Distributed Treewidth Computation

Jason Li

1 Carnegie Mellon University, jmli@cs.cmu.edu

Abstract

Of all the restricted graph families out there, the family of low treewidth graphs has continuously proven to admit many algorithmic applications. For example, many NP-hard algorithms can be solved in polynomial time on graphs of constant treewidth. Other algorithmic techniques, such as Baker’s technique, partition the graph into components of low treewidth. Therefore, computing the treewidth of a graph remains an important problem in algorithm design. For graphs of constant treewidth, linear-time algorithms are known in the classical setting, and well as polylog(n)-time parallel algorithms for computing an O(1)-approximation to treewidth. However, nothing is yet known in the distributed setting.

In this paper, we give near-optimal algorithms for computing the treewidth on a distributed network. We show that for graphs of constant treewidth, an O(1)-approximation to the treewidth can be computed in near-optimal ˜O(D) time, where D is the diameter of the network graph. In addition, we show that many NP-hard problems that are tractable on constant treewidth graphs can also be solved in ˜O(D) time on a distributed network of constant treewidth.

Our algorithms make use of the shortcuts framework of Ghaffari and Haeupler [SODA’16], which has proven to be a powerful tool in designing near-optimal distributed algorithms for restricted graph networks, such as planar graphs, low-treewidth graphs, and excluded minor graphs.

1 Introduction

In recent years, there has been an increasing interest in fast distributed algorithms on restricted graph families. Part of this recent action stemmed from the widespread lower bound of ˜Ω(√n + D) for distributed algorithms on general graphs [5], which holds for many basic graph optimization problems. By restricting the graph networks to exclude the pathological lower bound instances in [5], researchers have found success in beating the lower bound on nontrivial families of graph networks. For example, there are now distributed MST algorithms running in near-optimal O(D · n^{o(1)}) time on planar graphs, bounded treewidth graphs, and graphs with small mixing time [7, 10, 11, 8]. Adding onto this line of work, this paper investigates many algorithmic problems on graphs networks of bounded treewidth and gives efficient distributed algorithms running in near-optimal ˜O(D) rounds on these networks, where D is the diameter of the network graph.

The concept of treewidth, which dates back to the study of graph minors of Robertson and Seymour [18], has proven fruitful in the quest for efficient classical algorithms for computationally intractable problems. For problems of bounded treewidth, many difficult, NP-hard problems can be solved in polynomial time. Then, with the increasing popularity of the parallel PRAM model, the classical bounded treewidth algorithms were adapted to run in parallel [15, 16]. However, until this paper, nothing was yet known in the distributed setting.

1.1 Results

Our main result is a distributed O(1)-approximation algorithm to compute the treewidth of a network graph in ˜O(k^{O(k)}D) rounds of the CONGEST model, where k is the treewidth of the

1 We use ˜O(·) notation to hide polylogarithmic factors in n.
network graph $G$. To state this result in an approximation setting, we say that the algorithm distinguishes the instances where $\text{tw}(G) > k$ and the instances $\text{tw}(G) \leq O(k)$.

**Theorem 1.** Given a network graph $G$ and integer $k$, there is a distributed algorithm running in $O(k^{O(1)}D)$ rounds and either correctly concludes that $\text{tw}(G) > k$, or correctly concludes that $\text{tw}(G) \leq 7k + 4$. Every node in the network should know the conclusion of the algorithm.

Of course, to approximate the treewidth of a network graph $G$, we simply run the above algorithm with increasing values of $k = 1, 2, 3, \ldots$ until the algorithm outputs “$\text{tw}(G) \leq 7k + 4$”; the running time will be dominated by the last, successful $k$. Also, observe that the diameter factor, $D$, is necessary in the running time. Intuitively, this is because treewidth is a global property of a graph; in other words, one cannot say anything about the treewidth by only looking at a local neighborhood around a vertex. For example, given any approximation factor $\alpha$, consider the network graph consisting of a path of length $\Omega(n)$ with a clique of size $2\alpha$ attached to one end. For the information of the $2\alpha$-clique to reach the node $v$ on the other end of the path, the number of rounds required is at least the length of the path, which is $\Omega(n)$. Before the $2\alpha$-clique reaches node $v$, the only nodes that $v$ can possibly learn in the network form a path, which has treewidth 1. Therefore, for node $v$ to distinguish between treewidth 1 (the path without the clique) and treewidth $2\alpha - 1$ (the path with the clique attached), $\Omega(n)$ rounds are needed. Hence, to obtain an $\alpha$-approximation, $\Omega(n) = O(D)$ rounds are necessary.

Our distributed algorithm follows the outline of the parallel algorithm of Lagergren [15], which approximates the treewidth in $k^{O(1)}\text{polylog}(n)$ parallel time. The algorithm of [15] makes repeated calls to an algorithm that finds vertex disjoint paths between two given vertices. Our main technical contribution is a distributed algorithm solving this vertex disjoint paths problem on a graph network of treewidth $k$ in $O(k^{O(1)}D)$ time. This algorithm resembles the parallel vertex disjoint paths problem [14], but new ideas are required to construct a distributed algorithm. Our main insight is in viewing the algorithm of [14] in a graph contraction-based setting, and then applying the recent technology of partwise aggregation in distributed computing [7,9,14]. Outside of this subroutine, we adapt the framework of [15], which computes a treewidth decomposition given this subroutine, to the distributed setting.

**Lemma 2.** Given a graph $G = (V,E)$ of treewidth at most $k$ and two vertices $s,t \subseteq V$, we can either find $k$ vertex-disjoint $s-t$ paths, or output an $s-t$ node cut of size less than $k$, in $O(k^{O(1)}D)$ rounds. In the former case, every node knows whether it is on a path, and if so, its predecessor and successor on that path. In the latter case, every node knows the fact that $k$ vertex-disjoint paths do not exist, as well as whether it is in the node cut.

Perhaps more importantly, the algorithm of Theorem 1 also outputs a distributed version of a treewidth decomposition. Using this decomposition, we can solve many computationally difficult problems on bounded treewidth graph networks like in the classical setting.

**Theorem 3.** Let $G$ be a graph network with treewidth $k$. The problems maximum independent set, minimum vertex cover, chromatic number, and minimum dominating set can be solved in $O(k^{O(1)}D)$ rounds on network $G$.

We remark that we can extend Theorem 3 to solve many other optimization problems that are tractable on bounded-treewidth graphs.

### 1.2 Related Work

The shortcuts framework was introduced by Ghaffari and Haeupler [7], who used it to solve MST and $(1 + \epsilon)$-approximate minimum cut on planar graphs in near-optimal $O(D)$ time. This framework was expanded on in [10, 11], generalizing these algorithms to run on bounded-genus
and bounded-treewidth graphs in $\tilde{O}(D)$ time. Lately, [13] studied the shortcuts framework on minor-free graphs, leading to $\tilde{O}(D^2)$-round distributed algorithms for these problems on graphs excluded a fixed minor.

Efficient algorithms that do not use the shortcuts framework also exist. Recently, Ghaffari et al. [8] give a distributed algorithm for MST in time proportional to the mixing time of the network graph. For well-mixing graphs, such as expanders and random graphs, the algorithm runs in $2^{O(\sqrt{\log n \log \log n})}$ time.

The graph-theoretic property treewidth was introduced in Robertson and Seymour’s study of graph minors [15]. Since then, it has seen many algorithmic applications in solving NP-hard problems, such as maximum independent set and chromatic number, efficiently on bounded treewidth graphs; for an extensive study, see [17].

The algorithmic problem of computing or approximating the treewidth of a graph has also been studied extensively. Computing the treewidth exactly is NP-hard [2], but admits an approximate algorithm in polynomial time [1, 6], where $k$ is the treewidth of the input graph. For small values of $k$, faster algorithms were known since the work of Robertson and Seymour [15]. Bodlaender [3] gave the first linear-time algorithm for fixed $k$, running in $2^{O(k)} n$ time. In the parallel setting, computing an $O(1)$-approximation of the treewidth can be done in $k^{O(k)} \log^3(n)$ time using $O(n)$ processors [15] [10].

## 2 Preliminaries

All of our algorithms work under the CONGEST model of distributed computing. There is a network $G = (V, E)$ of $n$ nodes and diameter $D$. On each synchronous round, every node can send an $O(\log n)$-bit message to each of its neighbors in $G$, possibly a different message to each neighbor. We assume that between synchronous rounds, every node can perform unbounded local computation.

Throughout the paper, $G$ will always represent the graph network, and $D$ its diameter. Given any graph $H$, let $V(H)$ and $E(H)$ denote the vertices and edges of $H$. For a vertex $v \in V$, denote $N(v)$ as the neighbors of $v$ in $G$. For a vertex set $S \subseteq V$, denote $N(S)$ as the neighbor of set $S$ in $G$, i.e., the vertices in $V - S$ with a neighbor in $S$, and define $N[S] := N(S) \cup S$. A path in $H$ is a sequence of vertices such that adjacent vertices are neighbors in $H$. A path is simple if no vertex appears twice on the path. For vertices $s, t \in V$, an $s-t$ path is a (not necessarily simple) path connecting $s$ and $t$, and a vertex set $S$ is called a vertex cut if it intersects every $s-t$ path in $G$.

Lastly, for a positive integer $r$, let $[r]$ denote the set $\{1, 2, \ldots, r\}$ of positive integers from 1 to $r$.

### 2.1 Partwise and Subgraph Aggregation

The shortcuts framework of Ghaffari and Haeupler [7] has proved fruitful in designing distributed algorithms on restricted graph families. The inner workings of shortcuts is not necessary for the scope of this paper. Rather, we abstract out the primary task that is accomplished through the shortcuts framework, which we define as Partwise Aggregation (PA), following [9].

**Definition 4 (Partwise Aggregation [9]).** Let $G = (V, E)$ be a network graph, and let $\mathcal{P} = (P_1, \ldots, P_r)$ be a collection of pairwise disjoint and connected subsets $P_i \subseteq V$, called parts. For each part $P_i$, every node $v \in P_i$ knows the set $N(v) \cap P_i$, i.e., which of its neighbors belong to its part. Suppose that every node $v \in \bigcup_i P_i$ has an integer $x_v$ of $O(\log n)$ bits, and let $\oplus$ be an associative function operating on integers of length $O(\log n)$. Every node in $P_i$ wants to learn the value $\bigoplus_{v \in P_i} x_v$, i.e., the aggregate $\oplus$ of all of the values $x_v$ in $P_i$. We call such a task **partwise aggregation** with operator $\oplus$.
The power of the shortcuts framework is that it allows us to solve this partwise aggregation (PA) problem quickly, especially if the network graph $G$ has special structure. We provide an intuitive description below, referring the reader to [7] for more details. In an ideal case, if for every part $P_i$, the diameter of $G[P_i]$ is $O(D)$, then every part $P_i$ can simply aggregate inside $G[P_i]$ in $O(D)$ rounds. The trouble is when the diameter of $G[P_i]$ much larger than $D$, the diameter of $G$. The shortcuts framework resolves this issue by allowing these “long and skinny” parts to borrow edges from the rest of the network $G$ to aid in their partwise aggregation. For a part $P_i$, the borrowed edges $H_i$ should satisfy the property that the diameter of $G[P_i] \cup H_i$ is now comparable to the diameter of $G$. On the other hand, it is not ideal for a single edge in $G$ to be borrowed by too many parts, since it would induce “congestion” along the edge. The shortcuts framework resolves this issue by allowing these “long and skinny” parts to borrow edges from the rest of the network $G$ to aid in their partwise aggregation. For a part $P_i$, the borrowed edges $H_i$ should satisfy the property that the diameter of $G[P_i] \cup H_i$ is now comparable to the diameter of $G$.

A recent line of work [10, 11, 13] has built on the initial shortcuts framework of [7], leading to near-optimal PA algorithms for many special classes of graphs.

\textbf{Theorem 5} ([7,10,11,13]). For any associative operator $\oplus$, we can solve the partwise aggregation problem in $\tilde{O}(Q_G)$ rounds, where $Q_G$ is a parameter that depends on the graph $G$ and its diameter $D$, as follows:

1. For all graphs $G$, $Q_G = O(\sqrt{n} + D)$.
2. If $G$ has genus $g$, $Q_G = O(\sqrt{g + D})$.
3. If $G$ has treewidth $k$, $Q_G = O(kD)$.
4. If $G$ excludes a fixed minor $H$, $Q_G = \tilde{O}(f(H) \cdot D^2)$, where $f$ is a function depending only on $H$.

We will define a PA round to be one iteration of PA, where every node participating in PA initially knows its part ID, its value $v$, and the common operator $\oplus$, and at the end, every node learns the aggregate $\oplus$ of its part. Observe that the well-studied broadcast procedure can be formulated as a PA problem: if a leader node $v$ in a part $P_i$ wants to broadcast its value $x$, then we set $x_u \leftarrow x$, $x_v \leftarrow -\infty$ (more precisely, some $O(\log n)$-bit encoding of $-\infty$) for all $u \in P_i - v$, and $\oplus$ to be the max function.

The PA round assumes that every node knows the ID of its part. Often, we will not have this luxury: each node does not know its part ID, but only which of its neighbors also belong to its part, and in some cases, only a subset of this. Below, we formulate an aggregation task with this weaker assumption, and show that it can be solved using $O(\log n)$ iterations of PA as defined in Definition 4.

\textbf{Definition 6 (Subgraph Aggregation).} Let $G = (V, E)$ be a network graph, let $\mathcal{P} = (P_1, \ldots, P_{|\mathcal{P}|})$ be a collection of parts, and for each $P_i \in \mathcal{P}$, let $H_i$ be a connected subgraph of $G$ on the nodes in $P_i$, not necessarily the induced graph $G[P_i]$. Suppose that for each subgraph $H_i$, every node in $V(H_i)$ knows its neighbors in the subgraph $H_i$ and nothing else. Suppose that every node $v \in \bigcup P_i$ has an integer $x_v$ of $O(\log n)$ bits, and let $\oplus$ be an associative function operating on integers of length $O(\log n)$. Every node in $P_i$ wants to learn the value $\bigoplus_{v \in P_i} x_v$, i.e., the aggregate $\oplus$ of all of the values $x_v$ in $P_i$. We call such a task subgraph aggregation with operator $\oplus$.

Likewise, we define SA round to be one iteration of subgraph aggregation (SA), where every node $v \in P_i$ knows its neighbors in $H_i$, its value $x_v$, and the common operator $\oplus$, and at the end, learns the aggregate $\oplus$ of its part. The following lemma shows that while SA has a weaker assumption, it is no harder than partwise aggregation modulo an $O(\log n)$ factor. While this result has been implied in the literature, e.g., in [12], this is the first time it has been stated explicitly. We defer the proof of the following lemma to Appendix A.
Lemma 7. One SA round can be solved in $O(\log n)$ PA rounds.

Combining Lemma 7 with Theorem 5 gives the following corollary for treewidth $k$ graphs, which is the result we will use in this paper. Since SA solves a stronger problem than PA or broadcast, we will only use the term SA for the rest of the paper. In fact, the following corollary will be the only result in this section that is used for the rest of the paper.

Corollary 8. On a graph network of treewidth $k$, one SA round can be solved in $\tilde{O}(kD)$ rounds.

2.2 SA Helper Routines

Our main algorithms will use the following routines, all of which reduce to computing $O(\log n)$ rounds of SA. The proof of the statement below is deferred to Appendix A.

Lemma 9 (Spanning Tree). Given a connected subgraph $H \subseteq G$ of the network graph, we can compute a spanning tree of $G$ in $O(\log n)$ SA rounds. Every node knows its neighbors of the spanning tree.

Lemma 10 (Rooted Tree Aggregation). Consider a tree $T$ in $G$. Given a root $v_r \in V(T)$, we can compute the tree $T$ rooted at $v_r$ in $O(\log n)$ SA rounds, so that every node in $V(T) - v_r$ knows its parent in the tree $T$ rooted at $v_r$. Moreover, if each node $v_i$ knows an integer $x_i$, and a common associative operator $\oplus$, then we can let each node $v_i$ learn the subtree aggregate $\bigoplus_{j \in T(v_i)} x_j$, where $T(v_i)$ is the subtree rooted at $v_i$, i.e., all nodes in $T$ whose path to the root contains $v_i$.

Lemma 11 (Path Aggregation). Consider a directed path $P = \{v_1, \ldots, v_\ell\}$ in $G$, where each node $v_i$ knows its predecessor and successor on the path. In $O(\log n)$ SA rounds, each node $v_i$ can learn the value of $i$, its index in the path. Moreover, if each node $v_i$ knows an integer $x_i$ and a common associative operator $\oplus$, we can let each node $v_i$ learn the prefix aggregate $\bigoplus_{j \leq i} x_j$ and suffix aggregate $\bigoplus_{j \geq i} x_j$.

Lemma 12 ($s$→$t$ Path). Given a connected subgraph $H \subseteq G$ and two vertices $s, t \in V$, we can compute a directed $s$→$t$ path in $G$ in $O(\log n)$ SA rounds. Every node knows whether it is on the path, and if so, its predecessor and successor nodes on that path.

3 The Vertex-disjoint Paths Algorithm

This section is devoted to proving the following lemma. It is the most technical section of the paper, as well as our main technical contribution.

Lemma 13. Given a graph $G = (V,E)$ of treewidth at most $k$ and two vertices $s, t \subseteq V$, we can either find $k$ vertex-disjoint $s$→$t$ paths, or output an $s$→$t$ node cut of size less than $k$, in $\tilde{O}(k^{O(1)}D)$ rounds. In the former case, every node knows whether it is on a path, and if so, its predecessor and successor on that path. In the latter case, every node knows the fact that $k$ vertex-disjoint paths do not exist, as well as whether it is in the node cut.

When we are talking about graph algorithms in general, not necessarily in the distributed setting, we will use the term vertices. When talking about actual nodes in a distributed network, we will use the term nodes. This is to distinguish between graphs (in the graph-theoretic sense) and the physical graph network. Often times, it is simpler to first explain an algorithm in a classical setting, and then adapt it to run on a distributed network. For most of this section, we will take this approach, explaining our distributed implementation in gray boxes.
Before introducing our distributed algorithm, we first make one important transformation of the network graph that will be useful later on. Define $G^\ell$ to be the following graph: for each node $v \in G$, we add $r$ corresponding nodes $v^1, \ldots, v^r$ in $G^\ell$. For each $v \in G$, we connect the nodes $v^1, \ldots, v^r$ in a clique $K_r$, and for each edge $(u, v)$ in $G$, we connect the node set $u^1, \ldots, u^r$ and the node set $v^1, \ldots, v^r$ with a complete bipartite graph $K_{r,r}$.

We would like to *simulate* the network $G^\ell$ using the network $G$. In particular, we show how to simulate a $T$-round algorithm on the network $G^\ell$ in $O(\ell^2 T)$ rounds on the original network $G$.

We let every node $v \in V(G)$ in the original network *simulate* the nodes $v^1, \ldots, v^r$ in $G^\ell$. That is, node $v$ performs the computations that nodes $v^1, \ldots, v^r$ perform in the algorithm on $G^\ell$. Observe that two nodes $v^i, v^j \in V(G^\ell)$ can communicate in $G^\ell$ if either $u = v$, or the nodes $u, v \in V(G)$ that simulate them can communicate in $G$. If $u = v$, then since the same node $u$ simulates $v^i$ and $v^j$, no communication between nodes in $G$ is needed. If $u \neq v$, then the edge $(u, v) \in E(G)$ in the original network is responsible for $\ell^2$ communicating edges in $G^\ell$, namely the edges $(v^i, v^j)$ for $i, j \in [\ell]$. Therefore, for each round of the algorithm on $G^\ell$, we can take $\ell^2$ rounds to pass the at most $\ell^2$ messages in $G^\ell$ through the edge $(u, v) \in E(G)$.

It is easy to see that the diameter of $G^\ell$ is also $D$. The claim below, whose proof is deferred to Appendix B, bounds the treewidth of $G^\ell$. By Corollary 8 one SA round on $G^\ell$ can be solved in $\tilde{O}(k\ell D)$ rounds on $\tilde{O}(k\ell D)$, and the above argument shows that it can be simulated in $\tilde{O}(k\ell^3 D)$ rounds on $G$.

**Claim 14.** If $G$ has treewidth $k$, then $G^\ell$ has treewidth $O(\ell k)$.

For the rest of this section, we will always either run our distributed algorithm directly on $G$, or run it first on $G^\ell$ for some $\ell$ and then simulate it on $G$. In the latter case, we will simply state that the algorithm is run on $G^\ell$; simulating it on $G$ is implied.

The algorithm models off of the one of [14] for approximating the treewidth in a parallel model. However, new ideas are required to adapt the algorithm in a distributed model. In particular, as we will see, we need to adopt a *graph contraction-based approach* to support the use of SA.

The algorithm is iterative and represents the original Ford-Fulkerson algorithm for maximum flow. It maintains a set of disjoint $s-t$ paths, and, on each iteration, either increases the number of disjoint paths by one through an augmenting path step, or certifies that it is not possible to reach $k$ disjoint paths.

On the first iteration, the algorithm simply needs to find a single $s-t$ path. This can be done by Lemma 12. On a general iteration, for the rest of this section, we assume that the algorithm knows $r$ vertex-disjoint paths for $1 \leq r < k$, and needs to find $r + 1$ vertex-disjoint paths.

In the distributed setting, we assume that each node in a path knows its neighbor(s) on the path. We now make each node on a path learn its index on the path; this can be accomplished with Lemma 11 (on network $G$). In addition, we would like to assign each of the $r$ paths a unique path ID in $[r]$, known to all nodes in the path. This can be accomplished on network $G$ as follows: first, each node sets $x_v \leftarrow ID(v)$ if $v$ belongs in a path, or $-\infty$ otherwise. Then, we compute the maximum $x_v$ inside $G$ using SA. The (unique) node $u$ with $x_v = \max_{x_v}$ notifies this event to all nodes in its path $P$ using SA. We assign path $P$ the ID $r$, known to all its nodes. Then, all nodes in path $P$ drop out of the future path ID computations (i.e., they set $x_v \leftarrow -\infty$ from now on). There are $r-1$ paths left; we iterate with $r \leftarrow r - 1$ until we are done.
3.1 The Residual Graph

Recall the setting: the algorithm knows \( r \) vertex-disjoint paths and tries to find \( r + 1 \) vertex-disjoint paths. Our algorithm maintains a reachability graph, a directed graph with a source \( s \) and sink \( t \), such that there is an augmenting path in \( G \) iff \( t \) is reachable from \( s \) in this directed graph. Its construction is directly modeled off of the residual graph from the Ford-Fulkerson algorithm and is similar to the “graph decomposition into bridges” in [14]. In general, directed reachability is a hard problem in distributed models and even parallel models, but we will exploit the special structure of the residual graph in the \( k \)-vertex disjoint paths problem in order to compute \( s \to t \) reachability efficiently.

Construction of the reachability graph. The steps in our construction are illustrated in Figure 1. First, we construct a directed graph \( G'_{res} \) which represents the residual graph in the Ford-Fulkerson algorithm following the standard reduction from the vertex-disjoint paths problem to the (directed) edge-disjoint paths problem: for each vertex \( v \in V - \{s,t\} \), create two vertices \( v_{in} \) and \( v_{out} \) with a directed edge (arc) \( (v_{in}, v_{out}) \), and for each (undirected) edge \( (u,v) \in E \), add the two arcs \( (u_{out}, v_{in}) \) and \( (v_{out}, u_{in}) \). The following fact follows from standard analysis of the Ford-Fulkerson algorithm.

**Fact 15.** There is an augmenting path in \( G \) iff there is a directed \( s \to t \) path in \( G'_{res} \).

We now modify the Ford-Fulkerson residual graph \( G'_{res} \) as follows: for each vertex \( v \in V \) not on one of the \( r \) vertex-disjoint paths, we contract the vertices \( \{v_{in}, v_{out}\} \) into a single vertex \( v \). The resulting graph is our residual graph \( G_{res} \).

**Corollary 16.** There is an augmenting path in \( G \) iff there is a directed \( s \to t \) path in \( G_{res} \).

**Proof.** Observe that this contraction does not create new, simple \( s \to t \) paths, since given any simple directed \( s \to t \) path \( P \) in \( G_{res} \), for each vertex \( v \in P \) not on one of the \( r \) vertex-disjoint paths, replace the occurrence of \( v \) with \( v_{in}, v_{out} \) in that order; the resulting path is a directed \( s \to t \) path \( P \) in \( G'_{res} \). Clearly, since we only contract vertices, we do not destroy any \( s \to t \) paths. Therefore, there is a directed \( s \to t \) path in \( G_{res} \) iff there is one in \( G'_{res} \), and the statement follows from the equivalence in Fact 15.

In fact, since there is always a \( t \to s \) path in \( G_{res} \), we can translate this statement in terms of strong connectivity.

**Corollary 17.** There is an augmenting path in \( G \) iff \( s \) and \( t \) are strongly connected in \( G_{res} \).

The notion of strongly connected components forms the basis of our distributed algorithm. Throughout the algorithm, such as in the next step, we will modify the reachability graph in ways such that \( s \) and \( t \) are strongly connected after the contraction iff they were strongly connected before.

Let \( \overline{G_{res}} \) denote the graph \( G_{res} \) with its arcs replaced by undirected edges (with parallel edges removed). Observe that we can “embed” \( \overline{G_{res}} \) as a subgraph of \( G^2 \) as follows: the nodes \( v_{in}, v_{out} \in V(\overline{G_{res}}) \) map to \( v^1, v^2 \in V(G^2) \), and the nodes \( s, t \) map to \( s^1, t^1 \). Therefore, we can simulate the network \( G_{res} \) using \( G^2 \). From now on, every time we say the distributed algorithm runs on the network \( \overline{G_{res}} \), we mean that it runs on \( G^2 \) with this embedding.

For each connected component \( B \) in \( G - \bigcup_j V(P_j) \), we also have \( B \subseteq V(G_{res}) \); we call \( B \) a bridge in \( G_{res} \), following the terminology of [14]. Suppose we number the bridges \( B_1, B_2, \ldots \).

We claim that \( B_i \) is strongly connected in the \( G_{res} \). Indeed, for each edge \( (u, v) \) in the graph \( G - \bigcup_j V(P_j) \), the two arcs \( (u_{out}, v_{in}), (v_{out}, u_{in}) \) are present in \( G_{res} \), so the contracted vertices...
u, v in $G_{res}$ are connected by both arcs $(u, v)$ and $(v, u)$. Since $B_i$ is connected by such bi-directed arcs, it is strongly connected in $G_{res}$.

We now proceed to construct a reachability graph $G_R$. First, for each bridge $B_i$, contract it into a single vertex $\beta_i$, since $B_i$ is strongly connected, this does not change whether or not $s$ and $t$ are strongly connected. For each edge $(u, v)$ in one of the $r$ current paths where $u$ is to the left of $v$, we remove the arc $(v, u)$. This does not affect the SCCs, since $u_\in$ is still reachable from $v_\out$ along the path $v_\out \rightarrow v_\in \rightarrow u_\out \rightarrow u_\in$. For each of the $r$ existing vertex-disjoint paths $P$ in $G$, the set of vertices $\{v_\in, v_\out\}$ in the reachability graph now form a directed path from $t$ to $s$; see Figure 1. We number these directed paths $P_1, \ldots, P_r$. For each such directed path $P_j$ and two vertices $u, v \in V(P_j)$, we say that $u$ is to the left of $v$ if $v$ can reach $u$ on the directed path $P_j$. Equivalently, we say that $v$ is to the right of $u$. We also form a linear ordering of $V(P_j) \cup \{-\infty, +\infty\}$, where $u < v$ iff $u$ is to the left of $v$, and $v > -\infty$ and $v < +\infty$ for all $v \in V(P_j)$. Also, for each $j \in [r]$, arrange $P_j$ in rightward order as $v_1, \ldots, v_{|P_j|}$ (so that $v_1$ is adjacent to $s$), and for each $i \in [|P_j|]$, define $\pi(v_i) := i$, the index of $v_i$ on the path $P_j$.

The resulting directed graph, whose vertices are $(\bigcup_j P_j) \cup \{\beta_i : \text{bridge } B_i\}$, constitutes our reachability graph $G_R$.

For the distributed setting, the motivation for viewing the bridges $B_i$ as single vertices $\beta_i$ is that we can communicate within each bridge $B_i$ using SA. Initially, every node in $\bigcup V(P_j)$ broadcasts to its neighbors the fact that it belongs to some $P_j$. Then, for each bridge $B_i$, each node in $V_{B_i}$ knows that its neighbors in $B_i$ are precisely the neighbors from which it did not receive a broadcast. This knowledge is exactly what is needed for a single SA round.

Construction of the bridge graph.
We next construct the bridge graph similarly to \cite{14}.

For a bridge $B_i$ and a path $P_j$, let $l^i_j$ be the leftmost ingoing neighbor of $\beta_i$ on $P_j$ in $G_R$, or $-\infty$ if such a neighbor does not exist. In other words, $l^i_j = \min(N^-_{G_R}(\beta_i) \cap P_j)$ according to the linear ordering of $P_j$, or $-\infty$ if the set $N(B_i) \cap P_j$ is empty. Similarly, let $r^i_j$ be the rightmost outgoing neighbor of $B_i$ on $P_j$, or $\infty$ if such a neighbor does not exist. In other words, $r^i_j = \max(N^+_{G_R}(\beta_i) \cap P_j)$. Observe that, by the construction of $G_R$, we always have $r^i_j \geq l^i_j - 1$; see Figure \[1\].

By Lemma \[11\] we can assume that each node $v \in P_j$ knows its index on the path. Then, in $2k$ SA rounds, every node in every bridge $B_i$ can learn the values $l^i_j$ and $r^i_j$. First, every node on $P_j$ broadcasts its index to all its neighbors in $G_{res}$. Then, for each bridge $B_i$, every node $v \in B_i$ sets $x^i_v$ as the minimum index received from an in-neighbor of $P_j$ in $G_{res}$, and sets $x^i_v$ as the maximum index received from an out-neighbor of $P_j$ in $G_{res}$. Then, in two SA rounds, every node learns the minimum $x^i_v$ and maximum $x^i_v$ within its bridge, which are the values $l^i_j$ and $r^i_j$. We now construct the bridge graph $G_B$.

1. The vertices of $G_B$ is the set $\{\beta_i : B_i$ is a bridge\}.
2. For two bridges $B_i, B_x$ and $j \in [r]$, add an arc $(\beta_i, \beta_x)$ to a set $D_j$ if:
   a. For some path $P_y$, $l^i_y \leq r^i_x$. Intuitively, this means that we can reach $B_x$ from $B_i$ in $G_R$ by traveling leftward from $r^i_x$ to $l^i_y$ along path $P_y$.
   b. We have $r^i_x > r^i_j$. Intuitively, this means that we make “progress” along path $P_j$, in that we can now reach a vertex in $P_j$ further to the right.
   c. There is no $\beta_i$ such that $\beta_x$ satisfies the above two conditions, and either $r^i_x > r^i_x$, or $r^i_j = r^i_x$ and $ID(B_j) > ID(B_x)$. In other words, ties are broken by ID (assume that each bridge has a unique ID).

For each $D_j$, we add all arc in $D_j$ to $G_B$.

We say that a node $\beta_i$ is s-reachable if there is a node in $B_i$ with $s$ as an in-neighbor. Likewise, we say that a node $\beta_i$ is t-reachable if there is a node in $B_i$ with $t$ as an out-neighbor.

We can compute the arc set $D_j$ in a distributed fashion, such that for each $\beta_i$, every node in $B_i$ knows the arc $(\beta_i, \beta_x) \in D_j$, if any. We assume that every bridge $B_x$ has computed the values $l^i_x$ and $r^i_x$ for all $j \in [k]$. First, each node $v \in B_x$ broadcasts the value $(\pi(r^i_x), ID(B_x))$ to each in-neighbor in $G_{res}$ on a path $P_y$. Then, every path $P_y$ computes a prefix maximum of the values $(\pi(r^i_x), ID(B_x))$ sent on the previous step. By maximum, we mean lexicographic, so ties are broken by highest $ID(B_x)$; if nothing is sent over a given prefix, then the maximum is $-\infty$ with an arbitrary $ID$. By Lemma \[11\] this maximum can be computed in parallel for each $P_j$ in $O(\log n)$ SA rounds.

At this point, for each $y \in [k]$ and node $v \in P_y$, we have computed the maximum $r^i_x$ over all nodes $\beta_x$ with $l^i_y \leq v$. We now have every node $v \in P_y$ broadcast this maximum $(\pi(r^i_x), ID(B_x))$ to its in-neighbors in $G_{res}$. Finally, every bridge $B_j$ computes the maximum $(\pi(r^i_x), ID(B_x))$ received by one of its nodes through a SA round. If the maximum value $r^i_x$ satisfies $r^i_x > r^i_j$, then every node in $B_i$ now knows the arc $(\beta_i, \beta_x) \in D_j$.

We remark that our bridge graph construction is slightly different from the one in \cite{14}, in order to make it more amenable to distributed computing. We have a statement similar to Theorem 4.1 from \cite{14}, stating an equivalence between the residual graph and the bridge graph. By Fact \[15\] this equivalence also extends to augmenting paths in $G$. Because the proof resembles the one in \cite{14}, we defer it to Appendix \[3\].
Lemma 18. There is an augmenting path between $s$ and $t$ in $G$ iff there exists $s$-reachable $\beta_i$ and $t$-reachable $\beta_x$ and a directed $\beta_i \rightarrow \beta_x$ path in $G_B$.

Like in [14], our next goal is to determine whether there is an $s \rightarrow t$ path in $G_B$. Of course, since directed reachability is a difficult problem in general, we need to exploit the special structure of $G_B$. [14] proceeds by iteratively shortcutting the graph, while we proceed using contraction. This deviation from [14] is the main technical contribution of the paper.

From now on, we abuse notation, sometimes referring to $D_j$ as the directed graph whose arcs are precisely $D_j$. Observe that for each $j \in [r]$, every vertex has out-degree at most 1 in $D_j$. Also, the directed graph $D_j$ is acyclic, since an arc $\beta_i \rightarrow \beta_x$ implies that $r_x^i > r_x^j$. It follows that $D_j$ is composed of rooted trees, where the arcs point from away from the leaves towards the root. We now show that every rooted tree in $D_j$ is in fact strongly connected in $G_B$.

Claim 19. Suppose $\beta_i$ and $\beta_x$ belong in the same rooted tree in $D_j$. Then, $\beta_i$ and $\beta_x$ are strongly connected in $G_{res}$.

Proof. It suffices to prove the statement for pairs $\beta_i, \beta_x$ where arc $(\beta_i, \beta_x)$ exists in $D_j$. By condition 2(a) in the construction of $G_B$, there is a path $P_y$ with $l_y^i \leq r_y^j$. Therefore, from $\beta_i$, we move to $r_y^i$, then (left) along $P_y$ to $l_y^i$, and then to $\beta_x$. From $\beta_x$, we can move to $r_x^j$, then along $P_j$ to $l_j^i$ (since $r_x^j > r_x^j \geq l_j^i - 1$), and finally back to $\beta_i$, showing strong connectivity.

With the bridge graph $G_B$ computed, our two remaining steps are: (i) determine if there exists $s$-reachable $\beta_i$ and $t$-reachable $\beta_x$ and a directed $\beta_i \rightarrow \beta_x$ path in $G_B$, and (ii) return an augmenting path or a node separator of size $r$, depending on the outcome of (i).

For the rest of this section, we will shift our main focus from the classical setting to the distributed setting. That is, we will explain our algorithm from a distributed point of view, rather than commenting on distributed implementations in gray boxes.

3.2 Solving the Bridge Graph

Our distributed algorithm differs from the one in [14] by using a contraction-based approach, rather than a shortcutting-based one. A high-level outline of our algorithm is as follows. First, we contract every rooted tree of $D_1$, or equivalently, every connected component in $\overline{D_1}$, the underlying undirected graph of $D_1$. We now recompute the bridge graph with the corresponding bridges of each connected component merged into a single super-bridge in $G_{res}$. We repeat this process for the remaining $j \in [r]$: contract every connected component in $\overline{D_j}$, recompute the bridge graph, and repeat. The lemma below states the desired property of the contraction algorithm. The proof is deferred to Appendix [13] due to its length.

Lemma 20. At the end of the contraction algorithm, there exists $s$-reachable $\beta_i$ and $t$-reachable $\beta_x$ that contract to the same vertex iff $s$ and $t$ are strongly connected in $G_{res}$.

First, for each $j \in [r]$, in the distributed computation of the bridge graph $G_B$, we can augment the computation of the arc $(\beta_i, \beta_x)$ in $D_j$ (if any) so that for each $B_i$, only do the nodes know $\beta_x$, but also (i) $ID(B_x)$, (ii) the minimum value $y \in [r]$ for which $l_y^i \leq r_y^j$, and (iii) the value $\pi(l_y^i)$; this simply requires broadcasting the auxiliary information on each step, and breaking ties by value of $y$. Observe that since $\beta_i$ and $\beta_x$ belong in the same SCC in $G_{res}$, so do all nodes $v \in P_y$ with $l_y^i \leq v \leq r_y^j$.

We now describe our algorithm for contracting the connected components in $\overline{D_1}$. Our next goal is to find, for each connected component $C$ in $\overline{D_1}$, a subgraph $H_C^G$ in $G_{res}$ spanning the

---

2 It is not important that this is minimum and not maximum or even arbitrary. However, we assume minimum because uniqueness will make our analysis easier to present.
bridges \( B_i \) whose nodes \( \beta_i \) are in that component. Every node in the subgraph should know its neighbors in the subgraph. Moreover, this subgraph should be a tree, a property that will be useful later for recovery.

For each arc \((\beta_i, \beta_j)\) in \( C \), consider the corresponding pair \((l_y^v, r_y^v)\) as mentioned above. We want to connect together all nodes \( v \in P_y \) with \( l_y^v \leq v \leq r_y^v \) into \( H^1_C \). In other words, each node \( v \in P_y \) should know whether there exists an \((l_y^v, r_y^v)\) pair such that \( l_y^v < v < r_y^v \); if so, the node \( v \) connects to its leftward adjacent node on \( P_j \). We accomplish this task as follows: first, every bridge \( B_i \) with an arc \((\beta_i, \beta_j)\) in \( D_1 \) and corresponding pair \((l_y^v, r_y^v)\) sends the value \( \pi(l_y^v) \) to node \( r_y^v \), which is necessarily adjacent to \( B_i \) in \( \overline{G_{res}} \). Then, for each \( y \in [r] \), the nodes on \( P_y \) compute a suffix minimum of the sent \( \pi(l_y^v) \) values. It is clear that for each node \( v \in P_y \), there exists \((l_y^v, r_y^v)\) with \( l_y^v < v \leq r_y^v \) iff the suffix minimum at \( v \) is strictly less than \( \pi(v) \). Therefore, \( v \) joins its leftward neighbor iff its suffix minimum is less than \( \pi(v) \).

Observe that if node \( v \) has suffix minimum exactly \( \pi(v) \), then node \( v \) is at the left endpoint of some \((l_y^v, r_y^v)\) pair. We claim that the converse is true.

> **Claim 21.** Suppose the arc \((\beta_i, \beta_j)\) exists in \( D_1 \), and consider the minimum \( y \in [r] \) such that \( l_y^v \leq r_y^v \). Then, the suffix minimum at \( l_y^v \) is exactly \( \pi(l_y^v) \). Moreover, there does not exist an arc \((\beta_i', \beta_j')\) in \( D_1 \) such that \( x \neq x' \) and \( l_y^v = l_y^{v'} \).

**Proof.** Suppose the first statement is false. Then, there is some other arc \((\beta_i', \beta_j')\) in \( D_j \) such that \( l_y^{v'} \leq r_y^{v'} \leq r_y^v \). Assume that either \( r_y^{v'} > r_y^v \), or \( r_y^{v'} = r_y^v \) and \( ID(B_x) > ID(B_x') \). Then, since \( l_y^{v'} \leq l_y^v \leq r_y^v \) and \( r_y^{v'} > r_y^v \), \( \beta_i' \) violates condition 2(c) for \((\beta_i, \beta_j)\) in the definition of the bridge graph, contradiction. The other case, assuming that either \( r_y^{v'} > r_y^v \), or \( r_y^{v'} = r_y^v \) and \( ID(B_x) > ID(B_x') \) is symmetric: we have \( \beta_j' \) violating condition 2(c) for \((\beta_i, \beta_j)\). For the second case, if such an arc \((\beta_i', \beta_j')\) exists, then without loss of generality, assume that either \( r_y^{v'} > r_y^v \), or \( r_y^{v'} = r_y^v \) and \( ID(B_x') > ID(B_x) \) (we swap \( x \) and \( x' \) otherwise). Then, since \( l_y^{v'} \leq l_y^v \), we can apply the proof of the first statement to get a contradiction.

This finishes the edges of \( H^1_C \) within each \( P_j \). Of course, we can repeat the above in parallel for each connected component \( C \) in \( \overline{D_1} \). We now augment the above computation so that when computing suffix minimum, the value \( ID(B_x) \) is recovered along with the minimum \( \pi(l_y^v) \). Then, if a node \( v \in P_y \) has suffix minimum exactly \( \pi(v) \), then \( v \) connects to its neighbor(s) in \( B_i \) in \( H^1_C \), where \( C \) is the component containing \( \beta_x \). Then, for each arc \((\beta_i, \beta_j)\) in \( D_1 \) and minimum \( y \) for which \( l_y^v \leq r_y^v \), the bridge \( B_i \) connects to \( r_y^v \) in \( H^1_C \), where \( C \) is the component containing \( \beta_i \). Finally, within each bridge \( B_i \), add a spanning tree of \( B_i \) into the corresponding subgraph \( H^1_C \) using Lemma 9. This concludes our construction of the subgraph for each connected component in \( D_1 \).

We now prove some properties of the subgraphs \( H^1_C \). The tree property below does not help us in the contraction phase, but it will help in recovering paths in the recovery phase.

> **Claim 22.** The subgraphs \( H^1_C \) satisfy the following properties:

1. For every connected component \( C \) in \( \overline{D_1} \), there is a subgraph spanning (precisely) the bridges in \( C \).
2. Any two subgraphs are disjoint.
3. Every subgraph is a tree.

**Proof.** Claim 21 implies the following statement: for any two arcs \((\beta_i, \beta_j), (\beta_i', \beta_j')\) in \( D_1 \), either the path \( r_y^v \rightarrow l_y^v \) on \( P_y \) and the path \( r_y^{v'} \rightarrow l_y^{v'} \) on \( P_y \) are disjoint, or \( x = x' \). If \( x = x' \), then the arcs \((\beta_i, \beta_j), (\beta_i', \beta_j')\) belong to the same connected component in \( \overline{D_1} \). Therefore, if \((\beta_i, \beta_j), (\beta_i', \beta_j')\) belong to different connected components, then the segments \( r_y^v \rightarrow l_y^v \) and \( r_y^{v'} \rightarrow l_y^{v'} \) do not intersect. It is easy to see by the algorithm that each constructed \( H^1_C \) is disjoint. This proves (1) and (2).
For property (3), fix a subgraph $H_C^1$. For each $\beta_i \in C$, the corresponding bridge $B_i$ induces a tree in $H_C^1$ by construction. Therefore, the subgraph $H_C^1$ is acyclic iff the graph obtained by contracting each bridge $B_i$ in $H_C^1$ is acyclic. Suppose we take $H_C^1$ and contract each $B_i$ into a vertex $\beta_i$, so that the new graph, called $H$, is now a subgraph of $G_R$. To prove that $H$ is acyclic, we start with $D_1$ and transform it into $H$ while preserving the acyclicity of $D_1$. For each $\beta_x \in C$ with positive in-degree in $D_1$, let $\beta_{x_1}, \ldots, \beta_{x_t}$ be the in-neighbors of $\beta_x$. In $H_C^1$, the algorithm adds the union of the (now undirected) paths $r_{x_i}^y \rightarrow l_{x_i}^y$ to $H_C^1$; in addition, for each $\beta_{x_i}$, the algorithm adds an edge connecting $B_{x_i}$ to $r_{x_i}^y$. The set of edges added is a tree connecting the vertices $\beta_x, \beta_{x_1}, \ldots, \beta_{x_t}$ of $H$; call this tree $T_x$. For each $\beta_x$ in $D_1$, we delete the edges $(\beta_x, \beta_{x_i})$ in $D_1$ and add $T$ (as well as any extra vertices). Since we always delete a tree and add back a tree, the graph remains acyclic. At the end, we have added exactly the edges in $H_C^1$, proving that $H_C^1$ is acyclic. Lastly, $H_C^1$ is connected by property (1), so it is a tree.

Since the subgraphs $H_C^1$ are disjoint, they serve as the nodes after contracting every connected component of $D_1$ in the classical algorithm, and we can communicate within each contracted component in one SA round. We would like to continue this algorithm for $j = 2, \ldots, r$, always maintaining subgraphs $H_C^j$ that are trees spanning the super-bridges in the connected components of $D_j$, but we run into the following obstacle: when constructing the subgraph on the next iteration, we might reuse edges in $E(H_C^j) \cap E(\bigcup P_j)$, i.e., the edges in $H_C^j$ inside the paths $P_j$. Reusing these edges may destroy the tree property of $H_C^j$.

We fix this problem as follows. Our goal is to construct a graph $G^+$ that “embeds” into $G^{2r}$ (the same way $G_{res}$ embeds into $G^r$). Replace each node in $v \in \bigcup P_j$ with $r$ copies $v^1, \ldots, v^r$, connected together in a clique, with each node $v^j$ sharing the same neighbors as $v$. Let us call this network $G^+$. Observe that $G^+$ embeds into $(G_{res})^r$ and $G_{res}$ embeds into $G$, which means that $G^+$ embeds into $G^{2r}$.

We run the algorithm on $G^+$ instead, which we then simulate on $G$. For the first iteration $j = 1$, every node $v^1$ now takes the role of node $v \in \bigcup P_j$. This way, in future iteration $j = 2, \ldots, r$, we always have a fresh set of nodes, namely the nodes $v^j$, to use in iteration $j$. For
iteration \( j = 2, \ldots, r \), we repeat the algorithm for \( j = 1 \) with three main differences: (i) before the iteration, every super-bridge is now the nodes in some \( H_C^{j-1} \), (ii) we do not add a spanning tree inside each bridge, since we already have one from iteration \( j - 1 \), and (iii) every node \( v \) now takes the role of node \( v \in \bigcup_i P_i \). Since our algorithm emulates the classical algorithm, we know by Lemma 20 that after all \( r \) iterations, there is a directed \( s \to t \) path iff there is an \( s \)-reachable bridge and a \( t \)-reachable bridge in the same subgraph \( H_C^r \). Our next goal is to, depending on this outcome, either find an augmenting path in \( G_{res} \) or find an \( s \to t \) node cut of size at most \( r \).

### 3.3 Finding an Augmenting Path

First of all, it is easy to test if there exists a subgraph \( H_C^r \) with both an \( s \)-reachable and \( t \)-reachable bridge: in two SA rounds, the \( s \)-reachable bridges and the \( t \)-reachable bridges broadcast in their subgraphs \( H_C^r \). If there exists a subgraph \( H_C^r \) with both \( s \)-reachable and \( t \)-reachable bridges, then we show how find an augmenting path in \( G_{res} \). Note that this task is not trivial, since while there exists a (unique) path from the \( s \)-reachable and \( t \)-reachable bridges in \( H_C^r \), this path may go rightward along a path \( P_j \), which is not allowed.

For illustration, suppose first that in some \( H_C^r \), there already exist an \( s \)-reachable bridge \( B_s \) and a \( t \)-reachable bridge \( B_t \). Let \( \beta_s \) be the “root” of the component (tree) \( C \) in \( D_1 \), i.e., the unique \( \beta_x \) with no out-arc in \( D_1 \). The unique path from \( B_s \) to \( B_t \) in \( H_C^r \) only goes left along the paths \( P_j \), since every time we travel along an arc in \( D_1 \) from \( B_s \) to \( B_t \), we traverse leftward along one path \( P_j \) from one bridge to another. Moreover, we can compute this path in \( O(\log n) \) SA rounds using Lemma 12. Next, we “trim” the path by removing all edges completely inside \( B_s \) or \( B_t \).

If \( B_s = B_t \), then we skip the next step; otherwise, since there is a path from \( B_s \) to \( B_t \) in \( D_1 \), we have \( r^1_s > r^1_t \). Combining this with \( r^1_s \geq l^1_s - 1 \) gives \( r^1_t \geq l^1_s \). Therefore, we can extend this path from \( B_s \) to \( B_t \) to travel left from \( l^1_s \) to \( l^1_t \), and then enter \( B_t \). Again, by using Lemma 12, we can establish this path in \( O(\log n) \) SA rounds, and then trim it by removing all edges completely inside \( B_s \) or \( B_t \). Finally, we connect the ends of the \( B_s - B_s \) and \( B_s - B_t \) paths inside \( B_s \), which can be done in \( O(\log n) \) SA rounds.

Lastly, it remains to connect node \( s \) to the \( B_s \)-end of the path, and to connect the \( B_t \)-end of the path to \( t \). Again, these take \( O(\log n) \) SA rounds.

In general, we process the graphs in reverse order \( H_C^r, H_C^{r-1}, \ldots, H_C^1 \). For a graph \( H_C^j \), we first repeat the algorithm for the \( H_C^j \) case above, treating the connected components of \( H_C^{j-1} \) as the contracted bridges. Observe that all edges outside any component \( H_C^{j-1} \) must travel leftward along the paths \( P_j \). We now erase all edges inside each traversed component \( H_C^{j-1} \) and connect the two broken endpoints in \( H_C^{j-1} \) on the next iteration. At the end, we have constructed a path \( P^+ \) that only travels left along paths \( P_j \).

The last issue is that unlike the \( H_C^1 \) case, this path \( P^+ \) may not be simple. Indeed, since there are \( r \) copies of each node \( \bigcup_i P_j \), a single arc in some \( P_j \) can be traversed left up to \( r \) times, once in each copy. We can fix this issue with the following “shortcutting” step: first, number the nodes on \( P^+ \) from 1 to \( |P^+| \) using Lemma 14 and suppose every other node gets value \( x_v := 0 \). Then, for each node \( v \in \bigcup_i P_j \), every node \( \{v^1, \ldots, v^r\} \) updates \( x_v := \max(x_v, x_{v^i}) \), which can be done in one round since the nodes \( v^1, \ldots, v^r \) are connected by a clique. Then, every node \( v \in P^+ \) computes a prefix maximum of the \( x_v \) values in the path. For each \( v \in P^+ \), if this maximum is greater than \( x_v \), then the path \( P^+ \) can be pruned before \( x_v \) is reached; see Figure 3. Therefore, node \( v \) in \( P^+ \) drops out of the path \( P^+ \). Finally, we collapse the graph \( G^+ \) back into \( G \), giving our desired simple augmenting path in \( G_{res} \). Lastly, translating the augmenting path in \( G_{res} \) to \( r + 1 \) vertex-disjoint paths in \( G \) can be done as in the Ford-Fulkerson algorithm.
3.4 Finding a Node Cut

If there is no subgraph $H^C_G$ with both an $s$-reachable and $t$-reachable bridge, then there is no augmenting path, so the algorithm needs to find an $s-t$ node cut $S$ of size at most $k$.

Let $B_s$ denote all bridges $B_i$ inside some subgraph $H^C_G$ containing an $s$-reachable bridge. For each bridge $B_i$ to learn whether or not $B_i \in B_s$, we have every node inside a bridge that is adjacent to $s$ broadcast to its subgraph $H^C_G$; the bridges $B_i \in B_s$ are precisely the ones that receive such a broadcast.

We now present a set of $k$ nodes, one from each $P_k$, that form an $s-t$ node cut; a similar construction is presented in the proof of Theorem 4.1 in [14]. For path $P_j$, let $w'_j$ be the rightmost $r^j$ over all bridges $B_i \in B$. Necessarily, $w'_j = v_{in}$ for some $v$ on the vertex-disjoint path in $G_{res}$ corresponding to $P_j$; let $w_j$ be this node $v$. If this node does not exist, i.e., $r^j = -\infty$ for all bridges $B_i \in B$, then let $w_j$ be the first node on $P_j$ (the one adjacent to $s$).

We now show how to compute the nodes $w_j$ for each $j \in [r]$. First, every bridge $B_i \in B_s$ notifies node $r^j$; then, every node in $v \in P_j$ sets $x_v$ to be its index on $P_j$ if it is notified, and 1 otherwise. The nodes in $P_j$ then compute aggregate maximum of the values $x_v$. Finally, the node $v \in P_j$ whose index is exactly its value $x_v$ becomes $w_j$.

**Lemma 23.** The set $\{w_1, \ldots, w_r\}$ is a node cut of $G$.

**Proof.** First, we would like to extend Lemma 20 to the following statement: at the end of the contraction algorithm, an $s$-reachable vertex $\beta_s$ contract to the same vertex iff $s$ and $B_i$ are strongly connected in $G_{res}$. To do so, imagine changing the graph $G_{res}$ as follows: remove all arcs from any bridge to $t$ and add a single arc from a vertex in $B_s$ to $t$. With this modification, the graphs $G_R$ and $G_B$ do not change, but now, only $\beta_s$ is $t$-reachable; applying Lemma 20 proves the statement. From now on, we forget this modification, i.e., we stick with the original $G_{res}$.

Observe that the distributed algorithm follows the contraction algorithm of Lemma 20 in particular, every subgraph $H^C_G$ contains the bridges $B_i$ whose vertices $\beta_i$ get contracted to a single vertex in the contraction algorithm. Therefore, the bridges $B_i \in B$ are precisely the $\beta_i$ that get contracted to the same vertex as some $s$-reachable bridge. By the statement at the beginning of this proof, these bridges $B_i$ are precisely those strongly connected to $s$.

The rest of our proof resembles the last paragraph of the proof of Theorem 4.1 in [14]. Suppose for contradiction that there is a simple $s-t$ path $P$ in $G - \{w_1, \ldots, w_r\}$. Consider the subsequence of vertices in $P$ that are also in $\bigcup P_j$. At some point, we must have two vertices $v, v' \in \bigcup P_j$ adjacent on this subsequence such that for the paths $P_j, P_j'$ containing $v$ and $v'$ respectively, we have $v < w_j$ and $v' > w_{j'}$. The vertices $v, v'$ cannot be adjacent in $P$, so there must be vertices inside a single bridge $B_i$ in between the occurrences of $v$ and $v'$ on $P$. This bridge $B_i$ satisfies $l^j < w_j$ and $r^j > w_{j'}$. Since $w_j = r^j$ for some $B_i \in B$, vertex $w_j$ is reachable.
from $s$. Therefore, vertex $l'_j$ is also reachable from $s$, and so is $B_i$, which means that $s$ and $B_i$ are strongly connected in $G_{res}$. In particular, $B_i \in B$, so $w_{y^j} \geq r_{y^j}$ by definition of $w_{y^j}$, contradicting the assumption that $r_{y^j} > w_{y^j}$.

This finishes the $k$-vertex disjoint paths algorithm and Lemma 2.

### 3.5 Running on Multiple Subgraphs

We have proved our main result of this section, Lemma 2, restated below for reference. Below, we state some modifications of this result that are more directly useful in the next section.

**Lemma 2.** Given a graph $G = (V, E)$ of treewidth at most $k$ and two vertices $s,t \subseteq V$, we can either find $k$ vertex-disjoint $s$-$t$ paths, or output an $s$-$t$ node cut of size less than $k$, in $\tilde{O}(kO(1)D)$ rounds. In the former case, every node knows whether it is on a path, and if so, its predecessor and successor on that path. In the latter case, every node knows the fact that $k$ vertex-disjoint paths do not exist, as well as whether it is in the node cut.

First, we obtain a generalization where we want to find vertex-disjoint paths between two sets of nodes, not just $s,t$, within a connected subgraph of $G$, not $G$ itself. Moreover, this formulation includes forbidden nodes, those which cannot appear in any vertex-disjoint path.

**Corollary 24.** Given a graph $G = (V, E)$ of treewidth $k$, a set $U \subseteq V$ such that $G[U]$ is connected, and three disjoint vertex sets $A, B \subseteq V - U$ and $X \subseteq U$, we can either find $k$ vertex-disjoint $A$--$B$ paths whose internal nodes belong in $G[U] - X$, or conclude that $k$ vertex-disjoint paths do not exist, in $\tilde{O}(kO(1)D)$ rounds. In the positive case, every vertex knows whether it is on a path, and if so, its predecessor and successor on that path. In the negative case, every vertex knows the fact that $k$ vertex-disjoint paths do not exist.

**Proof.** The subgraph is not an issue, because any simulated network $G[U]^t$ in the algorithm is a subgraph of $G^t$, so we can simulate network $G[U]^t$ on $G^t$ first, and then on $G$.

To address the $A$--$B$ paths modification, imagine adding a node $s$ whose neighbors are precisely $A$, and a node $t$ whose neighbors are precisely $B$. The virtual nodes $s,t$ do not exist in the network, but observe that through the algorithm of Lemma 2, the only times when nodes $s$ and $t$ are active are (i) when we compute prefix/suffix aggregates on at most $k$ paths, and (ii) when we compute a path with $s$ and/or $t$ as an endpoint, in an attempt to find an augmenting path. In case (i), for each path $P_\ell$, the node $v \in P_\ell$ with $\pi(v) = 2$ (i.e., the node to the immediate right of $s$) can take the role of $s$ in prefix computations; likewise, the node with $\pi(v) = |P_\ell| - 1$ can take the role of $t$ in suffix computations.

In case (ii), the nodes in $A$ and $B$, which are in the network, can take the role of nodes $s$ and $t$. For example, if we have an $s$-reachable bridge and we want a path from $s$ to a specific node $v$ in the bridge $B_i$, then we instead compute a path from a node in $N(s) \cap B_i$ to $v$.

Lastly, the forbidden node set $X$ is also not a problem: when computing the bridges $B_i$, these nodes purposefully do not join any bridge.

The next generalization really emphasizes the power of the shortcuts framework: suppose, instead, that we want to solve $k$ vertex-disjoint paths on a subgraph $H \subseteq G$. Actually, we want to solve multiple instances of the problem on vertex-disjoint subgraphs $H_1, \ldots, H_\ell$. Then, we can solve them all simultaneously in $\tilde{O}(kO(1)D)$ rounds!

---

3 The corner case $|P_\ell| = 2$ can be ignored, since we can greedily choose the single edge from $s$ to $t$ as a path. Likewise, if any nodes in $A$ and $B$ are adjacent, we can greedily choose them as vertex-disjoint paths.
Corollary 25. Given multiple instances \((U_i, A_i, B_i, X_i)\) in Corollary 24 such that the node sets \(U_i\) are disjoint, we can, simultaneously for each \((U_i, A_i, B_i, X_i)\), either find \(k\) vertex-disjoint \(A_i\)—\(B_i\) paths whose internal nodes belong in \(G[U_i] - X_i\), or conclude that \(k\) vertex-disjoint paths do not exist, in \(\tilde{O}(k^{O(1)}D)\) total rounds.

Proof. For each instance \((U_i, A_i, B_i, X_i)\), every step of the algorithm either runs on network \(G[U_i]\), or on \(G[U_i][i]\), for some \(i \leq k\). Since the networks \(G[U_i][i]\) are disjoint subgraphs of \(G^k\) for different \(U_i\), we simulate every network \(G[U_i]\) or \(G[U_i][i]\) on \(G^k\). Therefore, on each step, the SA tasks of the different instances can be simultaneously solved on \(G^k\) in \(\tilde{O}(k^{O(1)}D)\) rounds, which is then simulated on \(G\) in \(\tilde{O}(k^{O(1)}D)\) rounds.

4 Algorithm Outside Disjoint Paths

In this section, we provide the rest of the algorithm for approximating treewidth, which uses the \(k\)-vertex disjoint paths problem as a subroutine. It is a combination of the efficient sequential and parallel algorithms in \(16, 15\). It will make repeated calls to the algorithm of Corollary 25, the corollary of Lemma 2 in the previous section.

The treewidth approximation algorithm, which is recursive, uses the concept of graph separators, defined below.

Definition 26 (Separation). Let \(G = (V, E)\) be a graph and let \(A, B, S \subseteq V\). We say that \(S\) separates \(G\) into \(A\) and \(B\) if \(A, B, S\) partition \(V\) and \(N(A) \subseteq S\) and \(N(B) \subseteq S\). Note that \(A\) and \(B\) are not necessarily connected, which means that there could be multiple choices for \(A\) and \(B\). In addition, for disjoint \(X, Y \subseteq \text{S}\), we say that \(S\) is an \(X—Y\) separator if there exist \(A, B \subseteq V\) with \(X \subseteq A, Y \subseteq B\) such that \(S\) separates \(G\) into \(A\) and \(B\).

Definition 27 (Balanced Separation). Let \(G = (V, E)\) be a graph and let \(X \subseteq V\). A set \(S \subseteq V\) is an \((X, \alpha)\)-balanced separator of \(G\) if there exist \(A, B \subseteq V\) such that \(S\) separates \(G\) into \(A\) and \(B\) and \(|A \cap X|, |B \cap X| \leq \alpha|X|\). When \(X = V\), we omit the \(X\), using the term \(\alpha\)-balanced separator instead.

The following well-known fact states that if \(G\) has bounded treewidth, then it admits constant-sized balanced separators for any \(X \subseteq V\).

Lemma 28 (Lemma 7.20 of \([4]\)). Let \(G = (V, E)\) be a graph of treewidth \(k\). For any set \(X \subseteq V\), there exists an \((X, 2/3)\)-balanced separator of \(G\) of size \(k + 1\).

The algorithm is recursive, always running on an instance \((U, X)\) with \(U \subseteq V\), \(X \subseteq N[U]\), and \(|X| \leq 7k + 4\). It starts with \((U, X) = (V, \emptyset)\). The algorithm has two cases, depending on whether the current recursion depth is odd or even (the initial instance \((V, \emptyset)\) has recursion depth 0). The even case finds a balanced separator \(S\) of size \(\leq k + 1\) in the graph \(G[U]\) separating the current graph into components whose sizes are a constant fraction smaller, and for each component \(U' \) of vertices, recursively calls \((U', N[U'] \cap (X \cup S))\). This guarantees the algorithm \(O(\log n)\) recursion depth, but comes at a cost: the size of \(X\) increases by \(\leq k + 1\) upon the next recursion call (to the odd case). This increase is remedied in the odd case below, which on input \((U, X)\) finds a set \(S\) that separates \(G[U]\) into components, each with \(\leq \frac{2}{3}|X|\) vertices in \(X\), and for each component \(U'\) of vertices, recursively calls \((U', N[U'] \cap (X \cup S))\). If \(|X| \geq 6k + 4\), then

\[
(U', N[U'] \cap (X \cup S)) \leq \frac{2}{3}|X| + |S| = |X| - \frac{1}{3}|X| + |S| \leq |X| - \frac{1}{3}(6k + 4) + (k + 1) = |X| - k - \frac{1}{3},
\]

\[\]

\(^4\) Of course, if no such balanced separator \(S\) is found, the algorithm can immediately exit and conclude that \(\text{tw}(G) > k\).
In this case, our goal is to reduce the size of $X$ sufficiently. We know by Lemma 28 there exists an $(X, 2/3)$-balanced separator in $G$. Suppose $S$ is this separator, which separates $G$ into $A$ and $B$ such that $|A \cap X|, |B \cap X| \leq (2/3)|X|$. If we let $Y := A \cap X$ and $Z := B \cap X$, then this means that $|Y|, |Z| \leq \frac{2}{3}|X|$ and the set $S$ contains $X - (Y \cup Z)$ and is an $Y-Z$ separator. The algorithm proceeds by trying all possible values of $Y$ and $Z$ and finding such a set $S$.

The algorithm for odd recursion depth proceeds as follows. For all partitions $X', Y, Z$ of $X$ with $|Y|, |Z| \leq \frac{2}{3}|X|$, try to find an $Y-Z$ separator in the graph $G[U] - X'$ of size $\leq (k+1) - |X'|$, which is an instance of $(k+1 - |X'|)$-VERTEX DISJOINT PATHS. If no such separator is found over all $Y, Z$, terminate the algorithm and output the conclusion that tw($G$) $> k$. Else, for partition $X', Y, Z$ and $Y-Z$ separator $S'$, let $S := X' \cup S'$, and for each component $U'$ of vertices in $G[U] - S$, recursively call $(U', N[U'] \cap (X \cup S))$. If the graph $G[U] - S$ is empty, then this recursion branch terminates.

4.1 Odd Recursion Depth

In this case, our goal is to reduce the size of $X$ sufficiently. We know by Lemma 28 there exists an $(X, 2/3)$-balanced separator in $G$. Suppose $S$ is this separator, which separates $G$ into $A$ and $B$ such that $|A \cap X|, |B \cap X| \leq (2/3)|X|$. If we let $Y := A \cap X$ and $Z := B \cap X$, then this means that $|Y|, |Z| \leq \frac{2}{3}|X|$ and the set $S$ contains $X - (Y \cup Z)$ and is an $Y-Z$ separator. The algorithm proceeds by trying all possible values of $Y$ and $Z$ and finding such a set $S$.

The algorithm for odd recursion depth proceeds as follows. For all partitions $X', Y, Z$ of $X$ with $|Y|, |Z| \leq \frac{2}{3}|X|$, try to find an $Y-Z$ separator in the graph $G[U] - X'$ of size $\leq (k+1) - |X'|$, which is an instance of $(k+1 - |X'|)$-VERTEX DISJOINT PATHS. If no such separator is found over all $Y, Z$, terminate the algorithm and output the conclusion that tw($G$) $> k$. Else, for partition $X', Y, Z$ and $Y-Z$ separator $S'$, let $S := X' \cup S'$, and for each component $U'$ of vertices in $G[U] - S$, recursively call $(U', N[U'] \cap (X \cup S))$. If the graph $G[U] - S$ is empty, then this recursion branch terminates.

4.2 Even Recursion Depth

In this case, our goal is to separate the current graph into components a constant factor smaller, in order to bound the recursion depth by $O(\log n)$. We first introduce the concept of splitters from 15.

Definition 29 (B-splitter). For a rooted, spanning tree $T \subseteq G[U]$, a set of vertices $R \subseteq U$ is a $B$-splitter if $R \leq n/B$ and every connected component in $T - R$ has less than $B$ vertices.

For a vertex $v \in U$, define desc($v$) and children($v$) as the children and descendants of $v$, respectively. Define sub_size($v$) as the number of vertices in the subtree rooted at $v$. Let $R$ be all the vertices $v \in U$ satisfying the following condition:

$$1 + \sum_{u \in \text{children}(v)} \text{sub_size}(u) \text{ mod } B > k.$$

Theorem 30 (Theorem 9.2 in 15). The set $R$ defined above is an $R$-splitter.
The distributed algorithm first computes an arbitrary spanning tree \( T \) of \( G[U] \), e.g., by computing an MST with arbitrary weights following [7], and roots it at an arbitrary vertex. At this point, every node knows its parent and children in the rooted tree. To compute the set \( R \), each node \( v \in U \) first computes the size of its subtree in \( T \); this can be done in \( O(\log n) \) SA rounds using tree aggregation techniques in [12]. Then, each node broadcasts \( \text{sub}_\text{size}(v) \) to its parent node in a single round, so that each node can determine whether it joins \( R \).

The rest of this even recursion section is based on [16]. For each vertex \( r \in R \), we define

\[
\text{sub}_\text{size}(r) = \sum_{r' \in \text{desc}(r) \cap R} \text{sub}_\text{size}(r'),
\]

and for \( R' \subseteq R \), \( \text{sub}_\text{size}(R') = \sum_{r \in R'} \text{sub}_\text{size}(r) \). In other words, \( w_r \) is the number of vertices \( v \) for which \( r \) is the first vertex in \( R \) encountered on the path from \( v \) to the root. Also, observe that \( \sum_{r \in R} w_r = |U| \). The following lemma states that if a separator \( X \) does not intersect \( R \), then the values \( w_r \) approximately determine the size of a separated component.

**Lemma 31** (Lemma 2 in [16]). Consider a set \( X \subseteq U \) with \(|X| \leq k \) and \( X \cap R = \emptyset \) which separates \( G \) into \( A \) and \( B \). Then,

\[
||A| - w(A \cap R)| \leq kB,
\]

and the same holds for \( B \).

We compute an \( R \)-splitter with \( B := n/(12k) \), so that \(|R| \leq 12k \). By Lemma 28, there exists an \((X, 2/3)-balanced \) separator of \( G \) into \( A \) and \( B \). If \( X \cap R = \emptyset \), then by Lemma 31,

\[
w(A \cap R) \leq |A| + kB \leq (2/3)|U| + (1/12)|U| = (9/12)|U|,
\]

and the same holds for \( B \). Therefore, letting \( Y := A \cap R \) and \( Z := B \cap R \), we conclude that there exists a partition \( Y, Z \) of \( R \) with \( w(Y), w(Z) \leq (9/12)|U| \) that admits a \( Y--Z \) separator in \( G[U] - R \) of size \( \leq k + 1 \). The algorithm tries all possible such \( Y, Z \) and tries to find a separator for each. If a separator \( S \) is found, then by Lemma 31

\[
|A| \leq w(A \cap R) + kB \leq (9/12)|U| + (1/12)|U| = (10/12)|U|,
\]

so \( S \) is an \((X, 10/12)-balanced \) separator, giving us the necessary constant factor decrease.

Otherwise, if no \( S \) is found, we must have \( X \cap R \neq \emptyset \). In this case, we brute-force over which one of the \(|R| \leq 12k \) vertices belongs in \( X \). If we guess \( r \in R \), then we would like to solve the instance \((U - r, X)\), except we look for a separator of size \( k \) instead of \( k + 1 \).

It is possible for \( U - r \) to be disconnected, in which case only the largest component still needs to be separated, since the other components have size \( \leq (1/2)|U| \). Therefore, if \( U' \) is the largest component of \( G[U] - r \), then we solve the instance \((U', N[U'] \cap X)\) with \( k \) decreased by 1. This is a recursion that is completely contained inside the \((U, X)\) instance; it has nothing to do with the main recursion, so it does not distinguish between even and odd recursion levels.

Overall, an instance with value \( k \) results in \(|R| \leq 12k \) recursive instances of value \( k - 1 \). A straightforward induction shows that this recursion tree has size \( \leq k^{O(k)} \).

The values \( w_r \) can be computed in a distributed setting as follows: every node \( r \in R \) sets \( x_r := w_r \) and every other node \( v \in V - R \) sets \( x_v := 0 \). Then, every node computes \( \sum_{u \in \text{desc}(v)} x_u \) in \( O(\log n) \) shortcut rounds using tree aggregation techniques in [12]. Since every node \( r \in R \) already knows \( \text{sub}_\text{size}(r) \), it can locally compute \( w_r = \text{sub}_\text{size}(r) - \sum_{u \in \text{desc}(r)} x_u \).

The recursion within each \((U, X)\) instance is done sequentially, which results in \( k^{O(k)} \) se-
sequential calls to $s$-VERTEX DISJOINT PATHS for $s \leq k + 1$, taking total time $O(k^O(k))$.

5 Applications

This section proves Theorem 3 restated below.

- **Theorem 3.** Let $G$ be a graph network with treewidth $k$. The problems maximum independent set, minimum vertex cover, chromatic number, and minimum dominating set can be solved in $O(k^O(k)D)$ rounds on network $G$.

For conciseness, we only provide a distributed algorithm for maximum independent set; the algorithms for the other problems are straightforward modifications.

We first introduce our notation for treewidth decompositions. A treewidth decomposition of a graph $G$ is a tree $T$ whose vertices, called bags, are subsets of $V(G)$. The tree $T$ satisfies three properties: (i) the union of vertices over all bags equals $V(G)$; (ii) for each $v \in V$, the set of bags containing $v$ is connected in $T$; (iii) for each edge $(u, v) \in E(G)$, there is a bag containing both $u$ and $v$. The treewidth of a graph $G$ is the minimum $k$ such that there exists a tree decomposition $T$ of $G$ whose bag sizes are at most $k + 1$.

5.1 The Classical Algorithm

Let us now sketch the traditional algorithm for maximum independent set on bounded treewidth graphs. For an input graph $G$ of treewidth $k$, the algorithm first computes a treewidth decomposition of the graph with bag sizes bounded by $O(k)$. Then, the algorithm applies dynamic programming on this treewidth decomposition; we sketch this dynamic program below. This presentation of the dynamic programming algorithm is not the most standard or the most efficient, but it will translate more smoothly when we adapt it to the distributed setting.

The dynamic program. Root the tree $T$ at a root vertex $r \in V(T)$. For each vertex $v \in V(T)$, let its bag be $B_v$. For each bag $B_v$ and subset $I \subseteq B_v$, we will define dynamic programming states $\text{Join}(B_v, I)$. For each vertex $v \in V(T) - \{r\}$ and its parent $p \in V(T)$ in the rooted tree $T$, for subsets $I_v \subseteq B_v$ and $I_p \subseteq B_p$, we will define dynamic programming states $\text{Extend}(B_v, I_v, B_p, I_p)$. These are defined as follows:

1. For each vertex $v \in V(T)$ that is a leaf in the rooted tree $T$ and each $I_v \subseteq B_v$,

   \[
   \text{Join}(B_v, I_v) := \begin{cases} |I_v| & \text{if } I_v \text{ is an independent set in } G[B_v], \\ -\infty & \text{otherwise} \end{cases} \tag{1}
   \]

   Our goal is for the optimal size of the independent set to be the best value of $\text{Join}(B_r, I_r)$, i.e., $\max_{B_v \subseteq I_v} \text{Join}(B_v, I_v)$. Clearly, if $r$ is a leaf (i.e., the tree $T$ is a single vertex), then this is true. Otherwise, we will define $\text{Join}$ for non-leaf vertices later.

2. For each vertex $v \in V(T) - \{r\}$ and its parent $p$, and each $I_v \subseteq B_v, I_p \subseteq B_p$,

   \[
   \text{Extend}(B_v, I_v, B_p, I_p) := \begin{cases} \text{Join}(B_v, I_v) + |I_p - I_v| & \text{if } I_v \cap (B_v \cap B_p) = I_p \cap (B_v \cap B_p) \\ -\infty & \text{otherwise} \end{cases} \tag{2}
   \]

   and $I_v, I_p$ are independent sets in $G[B_v], G[B_p]$.

In other words, we try to extend the state $I_v$ in $B_v$ to the state $I_p$ in $B_p$, but this is only valid if the sets $I_v, I_p$ agree on the vertices shared by $B_v$ and $B_p$, namely $B_v \cap B_p$. 

CVIT 2016
3. For each non-leaf vertex \( p \), let \( \text{children}(p) \) denote the children of \( p \) in \( \mathcal{T} \). For each \( I_p \subseteq B_p \),

\[
\text{Join}(B_p, I_p) := |I_p| + \sum_{v \in \text{children}(p)} \max_{I_v \subseteq B_v} \left( \text{Extend}(B_v, I_v, B_p, I_p) - |I_p| \right)
\]

(3) if \( I_p \) is an independent set in \( G[B_p] \),

\[
- \infty \quad \text{otherwise}
\]

It is a routine exercise in algorithm design on bounded treewidth graphs to argue that this algorithm is correct. The main observation is that in the dynamic program, once \( \text{Join}(B_p, I_p) \) “forgets” the vertices in \( B_v - B_p \) for some \( v \in \text{children}(p) \), these vertices never appear again on any \( \text{Join}(B_{p'}, I_{p'}) \) for any \( p' \) on the path from \( p \) to the root \( r \), due to property (ii) of the treewidth decomposition.

**Recovering the solution.** Thus, \( \max_{I_r \subseteq B_r} \text{Join}(B_r, I_r) =: OPT \) is the size of the maximum independent set in \( G \). To compute the actual maximum independent set, we follow the traditional procedure of “reversing” the dynamic program, starting from the root \( r \):

1. For \( I_r^* \), pick an arbitrary set \( I_r \subseteq B_r \) that satisfies \( \text{Join}(B_r, I_r) = OPT \), i.e.,

\[
I_r^* := \arg \max_{I_r \subseteq B_r} \text{Join}(B_r, I_r).
\]

(4)

2. For each non-leaf vertex \( p \) and each \( v \in \text{children}(p) \), define

\[
I_v^* := \arg \max_{I_v \subseteq B_v} \left( \text{Extend}(B_v, I_v, B_p^*, I_p^*) - |I_p^*| \right).
\]

(5)

3. At the end, the returned maximum independent set is \( \bigcup_{v \in V(\mathcal{T})} I_v^* \).

### 5.2 Distributed Implementation

Suppose the input (and network) graph \( G \) has treewidth \( t \). We first run the treewidth algorithm of Theorem 4 computing a treewidth decomposition with maximum bag size \( O(k) \). However, one immediate issue is that the nodes in each bag.

The first attempt is to adapt the treewidth algorithm of Theorem 4 into computing an actual treewidth decomposition. However, one caveat is that the nodes in each bag of the treewidth decomposition may not be connected in the network \( G \). Therefore, when performing the standard dynamic programming over a treewidth decomposition, the nodes in a bag cannot directly communicate with each other.

We resolve this issue by exploiting the special structure of the treewidth algorithm of Section 4. Recall that the algorithm is recursive: on each recursive instance \( (U, X) \), it either terminates prematurely, concluding that \( \text{tw}(G) > k \), or finds a node set \( S \) of size \( \leq k + 1 \) and, for each connected component \( U' \) in \( G[U] - S \), recursively calls \( (U', N[U'] \cap (X \cup S)) \). In Section 4, we observed several properties of the algorithm that will be helpful, listed below.

1. For each layer of the recursion depth, the sets \( U \) in the instances \( (U, X) \) are connected and node-disjoint.
2. For each instance \( (U, X) \) in the algorithm, \( X \subseteq N[U] \).
3. If instance \( (U, X) \) computes separator \( S \subseteq U \), then every recursive instance \( (U', X') \) called by this instance has \( N[U'] \cap S \not= \emptyset \).
4. The tree in the treewidth decomposition has depth \( O(\log n) \).
5. Every leaf instance \( (U, X) \) in the recursion tree has \( |U| \leq k + 1 \), since the computed separator \( S \) satisfies \( |S| \leq k + 1 \), and \( G[U] - S \) must be empty to end this recursion branch.

We now specify a tree decomposition \( \mathcal{T} \) “implicitly” produced by the algorithm of Section 4. For each instance \( (U, X) \) that produces separator \( S \), create a bag \( B_{U,X} \) in \( \mathcal{T} \) with vertices \( X \cup S \).
It is clear by the algorithm of Section 4 that \( T \) is a treewidth decomposition with maximum bag size \( O(k) \). We would like to apply dynamic programming on this treewidth decomposition \( T \) in a distributed fashion.

**Modifications.** We first augment the algorithm of Theorem [1] so that for each layer of the recursion, for each \((U, X)\) in that layer, all nodes \( v \in V \) know: (i) the depth of that layer, (ii) the ID of a node in \( U \), set to be the smallest ID of a node in \( U \), and (iii) the set \( X \) of size \( O(k) \). Moreover, if \( S \) is the separator computed at instance \((U, X)\), then for each instance \((U', X')\) called recursively at this instance, exactly one of the nodes in \( N[U'] \cap S \) knows the ID of \( U' \) and its neighbors in \( U' \); we say that node \( v \) is in charge of \( U' \). It is clear that all of this can be done in \( kO(1) \) polylog \((n)\) SA rounds.

**The dynamic program.** With this, we implement the dynamic program of Section 5.1 “bottom-up”, from layer \( T := \Theta(\log n) \) to (the single instance in) layer 0 in that order. For layer \( t \in [T] \), for each instance \((U, X)\) in that layer, perform the following:

1. If no nodes in \( U \) have received anything so far, then instance \((U, X)\) is a leaf in the recursion tree. In this case, the computed separator \( S \) in instance \((U, X)\) is exactly \( U \), so the bag corresponding to \((U, X)\) has node set \( X \cup S = X \cup U \). The nodes in \( U \) first learn each other’s IDs as well as the IDs in \( X \); this can be achieved because (i) \(|U| \leq k + 1 \), (ii) \( U \) is connected, (iii) \( X \subseteq N[U] \), and (iv) \(|X| = O(t) \). The bag \( B_{U,X} \) for this instance is a leaf in \( T \) and has node set \( X \cup U \), so for each \( I_{U,X} \subseteq B_{U,X} \), we compute \( \text{Join}(B_{U,X}, I_{U,X}) \) according to [1]. This is \( 2^{O(k)} \) values, one for each \( I_{U,X} \subseteq B_{U,X} \); we then broadcast all of the \( \text{Join}(B_{U,X}, I_{U,X}) \) pairs to \( N[U] \). In total, this takes \( 2^{O(k)} \) distributed rounds.

2. Otherwise, \((U, X)\) must have called some recursive instances after computing its separator \( S \). Any such instance \((U', X')\) satisfies \( N[U'] \cap S \neq \emptyset \), and instance \((U', X')\) has already broadcasted all \( \text{Join}(B_{U',X'}, I_{U',X'}) \) pairs to the unique node in \( U \) in charge of \( U' \). For each node \( v \in U \) and recursive instance \((U', X')\) such that \( v \) is in charge of \( U' \), node \( v \) locally computes, according to [2],

\[
\text{Extend}(B_{U',X'}, I_{U',X'}, B_{U,X}, I_{U,X}) \quad \text{for each } X \subseteq B_{U,X}, X' \subseteq B_{U',X'}.
\]

Note that a node can receive from multiple \((U', X')\), but it can locally compute all such states simultaneously. Then, for each \( I_{U,X} \subseteq B_{U,X} \), each node \( v \) locally computes the sum

\[
\sum_{I_{U',X'} \subseteq B_{U',X'}} \max_{I_{U',X'}} \left( \text{Extend}(B_{U',X'}, I_{U',X'}, B_{U,X}, I_{U,X}) - |I_{U,X}| \right),
\]

where the sum is taken over all instances \((U', X')\) such that \( v \) is in charge of \( U' \), and then the nodes in \( U \) compute an aggregate sum of all these values. Finally, each node can locally compute \( \text{Join}(B_{U,X}, I_{U,X}) \) for each \( I_{U,X} \subseteq B_{U,X} \) using [3]. In total, this step takes \( 2^{O(k)} \) SA rounds.

**Recovering the solution.** To recover the actual maximum independent set, we again follow the process in Section 5.1. This time, we process the layers from layer 0 to layer \( T \).

1. For the initial depth 0 with instance \((U, X)\), recall that all nodes in \( U \) have learned \( \text{Join}(B_{U,X}, I_{U,X}) \) for all \( I_{U,X} \subseteq B_{U,X} \). All nodes in \( U \) compute \( I_{U,X}^* \) according to [1], where the arg \( \max \) breaks ties lexicographically (so that all nodes in \( U \) can agree upon the same \( I_{U,X}^* \)), and then broadcast the set \( I_{U,X}^* \) to their neighbors. Since \( X \cup S \subseteq N[U] \), each node in \( X \cup S \) can learn whether it belongs to \( I_{U,X}^* \). Then, for each node \( v \in U \) in charge of a recursive instance \((U', X')\), node \( v \) locally computes \( I_{U',X'}^* \) according to [5], and sends the set \( I_{U',X'}^* \) to its neighbors in \( U' \). All of this can be done in \( O(k) \) distributed rounds.
2. For each instance \((U', X')\) called by instance \((U, X)\), the nodes in \(U'\) adjacent to the node in charge of \((U', X')\) have received \(I_{U', X'}\). These nodes broadcast the set \(I_{U', X'}\) to all nodes in \(U'\), taking \(O(k)\) SA rounds. Then each node in \(U'\) in charge of a recursive instance performs the same as above.

At the end, every node in \(V\) knows whether it belongs to the maximum independent set \(\bigcup_{U,X} I_{U,X}\). This completes the distributed implementation, which runs in \(2^{O(k)} \log n\) SA rounds. This concludes Theorem 3.

References

1. Eyal Amir. Efficient approximation for triangulation of minimum treewidth. In Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence, pages 7–15. Morgan Kaufmann Publishers Inc., 2001.
2. Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in ak-tree. SIAM Journal on Algebraic Discrete Methods, 8(2):277–284, 1987.
3. Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM Journal on computing, 25(6):1305–1317, 1996.
4. Marek Cygan, Fedor V Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. Parameterized algorithms, volume 3. Springer, 2015.
5. Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In STOC, pages 363–372, 2011.
6. Uriel Feige, MohammadTaghi Hajiaghayi, and James R Lee. Improved approximation algorithms for minimum weight vertex separators. SIAM Journal on Computing, 38(2):629–657, 2008.
7. Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks ii: Low-congestion shortcuts, mst, and min-cut. In Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, pages 202–219. Society for Industrial and Applied Mathematics, 2016.
8. Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed mst and routing in almost mixing time. In Proceedings of the ACM Symposium on Principles of Distributed Computing, pages 131–140. ACM, 2017.
9. Bernhard Haeupler, D Ellis Hershkowitz, and David Wajc. Round-and message-optimal distributed part-wise aggregation. arXiv preprint arXiv:1801.05127, 2018.
10. Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-congestion shortcuts without embedding. In Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, pages 451–460. ACM, 2016.
11. Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Near-optimal low-congestion shortcuts on bounded parameter graphs. In International Symposium on Distributed Computing, pages 158–172. Springer, 2016.
12. Bernhard Haeupler and Jason Li. Faster distributed shortest path approximations via shortcuts. arXiv preprint arXiv:1802.03671, 2018.
13. Bernhard Haeupler, Jason Li, and Goran Zuzic. Minor excluded network families admit fast distributed algorithms. arXiv preprint arXiv:1801.06237, 2018.
14. Samir Khuller and Baruch Schieber. Efficient parallel algorithms for testing k and finding disjoint s-t paths in graphs. SIAM Journal on Computing, 20(2):352–375, 1991.
15. Jens Lagergren. Efficient parallel algorithms for graphs of bounded tree-width. Journal of Algorithms, 20(1):20–44, 1996.
A Omitted Proofs in Section 2

Proof (Lemma 7). Let \( H_1, \ldots, H_\ell \) be the connected, disjoint subgraphs in an SA instance. The reduction from SA to PA uses the Heads/Tails clustering technique from parallel graph contraction algorithms. We sketch the Heads/Tails clustering algorithm below, which, given a graph, contracts each connected component into a single vertex. For \( O(\log n) \) rounds, each vertex flips Heads/Tails with probability \( 1/2 \) each and broadcasts the flip to its neighbors. If a vertex flips Tails and has a neighbor with Heads, it notifies one of its Heads neighbors. Afterwards, every Heads vertex and its Tails neighbors who notified it contract into a single vertex. Following the standard analysis, w.h.p., every connected component contracts to a single vertex after \( O(\log n) \) iterations.

Now we describe the distributed SA algorithm. As input, each node in each subgraph \( H_i \) knows its neighbors in that subgraph. Our goal is for the nodes in each \( H_i \) to agree on a common ID, unique to each \( H_i \). Then, a single round of PA solves the desired aggregation task.

At a high level, we want to run a distributed version of the Heads/Tails algorithm on the graph \( H_1 \cup \cdots \cup H_\ell \). On each iteration, for each vertex \( v \) in the contracted graph, the nodes in the original graph that contracted to \( v \) form a part for a PA round. In particular, every node in a part should know its (unique) part ID. At the beginning, each node is its own part, and it can set its part ID to be its node ID.

We now describe each of \( O(\log n) \) steps of the clustering algorithm. In \( O(1) \) PA rounds, each part collectively decides on a Heads/Tails flip for that part. Then, every node broadcasts its part ID and its flip to all its neighbors in its own subgraph. Then, in \( O(1) \) PA rounds, any node that received a Heads flip from a neighbor, and whose own part flipped Tails, notifies its part of the part ID received by such a neighbor. Then, each part that flipped Tails collectively decides on a common part ID (e.g., the minimum one) and sets its own part ID to be that one. The new parts are the nodes with a common part ID; clearly, every part is still connected. Finally, after \( O(\log n) \) iterations, each taking \( O(1) \) PA rounds, all nodes within each subgraph \( H_i \) contracted to a single vertex in the Heads/Tails algorithm, which means that they have agreed on a common part ID.

\[ \text{\textless} \]

Proof (Lemma 9). We can assign arbitrary weights to the edges of \( H \) use the MST algorithm of [7], which runs a modified Boruvka’s algorithm with the current contracted components as the parts; for details, see [7, 12].

\[ \text{\textless} \]

Proof (Lemma 10). Computing subtree aggregation in the tree rooted at \( v_r \) is covered in [12]. To determine the parent of each node except the root, we can set \( x_i \leftarrow 1 \) for each node \( v_i \) with sum as the aggregate. For each node \( v_i \in V(T) - v_r \), its parent is precisely the unique neighbor \( u \) whose aggregate \( \bigoplus_{j \in T(u)} x_j \) is larger than the aggregate \( \bigoplus_{j \in T(v_i)} x_j \) at \( v_i \).

\[ \text{\textless} \]

Proof (Lemma 11). For prefix aggregation, let \( v_\ell \) be the root of the tree \( P \). Then, subtree aggregate \( \bigoplus_{j \in T(v_\ell)} x_j \) is exactly the prefix aggregate \( \bigoplus_{i \leq \ell} x_j \), so we can apply Lemma 10. For each node \( v_i \) to learn the index \( i \), we set \( x_i \leftarrow 1 \) for all \( i \) so that value \( i \) is exactly the prefix
aggregate $\bigoplus_{j \leq i} x_j$. Finally, for suffix aggregation, we compute subtree aggregates with $v_1$ as the root instead. ▶

**Proof (Lemma 12).** First, compute a spanning tree $T \subseteq H$ using Lemma 9. Root the tree $T$ at $t$, and set $x_s \leftarrow 1$ and $x_v \leftarrow 0$ for all other nodes $v \in V(H) - s$. Now compute subtree aggregates with sum as the operator. Observe that a node has nonzero subtree sum iff it is on the path from $s$ to $t$. Finally, each node $v$ with $\sum_{j \in T_v} x_j \neq 0$ sends a message to its parent in the rooted tree, so that every node on the path learns its predecessor and successor. ▶

### B Omitted Proofs in Section 3

**Proof (Claim 14).** Since $G$ has treewidth $k$, there exists a treewidth decomposition with maximum bag size $k + 1$. We now construct a treewidth decomposition $T'$ of $G'$ with maximum bag size $(k+1)\ell$, which is sufficient to prove the claim. Starting from $T$, for each $v \in G$, we replace all occurrences of $v$ in bags in $T$ with the vertices $v^1, \ldots, v^\ell$. Clearly, the maximum bag size is now at most $(k + 1)\ell$. We now claim that the new decomposition $T'$ is a treewidth decomposition of $G'$. Property (i) is clearly satisfied. For property (1), for each vertex $v^r \in V(G')$, the set of bags containing it is precisely the set of bags containing $v$ in $T$, which is connected, proving property (2). Finally, for property (3), for vertices $u^i, v^j \in V(G')$ adjacent in $G'$, either (i) $u = v$, in which case any bag containing $u$ in $T$ contains both $u^i$ and $v^j$, or (ii) $u \neq v$, in which case there must be a bag in $T$ containing both $u$ and $v$, and this bag in $T'$ contains both $u^i$ and $v^j$.

**Proof (Lemma 18).** By Corollary 17, the existence of an augmenting path is equivalent to $s$ and $t$ being strongly connected in $G_{res}$. Since the transformation from $G_{res}$ to $G_R$ preserves SCCs, this is also equivalent to $s$ and $t$ being strongly connected in $G_R$. Also, there is always a $s \rightarrow t$ path in $G_R$, so strong connectivity of $s$ and $t$ is equivalent to the existence of a directed $s \rightarrow t$ path in $G_R$. Therefore, we prove the following statement instead: there is a directed $s \rightarrow t$ path in $G_R$ iff there exists $s$-reachable $\beta_s$ and $t$-reachable $\beta_t$ and a directed $\beta_s \rightarrow \beta_t$ path in $G_B$. The rest of the proof resembles the proof of Theorem 4.1 in [14].

For the if direction, suppose there is a directed $\beta_s \rightarrow \beta_t$ path in $G_B$ from $s$-reachable $\beta_s$ to $t$-reachable $\beta_t$; we will transform this path into an $s \rightarrow t$ path in $G_R$. We replace each arc $(\beta_x, \beta_y)$ with a directed path from $\beta_x$ to $\beta_y$ in $G_R$ as follows. By definition of $G_B$, there is $y \in [r]$ satisfying $r_y^i \geq l_y^i$. We replace arc $(\beta_x, \beta_y)$ with the path composed of the arc $(r_y^i, l_y^i)$, the left path along $P_y$ from $r_y^i$ to $l_y^i$, and finally the arc $(l_y^i, \beta_y)$. Finally, add the arcs $(s, \beta_s)$ and $(\beta_x, t)$ to the path, completing the $s \rightarrow t$ path.

For the only if direction, suppose there is a directed $s \rightarrow t$ path $P$ in $G_R$; without loss of generality, assume that $P$ is simple. A simple path in $G_R$ consists of vertices $\beta_i$ with a leftward path along some $P_j$ in between every two consecutive $\beta_i$. Let the $\beta_i$ vertices on $P$ be $\beta_{x_1}, \ldots, \beta_{x_t}$ from left to right; by definition, $\beta_{x_1}$ is $s$-reachable and $\beta_{x_t}$ is $t$-reachable. We now construct a (not necessarily simple) path from $\beta_{x_1}$ to possibly a different $t$-reachable vertex in $G_B$.

For $i \in [t - 1]$, let $y_i$ be such that the path $P$ travels along $P_{y_i}$ from $\beta_{x_i}$ to $\beta_{x_{i+1}}$, and let $y$ be an arbitrary integer in $[r]$. First, set $x'_1 \leftarrow x_1$. Then, one by one, for $i$ from 2 to $t$, we will replace the vertex $\beta_{x_i}$ with a vertex $\beta_{x'_i}$ such that (i) either $\beta_{x'_i} = \beta_{x_i}$ or the arc $(\beta_{x'_i}, \beta_{x'_i})$ exists in $G_B$, and (ii) we have $r_{y_i}^{x'_i} \geq r_{y_i}^{x_i}$. Note that condition (ii) is satisfied by definition for $i = 1$.

Fix an $i \in [2, t - 1]$: we assume that the invariant is satisfied at $i$. Since $r_{y_i}^{x'_i} \geq r_{y_i}^{x_i}$, we can travel from $\beta_{x'_i}$ to $\beta_{x_{i+1}}$ along path $P_{y_i}$. Consider the arc $(x_i, x'_i) \in D_{y_i+1}$; we first assume that it exists. By condition 2(c) of the construction of the bridge graph with $\beta_{x_i+1}$ as $\beta_z$, we have

---

5 For our definition of treewidth and treewidth decomposition, we refer the reader to Section 5.
Therefore, setting the two properties for index \(i + 1\) must exist. Therefore, setting \(\beta_{x_i} \leftarrow \beta_x\) maintains the two properties for index \(i + 1\).

Since \(\beta_{x_i}\) is \(t\)-reachable, we have \(r_y^x = |P_y|\) for all \(y \in [r]\), i.e., the rightmost out-neighbor of vertex \(\beta_x\) is \(t\) on each path \(P_y\). Since \(r_y^x \geq r_y^x = |P_y|\), vertex \(\beta_x\) is also \(t\)-reachable. We remove duplicates from the sequence \(\beta_{x_1}, \beta_{x_2}, \ldots, \beta_{x_i}, \ldots\), obtaining our desired path in \(G_B\) from \(s\)-reachable \(\beta_{x_1}\) to some \(t\)-reachable vertex.

**Proof (Lemma 20).** We define iteration \(i\) of the contraction algorithm as the iteration where the components of \(D_j\) are contracted. Also, let \(G_i^B\) be the bridge graph after iteration \(i \in [r]\), and \(G_B^0 := G_B\), and let \(D_j^i\) be the edge set \(D_j\) in \(G_B^i\).

For the only if direction, we first show that if two vertices \(\beta_i, \beta_x\) contract to the same vertex, then the bridges \(B_i, B_x\) are strongly connected in \(G_{res}\). Before the contraction algorithm begins (i.e., before iteration 1), this is clearly true. On iteration \(i\), suppose the statement holds for the bridge graph right before iteration \(i\). Iteration \(i\) contracts the bridges inside each connected component of \(D_j^{i-1}\), which, by Claim \(19\), is strongly connected. Therefore, the statement holds after iteration \(i\) as well. Applying induction on \(i\) proves the statement at the end of the algorithm.

Finally, taking \(\beta_i\) and \(\beta_x\) to be an \(s\)-reachable and \(t\)-reachable vertex finishes the only if direction.

For the if direction, we temporarily abuse notation, referring to vertices \(\beta_i\) in \(G_R\) and \(G_B^i\) as (super-) bridges as well as \(B_i\). For bridge \(\beta_i\), consider all directed simple paths in \(G_R\) that start from \(\beta_i\) and end at some \(t\)-reachable bridge. Define \(dist(\beta_i)\) to be the minimum possible number of bridges \(\beta_x\) on such a path, minus 1. In particular, \(dist(\beta_i) = \infty\) iff there is no such path, and \(dist(\beta_i) = 0\) iff \(\beta_i\) is \(t\)-reachable. Also, for bridges \(\beta_i, \beta_x\), we say that \(\beta_i\) can directly reach \(\beta_x\) in \(G_R\) if there is a directed \(\beta_i \rightarrow \beta_x\) path in \(G_R\) with no other bridges inside. We now prove, by induction on \(d \geq 0\), that all bridges \(\beta_i\) with \(dist(\beta_i) = d\) contract to the a vertex containing an \(t\)-reachable bridge. Since \(s\) and \(t\) are strongly connected in \(G_{res}\), there is an \(s\)-reachable bridge \(\beta_i\) with finite \(dist(\beta_i)\), proving the if direction.

The base case \(d = 0\) is trivial; we now consider the case \(d = 1\). Suppose a bridge \(\beta_i\) satisfies \(dist(\beta_i) = 1\). Then, exiting from bridge \(\beta_i\), we can move left along some path \(P_y\) and then enter an \(t\)-reachable bridge \(\beta_x\). Note that a \(t\)-reachable bridge necessarily satisfies \(r_y^x = |P_j|\) for all \(j \in [r]\), i.e., the rightmost out-neighbor of bridge \(\beta_x\) is \(t\) on each path \(P_j\). By definition of \(G_j\), this means that for all \(j\), there is an arc in \(D_j\) from \(\beta_i\) to \(\beta_x\), or possibly a different \(t\)-reachable bridge due to tie-breaking. In particular, this arc is in \(D_1\), so on iteration 1 of the algorithm, \(\beta_i\) and \(\beta_x\) already contract to the same vertex.

Now suppose \(dist(\beta_i) = d + 1\) for \(d \geq 1\). First, if \(\beta_i\) contracts to the same component as some other \(\beta_x\) with \(dist(\beta_x) \leq d\), then by induction, \(\beta_x\) contracts to a component containing a \(t\)-reachable bridge. \(\beta_i\) contracts to this same component, completing the inductive step.

Otherwise, consider the directed path \(P\) from \(\beta_i\) to \(t\) in \(G_R\) achieving \(dist(\beta_i) = d + 1\), and suppose \(\beta_i, \beta_{x_0}, \beta_y\) are the first three bridges on \(P\). Suppose that, from \(\beta_{x_0}\) to \(\beta_y\), the path \(P\) walks left along path \(P_j\) in \(G_R\); in particular, this means that \(r_y \geq l_y\). Furthermore, either the arc \((\beta_i, \beta_{x_0})\) is in \(D_j\), or some arc \((\beta_i, \beta_x)\) exists in \(D_j\) such that \(r^j_x \geq r^j_{x_0}\) and \(\beta_i\) can directly reach \(\beta_x\) in \(G_R\); in the former case, we set \(x := x_0\). The inequalities \(r_x^j \geq r_x^j \geq l_y\) imply that \(\beta_x\) can directly reach \(\beta_y\), and since \(dist(\beta_y) = d - 1\), we have \(dist(\beta_x) \leq d\). Now consider iteration \(j\) of the algorithm, which contracts all connected components in \(D_j^{j-1}\) in \(G_B^{j-1}\). Before iteration \(j\), let \(\beta_x\) and \(\beta_y\) in \(G_B^{j-1}\) denote the super-bridges that \(\beta_i\) and \(\beta_y\) have contracted to; we assume \(\beta_x \neq \beta_x\), otherwise we are in the first case. Observe that the corresponding super-bridge \(B_{\beta_x}\) can still directly reach \(B_{\beta_x}\). In particular, if \(\beta_x\) is the super-bridge for which \((\beta_x, \beta_x)\) is an arc in \(D_j^{j-1}\), then \(r_{\beta_x}^j \geq r_{\beta_x}^j \geq l_y^j \geq l_y^j \). In particular, some original bridge \(\beta_{x_0}\) that contracts to the super-bridge \(\beta_x^j\) satisfies \(r_{\beta_x}^j = r_{\beta_x}^j \geq l_y^j \), which again implies that \(dist(\beta_{x_0}) \leq d\). Then,
on iteration $j$, super-bridges $\beta_{x'},\beta_{x'}$ contract together, so bridges $\beta_i,\beta_{x_1}$ contract to the same super-bridge. The fact that $\text{dist}(\beta_{x_1}) \leq d$ completes the inductive step.