges to the PUs. Nodes with large out-degree may cause several groups containing only edges which emanate from this very node. The extra time needed to compute the prefix sums and schedules is $\mathcal{O}(\log n)$ per phase.

4.5.2 Improved Request Execution

What makes executing requests more difficult than generating them is that the in-degree of a node does not convey how many requests will appear in a particular phase. If some target node v is contained in many requests of a phase then it might even be necessary to set aside several processors to deal with the request for v.

Instead of the brute-force randomized dart-throwing as in Section 4.4, we use an explicit load balancing which groups different requests for the same target and only executing the strictest relaxation. On CRCW PRAMs, grouping can be done efficiently using the semi-sorting routine explained in Lemma 36. Then we can use prefix sums to schedule $\lfloor p \left|\mathrm{Req}(w)\right| / \left|\mathrm{Req}\right| \rfloor$ PUs for blocks of size at least $\left|\mathrm{Req}\right| / p$ and to assign smaller groups with a total of up to $\left|\mathrm{Req}\right| / p$ requests to individual PUs. The PUs concerned with a group collectively find a request with minimum distance in time $\mathcal{O}\left(\left|\mathrm{Req}\right| / p + \log p\right)$. Thus, each target node receives at most one request for relaxation. These selected requests will be load balanced over the PUs whp due to the random assignment of nodes to PUs. Figure 4.2 provides an illustration. Summing over all phases yields the desired bound.

Lemma 36 Semi-sorting n *records with integer keys, i.e., permuting them into an array of size n such that all records with equal key form a consecutive block, can be performed in time $\mathcal{O}\left(n/p + \log n\right)$ on a CRCW-PRAM with high probability.*

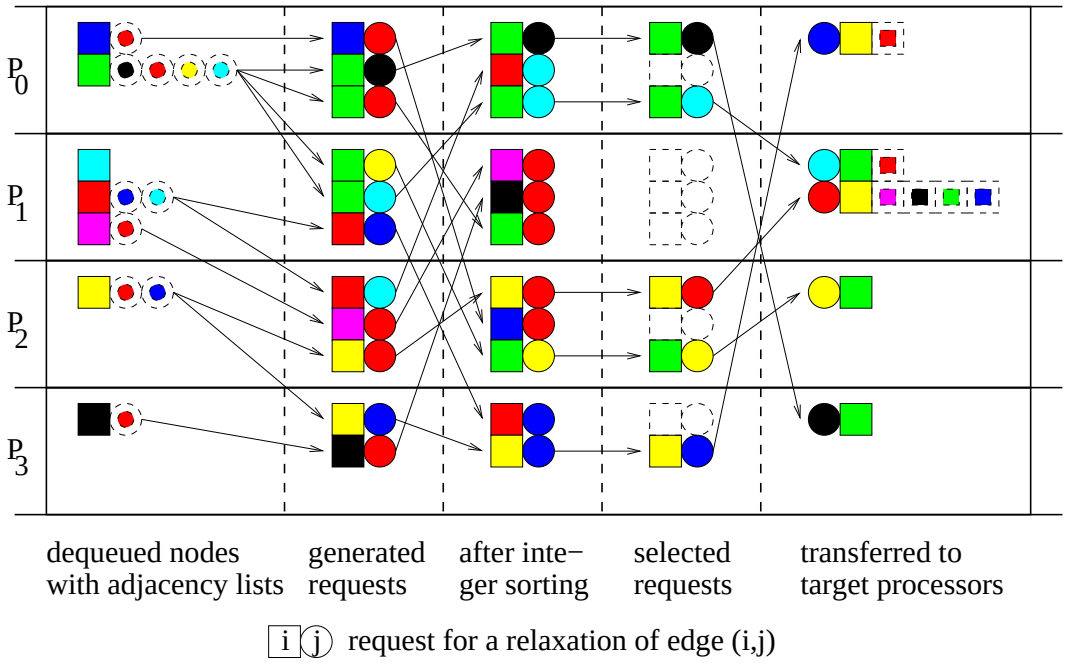


Figure 4.2: Load balancing for generating and performing requests: Requests are denoted by a box for the source node and a circle for the target node, colors are used to code node indices. The processors cooperate in building the total set of requests: Large adjacency lists are handled by groups of PUs. Subsequently, the generated requests are grouped by target nodes using semi-sorting. Then superfluous requests are filtered out, and the remaining request are sent to the processors which host the appropriate bucket structures. Without the balancing, processor P_0 would be over-loaded during the generation, and processor P_1 would receive too many requests.

Proof: First find a perfect hash function $h : V \to \{1, \ldots, c \cdot n\}$ for an appropriate constant c. Using the algorithm of Bast and Hagerup [14] this can be done in time $\mathcal{O}(n/p + \log p)$ (and even faster) whp. Subsequently, we apply a fast, work efficient sorting algorithm for small integer keys such as the one by Rajasekaran and Reif [122] to sort by the hash values in time $\mathcal{O}(n/p + \log n)$ whp. □

4.5.3 Conversion to Distributed Memory Machines

The straightforward way to obtain a DMM algorithm out of a PRAM algorithm is to use an efficient PRAM simulation method: given certain conditions, a step of a CRCW PRAM with p'-processors can be simulated on a p-processors BSP machine in $\mathcal{O}(p'/p)$ time if $p' \geq p^{1+\epsilon}$ for any constant $\epsilon > 0$; see [142]. Hence, if T is defined as in Theorem 8, the distributed algorithm runs in $\mathcal{O}(p^{\epsilon} \cdot T + \frac{n+m+T}{p})$ time.

Alternatively, for a direct coding on a DMM machine, one first has to solve the data allocation problem. Fortunately, the simple PRAM algorithm from Section 4.4 is already almost a distributed memory algorithm. The index array $\mathrm{ind}[\cdot]$ used in the PRAM algorithm can be replaced by a hash function $\mathrm{ind}(\cdot)$. PU P_i maintains the bucket structure $B_i[\cdot]$, and

also stores the adjacency lists of all nodes whose indices hash to the value i. Hence, the relaxation requests can be generated locally. The dart throwing process for assigning requests can be replaced by simply routing a request (w, x) to PU $\mathrm{ind}(w)$. The relaxations for the received requests happens locally again. As for the PRAM version, if the number of processors, is reasonably bounded then the accesses to the memory modules are sufficiently load-balanced in each phase with high probability. Each phase of the algorithm can be performed in a number of *supersteps* each of which consists of local computations, synchronization, and communication.

The advanced parallelization requires additional measures; for example long adjacency lists have to be spread over the local memories of the processors. Important ingredients for the conversion are standard DMM implementations for tree based reduction and broadcasting schemes [88]. They are needed for prefix-sum / minimum computations and distribution of values, e.g., when all edges (v, w_i) of a high-degree node v are to be relaxed, then the value of $\mathrm{tent}(v)$ must be made available to all PUs that store outgoing edges of v. The grouping steps of the algorithm can be implemented by DMM integer-sorting algorithms, e.g., [12]. The choice of the sorting algorithm determines how many supersteps are needed to implement a phase of the PRAM algorithm, and hence how many processors can be reasonably used.

4.6 Better Bounds for Random Graphs

So far we collected the following results for random graphs with random edge weights from class the class $D(n, \bar{d}/n)$ as defined in Section 3.9.3: by Lemma 26 and Corollary 6 we know $d = \mathcal{O}(\bar{d} + \log n)$ and $\mathcal{L} = \mathcal{O}((1/\bar{d}) \cdot \log^2 n)$ with high probability. For the advanced PRAM algorithm of the previous section (Theorem 9) this implies $\mathcal{O}(\log^4 n + (1/\bar{d}) \cdot \log^5 n)$ time using linear work with high probability. In the following we will improve the high-probability bound on $\mathcal{L}$ to $\mathcal{O}((1/\bar{d}) \cdot \log n)$ and sketch two further algorithmic modifications for random graphs.

4.6.1 Maximum Shortest-Path Weight

Theorem 10 *There is a constant $c^* > 1$ such that for random graphs from $D(n, \bar{d}/n)$, $\bar{d} > 3 \cdot c^*$, with independent random edge weights uniformly drawn from $[0, 1]$, the maximum shortest-path weight is bounded from above by $\mathcal{O}((1/\bar{d}) \cdot \log n)$ whp.*

Proof: Due to the results of [32, 57, 80, 120] on the maximum shortest-path weight for dense random graphs with random edge weights we can concentrate on the case $3 \cdot c^* < \bar{d} \leq \Theta(\log n)$ where $c^* > 1$ is the constant from Lemma 27.

The set of nodes reachable from s, denoted by $R(s)$, is either *small*, that is $|R(s)| = \mathcal{O}((1/\bar{d}) \cdot \log n)$ whp, or *giant*, that is $|R(s)| = \Theta(n)$ whp [90]. If $R(s)$ is small, then Theorem 10 follows immediately: any node in $R(s)$ can be reached by a path of at most $|R(s)|$ edges, each of which has weight at most one. Therefore, we may assume that $|R(s)|$ is giant.

Our proof proceeds as follows. First we show that a subgraph G' of G contains a strongly connected component C' of $\Theta(n)$ nodes so that any pair of nodes from C' is con-

nected by a path of total weight $\mathcal{O}((1/\bar{d}) \cdot \log n)$ whp. Then we prove that there is a path from s to C' not exceeding the total weight $\mathcal{O}(1 + (1/\bar{d}) \cdot \log n)$ whp. Finally, we show for all nodes in $R(s)$ that cannot be reached from s directly along a path of $\mathcal{O}(1 + (1/\bar{d}) \cdot \log n)$ edges, that these nodes can still be reached from C' via a path of weight $\mathcal{O}(1 + (1/\bar{d}) \cdot \log n)$ whp. For $\bar{d} = \mathcal{O}(\log n)$ the above path weights sum up to $\mathcal{O}((1/\bar{d}) \cdot \log n)$ whp. Figure 4.3 depicts the such a concatenated path from the source node s to a target node $v \in R(s)$.

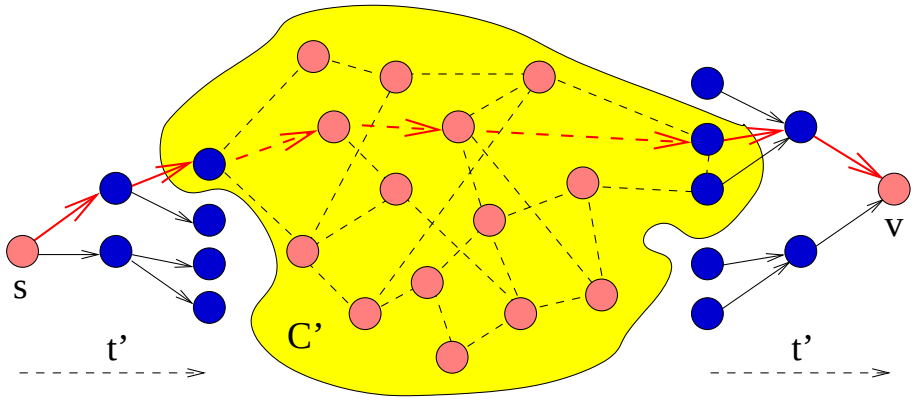


Figure 4.3: Example for a concatenated path of small total weight from the source s to a reachable node v: after at most $t' = \mathcal{O}(\frac{\log n}{\bar{d}})$ edges the strongly connected component C' is reached. There is a light path within C' to an intermediate node from which v can be reached via a path of another t' edges.

For this proof call edges with weight at most $3 \cdot c^*/\bar{d}$ *tiny*. Consider the subgraph $G' = (V, E')$ of G obtained by retaining tiny edges only. G' is a random graph with edge probability $3 \cdot c^*/n$; G' has a giant strong component C' of size $\alpha^* \cdot n$ with probability at least $1 - n^{-\alpha}$ for some arbitrary constant $\alpha \geq 1$ where the positive constant α^* depends on the choice of α (e.g. [8, 90]).

By Lemma 27, the diameter of the subgraph induced by the nodes in C' and its adjacent edges in E' is at most $\mathcal{O}(\log n)$ with probability at least $1 - n^{-c}$. Since all edges in E' have weight at most $3 \cdot c^*/\bar{d}$, any pair of nodes from C' is connected by a path of total weight $\mathcal{O}((1/\bar{d}) \cdot \log n)$ with probability at least $1 - n^{-c}$.

Now we show that there is a path from s to C' not exceeding the total weight $\mathcal{O}(1 + (1/\bar{d}) \cdot \log n)$ whp. We apply the node exploration procedure of [90] and [8, Section 10.5] starting in the source node s and using the edges in E: Initially, s is active and all other nodes are neutral. In each iteration we select an arbitrary active node v, declare it dead and make all neutral nodes w with $(v, w) \in E$ active. The process terminates when there are no active nodes left. Let Y_t be the number of active nodes after t iterations ($Y_0 = 1$). We are interested in the set of nodes reached from s after $t' := \frac{\beta \cdot \ln n}{\bar{d}}$ iterations for some appropriate constant $\beta > 1$. Let BIN $[n, p]$ denote the binomial distribution with parameters n and p, i.e., the number of heads in n independent coin tosses with probability of heads p. Provided that $Y_{t-1} \geq 1$, Y_t is distributed as follows [8, 90]: BIN $\left[n - 1, 1 - (1 - \bar{d}/n)^t\right] + 1 - t$, i.e., Y_t is sharply concentrated around

$$g(t) := (n - 1) \cdot (1 - (1 - \bar{d}/n)^t) + 1 - t.$$

In fact, we can assume $Y_{t'-1} \geq 1$ since otherwise all nodes of $R(s)$ are reached using paths of total weight $\mathcal{O}((1/\bar{d}) \cdot \log n)$. We use the inequality

$$\begin{aligned}(1-x)^y &< 1 - x \cdot y + \frac{(x \cdot y)^2}{2} \quad \text{if} \quad x \cdot y < 1 \\ &\leq 1 - \frac{3}{4} \cdot x \cdot y \quad \text{if} \quad 2 \cdot x \cdot y < 1\end{aligned}$$

in order to derive a lower bound on $g(t')$. Indeed we have $2 \cdot \frac{\bar{d}}{n} \cdot t' = \frac{2 \cdot \beta}{\ln n}$. Hence, for $n \to \infty$ and $\bar{d} > 3 \cdot c^* > 3$, we find

$$g(t') \geq (n-1) \cdot \frac{3}{4} \cdot \bar{d} \cdot t'/n + 1 - t' \geq \frac{\bar{d}}{3} \cdot t' = \frac{\beta \cdot \ln n}{3} \, .$$

Thus, for $Y_{t'-1} \geq 1$, $\mathbf{E}[Y_{t'}] \geq \frac{\beta \cdot \ln n}{3}$. Furthermore, $Y_{t'} \geq (\beta/6) \cdot \ln n$ with probability at least $1 - e^{-\mathbf{E}[Y_{t'}]/8} \geq 1 - n^{-\beta/24}$ by the Chernoff bounds (Lemma 7). That is high probability if the constant β is chosen sufficiently large. So, there are $Y_{t'} \geq (\beta/6) \cdot \log n$ active nodes whp whose outgoing edges have not yet been inspected in the search procedure from s. Let A be the set of these nodes. We are interested in the probability that there is at least one edge from a node in A to a node in C' (if the node sets overlap then we are done as well).

We can assume that G' (and hence C') has been generated without complete knowledge of G in the following way: For every ordered pair of nodes (v, w) we first throw a biased coin with $\mathbf{P}[\texttt{head}] = 3 \cdot c^*/\bar{d}$ and only if it shows a head we find out whether $(v, w) \in E$: in this case we generate a random tiny edge weight, and (v, w) is put to E'. This information suffices to determine C'. Hence, looking at the two node sets A and C' with $v' \in A$ and $w' \in C'$ it is still true that $(v', w') \in E$ with probability $\frac{\bar{d}}{n}$, and these probabilities are independent for all ordered pairs of nodes (v', w'). Therefore, we expect $|C'| \cdot |A| \cdot \frac{\bar{d}}{n}$ edges from A to C'. Since $|C'| \geq \alpha^* \cdot n$ with probability at least $1 - n^{-\alpha}$ and $|A| \geq (\beta/6) \cdot \ln n$ with probability at least $1 - n^{-\beta/24}$, the probability that there exists at least one edge (v, w) between A and C' is at least

$$\begin{aligned}1 - n^{-\alpha} - n^{-\beta/24} - \left(1 - \frac{\bar{d}}{n}\right)^{\alpha^* \cdot n \cdot (\beta/6) \cdot \ln n} &\geq 1 - n^{-\alpha} - n^{-\beta/24} - (1/e)^{\alpha^* \cdot (\beta/6) \cdot \ln n} \\ &\geq 1 - n^{-\alpha} - n^{-\beta/24} - n^{-\alpha^* \cdot \beta/6} \qquad (4.1)\end{aligned}$$

which is high probability for appropriately chosen constants (in particular, we are free to chose α and β large). Consequently, there is a path from s to C' using at most $\mathcal{O}(1 + (1/\bar{d}) \cdot \log n)$ edges whp. Since all edges on this path have weight at most one, the desired bound on the total path weight from s to C' follows immediately.

The same technique is applied to show that any node $v \in R(s)$ that neither belongs to C' nor is reached from s directly along a path of t' edges, can still be reached from C' via a path of $\mathcal{O}(1 + (1/\bar{d}) \cdot \log n)$ edges whp. However, now the search procedure follows the edges in the opposite direction. Note that the different search procedures are not mutually independent. Still, each single search succeeds with the probability bound given in (4.1). Hence, using Boole's inequality all nodes from $R(s)$ can be reached along paths with total weight $\mathcal{O}(1 + (1/\bar{d}) \cdot \log n)$ with probability at least $1 - n^{-c} - n \cdot (n^{-\alpha} - n^{-\beta/24} - n^{-\alpha^* \cdot \beta/6})$. This completes the proof of Theorem 10 for $3 \cdot c^* < \bar{d} \leq \Theta(\log n)$. □

4.6.2 Larger Step Width

In Section 3.9 we have seen that ABI-Dijkstra and Δ-Stepping cause $\mathcal{O}(n)$ node rescans on the average if the bucket width is chosen as $\Delta = 2^{-\lceil \log_2 d \rceil}$ for maximum node degree d. The result was obtained by reducing the analysis to that of **SP-C** (Section 3.6) with a single hierarchy level and buckets of width Δ. In particular, the number of times a node v is scanned on the average had been bounded by the expected number of simple paths of total weight Δ into v (proof of Lemma 18); we argued that there are at most d^l simple paths of l edges into v. Using the fact that long paths with small total weight are unlikely (Lemma 17) it turned out that node v is rescanned at most $\sum_{l=1}^{\infty} d^l \cdot \Delta^{-l}/l!$ times on average.

For random graphs from $D(n, \bar{d}/n)$, we have seen $d = \mathcal{O}(\bar{d} + \log n)$ with high probability (Lemma 26). However, due to the random graph structure, the average-case number of simple paths with l edges into a node v is bounded by $\bar{d}^{\,l}$. Thus, the number of rescans for node v is at most $\sum_{l=1}^{\infty} \bar{d}^{\,l} \cdot \Delta^{-l}/l! = \mathcal{O}(1)$ for $\Delta = 2^{-\lceil \log_2 \bar{d} \rceil}$ on the average. In other words, for sparse random graphs we can choose the bucket width larger without risking increased overhead due to node rescans. This results in less phases for the parallel SSSP algorithm.

4.6.3 Inserting Shortcuts

Our parallelizations of Sections 4.4 and 4.5 need $\mathcal{O}(\log n / \log\log n)$ phases whp before they turn to the next non-empty bucket. However, under certain conditions, a constant number of phases per bucket suffices. We have already seen this behavior for random geometric graphs (Section 3.11.1): we exploited the fact that whenever there is a path $\mathcal{P} = \langle u = v_0, \ldots, v_k = v \rangle$ of total weight $c(\mathcal{P}) < \Delta = 2^{-\lceil \log_2 \bar{d} \rceil}$ in the input graph G, then G also contains the "shortcut" edge (u, v) having weight $c(u, v) \leq c(\mathcal{P})$. Our idea for random graphs is to manually insert these shortcut edges.

Shortcuts for random graphs from $D(n, \bar{d}/n)$ are found by exploring Δ-paths emanating from all nodes in parallel. This is affordable for random edge weights because we know that there are only few simple Δ-paths: for each node we expect $\sum_{l=1}^{\infty} \bar{d}^{\,l} \cdot \Delta^{-l}/l! = \mathcal{O}(1)$ such outgoing paths. Furthermore, none of these paths is longer than $\mathcal{O}(\log n / \log\log n)$ with high probability. The only additional complication is that we have to make sure that only simple paths are explored.

Figure 4.4 outlines a routine which finds shortcuts by applying a variant of the Bellman-Ford algorithm to all nodes in parallel. It solves an all-to-all shortest path problem constrained to Δ-paths. The shortest connections found so far are kept in a hash table of expected size $\mathcal{O}(n)$. The set Q stores *active* connections, i.e., triples (u, v, y) where y is the weight of a shortest known path from u to v and where paths $(u, \ldots, v, w)$ have not yet been considered as possible shortest connections from u to w with weight $y + c(v, w)$. In iteration i of the main loop, the shortest connections using i edges are computed and are then used to update 'found'. Using similar techniques as in Section 4.5, this routine can be implemented to run in $\mathcal{O}\left(\log^2 n / \log\log n\right)$ parallel time using $\mathcal{O}\left(n + m\right)$ work on average: we need $\mathcal{O}(\log n / \log\log n)$ iterations each of which takes time $\mathcal{O}\left(\log n\right)$ and work $\mathcal{O}\left(|Q| + |Q'|\right)$ whp. The overall work bound holds since for each simple Δ-path $(u, \ldots, v)$, $\langle u, v \rangle$ can be a member of Q only once. Hence, $\mathbf{E}[\sum_i |Q|] = \mathcal{O}\left(n\right)$ and $\mathbf{E}[\sum_i |Q'|] = \mathcal{O}\left(n + m\right)$.

```
Function findShortcuts(Δ) : set of weighted edges
  found : HashArray[V × V]                    (* return ∞ for undefined entries *)
  Q := {(u, u, 0) | u ∈ V}                    (* (start, destination, weight) *)
  Q' : MultiSet
  while Q ≠ ∅ do
    Q' := ∅
    foreach (u, v, x) ∈ Q dopar
      foreach light edge (v, w) ∈ E dopar
        Q' := Q' ∪ {(u, w, x + c(v, w))}
    semi-sort Q' by common start and destination node
    Q := {(u, v, x) | x = min {y | (u, v, y) ∈ Q'}}
    Q := {(u, v, x) ∈ Q | x ≤ Δ ∧ x < found[(u, v)]}
    foreach (u, v, x) ∈ Q dopar found[(u, v)] := x
  return {(u, v, x) | found[(u, v)] < ∞}
```

Figure 4.4: CRCW-PRAM routine for finding shortcut edges

Combining the observations of Section 4.6.1 (better bound on $\mathcal{L}$), Section 4.6.2 (larger bucket width), and Section 4.6.3 (less phases per bucket after preprocessing) with the advanced parallelization of Section 4.5 we find:

Corollary 10 *SSSP on random graphs with independent random edge weights uniformly distributed in* $[0, 1]$ *can be solved on a CRCW PRAM in* $\mathcal{O}(\log^2 n)$ *time using linear work on the average.*

4.7 Parallel Individual Step-Widths

The parallelizations presented so far perform poorly if the maximum node degree d happens to be huge and the expected maximum shortest-path weight is at least constant: in that case $\Omega(d)$ parallel time is needed on the average. This is not surprising since the underlying sequential algorithms (ABI-Dijkstra oder Δ-Stepping with $\Delta = 2^{-\lceil \log_2 d \rceil}$ suffer from the same shortcoming. Therefore, let us look at a straightforward parallelization of the linear average-case time **SP-C** algorithm from Section 3.4: similar to the previous parallelizations, it could perform the operations for all nodes of the current bucket in parallel. Unfortunately, at least one phase is needed for each non-empty current bucket: for example, consider the graph with source node 0, and the edges $(0, 1), (0, 2), \ldots, (0, n-1)$. The expected number of non-empty intervals $[i/n, (i+1)/n)$ for the distances of nodes $1, \ldots, n-1$ is asymptotic to $n \cdot (1 - e^{-1}) = \Omega(n)$ [129]. Processing the source node 0 of the graph above with **SP-C** creates a second bucket level of $\Theta(n)$ new buckets. An expected constant fraction of them is non-empty. Hence, $\Omega(n)$ phases are still needed on the average.

The example above highlights a weakness of a parallel **SP-C** algorithm: once a node with large degree forced a reduction of the step-width in order to limit the risk of node re-insertions, this step-width is kept for a certain distance range – even if no high-degree nodes

remain in the new buckets of this range. In the following we will show how to remove this drawback.

4.7.1 The Algorithm.

In this section we introduce the new parallel SSSP algorithm called **PIS-SP** (for **P**arallel **I**ndividual **S**tep-Widths). The algorithm applies *different* step-widths on disjoint node sets *at the same time* using a new split-free bucket data structure.

PIS-SP applies a queue data structure Q that consists of $\mathcal{O}(n)$ buckets in total, organized into $\mathcal{O}(\log n)$ arrays $B_i[\cdot]$: each array covers a total tentative distance range of width two. The buckets of each array are used in a cyclical fashion in order to subsequently store nodes with larger and larger tentative distances. The array $B_i[\cdot]$, $1 \leq i \leq \lceil \log_2 n \rceil =: h$ consists of 2^{i+2} buckets of width $\Delta_i := 2^{-i-1}$ each. $B_i[\cdot]$ exclusively stores nodes with in-degree in $\{2^{i-1}, \ldots, 2^i - 1\}$. Hence, for example a node v with in-degree $d = 6$ and tentative distance $\mathrm{tent}(v) = 5.3$ is stored in the array $B_i[\cdot]$, $i = \lfloor \log_2 d \rfloor + 1 = 3$, and more concretely in bucket $B_3[\lfloor (5.3 \bmod 2)/\Delta_3 \rfloor] = B_3[\lfloor 1.3/0.0625 \rfloor] = B_3[20]$. Note that there is no array for vertices with in-degree zero, as they cannot be reached anyway.

PIS-SP works as follows: initially, all nodes v reachable from the source node s via an edge (s, v) are put in parallel into their respective bucket arrays using the edge weights $c(s, v)$ as tentative distances. After that, **PIS-SP** operates in phases: At the beginning of a phase it first determines the globally smallest tentative distance $M := \min_{v \in Q} \mathrm{tent}(v)$ among all currently queued nodes. This step is quite intricate since the bucket structure only implements an approximate priority queue. Hence, M is not readily available; not too many operations must be used to find it. In Section 4.7.3 we will show how M can be obtained efficiently. Knowing M, **PIS-SP** scans all nodes from $B_i\,[\lfloor (M \bmod 2)/\Delta_i \rfloor]$ in parallel for each i, $1 \leq i \leq h$. This may insert new nodes into Q or reinsert previously scanned nodes with improved tentative distances. However, each phase settles at least one node. The algorithm stops if there are no queued nodes left after a phase. Hence, it requires at most n phases.

Each bucket is implemented by a separate array with dynamic space adjustment; additional arrays are used in order to keep track of tentative distances and to remember in which bucket and which array cell a queued node is currently stored. Removing nodes from buckets in parallel is clearly congestion-free. However, insertions must be organized more carefully, see Figure 4.5: Having removed all nodes from the buckets $B_i[\lfloor (M \bmod 2)/2^{-i-1} \rfloor]$ for a phase (resulting in a node set R), the set Req of all edges emanating from nodes in R is built. An immediate parallel relaxation of the set Req might cause conflicts, therefore Req is first grouped by target nodes (using semi-sorting with small hashed values as seen in Section 4.5), and then the strictest relaxation request for each target node (group) is selected. The selected requests are grouped once more by target buckets and finally each group is appended in parallel after the last used position in the array for the target bucket. If there are not sufficiently many contiguous free positions left (the free positions may be scattered due to nodes that have been moved to other buckets during edge relaxations), then the whole content of this bucket is compacted and then copied to a new array of twice the size.

Each phase can be performed in $\mathcal{O}(\log n)$ average-case time; the respective number of

PIS-SP

foreach $(s,v) \in G$ **dopar**
  $i := \lfloor \log_2 \text{indegree}(v) \rfloor + 1$
  Insert v into $B_i[\lfloor c(s,v)/2^{-i-1} \rfloor]$
while $Q \neq \emptyset$ **do**
  Determine $M := \min_{v \in Q} \text{tent}(v)$ /* Phase starts */
  foreach $1 \leq i \leq \lceil \log_2 n \rceil$ **dopar**
    Empty bucket $B_i[\lfloor (M \bmod 2)/2^{-i-1} \rfloor]$
  $\Rightarrow$ node set R
  Build set Req for all edges out of R.
  Group Req by target nodes.
  foreach target node v of Req **dopar**
    Select best relaxation of an edge into v.
  Group selected edges by target buckets.
  Perform relaxations (& enlarge buckets if needed). /* Phase ends */

Figure 4.5: Pseudo-code for **PIS-SP**.

operations excluding array size adjustments is linear in $|R| + |Req|$. The total work needed to adjust the array sizes can be amortized over the total number of edge (re-) relaxations.

4.7.2 Performance for Random Edge Weights.

In the following we consider the expected numbers of node rescans and phases for **PIS-SP** on graphs with random edge weights:

Definition 5 *Let $P := \langle v_j, \ldots, v_0 \rangle$ be a path into an arbitrary node v_0. P is called* **degree-weight balanced** *(***dwb***) if $c(v_{i+1}, v_i) \leq 2^{-\lfloor \log_2 \text{indegree}(v_i) \rfloor - 2}$, for all i, $0 \leq i < j$.*

Lemma 37 *For each node $v \in V$, the number of rescans during the execution of* **PIS-SP** *is bounded by the number of simple* **dwb** *paths into v.*

Proof: Let $\text{tent}^j(v)$ and Q_j denote the value of $\text{tent}(v)$ and the set of nodes in Q at the beginning of phase j, respectively; define $M^j := \min_{v \in Q_j} \text{tent}^j(v)$. Clearly, for nonnegative edge weights, $M^{j+1} \geq M^j$ and $\text{tent}^{j+1}(v) \leq \text{tent}^j(v)$ for all $j \geq 1$ and $v \in V$. Let $T(v,j)$ denote the total number of rescans for node v during the phases 1 to j; let $R(v,k,t)$ denote be the k-th rescan of he node v happening in phase t.

The proof of the lemma is by induction; we show that all rescans of the phases 1 to t can be injectively mapped onto simple paths. More specifically, a rescan of a node $v \in V$ in phase t is mapped onto a simple **dwb** path $P = \langle v_l, \ldots, v_1, v_0 = v \rangle$ of $l \leq t-1$ edges into v where v_j is scanned in phase $t-j$, and $\text{tent}^{t-j-1}(v_{j+1}) + c(v_{j+1}, v_j) = \text{tent}^{t-j}(v_j)$. It follows immediately that rescans of different nodes are mapped onto different simple paths. In the remainder we are concerned with different rescans of the same node.

Each node is scanned at most once per phase; no node is rescanned in the first phase. If the node v with degree d_v is scanned for the first time in phase j then it is scanned from

bucket $B_{i_v}\left[\lfloor (M^j \bmod 2)/\Delta_{i_v} \rfloor\right]$ where $i_v = \lfloor \log_2 d_v \rfloor + 1$, and

$$\begin{aligned} M^j &\leq \mathrm{dist}(v) \leq \mathrm{tent}^j(v) < \lfloor M^j/\Delta_{i_v} \rfloor \cdot \Delta_{i_v} + \Delta_{i_v} \\ &\leq M^j + \Delta_{i_v} = M^j + 2^{-\lfloor \log_2 d_v \rfloor - 2} . \end{aligned}$$

Now we consider $R(v,k,t)$ where $t \geq k+1 \geq 2$ (i.e., $\mathrm{tent}^{t-1}(v) > \mathrm{tent}^t(v)$ and $T(v,t-1) = k-1$). As v was scanned for the first time in some phase $t' < t$, all nodes with final distances less than $M^{t'}$ were already settled at that time; their respective edges into v_0 (if any) will neither be relaxed in phase t' nor later. Therefore, $R(v,k,t)$ requires that some node $v' \neq v$ having $(v',v) \in E$ is scanned in phase $t-1$ (i.e., $\mathrm{dist}(v') \geq M^{t-1} \geq M^{t'}$) where $\mathrm{tent}^t(v) =$

$$\mathrm{tent}^{t-1}(v') + c(v',v) < \mathrm{tent}^{t-1}(v) < M^{t'} + 2^{-\lfloor \log_2 d_v \rfloor - 2}.$$

These conditions imply $c(v',v) \leq 2^{-\lfloor \log_2 d_v \rfloor - 2}$.

If $T(v',t-1) = 0$ then $R(v,k,t)$ can be uniquely mapped onto the edge (v',v), which is relaxed for the first time in phase $t-1$. Obviously, $\langle v',v \rangle$ is a simple **dwb** path into v of at most $t-1$ edges because $c(v',v) \leq 2^{-\lfloor \log_2 d_v \rfloor - 2}$.

Otherwise, i.e., if $T(v',t-1) = k' > 0$ then $R(v',k',t-1)$ was inductively mapped onto some simple **dwb** path $P' = \langle v'_{l'}, \ldots, v'_1, v'_0 = v' \rangle$ of $l' \leq t-2$ edges where v'_j is scanned in phase $t-1-j$, and $\mathrm{tent}^{t-2-j}(v'_{j+1}) + c(v'_{j+1}, v'_j) = \mathrm{tent}^{t-1-j}(v'_j)$. Hence, $\mathrm{tent}^{t-1-j}(v'_j) \leq \mathrm{tent}^{t-1}(v'_0)$, whereas $\mathrm{tent}^{t-1-j}(v) \geq \mathrm{tent}^{t-1}(v) > \mathrm{tent}^{t-1}(v'_0)$ for all $0 \leq j < l'$. Consequently, v is not part of P'. We map $R(v,k,t)$ onto the path $P = \langle P', v \rangle$ that is built from the concatenation of P' and (v',v). As required, P is a simple **dwb** path of at most $t-1$ edges where the nodes are scanned in proper order and the equations for the tentative distances hold. Furthermore, P is different from any other path $\tilde{P} = \langle \tilde{P}', v \rangle$ constructed for some $R(v,\tilde{k},\tilde{t})$ where $1 \leq \tilde{k} < \tilde{t} \leq k < t$: when constructing $\tilde{P}$ we either considered another edge $(v'',v) \neq (v',v)$, but then the subpaths P' and $\tilde{P}'$ end in different nodes; or we considered different rescans $R(v',k',t-1)$ and $R(v',k'',\tilde{t}-1)$ of the same node v', but then these rescans were mapped on different paths P' and $\tilde{P}'$ by induction. □

Lemma 38 *For random edge weights uniformly drawn from* $[0,1]$, **PIS-SP** *rescans each node at most once on the average.*

Proof: Let $X_l^{v_0}$ denote the number of simple **dwb** paths of $l \geq 1$ edges into an arbitrary node v_0 of a graph G. We first show $\mathbf{E}[X_l^{v_0}] \leq 2^{-l}$. The argument is by induction: let d_i denote the in-degree of node v_i. Excluding self-loops there is a set S_1 of at most d_0 edges into node v_0, hence $\mathbf{E}[X_1^{v_0}] \leq d_0 \cdot 2^{-\lfloor \log_2 d_0 \rfloor - 2} \leq d_0 \cdot 2^{-\log_2 d_0 - 1} = 1/2$.

Now consider the set $\mathcal{P}_l = \{p_1, \ldots p_k\}$ of all simple paths with l edges into node v_0 in G. By assumption $\mathbf{E}[X_l^{v_0}] = \sum_{p_i \in \mathcal{P}_l} \mathbf{P}[\, p_i \text{ is } \mathbf{dwb} \,] \leq 2^{-l}$.

For each $p_i = \langle v_i, \ldots, v_0 \rangle \in \mathcal{P}_l$ there are at most d_i edges into v_i so that the concatenation with p_i results in a simple path of $l+1$ edges into v_0. In particular, for each such path $\langle v_j, v_i, \ldots, v_0 \rangle$, the weight of the newly attached edge (v_j, v_i) is independent of the weights on $\langle v_i, \ldots, v_0 \rangle$. Therefore, $\mathbf{P}[\, \langle v_j, v_i, \ldots, v_0 \rangle \text{ is } \mathbf{dwb} \,] \leq 2^{-\lfloor \log_2 d_i \rfloor - 2}$.

$\mathbf{P}[\ \langle v_i, \ldots, v_0\rangle$ is **dwb** $]$. By linearity of expectation, $\mathbf{E}[X_{l+1}^{v_0}]$ is bounded from above by

$$\begin{aligned} & \sum_{p_i \in \mathcal{P}_l} d_i \cdot 2^{-\lfloor \log_2 d_i \rfloor - 2} \cdot \mathbf{P}[\ p_i = \langle v_i, \ldots, v_0\rangle \text{ is } \mathbf{dwb}\ ] \\ \leq & \sum_{p_i \in \mathcal{P}_l} 1/2 \cdot \mathbf{P}[\ p_i \text{ is } \mathbf{dwb}\ ] \leq 1/2 \cdot \mathbf{E}[X_l^{v_0}] \leq 2^{-l-1}. \end{aligned}$$

By Lemma 37, v_0 is rescanned at most $\sum_{j\geq 1} X_j^{v_0}$ times, therefore the average-case number of rescans of v_0 is at most $\sum_{j\geq 1} \mathbf{E}[X_j^{v_0}] \leq \sum_{j\geq 1} 2^{-j} = 1$. □

Lemma 39 *For graphs with random edge weights uniformly drawn from* $[0,1]$, **PIS-SP** *needs* $r = \mathcal{O}(\log n \cdot \min_i\{2^i \cdot \mathbf{E}[\mathcal{L}] + |V_i|\})$ *phases on the average where* $\mathcal{L}$ *denotes the maximum shortest-path weight and* $|V_i|$ *is the number of graph vertices with in-degree at least* 2^i.

Proof: Let x, $0 \leq x \leq \lceil \log_2 n \rceil - 1$, be some arbitrary integer. For the analysis, let us distinguish two kinds of phases for the algorithm: B_x*-phases* scan only nodes with in-degree less than 2^x whereas S_x*-phases* scan at least one node with in-degree 2^x or larger. Let s_x denote the number of S_x-phases. By Lemma 38, each node is rescanned $\mathcal{O}(1)$ times on average. Hence, $\mathbf{E}[s_x] = \mathcal{O}(|V_x|)$.

In the following we are interested in B_x*-chunks*: for some constant $c \geq 1$, a B_x-chunk consists of $t = (c+5)\cdot \log_2 n + 1$ consecutive B_x-phases without any intermediate S_x-phases. Observe that less than $(s_x + 1)\cdot t$ of all B_x-phases do not belong to any B_x-chunk. Let M^j be the smallest tentative distance among queued nodes at the beginning of the j-th phase of a B_x-chunk. We will show

$$\lfloor M^t/\Delta_x \rfloor > \lfloor M^1/\Delta_x \rfloor \quad \text{with high probability.} \tag{4.2}$$

During the j-th phase of the B_x-chunk, **PIS-SP** scans all nodes from the current buckets $B_i\left[\left\lfloor (M^j \bmod 2)/\Delta_i \right\rfloor\right]$, $1 \leq i \leq x$. Let $\mathcal{N}_j$ denote the set of these nodes. Observe that (4.2) holds if $\left\lfloor (M^1 \bmod 2)/\Delta_i \right\rfloor \neq \left\lfloor (M^t \bmod 2)/\Delta_i \right\rfloor$ for at least one i, $1 \leq i \leq x$, i.e., **PIS-SP** has advanced at least one current bucket from $B_1[\cdot], \ldots, B_x[\cdot]$ according to the cyclical ordering. So, let us assume that the current buckets from $B_1[\cdot], \ldots, B_x[\cdot]$ in phase t are those of phase 1. But then there must be at least one node $v_t \in \mathcal{N}_t$ with $\mathrm{tent}(v_t) < M^1 + \Delta_x$. In particular, there must be a simple **dwb** path $P = \langle v_1, \ldots v_t\rangle$ of total weight less than $\Delta_x = 2^{-x-1}$ where $v_j \in \mathcal{N}_j$, and all v_j, $1 \leq j \leq t$, have in-degree less than 2^x. Into any node $v \in \mathcal{N}_t$ there are less than $2^{x\cdot(t-1)}$ such paths. As shown in Lemma 17, the sum of l independent random edge weights (uniformly distributed in $[0,1]$) is at most $\Delta \leq 1$ with probability at most $\Delta^l/l!$. Hence, the probability that such a path exists into any node $v \in \mathcal{N}_t$ is bounded by $n \cdot 2^{x\cdot(t-1)} \cdot \Delta_x^{t-1}/(t-1)! \leq n \cdot 2^{-t+1} \leq n^{-c-4}$. That proves (4.2). Therefore, after $\lceil \mathcal{L}/\Delta_x \rceil$ B_x-chunks the current buckets have been advanced so much that no node remains in the bucket structure with probability at least $1 - \lceil \mathcal{L}/\Delta_x \rceil \cdot n^{-c-4} \geq 1 - n^{-c-1}$. This accounts for at most another $\mathcal{O}(2^x \cdot \mathcal{L} \cdot \log n)$ B_x-phases with probability at least $1 - n^{-c-1}$; as **PIS-SP** requires at most n phases in the worst case, it needs at most $\mathcal{O}(2^x \cdot \mathbf{E}[\mathcal{L}] \cdot \log n + n \cdot n^{-c-1}) = \mathcal{O}(2^x \cdot \mathbf{E}[\mathcal{L}] \cdot \log n)$ additional B_x-phases on the average.

Altogether the algorithm runs in $\mathcal{O}((2^x \cdot \mathbf{E}[\mathcal{L}] + |V_x|) \cdot \log n)$ phases on the average. Since this analysis holds for any integer x, the average-case bound for all phases can be restated as $\mathcal{O}(\min_i\{2^i \cdot \mathbf{E}[\mathcal{L}] + |V_i|\} \cdot \log n)$. Note once more that the algorithm itself does not have to find an optimal compromise between B_x-phases and S_x-phases; they are just a theoretical concept of the analysis. □

4.7.3 Fast Node Selection.

In the following we show how the bucket data structure can be modified in order to determine M fast and efficiently: each array $B_i[\cdot]$ is augmented by a pointer to the first nonempty bucket B_i^* of range $[M_i^*, M_i^* + \Delta_i]$ according to the cyclic ordering.

Lemma 40 *For random edge weights uniformly drawn from* $[0, 1]$*, the total number of operations needed to maintain the pointers to* B_i^* *during the execution of* **PIS-SP** *is bounded by* $\mathcal{O}(n + m)$ *on the average. For each phase, maintaining the pointers takes at most* $\mathcal{O}(\log n)$ *time.*

Proof: B_i^* may have changed after a phase due to (i) an insertion of a node v into $B_i[\cdot]$, (ii) a decrease of $\mathrm{tent}(v)$ where $v \in B_i[\cdot]$, (iii) a scan of v from $B_i[\cdot]$. Cases (i) and (ii) are straightforward: if v moves to a bucket dedicated to smaller tentative distances then B_i^* is set to this new bucket otherwise it stays unchanged. The grouping and selection procedure taking place in each phase prior to parallel node insertions can be adapted such that for each array the smallest inserted tentative distance is determined as well. Only this value needs to be checked for a possible update of B_i^*.

Maintaining B_i^* for case (iii) is only non-trivial if the bucket of the scanned node v remains empty after a phase: all buckets of $B_i[\cdot]$ must be checked in order to find the new first nonempty bucket according to the cyclic ordering. On a PRAM this can be accomplished using standard parallel prefix-sum and minimum computations [88] in $\mathcal{O}(\log n)$ time and $\mathcal{O}(2^{i+2})$ work. These operations can be attributed to the degree of the scanned node v, which is at least 2^{i-1}. Since all degrees add up to $\mathcal{O}(m)$ and since each node is scanned $\mathcal{O}(1)$ times on average (Lemma 38), the total amount of operations needed for all occurrences of case (iii) is bounded by $\mathcal{O}(n + m)$ on the average, as well. □

Finding the Minimum. We show how the values M_i^* of the first nonempty buckets $B_i^* = [M_i^*, M_i^* + \Delta_i)$ introduced above are used to identify the value M of the globally smallest tentative distance among all queued nodes. At the beginning of a phase **PIS-SP** computes the *suffix-minima* of the values M_i^*, i.e., for each i, it builds $M(i) := \min\{M_i^*, M_{i+1}^*, \ldots, M_h^*\}$. We set $M_i^* := +\infty$ if $B_i[\cdot]$ is empty. Note that $M(1) \leq M < M(1) + \Delta_1$. Next, the procedure refines safe estimates $\widehat{M}_i \leq M \leq \widehat{T}_i$ for the globally smallest tentative distance among all queued nodes: initially, we set $\widehat{M}_1 := M(1)$ and $\widehat{T}_1 := M(1) + \Delta_1$. The minimum detection takes place in at most h stages:

For the first stage, **PIS-SP** checks the bucket $B_1\left[\left\lfloor(\widehat{M}_1 \bmod 2)/\Delta_1\right\rfloor\right]$: either, this bucket is empty, or it constitutes the bucket B_1^* keeping the node with smallest tentative distance for the array $B_1[\cdot]$. All nodes from $B_1^* = B_1\left[\left\lfloor(\widehat{M}_1 \bmod 2)/\Delta_1\right\rfloor\right]$ can be scanned

during the phase according to the scan criterion. Let T_1 be the smallest tentative distance among all nodes of B_1^* ($T_1 := \infty$ if B_1^* is empty). The value of T_1 is determined with a parallel minimum algorithm causing work linear in the number of elements stored in bucket B_1^*. If $\min\{T_1, \widehat{T}_1\} \leq M(2)$ then either the whole bucket structure is empty or a node v' with globally smallest tentative distance is stored in B_1^*, and therefore $M = \min\{T_1, \widehat{T}_1\}$. The procedure stops. Otherwise, if $\min\{T_1, \widehat{T}_1\} > M(2)$, the queued node with smallest tentative distance may still be found in one of the subsequent arrays $B_2[\cdot], \ldots, B_h[\cdot]$. Therefore, the procedure sets $\widehat{M}_2 := M(2)$, $\widehat{T}_2 := \min\{T_1, \widehat{T}_1, M(2) + \Delta_2\}$, and it continues with stage 2 on the bucket array $B_2[\cdot]$.

In general, if the detection procedure reaches stage $i \geq 2$ then we may assume by induction that it has identified estimates $\widehat{M}_i \leq M < \widehat{T}_i \leq \widehat{M}_i + \Delta_i$. The procedure computes the smallest distance value T_i among the nodes in $B_i\left[\left\lfloor (\widehat{M}_i \bmod 2)/\Delta_i \right\rfloor\right]$. Observe that this bucket is either empty (then $T_i := +\infty$) or its nodes will be scanned by **PIS-SP**. Therefore, the work to identify T_i can be amortized over the node scans. Having computed T_i we may find $\min\{T_i, \widehat{T}_i\} \leq M(i+1)$, then $\min\{T_i, \widehat{T}_i\}$ definitely equals M; the procedure stops. Otherwise, that is if $\min\{T_i, \widehat{T}_i\} > M(i+1)$, we only know for sure that $M(i+1) \leq M_i \leq \min\{T_i, \widehat{T}_i, M(i+1) + \Delta_{i+1}\}$. The procedure will continue in $B_{i+1}[\cdot]$ with the new estimates: $\widehat{M}_{i+1} := M(i+1)$ and $\widehat{T}_{i+1} := \min\{T_i, \widehat{T}_i, M(i+1) + \Delta_{i+1}\}$. After at most $h = \mathcal{O}(\log n)$ stages, M is eventually determined. Figure 4.6 provides an example.

Scanning the nodes from the examined buckets can be overlapped with the global minimum detection. However, asymptotically this does not improve the running time. The time required for a phase depends on the applied procedure for the $\mathcal{O}(\log n)$ local minimum computations: with the standard approach based on balanced trees [88] a phase of **PIS-SP** needs $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(k + \log^2 n)$ work where k denotes the number of nodes scanned in that phase. Using the constant-time linear-work randomized minimum computation from [58] reduces the time by a logarithmic factor. Combining the results of this section we find:

Theorem 11 *For graphs with random edge weights uniformly drawn from* $[0, 1]$, **PIS-SP** *requires* $T = \mathcal{O}(\log^2 n \cdot \min_i\{2^i \cdot \mathbf{E}[\mathcal{L}] + |V_i|\})$ *average-case time using* $\mathcal{O}(n + m + T)$ *operations on a CRCW PRAM where* $\mathcal{L}$ *denotes the maximum shortest-path weight and* $|V_i|$ *is the number of graph vertices with in-degree at least* 2^i.

4.7.4 Performance Gain on Power Law Graphs.

Many sparse massive graphs such as the WWW graph and telephone call graphs share universal characteristics which can be described by the so-called "power law" [5, 13, 95]: the number of nodes, y, of a given in-degree x is proportional to $x^{-\beta}$ for some constant $\beta > 0$. For most massive graphs, $2 < \beta \leq 4$: independently, Kumar et al. [95] and Barabasi et al. [13] reported $\beta \approx 2.1$ for the in-degrees of the WWW graph, and the same value was estimated for telephone call graphs [5]. Further studies mentioned in [13] on social networks resulted in $\beta \approx 2.3$, and for graphs of the electric power grid in the US, $\beta \approx 4$.

The observed diameters are usually very small; on random graph classes – with $\mathcal{O}(n)$ edges – that are widely considered to be appropriate models of real massive graphs like

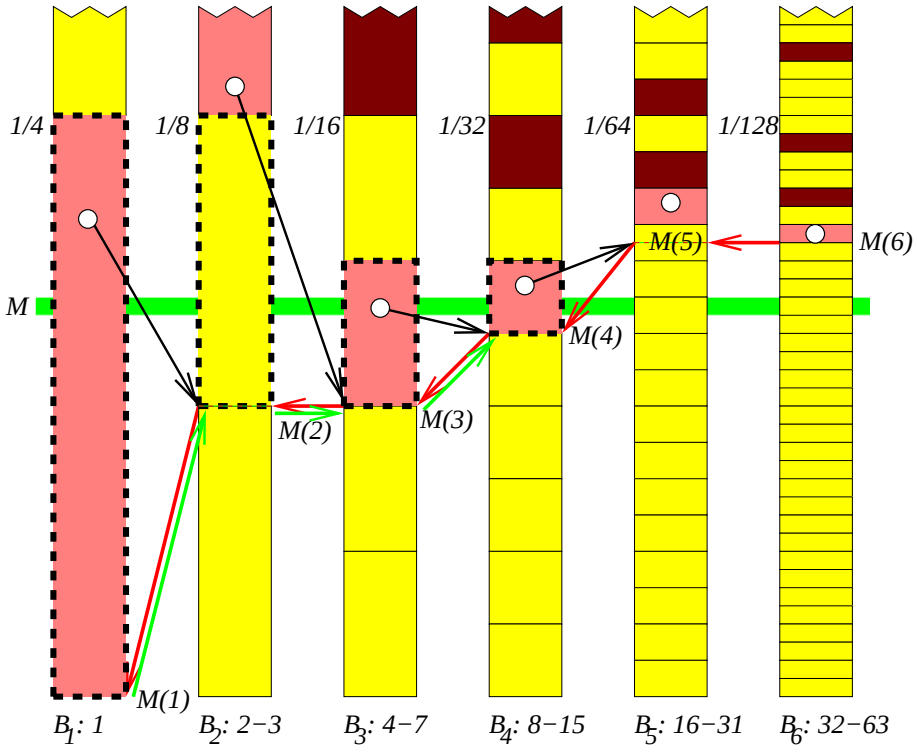


Figure 4.6: Determination of the smallest tentative distance M in the bucket structure. The circles denote the smallest distance T_i in each array $B_i[\cdot]$. In this example, only the nodes in the first nonempty buckets of $B_1[\cdot]$, $B_3[\cdot]$, and $B_4[\cdot]$ are examined to obtain M: after testing the nodes from the first nonempty bucket of $B_1[\cdot]$ we find $T_1 > M(2)$; hence, the bucket $B_2\,[\lfloor (M(2) \bmod 2)/\Delta_2 \rfloor]$ is tested, but this bucket is empty. As $\widehat{T}_2\ (= T_1) > M(3)$, the search continues in $B_3\,[\lfloor (M(3) \bmod 2)/\Delta_3 \rfloor]$ where we find a new smallest element (T_3), but it is still larger than $M(4)$. We continue in $B_4\,[\lfloor (M(4) \bmod 2)/\Delta_4 \rfloor]$ where no smaller element is found. Finally, $M(5)$ is larger than the smallest distance seen so far (T_3). Thus, all other elements in $B_i[\cdot]$, $i \geq 5$ will be larger as well, therefore $M = T_3$.

the WWW it turns out that $\mathcal{L} = \mathcal{O}(\log n)$ whp [98]. For such graphs the number of nodes having in-degree at least d^* is approximately given by $\Theta(n \cdot \sum_{x \geq d^*} x^{-\beta})$, which for constant $\beta \geq 2$ and arbitrary $d^* \geq 1$ can be bounded by $\Theta(n \cdot \int_{d^*}^{\infty} x^{-\beta}\, dx) = \Theta(n \cdot {d^*}^{-\beta+1})$. Taking $d^* = n^{1/\beta}$ one expects $\mathcal{O}(n \cdot n^{1/\beta \cdot (-\beta+1)}) = \mathcal{O}(n^{1/\beta})$ nodes with in-degree at least d^*. On the other hand, we expect at least one node with in-degree $d = \Omega(n^{\frac{1}{\beta-1}})$ by solving $n \cdot d^{-\beta+1} = 1$.

Therefore, assuming independent random edge weights uniformly distributed in $[0, 1]$, the average-case time for SSSP on WWW-like graphs can be estimated as follows: The parallel linear average-case work SSSP algorithm from Section 4.5 requires $\Omega(n^{\frac{1}{\beta-1}})$ time since it has to traverse $\Omega(d) = \Omega(n^{\frac{1}{\beta-1}})$ buckets sequentially. **PIS-SP** needs only $\mathcal{O}(\log^2 n \cdot (n^{1/\beta} \cdot \log n + n^{1/\beta})) = \mathcal{O}(n^{1/\beta} \cdot \log^3 n)$ average-case time using linear $\mathcal{O}(n + m)$ work. Hence, while retaining the linear work bound, the average-case running time could be improved by a factor of $\Omega(n^{1/(\beta^2-\beta)} / \log^3 n)$.

4.8 Conclusions

Our parallel SSSP algorithms - different as they are - share a common feature: they operate in phases, and any node v that needs at least k edges to be reached from the source node s in the graph will not be found before phase k. Hence, the diameter of the input graph is a lower bound on the parallel time of our approaches. Any work-efficient algorithm with sublinear running time that is *independent* of the diameter would be of great interest.

For the future it would also be desirable to solve the SSSP on WWW-like graphs in polylogarithmic time using at most $\mathcal{O}(n \cdot \log n + m)$ work.

In order to obtain fast SSSP algorithms with linear average-case work, one might also try to parallelize Goldberg's new sequential label-setting algorithm [67]. A parallel version of this algorithm can be seen as an improved implementation of the IN-approach [33] (see also Lemma 5) in the sense that exact priority queues are not needed: the smallest used bucket-width of the radix heap equals the globally smallest edge weight. Hence, for random edge weights there are still at most $\mathcal{O}(\log n)$ buckets in the whole radix heap data structure whp; a linear number of operations can be achieved on the average. Both approaches maintain (explicitly or implicitly) a set F that keeps nodes v having $\mathrm{tent}(v) - \min_w c(w,v) \leq M$ where M denotes the smallest tentative distance currently stored in the heap. All nodes in F can be scanned in parallel (in a phase). The algorithms differ in the way F is refilled after a phase: the IN-approach considers all queued nodes whereas Goldberg's algorithm only checks a subset: namely those nodes whose tentative distances were just improved or nodes in the first non-empty level of the radix heap. Thus, the total number of phases for Goldberg's algorithm is definitely not smaller than that for the IN-approach. In the following we prove comparably poor average-case performance of the IN-approach on a simple graph class:

Lemma 41 *There are graphs with n nodes, $\mathcal{O}(n)$ edges, independent random edge weights uniformly distributed in $[0,1]$, and maximum shortest-path weight $\mathcal{L} \leq 1$ such that the IN-approach requires $\Omega(\sqrt{n})$ phases on the average.*

Proof: Let the node with index zero be the source. For the remaining nodes with index i, $1 \leq i \leq n-1$, we include the edges $(0,i)$ and (i,i), thus resulting in a graph with $2 \cdot n - 2$ edges. Due to the simple star graph structure, after relaxing the outgoing edges of s, all nodes are already assigned their final distances $\mathrm{dist}(i) \leq 1$. However, some of the otherwise meaningless self-loop edges will have small edge weights, thus preventing the IN-approach to remove these nodes fast from the heap. Let K be the set of nodes having a self-loop edge of weight less than $r := 1/\sqrt{n-1}$, then $\mathbf{E}[|K|] = \sqrt{n-1}$. Furthermore, let us subdivide the range of final node distance into pieces of width $r := 1/\sqrt{n-1}$ each, such that $R[i]$ denotes the range $[i \cdot r, (i+1) \cdot r)$. Finally, let k let denote the number of distinct pieces of $R[\cdot]$ occupied by nodes of the set K; $\mathbf{E}[k] = \Theta(\sqrt{n})$. Consider the beginning of a phase for the IN-approach where some node u defines the minimum distance M among all nodes in the priority queue for this phase, i.e. $M = \mathrm{dist}(u)$. Let us assume $\mathrm{dist}(u) \in R[j]$, then the current phase is unable to remove any queued node $v \in K$ where $\mathrm{dist}(v) \in R[i]$ and $i \geq j+2$:

$$\mathrm{tent}(v) - \min_w c(w,v) \geq (j+2) \cdot r - r = (j+1) \cdot r > M.$$

Therefore, after x phases of the IN-approach, the remaining number of queued nodes belonging to the set K is at least $k - 2 \cdot x$. Using $\mathbf{E}[k] = \Theta(\sqrt{n})$ we see that $\Omega(\sqrt{n})$ phases are needed on the average to remove all nodes of from the queue. □

The graph class used to show Lemma 41 features one node with degree $\Theta(n)$ whereas all other nodes have constant degree, and $\mathcal{L} \leq 1$. Hence, this graph class allows solving SSSP with the **PIS-SP** algorithm in polylogarithmic time and linear work on the average. Alternative constructions with constant maximum node degree[3] result in similar lower bounds for the IN-approach; $\Omega(\sqrt{n}/\log n)$ phases are needed on the average, whereas even the simple parallelizations of ABI-Dijkstra run in polylogarithmic average-case time and linear work.

Reviewing the proof of Lemma 41, one might argue that both the IN-approach and Goldberg's algorithm could be properly adapted in order not to consider certain irrelevant incoming edges like self-loops or edges out of nodes that cannot be reached from s themselves. However, even such an adapted criterion can still will be weak: we may augment the graph by an extra path path $\langle s, \ldots, u \rangle$ of $\Theta(\log n)$ edges, thus increasing $\mathcal{L}$ to $\mathcal{O}(\log n)$; furthermore, we replace each self-loop (i, i) by an edge (u, i). This modification will most probably not influence the distances of the previous nodes i in the star since $\mathrm{dist}(i) \leq 1$ but $\mathrm{dist}(u) = \Omega(\log n)$ with high probability. However, the adapted criterion will consider all incoming edges. Hence, even the adapted algorithm requires $\Omega(\sqrt{n})$ phases on the average.

[3]For example, one can take a complete binary tree where the root is the source node and the edges are directed towards the leafs. Additionally, each leaf node is equipped with a self-loop edge.

Bibliography

[1] P. Adamson and E. Tick. Greedy partitioned algorithms for the shortest path problem. *International Journal of Parallel Programming*, 20(4):271–298, 1991.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

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

[4] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.

[5] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proc. 32nd Annual ACM Symposium on Theory of Computing*, pages 171–180. ACM, 2000.

[6] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989.

[7] P. Alefragis, P. Sanders, T. Takkula, and D. Wedelin. Parallel integer optimization for crew scheduling. *Annals of Operations Research*, 99(1):141–166, 2000.

[8] N. Alon, J. H. Spencer, and P. Erdõs. *The Probabilistic Method*. Wiley, 1992.

[9] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian paths and matchings. *J. Comput. System Sci.*, 18:155–193, 1979.

[10] M. J. Atallah, D. Z. Chen, and O. Daescu. Efficient parallel algorithms for planar st–graphs. In *Proc. 8th. Int. Symp. Algorithms and Computation*, volume 1350 of *LNCS*, pages 223–232. Springer, 1997.

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

[12] D. A. Bader, D. R. Helman, and J. Jájá. Practical parallel algorithms for personalized communication and integer sorting. *Journal of Experimental Algorithms*, 3(1):1–42, 1996.

[13] A. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[14] H. Bast and T. Hagerup. Fast and reliable parallel hashing. In *3rd Symposium on Parallel Algorithms and Architectures*, pages 50–61, 1991.

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

[16] B. Bollobás. Degree sequences of random graphs. *Discrete Mathematics*, 33:1–19, 1981.

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

[18] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–208, 1974.

[19] G. S. Brodal. Worst-case efficient priority queues. In *Proc. 7th Ann. Symp. on Discrete Algorithms*, pages 52–58. ACM–SIAM, 1996.

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

[21] 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.

[22] K. M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Communications of the ACM*, 25(11):833–837, 1982.

[23] 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.

[24] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I : Sequential algorithms. *Algorithmica*, 27(3/4):212–226, 2000.

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

[26] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

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

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

[29] E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM*, 47:132–166, 2000.

[30] R. Cole and U. Vishkin. Approximate coin tossing with applications to list, tree and graph porblems. In *Proc. 27th Ann. Symp. on Foundations of Computer Science*, pages 478–491. IEEE, 1986.

[31] J. Cong, A. B. Kahng, and K. S. Leung. Efficient algorithms for the minimum shortest path Steiner arborescence problem with applications to VLSI physical design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(1):24–39, 1998.

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

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

[34] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramanian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.

[35] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):61–67, 1981.

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

[37] N. Deo and C. Pang. Shortest–path algorithms: Taxonomy and annotation. *Networks*, 14:275–323, 1984.

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

[39] R. B. Dial, F. Glover, D. Karney, and D. Klingman. A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. *Networks*, 9:215–248, 1979.

[40] J. Díaz, J. Petit, and M. Serna. Random geometric problems on $[0,1]^2$. In *RANDOM: International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 1518 of *LNCS*, pages 294–306. Springer, 1998.

[41] J. Díaz, J. Petit, and M. Serna. A guide on concentration bounds. In S. Rajasekaran, P. Pardalos, J. Reif, and J. Rolim, editors, *Handbook of Randomized Computing*, pages 457–507. Kluwer, 2001.

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

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

[44] 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(11):1343–1354, 1988.

[45] 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.

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

[47] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proc. 42th Symp. on Foundations of Computer Science (FOCS)*. IEEE, 2001.

[48] W. Feller. *An introduction to probability theory and its applications*, volume I. Wiley, 1968.

[49] W. Feller. *An introduction to probability theory and its applications*, volume II. Wiley, 1971.

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

[51] A. Formella, J. Keller, and T. Walle. HPP: A high performance PRAM. In *Proc. Euro-Par 1996 Parallel Processing*, volume 1124 II of *LNCS*, pages 425–434. Springer, 1996.

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

[53] 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.

[54] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.

[55] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48:533–551, 1994.

[56] A. Frieze. Shortest path algorithms for knapsack type problems. *Mathematical Programming*, 11:150–157, 1976.

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

[58] 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.

[59] G. Gallo and S. Pallottino. Shortest path methods: A unifying approach. *Math. Programming Study*, 26:38–64, 1986.

[60] G. Gallo and S. Pallottino. Shortest path algorithms. *Ann. Oper. Res.*, 13:3–79, 1988.

[61] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, August 1994.

[62] A. M. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[63] A. M. Gibbons and W. Spirakis. *Lectures on Parallel Computation*. Cambridge University Press, 1993.

[64] F. Glover, R. Glover, and D. Klingman. Computational study of an improved shortest path algorithm. *Networks*, 14:23–37, 1984.

[65] F. Glover, D. Klingman, and N. Phillips. A new polynomially bounded shortest path algorithm. *Operations Research*, 33:65–73, 1985.

[66] F. Glover, D. Klingman, N. Phillips, and R. F. Schneider. New polynomial shortest path algorithms and their computational attributes. *Management Science*, 31:1106–1128, 1985.

[67] A. V. Goldberg. A simple shortest path algorithm with linear average time. In *Proc. 9th Ann. European Symposium on Algorithms (ESA)*, number 2161 in LNCS, pages 230–241. Springer, 2001.

[68] A. V. Goldberg. Shortest path algorithms: Engineering aspects. In *Proc. 12th Intern. Symposium on Algorithms and Computation (ISAAC 2001)*, number 2223 in LNCS, pages 502–513. Springer, 2001.

[69] A. V. Goldberg and T. Radzik. A heuristic improvement of the Bellman-Ford algorithm. *Applied Mathematics Letters*, 6:3–6, 1993.

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

[71] B. Golden and T. Magnanti. Transportation planning: Network models and their implementation. *Studies in Operations Management*, pages 365–518, 1978.

[72] L. M. Goldschlager. A unified approach to models of synchronous parallel machines. *Journal of the ACM*, 29(4):1073–1086, 1982.

[73] G. R. Grimmett. *Percolation*, volume 321 of *Grundlehren der mathematischen Wissenschaften*. Springer, 2nd edition, 1999.

[74] G. R. Grimmett and D. R. Stirzaker. *Probability and random processes*. Oxford University Press, 3. edition, 2001.

[75] 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.

[76] 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.

[77] T. Hagerup and C. R¨ub. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1990.

[78] 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.

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

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

[81] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, August 1997.

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

[83] D. Houck, J. Picard, M. Queyranne, and R. Vemuganti. The traveling salesman problem as a constrained shortest path problem: Theory and computation experience. *Opsearch (India)*, 17:94–109, 1980.

[84] M. R. Hribar and V. E. Taylor. Performance study of parallel shortest path algorithms: Characteristics of good decompositions. In *Proc. 13th Ann. Conf. Intel Supercomputers Users Group (ISUG)*, 1997.

[85] M. R. Hribar, V. E. Taylor, and D. E. Boyce. Termination detection for parallel shortest path algorithms. *Journal of Parallel and Distributed Computing*, 55:153–165, 1998.

[86] M. R. Hribar, V. E. Taylor, and D. E. Boyce. Implementing parallel shortest path for parallel transportation applications. *Parallel Computing*, 27(12):1537–1568, 2001.

[87] M. S. Hung and J. J. Divoky. A computational study of efficient shortest path algorithms. *Comput. Oper. Res.*, 15(6):567–576, 1988.

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

[89] N. L. Johnson and S. Kotz. *Urn Models and Their Applications*. Wiley, 1977.

[90] R. M. Karp. The transitive closure of a random digraph. *Random Structures and Algorithms*, 1(1):73–93, 1990.

[91] R. M. Karp and Y. Zhang. Finite branching processes and AND/OR tree evaluation. Technical Report TR-93-043, ICSI, Berkeley, 1993.

[92] 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.

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

[94] V. F. Kolchin, B. A. Sevatyanov, and V. P. Chistiakov. *Random Allocations*. V. H. Winston, 1978.

[95] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. In *Proc. 8th International World-Wide Web Conference*, 1999.

[96] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[97] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[98] L. Lu. The diameter of random massive graphs. In *Proc. 12th Ann. Symp. on Discrete Algorithms*, pages 912–921. ACM–SIAM, 2001.

[99] M. Marín. Asynchronous (time-warp) versus synchronous (event-horizon) simulation time advance in BSP. In *Proc. Euro-Par 1998 Parallel Processing*, volume 1470 of *LNCS*, pages 897–905, 1998.

[100] C. McDiarmid. Concentration. In M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, and B. Reed, editors, *Probabilistic Methods for Algorithmic Discrete Mathematics*, volume 16 of *Algorithms and Combinatorics*, pages 195–248. Springer, 1998.

[101] B. D. McKay and N. C. Wormald. The degree sequence of a random graph, I: The models. *Random Structures and Algorithms*, 11:97–118, 1997.

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

[103] U. Meyer. Heaps are better than buckets: Parallel shortest paths on unbalanced graphs. In *Proc. Euro-Par 2001 Parallel Processing*, volume 2150 of *LNCS*, pages 343–351. Springer, 2001.

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

[105] U. Meyer. Buckets strike back: Improved parallel shortest paths. In *Proc. 16th Intern. Parallel and Distributed Processing Symposium (IPDPS 2002)*. IEEE, 2002.

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

[107] 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.

[108] G. L. Miller and J. H. Reif. Parallel tree contraction, Part 1: Fundamentals. In S. Micali, editor, *Advances in Computing Research 5: Randomness and Computation*. JAI Press, 1989.

[109] 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.

[110] M. Molloy. The Probabilistic Method. In M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, and B. Reed, editors, *Probabilistic Methods for Algorithmic Discrete Mathematics*, volume 16 of *Algorithms and Combinatorics*, pages 1–35. Springer, 1998.

[111] J.-F. Mondou, T. G. Crainic, and S. Nugyen. Shortest path algorithms: A computational study with the C programming language. *Comput. Oper. Res.*, 18:767–786, 1991.

[112] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[113] K. Mulmuley and P. Shah. A lower bound for the shortest path problem. In *Proc. 15th Annual Conference on Computational Complexity*, pages 14–21. IEEE, 2000.

[114] M. Nonato, S. Pallottino, and B. Xuewen. SPT_L shortest path algorithms: review, new proposals and some experimental results. Technical report TR-99-16, University of Pisa, Department of Computer Science, 1999.

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

[116] 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.

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

[118] U. Pape. Implementation and efficiency of Moore-algorithms for the shortest route problem. *Math. Programming*, 7:212–222, 1974.

[119] S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proc. 13th Ann. Symp. on Discrete Algorithms*, pages 267–276. ACM-SIAM, January 6–8 2002.

[120] V. Priebe. *Average-case complexity of shortest-paths problems*. PhD thesis, Universit¨at des Saarlandes, 2001.

[121] P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

[122] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.

[123] 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.

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

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

[126] O. Riordan and A. Selby. The maximum degree of a random graph. *Combinatorics, Probability and Computing*, 9:549–572, 2000.

[127] C. Scheideler. *Probabilistic Methods for Coordination Problems*. HNI–Verlagsschriftenreihe Vol. 78, University of Paderborn, 2000.

[128] M. Schwartz and T. E. Stern. Routing techniques used in computer communication networks. *IEEE Transactions on Communications*, pages 539–552, 1980.

[129] R. Sedgewick and P. Flajolet. *An Introduction to the analysis of algorithms*. Addison-Wesley, 1996.

[130] R. Sedgewick and J. S. Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1:31–48, 1986.

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

[132] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, 1981.

[133] 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.

[134] S. Subramanian, R. Tamassia, and J. S. Vitter. An efficient parallel algorithm for shortest paths in planar layered digraphs. *Algorithmica*, 14(4):322–339, 1995.

[135] T. Takaoka. Theory of 2-3 heaps. In *Computing and Combinatorics*, volume 1627 of *LNCS*, pages 41–50. Springer, 1999.

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

[137] M. Thorup. Floats, integers, and single source shortest paths. *Journal of Algorithms*, 35:189–201, 2000.

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

[139] J. L. Tr¨aff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21:1505–1532, 1995.

[140] J. L. Tr¨aff 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.

[141] J. D. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, February 1991.

[142] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[143] M. S. Waterman. *Mathematical Methods for DNA Sequences*. CRC Press, Boca Raton, FL, 1988.

[144] J. W. J. Williams. Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

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

[146] U. Zwick. Exact and approximate distances in graphs - a survey. In *Proc. 9th Ann. European Symposium on Algorithms (ESA)*, number 2161 in LNCS, pages 33–48. Springer, 2001.

Summary

This thesis deals with the average-case complexity of algorithms for a basic combinatorial-optimization problem: computing shortest paths on directed graphs with weighted edges. We focus on the single-source shortest-paths (SSSP) version that asks for minimum weight paths from a designated source node of a graph to all other nodes; the weight of a path is given by the sum of the weights of its edges. Shortest-paths problems are among the most fundamental and also the most commonly encountered graph problems, both in themselves and as subproblems in more complex settings. We consider SSSP algorithms under the classical sequential (single processor) model and for parallel processing, that is, having several processors working in concert. Computing SSSP on a parallel computer may serve two purposes: solving the problem faster than on a sequential machine and/or taking advantage of the aggregated memory in order to avoid slow external memory computing. Currently, however, parallel and external memory SSSP algorithms still constitute major performance bottlenecks. In contrast, internal memory sequential SSSP for graphs with nonnegative edge weights is quite well understood: numerous SSSP algorithms have been developed, achieving better and better asymptotic worst-case running times. On the other hand, many sequential SSSP algorithms with less attractive worst-case behavior perform very well in practice but there are hardly any theoretical explanations for this phenomenon. Mathematical average-case analysis for shortest-paths algorithms has focused on the All-Pairs Shortest-Paths problem for a simple graph model, namely the complete graph with random edge weights.

One of the main contributions of this thesis is a thorough mathematical average-case analysis of sequential SSSP algorithms on *arbitrary* directed graphs. Our problem instances are directed graphs with n nodes and m edges whose edge weights are randomly chosen according to the uniform distribution on $[0, 1]$. We present both label-setting and label-correcting algorithms that solve the SSSP problem on such instances in time $O(n + m)$ on the average. For independently random edge weights, the average-case time-bound can also be obtained with high probability. All previous algorithms required superlinear time. The new SSSP algorithms do not use exact priority queues, but simple hierarchical bucket structures with adaptive splitting instead: The label-setting algorithm **SP-S** aims to split the current bucket until a single vertex remains in it, whereas the label-correcting algorithm **SP-C** adapts the width of the current bucket to the maximum degree of the vertices contained in it. Our result also yields an $\mathcal{O}(n^2 + n \cdot m)$ average-case time algorithm for All-Pairs Shortest-Paths, thus improving upon the best previous bounds on sparse directed graphs.

Only very little is known about the average-case performance of previous SSSP algorithms. Our research yields the first theoretical average-case analysis for the "**A**pproximate

Bucket **I**mplementation" of Dijkstra's algorithm (ABI–Dijkstra): for random edge weights and either random graphs or graphs with constant maximum node degree we show how the bucket width must be chosen in order to achieve linear $\mathcal{O}(n + m)$ average-case execution time.

Worst-case inputs for label-correcting SSSP algorithms are usually based on the following principle: Paths with a few edges are found earlier but longer paths have smaller total weights and hence lead to many costly updates on the tentative distances. For random edge weights, however, it is unlikely that a given long path has a small total path weight. Thus, the carefully constructed worst-case graphs will usually not cause large running times. We present a general method to deal with this problem. We come up with constructive existence proofs for graph classes with random edge weights on which ABI–Dijkstra and several other well-known SSSP algorithms are forced to run in superlinear time on average. It is worth mentioning that the constructed graphs contain only $\mathcal{O}(n)$ edges, thus maximizing the performance gap as compared to our new approaches with linear average-case time.

The second part of the thesis deals with parallel SSSP algorithms. The parallel random access machine (PRAM) is one of the most widely studied abstract models of a parallel computer. A PRAM consists of p independent processors and a shared memory, which these processors can synchronously access in unit time. The performance of PRAM algorithms is usually described by the two parameters *time* (assuming an unlimited number of available PUs) and *work* (the total number of operations needed). A fast and efficient parallel algorithm minimizes both time and work; ideally the work is asymptotic to the sequential complexity of the problem. A number of SSSP PRAM algorithms has been invented to fit the needs of parallel computing. Unfortunately, most of them require significantly more work than their sequential counterparts.

We present new results for a number of important graph classes; for example, we provide the first work-optimal PRAM algorithms that require sublinear average-case time for sparse random graphs, and graphs modeling the WWW, telephone calls or social networks. Most of our algorithms are derived from the new sequential label-correcting approach exploiting the fact that certain operations can be performed independently on different processors or disks. The algorithms are analyzed in terms of quite general graph properties like (expected) diameter, maximum shortest-path weight or node degree sequences. For certain parameter ranges, already very simple extensions provably do the job; other parameters require more involved data structures and algorithms. Sometimes, our methods do not lead to improved algorithms at all, e.g., on graphs with linear diameter. However, such inputs are quite atypical.

Zusammenfassung

In dieser Arbeit untersuchen wir die *average-case*-Komplexit¨at von Algorithmen f¨ur das K¨urzeste-Wege Problem mit einem Startpunkt (*Single-Source Shortest-Paths*, SSSP). K¨urzeste-Wege Probleme nehmen einen breiten Raum der kombinatorischen Optimierung ein und haben viele praktische Anwendungen.

Um Aussagen ¨uber das "durchschnittliche Verhalten" eines Algorithmus zu treffen werden die Eingaben gem¨aß einer Wahrscheinlichkeitsverteilung auf der Menge aller m¨oglichen Eingaben erzeugt. Eingaben f¨ur SSSP bestehen aus gerichteten Graphen mit n Knoten, m Kanten und nichtnegativen Kantengewichten. Im Gegensatz zu den meisten fr¨uheren Arbeiten nehmen wir lediglich eine zuf¨allige Verteilung der Kantengewichte an; die Graphstruktur kann beliebig sein. Deshalb k¨onnen sich unsere Algorithmen nicht auf strukturelle Eigenarten verlassen, die bei Graphen mit zuf¨alliger Kantenverkn¨upfung mit hoher Wahrscheinlichkeit auftreten und die SSSP Berechnung erleichtern.

Alle bisherigen SSSP Algorithmen ben¨otigten auf d¨unnen gerichteten Graphen superlineare Zeit. Wir stellen den ersten SSSP Algorithmus vor, der auf beliebigen gerichteten Graphen mit zuf¨alligen Kantengewichten eine beweisbar lineare *average-case*-Komplexit¨at $\mathcal{O}(n + m)$ besitzt. Sind die Kantengewichte unabh¨angig, so wird die lineare Zeitschranke auch mit hoher Wahrscheinlichkeit eingehalten. Außerdem impliziert unser Ergebnis verbesserte *average-case*-Schranken f¨ur das *All-Pairs Shortest-Paths* (APSP) Problem auf Graphen mit $o(n \cdot \log n)$ Kanten.

Der neue Algorithmus benutzt eine approximative Priorit¨ats-Datenstruktur. Diese besteht anfangs aus einem eindimensionalen Feld von *Buckets*. Je nachdem wie sich die Eintr¨age in der Datenstruktur verteilen, k¨onnen die Buckets feiner unterteilt werden. Durch wiederholtes Aufspalten der Buckets bildet sich eine Hierarchie, die es erm¨oglicht kritische Eintr¨age schnell zu lokalisieren.

Unser neuer Ansatz kommt in zwei Varianten: **SP-S** folgt dem *label-setting*-Paradigma, w¨ahrend **SP-C** einen *label-correcting*-Algorithmus darstellt. Die besten *beweisbaren* Laufzeitschranken f¨ur SSSP wurden bisher durchg¨angig f¨ur *label-setting*-Algorithmen gefunden. Andererseits waren *label-correcting*-Ans¨atze in der Praxis oft erheblich schneller als *label-setting*-Algorithmen. Unsere Arbeit zeigt, daß zumindest im *average-case* beide Paradigmen eine asymptotisch optimale Leistung erlauben.

Quasi als Nebenprodukt der Analyse f¨ur **SP-C** erhalten wir die erste theoretische *average-case*-Analyse f¨ur die *"Approximate Bucket Implementierung"* von Dijkstras SSSP Algorithmus (ABI-Dijkstra): insbesondere f¨ur Zufallsgraphen und Graphen mit konstanten Knotengraden ist die durchschnittliche Laufzeit bei zuf¨alligen Kantengewichten linear.

Zuf¨allige nichtnegative Kantengewichte erleichtern das SSSP Problem insofern, als daß

das Auftreten langer Pfade mit kleinem Gesamtgewicht sehr unwahrscheinlich wird. Andererseits beruhen *worst-case*-Eingaben f¨ur viele altbekannte *label-correcting*-Algorithmen wie ABI-Dijkstra gerade auf einer Staffelung langer Pfade mit kleinen Gesamtgewichten: w¨ahrend der Ausf¨uhrung findet der Algorithmus immer wieder bessere Verbindungen, die das Korrigieren vieler Distanzwerte erfordern, und somit den Algorithmus ausbremsen. Dieses ausgekl¨ugelte Schema f¨allt bei zuf¨alligen Kantengewichten in der Regel vollkommen in sich zusammen.

Wir stellen eine allgemeine Methode vor, um *worst-case*-Eingaben mit festen Kantengewichten in schwierige Eingaben mit zuf¨alligen Kantengewichten zu verwandeln. Mittels dieser Methode zeigen wir, daß es Graphklassen mit zuf¨alligen Kantengewichten gibt, auf denen eine Reihe etablierter SSSP Algorithmen superlineare *average-case*-Laufzeiten ben¨otigen. F¨ur den *Bellman-Ford* Algorithmus ergibt sich zum Beispiel eine durchschnittliche Laufzeit von $\Omega(n^{4/3-\epsilon})$ f¨ur beliebig kleine $\epsilon > 0$; bei ABI-Dijkstra betr¨agt die durchschnittliche Laufzeit auf solchen Graphklassen immerhin noch $\Omega(n \cdot \log n / \log\log n)$. Diese Ergebnisse unterstreichen die Bedeutung unseres neuen SSSP Algorithmus, der auf beliebigen Graphen mit zuf¨alligen Kantengewichten in linearer *average-case*-Zeit terminiert.

Neben dem klassischen seriellen (Ein-Prozessor) Berechnungsmodell untersuchen wir das SSSP Problem auch f¨ur Parallelverarbeitung. Ein guter paralleler Algorithmus ben¨otigt sowohl wenige parallele Schritte (Zeit) als auch eine m¨oglichst geringe Gesamtzahl an Operationen (Arbeit). Leider gibt es bisher keine parallelen SSSP Algorithmen, die bei beiden Parametern beweisbar gut abschneiden. Wir erzielen Verbesserungen f¨ur umfangreiche Graphklassen mit zuf¨alligen Kantengewichten, wie z.B. d¨unne Zufallsgraphen oder Graphen, die das WWW, Telefonanrufe oder soziale Netzwerke modellieren. Unsere Ans¨atze stellen die derzeit schnellsten parallelen SSSP Algorithmen mit durchschnittlich linearer Arbeit dar.

Dabei betrachten wir zun¨achst einfache Parallelisierungen des ABI-Dijkstra Algorithmus. Diese sind schon recht effizient, wenn der maximale Knotengrad nicht wesentlich gr¨oßer ist als der durchschnittliche Knotengrad. Bei extrem unbalancierten Knotengraden st¨oßt diese Methode jedoch sehr schnell an ihre Grenzen: zu viele Buckets m¨ussen nacheinander durchlaufen werden. Eine direkte Parallelisierung von **SP-C** w¨urde das Problem nur teilweise beseitigen, da es durch dynamische Bucket-Spaltungen immer noch zu einer großen Zahl zu traversierender Buckets kommen k¨onnte. Deshalb haben wir eine spezielle parallele Variante entwickelt, die auf mehreren Bucketstrukturen unterschiedlicher Auspr¨agung gleichzeitig arbeitet und ohne Bucket-Spaltungen auskommt. Innerhalb jeder Bucketstruktur k¨onnen leere Buckets schnell ¨ubersprungen werden. Bei durchschnittlich linearer Arbeit kann SSSP auf Graphen wie sie zum Beispiel durch das World-Wide-Web gegeben sind in $o(\sqrt{n})$ durchschnittlicher paralleler Zeit gel¨ost werden.

Curriculum Vitae

Personal Data

Last name	Meyer
First names	Ulrich
Citizenship	German
Place of birth	Dudweiler, Germany
Date of birth	10-11-1971
Social status	married, one daughter

Education

Secondary School	1982-1991	Staatliches Gymnasium Sulzbach
Study	1991 - 1997	Computer Science (Minor: Economics) at Univ. des Saarlandes, Saarbr¨ucken
Bachelor's degree	1993	Grade : good
Master's degree	1997	Grade : very good
Ph.D. research	1997 -	Max-Planck Institute (MPI) for Computer Science, Saarbr¨ucken. Advisor: Prof. Dr. Kurt Mehlhorn
Guest researcher	01/2002 - 02/2002	Computer and Automation Research Institute of the Hungarian Academy of Sciences

Publications

Master Thesis:

[1] U. Meyer. Deterministische Simulation einer PRAM auf Gitterrechnern. Mastersthesis (in German), Universit¨at des Saarlandes, 1995.

Refereed Conference Papers:

[2] L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. In *Proc. 7th Intern. Workshop on Algorithms and Data Structures (WADS 2001)*, volume 2125 of *LNCS*, pages 471–482. Springer, 2001.

[3] K. Brengel, A. Crauser, U. Meyer, and P. Ferragina. An experimental study of priority queues in external memory. In *Proc. 3rd Intern. Workshop on Algorithm Engineering (WAE-99)*, volume 1668 of *LNCS*, pages 345–359. Springer, 1999.

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

[5] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos. Randomized external-memory algorithms for some geometric problems. In *Proc. 14th ACM Symposium on Computational Geometry (SCG-98)*, pages 259–268. ACM, 1998.

[6] S. Edelkamp and U. Meyer. Theory and practice of time-space trade-offs in memory limited search. In *Proc. Joint German/Austrian Conference on Artificial Intelligence (KI-2001)*, volume 2174 of *LNAI*, pages 169–184. Springer, 2001.

[7] M. Kaufmann, U. Meyer, and J. F. Sibeyn. Towards practical permutation routing on meshes. In *Proc. 6th IEEE Symposium on Parallel and Distributed Processing*, pages 656–663. IEEE, 1994.

[8] M. Kaufmann, U. Meyer, and J. F. Sibeyn. Matrix transpose on meshes: Theory and practice. In *Proc. 11th International Parallel Processing Symposium (IPPS-97)*, pages 315–319. IEEE, 1997.

[9] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th Ann. European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 723–735. Springer, 2002.

[10] U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proc. 12th Ann. Symp. on Discrete Algorithms*, pages 87–88. ACM–SIAM, 2001.

[11] U. Meyer. Heaps are better than buckets: Parallel shortest paths on unbalanced graphs. In *Proc. Euro-Par 2001 Parallel Processing*, volume 2150 of *LNCS*, pages 343–351. Springer, 2001.

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

[13] U. Meyer. Buckets strike back: Improved parallel shortest paths. In *Proc. 16th Intern. Parallel and Distributed Processing Symposium (IPDPS 2002)*. IEEE, 2002.

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

[15] 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.

[16] U. Meyer and J. F. Sibeyn. Simulating the simulator: Determinstic pram simulation on a mesh simulator. In *Eurosim '95*, pages 285–290. Elsevier, 1995.

[17] U. Meyer and J. F. Sibeyn. Gossiping large packets on full-port tori. In *Proc. Euro-Par 1998 Parallel Processing*, volume 1470 of *LNCS*, pages 1040–1046. Springer, 1998.

[18] J. F. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth first search on directed graphs. In *Proc. Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 282–292. ACM, 2002.

Refereed Journal Papers:

[19] K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An experimental study of priority queues in external memory. *Journal of Experimental Algorithmics*, 5, 2000.

[20] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos. I/O-optimal computation of segment intersections. In *Proc. DIMACS Workshop on External Algorithms and Visualization*, volume 50 of *DIMACS Series in Discr. Math. and Theor. Comp. Sci.*, pages 131–138. AMS, 1999.

[21] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. A. Ramos. Randomized external-memory algorithms for line segment intersection and other geometric problems. *International Journal of Computational Geometry and Applications*, 11(3):305–338, June 2001.

[22] M. Kaufmann, U. Meyer, and J. F. Sibeyn. Matrix transpose on meshes: Theory and practice. *Computers and Artificial Intelligence*, 16(2):107–140, 1997.

[23] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *Journal of Algorithms*, 2002. Accepted for publication.

[24] U. Meyer and J. F. Sibeyn. Oblivious gossiping on tori. *Journal of Algorithms*, 42(1):1–19, 2002.