Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT
Journal of Algorithms 49 (2003) 114–152
Journal of Algorithms
www.elsevier.com/locate/jalgor

Δ-stepping: a parallelizable shortest path algorithm

U. Meyer and P. Sanders [*,1]

Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany

Received 23 December 1998

Abstract

The single source shortest path problem for arbitrary directed graphs with n nodes, m edges and nonnegative edge weights can sequentially be solved using $\mathcal{O}(n \cdot \log n + m)$ operations. However, no work-efficient parallel algorithm is known that runs in sublinear time for arbitrary graphs. In this paper we present a rather simple algorithm for the single source shortest path problem. Our new algorithm, which we call Delta-stepping, can be implemented very efficiently in sequential and parallel setting for a large class of graphs. For random edge weights and arbitrary graphs with maximum node degree d, sequential Δ-stepping needs $\mathcal{O}(n + m + d \cdot L)$ total average-case time, where L denotes the maximum shortest path weight from the source node s to any node reachable from s. For example, this means linear time on directed graphs with constant maximum degree. Our best parallel version for a PRAM takes $\mathcal{O}(d \cdot L \cdot \log n + \log^2 n)$ time and $\mathcal{O}(n + m + d \cdot L \cdot \log n)$ work on average. For random graphs, even $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n + m)$ work on average can be achieved. We also discuss how the algorithm can be adapted to work with nonrandom edge weights and how it can be implemented on distributed memory machines. Experiments indicate that already a simple implementation of the algorithm achieves significant speedup on real machines.

1. Introduction

The *single-source shortest-path problem* (SSSP) is a fundamental and well-studied combinatorial optimization problem with many practical and theoretical applications [2]. Numerous SSSP algorithms have been developed, achieving better and better asymptotic running times. Unfortunately, on parallel machines, fast and efficient SSSP computations still constitute a major bottleneck. On the other hand, many sequential SSSP algorithms

[*] Corresponding author.
E-mail addresses: umeyer@mpi-sb.mpg.de (U. Meyer), sanders@mpi-sb.mpg.de (P. Sanders).
[1] Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

0196-6774/$ – see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/S0196-6774(03)00076-2

with less attractive worst-case behavior perform very well in practice but there are no theoretical explanations for this phenomenon. In this paper we propose a simple SSSP algorithm of the above kind that provably runs in linear average time for a large class of graphs and at the same time allows efficient parallelization.

Let $G = (V, E)$ be a directed graph with $|V| = n$ nodes and $|E| = m$ edges, let s be a distinguished vertex ("source") of the graph, and c be a function 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 weight of a minimum-weight ("shortest") path from s to v, denoted by $\mathrm{dist}(s, v)$, abbreviated $\mathrm{dist}(v)$; the *weight of a path* is the sum of the weights of its edges. We set $\mathrm{dist}(u, v) := \infty$ if v is unreachable from u. The *maximum shortest path weight* for the source node s is defined as $L(s) := \max\{\mathrm{dist}(s, v)\colon \mathrm{dist}(s, v) < \infty\}$, abbreviated L. Finally, note the distinction between the weight of a path and the *size of a path* which is defined to be the number of edges on the path.

Sequential shortest path algorithms commonly apply iterative labeling methods based on maintaining a *tentative distance* for all nodes; $\mathrm{tent}(v)$ is always ∞ or the weight of some path from s to v and hence an upper bound on $\mathrm{dist}(v)$. Tentative distances are improved by performing *edge relaxations*, i.e., for an edge $(v, w) \in E$ the algorithm sets $\mathrm{tent}(w) := \min\{\mathrm{tent}(w), \mathrm{tent}(v) + c(v, w)\}$. There are two major types of labeling methods—*label-setting* and *label-correcting*. Label-setting algorithms, such as Dijkstra's algorithm [29] designate the distance label of one node v as permanent (optimal, "settled") at each iteration. Hence, at most n iterations are required. Label-correcting algorithms can relax edges of nonsettled nodes and may vary in the number of iterations needed to complete the computation: all labels are considered temporary (they may be considered several times) until the final step, when they all become permanent. The best label-setting algorithms have significantly better worst case bounds than that of any label-correcting algorithm. In spite of good practical performance [17,93], the power of label-correcting approaches for SSSP is hardly covered by theoretical average-case analysis.

We address the deficit described above by providing a simple sequential label-correcting algorithm which *provably* achieves optimal linear time with high probability (whp)[2] for a large graph class with random edge weights. Our Δ-stepping algorithm maintains eligible nodes with tentative distances in an array of buckets each of which represents a distance range of size Δ. During each phase, the algorithm removes all nodes of the first nonempty bucket and relaxes all outgoing edges of weight at most Δ. Edges of higher weight are only relaxed after their respective starting nodes are surely settled. The choice of Δ should provide a good trade-off between too many node re-considerations on the one hand and too many bucket traversals on the other hand. We show that taking $\Delta = \Theta(1/d)$ for graphs with maximum node degree d and *random* edge weights uniformly distributed in $[0, 1]$, the algorithm needs $\mathcal{O}(n + m + d \cdot L)$ total average-case time where L denotes the maximum shortest path weight from the source node s to any node reachable from s. For example, this means linear average-case time on arbitrary directed graphs with bounded constant degree.

[2] *With high probability* (*whp*) means that the probability for some event is at least $1 - n^{-\beta}$ for any constant $\beta > 0$.

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) [36,47,58] 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. Even though the strict PRAM model is only implemented on a few experimental parallel machines like the SB-PRAM [35] it is valuable to highlight the main ideas of a parallel algorithm without tedious details caused by a particular architecture. Other, more realistic models like BSP [91] and LogP [23] view a parallel computer as a collection of sequential processors, each one having its own local memory, so-called *distributed memory machines* (*DMMs*). The PUs are interconnected by a network that allows them to communicate by sending and receiving messages. Communication constraints are imposed by latency, limited network bandwidth and synchronization delays. In this article, we use the PRAM model to describe and analyze the basic structure of our parallel algorithm first, then we provide details of a conversion to DMMs.

A number of new SSSP algorithms has been invented to fit the needs of parallel computation models. A fast and efficient parallel algorithm minimizes both *time* and *work* (product of time and number of processors). Ideally, the work bound matches the complexity of the best (known) sequential algorithm. Unfortunately, even on the stronger PRAM model, most of them perform significantly more work than their sequential counterparts. Currently, there is no work efficient algorithm which achieves sublinear time on arbitrary graphs with nonnegative edge weights. Implementations on parallel computers mostly apply some simple kind of graph partitioning, and then each processor runs a sequential label-correcting algorithm on its own partition. In between, the processors exchange distance information. For certain input classes, some of these implementations perform fairly well even though no speed-up can be achieved in the worst case. However, thorough theoretical justification for the actually observed performance is largely missing.

Based on the sequential Δ-stepping algorithm we give work-efficient extensions to PRAMs and distributed memory machines (DMMs). For random edge weights, the expected parallel execution time can again be stated in terms of the maximum node degree and the expected maximum weight among all shortest paths in the graph. We prove sublinear average-case time and linear average-case work for several graph classes with random edge weights. In particular, we show $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n+m)$ work on average for random graphs with random edge weights.

In the following we will provide an outline of previous and related work. Then we will give an overview of our new algorithms, interpret the theoretical results, and sketch the organization of the article.

1.1. Previous and related work

1.1.1. Sequential label-setting algorithms

The basic label-setting approach is Dijkstra's algorithm [29]. It maintains a partition of the node set V into *settled*, *queued* and *unreached* nodes. Settled nodes have reached their final distance value. Queued nodes have finite tentative distance and unreached nodes have $\text{tent}(v) = \infty$. Initially, s is queued, $\text{tent}(s) = 0$, and all other nodes are unreached. In each iteration, the queued node v with the smallest tentative distance is removed from the queue, and all edges (v, w) are relaxed, i.e., $\text{tent}(w)$ is set to $\min\{\text{tent}(w), \text{tent}(v) + c(v, w)\}$. If w was unreached, it is now queued. It is well known that $\text{tent}(v) = \text{dist}(v)$ when v is removed from the queue, i.e., v is settled. Using Fibonacci heaps [37], Dijkstra's algorithm can be implemented to run in $\mathcal{O}(n \cdot \log n + m)$ time. This constitutes the best known bound in the standard model of computation, which forbids bit-manipulations of the edge weights.

A number of faster algorithms have been developed on the more powerful RAM (random access machine) model which basically reflects what one can use in a programming language such as `C`. Nearly all of these algorithms are based on Dijkstra's algorithm and improve the priority queue data structure (see [78,85] for an overview). Thorup [85,86] has given the first $\mathcal{O}(n + m)$ time RAM algorithm for *undirected* graphs with nonnegative floating-point or integer edge weights in $\{0, \ldots, 2^w - 1\}$ for word length w. His approach applies label-setting, too, but significantly deviates from Dijkstra's algorithm in that it does not visit the nodes in order of increasing distance from s but traverses a so-called *component tree*. Unfortunately, Thorup's algorithm requires atomic heaps [38] which are only defined for $n \geqslant 2^{12^{20}}$. Hagerup [49] recently generalized Thorup's approach to directed graphs, the time complexity, however, remains super-linear $\mathcal{O}(n + m \cdot \log w)$. The currently fastest RAM algorithm for sparse directed graphs is due to Thorup [87] and needs $\mathcal{O}(n + m \cdot \log\log n)$ time. Alternative approaches for somewhat denser graphs have been proposed by Raman [77,78]: 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.

Considering the algorithms which we present in this paper we review some basic implementations of Dijkstra's algorithm with *bucket* based priority queues. Bucket approaches are particularly interesting for small *integer* edge weights in $\{1, \ldots, C\}$: in its simplest form the priority queue consists of a linear array B such that bucket $B[i]$ stores the set $\{v \in V\colon\ v \text{ is queued and } \text{tent}(v) \in [i \cdot \Delta, (i+1) \cdot \Delta)\}$. The parameter Δ is called the *bucket width*. For integer weights, taking $\Delta = 1$ ensures that any node in the first nonempty bucket can be settled [26]. The statement remains true for $\Delta \leqslant \Delta_0 = \min\{c(e)\colon\ e \in E\}$ [25,30]. Hence, if $\Delta \leqslant \Delta_0$, the resulting running time is bounded by $\mathcal{O}(m + n \cdot \lceil C/\Delta \rceil)$, also for noninteger weights from $[\Delta_0, C]$. Choosing $\Delta > \Delta_0$ either requires to search a node with smallest tentative distance in the first nonempty bucket (e.g., by an additional heap [25]) or results in a label-correcting algorithm. The latter variant which is also called the "approximate bucket implementation of Dijkstra's algorithm" [2] comes closest to the sequential version of our Δ-stepping algorithm. The description in [2] provides a worst-case analysis for *integer* weights, the average-case performance (in particular for *noninteger* edge weights) is not considered.

Alternative bucket approaches include nested (multiple levels) buckets and/or buckets of different widths [3,25]. The currently best worst-case bound for SSSP on arbitrary *directed* graphs with integer edge weights in $\{1,\dots,C\}$ is $\mathcal{O}(m + n \cdot (\log C)^{1/4+\epsilon})$ expected time for any fixed $\epsilon > 0$ [78].

1.1.2. *Sequential label-correcting algorithms*

The generic SSSP label-correcting algorithm repeatedly selects an arbitrary edge $e = (u, v)$ that violates the optimality condition, i.e., $\text{tent}(v) > \text{tent}(u) + c(e)$, and updates $\text{tent}(v)$ appropriately. The total running time depends on the order of edge relaxations. In the worst case it is pseudo-polynomial: $\mathcal{O}(n^2 \cdot m \cdot C)$ for integer weights, and $\mathcal{O}(m \cdot 2^n)$ otherwise [2]. Improved label-correcting algorithms keep a set Q of nodes so that whenever a distance label $\text{tent}(v)$ decreases, then v is added to Q. In each iteration, the algorithm selects a node u from Q and examines its outgoing edges to update tentative distances (see [17,41] for an overview). The classic Bellman–Ford version [11,34] implements Q as a FIFO-Queue and achieves running time $\mathcal{O}(n \cdot m)$. Since then, none of the newly developed label-correcting algorithms could asymptotically improve this worst-case bound. However, a number of experimental studies [17,25,27,42,44,57,71,93] showed that some recent label-correcting approaches are much faster than Bellman–Ford and even frequently outperform label-setting algorithms.

1.1.3. *Random edge weights and special graph classes*

Average-case analysis of shortest paths algorithms mainly focused on the *All-Pairs Shortest Paths* (APSP) problem on the *complete* graph with random edge weights [21,39, 52,70,79,83]. Mehlhorn and Priebe [65] 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 [74] and Goldberg and Tarjan [46] analyzed the expected number of *decreaseKey* operations in Dijkstra's algorithm; the time bound, however, does not improve over the worst-case complexity of the algorithm.

Improved SSSP algorithms exist for special graph classes, e.g., there are linear time approaches for planar graphs [53] or graphs with constant tree width [16].

1.1.4. *PRAM algorithms*

No parallel PRAM algorithm is known that executes 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. [31] (refining a result of Paige and Kruskal [75]) 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. [13]. These algorithms settle the nodes one by one in the order of Dijkstra's algorithm and only perform edge relaxations in parallel. Hence, using that 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. [50] (based on [24]) implicitly solves the APSP problem by reducing the shortest path 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 [40] 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 [61] for SSSP on sparse graphs with integral nonnegative edge weights summing to W. It requires $\mathcal{O}(\text{polylog}(m+W))$ time with m^2 processors. Recently, Mulmuley and Shah [73] 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 [90]. 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. Based on the solution of an APSP problem on the auxiliary graph the distance values of nondistinguished nodes are properly 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 $t \leqslant \sqrt{n}$ whp.

Klein and Subramanian [62] 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 in order to compute exact single-shortest paths with maximum path weight L by solving a series of $\log L$ 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 [19] and Shi and Spencer [81]. Recently, Cohen [20] 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 gave a SSSP algorithm that takes polylogarithmic time and $\mathcal{O}((n + n^{3 \cdot \mu}) \cdot \text{polylog}(n))$ work provided that a $\mathcal{O}(n^{\mu})$-separator decomposition for the problem instance is provided as a part of the input.

More efficient parallel SSSP algorithms have been designed for special graph classes. Here are some examples: Combining the data structure of [13] with the ideas from [89] gives an algorithm which solves the SSSP problem on planar digraphs with arbitrary nonnegative edge weights in $\mathcal{O}(n^{2\epsilon} + n^{1-\epsilon})$ time and $\mathcal{O}(n^{1+\epsilon})$ work on a CREW PRAM. In contrast, the randomized algorithm of [61] requires planar graphs and integral edge weights summing to W. It runs in $\mathcal{O}(\text{polylog}(m+W))$ time with n processors. Work-efficient SSSP algorithms for planar layered graphs have been proposed by Subramanian et al. [84] and Atallah et al. [7]. Furthermore, there is an $\mathcal{O}(\log^2 n)$ time and linear work EREW PRAM algorithm for graphs with constant tree width [15].

Random graphs [12,33] with unit weight edges have been considered by Clementi et al. [18]: their solution is restricted to constant edge probabilities or edge probability $\Theta(\log^k n/n)$ $(k > 1)$. 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 [79] 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 [40] and Gu and Takaoka [48] 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. [22] gave criteria that divide Dijkstra's algorithm into a number of phases, such that the operations within a phase can be done in parallel; for random graphs with random edge weights, SSSP can be solved in $\mathcal{O}(n^{1/3} \cdot \log n)$ time using $\mathcal{O}(n \cdot \log n + m)$ work on average.

1.1.5. Distributed memory machines (DMMs)

PRAM algorithms can be emulated on distributed memory machines. The loss factors depend on the concrete parameters of the models, e.g., see [43] for emulation results on the BSP model. However, existing implementations [1,14,54–56,88] on parallel computers with distributed memory mostly do not use such emulations but apply some kind of graph partitioning where each processors runs a *sequential* label-correcting algorithm on its subgraph(s). Heuristics are used for the frequency of the inter-processor exchange of distance information, 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 is given.

1.2. New results

We propose and analyze sequential and parallel versions of a label-correcting algorithm for SSSP. Our sequential Δ-stepping approach is similar to the "approximate bucket implementation of Dijkstra's algorithm" [17] in that it maintains eligible nodes with tentative distances in an array B of buckets each of which represents a distance range of Δ. The parameter Δ is a positive real number that is also called the "step width" or "bucket width." Opposite to the algorithm of [17], our approach distinguishes *light edges* (which have weight at most Δ) and *heavy edges* ($c(e) > \Delta$).

Parallelism is obtained by concurrently removing all nodes of the first nonempty bucket (the so-called *current* bucket) and relaxing their outgoing light edges in a single phase. If a node v has been removed from the current bucket $B[i]$ with nonfinal distance value then, in some subsequent phase, v will eventually be *reinserted* into $B[i]$, and the outgoing light edges of v will be re-relaxed. The remaining heavy edges emanating from all nodes that have been removed from $B[i]$ so far are relaxed once and for all when $B[i]$ finally remains empty. Subsequently, the algorithm searches for the next nonempty bucket and proceeds as described above.

The performance of our approach depends on the choice of the step width Δ. We analyze this dependence both in terms of abstract graph parameters and in a more concrete average-case setting.

1.2.1. Random edge weights

We assume independent *random* edge weights that are uniformly distributed[3] in $[0, 1]$. We show that the average-case overhead for re-insertions and re-relaxations is bounded by

[3] The results can be adapted to other distribution functions F, too. For example, it is sufficient if $F(0) = 0$ and $F'(0)$ is bounded from above by some positive constant.

$\mathcal{O}(n+m)$ if we choose $\Delta = \Theta(1/d)$ for graphs with maximum node degree d. We prove that for maximum shortest path weight $L := \max\{\mathrm{dist}(v)\colon \mathrm{dist}(v) < \infty\}$, purely sequential $\Theta(1/d)$-stepping needs $\mathcal{O}(n+m+d\cdot L)$ total average-case time. Thus, SSSP can be solved in linear average-case time whenever $d\cdot L = \mathcal{O}(n+m)$. For example, this is the case on arbitrary directed graphs with bounded constant degree.

A simple parallel version for a CRCW PRAM takes $\mathcal{O}(d^2\cdot L\cdot \log^2 n)$ time on average, our best algorithm achieves $\mathcal{O}(d\cdot L\cdot \log n + \log^2 n)$ time and $\mathcal{O}(n+m+d\cdot L\cdot \log n)$ work on average. For several important graph classes with random edge weights, L is sufficiently small, i.e., $d\cdot L\cdot \log n = o(n)$ with high probability, so that the parallel SSSP can be solved in sublinear time and linear work on average. For example, for r-dimensional meshes with random edge weights we have $L = \mathcal{O}(n^{1/r})$ whp and hence execution time $\mathcal{O}(n^{1/r}\log^2 n)$ using linear work for any constant r whp.

We also consider random graphs [12,33] with random edge weights. We use the random digraph model $D(n,\bar{d}/n)$ that was introduced by Angluin and Valiant [6]. 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) $\bar{d}$, and $m = \bar{d}\cdot n + o(n)$ whp if $\bar{d} \geqslant 1$. For random graphs from $D(n,\bar{d}/n)$ with $\bar{d} \geqslant 2$ and random edge weights, we show $L = \mathcal{O}((1/\bar{d})\cdot \log n)$ whp. Together with the observation $d = \mathcal{O}(\bar{d} + \log n)$ whp this immediately yields an average-case bound of $\mathcal{O}(\log^3 n)$ time and $\mathcal{O}(n+m)$ work. Furthermore, we show that a modified version for random graphs only takes $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n+m)$ work on average (Corollary 19).

In Section 10 we also discuss the performance on *random geometric graphs* which are considered to be a relevant abstraction for many real world situations [28,80].

1.2.2. Arbitrary positive edge weights

The results presented above can be seen as concrete instantiations within a more general framework that does not necessarily assume random edge weights: call a path with total weight at most Δ and without edge repetitions, i.e., every edge appears on the path at most once, a *Δ-path*. Let C_Δ denote the set of all node pairs $\langle u,v\rangle$ connected by some Δ-path $(u,\ldots,v)$ and let $n_\Delta := |C_\Delta|$. Similarly, define $C_{\Delta+}$ as the set of triples $\langle u,v',v\rangle$ such that $\langle u,v'\rangle \in C_\Delta$ and (v',v) is a light edge and let $m_\Delta := |C_{\Delta+}|$. Furthermore, we keep our definition for the maximum weight of a shortest path, $L := \max\{\mathrm{dist}(v)\colon \mathrm{dist}(v) < \infty\}$. Using these parameters we show that the sequential Δ-stepping needs at most $\mathcal{O}(n+m+n_\Delta+m_\Delta+L/\Delta)$ operations. The result for random edge weights mentioned above is obtained by proving $n_\Delta + m_\Delta = \mathcal{O}(n+m)$ whp for $\Delta = \Theta(1/d)$.

For the parallel version we also need a bound on the total number of phases. The number of phases for each nonempty bucket is bounded by one plus the maximum number of edges that occur on any traversed Δ-path: the *maximum Δ-size l_Δ* is defined to be one plus the maximum number of edges needed to connect any pair $\langle u,v\rangle \in C_\Delta$ by a path of minimum weight, i.e.,

$$l_\Delta = 1 + \max_{\langle u,v\rangle \in C_\Delta} \min\bigl\{|A|\colon\ A = (u,\dots,v) \text{ is a minimum-weight } \Delta\text{-path}\bigr\}.$$

A simple parallelization of Δ-stepping runs in time $\mathcal{O}(\frac{L}{\Delta} \cdot d \cdot l_\Delta \cdot \log n)$ and needs work $\mathcal{O}(n+m+n_\Delta+m_\Delta+\frac{L}{\Delta}\cdot d\cdot l_\Delta\cdot\log n)$ whp. The respective average-case bound mentioned before follows by proving $l_\Delta = \mathcal{O}(\log n/\log\log n)$ whp for $\Delta = \Theta(1/d)$. We also show that the factor d can be removed from the execution time using more sophisticated load balancing algorithms.

A further acceleration can be achieved by a preprocessing that actively introduces *shortcut edges* into the graph, i.e., it inserts an edge (u, v) with weight $c(u, v) = \operatorname{dist}(u, v)$ for each shortest path $(u,\dots,v)$ with $\operatorname{dist}(u, v) \leqslant \Delta$. After shortcuts are present, the SSSP algorithm needs at most a constant number of phases for each nonempty bucket. Let n'_Δ (m'_Δ) denote the number of *simple* Δ-paths (plus a light edge) and let l'_Δ denote the number of edges in the longest simple Δ-path. We propose a shortcut insertion algorithm that needs $\mathcal{O}(l'_\Delta \log n)$ time and $\mathcal{O}(n + m + n'_\Delta + m'_\Delta + l'_\Delta \cdot \log n)$ work on a CRCW PRAM whp. For random edge weights and $\Delta = \Theta(1/d)$ the terms simplify as follows: $n'_\Delta + m'_\Delta = \mathcal{O}(n+m)$ and $l'_\Delta = \mathcal{O}(\log n/\log\log n)$ on average, thus resulting in our fastest SSSP algorithm with $\mathcal{O}(d \cdot L \cdot \log n + \log^2 n)$ time and $\mathcal{O}(n + m + d \cdot L \cdot \log n)$ work on average.

Based on the shortcut insertion algorithm, we propose an efficient method which finds the largest possible bucket width Δ that still ensures an $\mathcal{O}(n + m)$ bound on the overhead for node reinsertions and edge re-relaxations.

1.2.3. Further results

Many of the results also transfer to distributed memory machines, and experiments show that already a simple implementation of the algorithm achieves significant speedup on real machines.

Preliminary accounts of the results of the present paper have been published in [67,68]. Subsequently, it was shown that sequential SSSP on arbitrary directed graphs can be solved in linear average-case time [45,66].

1.3. Overview

The rest of the paper is organized as follows: in Section 2, we introduce the Δ-stepping algorithm and analyze its basic properties including sequential execution time in Section 3. Sections 4–8 develop parallel algorithms. The simple algorithms explained in Section 4 are improved in Section 5 using more sophisticated load balancing algorithms. A further acceleration is obtained in Section 6 by precomputing paths consisting of many short edges. Section 7 generalizes the algorithm to work with arbitrary edge weights by providing a parallel algorithm that finds a good value for the step width Δ. In Section 8 we consider a more practical distributed memory model with p processing units (PUs) connected by a network. Simulations and prototypical implementations described in Section 9 demonstrate that at least the simple variants of Δ-stepping are quite practicable. A discussion of some further possibilities for refinements in Section 10 concludes the paper.

2. The basic algorithm

Our sequential Δ-stepping algorithm shown in Fig. 1 resembles the "approximate bucket implementation of Dijkstra's algorithm" [17]. It maintain a one-dimensional array B of *buckets* such that $B[i]$ stores the set $\{v \in V\colon\ v \text{ is queued and } \operatorname{tent}(v) \in [i \cdot \Delta, (i+1) \cdot \Delta)\}$. The parameter Δ is a positive real number that is also called the "step width" or "bucket width." For maximum shortest path weight L, the array B must contain $\lceil L/\Delta \rceil$ buckets. However, by cyclically reusing empty buckets, already $b = \max_{e \in E} \lceil c(e)/\Delta \rceil + 1$ buckets are sufficient. In that case $B[i]$ is in charge of all tentative distances in $[(j \cdot b + i) \cdot \Delta, (j \cdot b + i + 1) \cdot \Delta)$ for all $j \geqslant 0$.

In each *phase*, i.e., each iteration of the inner while-loop, the algorithm removes all nodes from the first nonempty bucket (current bucket) and relaxes all *light* edges ($c(e) \leqslant \Delta$) out of these nodes. This may result in new nodes entering the current bucket which are deleted in the next phase. Furthermore, nodes previously deleted from this bucket may be *reinserted* if their tentative distance has been improved by the previous phase. The relaxation of *heavy* edges ($c(e) > \Delta$) 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).

```
foreach $v \in V$ do tent$(v) := \infty$
relax$(s, 0)$;                                          (* Insert source node with distance 0      *)
while $\neg$isEmpty(B) do                               (* A phase: Some queued nodes left (a) *)
    $i := \min\{j \geqslant 0\colon\ B[j] \neq \emptyset\}$           (* Smallest nonempty bucket (b) *)
    $R := \emptyset$                                     (* No nodes deleted for bucket $B[i]$ yet      *)
    while $B[i] \neq \emptyset$ do                        (* New phase (c) *)
        Req := findRequests($B[i]$, light)               (* Create requests for light edges (d) *)
        $R := R \cup B[i]$                               (* Remember deleted nodes (e) *)
        $B[i] := \emptyset$                              (* Current bucket empty      *)
        relaxRequests(Req)                              (* Do relaxations, nodes may (re)enter $B[i]$ (f) *)
    Req := findRequests($R$, heavy)                      (* Create requests for heavy edges (g) *)
    relaxRequests(Req)                                  (* Relaxations will not refill $B[i]$ (h) *)

Function findRequests($V'$, kind : {light, heavy}) : set of Request
    return $\{(w, \operatorname{tent}(v) + c(v, w))\colon\ v \in V' \wedge (v, w) \in E_{\text{kind}}\}$

Procedure relaxRequests(Req)
    foreach $(w, x) \in$ Req do relax$(w, x)$

Procedure relax$(w, x)$                                  (* Insert or move $w$ in $B$ if $x <$ tent$(w)$ *)
    if $x <$ tent$(w)$ then
        $B[\lfloor \operatorname{tent}(w)/\Delta \rfloor] := B[\lfloor \operatorname{tent}(w)/\Delta \rfloor] \setminus \{w\}$   (* If in, remove from old bucket *)
        $B[\lfloor x/\Delta \rfloor] := B[\lfloor x/\Delta \rfloor] \cup \{w\}$             (* Insert into new bucket *)
        tent$(w) := x$
```

Fig. 1. A sequential variant of Δ-stepping (with cyclical bucket reusage). The sets of light and heavy edges are denoted by E_{light} and E_{heavy}, respectively. Requests consist of a tuple (node, weight).

Subsequently, all heavy edges emanating from these nodes are relaxed once and for all. Then the algorithm sequentially searches for the next nonempty bucket.

For the sequential version, buckets 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. Deletion and edge relaxation for an entire bucket can be done in parallel and in arbitrary order as long as an individual relaxation is atomic, i.e., the relaxations for a particular node are done sequentially. If we want to use more parallelism, we can combine all relaxations for the same node to a single relaxation with smallest new distance. We will give further details of the parallelization in Section 4.

2.1. Difficult inputs

The performance of Δ-stepping crucially depends on the choice of the parameter Δ. For integer weights and $\Delta = 1$, Δ-stepping coincides with Dial's implementation of Dijkstra's algorithm (e.g., [2, Section 4.6]). L buckets will have to be traversed, but no reinsertions occur. Even though the depth of the shortest path tree may be small, there are graphs having $L = \Omega(n \cdot \log n + m)$ such that both sequential and parallel Δ-stepping perform poorly.

On the other hand, for $\Delta \geqslant n \cdot \max_{e \in E} c(e)$ our algorithm will exclusively use the first bucket, hence we obtain a parallel version of the Bellman–Ford algorithm which has high parallelism since all edges can be relaxed in parallel. If k is the maximum depth of the shortest path tree, k passes through the adjacency list of all queued nodes are sufficient. Unfortunately, this may be quite inefficient compared to Dijkstra's algorithm; $\Theta(km)$ operations are needed in the worst-case.

The idea behind Δ-stepping is to find an easily computable fixed Δ that yields a good compromise between these two extremes. However, in particular for the parallel version, this is not always possible.

Figure 2 provides an input with $\mathcal{O}(n)$ edges where *any* fixed Δ either leads to $\Omega(n)$ phases or $\Omega(n)$ operations: The L-part with long edge weights consists of $n' = \Theta(\sqrt{n})$ chains of $n'' = \Theta(\sqrt{n})$ nodes. It is responsible for $\Theta(n)$ distance values that are interspaced by at last n. The S-part consists of a chain of size n'' where the last node spans $\Theta(n)$ single nodes and has in-degree $\Theta(n'')$ from short-cuts along the chain. Thus, if $\Delta = o(n)$ then $\Omega(n)$ different buckets must be inspected, for L, i.e., $\Omega(n)$ phases. On the other hand, $\Delta = \Omega(n)$ leads to $\Theta(n''n) = \Theta(n^{1.5})$ operations due to re-relaxations in part S.

However, in Section 3 we give conditions on when Δ is "good" and it turns out that at least for random edge weights, a good Δ is easy to find. In Section 7 we discuss how to find a good Δ for arbitrary nonnegative edge weights.

3. Analysis of the Δ-stepping algorithm

The purpose of this section is to analyze Δ-stepping on an abstract level that is independent of the machine model used thus producing tools that can be used in the analysis of sequential, PRAM, and distributed memory implementations. Our analysis proceeds in three stages in order to make the results adaptable to different graph classes. In Section 3.1 we consider the number of reinsertions (needed to bound the total number

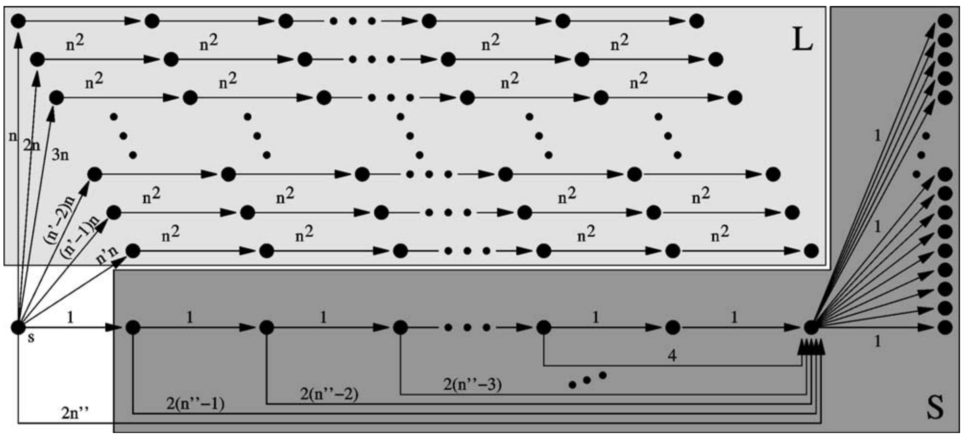


Fig. 2. Bad input graph for Δ-stepping with fixed step width.

of operations) and the number of phases (needed to bound the parallel time) in terms of parameters like d, L, and l_Δ (as described in Section 1.2). Hence, we do not make any assumptions on the class of graphs investigated. Section 3.2 analyzes these conditions for the case of random edge weights. Finally, Section 3.3 completes the analysis by additionally assuming random graphs.

3.1. Reinsertions and progress

The sets C_Δ and $C_{\Delta+}$ as introduced in Section 1.2 play a key role in analyzing both the overhead and the progress of Δ-stepping. The overhead compared to Dijkstra's algorithm is due to *reinsertions* and *re-relaxations*, i.e., insertions of nodes which have previously been deleted and relaxation of their outgoing edges. The following lemma bounds this overhead.

Lemma 1. *The total number of reinsertions is bounded by $n_\Delta = |C_\Delta|$ and the total number of re-relaxations is bounded by $m_\Delta = |C_{\Delta+}|$.*

The proof is quite similar to the correctness proof of Dijkstra's algorithm.

Proof. We give an injective mapping from the set of reinsertions into C_Δ. Consider a node v which is reinserted in phase t. There must be a most recent phase $t' \leqslant t$ when v was deleted. Consider any shortest path $A(v) = (s, \dots, v', \dots v)$ where v' is the first unsettled node on $A(v)$ immediately before phase t'. Hence, v' already has its final distance. Furthermore, since v' is at least as close to s as v both v' and v will be deleted in phase t' – v' is settled then whereas v is reinserted later in step t. In particular, $v' \neq v$. Note that the part $A'(v', v) = (v', \dots, v)$ of $A(v)$ is a shortest path from v' to v. Since both v and v' are deleted by phase t', $A'(v', v)$ is also a Δ-path, i.e., $\langle v', v\rangle \in C_\Delta$. Since v' becomes settled, the reinsertion of v in phase t can be uniquely mapped to $\langle v', v\rangle$.

Similarly, each re-relaxation can be uniquely identified by a reinsertion plus a light edge e. □

The number of phases needed can be bounded as a function of the step width Δ, the maximum path weight L and the maximum Δ-size l_Δ:

Lemma 2. *For any step width Δ, the number of phases is bounded by $\frac{L}{\Delta} l_\Delta$.*

Proof. The algorithm repeatedly removes all nodes from the current bucket and relaxes outgoing edges until the bucket finally remains empty. Then it proceeds with the next bucket. For maximum path weight L it has to traverse L/Δ buckets. Hence, in order to prove the lemma it suffices to prove that at most l_Δ phases are required until any bucket remains empty. Consider any node $v \neq s$ with $\mathrm{dist}(v) \in B[i]$ and let $(s, \ldots, v_0, v_1, \ldots, v_l = v)$ denote a shortest path from s to v such that v_1 through v_l are the only nodes settled by iterations of $B[i]$ and such that l is minimal, i.e., among all the shortest paths to v choose one with a minimal number of nodes in $B[i]$. By definition, before the first node removal from bucket $B[i]$, v_0 must have been settled in a previous bucket. Hence, the edge (v_0, v_1) has been relaxed with the proper distance value for v_0, so that v_1 has already reached its final distance. Therefore, v_1 will be settled in the first phase for bucket $B[i]$. Similarly, it can be concluded inductively that in phase j for bucket $B[i]$, $1 \leqslant j \leqslant l$, v_j is settled. Now, l cannot exceed l_Δ since otherwise $\langle v_1, v_l \rangle \in C_\Delta$ would have no connection with less than l_Δ edges, thus contradicting the definition of l_Δ. So, after l_Δ iterations the last node in bucket $B[i]$ must be settled. □

We now have enough information to determine the execution time of sequential Δ-stepping in terms of n, m, Δ, n_Δ, and m_Δ:

Theorem 3. *Sequential Δ-stepping can be implemented to run in time $\mathcal{O}(n + m + L/\Delta + n_\Delta + m_\Delta)$.*

Proof. We charge the operations performed by Δ-stepping to one of n nodes, one of m edges of G, one of the at most L/Δ empty buckets traversed, or to an element in C_Δ or $C_{\Delta+}$. The time bound is established by showing that none of the mentioned objects gets more than a constant charge.

The loop control in lines (a) and (c) can be implemented in constant time by keeping counters of the buckets sizes and their sum. These costs can be charged to nodes since in each iteration at least one node is settled.

Finding the next nonempty bucket in line (b) can be charged to empty buckets each of which is only considered once.

Identifying light requests in line (d) and doing the relaxations in line (f) needs constant work for each re-relaxation and reinserted node if an $\mathcal{O}(m + n)$ time preprocessing phase orders the adjacency lists into sublists for light and heavy edges, respectively. Nodes once deleted from a bucket in line (e) can be remembered by storing them in a list. In order to avoid duplicates in the list we use a boolean array $I[\cdot]$ for the node indices, initialized with

0 and set $I[i] := 1$ when node i is deleted for the first time. By Lemma 1 these operations can be charged to elements in C_Δ and $C_{\Delta+}$.

The work done for relaxing heavy edges in lines (g) and (h) can be charged to nodes and edges since heavy edges are relaxed only once. □

We get a linear time algorithm if $n_\Delta + m_\Delta + L/\Delta = \mathcal{O}(n+m)$. For a parallel algorithm, the goal is to find a Δ which is small enough to preserve this work efficiency yet large enough to yield sufficient parallelism.

A very conservative approach to choose Δ goes back to Dinitz (and, independently, Denardo and Fox) [25,30]. They observe that there are no reinsertions if we choose the step width Δ just below the minimum edge weight and we have $l_\Delta = 1$ in this case. Another way to ensure a small l_Δ ($l_\Delta \leqslant 2$) it to add a "shortcut" edge (v, w) whenever $\langle v, w\rangle \in C_\Delta$. We will discuss this approach in Section 6.

3.2. Random edge weights

This section is devoted to proving that for graphs with maximum node degree d and random edge weights uniformly distributed in $[0, 1]$, $\Delta = \Theta(1/d)$ is a good step width in the sense that Δ is small enough to limit the overhead of reinsertions and re-relaxations yet large enough to yield sufficient parallelism.

The results also hold for random graphs from $D(n, \bar{d}/n)$ with random edge weights if we replace the maximum degree d by the average node degree $\bar{d}$.

Theorem 4. *For random edge weights, a $\Theta(1/d)$-stepping scheme without shortcut edges performs $\mathcal{O}(n + m + dLl_\Delta)$ sequential work divided between $\mathcal{O}(dLl_\Delta)$ phases whp where $l_\Delta = \mathcal{O}(\log n/\log\log n)$.*

Proof. By Lemma 2 the number of phases is bounded by $\frac{L}{\Delta}l_\Delta = \mathcal{O}(dLl_\Delta)$ for $\Delta = \Theta(1/d)$. Subsequently, we show in Lemma 5 that long paths with small total path weight are unlikely.[4] This observation is then used in Lemma 6 in order to prove that $l_\Delta = \mathcal{O}(\log n/\log\log n)$ is a bound on the maximum size of a Δ-path whp.

To bound the amount of work needed, we argue as in the proof of Theorem 3 and charge the operations performed to nodes, edges, and buckets. Using Lemma 1 it remains to bound n_Δ and m_Δ. Lemma 7 shows that taking $\Delta = \mathcal{O}(1/d)$ implies $\mathbf{E}[n_\Delta] = \mathcal{O}(n)$ and $\mathbf{E}[m_\Delta] = \mathcal{O}(n)$. Finally, Lemma 9 states that the same bounds also hold with high probability. □

Lemma 5. *Consider independent random edge weights uniformly distributed in* $[0, 1]$ *and a given path of l nonrepeated edges. The probability that the total path weight is at most $\Delta \leqslant 1$ is given by $\Delta^l/l!$.*

Proof. Let X_i denote the weight of the ith 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 $\mathbf{P}[\sum_{i=1}^{l} X_i \leqslant \Delta] = \Delta^l/l!$ for

[4] This is the only part in our analysis that would have to be adapted for other than uniform weight distributions.

$\Delta \leqslant 1$: if $l = 1$ then due to the uniform distribution the probability that a single edge weight is at most $\Delta \leqslant 1$ is given by Δ itself: $\mathbf{P}[X_1 \leqslant \Delta] = \Delta$. Now we assume that $\mathbf{P}[\sum_{i=1}^{l} X_i \leqslant \Delta] = \Delta^l/l!$ for $\Delta \leqslant 1$ is true for some $l \geqslant 1$. In order to prove the result for a path of $l+1$ edges, we split the path into a first part of l edges and a second part of one edge. For a total path weight at most Δ we have to consider all combinations for $0 \leqslant z \leqslant \Delta \leqslant 1$ such that the first part of l edges has weight at most $\Delta - z$ and the second part (one edge) has weight z. Thus,

$$\mathbf{P}\left[\sum_{i=1}^{l+1} X_i \leqslant \Delta\right] = \int_0^{\Delta} \mathbf{P}\left[\sum_{i=1}^{l} X_i \leqslant \Delta - x\right] \mathrm{d}x = \int_0^{\Delta} \frac{(\Delta - x)^l}{l!}\,\mathrm{d}x = \frac{\Delta^{l+1}}{(l+1)!}. \qquad \square$$

Lemma 6. *For $\Delta = \mathcal{O}(1/d)$, no Δ-path contains more than $l_\Delta = \mathcal{O}(\log n/\log\log n)$ edges whp.*

Proof. There can be at most d^l paths without edge repetitions of size l leading into a given node v or nd^l such paths overall. (For random graphs, there are at most n^l possible paths without edge repetitions per node with a probability of $(\bar{d}/n)^l$ each.) Using Lemma 5 we can conclude that the probability for the presence of any Δ-path of size l is bounded by $n(d\Delta)^l/l!$ (recall that Δ-paths do not use edges more than once). Therefore

$$\begin{aligned}\mathbf{P}\big[\exists \Delta\text{-path } A\colon |A| \geqslant l_\Delta\big] &\leqslant \sum_{l \geqslant l_\Delta} n\frac{(d\Delta)^l}{l!} \leqslant n\frac{(d\Delta)^{l_\Delta}}{l_\Delta!}\sum_{l\geqslant 0}\frac{(d\Delta)^l}{l!} n\frac{(d\Delta)^{l_\Delta}}{l_\Delta!}\mathrm{e}^{d\Delta} \\ &= \mathcal{O}(n)\frac{(d\Delta)^{l_\Delta}}{l_\Delta!} \leqslant \left(\frac{\mathrm{e}d\Delta}{l_\Delta}\right)^{l_\Delta} \cdot \mathcal{O}(n) \quad \text{since } k! \geqslant (k/\mathrm{e})^k.\end{aligned}$$

For $\Delta = \mathcal{O}(1/d)$ and $l_\Delta = \mathcal{O}(\log n/\log\log n)$ the above probability is polynomially small. $\square$

Lemma 7. *For $\Delta = \mathcal{O}(1/d)$, $\mathbf{E}[n_\Delta] = \mathcal{O}(n)$ and $\mathbf{E}[m_\Delta] = \mathcal{O}(n)$.*

Proof. The total number of Δ-paths in the graph is clearly an upper bound on $|C_\Delta| = n_\Delta$, the number of node pairs that are connected by some Δ-path. Counting Δ-paths and their probabilities as in the proof of Lemma 6, we see that the expected number of Δ-paths is bounded by

$$\sum_{l\geqslant 1} n\frac{d^l\Delta^l}{l!} = n\big(\mathrm{e}^{d\Delta} - 1\big) = \mathcal{O}(n)$$

for $\Delta = \mathcal{O}(1/d)$. Thus, $\mathbf{E}[n_\Delta] = \mathcal{O}(n)$.

Each triple $(u, v, w) \in C_{\Delta+}$ corresponds to a (2Δ)-path $(u, \ldots, v, w)$. The expected number of (2Δ)-paths is given by

$$\sum_{l\geqslant 1} n\frac{d^l(2\Delta)^l}{l!} = n\big(\mathrm{e}^{2d\Delta} - 1\big) = \mathcal{O}(n)$$

for $\Delta = \mathcal{O}(1/d)$. Hence, $\mathbf{E}[m_\Delta] = \mathcal{O}(n)$. $\square$

In order to show that the expected bounds also hold with high probability, we first bound the number of nodes reached from a given node via a Δ-path:

Lemma 8. *For $\Delta = \mathcal{O}(1/d)$ and any node v,*

$$\left|\left\{u\colon \langle v,u\rangle \in C_\Delta \vee \langle u,v\rangle \in C_\Delta\right\}\right| \leqslant n^{\mathcal{O}(1/\log\log n)}$$

with high probability.

Proof. Consider the nodes reached from v via a Δ-path. We argue that the number of nodes reachable by connections in C_Δ of length l can be bounded by the offspring in the lth generation of the following branching process ([9,51] provide an introduction to branching processes): An individual (a node) v has its offspring defined by the number Y_v of light edges leaving it. Let Z_l denote the total offspring after l generations, i.e., $Z_1 = Y_v$. The *branching factor* of a branching process is given by $\rho = \mathbf{E}[Z_1]$. Furthermore, for branching processes with identical and independent probabilities for the generation of new nodes, $\mathbf{E}[Z_l] = \rho^l$. Additionally, it is shown in [8] (as cited in [60, Theorem 1]) that $Z_l = \mathcal{O}(\rho^l \log n)$ whp.

As long as new edges are encountered when following paths out of v, the offspring of the branching process is an exact model for the number of paths leaving v which traverse only light edges. After a node has been reached from multiple paths, the events on those paths are no longer independent. However, all but one of the multiple paths produce only duplicate entries into C_Δ. The additional paths can therefore be discarded. All remaining events are independent.

For independent edge weights uniformly distributed in $[0,1]$, we have $\mathbf{E}[Y_v] \leqslant d\Delta$ ($\mathbf{E}[Y_v] = \bar{d}\Delta$ for random graphs), and by the discussion above, $\mathbf{E}[Z_l] \leqslant (d\Delta)^l$ and $Z_l = \mathcal{O}((d\Delta)^l \log n)$ whp. In order to asymptotically bound the sum $\sum_{i\leqslant l} Z_i$ for $d\Delta > 1$ we can concentrate on term Z_l since a growing exponential sum is dominated by its last summand. For $\Delta = \mathcal{O}(1/d)$ we can substitute $l = l_\Delta = \mathcal{O}(\log n/\log\log n)$ and $d\Delta = \mathcal{O}(1)$ yielding the desired bound. □

Although the bound from Lemma 8 is quite rough (in particular for random graphs and also if $d\Delta < 1$) it suffices to prove that n_Δ and m_Δ are rather sharply concentrated around their means:

Lemma 9. *For $\Delta = \mathcal{O}(1/d)$, $n_\Delta \leqslant \mathbf{E}[n_\Delta] + \mathcal{O}(n)$ and $m_\Delta \leqslant \mathbf{E}[m_\Delta] + \mathcal{O}(n)$ whp.*

Proof. We only give the argument for n_Δ; the proof for m_Δ can be made in a largely analogous way. The proof can also be adapted to random graphs with average degree $\bar{d}$.

Let Q be the event that at most $a_\beta \log n$ light edges leave any node in G and that the number of Δ-paths entering or leaving any node is at most $n^{a_\beta/\log\log n}$ for some constant a_β; let a_β depend on another positive constant β which we are free to choose. Using Chernoff bounds and Lemma 8 we can see that Q holds with probability at least $1 - n^{-\beta}$ for an appropriate constant a_β and sufficiently large n.

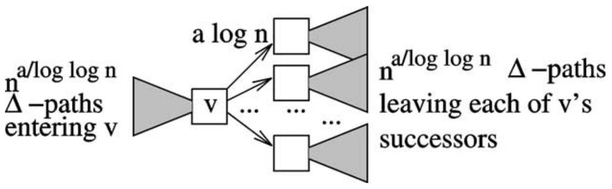

a log n
n^a/log log n Δ –paths entering v
v
n^a/log log n Δ –paths leaving each of v's successors

Fig. 3. Δ-paths possibly affected by changing the weights of edges out of node v.

For each $v \in V$, define the random variable $X_v := \{(v, w, c(v, w)) \colon (v, w) \in E)\}$, i.e., X_v describes the edges leaving node v including their weight.[5] Now, n_Δ can be viewed as a function of the independent random variables X_v. In the following we will investigate how much n_Δ can change if a single X_v is altered.

If Q holds, then changing X_v has a limited effect on n_Δ: by the definition of Q, there can be at most $n^{a_\beta/\log\log n}$ Δ-paths entering a node; the number of edges leaving v is at most $a_\beta \log n$ and from each node reached over these edges there are at most $n^{a_\beta/\log\log n}$ outgoing Δ-paths. Multiplying these three quantities gives an upper bound on the number of Δ-paths affected if the weights of the edges leaving v are changed. Figure 3 depicts this situation. Hence, $(n^{a_\beta/\log\log n})^2 a_\beta \log n = n^{\mathcal{O}(1/\log\log n)}$ is an upper bound on the impact on n_Δ if Q holds. On the other hand, if Q does not hold, then altering X_v may change n_Δ by at most n^2 (since $n_\Delta \leqslant n^2$). Therefore, the *average* impact on n_Δ when changing X_v is bounded by

$$c_v \leqslant \left(1 - n^{-\beta}\right) n^{\mathcal{O}(1/\log\log n)} + n^{2-\beta} = n^{\mathcal{O}(1/\log\log n)} \quad \text{for } \beta \geqslant 2.$$

This can be used together with the *method of bounded martingale differences* [32,64] in order to show

$$\mathbf{P}\left[n_\Delta \leqslant \mathbf{E}[n_\Delta] + t\right] \geqslant \mathbf{P}[Q] - \exp\left(\frac{-t^2}{2\sum_{v=1}^{n} c_v^2}\right).$$

Substituting $2\sum_{v=1}^{n} c_v^2 \leqslant 2n(n^{\mathcal{O}(1/\log\log n)})^2 = nn^{\mathcal{O}(1/\log\log n)}$ and $t = n$ we get

$$\begin{aligned}\mathbf{P}\left[n_\Delta \leqslant \mathbf{E}[n_\Delta] + n\right] &\geqslant 1 - n^{-\beta} - \mathrm{e}^{-\frac{n^2}{\mathcal{O}(n) n^{\mathcal{O}(1/\log\log n)}}} \geqslant 1 - n^{-\beta} - \mathrm{e}^{-n^{1-\mathcal{O}(1/\log\log n)}} \\ &\geqslant 1 - 2n^{-\beta}\end{aligned}$$

for $\beta \geqslant 2$. □

3.3. Random graphs

So far, the analysis treated the maximum shortest path weight, L, as a parameter. Clearly, there are graphs—even with random edge weights—where $L/\Delta = \Omega(n)$ so that it makes no sense to relax edges from different nodes in parallel. But this is quite atypical. In particular, for random graphs with average degree $\bar{d}$ we now give results which

[5] This way to define a sequence of random variables is related to the *edge exposure martingale* as described in [72, Exercise 4.10].

show that L is usually rather small. Substituting this result into Theorem 4 we see that $\mathcal{O}(l_\Delta \log n) = \mathcal{O}(\log^2 n/\log\log n)$ phases of a $\Theta(1/\bar{d})$-stepping algorithm suffice to solve the SSSP. If we have introduced shortcut edges this reduces to $\Theta(\log n)$ phases.

In order to bound L we can use a result on the *diameter* of sparse random graphs: let 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 the $\max_{u,v\in V}\{\text{minsize}(u, v), 1\}$.

Lemma 10 [79]. *The diameter of a random directed graph from $D(n, \bar{d}/n)$ is bounded by $\mathcal{O}(\log n)$ whp whenever $\bar{d} < c^*$ or $\bar{d} > 2 \cdot c^*$, where $c^* > 1$ is some constant.*[6]

Since each edge has weight at most one, $L = \mathcal{O}(\log n)$ 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:

Theorem 11. *Let G be a random graph from $D(n, \bar{d}/n)$ with diameter $\mathcal{O}(\log n)$. For an arbitrary source node s and independent random edge weights uniformly drawn from $[0, 1]$, the maximum shortest path weight in G, $L = \max\{\text{dist}(s, v)\colon \text{dist}(s, v) < \infty\}$ is bounded from above by $\mathcal{O}((1/\bar{d}) \cdot \log n)$ whp.*

This is a well-known result for large expected degrees, $\bar{d} \geqslant a \cdot \log n$, where a is some constant so that all nodes of G are reachable from the source node whp [39,52]. In the following we will deal with the case $\bar{d} < a \cdot \log n$.

Proof. The set of nodes reachable from s, denoted by $R(s)$ is either *small*, $|R(s)| = \mathcal{O}((1/\bar{d}) \cdot \log n)$ whp, or *giant*, $|R(s)| = \Theta(n)$ whp [59]. Hence, if $R(s)$ is small, then Theorem 11 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, from now on we 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 connected 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 that all nodes in $R(s)$ can 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.

If $\bar{d} = \mathcal{O}(1)$, the result is an immediate consequence of the assumption that the graph has logarithmic diameter. Otherwise, we can assume that $3 \cdot c^* < \bar{d} < a \cdot \log n$. 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 $3 \cdot c^*/n$, and it has a giant strong component C' of size $\alpha \cdot n$ for some constant α (e.g., [5,59]).

[6] The value of c^* depends on the constant used in the definition of "whp." We do not dwell on the details since this gap seems to be an artifact of the analysis [79].

By Lemma 10 the diameter of the subgraph induced by the nodes in C' and its adjacent edges in E' is at most $\mathcal{O}(\log n)$ whp. 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 $\mathcal{O}((1/\bar{d}) \cdot \log n)$ whp.

We now 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 [59] and [5, Section 10.5] starting in the source node s and using the edges of 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 $t' = (c \cdot \log n)/\bar{d}$ iterations for some appropriate constant $c > 1$. If any of those nodes is in C' then we are done. Otherwise, $Y_1, \ldots, Y_{t'} \geqslant 1$ since $R(s)$ is giant by assumption. Let $\mathrm{BIN}[n, p]$ 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 $Y_{t-1} \geqslant 1$, Y_t is distributed as follows [5,59]: $\mathrm{BIN}[n-1, 1-(1-\bar{d}/n)^t] + 1 - t$, i.e., Y_t is sharply concentrated around

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

We use the inequality

$$\begin{aligned}(1-x)^y &< 1 - x \cdot y + (x \cdot y)^2/2 \quad \text{if } x \cdot y < 1,\\ &\leqslant 1 - \frac{2 \cdot c - 1}{2 \cdot c} \cdot x \cdot y \quad \text{if } c \cdot x \cdot y < 1,\ c > 1,\end{aligned}$$

in order to derive a lower bound on $f(t)$. Indeed we have $c \cdot \bar{d} \cdot t'/n = \Theta(\log n)/n < 1$, hence

$$\begin{aligned}f(t') &\geqslant (n-1) \cdot \frac{2 \cdot c - 1}{2 \cdot c} \cdot \bar{d} \cdot t'/n + 1 - t'\\ &= \frac{c \cdot (n-1)}{\bar{d} \cdot n} \cdot \left(\bar{d} - 1 - \frac{1}{2 \cdot c}\right) \cdot \log n - 1.\end{aligned}$$

Since $\bar{d} > 3 \cdot c^* > 3$ there is a choice for the constants c and $c'(c)$ such that $f(t') \geqslant c' \cdot \log n$ and $Y_{t'} \geqslant (c'/2) \cdot \log n$ whp by Chernoff bounds.

So, there are $Y_{t'} \geqslant (c'/2) \cdot \log n$ active nodes whp whose outgoing edges have not yet been inspected in the search procedure from s. Now consider an arbitrary active node $v \notin C'$ and an arbitrary node $w \in C'$.

We only know that a possible edge (v, w) is not tiny in case there is also a tiny edge (w', v) from some node $w' \in C'$ (otherwise v would be in C'). We will find a nontiny edge (v, w) with probability $(\bar{d} - c^*)/n$, and for all outgoing edges under consideration these probabilities are independent. Since $|C'| = \Omega(n)$ and $Y_{t'} = \Omega(\log n)$, the probability that at least one such edge (v', w) exists is at least $1 - (1 - (\bar{d} - c^*)/n)^{c' \cdot n \log n} \geqslant 1 - n^{-\beta}$ for any constant $\beta > 0$, an appropriate choice of $c'(\beta)$ and sufficiently large n. 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 one this path have weight at most one, the desired bound on the path weight from s to C' follows immediately.

The same technique is applied to show that any node v in $R(s)$ which does not belong to C' can be reached from C' via a path of $\mathcal{O}(1 + (1/\bar{d}) \cdot \log n)$ edges in G 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 fails with probability at most $n^{-\beta}$. Hence, using Boole's inequality all nodes can be reached within the claimed bounds with probability $1 - n \cdot n^{-\beta} = 1 - n^{-(\beta-1)}$ which is high probability since we are free to choose β. This completes the proof of Theorem 11 for $3 \cdot c^* < \bar{d} < a \cdot \log n$. $\square$

4. A simple parallelization

Our parallel SSSP algorithms are based on sequential Δ-stepping: we stick to the sequential search for the next nonempty bucket but perform the operations for a phase of the current bucket in parallel. In order to do so we need to devise strategies how the work for a phase is subdivided among the processors.

We start with a straightforward algorithm based on a *random* distribution of the nodes to PUs (Section 4.1): the bucket structure is distributed over the PUs. In a phase, each PU does the work for the nodes randomly assigned to its own structure. This strategy is already quite good for sparse graphs where the maximum node degree is not much larger than the average node degree, e.g., road-maps.

The algorithm also serves as a basis for the distributed memory algorithms discussed in Section 8 and the implementation described in Section 9. In Section 4.2 we show how a small modification can yield better results for random graphs. Only for nonrandom graphs with large node degree variations, the random node allocation used in this section is not sufficient to guarantee good load-balancing for many processors. We will deal with this problem in Section 5.

4.1. Graphs with low maximum degree

Theorem 12. *The single source shortest path problem for directed graphs with n nodes, m edges, maximum in-degree and out-degree d, maximum path weight L, maximum Δ-size l_Δ, n_Δ, and m_Δ defined as in Section* 1.2 *can be solved on a CRCW PRAM in time $\mathcal{O}(\frac{L}{\Delta} d l_\Delta \log n)$ and work $\mathcal{O}(n + m + n_\Delta + m_\Delta + \frac{L}{\Delta} d l_\Delta \log n)$ whp.*

To implement the operations from Fig. 1 in a distributed way, we first have to explain the layout of the data structures. The nodes are distributed randomly over the PUs but each node has its adjacency list locally accessible. These adjacency lists are partitioned into heavy and light edges. The buckets $B[i]$ of the bucket queue and the set R of deleted nodes are split among the PUs following the distribution of the nodes, i.e., if PU j stores the nodes V_j then it also stores $B[i] \cap V_j$ and $R \cap V_j$. Similarly, the relaxation requests Req are generated locally for each $B[i] \cap V_j$. Then the requests are redistributed according to their target nodes and the PUs responsible for them. All required operations can be implemented using fairly standard parallelization techniques. The difficult part is to show that the work is sufficiently evenly distributed among the PUs.

To prove Theorem 12, we explain in a step-by-step manner, how the algorithm from Fig. 1 can be parallelized using $p = \lceil (n + m + n_\Delta + m_\Delta)/((L/\Delta) d l_\Delta \log n) \rceil < n$ PUs.

Preparations: The nodes are assigned to random PUs by generating an array "ind" of random PU indices. Entry ind$[v]$ gives the PU responsible for node v. The adjacency lists are reorganized into separate arrays of heavy and light edges for each node. Each of the p PUs maintains its own bucket structure and stores there the queued nodes it is responsible for. These operations can be done in time $\mathcal{O}((n+m)/p+\log(n))$ and take $\mathcal{O}(n+m)$ space.

Loop control: Skipping k empty buckets can be done in time $\mathcal{O}(k+\log n)$—each PU can traverse locally empty buckets 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.

Maintaining R: The set of removed nodes R can be represented as a simple list per PU. Inserting a node several times can be avoided by storing a flag with each node that is set when the node is inserted for the first time.

Generating requests: Let $k = |B[i]|$ and let K denote the total number of edges emanating from nodes in $B[i]$. Since the nodes have been randomly assigned, the k adjacency lists to be scanned can be viewed as jobs of size at most d which have been randomly assigned to the PUs. This situation can be analyzed using the following useful result based on weighted Chernoff bounds [4, Lemma 2]:

Lemma 13. *Consider any number of subproblems of size in $(0, d]$. Let K denote the sum of all subproblem sizes. If the subproblems are randomly allocated to p PUs, then the maximum load of any PU will be bounded by $\mathcal{O}(K/p + d\log(n + p))$ whp.*[7]

Assigning Requests to PUs: Let $\mathtt{Req}_i$ be the set of requests generated by PU i. Any request $(w, x) \in \mathtt{Req}_i$ must now be transferred to the bucket structure associated with PU $P_{\mathrm{ind}(w)}$. Using Lemma 13, it can be seen that due to the random indexing, each PU receives $\mathcal{O}(|\bigcup \mathtt{Req}_j|/p + d\log n)$ requests whp. The value of $|\bigcup \mathtt{Req}_j|$ 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" [69] where PU i tries to write $(w, x) \in \mathtt{Req}_i$ to a random position of the target buffer of PU ind(w) (see Fig. 4). 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, $\mathcal{O}(|\bigcup \mathtt{Req}_j|/p + d\log n)$ 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.

[7] The seemingly unrelated parameter n comes into play since we based our definition of "whp" on it. Also note, that using a more detailed calculation, a bound $K(1 + o(1))/p$ can be proven for $K = \omega(pd\log(n + p))$.

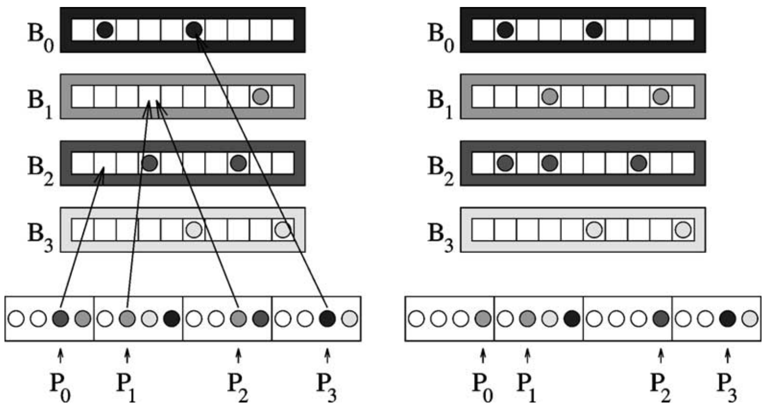


Fig. 4. Placing requests into buffers by randomized dart-throwing. Processor P_0 succeeds in placing its currently treated request in B_2 and can turn to its next request. P_1 and P_2 contend for a free slot in B_1. P_2 wins. P_1 has to probe another position in B_1. P_3 fails because the chosen position is already occupied.

Performing relaxations: Each PU scans its request buffer and sequentially performs the relaxations assigned to it. Since no other PUs work on its nodes the relaxations will be atomic.

To summarize, control overhead accounts for at most $\mathcal{O}(\log n)$ time per phase; the load imbalance accounts for at most $\mathcal{O}(d \log n)$ time per phase. Summing over all phases and by using Lemma 2, we get an execution time of $\mathcal{O}((n + m + n_\Delta + m_\Delta)/p + \frac{L}{\Delta} d l_\Delta \log n)$.

4.2. *Improvements for random graphs*

We now outline how explicit load-balancing for the generation of requests yields a more efficient parallelization for the case of random graphs. Although this result will be superseded in subsequent sections, it introduces techniques needed later and demonstrates that simple algorithms can work quite well for "benign" inputs:

Theorem 14. *Consider random graphs from $D(n, \bar{d}/n)$, with arbitrary nonnegative edge weights, maximum path weight L, and maximum Δ-size l_Δ, where Δ is chosen sufficiently small in order to ensure $n_\Delta + m_\Delta = \mathcal{O}(n)$. In that case, SSSP can be solved on a CRCW PRAM in time $\mathcal{O}(\bar{d} + (L/\Delta + \log n) l_\Delta \log n)$ and work $\mathcal{O}(\bar{d}n + (L/\Delta + \log n) l_\Delta \log n)$ whp.*

If the edge weights are also random, then all but the $c \log n$ smallest edges per node can be ignored without changing the shortest paths for some constant c [39,52]. Thus, after an initial pruning step we may assume $\bar{d} = \mathcal{O}(\log n)$ for random edge weights. Together with Lemmas 6 and 7, and Theorem 11 we get:

Corollary 15. *The SSSP on random graphs from $D(n, \bar{d}/n)$ with random edge weights can be solved using $\Theta(1/\bar{d})$-stepping in time $\mathcal{O}(\log^3 n / \log\log n)$ and $\mathcal{O}(\bar{d}n)$ work on a CRCW PRAM whp.*

```
Function findRequests($V'$, kind : {light, heavy}) : set of Request
    $k := \sum_{v \in V'} \text{outdegree}_{\text{kind}}(v)$
    assign a unique number $1 \leqslant j \leqslant k$                                  (* Prefix sums *)
        to each edge of type ‘kind’ $(v_j, w_j)$ leaving a node in $V'$
    for $1 \leqslant j \leqslant k$ dopar  $\text{Req}[j] := (w_j, \text{tent}(v_j) + c(v_j, w_j))$
```

Fig. 5. Finding requests in an explicitly load balanced way.

In the following we show how Theorem 14 is obtained. From the previous parallelization we maintain the idea that there are at most $p \leqslant n$ separate bucket structures, where each processor is in charge of its own structure. The only 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 i exclusively by PU 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; see Fig. 5 for the pseudo code. 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.

Now we exploit the structure of random graphs to show that for relaxing requests there is no load balancing problem. For heavy edges this is easy to see since they are relaxed only once: for random graphs their targets are independently distributed and therefore by Chernoff bounds, each bucket structure (PU) receives only $\mathcal{O}(|\text{Req}|/p + \log n)$ requests whp. Since there are at most $\mathcal{O}(\frac{L}{\Delta} l_\Delta)$ phases and at most $\mathcal{O}(\bar{d}n)$ relaxations of heavy edges in total, this accounts for $\mathcal{O}(\bar{d}n/p + \frac{L}{\Delta} l_\Delta \log n)$ time.

Although assigning light requests works as for heavy requests, we get a technical problem here. The targets of re-relaxed edges are no longer independent. However, targets are still independent when edges are relaxed for the first time. Let

$$K_i := \bigl|\bigl\{(v, w) \in E\colon \text{dist}(v) \in \bigl[i\Delta, (i+1)\Delta\bigr) \wedge c\bigl((v, w)\bigr) \leqslant \Delta\bigr\}\bigr|,$$

i.e., the number of light edges ever relaxed in bucket i not counting re-relaxations. Then, by Chernoff bounds, no node receives more than $\mathcal{O}(\lceil K_i \log n/n \rceil)$ requests in any phase for bucket i whp. Let K'_{ij} denote the number of requests sent in the jth phase for bucket i. Since nodes are placed independently of the computation, we can use Lemma 13 again, to see that no PU receives more than $\mathcal{O}(K'_{ij}/p + \lceil K_i \log n/n \rceil \log n)$ requests in phase j for bucket i whp. For the request contention K^* summed over all phases we use $\sum_{ij} K'_{ij} \leqslant n + n_\Delta + m_\Delta$, $\sum_{ij} K_i = \mathcal{O}(l_\Delta(n + n_\Delta + m_\Delta))$, and $n_\Delta + m_\Delta = \mathcal{O}(n)$ whp by Lemmas 7 and 9:

$$\begin{aligned}
K^* &= \mathcal{O}\biggl(\sum_{ij} \frac{K'_{ij}}{p} + \biggl\lceil \frac{K_i \log n}{n} \biggr\rceil \log n\biggr) \\
&= \mathcal{O}\biggl(\frac{n + n_\Delta + m_\Delta}{p} + \sum_{ij}\biggl(1 + \frac{K_i \log n}{n}\biggr) \log n\biggr) \\
&= \mathcal{O}\biggl(\frac{n}{p} + \biggl(\frac{L l_\Delta}{\Delta} + \frac{l_\Delta(n + n_\Delta + m_\Delta) \cdot \log n}{n}\biggr) \log n\biggr)
\end{aligned}$$

$$= \mathcal{O}\left(\frac{n}{p} + \left(\frac{L}{\Delta} + \log n\right) l_\Delta \log n\right).$$

Hence, the total time consumption is given by $\mathcal{O}(\bar{d}n/p + (L/\Delta + \log n)l_\Delta \log n)$. The work and time bounds for the PRAM follow from choosing $p = \min\{n, \lceil \bar{d}n/((L/\Delta + \log n)l_\Delta \log n)\rceil\}$.

5. Faster parallel bucket traversal

The parallelization of Section 4.1 is well-suited for sparse graphs with low node degrees, e.g., for road-maps where the maximum node degree d is usually bounded by a small constant. However, the performance may deteriorate on graphs with high degree nodes because of poor load balancing for generating and performing requests. In Section 4.2 we already demonstrated improvements for *random* graphs by explicitly load balancing the generation of requests. Now we show how to gain a factor of $\Theta(d)$ for *arbitrary* graphs by also load balancing the request executions.

In short, the additional balancing works as follows: after their generation, all requests are rearranged according to their target nodes. This rearrangement turns out to be a bottleneck of the parallelization. The algorithmically most interesting part here is that rearranging does not require sorting but only *semisorting* which can be performed significantly faster. The remaining operations are fairly simple. The strictest request within each group is selected. Only the selected requests are finally executed. Thus, each target node receives at most one request. These selected requests will be load balanced over the PUs whp due to the random assignment of nodes to PUs.

This measure is also a prerequisite for removing the factor l_Δ in Section 6—now we can insert the shortcuts from Section 3.1 without second thoughts about increasing the maximum degree of the graph.

Theorem 16. *The single source shortest path problem for directed graphs with n nodes, m edges, maximum path weight L, maximum Δ-size l_Δ, and n_Δ and m_Δ defined as in Section* 3.1 *can be solved on a CRCW PRAM in time $\mathcal{O}(\frac{L}{\Delta} l_\Delta \log n)$ and work $\mathcal{O}(n + m + n_\Delta + m_\Delta + \frac{L}{\Delta} l_\Delta \log n)$ whp.*

Proof (outline). We concentrate on load balancing the execution of requests. The remainder of the parallelization works as in Section 4. 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.

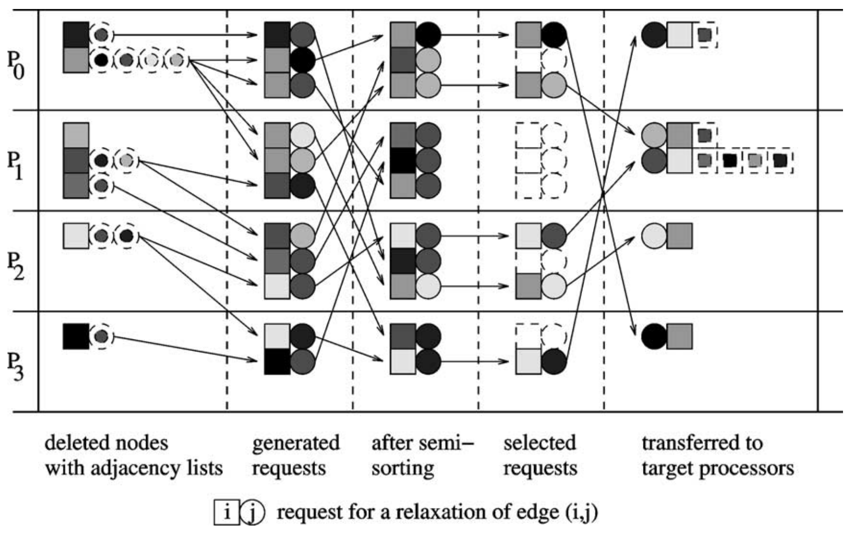
Instead of the brute-force randomized dart-throwing as in Section 4.1, we use an explicit load balancing which groups different requests for the same target and only executing the strictest relaxation. On CRCW PRAMs, grouping can be done efficiently using the semi-sorting routine explained in Lemma 17. Then we can use prefix sums to schedule $\lfloor p|\mathrm{Req}(w)|/|\mathrm{Req}|\rfloor$ PUs for blocks of size at least $|\mathrm{Req}|/p$ and to assign smaller groups with a total of up to $|\mathrm{Req}|/p$ requests to individual PUs. The PUs concerned with a group collectively find a request with minimum distance in time $\mathcal{O}(|\mathrm{Req}|/p + \log p)$ and then relax it in constant time. Summing over all phases yields the desired bound. □

Figure 6 outlines a fast parallelization of the procedure ‘relaxRequest’ and Fig. 7 provides an illustration.

Lemma 17. Semi-sorting K *records with integer keys, i.e., permuting them into an array of size* K *such that all records with equal key form a consecutive block, can be performed in time* $\mathcal{O}(K/p + \log K)$ *on a CRCW-PRAM whp.*[8]

```
**Procedure** relaxRequests(Req)
    semi-sort ‘Req’ using $v$ as key for request $(v, x)$
    **foreach** block $\mathrm{Req}(w) = \{(w, x) \in \mathrm{Req}\}$ of requests for node $w$ **dopar**
        schedule $\max\{1, \lfloor p|\mathrm{Req}(w)|/|\mathrm{Req}|\rfloor\}$ PUs
           (* possibly several small blocks per PU (prefix sums) *)
        $y := \min\{x\colon (w, x) \in \mathrm{Req}(w)\}$
        relax($w$, $y$)
```

Fig. 6. Load balanced edge relaxation using semi-sorting.



P0
P1
P2
P3
deleted nodes with adjacency lists
generated requests
after semi–sorting
selected requests
transferred to target processors
i j request for a relaxation of edge (i,j)

Fig. 7. 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.

[8] For this lemma whp is defined as a function of K.

Proof. First find a perfect hash function $h : V \to 1..cK$ for an appropriate constant c. Using the algorithm of Bast and Hagerup [10] this can be done in time $\mathcal{O}(K/p + \log p)$ (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 [76] to sort by the hash values in time $\mathcal{O}(K/p + \log K)$ whp. □

6. Finding shortcuts

In this section we explain how the multiplicative factor l_Δ in Theorem 16 can be moved to an additive term which is independent of the maximum path length L by explicitly inserting the shortcut edges between node pairs in C_Δ as mentioned in Section 3.1. Shortcuts 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. Furthermore, a lot of parallelism is available. The only additional complication is that we have to make sure that only simple paths are explored. This can be achieved by maintaining a hash table of connections already found and keeping only the shortest one found.

Theorem 18. *Let n'_Δ (m'_Δ) denote the number of* simple[9] *Δ-paths (plus a light edge) and let l'_Δ denote the number of edges in the longest simple Δ-path. There is an algorithm which inserts an edge (u, v) with weight $c(u, v) = \mathrm{dist}(u, v)$ for each shortest path $(u, \dots, v)$ with $\mathrm{dist}(u, v) \leqslant \Delta$ using $\mathcal{O}(l'_\Delta \log n)$ time and $\mathcal{O}(n + m + n'_\Delta + m'_\Delta)$ work on a CRCW-PRAM whp.*

Note that n_Δ counts node pairs whereas n'_Δ counts simple paths. On some graphs we may have $n'_\Delta \gg n_\Delta$. However, often the two ways of counting paths make no big difference. In particular, our bounds for random edge weights from Section 3.2 (except the sharp concentration results) also apply to the primed quantities. We get the following improved time bounds by adding the bounds from Theorem 18 and the time bounds from the previous sections taking into account that now $l_\Delta = \mathcal{O}(1)$.

Corollary 19. *The single source shortest path problem for directed graphs with n nodes, m edges, maximum path weight L and n'_Δ, m'_Δ, l'_Δ as defined in Theorem* 18 *can be solved on a CRCW PRAM in time $\mathcal{O}((l'_\Delta + L/\Delta)\log n)$ and work $\mathcal{O}(n + m + n'_\Delta + m'_\Delta)$ whp. For random edge weights, maximum in-degree and out-degree at most d and $\Delta = \Theta(1/d)$ we get expected time $\mathcal{O}(dL \log n)$ and expected work $\mathcal{O}(n + m)$. For random graphs with random edge weights and edge probability $\bar{d}/n$, $\Theta(1/\bar{d})$-stepping with shortcuts works in expected time $\mathcal{O}(\log^2 n)$ and expected work $\mathcal{O}(n + m)$.*

Assume for now that shortcuts are already present. Figure 8 describes a variant of Δ-stepping which exploits their existence. Classifying edges as light or heavy is no longer

[9] A simple Δ-path is a Δ-path on which all nodes are pairwise distinct.

```
$E := E \cup$ findShortCuts($\Delta$)
**foreach** $v \in V$ **do** tent($v$) := $\infty$                          (* Unreached *)
relax($s$, 0); $i := 0$                                       (* Source node at distance 0 *)
**while** $\neg$isEmpty(B) **do**                                   (* Some queued nodes left *)
    $i := \min\{j > i\colon B[j] \neq \emptyset\}$                        (* Smallest nonempty bucket *)
    **foreach** $(v, w) \in B[i]$ **do**
        **if** tent($v$) + $c(v, w) \leqslant (i + 1)\Delta$ **then**
            relax($w$, tent($v$) + $c(v, w)$)                          (* Intra-bucket edge *)
    **foreach** $(v, w) \in B[i]$ **do**
        **if** tent($v$) + $c(v, w) > (i + 1)\Delta$ **then**
            relax($w$, tent($v$) + $c(v, w)$)                          (* Extra-bucket edge *)
```

Fig. 8. High level Δ-stepping SSSP algorithm which assumes the presence of shortcut edges.

important for the shortest path search itself. By explicitly treating intra-bucket edges (source and target reside in the same bucket) first, each edge is relaxed at most once: After buckets 0 through $i - 1$ have been emptied, a single relaxation pass through the edges reaching from $B[i]$ into $B[i]$ suffices to settle all nodes now in $B[i]$. After that, $B[i]$ can be emptied by relaxing all edges reaching out of $B[i]$ once.

Figure 9 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 size $\mathcal{O}(m'_\Delta)$ (we can use dynamic hashing if we do not know a good bound for m'_Δ). This table plays a role analogous to that of tent($\cdot$) in the main routine of Δ-stepping. 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 5, this routine can be implemented to run in $\mathcal{O}(l'_\Delta \log n)$ parallel time using $\mathcal{O}(n + m + n'_\Delta + m'_\Delta)$ work: We need l'_Δ iterations each of which takes time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|Q'|)$ whp. The overall work bound holds

```
**Function** findShortcuts($\Delta$) : set of weighted edges
    found : HashArray[$V \times V$]                    (* return $\infty$ for undefined entries *)
    $Q := \{(u, u, 0)\colon u \in V\}$                      (* (start, destination, weight) *)
    $Q'$ : MultiSet
    **while** $Q \neq \emptyset$ **do**
        $Q' := \emptyset$
        **foreach** $(u, v, x) \in Q$ **dopar**
            **foreach** light edge $(v, w) \in E$ **dopar**
                $Q' := Q' \cup \{(u, w, x + c(v, w))\}$
        semi-sort $Q'$ by common start and destination node
        $Q := \{(u, v, x)\colon x = \min\{y\colon (u, v, y) \in Q'\}\}$
        $Q := \{(u, v, x) \in Q\colon x \leqslant \Delta \wedge x < \text{found}[(u, v)]\}$
        **foreach** $(u, v, x) \in Q$ **dopar** found[$(u, v)$] := $x$
    **return** $\{(u, v, x)\colon \text{found}[(u, v)] < \infty\}$
```

Fig. 9. CRCW-PRAM routine for finding shortcut edges.

since for each simple Δ-path $(u, \dots, v)$, $\langle u, v\rangle$ can be a member of Q only once. Hence, $\sum_i |Q| \leqslant n + n'_\Delta$ and $\sum_i |Q'| \leqslant n + m'_\Delta$.

7. Determining $\boldsymbol{\Delta}$

In the case of arbitrary edge weights it is necessary to find a step width Δ which is large enough to allow for sufficient parallelism and small enough to keep the algorithm work-efficient. Although we expect that application specific heuristics can often give us a good guess for Δ relatively easily, for a theoretically satisfying result we would like to be able to find a good Δ systematically. This can be done by starting with a small value of Δ that is certain to be save and then iteratively double Δ until there are too many Δ-paths. The previous value of Δ is then a good choice for Δ stepping.

At the first glance, it seems that $\log \Delta$ calls to the procedure findShortcuts from Fig. 9 would be necessary to find Δ. But we will see now that the same can be done with linear work. We now outline an algorithm that while exploring Δ-paths already prepares information needed for an exploration of 2Δ-paths. When the next iteration is started, it only has to continue the exploration from the previous phase. By continuously monitoring the amount of work performed, the search can be stopped when Δ has become too large to finish an iteration.

We now explain more details assuming that the adjacency lists have been preprocessed to be *partially sorted*: Let $\Delta_0 := \min_{e\in E} c(e)$ and assume[10] that $\Delta_0 > 0$. The adjacency lists are organized into *blocks* of edges with weight $2^j \Delta_0 \leqslant c(e) < 2^{j+1}\Delta_0$ for some integer j. Blocks with lighter edges precede blocks with heavier edges.[11]

Theorem 20. *Let n'_Δ, m'_Δ, and l'_Δ be defined as in Theorem* 18 *and consider an input with partially sorted adjacency lists. For any constant α, there is an algorithm which identifies a step width Δ, such that $n'_\Delta + m'_\Delta \leqslant \alpha \cdot (n+m)$, and $n'_{(2\Delta)} + m'_{(2\Delta)} > \alpha \cdot (n+m)$ which can be implemented to run in $\mathcal{O}((l'_\Delta + \log \Delta/\Delta_0)\log n)$ time using $\mathcal{O}(n+m)$ work whp.*

The basic idea is to reuse the procedure findShortcuts(Δ) of Fig. 9 but to divide the computation into *phases*. In phase i $(0 \leqslant i \leqslant \log(\max_{e\in E} c(e))/\Delta_0)$ we set $\Delta_{\text{cur}} = 2^i \Delta_0$ and find all connections (u, v, x) with $\Delta_{\text{cur}} \leqslant x < 2\Delta_{\text{cur}}$, $0 \leqslant i \leqslant \log(\max_{e\in E} c(e))/\Delta_0$.

In order to remain work efficient, a number of additional measures are necessary however. Since this routine 'findDelta' contains a number of technical details, we confine the pseudo code to Appendix A and only outline the changes compared to the routine 'findShortcuts' from Fig. 9 here. Most importantly, we have a bucketed *todo-list* T. $T[i]$ stores a list of entries (u, v, x, b) where (u, v, x) stands for a connection from u to v with weight x and b points to the first block in the adjacency list of v which may contain

[10] The result can be modified for the case when weight 0 edges are allowed.

[11] This preprocessing can be done efficiently sequentially. Since it is also trivially parallelizable on a node-by-node basis, we get a good parallel preprocessing algorithm for the case $p = \mathcal{O}(n/d)$. Otherwise, we know no algorithm which is always better than sorting. Sorting introduces a factor $\mathcal{O}(\log n)$ work overhead which can be amortized over multiple sources.

edges (v, w) with $2^i \Delta_0 \leqslant x + c(v, w) < 2 \cdot 2^i \Delta_0$. (Note that the number of buckets may be arbitrarily large; in this case, we store the buckets in a dynamic hash table and only initialize those buckets which actually store elements.)

At the starting of phase i, for each entry (u, v, x, b) of $T[i]$, the adjacency list of v is scanned beginning at block b until a block is encountered which cannot produce any candidate connections for bucket i. A new entry of the todo list is produced for the first bucket $k > i$ for which it can produce candidate connections. The candidate connections found are used to initialize Q'.

Both this initialization step and the iteration on Q can produce candidate connections whose weights reach into bucket $i + 1$. After removing duplicates and longer connections than found before, we therefore split the remaining candidates into the new content of Q and a set Q_{next} storing connections with weight in bucket $i + 1$.

At the end of phase i, when Q finally remains empty, we create new entries in the todo list for all connections newly encountered in phase i. In order to do that, we keep track of all new entries into 'found' using two sets S and S_{next} for connections with weights in bucket i and $i + 1$, respectively. S_{next} is used to initialize S in the next phase.

The total number of connection-edge pairs considered is monitored so that the whole procedure can be stopped as soon as it is noticed that this figure exceeds $\alpha \cdot (n + m)$. At this time, the entries of 'found' constitute at least all simple $(\Delta_{\text{cur}}/2)$-paths. Thus, taking $\Delta := \Delta_{\text{cur}}/2$ as the final step width, it is guaranteed that the number of reinsertions and re-relaxations in a subsequent application of the Δ-stepping will be bounded by $\mathcal{O}(n + m)$. On the other hand, $n'_{(2\Delta)} + m'_{(2\Delta)} > \alpha \cdot (n + m)$.

Using an analogous argument as in Section 6 for finding shortcuts it turns out that the search for Δ can be implemented to run in $\mathcal{O}((l'_\Delta + \log \Delta / \Delta_0) \log n)$ time using $\mathcal{O}(n + m)$ work where l'_Δ denotes the number of edges in the longest simple Δ-path.

8. Adaptation to distributed memory machines

After describing our machine model, we proceed to analyze a distributed memory version of the simple algorithm from Section 4.1 in Section 8.1. We refine it for random graphs in Section 8.2 and in Section 8.3, a general algorithm with more parallelism is introduced.

Consider the following abstract machine model: There are p PUs numbered 0 through $p - 1$ which are connected by a communication network. Let $T_{\texttt{routing}}(k)$ denote the time required to route k constant size messages per PU to random destinations. Let $T_{\texttt{coll}}(k)$ bound the time to perform a (possibly segmented) reduction or broadcast involving a message of length k and assume that $T_{\texttt{coll}}(x) + T_{\texttt{coll}}(y) \leqslant T_{\texttt{coll}}(1) + T_{\texttt{coll}}(x + y)$, i.e., concentrating message length does not decrease execution time. The analysis can focus on finding the number of necessary basic operations. The execution time for a particular network or abstract model is then easy to determine. For example, in the BSP model [63,92] we can substitute $T_{\texttt{routing}}(k) = \mathcal{O}(l + (k + \log p)g)$ whp and, using a pipelined implementation of collective communication, $T_{\texttt{coll}}(k) = \mathcal{O}(l \log p + gk)$. Note, that on powerful interconnection networks like multi-ported hypercubes we can achieve a time $\mathcal{O}(\log p + k)$ whp for $T_{\texttt{routing}}(k)$ and $T_{\texttt{coll}}(k)$.

We assume that the input is distributed over the local memories of the PUs such that each PU holds $\mathcal{O}(n/p)$ nodes. An adjacency list with k edges is evenly distributed over $\lceil kp/m \rceil$ consecutive PUs. The nodes can then be redistributed using a hash function $\mathrm{ind}(\cdot)$ which we assume to be computable in constant time. (Essentially the same assumptions are made for efficient PRAM simulation algorithms [92, Section 4.3] and this is certainly warranted for the simple hash functions used in practice.)

8.1. Adapting the simple algorithm

The PRAM algorithm from Section 4.1 is already almost a distributed memory algorithm. The hash function $\mathrm{ind}(w)$ replaces the index array used in the PRAM algorithm. The dart throwing process for assigning requests can be replaced by simply routing a request (w, x) to PU $\mathrm{ind}(w)$.

An analysis similar to the PRAM case yields the following bound:

Theorem 21. *The single source shortest path problem for directed graphs with n nodes, m edges, maximum in-degree and out-degree d, maximum path weight L, maximum Δ-size l_Δ and m_Δ defined as in Section 3.1 can be solved on a distributed memory machine with $p = \mathcal{O}(m\Delta/(dl_\Delta L))$ PUs in time*

$$\mathcal{O}\left(\overline{m} + T_{\mathtt{routing}}(\overline{m}) + T_{\mathtt{coll}}(\overline{m}) + dl_\Delta \frac{L}{\Delta}\big(T_{\mathtt{coll}}(1) + T_{\mathtt{routing}}(1)\big)\right) \quad \text{whp},$$

where $\overline{m} = n + m + n_\Delta + m_\Delta/p$.

Note, that on powerful interconnection networks like multi-ported hypercubes we get the same asymptotic performance as our CRCW-PRAM algorithm.

8.2. Improvements for random graphs

As in Section 4.2 we use the special case of random graphs as an opportunity to introduce fast parallel algorithms for handling large adjacency list. In a distributed memory setting we cannot dynamically schedule outgoing edges between the PUs using prefix sums as we did for PRAMs in Section 4.2. Instead, we introduce processor groups of size 2^i, $0 \leqslant i \leqslant \lceil \log P \rceil$. Each processor is member in one group of each size. Now adjacency lists are not assigned to random processors as before but to random processor groups. These groups can efficiently cooperate using pipelined collective broadcast and reduction operations. Similarly, if $p > n$, groups of processors cooperate to find the smallest incoming relaxation requests.

Theorem 22. *The SSSP on random graphs from $D(n, \bar{d}/n)$, maximum path weight L, maximum Δ-size l_Δ and $n_\Delta + m_\Delta = \mathcal{O}(n + m)$ defined as in Section 3.1 can be solved on a distributed memory machine with $p = \mathcal{O}((n + m)((L/\Delta + \log n)l_\Delta))$ PUs in time*

$$\mathcal{O}\left(\frac{n+m}{p} + T_{\texttt{routing}}\left(\frac{n+m}{p}\right) + T_{\texttt{coll}}\left(\frac{n+m}{p}\right) + l_{\Delta}\frac{L}{\Delta}\left(T_{\texttt{coll}}(1) + T_{\texttt{routing}}(1)\right)\right) \quad \textit{whp.}$$

In particular, Corollary 15 also transfers and for some distributed memory machines with powerful interconnection network we again get the same bounds as for CRCW-PRAMs.

We do a somewhat more detailed analysis here than for the corresponding result in Section 4.2 since Theorem 22 will not be fully superseded by a later result and because the techniques used here are also applicable to CRCW-PRAMs (where randomized dart throwing replaces routing).

Scanning adjacency lists to generate requests is load balanced using a static assignment of edges to PUs: An adjacency list of size outdegree(v) is collectively handled by an *out-group* of PUs. Out-groups are selected as follows: W.l.o.g., assume that p is a power of two minus one and the PUs are logically arranged as a complete binary tree. If outdegree$(v) > p$ then all PUs participate in v's out-group. Otherwise, a subtree rooted at a random PU is chosen which is just large enough to accommodate one edge per PU, i.e., if it contains $2^{\lceil \log(\text{outdegree}(v)+1)\rceil} - 1$ nodes. Requests for a bucket can now be generated by first sending the tentative distance of the nodes in $B[i]$ to the roots of out-groups responsible for them. (We will later see where this information comes from.) Then, the PUs pass all the node-distance pairs they have received down the tree in a pipelined fashion and do the same for the distances of the nodes received from above.

Now consider a fixed leaf PU j for a fixed iteration of the algorithm. (Since interior tree-nodes pass all their work downwards, interior PUs have no more work to do than a leaf node.) Let $X_i := 1$ if PU j is part of the out-group of a node i expanded in this iteration and $X_i = 0$ otherwise. We have $\mathbf{P}[X_i = 1] = 2^{-h(i)}$ if the root of the out-group of node i is $h(i)$ levels away from the root of the PU-tree. The total number of nodes PU j has to work on is $Y := \sum_{i=1}^{k} X_i$ if k is the number of nodes expanded in the current iteration and $\mathbf{E}[Y] = \sum_i 2^{-h(i)}$. By the definition of the size of subtrees, we get $\mathbf{E}[Y] = \mathcal{O}(K/p)$ if K is the total number of edges leaving nodes expanded in this iteration. Using a Chernoff bound with nonuniform probabilities [72, Theorem 4.1], it is now easy to see that $Y = \mathcal{O}(K/p + \log n)$ whp. Since the communication pattern is just a slightly generalized form of a broadcast, distributing the tentative distances can be done in time $\mathcal{O}(T_{\texttt{coll}}(K/p + \log n))$ whp. Summing over all iterations we get time $\mathcal{O}(T_{\texttt{coll}}((n+m)/p + \log n) + T_{\texttt{coll}}(1)L/\Delta)$. Generating the requests is then possible using local computations only.

For $d = \mathcal{O}((L/\Delta + \log n)l_{\Delta}\log n)$, the analysis in Section 4.2 can be applied to see that the random graph structure ensures good load balancing. Otherwise, we would like to apply more than n PUs. Then, there is no need for $p > n$ explicit local bucket structures any more. Rather, we organize the PUs into in-groups of $\lfloor p/n \rfloor$ PUs—one for each node and no hash function is needed any more. Requests (w, x) are now routed to random members of the in-group for w and Chernoff-bounds ensure good load balancing. At the end of each phase, a minimum reduction for each in-group determines the value used for relaxing w.

8.3. A faster algorithm using shortcuts

This section outlines a distributed memory implementation of the simple Δ-stepping algorithm with shortcuts from Fig. 8. This implementation works for arbitrary graphs. We limit ourselves to the shortest path search itself and only note that preprocessing can be done (somewhat inefficiently) by implementing semi-sorting using ordinary sorting or using a slower yet work efficient algorithm requiring $\mathcal{O}(T_{\mathtt{routing}}(n^{\epsilon}))$ time for any positive constant ϵ. Both alternatives yield a work-efficient algorithm for powerful interconnection networks if the preprocessing overhead can be amortized over sufficiently many source nodes.

Recall that the additional difficulty compared to random graphs is that we have to actively load balance incoming relaxation requests in the case of arbitrary graphs. This is more difficult than balancing outgoing requests since the number of requests for a node is not predictable. The semi-sorting routine we used in the PRAM algorithm does not transfer to a distributed memory setting. We circumvent semi-sorting by performing relaxations lazily at the last possible moment. Since edges need to be relaxed only once in a graph with shortcuts, performing relaxations becomes similar to generating requests because the total number of relaxations for a node is at most its in-degree.

Theorem 23. *Consider a directed graph G with n nodes, m edges, maximum path weight L and n_Δ, m_Δ, l_Δ as defined in Section* 3.1. *If G has been augmented with shortcut edges and for each edge (v, w), the in-degree of w is known, then the single source shortest path problem can be solved in time*

$$\mathcal{O}\left(\overline{m} + T_{\mathtt{routing}}(\overline{m}) + T_{\mathtt{coll}}(\overline{m}) + \frac{L}{\Delta}\big(T_{\mathtt{coll}}(1) + T_{\mathtt{routing}}(\log n)\big)\right) \quad whp,$$

on a distributed memory machine with p PUs for $\overline{m} = (n + m + n_\Delta + m_\Delta)/p$ any given source node s.

Scanning adjacency lists to generate requests can be done as in Section 8.2. The more difficult part is to assign the requests to nodes and schedule PUs for performing the relaxations.

The idea for assigning requests is to postpone the relaxation of an edge until the latest possible moment—just before the bucket of the target node is emptied. Since edges are relaxed only once, it pays to allocate an *in-group* of size $2^{\lceil \log(\mathrm{indegree}(v)+1)\rceil} - 1$ for node v analogously to the way out-groups are allocated. Each PU maintains an additional bucket structure B_q for the nodes for which it is part of the in-group. Requests are routed to a preassigned position in the in-group, but this information is only used to place the node into B_q. So, after iteration $i - 1$ is computed, the content of $B[i]$ is not yet known. Rather, we first have to find $B[i] = \bigcup B_q[i]$. This can be done locally for each in-group using a pipelined tree operation which is the converse of the operation used for broadcasting in the out-groups. (Each PU maintains a hash table of nodes already passed up the tree.) Then, the result is broadcast to all PUs in the in-groups so that from now on, redundant entries of nodes in buckets beyond $B[i]$ can be deleted. Also, edges which have not received a request yet are marked as superfluous. Requests ending up there in later iterations will simply be discarded. Finally, the actual global minima are computed using another pipelined reduction operation. Now the heads of the in-groups are ready to send the tentative distances of nodes in $B[i]$ to the heads of the out-groups. The analysis of these tree-operations is analogous the analysis for the out-groups in Section 8.2

9. Simulations and implementations

Simulations of different algorithm variants played a key role in designing the Δ-stepping algorithm. Here we report a few results obtained for random graphs with random edge weights and without shortcuts since they give a feeling for factors hidden behind our asymptotic analysis. Reachable nodes could be accessed by paths of weight $2.15\frac{\ln n}{d}$ or shorter. Figure 10 shows the tradeoff between the number of phases and reinsertions for a particular graph size and different edge probabilities. Interestingly, the number of phases needed seems to be in $\mathcal{O}(\log n)$ even though our analysis only guarantees $\mathcal{O}(\log^2 n/\log\log n)$. For example, for $\Delta = 4/d$ and $d \geqslant 2$ we never encountered more than $5 \ln n$ phases, the number of reinsertions was bounded by $0.25n$. The tested graphs ranged from 10^3 up to 10^6 nodes and comprised up to $3 \cdot 10^6$ edges.

Successfully implementing a linear work algorithm which requires a linear number of tiny messages with irregular communication pattern is not easy. However, for small p and large n, a machine with high bandwidth interconnection network and an efficient library routine for personalized all-to-all communication can do the job. We have implemented a simple version of the algorithm for distributed memory machines and random d-regular graphs using the library MPI [82]. Tests were run on an INTEL Paragon with 16 processors. For $n = 2^{19}$ nodes and $d = 3$, speedup 9.2 was obtained against the sequential Δ-stepping approach. The latter in turn is 3.1 times faster than an optimized implementation of Dijkstra's algorithm. Due to the increased communication costs, our results on dense graphs are slightly worse: for $n = 2^{16}$ and $d = 32$ the speedup of parallel Δ-stepping

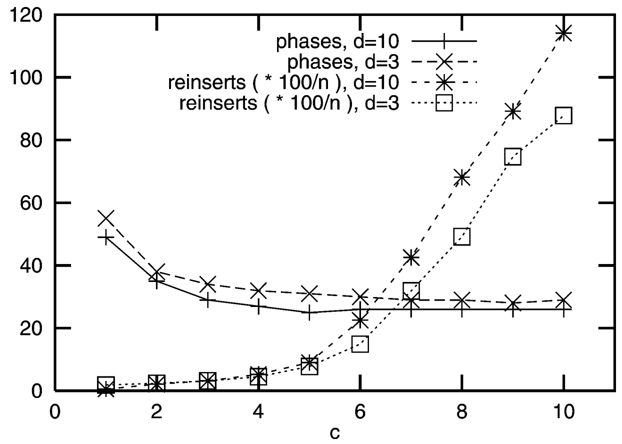


Fig. 10. Number of phases and reinserted nodes using Δ-stepping under different values of $\Delta = c/d$. All tests on graphs from $D(65\,536, d/n)$, $d = 3$, and $d = 10$.

compared to its sequential counterpart was 7.5,[12] sequential Δ-stepping was 1.8 times faster than Dijkstra's algorithm.

We also looked for other ways of determining the set R of nodes to be deleted in a phase. We have made experiments where $|R|$ is some fraction of the total priority queue size $|Q|$. In our simulations this works as well as $\Theta(1/d)$-stepping for random graphs with $|R| = \Theta(|Q|/\log|Q|)$, for random planar graphs we could even use $|R| = |Q|/2$. We also tested this approach on real world graphs and edge weights: starting with a road-map of a town ($n = 10,000$) the tested graphs successively grew up to a large road-map of Southern Germany ($n = 157,457$). Good performance was found for $|R| = \Theta(|Q|^{3/4})$. While repeatedly doubling the number of nodes, the average number of phases (for different starting points) only increased by a factor of about 1.5; for $n = 157,457$ the simulation needed 1178 phases, the number of reinserts was bounded by $0.2n$.

10. Discussion

We have developed a parallel algorithm for the shortest path problem which works for arbitrary directed graphs. How many processors can be used efficiently depends on parameters of the graph; most prominently on the ratio L/Δ between its maximal path weight and a step width Δ with the property that there is at most a linear number of shorter connections.

We have shown that Δ can easily be chosen for independent random edge weights. An example for dependent edge weights are random geometric graphs $G_n(r)$ 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 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 [28,80]. Taking $r = \Theta(\sqrt{\log(n)/n})$ results in a connected graph with $m = \Theta(n \log n)$ edges and $L = \mathcal{O}(1)$ whp. For $\Delta = r$ the graph already comprises all relevant Δ-shortcuts such that we do not have to explicitly insert them. Consequently our PRAM algorithm runs in $\mathcal{O}((1/r)\log n)$ parallel time and performs $\mathcal{O}(n + m)$ work whp.

We believe that the parameters l_Δ and l'_Δ which indicate the number of edges on Δ-paths the algorithm needs to traverse, are less important. For graphs where l_Δ and l'_Δ do matter, we could further diminish their influence by speeding up the introduction of shortcuts using the pointer doubling technique, i.e., by introducing the new shortcuts found in one phase of Algorithm 9 after each phase. In this way, $\mathcal{O}(\log l'_\Delta)$ phases rather than $\Theta(l'_\Delta)$ would be sufficient. Even a careful implementation of this idea might be slightly work-inefficient (polylogarithmic factors) but as for the distributed memory algorithm, the preprocessing could be amortized over multiple sources.

The main focus of this paper is a theoretical one, namely, to devise parallel shortest path algorithms which exhibit high parallelism for a large class of graphs. However, our

[12] Our current implementation does not distinguish between heavy and light edges. This increases the communication overhead. Therefore, we expect somewhat higher speedups for an improved implementation which is under development.

experiments indicate that at least the simple algorithm from Section 4.1 may also be of practical use for large graphs and a moderate number of processors. The extreme case is sequential Δ-stepping which can be a practical improvement since it needs no priority queue data structure and is more generally applicable than Dial's algorithm or Dinitz' improvement [30]. We have started an experimental study on parallel implementation which explores additional optimizations like partitioning the graph to increase locality and to find shortcuts efficiently for small p. The ultimate goal is to achieve useful speedups on real machines using graphs taken from real world problems, e.g., street graphs.

In recent work [66] it was shown how the Δ-stepping idea can be augmented by adaptive bucket-splitting, thus yielding the first sequential SSSP algorithm with provably linear average-case time for arbitrary directed graphs with random edge weights.

Acknowledgments

We thank in particular Hannah Bast, Kurt Mehlhorn, and Volker Priebe for many fruitful discussions and suggestions. Hannah Bast also pointed out the elegant solution of using a perfect hash function for semi-sorting requests. Furthermore, we thank the referees for their suggestions which considerably improved the presentation.

Appendix A. Pseudo code for 'findDelta'

```
**Function** findDelta() : step width
    found : HashArray[$V \times V$]                        (* return $\infty$ for undefined entries *)
    $Q := \{(u, u, 0)\colon u \in V\}$                      (* (start, destination, weight) *)
    $T[0] := \emptyset$                                     (* (start, destination, weight, block) *)
    $S := \emptyset$                                        (* remember new connections found *)
    $\Delta_{\mathrm{cur}} := \Delta_0 := \min_{e \in E} c(e)$          (* current step width *)
    $Q' := \emptyset$ : MultiSet
    for $i := 0$ to $\lceil \log(\max_{e \in E} c(e)/\Delta_0) \rceil$ do
        $S_{\mathrm{next}} := \emptyset$; $Q_{\mathrm{next}} := \emptyset$
        foreach $(u, v, x, b) \in T[i]$ dopar (if too many ops return $\Delta_{\mathrm{cur}}/2$)
            foreach edge $(v, w)$ of block $b$ in $v$'s adjacency list dopar
                $Q' := Q' \cup \{(u, w, x + c(v, w))\}$
            $j := \lfloor \log((x + c(\text{first edge of block } b + 1 \text{ in } v\text{'s adjacency list}))/\Delta_0) \rfloor$
            $T[j] := T[j] \cup \{(u, v, x, b + 1))\}$
        while $Q \neq \emptyset$ do
            foreach $(u, v, x) \in Q$ dopar (if too many ops return $\Delta_{\mathrm{cur}}/2$)
                foreach edge $(v, w) \in E$ having $c(v, w) \leqslant \Delta_{\mathrm{cur}}$ dopar
                    $Q' := Q' \cup \{(u, w, x + c(v, w))\}$
            semi-sort $Q'$ by common start and destination node
            $H := \{(u, v, x)\colon x = \min\{y\colon (u, v, y) \in Q'\}\}$
            $H := \{(u, v, x) \in H\colon x < \text{found}[(u, v)]\}$
            foreach $(u, v, x) \in H$ dopar
                if $x < \Delta_{\text{cur}}$ then
                    $Q := Q \cup \{(u, v, x)\}$
                    if found$[(u, v)] = \infty$ then $S := S \cup \langle u, v\rangle$
                else
                    $Q_{\text{next}} := Q_{\text{next}} \cup \{(u, v, x)\}$
                    if found$[(u, v)] = \infty$ then $S_{\text{next}} := S_{\text{next}} \cup \langle u, v\rangle$
                found$[(u, v)] := x$
            $Q' := \emptyset$
        od
        foreach $(u, v, b) \in S$ do
            $b :=$ first block in $v$'s adj. list having edges heavier than $\Delta_{\text{cur}}$
            $x :=$ found$[(u, v)]$
            $j := \lfloor \log((x + c(\text{first edge of block } b \text{ in } v\text{'s adjacency list}))/\Delta_0)\rfloor$
            $T[j] := T[j] \cup (u, v, x, b)$
        $Q := Q_{\text{next}}$; $S := S_{\text{next}}$
        $\Delta_{\text{cur}} := 2\Delta_{\text{cur}}$
    return $\max_{e\in E} c(e)$
```

References

[1] P. Adamson, E. Tick, Greedy partitioned algorithms for the shortest path problem, Internat. J. Parallel Program. 20 (4) (1991) 271–298.

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

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

[4] P. Alefragis, P. Sanders, T. Takkula, D. Wedelin, Parallel integer optimization for crew scheduling, Ann. Oper. Res. 99 (1) (2000) 141–166.

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

[6] D. Angluin, L.G. Valiant, Fast probabilistic algorithms for Hamiltonian paths and matchings, J. Comput. System Sci. 18 (1979) 155–193.

[7] M.J. Atallah, D.Z. Chen, O. Daescu, Efficient parallel algorithms for planar st-graphs, in: Proc. 8th Int. Symp. Algorithms and Computation, in: Lecture Notes in Comput. Sci., Vol. 1350, Springer-Verlag, 1997, pp. 223–232.

[8] K.B. Athreya, Large deviation rates for branching processes—I, single type case, Ann. Appl. Probab. 4 (3) (1994) 779–790.

[9] K.B. Athreya, P. Ney, Branching Processes, Springer-Verlag, 1972.

[10] H. Bast, T. Hagerup, Fast and reliable parallel hashing, in: 3rd Symposium on Parallel Algorithms and Architectures, 1991, pp. 50–61.

[11] R. Bellman, On a routing problem, Quart. Appl. Math. 16 (1958) 87–90.

[12] B. Bollobás, Random Graphs, Academic Press, 1985.

[13] G.S. Brodal, J.L. Träff, C.D. Zaroliagis, A parallel priority queue with constant time operations, J. Parallel Distrib. Comput. 49 (1) (1998) 4–21.

[14] K.M. Chandy, J. Misra, Distributed computation on graphs: Shortest path algorithms, Comm. ACM 25 (11) (1982) 833–837.

[15] S. Chaudhuri, C.D. Zaroliagis, Shortest paths in digraphs of small treewidth. Part II: Optimal parallel algorithms, Theoret. Comput. Sci. 203 (2) (1998) 205–223.

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

[17] B.V. Cherkassky, A.V. Goldberg, T. Radzik, Shortest path algorithms: Theory and experimental evaluation, Math. Programming 73 (1996) 129–174.
[18] A. Clementi, J. Rolim, E. Urland, Randomized parallel algorithms, in: Solving Combinatorial Problems in Parallel, in: Lecture Notes in Comput. Sci., Vol. 1054, 1996, pp. 25–50.
[19] E. Cohen, Using selective path-doubling for parallel shortest-path computations, J. Algorithms 22 (1) (1997) 30–56.
[20] E. Cohen, Polylog-time and near-linear work approximation scheme for undirected shortest paths, J. ACM 47 (2000) 132–166.
[21] C. Cooper, A. Frieze, K. Mehlhorn, V. Priebe, Average-case complexity of shortest-paths problems in the vertex-potential model, Random Structures Algorithms 16 (2000) 33–46.
[22] A. Crauser, K. Mehlhorn, U. Meyer, P. Sanders, A parallelization of Dijkstra's shortest path algorithm, in: Proc. 23rd Symp. on Mathematical Foundations of Computer Science, in: Lecture Notes in Comput. Sci., Vol. 1450, Springer-Verlag, 1998, pp. 722–731.
[23] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramanian, T. von Eicken, LogP: A practical model of parallel computation, Comm. ACM 39 (11) (1996) 78–85.
[24] E. Dekel, D. Nassimi, S. Sahni, Parallel matrix and graph algorithms, SIAM J. Comput. 10 (4) (1981) 61–67.
[25] E.V. Denardo, B.L. Fox, Shortest route methods: 1. Reaching pruning and buckets, Oper. Res. 27 (1979) 161–186.
[26] R.B. Dial, Algorithm 360: Shortest-path forest with topological ordering, Comm. ACM 12 (11) (1969) 632–633.
[27] R.B. Dial, F. Glover, D. Karney, D. Klingman, A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees, Networks 9 (1979) 215–248.
[28] J. Díaz, J. Petit, M. Serna, Random geometric problems on $[0,1]^2$, in: RANDOM: International Workshop on Randomization and Approximation Techniques in Computer Science, in: Lecture Notes in Comput. Sci., Vol. 1518, Springer-Verlag, 1998, pp. 294–306.
[29] E.W. Dijkstra, A note on two problems in connexion with graphs, Num. Math. 1 (1959) 269–271.
[30] E.A. Dinitz, Economical algorithms for finding shortest paths in a network, in: Transportation Modeling Systems, 1978, pp. 36–44.
[31] J.R. Driscoll, H.N. Gabow, R. Shrairman, R.E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, Comm. ACM 31 (11) (1988) 1343–1354.
[32] D.P. Dubhashi, A. Panconesi, Concentration of measure for the analysis of randomized algorithms, Draft Manuscript, http://www.brics.dk/~ale/papers.html, October, 1998.
[33] P. Erdős, A. Rényi, On the evolution of random graphs, Publ. Math. Inst. Hungar. Acad. Sci. 5(A) (1960) 17–61.
[34] L.R. Ford, D.R. Fulkerson, Flows in Networks, Princeton Univ. Press, Princeton, NJ, 1963.
[35] A. Formella, J. Keller, T. Walle, HPP: A high performance PRAM, in: Proc. Euro-Par 1996 Parallel Processing, in: Lecture Notes in Comput. Sci., Vol. 1124 II, Springer-Verlag, 1996, pp. 425–434.
[36] S. Fortune, J. Wyllie, Parallelism in random access memories, in: Proc. 10th Symp. on the Theory of Computing, ACM, 1978, pp. 114–118.
[37] M.L. Fredman, R.E. Tarjan, Fibonacci heaps their uses in improved network optimization algorithms, J. ACM 34 (1987) 596–615.
[38] M.L. Fredman, D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, J. Comput. System Sci. 48 (1994) 533–551.
[39] A.M. Frieze, G.R. Grimmett, The shortest-path problem for graphs with random arc-lengths, Discrete Appl. Math. 10 (1985) 57–77.
[40] A.M. Frieze, L. Rudolph, A parallel algorithm for all-pairs shortest paths in a random graph, in: Proc. 22nd Allerton Conference on Communication, Control and Computing, 1985, pp. 663–670.
[41] G. Gallo, S. Pallottino, Shortest path methods: A unifying approach, Math. Programming Study 26 (1986) 38–64.
[42] G. Gallo, S. Pallottino, Shortest path algorithms, Ann. Oper. Res. 13 (1988) 3–79.
[43] A.V. Gerbessiotis, L.G. Valiant, Direct bulk-synchronous parallel algorithms, J. Parallel Distrib. Comput. 22 (2) (1994) 251–267.
[44] F. Glover, R. Glover, D. Klingman, Computational study of an improved shortest path algorithm, Networks 14 (1984) 23–37.

[45] A.V. Goldberg, A simple shortest path algorithm with linear average time, in: Proc. 9th Ann. European Symposium on Algorithms (ESA), in: Lecture Notes in Comput. Sci., Vol. 2161, Springer-Verlag, 2001, pp. 230–241.

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

[47] L.M. Goldschlager, A unified approach to models of synchronous parallel machines, J. ACM 29 (4) (1982) 1073–1086.

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

[49] T. Hagerup, Improved shortest paths on the word RAM, in: 27th Colloquium on Automata, Languages and Programming (ICALP), in: Lecture Notes in Comput. Sci., Vol. 1853, Springer-Verlag, 2000, pp. 61–72.

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

[51] T. Harris, The Theory of Branching Processes, Springer-Verlag, 1963.

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

[53] M.R. Henzinger, P. Klein, S. Rao, S. Subramanian, Faster shortest-path algorithms for planar graphs, J. Comput. System Sci. 55 (1) (1997) 3–23.

[54] M.R. Hribar, 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.

[55] M.R. Hribar, V.E. Taylor, D.E. Boyce, Reducing the idle time of parallel shortest path algorithms, Technical Report CPDC-TR-9803-016, Northwestern University, Evanson, IL, 1998, J. Parallel Distrib. Comput., submitted for publication.

[56] M.R. Hribar, V.E. Taylor, D.E. Boyce, Termination detection for parallel shortest path algorithms, J. Parallel Distrib. Comput. 55 (1998) 153–165.

[57] M.S. Hung, J.J. Divoky, A computational study of efficient shortest path algorithms, Comput. Oper. Res. 15 (6) (1988) 567–576.

[58] J. Jájá, An Introduction to Parallel Algorithms, Addison–Wesley, Reading, 1992.

[59] R.M. Karp, The transitive closure of a random digraph, Random Structures Algorithms 1 (1) (1990) 73–93.

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

[61] P.N. Klein, S. Subramanian, A linear-processor polylog-time algorithm for shortest paths in planar graphs, in: Proc. 34th Annual Symposium on Foundations of Computer Science, IEEE, 1993, pp. 259–270.

[62] P.N. Klein, S. Subramanian, A randomized parallel algorithm for single-source shortest paths, J. Algorithms 25 (2) (1997) 205–220.

[63] W.F. McColl, Universal computing, in: L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Proc. Euro-Par '96 Parallel Processing, in: Lecture Notes in Comput. Sci., Vol. 1123, Springer-Verlag, 1996, pp. 25–36.

[64] C. McDiarmid, Concentration, in: M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, B. Reed (Eds.), Probabilistic Methods for Algorithmic Discrete Mathematics, in: Algorithms Combin., Vol. 16, Springer-Verlag, 1998, pp. 195–248.

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

[66] U. Meyer, Single-source shortest-paths on arbitrary directed graphs in linear average-case time, in: Proc. 12th Ann. Symp. on Discrete Algorithms, ACM–SIAM, 2001, pp. 797–806.

[67] U. Meyer, P. Sanders, Δ-stepping: A parallel single source shortest path algorithm, in: Proc. 6th Ann. European Symposium on Algorithms (ESA), in: Lecture Notes in Comput. Sci., Vol. 1461, Springer-Verlag, 1998, pp. 393–404.

[68] U. Meyer, P. Sanders, Parallel shortest path for arbitrary graphs, in: Proc. Euro-Par 2000 Parallel Processing, in: Lecture Notes in Comput. Sci., Vol. 1900, Springer-Verlag, 2000, pp. 461–470.

[69] G.L. Miller, J.H. Reif, Parallel tree contraction, Part 1: Fundamentals, in: S. Micali (Ed.), Advances in Computing Research 5: Randomness and Computation, JAI Press, 1989.

[70] A. Moffat, T. Takaoka, An all pairs shortest path algorithm with expected time $O(n^2 \log n)$, SIAM J. Comput. 16 (1987) 1023–1031.

[71] J.-F. Mondou, T.G. Crainic, S. Nugyen, Shortest path algorithms: A computational study with the C programming language, Comput. Oper. Res. 18 (1991) 767–786.
[72] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge Univ. Press, 1995.
[73] K. Mulmuley, P. Shah, A lower bound for the shortest path problem, in: Proc. 15th Annual Conference on Computational Complexity, IEEE, 2000, pp. 14–21.
[74] K. Noshita, A theorem on the expected complexity of Dijkstra's shortest path algorithm, J. Algorithms 6 (1985) 400–408.
[75] R.C. Paige, C.P. Kruskal, Parallel algorithms for shortest path problems, in: International Conference on Parallel Processing, IEEE, 1985, pp. 14–20.
[76] S. Rajasekaran, J.H. Reif, Optimal and sublogarithmic time randomized parallel sorting algorithms, SIAM J. Comput. 18 (3) (1989) 594–607.
[77] R. Raman, Priority queues: Small, monotone and trans-dichotomous, in: 4th Annual European Symposium on Algorithms (ESA), in: Lecture Notes in Comput. Sci., Vol. 1136, Springer-Verlag, 1996, pp. 121–137.
[78] R. Raman, Recent results on the single-source shortest paths problem, ACM SIGACT News 28 (2) (1997) 81–87.
[79] J. Reif, P. Spirakis, Expected parallel time and sequential space complexity of graph and digraph problems, Algorithmica 7 (1992) 597–630.
[80] R. Sedgewick, J.S. Vitter, Shortest paths in euclidean graphs, Algorithmica 1 (1986) 31–48.
[81] H. Shi, T.H. Spencer, Time-work tradeoffs of the single-source shortest paths problem, J. Algorithms 30 (1) (1999) 19–32.
[82] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, MPI—The Complete Reference, MIT Press, 1996.
[83] 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 J. Comput. 2 (1973) 28–32.
[84] S. Subramanian, R. Tamassia, J.S. Vitter, An efficient parallel algorithm for shortest paths in planar layered digraphs, Algorithmica 14 (4) (1995) 322–339.
[85] M. Thorup, Undirected single-source shortest paths with positive integer weights in linear time, J. ACM 46 (1999) 362–394.
[86] M. Thorup, Floats, integers, and single source shortest paths, J. Algorithms 35 (2000) 189–201.
[87] M. Thorup, On RAM priority queues, SIAM J. Comput. 30 (2000) 86–109.
[88] J.L. Träff, An experimental comparison of two distributed single-source shortest path algorithms, Parallel Comput. 21 (1995) 1505–1532.
[89] J.L. Träff, C.D. Zaroliagis, A simple parallel algorithm for the single-source shortest path problem on planar digraphs, J. Parallel Distrib. Comput. 60 (9) (2000) 1103–1124.
[90] J.D. Ullman, M. Yannakakis, High-probability parallel transitive closure algorithms, SIAM J. Comput. 20 (1) (1991) 100–125.
[91] L.G. Valiant, A bridging model for parallel computation, Comm. ACM 33 (8) (1990) 103–111.
[92] L.G. Valiant, General purpose parallel architectures, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, Elsevier, 1990, pp. 943–971.
[93] F.B. Zhan, C.E. Noon, Shortest path algorithms: An evaluation using real road networks, Transp. Sci. 32 (1998) 65–73.