Fast Hybrid Network Algorithms for Shortest Paths in Sparse Graphs

Michael Feldmann
Universität Paderborn, Germany
michael.feldmann@upb.de

Kristian Hinnenthal
Universität Paderborn, Germany
krijan@mail.upb.de

Christian Scheideler
Universität Paderborn, Germany
scheideler@upb.de

Abstract

We consider the problem of computing shortest paths in *hybrid networks*, in which nodes can make use of different communication modes. For example, mobile phones may use ad-hoc connections via Bluetooth or Wi-Fi in addition to the cellular network to solve tasks more efficiently. Like in this case, the different communication modes may differ considerably in range, bandwidth, and flexibility. We build upon the model of Augustine et al. [SODA '20], which captures these differences by a *local* and a *global* mode. Specifically, the local edges model a fixed communication network in which $O(1)$ messages of size $O(\log n)$ can be sent over every edge in each synchronous round. The global edges form a clique, but nodes are only allowed to send and receive a total of at most $O(\log n)$ messages over global edges, which restricts the nodes to use these edges only very sparsely.

We demonstrate the power of hybrid networks by presenting algorithms to compute Single-Source Shortest Paths and the diameter very efficiently in *sparse graphs*. Specifically, we present exact $O(\log n)$ time algorithms for cactus graphs (i.e., graphs in which each edge is contained in at most one cycle), and 3-approximations for graphs that have at most $n + O(n^{1/3})$ edges and arboricity $O(\log n)$. For these graph classes, our algorithms provide exponentially faster solutions than the best known algorithms for general graphs in this model. Beyond shortest paths, we also provide a variety of useful tools and techniques for hybrid networks, which may be of independent interest.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases hybrid networks, overlay networks, sparse graphs, cactus graphs

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.31

Related Version A full version of the paper is available at `https://arxiv.org/abs/2007.01191`.

Funding This work is supported by the German Research Foundation (DFG) within the CRC 901 "On-The-Fly Computing" (project number 160364472-SFB901).

1 Introduction

The idea of *hybrid networks* is to leverage multiple communication modes with different characteristics to deliver scalable throughput, or to reduce complexity, cost or power consumption. In *hybrid data center networks* [10], for example, the server racks can make use of optical switches [13] or wireless antennas [11] to establish direct connections in addition to using the traditional electronic packet switches. Other examples of hybrid communication are combining multipoint with standard VPN connections [30], hybrid WANs [32], or mobile phones using device-to-device communication in addition to cellular networks as in 5G [22]. As a consequence, several theoretical models and algorithms have been proposed for hybrid networks in recent years [16, 20, 4, 5].

In this paper, we focus on the general hybrid network model of Augustine et al. [5]. The authors distinguish two different modes of communication, a *local* mode, which nodes can use to send messages to their neighbors in an input graph G, and a *global* mode, which allows the nodes to communicate with *any* other node of G. The model is parameterized by the number of messages λ that can be sent over each local edge in each round, and the total number of messages γ that each node can send and receive over global edges in a single round. Therefore, the local network rather relates to *physical* networks, where an edge corresponds to a dedicated connection that cannot be adapted by the nodes, e.g., a cable, an optical connection, or a wireless ad-hoc connection. On the other hand, the global network captures characteristics of *logical* networks, which are formed as overlays of a shared physical infrastructure such as the internet or a cellular network. Here, nodes can in principle contact any other node, but can only perform a limited amount of communication in each round.

Specifically, we consider the hybrid network model with $\lambda = O(1)$ and $\gamma = O(\log n)$, i.e., the local network corresponds to the CONGEST model [28], whereas the global network is the so-called *node-capacitated clique* (NCC) [4, 1, 29]. Thereby, we only grant the nodes very limited communication capabilities for both communication modes, disallowing them, for example, to gather complete neighborhood information to support their computation. With the exception of a constant factor SSSP approximation, none of the shortest paths algorithms of [5], for example, can be directly applied to this very restricted setting, since [5] assumes the LOCAL model for the local network. Furthermore, our algorithms do not even exploit the power of the NCC for the global network; in fact, they would also work if the nodes would initially only knew their neighbors in G and had to learn new node identifiers via introduction (which has recently been termed the NCC_0 model [3]).

As in [5], we focus on *shortest paths problems*. However, instead of investigating general graphs, we present polylogarithmic time algorithms to compute Single-Source Shortest Paths (SSSP) and the diameter in *sparse graphs*. Specifically, we present randomized $O(\log n)$ time algorithms for *cactus graphs*, which are graphs in which any two cycles share at most one node. Cactus graphs are relevant for wireless communication networks, where they can model combinations of star/tree and ring networks (e.g., [9]), or combinations of ring and bus structures in LANs (e.g., [24]). However, research on solving graph problems in cactus graphs mostly focuses on the sequential setting.

Furthermore, we present 3-approximate randomized algorithms with runtime $O(\log^2 n)$ for graphs that contain at most $n + O(n^{1/3})$ edges and have arboricity[1] $O(\log n)$. Graphs with bounded arboricity, which include important graph families such as planar graphs, graphs with bounded treewidth, or graphs that exclude a fixed minor, have been extensively studied in the past years. Note that although these graphs are very sparse, in contrast to cactus graphs they may still contain a polynomial number of (potentially nested) cycles. Our algorithms are exponentially faster than the best known algorithms for general graphs for shortest paths problems [4, 23].

For the *All-Pairs Shortest Paths* (APSP) problem, which is not studied in this paper, there is a lower bound of $\widetilde{\Omega}(\sqrt{n})$ [5, Theorem 2.5] that even holds for $\widetilde{O}(\sqrt{n})$-approximations[2]. Recently, this lower bound was shown to be tight up to polylogarithmic factors [23]. The bound specifically also holds for trees, which, together with the results in this paper, shows an exponential gap between computing the diameter and solving APSP in trees. Furthermore,

[1] The arboricity of a graph G is the minimum number of forests into which its edges can be partitioned.
[2] The $\widetilde{O}$-notation hides polylogarithmic factors.

the results of [23] show that computing (an approximation of) the diameter in general graphs takes time roughly $\Omega(n^{1/3})$ (even with unbounded local communication). Therefore, our paper demonstrates that sparse graphs allow for an exponential improvement.

1.1 Model and Problem Definition

We consider a *hybrid network model* in which we are given a fixed node set V consisting of n nodes that are connected via *local* and *global* edges. The local edges form a fixed, undirected, and weighted graph $G = (V, E, w)$ (the *local network*), where the edge weights are given by $w : E \to \{1, \ldots, W\} \subset \mathbb{N}$ and W is assumed to be polynomial in n. We denote the degree of a node v in the local network by $\deg(v)$. Furthermore, every two nodes $u, v \in V$ are connected via a global edge, i.e., the *global network* forms a clique. Every node $v \in V$ has a unique identifier $\mathrm{id}(v)$ of size $O(\log n)$, and, since the nodes form a clique in the global network, every node knows the identifier of every other node. Although this seems to be a fairly strong assumption, our algorithms would also work in the NCC_0 model [3] for the global network, in which each node initially only knows the identifiers of its neighbors in G, and new connections need to be established by sending node identifiers (which is very similar to the overlay network models of [16, 6, 17]). We further assume that the nodes know n (or an upper bound polynomial in n).

We assume a synchronous message passing model, where in each round every node can send messages of size $O(\log n)$ over both local and global edges. Messages that are sent in round i are collectively received at the beginning of round $i + 1$. However, we impose different communication restrictions on the two network types. Specifically, every node can send $O(1)$ (distinct) messages over each of its incident local edges, which corresponds to the CONGEST model for the local network [28]. Additionally, it can send and receive at most $O(\log n)$ many messages over global edges (where, if more than $O(\log n)$ messages are sent to a node, an arbitrary subset of the messages is delivered), which corresponds to the NCC model [4]. Therefore, our hybrid network model is precisely the model proposed in [5] for parameters $\lambda = O(1)$ and $\gamma = O(\log n)$. Note that whereas [5] focuses on the much more generous LOCAL model for the local network, our algorithms do not require nor easily benefit from the power of unbounded communication over local edges.

We define the *length* of a path $P \subseteq E$ as $w(P) := \sum_{e \in P} w(e)$. A path P from u to v is a *shortest path*, if there is no path P' from u and v with $w(P') < w(P)$. The *distance* between two nodes u and v is defined as $d(u, v) := w(P)$, where P is a shortest path from u to v.

In the *Single-Source Shortest Paths Problem* (SSSP), there is one node $s \in V$ and every node $v \in V$ wants to compute $d(s, v)$. In the *Diameter Problem*, every node wants to learn the *diameter* $D := \max_{u,v \in V} d(u, v)$. An algorithm computes an α-approximation of SSSP, if every node $v \in V$ learns an estimate $\widetilde{d}(s, v)$ such that $d(s, v) \leq \widetilde{d}(s, v) \leq \alpha \cdot d(s, v)$. Similarly, for an α-approximation of the diameter, every node $v \in V$ has to compute an estimate $\widetilde{D}$ such that $D \leq \widetilde{D} \leq \alpha \cdot D$.

1.2 Contribution and Structure of the Paper

The first part of the paper revolves around computing SSSP and the diameter on cactus graphs (i.e., connected graphs in which each edge is only contained in at most one cycle). For a more comprehensive presentation, we establish the algorithm in several steps. First, we consider the problems in path graphs (i.e., connected graphs that contain exactly two nodes with degree 1, and every other node has degree 2; see Section 2), then in cycle graphs (i.e., connected graphs in which each node has degree 2, see Section 3), trees (Section 4), and

pseudotrees (Section 5), which are graphs that contain at most one cycle. For each of these graph classes, we present deterministic algorithms to solve both problems in $O(\log n)$ rounds, each relying heavily on the results of the previous sections. We then extend our results to cactus graphs (Section 6) and present randomized algorithms for SSSP and the diameter with a runtime of $O(\log n)$, w.h.p.[3]

In Section 7, we consider a more general class of sparse graphs, namely graphs with at most $n + O(n^{1/3})$ edges and arboricity $O(\log n)$. By using the techniques established in the first part and leveraging the power of the global network to deal with the additional $O(n^{1/3})$ edges, we obtain algorithms to compute 3-approximations for SSSP and the diameter in time $O(\log^2 n)$, w.h.p. As a byproduct, we also derive a deterministic $O(\log^2 n)$-round algorithm for computing a (balanced) hierarchical tree decomposition of the network.

We remark that our algorithms heavily use techniques from the PRAM literature. For example, *pointer jumping* [19], and the *Euler tour* technique (e.g., [31, 2]), which extends pointer jumping to certain graphs such as trees, have been known for decades, and are also used in distributed algorithms (e.g., [16, 6]). As already pointed out in [4], the NCC in particular has a very close connection to PRAMs. In fact, if G is very sparse, PRAM algorithms can efficiently be simulated in our model even if the edges are very unevenly distributed (i.e., nodes have a very high degree). We formally prove this in the full version of this paper [14]. This allows us to obtain some of our algorithms for path graphs, cycle graphs, and trees by PRAM simulations (see Section 1.3). We nonetheless present our distributed solutions without using PRAM simulations, since (1) a direct simulation only yields randomized algorithms, (2) the algorithms of the later sections heavily build on the basic algorithms of the first sections, (3) a simulation exploits the capabilities of the global network more than necessary. As already pointed out, *all* of our algorithms would also work in the weaker NCC_0 model for the global network, or if the nodes could only contact $\Theta(\log n)$ random nodes in each round.[4] Furthermore, if we restrict the degree of G to be $O(\log n)$, our algorithms can be modified to run in the NCC_0 without using the local network.

Beyond the results for sparse graphs, this paper contains a variety of useful tools and results for hybrid networks in general, such as Euler tour and pointer jumping techniques for computation in trees, a simple load-balancing framework for low-arboricity graphs, an extension of the recent result of Götte et al. [18] to compute spanning trees in the NCC_0, and a technique to perform matrix multiplication. In combination with sparse spanner constructions (see, e.g., [8]) or skeletons (e.g., [33]), our algorithms may lead to efficient shortest path algorithms in more general graph classes. Also, our algorithm to construct a hierarchical tree decomposition may be of independent interest, as such constructions are used for example in routing algorithms for wireless networks (see, e.g., [15, 21]).

Due to space constraints, all proofs and figures, as well as the detailed description and some lemmas of our algorithms, are deferred to the full version of this paper [14].

1.3 Further Related Work

As theoretical models for hybrid networks have only been proposed recently, only few results for such models are known at this point [16, 4, 5]. Computing an exact solution for SSSP in arbitrary graphs can be done in $\widetilde{O}(\sqrt{\mathsf{SPD}})$ rounds [5], where SPD is the so-called

[3] An event holds with high probability (w.h.p.) if it holds with probability at least $1 - 1/n^c$ for an arbitrary but fixed constant $c > 0$.

[4] We remark that for the algorithms in Section 7 this requires to setup a suitable overlay network like a butterfly in time $O(\log^2 n)$, which can be done using well-known techniques.

shortest path diameter of G. For large SPD, this bound has recently been improved to $\widetilde{O}(n^{2/5})$ [23]. The authors of [5] also present several approximation algorithms for SSSP: A $(1+\varepsilon)$-approximation with runtime $\widetilde{O}(n^{1/3}/\varepsilon^6)$, a $(1/\varepsilon)^{O(1/\varepsilon)}$-approximation running in $\widetilde{O}(n^\varepsilon)$ rounds and a $2^{O(\sqrt{\log n \log\log n})}$-approximation with runtime $2^{O(\sqrt{\log n \log\log n})}$. For APSP there is an exact algorithm that runs in $\widetilde{O}(n^{2/3})$ rounds, a $(1+\varepsilon)$-approximation running in $\widetilde{O}(\sqrt{n/\varepsilon})$ rounds (only for unweighted graphs) and a 3-approximation with runtime $\widetilde{O}(\sqrt{n})$ [5]. In [23], the authors give a lower bound of $\widetilde{\Omega}(n^{1/3})$ rounds for computing the diameter in arbitrary graphs in our model. They also give approximation algorithms with approximation factors $(3/2+\varepsilon)$ and $(1+\varepsilon)$ that run in time $\widetilde{O}(n^{1/3}/\varepsilon)$ and $\widetilde{O}(n^{0.397}/\varepsilon)$, respectively. Even though APSP and the diameter problem are closely related, we demonstrate that the diameter can be computed much faster in our hybrid network model for certain graphs classes.

As already pointed out, the global network in our model has a close connection to overlay networks. The NCC model, which has been introduced in [4], mainly focuses on the impact of node capacities, especially when the nodes have a high degree. Since, intuitively, for many graph problems the existence of *each* edge is relevant for the output, most algorithms in [4] depend on the arboricity a of G (which is, roughly speaking, the time needed to efficiently distribute the load of all edges over the network). The authors present $\widetilde{O}(a)$ algorithms for local problems such as MIS, matching, or coloring, an $\widetilde{O}(D+a)$ algorithm for BFS tree, and an $\widetilde{O}(1)$ algorithm to compute a minimum spanning tree (MST). Recently, $\widetilde{O}(\Delta)$-time algorithms for graph realization problems have been presented [3], where Δ is the maximum node degree; notably, most of the algorithms work in the NCC_0 variant. Furthermore, Robinson [29] investigates the information the nodes need to learn to jointly solve graph problems and derives a lower bound for constructing spanners in the NCC. For example, his result implies that spanners with constant stretch require polynomial time in the NCC, and are therefore harder to compute than MSTs. Since our global network behaves like an overlay network, we can make efficient use of the so-called *shortest-path diameter reduction technique* [26]. By adding shortcuts between nodes in the global network, we can bridge large distances quickly throughout our computations.

As argued before, we could apply some of the algorithms for PRAMs to our model instead of using native distributed solutions by using PRAM simulations. For example, we are able to use the algorithms of [12] to solve SSSP and diameter in trees in time $O(\log n)$, w.h.p. Furthermore, we can compute the distance between any pair s and t in *outerplanar graphs* in time $O(\log^3 n)$ by simulating a CREW PRAM. For planar graphs, the distance between s and t can be computed in time $O(\log^3 n(1+M(q))/n)$, w.h.p., where the nodes know a set of q faces of a planar embedding that covers all vertices, and $M(q)$ is the number of processors required to multiply two $q \times q$ matrices in $O(\log q)$ time in the CREW PRAM.

For graphs with polylogarithmic arboricity, a $(1+\varepsilon)$-approximation of SSSP can be computed in polylog time using [25] and our simulation framework (with huge polylogarithmic terms). For general graphs, the algorithm can be combined with well-known spanner algorithms for the CONGEST model (e.g., [8]) to achieve constant approximations for SSSP in time $\widetilde{O}(n^\varepsilon)$ time in our hybrid model. This yields an alternative to the SSSP approximation of [5], which also requires time $\widetilde{O}(n^\varepsilon)$ but has much smaller polylogarithmic factors.

2 Path Graphs

To begin with an easy example, we first present a simple algorithm to compute SSSP and the diameter of path graphs. The simple idea of our algorithms is to use *pointer jumping* to select a subset of global edges S, which we call *shortcut edges*, with the following properties:

S is a weighted connected graph with degree $O(\log n)$ that contains all nodes of V, and for every $u, v \in V$ there exists a path $P \subseteq S$, $|P| = O(\log n)$ (where $|P|$ denotes the number of edges of P), such that $w(P) = d(u, v)$, and no path P such that $w(P) < d(u, v)$. Given such a graph, SSSP can easily be solved by performing a broadcast from s in S for $O(\log n)$ rounds: In the first round, s sends a message containing $w(e)$ over each edge $e \in S$ incident to s. In every subsequent round, every node $v \in V$ that has already received a message sends a message $k + w(e)$ over each edge $e \in S$ incident to v, where k is the smallest value v has received so far. After $O(\log n)$ rounds, every node v must have received $d(s, v)$, and cannot have received any smaller value. Further, the diameter of the line can easily be determined by performing SSSP from both of its endpoints u, v, which finally broadcast the diameter $d(u, v)$ to all nodes using the global network.

We construct S using the following simple *Introduction Algorithm*. S initially contains all edges of E. Additional shortcut edges are established by performing *pointer jumping*: Every node v first selects one of its at most two neighbors as its *left* neighbor ℓ_1; if it has two neighbors, the other is selected as v's *right* neighbor r_1. In the first round of our algorithm, every node v with degree 2 establishes $\{\ell_1, r_1\}$ as a new shortcut edge of weight $w(\{\ell_1, r_1\}) = w(\{\ell_1, v\}) + w(\{v, r_1\})$ by sending the edge to both ℓ_1 and r_1. Whenever at the beginning of some round $i > 1$ a node v with degree 2 receives shortcut edges $\{u, v\}$ and $\{v, w\}$ from ℓ_{i-1} and r_{i-1}, respectively, it sets $\ell_i := u$, $r_i := w$, and establishes $\{\ell_i, r_i\}$ by adding up the weights of the two received edges and informing ℓ_i and r_i. The algorithm terminates after $\lfloor \log(n-1) \rfloor$ rounds. Afterwards, for every simple path in G between u and v with 2^k hops for any $k \leq \lfloor \log(n-1) \rfloor$ we have established a shortcut edge $e \in S$ with $w(e) = d(u, v)$. Therefore, S has the desired properties, and we conclude the following theorem.

▶ **Theorem 1.** *SSSP and the diameter can be computed in any path graph in time $O(\log n)$.*

3 Cycle Graphs

In cycle graphs, there are two paths between any two nodes that we need to distinguish. For SSSP, this can easily be achieved by performing the SSSP algorithm for path graphs in both directions along the cycle, and let each node choose the minimum of its two computed distances. Formally, let $v_1, v_2, \ldots, v_n$ denote the n nodes along a *left* traversal of the cycle starting from $s = v_1$ and continuing at s's neighbor of smaller identifier, i.e., $\mathrm{id}(v_2) < \mathrm{id}(v_n)$. For any node u, a shortest path from s to u must follow a left or right traversal along the cycle, i.e., $(v_1, v_2, \ldots, u)$ or $(v_1, v_n, \ldots, u)$ is a shortest path from s to u. Therefore, we can solve SSSP on the cycle by performing the SSSP algorithm for the path graph on $\mathcal{L} := (v_1, v_2, \ldots, v_n)$ and $\mathcal{R} := (v_1, v_n, v_{n-1}, \ldots, v_2)$. Thereby, every node v learns $d_\ell(s, v)$, which is the distance from s to v in $\mathcal{L}$ (i.e., along a left traversal of the cycle), and $d_r(s, v)$, which is their distance in $\mathcal{R}$. It is easy to see that $d(s, v) = \min\{d_\ell(s, v), d_r(s, v)\}$.

Using the above algorithm, s can also easily learn its *eccentricity* $\mathrm{ecc}(s) := \max_{v \in V}\{d(s, v)\}$, as well as its *left and right farthest nodes* s_ℓ and s_r. The left farthest node s_ℓ of s is defined as the farthest node v_i along a left traversal of the cycle such that the subpath in $\mathcal{L}$ from $s = v_1$ to v_i is still a shortest path. Formally, $s_\ell = \arg\max_{v \in V, d_\ell(s,v) \leq \lfloor W/2 \rfloor} d_\ell(s, v)$, where $W = \sum_{e \in E} w(e)$. The right farthest node s_r is the successor of s_ℓ in $\mathcal{L}$ (or s, if s_ℓ is the last node of $\mathcal{L}$), for which it must hold that $d_r(s, s_r) \leq \lfloor W/2 \rfloor$. Note that $d_\ell(s, s_\ell) = d(s, s_\ell)$, $d_r(s, s_r) = d(s, s_r)$, and $\mathrm{ecc}(s) = \max\{d_\ell(s, s_\ell), d_r(s, s_r)\}$.

To determine the diameter of G, for every node $v \in V$ our goal is to compute $\mathrm{ecc}(v)$; as a byproduct, we will compute v's left and right farthest nodes v_ℓ and v_r. The diameter can then be computed as $\max_{v \in V} \mathrm{ecc}(v)$. A simple way to compute these values is to employ a

binary-search style approach from all nodes in parallel, and use load balancing techniques from [4] to achieve a runtime of $O(\log^2 n)$, w.h.p. Coming up with a deterministic $O(\log n)$ time algorithm, however, is more complicated.

Due to space constraints, we defer the description of the algorithm to the full version.

▶ **Theorem 2.** *SSSP and the diameter can be computed in any cycle graph G in time $O(\log n)$.*

4 Trees

We now show how the algorithms of the previous sections can be extended to compute SSSP and the diameter on trees. As in the algorithm of Gmyr et al. [16], we adapt the well-known *Euler tour* technique to a distributed setting and transform the graph into a path L of *virtual nodes* that corresponds to a depth-first traversal of G. More specifically, every node of G simulates one virtual node for each time it is visited in that traversal, and two virtual nodes are neighbors in L if they correspond to subsequent visitations. To solve SSSP, we assign weights to the edges from which the initial distances in G can be inferred, and then solve SSSP in L instead. Finally, we compute the diameter of G by performing the SSSP algorithm twice, which concludes this section.

However, since a node can be visited up to $\Omega(n)$ times in the traversal, it may not be able to simulate all of its virtual nodes in L. Therefore, we first need to reassign the virtual nodes to the node's neighbors such that every node only has to simulate at most 6 virtual nodes using the *Nash-Williams forests decomposition* technique [27]. More precisely, we compute an *orientation* of the edges in which each node has outdegree at most 3, and reassign nodes according to this orientation (in the remainder of this paper, we refer to this as the *redistribution framework*). Due to space constraints, we defer a precise description of the algorithm to the full version and only state our main results.

The following two lemmas follow from applying PRAM techniques.

▶ **Lemma 3.** *Let $H = (V, E)$ be a forest in which every node $v \in V$ stores some value p_v, and let f be a distributive aggregate function[5]. Every node $v \in V$ can learn $f(\{p_u \mid u \in C_v\})$, where C_v is the tree of H that contains v, in time $O(\log n)$.*

▶ **Lemma 4.** *Any tree G can be rooted in $O(\log n)$ time.*

By assigning positive or negative weights to the edges of L according to their direction in the rooted version of G, we easily obtain the following theorem.

▶ **Theorem 5.** *SSSP can be computed in any tree in time $O(\log n)$.*

Similar techniques lead to the following lemmas, which we will use in later sections.

▶ **Lemma 6.** *Let $H = (V, E)$ be a forest and assume that each node $v \in V$ stores some value p_v. The goal of each node v is to compute the value $sum_v(u) := \sum_{w \in C_u} p_w$ for each of its neighbors u, where C_u is the connected component C of the subgraph H' of H induced by $V \setminus \{v\}$ that contains u. The problem can be solved in time $O(\log n)$.*

[5] An aggregate function f is called *distributive* if there is an aggregate function g such that for any multiset S and any partition $S_1, \ldots, S_\ell$ of S, $f(S) = g(f(S_1), \ldots, f(S_\ell))$. Classical examples are MAX, MIN, and SUM.

▶ **Lemma 7.** *Let G be a tree rooted at s. Every node $v \in V$ can compute its height $h(v)$ in G, which is length of the longest path from v to any leaf in its subtree, in time $O(\log n)$.*

For the diameter, we use the following well-known lemma.

▶ **Lemma 8.** *Let G be a tree, $s \in V$ be an arbitrary node, and let $v \in V$ such that $d(s, v)$ is maximal. Then $ecc(v) = D$.*

Therefore, for the diameter it suffices to perform SSSP once from the node s with highest identifier, then choose a node v with maximum distance to s, and perform SSSP from v. Since $\text{ecc}(v) = D$, the node with maximum distance to v yields the diameter. Together with Lemma 3, we conclude the following theorem.

▶ **Theorem 9.** *The diameter can be computed in any tree in time $O(\log n)$.*

5 Pseudotrees

Recall that a pseudotree is a graph that contains at most one cycle. We define a *cycle node* to be a node that is part of a cycle, and all other nodes as *tree nodes*. For each cycle node v, we define v's tree T_v as the connected component that contains v in the graph in which v's two adjacent cycle nodes are removed, and denote $h(v)$ as the height of v in T_v. Due to space constraints, we omit the details of the algorithm for pseudotrees and only give a brief description. To compute SSSP, we first need to distinguish the cycle nodes from the tree nodes. We do this by establishing rings of virtual nodes using the approach of Section 4 (which must create two rings in a pseudotree). Then, we can reduce the problem to computing SSSP in cycles and trees, for which we use the algorithms from the previous sections.

▶ **Theorem 10.** *SSSP can be computed in any pseudotree in time $O(\log n)$.*

Computing the diameter is more complicated. Since the longest path may not use any cycle node at all, each cycle node v first contributes the diameter of its tree T_v as a possible candidate. Furthermore, v needs to compute its eccentricity $\text{ecc}(v)$, and, if its eccentricity is larger than the height $h(v)$ of its tree T_v, contribute $\text{ecc}(v)+h(v)$. To compute its eccentricity, every cycle node needs to compute the distance to its farthest nodes using the algorithm of Theorem 2, but also take into account the heights of the trees on the path to these nodes (as a longer path may lead into those trees).

▶ **Theorem 11.** *The diameter can be computed in any pseudotree in time $O(\log n)$.*

6 Cactus Graphs

Our algorithm for cactus graphs relies on an algorithm to compute the maximal biconnected components (or *blocks*) of G, where a graph is called biconnected if the removal of a single node would not disconnect the graph. Note that for any graph, each edge lies in exactly one block. In case of cactus graphs, each block is either a single edge or a simple cycle. By computing the blocks of G, each node $v \in V$ classifies its incident edges into bridges (if there is no other edge incident to v contained in the same block) and pairs of edges that lie in the same cycle. To do so, we first give a variant of [18, Theorem 1.3] for the NCC_0 under the constraint that the input graph (which is not necessarily a cactus graph) has constant degree. We point out how the lemma is helpful for cactus graphs, and then use a simulation of the biconnectivity algorithm of [31] as in [18, Theorem 1.4] to compute the blocks of G. The description and proofs of the following three lemmas are very technical and mainly describe adaptions of [18]. Therefore, we defer them to the full version [14].

▶ **Lemma 12** (Variant of [18, Theorem 1.3]). *Let G be any graph with constant degree. A spanning tree of G can be computed in time $O(\log n)$, w.h.p., in the NCC_0.*

▶ **Lemma 13.** *A spanning tree of a cactus graph G can be computed in time $O(\log n)$, w.h.p.*

▶ **Lemma 14.** *The biconnected components of a cactus graph G can be computed in time $O(\log n)$, w.h.p.*

Thus, every node can determine which of its incident edges lie in the same block in time $O(\log n)$, w.h.p. Let s be the source for the SSSP problem. First, we compute the *anchor node* of each cycle in G, which is the node of the cycle that is closest to s (if s is a cycle node, then the anchor node of that cycle is s itself). To do so, we replace each cycle C in G by a binary tree T_C of height $O(\log n)$ as described in [16]. More precisely, we first establish shortcut edges using the Introduction algorithm in each cycle, and then perform a broadcast from the node with highest identifier in C for $O(\log n)$ rounds. If in some round a node receives the broadcast for the first time from ℓ_i or r_i, it sets that node as its parent in T_C and forwards the broadcast to ℓ_j and r_j, where $j = \min\{i-1, 0\}$. After $O(\log n)$ rounds, T_C is a binary tree that contains all nodes of C and has height $O(\log n)$. To perform the execution in all cycles in parallel, each node simulates one virtual node for each cycle it lies in and connects the virtual nodes using their knowledge of the blocks of G. To keep the global communication low, we again use the redistribution framework described in Section 4 (note that the arboricity of G is 2).

▶ **Lemma 15.** *Let T be the (unweighted) tree that results from taking the union of all trees T_C and all bridges in G. For each cycle C, the node $a_C := \arg\min_{v \in C} d_T(s, v)$ is the anchor node of C.*

The correctness of the lemma above simply follows from the fact that any shortest path from s to any node in C must contain the anchor node of C both in G and in T. Therefore, the anchor node of each cycle can be computed by first performing the SSSP algorithm for trees with source s in T and then conducting a broadcast in each cycle. Now let v be the anchor node of some cycle C in G. By performing the diameter algorithm of Theorem 2 in C, v can compute its left and right farthest nodes v_ℓ and v_r in C. Again, to perform all executions in parallel, we use our redistribution framework.

▶ **Lemma 16.** *Let S_G be the graph that results from removing the edge $\{v_\ell, v_r\}$ from each cycle C with anchor node v. S_G is a shortest path tree of G with source s.*

Therefore, we can perform the SSSP algorithm for trees of Theorem 5 on S_G and obtain the following theorem.

▶ **Theorem 17.** *SSSP can be computed in any cactus graph in time $O(\log n)$.*

To compute the diameter, we first perform the algorithm of Lemma 16 with the node that has highest identifier as source s,[6] which yields a shortest path tree S_G. This tree can easily be rooted using Lemma 4. Let $Q(v)$ denote the children of v in S_G, and let $B(v)$ denote the block of node v in G. Using Lemma 7, each node v can compute its height $h(v)$ in S_G and can locally determine the value $m(v) := \max_{u,w \in Q(v), B(u) \neq B(w)} (h(u) + h(w) + w(v,u) + w(v,w))$. We further define the pseudotree Π_C of each cycle C as the graph that contains all edges of

[6] In the NCC_0, this node can be determined by constructing the tree T from Lemma 15 and using Lemma 3 on T.

C and, additionally, an edge $\{v, t_v\}$ for each node $v \neq a_C$ of C, where t_v is a node that is simulated by v, and $w(\{v, t_v\}) = \max_{u \in Q(v) \setminus C}(h(u) + w(\{v, u\}))$. Intuitively, each node v of C that is not the anchor node is attached an edge whose weight equals the height of its subtree in S_G without considering the child of v that also lies in C (if that exists). Then, for each cycle C in parallel, we perform the algorithm of Theorem 11 on Π_C to compute its diameter $D(\Pi_C)$ (using the redistribution framework). We obtain the diameter of G as the value $\hat{D} := \max\{\max_{v \in V}(h(v)), \max_{v \in V}(m(v)), \max_{\text{cycle } C}(D(\Pi_C))\}$. By showing that $\hat{D} = D$, we conclude the following theorem.

▶ **Theorem 18.** *The diameter can be computed in any cactus graph in time $O(\log n)$.*

7 Sparse Graphs

In this final section, we present constant factor approximations for SSSP and the diameter in graphs that contain at most $n + O(n^{1/3})$ edges and that have arboricity at most $O(\log n)$. Our algorithm for such graphs relies on an MST $M = (V, E')$ of G, where $E' \subseteq E$. M can be computed deterministically in time $O(\log^2 n)$ using [16], Observation 4, in a modified way[7].

▶ **Lemma 19.** *The algorithm computes an MST of G deterministically in time $O(\log^2 n)$.*

We call each edge $e \in E \setminus E'$ a *non-tree edge*. Further, we call a node *shortcut node* if it is adjacent to a non-tree edge, and define $\Sigma \subseteq V$ as the set of shortcut nodes. Clearly, after computing M every node $v \in \Sigma$ knows that it is a shortcut node, i.e., if one of its incident edges has not been added to E'. In the remainder of this section, we will compute approximate distances by (1) computing the distance from each node to its closest shortcut node in G, and (2) determining the distance between any two shortcut nodes in G. For any $s, t \in V$, we finally obtain a good approximation for $d(s, t)$ by considering the path in M as well as a path that contains the closest shortcut nodes of both s and t.

Our algorithms rely on a *balanced decomposition tree* T_M, which allows us to quickly determine the distance between any two nodes in G, and which is presented in Section 7.1. In Section 7.2, T_M is extended by a set of edges that allow us to solve (1) by performing a distributed multi-source Bellman-Ford algorithm for $O(\log n)$ rounds. For (2), in Section 7.3 we first compute the distance between any two shortcut nodes in M, and then perform matrix multiplications to obtain the pairwise distances between shortcut nodes in G. By exploiting the fact that $|\Sigma| = O(n^{1/3})$, and using techniques of [4], we are able to distribute the $\Theta(n)$ operations of each of the $O(\log n)$ multiplications efficiently using the global network. In Section 7.4, we finally show how the information can be used to compute 3-approximations for SSSP and the diameter.

For simplicity, in the following sections we assume that M has degree 3. Justifying this assumption, we remark that M can easily be transformed into such a tree while preserving the distances in M. First, we root the tree at the node with highest identifier using Lemma 4. Then, every node v replaces the edges to its children by a binary tree of virtual nodes, where the leaf nodes are the children of v, the edge from each leaf u to its parent is assigned the weight $w(\{v, u\})$, and all inner edges have weight 0.[8] The virtual nodes are distributed evenly among the children of v such that each child is only tasked with the simulation of at most one virtual node. Note that the virtual edges can be established using the local network.

[7] The algorithm of [16] computes a (not necessarily minimum) spanning tree, which would actually already suffice for the results of this paper. However, if G contains edges with exceptionally large weights, an MST may yield much better results in practice.

[8] Note that the edge weights are no longer strictly positive; however, one can easily verify that the algorithms of this section also work with non-negative edge weights.

7.1 Hierarchical Tree Decomposition

We next present an algorithm to compute a hierarchical tree decomposition of M, resulting in a *balanced decomposition tree* T_M. T_M will enable us to compute distances between nodes in M in time $O(\log n)$, despite the fact that the diameter of M may be very high.

Our algorithm constructs T_M as a binary rooted tree $T_M = (V, E_T)$ of height $O(\log n)$ with root $r \in V$ (which is the node that has highest identifier) by selecting a set of global edges E_T. Each node $v \in V$ knows its parent $p_T(u) \in V$. To each edge $\{u, v\} \in E_T$ we assign a weight $w(\{u, v\})$ that equals the sum of the weights of all edges on the (unique) path from u to v in M. Further, each node $v \in V$ is assigned a distinct label $l(v) \in \{0, 1\}^{O(\log n)}$ such that $l(v)$ is a prefix of $l(u)$ for all children u of v in T_M, and $l(r) = \varepsilon$ (the empty word).

From a high level, the algorithm works as follows. Starting with M, within $O(\log n)$ iterations M is divided into smaller and smaller components until each component consists of a single node. More specifically, in iteration i, every remaining component A handles one *recursive call* of the algorithm, where each recursive call is performed independently from the recursive calls executed in other components. The goal of A is to select a *split node* x, which becomes a node at depth $i - 1$ in T_M, and whose removal from M divides A into components of size at most $|A|/2$. The split node x then recursively calls the algorithm in each resulting component; the split nodes that are selected in each component become children of x in T_M.

When the algorithm is called at some node v, it is associated with a *label* parameter $l \in \{0, 1\}^{O(\log n)}$ and a *parent* parameter $p \in V$. The first recursive call is initiated at node r with parameters $l = \varepsilon$ and $p = \emptyset$. Assume that a recursive call is issued at $v \in V$, let A be the component of M in which v lies, and let A_1, A_2 and A_3 be the at most three components of A that result from removing v. Using Lemma 6, every node u in A_1 can easily compute the number of nodes that lie in each of its adjacent subtrees in A_1 (i.e., the size of the resulting components of A_1 after removing u). It is easy to see that there must be a *split node* x_1 in A_1 whose removal divides A_1 into components of size at most $|A|/2$ (see, e.g., [5, Lemma 4.1]); if there are multiple such nodes, let x_1 be the one that has highest identifier. Correspondingly, there are split nodes x_2 in A_2 and x_3 in A_3. v learns x_1, x_2 and x_3 using Lemma 3 and sets these nodes as its children in T_M. By performing the SSSP algorithm of Theorem 5 with source v in A_1, x_1 learns $d_M(x_1, v)$, which becomes the weights of the edge $\{v, x_1\}$ (correspondingly, the edges $\{v, x_2\}$ and $\{v, x_3\}$ are established). To continue the recursion in A_1, x calls x_1 with label parameter $l \circ 00$ and parent parameter v. Correspondingly, x_2 is called with $l \circ 01$, and x_3 with $l \circ 10$.

▶ **Theorem 20.** *A balanced decomposition tree T_M for M can be computed in time $O(\log^2 n)$.*

It is easy to see that one can route a message from any node s to any node t in $O(\log n)$ rounds by following the unique path in the tree from s to t, using the node labels to find the next node on the path. However, the sum of the edge's weights along that path may be higher than the actual distance between s and t in M.

7.2 Finding Nearest Shortcut Nodes

To efficiently compute the nearest shortcut node for each node $u \in V$, we extend T_M to a *distance graph* $D_T = (V, E_D)$, $E_D \supseteq E_T$, by establishing additional edges between the nodes of T_M. Specifically, unlike T_M, the distance between any two nodes in D_T will be equal to their distance in M, which allows us to employ a distributed Bellman-Ford approach.

We describe the algorithm to construct D_T from the perspective of a fixed node $u \in V$. For each edge $\{u, v\} \in E_T$ such that $u = p_T(v)$ for which there does *not* exist a local edge $\{u, v\} \in E'$, we know that the edge $\{u, v\}$ "skips" the nodes on the unique path between

u and v in M. Consequently, these nodes must lie in a subtree of v in T_M. Therefore, to compute the exact distance from u to a skipped node w, we cannot just simply add up the edges in E_T on the path from u to w, as this sum must be larger than the distance $d(u, w)$.

To circumvent this problem, u's goal is to establish additional edges to some of these skipped nodes. Let $x \in V$ be the neighbor of u in M that lies on the unique path from u to v in M. To initiate the construction of edges in each of its subtrees, u needs to send messages to *each* child v in T_M that skipped some nodes (recall that u is able to do so because it has degree 3 in T_M). Such a message to v contains $l(x)$, $l(u)$, $\mathrm{id}(u)$ and $w(\{u, v\})$. Upon receiving the call from u, v contacts its child node y in T_M whose label is a prefix of $l(x)$, forwarding u's identifier, $l(x)$ and the (updated) weight $w(\{y, u\}) = w(\{u, v\}) - w(\{v, y\})$. y then adds the edge $\{y, u\}$ with weight $w(\{y, u\})$ to the set E_D by informing u about it. Then, y continues the recursion at its child in T_M that lies in x's direction, until the process reaches x itself. Since the height of T_M is $O(\log n)$, u learns at most $O(\log n)$ additional edges and thus its degree in D_T is $O(\log n)$.

Note that since the process from u propagates down the tree level by level, we can perform the algorithm at all nodes in parallel, whereby the separate construction processes follow each other in a pipelined fashion without causing too much communication. Together with Theorem 20, we obtain the following lemma.

▶ **Lemma 21.** *The distance graph $D_T = (V, E_D)$ for M can be computed in time $O(\log^2 n)$.*

From the way we construct the node's additional edges in E_D, and the fact that the edges in E_T preserve distances in M, we conclude the following lemma.

▶ **Lemma 22.** *For any edge $\{u, v\} \in E_D$ it holds $w(\{u, v\}) = d_M(u, v)$, where $d_M(u, v)$ denotes the distance between u and v in M.*

The next lemma is crucial for showing the correctness of the algorithms that follow.

▶ **Lemma 23.** *For every $u, v \in V$ we have that (1) every path from u to v in D_T has length at least $d_M(u, v)$, and (2) there exists a path P with $w(P) = d_M(u, v)$ and $|P| = O(\log n)$ that only contains nodes of the unique path from u to v in T_M.*

For any node $v \in V$, we define the *nearest shortcut node* of v as $\sigma(v) = \arg\min_{u \in \Sigma} d(v, u)$. To let each node v determine $\sigma(v)$ and $d(v, \sigma(v))$, we perform a distributed version of the Bellman-Ford algorithm. From an abstract level, the algorithm works as follows. In the first round, every shortcut node sends a message associated with its own identifier and distance value 0 to itself. In every subsequent round, every node $v \in V$ chooses the message with smallest distance value d received so far (breaking ties by choosing the one associated with the node with highest identifier), and sends a message containing $d + w(\{v, u\})$ to each neighbor u in D_T. After $O(\log n)$ rounds, every node v knows the distance $d_M(v, u)$ to its closest shortcut node u in M. Since for any closest shortcut node w in G there must be a shortest path from v to w that only contains edges of M, this implies that u must also be closest to v in G, i.e., $u = \sigma(v)$, and $d_M(v, u) = d(v, \sigma(v))$.

Note that each node has only created additional edges to its descendants in T_M during the construction of D_T, therefore the degree of D_T is $O(\log n)$ and we can easily perform the algorithm described above using the global network.

▶ **Lemma 24.** *After $O(\log n)$ rounds, each node $v \in V$ knows $id(u)$ of its nearest shortcut node $\sigma(v)$ in G and its distance $d(v, \sigma(v))$ to it.*

7.3 Computing APSP between Shortcut Nodes

In this section, we first describe how the shortcut nodes can compute their pairwise distances in M by using D_T. Then, we explain how the information can be used to compute all pairwise distances between shortcut nodes in G by performing matrix multiplications.

Compute Distances in M. First, each node learns the total number of shortcut nodes $n_c := |\Sigma|$, and each shortcut node is assigned a unique identifier from $[n_c]$.[9] The first part can easily achieved using Lemma 3. For the second part, consider the Patricia trie P on the node's identifiers, which, since each node knows all identifiers, is implicitly given to the nodes. By performing a convergecast in P (where each inner node is simulated by the leaf node in its subtree that has highest identifier), every inner node of P can learn the number of shortcut nodes in its subtree in P. This allows the root of P to assign intervals of labels to its children in P, which further divide the interval according to the number of shortcut nodes in their children's subtrees, until every shortcut node is assigned a unique identifier.

Note that it is impossible for a shortcut node to explicitly learn all the distances to all other shortcut nodes in polylogarithmic time, since it may have to learn $\Omega(n^{1/3})$ many bits. However, if we could distribute the distances of all $O(n^{2/3})$ pairs of shortcut nodes uniformly among all nodes of V, each node would only have to store $O(\log n)$ bits[10]. We make use of this in the following way. To each pair (i, j) of shortcut nodes we assign a *representative* $h(i, j) \in V$, which is chosen using (pseudo-)random hash function $h : [n_c]^2 \to V$ that is known to all nodes and that satisfies $h(i, j) = h(j, i)$.[11] The goal of $h(i, j)$ is to infer $d_M(i, j)$ from learning all the edges on the path from i to j in D_T.

Due to space reasons, we defer a precise description to the full version. From a high level, each $h(i, j)$ first needs to retrieve the labels of both i and j in T_M, which it cannot do directly, as the nodes may be contacted by many other nodes. Instead, we use techniques from [4] to distribute the load: $h(i, j)$ participates in the construction of a *multicast tree* towards both i and j. Using randomization, these trees can be used to disseminate information from each node to all nodes in its multicast tree in a broadcast fashion with low congestion rather than communicating directly. Afterwards, $h(i, j)$ can infer the labels of all nodes on the path from i to j in D_M, and learn their edge weights in a very similar way. By observing that each node only has to learn $O(\log n)$ values, we can use Theorems 2.3 and Theorem 2.4 of [4] in a straight-forward manner to obtain the following lemma.

▶ **Lemma 25.** *Every representative $h(i, j)$ learns $d_M(i, j)$ in time $O(\log n)$, w.h.p.*

Compute Distances in G. Let $A \in \mathbb{N}_0^{n_c \times n_c}$ be the *distance matrix* of the shortcut nodes, where $A_{i,j} = \min\{w(\{i, j\}), d_M(i, j)\}$, if $\{i, j\} \in E$, and $A_{i,j} = d_M(i, j)$, otherwise. Our goal is to square A for $\lceil \log n \rceil + 2$ many iterations in the *min-plus semiring*. More precisely, we define $A^1 = A$, and for $t \geq 1$ we have that $A^{2^t}_{i,j} = \min_{k \in [n_c]}(A^{2^{t-1}}_{i,k} + A^{2^{t-1}}_{k,j})$. The following lemma shows that after squaring the matrix $\lceil \log n \rceil + 2$ times, its entries give the distances in G.

[9] We denote $[k] = \{0, \ldots, k - 1\}$.

[10] In fact, for this we could even allow n pairs, i.e., $n_c = O(\sqrt{n})$; the reason for our bound on n_c will become clear later.

[11] Note that sufficient shared randomness can be achieved in our model by broadcasting $\Theta(\log^2 n)$ random bits in time $O(\log n)$ [4]. Further, note that for a node $v \in V$ there can be up to $O(\log n)$ keys (i, j) for which $h(i, j) = v$, w.h.p., thus v has to act on behalf of at most $O(\log n)$ nodes.

▶ **Lemma 26.** *$A_{i,j}^{2^{\lceil \log n \rceil + 2}} = d(i,j)$ for each $i,j \in \Sigma$.*

We now describe how the matrix can efficiently be multiplied. As an invariant to our algorithm, we show that at the beginning of the t-th multiplication, every representative $h(i,j)$ stores $A_{i,j}^{2^{t-1}}$. Thus, for the induction basis we first need to ensure that every representative $h(i,j)$ learns $A_{i,j}$. By Lemma 25, $h(i,j)$ already knows $d_M(i,j)$, thus it only needs to retrieve $w(\{i,j\})$, if that edge exists. To do so, we first compute an orientation with outdegree $O(\log n)$ in time $O(\log n)$ using [7, Corollary 3.12] in the local network. For every edge $\{i,j\}$ that is directed from i to j, i sends a message containing $w(\{i,j\})$ to $h(i,j)$; since the arboricity of G is $O(\log n)$, every node only has to send at most $O(\log n)$ messages.

The t-th multiplication is then done in the following way. We use a (pseudo-)random hash function $h : [n_c]^3 \to V$, where $h(i,j,k) = h(j,i,k)$. First, every node $h(i,j,k) \in V$ needs to learn $A_{i,j}^{2^{t-1}}$.[12] To do so, $h(i,j,k)$ joins the multicast group of $h(i,j)$ using [4, Theorem 2.3]. With the help of [4, Theorem 2.4], $h(i,j)$ can then multicast $A_{i,j}^{t-1}$ to all $h(i,j,k)$. Since there are $L \leq [n_c]^3 = O(n)$ nodes $h(i,j,k)$ that each join a multicast group, and each node needs to send and receive at most $\ell = O(\log n)$ values, w.h.p., the theorems imply a runtime of $O(\log n)$, w.h.p.

After $h(i,j,k)$ has received $A_{i,j}^{2^{t-1}}$, it sends it to both $h(i,k,j)$ and $h(j,k,i)$. It is easy to see that thereby $h(i,j,k)$ will receive $A_{i,k}^{2^{t-1}}$ from $h(i,k,j)$ and $A_{k,j}^{2^{t-1}}$ from $h(k,j,i)$. Afterwards, $h(i,j,k)$ sends the value $A_{i,k}^{2^{t-1}} + A_{k,j}^{2^{t-1}}$ to $h(i,j)$ by participating in an aggregation using [4, Theorem 2.2] and the minimum function, whereby $h(i,j)$ receives $A_{i,j}^{2t}$. By the same arguments as before, $L = O(n)$, and $\ell = O(\log n)$, which implies a runtime of $O(\log n)$, w.h.p.

▶ **Lemma 27.** *After $\lceil \log n \rceil + 2$ many matrix multiplications, $h(i,j)$ stores $d(i,j)$ for every $i,j \in [n_c]$. The total number of rounds is $O(\log^2 n)$, w.h.p.*

7.4 Approximating SSSP and the Diameter

We are now all set in order to compute approximate distances between any two nodes $s,t \in V$. Specifically, we approximate $d(s,t)$ by

$$\widetilde{d}(s,t) = \min\{d_M(s,t), d(s,\sigma(s)) + d(\sigma(s),\sigma(t)) + d(\sigma(t),t)\}.$$

We now show that $\widetilde{d}(s,t)$ gives a 3-approximation for $d(s,t)$.

▶ **Lemma 28.** *Let $s,t \in V$ and $d(s,t)$ be the length of the shortest path from s to t. It holds that $d(s,t) \leq \widetilde{d}(s,t) \leq 3d(s,t)$.*

To approximate SSSP, every node v needs to learn $\widetilde{d}(s,v)$ for a given source s. To do so, the nodes first have to compute $d_M(s,v)$, which can be done in time $O(\log n)$ by performing SSSP in M using Theorem 5. Then, the nodes construct D_T in time $O(\log^2 n)$ using Lemma 21. With the help of D_T and Lemma 24, s can compute $d(s,\sigma(s))$, which is then broadcast to all nodes in time $O(\log n)$ using Lemma 3. Then, we compute all pairwise distances in G between all shortcut nodes in time $O(\log^2 n)$, w.h.p., using Lemma 27; specifically, every shortcut node v learns $d(\sigma(s),v)$. By performing a slight variant of the algorithm of Lemma 24, we can make sure that every node t not only learns its closest shortcut node $\sigma(t)$ in M, but also retrieves $d(\sigma(s),\sigma(t))$ from $\sigma(t)$ within $O(\log n)$ rounds. Since t is now able to compute $\widetilde{d}(s,t)$, we conclude the following theorem.

[12] We will again ignore the fact that a node may have to act on behalf of at most $O(\log n)$ nodes $h(i,j,k)$.

▶ **Theorem 29.** *3-approximate SSSP can be computed in graphs that contain at most $n + O(n^{1/3})$ edges and have arboricity $O(\log n)$ in time $O(\log^2 n)$, w.h.p.*

For a 3-approximation of the diameter, consider $\widetilde{D} = 2\max_{s\in V} d(s,\sigma(s)) + \max_{x,y\in\Sigma} d(x,y)$. $\widetilde{D}$ can easily be computed using Lemmas 21, 24, and 27, and by using Lemma 3 on M to determine the maxima of the obtained values. By the triangle inequality, we have that $D \leq \widetilde{D}$. Furthermore, since $d(s,\sigma(s)) \leq D$ and $\max_{x,y\in\Sigma} d(x,y) \leq D$, we have that $\widetilde{D} \leq 3D$.

▶ **Theorem 30.** *A 3-approximation of the diameter can be computed in graphs that contain at most $n + O(n^{1/3})$ edges and have arboricity $O(\log n)$ in time $O(\log^2 n)$, w.h.p.*

References

1 Dana Angluin, James Aspnes, Jiang Chen, Yinghua Wu, and Yitong Yin. Fast Construction of Overlay Networks. In *Proc. of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–154, 2005.

2 Mikhail Atallah and Uzi Vishkin. Finding euler tours in parallel. *J. of Computer and System Sciences*, 29(3):330–337, 1984.

3 John Augustine, Keerti Choudhary, Avi Cohen, David Peleg, Sumathi Sivasubramaniam, and Suman Sourav. Distributed graph realizations, 2020. `arXiv:2002.05376`.

4 John Augustine, Mohsen Ghaffari, Robert Gmyr, Kristian Hinnenthal, Fabian Kuhn, Jason Li, and Christian Scheideler. Distributed computation in node-capacitated networks. In *Proc. of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 69–79, 2019.

5 John Augustine, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, and Philipp Schneider. Shortest paths in a hybrid network model. In *Proc. of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1280–1299, 2020.

6 John Augustine and Sumathi Sivasubramaniam. Spartan: A framework for sparse robust addressable networks. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1060–1069, 2018.

7 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.

8 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007.

9 Boaz Ben-Moshe, Amit Dvir, Michael Segal, and Arie Tamir. Centdian computation in cactus graphs. *J. of Graph Algorithms and Applications*, 16(2):199–224, 2012.

10 Tao Chen, Xiaofeng Gao, and Guihai Chen. The features, hardware, and architectures of data center networks: A survey. *J. of Parallel and Distributed Computing*, 96:45–74, 2016.

11 Yong Cui, Hongyi Wang, and Xiuzhen Cheng. Channel allocation in wireless data center networks. In *Proc. of IEEE INFOCOM*, pages 1395–1403, 2011.

12 Hristo N. Djidjev, Grammati E. Pantziou, and Christos D. Zaroliagis. Computing shortest paths and distances in planar graphs. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 327–338, 1991.

13 Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proc. of the ACM SIGCOMM 2010 conference*, pages 339–350, 2010.

14 Michael Feldmann, Kristian Hinnenthal, and Christian Scheideler. Fast hybrid network algorithms for shortest paths in sparse graphs, 2020. `arXiv:2007.01191`.

15 Jie Gao and Li Zhang. Well-separated pair decomposition for the unit-disk graph metric and its applications. *SIAM J. Comput.*, 35(1):151–169, 2005.

16 Robert Gmyr, Kristian Hinnenthal, Christian Scheideler, and Christian Sohler. Distributed monitoring of network properties: The power of hybrid networks. In *Proc. of the 44th International Colloquium on Algorithms, Languages, and Programming (ICALP)*, pages 137:1–137:15, 2017.

17 Thorsten Götte, Kristian Hinnenthal, and Christian Scheideler. Faster construction of overlay networks. In *International Colloquium on Structural Information and Communication Complexity*, pages 262–276. Springer, 2019.

18 Thorsten Götte, Kristian Hinnenthal, Christian Scheideler, and Julian Werthmann. Time-optimal construction of overlay networks, 2020. `arXiv:2009.03987`.

19 Joseph JaJa. *An Introduction to Parallel Algorithms*, volume 17. Addison Wesley, 1992.

20 Daniel Jung, Christina Kolb, Christian Scheideler, and Jannik Sundermeier. Competitive routing in hybrid communication networks. In *ALGOSENSORS*, volume 11410, pages 15–31, 2018.

21 Haim Kaplan, Wolfgang Mulzer, Liam Roditty, and Paul Seiferth. Routing in unit disk graphs. *Algorithmica*, 80(3):830–848, 2018.

22 Udit Narayana Kar and Debarshi Kumar Sanyal. An overview of device-to-device communication in cellular networks. *ICT Express*, 4(3):203–208, 2018.

23 Fabian Kuhn and Philipp Schneider. Computing shortest paths and diameter in the hybrid network model. In *2020 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 109–118, 2020.

24 Yu-Feng Lan and Yue-Li Wang. An optimal algorithm for solving the 1-median problem on weighted 4-cactus graphs. *European Journal of Operational Research*, 122(3):602–610, 2000.

25 Jason Li. Faster parallel algorithm for approximate shortest path. In *Proc. of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 308–321, 2020.

26 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proc. of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 565–573, 2014.

27 C. St. J. A. Nash-Williams. Decomposition of Finite Graphs Into Forests. *J. of the London Mathematical Society*, 39(1):12–12, 1964.

28 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

29 Peter Robinson. Being fast means being chatty: The local information cost of graph spanners, 2020. `arXiv:2003.09895`.

30 Michael Rossberg and Guenter Schaefer. A survey on automatic configuration of virtual private networks. *Computer Networks*, 55(8):1684–1699, 2011.

31 Robert E. Tarjan and Uzi Vishkin. An Efficient Parallel Biconnectivity Algorithm. *SIAM J. on Computing*, 14(4):862–874, 1985.

32 Anis Tell, Wale Babalola, George Kalebiala, and Krishna Chinta. Sd-wan: A modern hybrid-wan to enable digital transformation for businesses. *IDC White Paper*, April 2018.

33 Jeffrey D. Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM J. on Computing*, 20(1):100–125, 1991.