

AN $O(\log n \log \log n)$ SPACE ALGORITHM FOR UNDIRECTED ST-CONNECTIVITY

VLADIMIR TRIFONOV*

Abstract. We present a deterministic $O(\log n \log \log n)$ space algorithm for undirected st-connectivity. It is based on a space-efficient simulation of the deterministic EREW algorithm of Chong and Lam [6], an approach suggested by Prof. Vijaya Ramachandran, and uses the universal exploration sequences for trees constructed by Koucký [13]. Our result improves the $O(\log^{4/3} n)$ bound of Armoni et al. [2] and is a big step towards the optimal $O(\log n)$. Independently of our result and using a different set of techniques, the optimal bound was achieved by Reingold [18].

Key words. undirected st-connectivity, space-bounded computation

AMS subject classifications. 05C40, 05C85, 68Q25, 68Q15

1. Introduction. The problem we are concerned with is st-connectivity in an undirected graph G with n vertices (USTCONN), i.e. given two vertices s and t of G we want to answer the question whether there is a path between s and t . This is one of the most basic graph problems with applications ranging from image processing and VLSI design to solving more complex graph problems. Furthermore, st-connectivity plays an important role in complexity theory because directed st-connectivity (STCONN) is **NL**-complete [20] and USTCONN is **SL**-complete [14].

Linear time and space sequential algorithm for solving even the harder STCONN problem have been known for a long time [22]. The problem of developing more efficient space and parallel algorithms was posed. The result of Aleliunas et al. [1] shows that USTCONN can be solved in $O(\log n)$ space with a randomized algorithms with one-sided error, i.e. an algorithm which produces an answer in polynomial time and, if the two vertices are not connected, then the algorithm is correct, otherwise it answers incorrectly with probability at most $1/2$. The starting point of deterministic space-efficient sequential algorithms is the $O(\log^2 n)$ space algorithm for STCONN of Savitch [20]. For a long time this was the best result even for USTCONN. The space bound for undirected graphs was first improved by Nisan et al. [16] to $O(\log^{3/2} n)$ and then to $O(\log^{4/3} n)$ by Armoni et al. [2]. Both of these results depend on the efficient construction of universal traversal sequences by Nisan [15]. Finally, simultaneous to our result and using different set of techniques, the space complexity of USTCONN was shown by Reingold [18] to be $O(\log n)$.

Developing efficient parallel algorithms for USTCONN has a very rich history. The situation here is complicated further by the existence of multiple models of parallel computation. The models from the PRAM family are generally considered to be of great theoretical value. The results of [10, 19, 4, 11, 12] are concerned with the CREW PRAM model, [21, 3, 7, 8] with the CRCW PRAM model, and [9, 6, 5, 17] with the EREW PRAM. The state of the art in parallel algorithms for USTCONN are the results of Chong, Han, and Lam [5], which shows that the problem can be solved on the EREW PRAM in $O(\log n)$ time with $O(m + n)$ processors, and Pettie and Ramachandran [17], which demonstrates a randomized EREW PRAM algorithm running in time $O(\log n)$ with optimal number of processors. The algorithms of [5]

*Department of Computer Sciences, University of Texas at Austin, supported in part by NSF grant CCF-0430695 and by Texas Advanced Research Program Grant 003658-0029-1999.

and [17] actually solve the harder problem of finding the minimum spanning tree of a weighted graph.

The starting point of our algorithm is the $O(\log n \log \log n)$ time deterministic EREW PRAM algorithm with $O(m + n)$ processors of Chong and Lam [6], which we call the CL algorithm. This parallel algorithm can be trivially simulated sequentially in linear space. We use the sequential algorithm to define a mathematical structure called configuration, which captures the state of the algorithm. We define also a sequence of configurations, such that every element of this sequence corresponds to the state of the sequential algorithm at certain point of its execution. We use the sequence of configurations to trivially define an $O(\log^2 n)$ space algorithm, which instead of storing all of its current state, recomputes parts of it when it needs them. This technique is standard for designing space-efficient algorithms. Finally, we modify the $O(\log^2 n)$ space algorithm into an algorithm which uses $O(\log n \log \log n)$ space.

The possibility of simulating parallel algorithms for USTCONN space-efficiently was suggested to the author by Prof. Vijaya Ramachandran in 2000. She conjectured an $O(\log n \log \log n)$ space algorithm derived from the CL algorithm and an alternate simple $O(\log^{3/2} n)$ space algorithm derived from the algorithm of Johnson and Metaxas [11], by using the max-degree hooking scheme of [6]. She observed that the step needing derandomization in [16] is not necessary in a tree-based hook and contract approach, because the trees automatically give rise to disjoint clusters of vertices. The max-degree hooking scheme employed by [6] gives the additional benefit that small trees have small neighborhoods. The main challenge was to implement the levels of recursion, so that they process small trees in $o(\log n)$ space. Solving this problem is the main contribution of this paper.

In the algorithm presented here the space of a level of recursion is between $\Omega(\log \log n)$ and $\Theta(\log n)$, depending on the level. A key tool for our method are the exploration walks on trees defined by Koucký [13]. Exploration walks on trees are similar to the Euler tour technique used by Tarjan and Vishkin [23] in the parallel context. These walks play the role of the edge-list plugging technique and pointer jumping employed by the CL algorithm, because they allow us to traverse trees very efficiently.

Section 2 contains a high-level overview of the CL algorithm. Section 3 defines formally a labeled multi-graph and operations on it. Furthermore, it provides a complete description of the sequential version of the CL algorithm. Section 4 gives detailed description of the space-efficient implementation of the CL algorithm. Section 5 proves the correctness of the sequential CL algorithm. The proofs in section 5 are from [6] and for completeness are adapted here to our framework. Although we make frequent references to [6], the exposition in this paper is self-contained, except for Proposition 3.4 whose proof is not too difficult.

Define $[k] = \{1, \dots, k\}$ and the double exponential function $\text{dexp}(x) = 2^{2^x}$. All logarithms in the paper are of base 2.

2. The Chong–Lam algorithm. This section gives a high-level overview of the Chong–Lam (CL) algorithm [6] to motivate the definitions in the next section. The CL algorithm uses a hook and contract approach. There are several phases of hooking and contraction. Before every phase every vertex of the current graph is in exactly one of three states – active, inactive, and done; all active and inactive vertices have non-zero degree, the done vertices have zero degree, and there are no multi-edges between active vertices; the inactive vertices are organized in a set of hooking trees. A hooking tree is a subgraph of the current graph, which is a tree and whose internal

edges, which we call hooking edges, are obtained as described below. Initially all vertices with non-zero degree are active, and the rest are done.

In a hooking phase the active vertices in parallel choose to hook to one of their current neighbors, establishing their hooking edges, and thus either become part of existing hooking trees or form new ones. The fact that the connected components formed by the hooking of vertices are trees is ensured by the hooking scheme of the CL algorithm. This hooking scheme uses an ordering $<_d$ of the vertices such that $u <_d v$ iff the degree of u is less than the degree of v or they are the same, but u is less than v , thinking of the vertices as elements of \mathbb{N} . To choose their hooking edges the active vertices of the graph perform in parallel two consecutive, synchronized steps. First, if a vertex v has a neighbor larger according to $<_d$ than itself, then v hooks to the largest such neighbor. Second, if after the first step all neighbors of v hooked to it, then v hooks to itself, i.e. it does not choose a hooking edge. Otherwise, if after the first step a neighbor u of v hooked to a vertex different than v , then v hooks to u .

In a contraction phase some of the current hooking trees are contracted to a representative vertex. The representative vertex is the only vertex in the tree which hooked to itself. Which trees are contracted is determined by a parameter, which depends on the phase and sets an upper bound on the total degree, i.e. the sum of the degrees of the vertices, of the trees which are contracted. For every contracted tree its representative becomes a new active vertex and the rest of its vertices become done. Also all multi-edges between new active vertices are cleaned-up. Finally, the vertices of every uncontracted tree become inactive.

The processing required by a hooking phase is performed in parallel time $O(\log d)$, where d is the degree of the active vertex, using pointer jumping. The important part of a contraction phase is checking the degree of a hooking tree. In parallel this could be done in $O(\log c)$ time, where c is the contraction parameter, by using pointer jumping and a constant time edge-list plugging technique.

Finally the CL algorithm is given by the following recursive procedure. Here **MaxHook** and **Contract**(c) denote correspondingly a hooking and a contracting phase with parameter c .

```
procedure Connect( $k$ )
  MaxHook;
  if  $k > 0$  then
    Contract(dexp( $k$ ));
    Connect( $k - 1$ );
    Connect( $k - 1$ );
  Contract(dexp( $k+1$ ));
```

The correctness of the CL algorithm ensures that a call to **Connect**($\lceil \log \log n \rceil$) contracts every connected component of the graph to a single vertex and all the other vertices are organized in a set of rooted parent trees such that the root of the tree of a vertex u is the vertex to which the connected component of u contracted.

We simulate the CL algorithm trivially with a sequential algorithm using linear space. We fix an ordering on the edges incident to a vertex and instead of performing the hooking in parallel for all active vertices, we do it sequentially for each of them. This is possible, because by changing the hooking scheme of CL slightly we can ensure that the hooking of an active vertex does not depend on the hooking of the other active vertices. The new hooking strategy gives preference to neighbors which are inactive or have an inactive neighbor, but it still ensures that small trees composed of active vertices have small degree. The details of the sequential algorithm are given

in the following section, and space-efficient versions of the same algorithm are given in section 4.

3. Definitions.

3.1. Multi-graphs and exploration walks. An undirected multi-graph is a graph with possibly multiple edges between two vertices and such that every edge has a label on each side, where the labels of the edges incident to a vertex v have distinct labels on the side of v . We also have a single self-loop with label 0 at every vertex. Formally we have

DEFINITION 3.1. An undirected multi-graph G is a triple $\langle V, \delta, \mu \rangle$, where V is a set, $\delta : V \rightarrow \mathbb{N}$, and $\mu : E \rightarrow E$ is a bijection such that $\mu(\mu(e)) = e$ and $\mu(v, 0) = (v, 0)$, where $E = \{(v, i) : v \in V \text{ and } 0 \leq i \leq \delta(v)\}$.

V is the set of vertices of G , E is the set of edges of G , $\delta(v)$ is the degree of v , and $\mu(e)$ is the reverse edge of e . For an edge e , call the set $\{e, \mu(e)\}$ an undirected edge.

Let $\eta : E \rightarrow V$ and $\beta : E \rightarrow \mathbb{N}$ be the first and the second component of μ , respectively. Then $\eta(v, i)$ is the i -th neighbor of v , i is the label of the edge (v, i) , and $\beta(v, i)$ is its back-label.

Define the size of G , $\text{size}(G)$, to be $|V|$.

In the following a graph means undirected multi-graph.

DEFINITION 3.2. A graph $G' = \langle V', \delta', \mu' \rangle$ is a subgraph of a graph $G = \langle V, \delta, \mu \rangle$, if $V' \subseteq V$ and for every $u, v \in V'$,

$$|\{i : \eta'(u, i) = v\}| \leq |\{i : \eta(u, i) = v\}|.$$

Define (simple) path, connected vertices, forest and tree in the usual way.

DEFINITION 3.3. Let G be a graph. Let $\Delta : E \times \mathbb{Z} \rightarrow E$ be such that $\Delta((v, i), j)$ changes by j the label of the edge (v, i) . More precisely for $i \neq 0$,

$$\Delta((v, i), j) = (v, 1 + (i - 1 + j \bmod \delta(v))).$$

Define $\Gamma_{G,k}, \Gamma'_{G,k} : E \rightarrow E$ inductively on $k \geq 0$. First $\Gamma_{G,0}(e) = \Gamma'_{G,0}(e) = e$. Now let

$$\begin{aligned} \Gamma_{G,k+1}(e) &= \Delta(\mu(\Gamma_{G,k}(e)), 1), \\ \Gamma'_{G,k+1}(e) &= \mu(\Delta(\Gamma'_{G,k}(e), -1)). \end{aligned}$$

The sequence $\Gamma_{G,\leq l}(e) = \langle \Gamma_{G,0}(e), \Gamma_{G,1}(e), \dots, \Gamma_{G,l}(e) \rangle$ is called the exploration walk of length $l + 1$ starting from the edge e . We will also refer to the corresponding infinite sequence as the exploration walk. Let $e_k = (v_k, i_k) = \Gamma_{G,k}(e)$. Then v_k and e_k are correspondingly the k -th vertex and the k -th edge visited by the exploration walk starting from e . We have the equivalent notions for the reverse exploration walk, where Γ is replaced with Γ' .

Exploration walks were introduced by Koucký [13]. We define them only for the exploration sequence which always changes by one the label of the current edge because this is the case of interest to us. The fact that exploration walks are reversible, i.e. $\Gamma'_{G,l}(\Gamma_{G,l}(e)) = e$, was noticed by Koucký and is what makes them important to us. We will use the following property of exploration walks on trees.

PROPOSITION 3.4 ([13]). Let G be a tree with at most one undirected edge between any two vertices, $e = (v, i) \in E$, $i \neq 0$, be an edge of G , and $l = 2(\text{size}(G) - 1)$. Then

the exploration walk $\Gamma_{G, \leq l-1}(e)$ of length l starting from e visits every edge of G which is not a self-loop exactly once. Furthermore, this is the shortest exploration walk which visits v exactly $\delta(v) + 1$ times.

3.2. Operations on graphs.

3.2.1. Configuration. A configuration is the state of the sequential algorithm outlined at the end of section 2. Formally

DEFINITION 3.5. A configuration is a tuple $\mathcal{C} = \langle G, A, I, D, H, R \rangle$, where G is a graph with $V = [n]$, for some $n \in \mathbb{N}$. A, I , and D form a partition of V . $v \in D$ iff $\delta(v) = 0$. $H : V \rightarrow \mathbb{N}$ and $R : V \rightarrow V$ are such that $H(v) \leq \delta(v)$ and if $R(v) \neq v$, then $v \in D$.

The elements of A, I , and D are called correspondingly the active, the inactive, and the done vertices of G . For $u \in V$, $(u, H(u))$ is the hooking edge of u , and $R(u)$ is the immediate representative of u .

We require that H and R do not have non-trivial cycles in the following sense. Let $v_1, \dots, v_k \in V$, $k \geq 2$. Then

- (i) if $v_{i+1} = \eta(v_i, H(v_i))$, $i \in [k-1]$, and $v_1 = \eta(v_k, H(v_k))$, then $H(v_1) = 0$,
- (ii) if $v_{i+1} = R(v_i)$, $i \in [k-1]$, and $v_1 = R(v_k)$, then $R(v_1) = v_1$.

We also require that there is at most one undirected edge between any two active vertices, i.e. if $u, v \in A$, then $|\{i : \eta(v, i) = u\}| \leq 1$, and there is no hooking edge from an inactive to an active vertex, i.e. if $u \in I$, then $\eta(u, H(u)) \in I$.

Define the representative of v according to R to be

$$\text{rep}_R(v) = \begin{cases} v, & R(v) = v, \\ \text{rep}_R(R(v)), & \text{otherwise.} \end{cases}$$

Definition 3.5(ii), ensures the correctness of this definition.

DEFINITION 3.6. Let $\mathcal{C} = \langle G, A, I, D, H, R \rangle$ be a configuration. H defines a subforest $F = \langle V, \delta_F, \mu_F \rangle$ of G , called the hooking forest of \mathcal{C} , with at most one undirected edge between any two vertices in the following way. Fix $v \in V$. Let ε be 1, if $H(v) \neq 0$, and 0, otherwise. Let $0 < i_1 < \dots < i_k$ be such that

$$\{i_1, \dots, i_k\} = \{i : \exists u \in V \text{ such that } \mu(u, H(u)) = (v, i)\}.$$

First define $\delta_F(v) = k + \varepsilon$. Now define $\eta_F(v, j) = \eta(v, i_j)$, for $1 \leq j \leq k$, and, if $\varepsilon = 1$, $\eta_F(v, k + \varepsilon) = \eta(v, H(v))$. Finally define $\beta_F(v, j) = i$, where $\eta_F(v, j) = u$ and $\eta_F(u, i) = v$.

Let T be a maximal connected subtree of the forest F . We call T a hooking tree in \mathcal{C} . The root of T , $\text{root}(T)$, is the only vertex v in T such that $H(v) = 0$. The degree of T , $\text{deg}(T)$, is $\sum_{v \in V_T} \delta(v)$. For a vertex $v \in V$ we denote with T_v the subtree of F which contains v .

The correctness of this definition and the fact that F is a forest with at most one undirected edge between any two vertices follow from Definition 3.5(i).

3.2.2. Hooking. We will define $\text{Hook}(\mathcal{C})$ so that, if \mathcal{C} describes the state of the sequential algorithm, then $\text{Hook}(\mathcal{C})$ is its state after a hooking phase.

Being elements of \mathbb{N} , the elements of V are ordered. Define the linear ordering $<_d$ on V so that

$$u <_d v \text{ iff } \delta(u) < \delta(v) \text{ or } \delta(u) = \delta(v) \text{ and } u < v.$$

The hooking operation $\text{Hook}(\mathcal{C})$ produces the configuration $\langle G, A, I, D, H', R \rangle$ defined in the following way. If v is inactive, then $H'(v) = H(v)$. If v is active, let $v_1, \dots, v_{\delta(v)}$ be the neighbors of v , i.e. $v_i = \eta(v, i)$. For the rest of the definition when we have to choose an index i , we always pick the smallest one with the corresponding property. If v has an inactive neighbor v_i , let $H'(v) = i$. If all neighbors of v are active, let v_i be the largest according to $<_d$ amongst the neighbors of v . If $v <_d v_i$, let $H'(v) = i$. If all neighbors of v are active and smaller than v according to $<_d$, then if v has a neighbor v_i which has an inactive neighbor, let $H'(v) = i$. If all neighbors of v and their neighbors are active, then if v has a neighbor v_i which has a neighbor larger than v according to $<_d$, let $H'(v) = i$. Finally, if all neighbors of v and their neighbors are active and smaller than v according to $<_d$, define $H'(v) = 0$.

The hooking strategy described above differs from the hooking strategy of the CL algorithm because it gives preference to neighbors which are inactive or have an inactive neighbor. For example, in our hooking strategy a vertex hooks to an inactive neighbor, if it has one, regardless of its degree. In the hooking strategy of the CL algorithm, a vertex hooks to its largest according to $<_d$ neighbor, regardless of its state. The new hooking strategy does not change the correctness of the CL algorithm because first an active vertex still hooks to itself iff all of its neighbors are active and hooked to it, and second along a sequence of hooking edges of active vertices the vertices increase according to $<_d$ the same way as in the CL algorithm. The reason we chose the new strategy is to ensure that we do not lookup the degree of an inactive vertex. The essential properties of the hooking operation are given by the following lemmas. The proofs are the same as in [6].

LEMMA 3.7. *Let $k \geq 3$ and $v_i \in A$, $i \in [k]$, are distinct and such that $v_{i+1} = \eta(v_i, H'(v_i))$, $i \in [k-1]$. Then $v_1 <_d \max_d\{v_{k-1}, v_k\}$.*

Proof. By induction on k . For $k = 3$, the statement holds, because v_1 hooked to v_2 , either because $v_2 >_d v_1$ or because $v_2 <_d v_1$, but v_2 had an inactive neighbor or an active neighbor larger than v_1 . Since v_2 is hooked to v_3 , the second must be the case. So $v_1 <_d \max_d\{v_2, v_3\}$. For the inductive step, we have that $v_{k-1} <_d \max_d\{v_k, v_{k+1}\}$ and $v_1 <_d \max_d\{v_{k-1}, v_k\}$, and so $v_1 <_d \max_d\{v_k, v_{k+1}\}$. \square

COROLLARY 3.8. *Hook(\mathcal{C}) is a configuration.*

Proof. Since Hook only changes the hooking edges of active vertices, the only thing we have to check is that it does not create non-trivial cycles of active vertices. Assume that there is $k \geq 2$, and $v_i \in A$, $i \in [k]$, distinct and such that $v_{i+1} = \eta(v_i, H'(v_i))$ and $v_1 = \eta(v_k, H'(v_k))$. The case $k = 2$ is impossible because we must either have $v_1 <_d v_2$ or $v_1 >_d v_2$. In the first case v_2 will hook to v_1 , only if v_1 hooked to a vertex different than v_2 . The second case is also impossible, hence $k \geq 3$. By Lemma 3.7 $v_1 <_d \max_d\{v_{k-1}, v_k\}$ and $v_2 <_d \max_d\{v_k, v_1\}$. If $v_{k-1} <_d v_k$, then $v_1, v_2 <_d v_k$, a contradiction with Lemma 3.7 for v_k, v_1 , and v_2 . If $v_k <_d v_{k-1}$, then $v_1, v_k <_d v_{k-1}$, a contradiction with Lemma 3.7 for v_{k-1}, v_k , and v_1 . Thus we must have that $H'(v_1) = 0$. \square

LEMMA 3.9. *Let T be a hooking tree in Hook(\mathcal{C}) composed of active vertices. Then $\text{size}(T) \geq 2$ and $\text{deg}(T) < \text{size}^2(T)$.*

Proof. Let $r = \text{root}(T)$. r hooks to itself, i.e. $H'(r) = 0$, iff all of its neighbors are active and hooked to it. So T must contain at least two vertices – r and its neighbors.

Let us see now that for $v \in V_T$,

$$(3.1) \quad \delta(v) \leq \delta(r).$$

If v is a neighbor of r in T , then since r hooked to itself, we must have that $v <_d r$

and hence that $\delta(v) \leq \delta(r)$. Otherwise, let u be the neighbor of r on the path in T from v to r . Then, by Lemma 3.7, $v <_d \max_d\{u, r\}$, and so again $\delta(v) \leq \delta(r)$.

We have that

$$\deg(T) = \sum_{v \in V_T} \delta(v) \leq \text{size}(T)\delta(r) < \text{size}^2(T),$$

where in the first inequality we use (3.1), and the second follows because, as explained earlier, T must contain all neighbors of r . $\delta(r)$ is exactly the number of distinct neighbors of r , since there are no multiple edges between active vertices. \square

3.2.3. Contraction. We will define $\text{Contract}(\mathcal{C}, d)$ so that, if \mathcal{C} describes the state of the sequential algorithm, then $\text{Contract}(\mathcal{C}, d)$ is its state after a contraction phase with parameter d . A hooking tree T in \mathcal{C} is d -contractable, if $\deg(T) \leq d$.

The result of $\text{Contract}(\mathcal{C}, d)$ is the configuration $\mathcal{C}' = \langle G', A', I', D', H', R' \rangle$ defined in the following way. First define

$$\begin{aligned} A'' &= \{v : v \notin D \text{ and } \deg(T_v) \leq d \text{ and } \text{root}(T_v) = v\}, \\ I' &= \{v : v \notin D \text{ and } \deg(T_v) > d\}, \\ D'' &= \{v : v \in D \text{ or } \deg(T_v) \leq d \text{ and } \text{root}(T_v) \neq v\}. \end{aligned}$$

Now define

$$H'(v) = \begin{cases} H(v), & v \in I', \\ 0, & \text{otherwise,} \end{cases}$$

and

$$R'(v) = \begin{cases} R(v), & v \in D, \\ \text{root}(T_v), & v \in D'' - D, \\ v, & \text{otherwise.} \end{cases}$$

Let T be a hooking tree in \mathcal{C} and $s = \text{size}(T)$. Let $\langle v_1, \dots, v_s \rangle$ be the sequence of the vertices of T in the order in which they are visited by the exploration walk on T of length $2(s-1)$ starting from the edge $(\text{root}(T), 1)$ of T , where we include a vertex only the first time it is visited by the exploration walk. Let $\langle e_1, \dots, e_k \rangle$ be the sequence of the edges of G incident to the vertices of T defined in the following way – list in order of their labels all edges incident to v_1 , then all edges incident to v_2 , and so on. Obviously $k = \deg(T)$.

Let us define now G' . For $u \in A''$, define $l_u \geq 0$ and the sequence of edges $\langle e_{u,1}, \dots, e_{u,l_u} \rangle$. First, consider the sequence $\langle e_1, \dots, e_k \rangle$ of the edges of G incident to the vertices of T_u from the previous paragraph. Remove from this sequence all edges which are internal to T_u , i.e. such that $\eta(e_i) \in V_{T_u}$. From every subsequence of edges whose other end belongs to the same d -contractable hooking tree leave only the first edge, i.e. for every d -contractable hooking tree $T \neq T_u$ of \mathcal{C} , if the sequence $\langle e_{i_1}, \dots, e_{i_h} \rangle$, $i_1 < \dots < i_h$, contains all the edges from the sequence $\langle e_1, \dots, e_k \rangle$ such that $\eta(e_{i_j}) \in V_T$, then remove e_{i_j} , $j > 1$. Let l_u be the number of remaining edges in the sequence and $\langle e_{u,1}, \dots, e_{u,l_u} \rangle$ be the resulting sequence. Naturally we call the edges $e_{u,j}$ the *remaining edges of T_u* .

Define

$$\begin{aligned} A' &= A'' - \{v \in A'' : l_v = 0\}, \\ D' &= D'' \cup \{v \in A'' : l_v = 0\}, \end{aligned}$$

and

$$\delta'(v) = \begin{cases} l_v, & v \in A', \\ \delta(v), & v \in I', \\ 0, & v \in D'. \end{cases}$$

We are left to define $\mu'(v, i)$. First assume $v \in A'$. Let $(u, j) = \mu(e_{v, i})$. If T_u is not d -contractable, then define $\mu'(v, i) = (u, j)$. If T_u is d -contractable, then define $\mu'(v, i) = (w, k)$, where $w = \text{root}(T_u)$ and k is the only index such that $\eta(e_{w, k}) \in V_{T_v}$. Now assume $v \in I'$. Let $(u, j) = \mu(v, i)$. If T_u is not d -contractable, then define $\mu'(v, i) = (u, j)$. If T_u is d -contractable, let $\mu'(v, i) = (w, k)$, where $w = \text{root}(T_u)$ and k is the only index such that $e_{w, k} = (u, j)$.

LEMMA 3.10. *Let $\mathcal{C}' = \text{Contract}(\mathcal{C}, d)$. Then \mathcal{C}' is a configuration such that $\delta'(v) \leq d$, for $v \in A'$, and $\deg(T_v) > d$, for $v \in I'$. Furthermore, in the hooking forest of \mathcal{C}' every $v \in A' \cup D'$ is in a hooking tree which contains only v .*

Proof. Immediate from the definition of $\text{Contract}(\mathcal{C}, d)$. \square

3.3. Sequential version of the CL algorithm. For every $k \in \mathbb{N}$, we will define a sequence of configurations $\mathcal{C}_l = \langle G_l, A_l, I_l, D_l, H_l, R_l \rangle$, $0 \leq l \leq r(k)$, where $r(k) = 5 \cdot 2^k - 3$.

First, define recursively a sequence of pairs, the first element of which is always either *Hook* or *Contract*, and the second is a natural number, in the following way

$$S_k = \begin{cases} (\langle \text{Hook}, 0 \rangle, \langle \text{Contract}, 1 \rangle), & k = 0, \\ (\langle \text{Hook}, k \rangle, \langle \text{Contract}, k \rangle, S_{k-1}, S_{k-1}, \langle \text{Contract}, k+1 \rangle), & k > 0. \end{cases}$$

The definition of S_k is obtained by “linearizing” the recursive definition of $\text{Connect}(k)$.

By induction, S_k has $r(k)$ elements $P_1, \dots, P_{r(k)}$. Let $P_l = \langle Op_l, Arg_l \rangle$, $1 \leq l \leq r(k)$. Let \mathcal{C}_0 be some configuration. Assume that we have already defined $\mathcal{C}_0, \dots, \mathcal{C}_{l-1}$ for $1 \leq l \leq r(k)$. Define

$$\mathcal{C}_l = \begin{cases} \text{Hook}(\mathcal{C}_{l-1}), & Op_l = \text{Hook}, \\ \text{Contract}(\mathcal{C}_{l-1}, \text{dexp}(Arg_l + 1)), & \text{otherwise.} \end{cases}$$

The sequential version of the CL algorithm for testing whether two vertices u and v are connected in a graph G consists of computing the sequence of configurations $\mathcal{C}_0, \dots, \mathcal{C}_{r(k)}$, where \mathcal{C}_0 is initialized according to G and $k = \lceil \log \log n \rceil$, and checking whether u and v have the same representative in $\mathcal{C}_{r(k)}$: the two vertices are connected iff they have the same representative. The correctness of this algorithm follows from the statements below (see Section 5 for their proofs). We make the following definition first.

DEFINITION 3.11. *A configuration \mathcal{C}_l , $0 \leq l < r(k)$ is nice, if*

- (i) *if $Op_{l+1} = \text{Hook}$, then for every $v \in I_l$, $\text{size}(T_v) > \text{dexp}(Arg_{l+1} + 1)$ and $\deg(T_v) > \text{dexp}(Arg_{l+1} + 2)$, and*
- (ii) *for every $v \in A_l$, $\delta_l(v) \leq \text{dexp}(Arg_{l+1} + 2)$*

By definition, if $Op_{l+1} = \text{Hook}$, then \mathcal{C}_l is nice iff the state described by it fulfills the preconditions given in [6] for executing $\text{Connect}(Arg_{l+1})$. These preconditions ensure the correctness of the algorithm and that it can be executed efficiently in parallel. Considering the correspondence between the sequence $\mathcal{C}_0, \dots, \mathcal{C}_{r(k)}$ and the $\text{Connect}(k)$ procedure of the CL algorithm, the following theorem is a consequence of the results in [6].

THEOREM 3.12. *If \mathcal{C}_0 is a nice configuration, then \mathcal{C}_l is nice, for every $1 \leq l < r(k)$, $|A_{r(k)}| \leq \max\{|A_0|/\text{dexp}(k), 1\}$, and $\text{size}(T_v) > \text{dexp}(k + 1)$, for $v \in I_{r(k)}$.*

Finally, again similarly to [6], we have the following corollary, which says that if we initialize \mathcal{C}_0 according to some undirected graph G , then in \mathcal{C}_r all components of G are contracted.

COROLLARY 3.13. *Let G be a graph with at most one undirected edge between any two vertices and $V = [n]$. Let $k = \lceil \log \log n \rceil$ and $\mathcal{C}_0 = \langle G, A, I, D, H, R \rangle$, where $A = \{v : \delta(v) \neq 0\}$, $I = \emptyset$, $D = V - A$, $H(v) = 0$ and $R(v) = v$, for $v \in V$. Then u and v are connected in G iff $\text{rep}_{R_{r(k)}}(u) = \text{rep}_{R_{r(k)}}(v)$.*

4. Space-efficient algorithm. Since giving directly the Turing Machine which solves the problem and reasoning about its space complexity is rather cumbersome, we define the algorithms outlined in sections 4.1 and 4.2 using pseudo-code and then explain how to translate the pseudo-code to a Turing Machine. The algorithms, as given by the pseudo-code, are almost a literal rephrasing of the definitions given in sections 3.2 and 3.3 into a precise language in which analyzing space complexity is possible. The deviations from a literal rephrasing exist only to decrease the space requirements of the algorithms. Besides that, the correctness of the algorithms follows essentially from Corollary 3.13. In fact, the algorithm in section 4.1 can be thought of as a literal $O(\log^2 n)$ space implementation of the sequential algorithm described in section 3.3.

In section 4.1 we outline an $O(\log^2 n)$ space implementation of the sequential algorithm derived from the definitions in sections 3.2 and 3.3, which instead of storing all of the current configuration, recomputes parts of it when it needs them. In section 4.2 we describe the changes we make to the algorithm from section 4.1 to reduce its space complexity to $O(\log n \log \log n)$. Sections 4.4 discusses in detail the pseudo-code for the algorithm from section 4.2.

4.1. An $O(\log^2 n)$ space algorithm. Let G be a graph with $V = [n]$ and with at most one undirected edge between any two vertices. Let

$$r = 5 \cdot 2^{\lceil \log \log n \rceil} - 3.$$

Consider the sequence of configurations $\mathcal{C}_0, \dots, \mathcal{C}_r$ from Corollary 3.13 for k equal to $\lceil \log \log n \rceil$. The starting point for a space-efficient algorithm comes directly from the definition of this sequence. More precisely, we define functions **Active**(l, v), **Inactive**(l, v), **Done**(l, v), **Degree**(l, v), **Neighbor**(l, v, i), **BackLabel**(l, v, i), **Rep**(l, v), and **Hook**(l, v), where $0 \leq l \leq r$, v is a vertex, and i is the label of an edge incident to v , which return the corresponding component of \mathcal{C}_l . Namely, **Active**(l, v), **Inactive**(l, v), and **Done**(l, v), check, correspondingly, whether v is active, inactive, or done in \mathcal{C}_l , i.e. whether it belongs to A_l , I_l , or D_l ; **Degree**(l, v) returns $\delta_l(v)$, the degree of v in G_l ; **Neighbor**(l, v, i) returns $\eta_l(v, i)$, the i -th neighbor of v in G_l ; **BackLabel**(l, v, i) returns $\beta_l(v, i)$, the back-label of the i -th edge of v ; **Hook**(l, v) returns $H_l(v)$, the label of the hooking edge of v in \mathcal{C}_l ; **Rep**(l, v) returns $R_l(v)$, the immediate representative of v in \mathcal{C}_l .

Call the parameter l of the above functions, the *level of recursion*. Thus the levels of recursion of our algorithm correspond to the elements of the sequence $\mathcal{C}_0, \dots, \mathcal{C}_r$. For $l = 0$, the bottom of the recursion, all of these functions just use the input graph G (see Corollary 3.13). According to the definitions in section 3.3, \mathcal{C}_{l+1} is derived from \mathcal{C}_l , and thus the outputs of the functions mentioned in the previous paragraph for level $l + 1$ is determined recursively from their output for level l . Using these functions, to solve undirected st-connectivity we apply Corollary 3.13.

If $Op_l = Contract$, then \mathcal{C}_l is obtained from \mathcal{C}_{l-1} by contracting some of its hooking trees as defined in section 3.2.3. In this case for a hooking tree T in \mathcal{C}_{l-1} we must determine $\deg(T)$ and be able to enumerate the vertices of T as they are visited by the exploration walk on T starting from $(v, 1)$, for $v \in V_T$. For these purposes we define $\mathbf{TreeSize}(l, v)$ and $\mathbf{TreeWalk}(l, v, i)$. Let T be the hooking tree in \mathcal{C}_{l-1} containing v . If s is the size of T , $\mathbf{TreeSize}(l, v)$ returns $2(s-1)$. Notice that $2(s-1)$ is the length of the exploration walk on T given in Proposition 3.4. $\mathbf{TreeWalk}(l, v, i)$ returns $\Gamma_{T,i}(v, 1)$.

Going over the details of the definitions of each of the functions mentioned above, it is not hard to see that each level of recursion can be implemented in $O(\log n)$ space, and since we have $r = O(\log n)$ levels of recursion, this results in an $O(\log^2 n)$ algorithm for USTCONN. We omit those details, because they are rather immediate and are superceded by the discussion in the next section and the precise definitions in section 4.4.

4.2. The $O(\log n \log \log n)$ space algorithm. The $O(\log n)$ space per level of recursion in the algorithm outlined in the previous section comes mainly from having to store vertices in the local variables of the functions, since each vertex takes $\Theta(\log n)$ space. To see more precisely what is going on consider the following. The definition of $H_l(v)$ contains a comparison of the i -th neighbor of v in \mathcal{C}_{l-1} with its j -th neighbor, i.e. the definition of $\mathbf{Hook}(l, v)$ contains a comparison $\mathbf{Neighbor}(l-1, v, i) = \mathbf{Neighbor}(l-1, v, j)$. Let us say that v is passed to \mathbf{Hook} through a global variable. Obviously, this global variable must be stored locally before the execution of such comparison, because otherwise its value might be overwritten during the two calls to $\mathbf{Neighbor}$. To take care of this bottleneck we define the functions so that they never store a vertex in their local variables.

The first step towards such definitions is to remove the vertex v from the argument list of the functions. Instead of this argument, we maintain one current vertex in a global variable and all functions return information about this vertex. A function which otherwise must return a vertex is defined so that after its execution the current vertex is its result (in this case we say that the function moves the current vertex). It is a responsibility of the calling function to keep enough information locally to restore the original current vertex, if it needs to. Denote the current vertex with cv .

To implement this, first we change the definitions of some of our functions. Instead of $\mathbf{Neighbor}(l, v, i)$, we have $\mathbf{Neighbor}(l, i)$, which moves the current vertex to $\eta_l(cv, i)$, its i -th neighbor in \mathcal{C}_l . Let T be the hooking tree of cv in \mathcal{C}_{l-1} . Instead of $\mathbf{TreeWalk}(l, v, i)$ we have $\mathbf{TreeForward}(l, i)$, which returns j and moves the current vertex to u , where $(u, j) = \Gamma_{T,i}(cv, 1)$. Similarly we have $\mathbf{TreeBack}(l, i, j)$ which moves the current vertex to the vertex of $\Gamma_{T,i}'(cv, j)$.

The most important part of our idea to avoid storing vertices locally is to be able to move the current vertex temporarily, perform something at the new current vertex, and then return to the original current vertex. For this define $\mathbf{Move}(l, i)$ to return $\beta_{l-1}(cv, i)$, i.e. $\mathbf{BackLabel}(l-1, i)$, and move the current vertex to $\eta_{l-1}(cv, i)$, i.e. a call to $\mathbf{Neighbor}(l-1, i)$. Call $\mathbf{Move}(l, i)$ and $\mathbf{TreeForward}(l, i)$ *forward moves*. For a forward move M , let $\mathbf{Reverse}(M, j)$ be its reverse, i.e. it is correspondingly $\mathbf{Move}(l, j)$ or $\mathbf{TreeBack}(l, i, j)$, where j is the result of M . We use the reversibility of exploration walks here, so that $\mathbf{TreeBack}$ reverts $\mathbf{TreeForward}$. Finally, we have a "dummy" forward move $\mathbf{Current}$ which does not change the current vertex and whose purpose is to address the current vertex. The reverse of $\mathbf{Current}$ is $\mathbf{Current}$ again.

We use forward moves to change the current vertex and their reverses to restore

it. Call a sequence of forward moves *path description relative to the current vertex*. If P is a path description relative to the current vertex and B is some instruction(s), then define **after** P **do** B to change the current vertex according to P , perform B , and then use the reverses of the moves in P to restore the current vertex.

A simple example of the use of **after** is the comparison operator $=$, which compares two vertices given their path descriptions relative to the current vertex and returns true iff they are the same. Using **after** we can move to the first vertex and store it in a local variable, then go to the second vertex and compare the two. This takes $\Theta(\log n)$ space. Instead of this, going back and forth between the two vertices, using the reversibility of the moves along the edges and the exploration walks on the trees, we perform the comparison bit by bit. Aside from the information stored for the ways back, this takes only the $\Theta(\log \log n)$ space necessary to store the index of a bit. This way the bottleneck of $\Omega(\log n)$ space is reduced to $\Omega(\log \log n)$.

For example, the $=$ operator is used in the definition of $\text{TreeSize}(l)$, which determines the size of the hooking tree T of cv in \mathcal{C}_{l-1} , in the following way. Starting at cv and using Proposition 3.4, we can make steps from the exploration walk on T until we go back to cv sufficiently many times. For this we can store cv , so that we can compare it with each new vertex of the walk. This takes $\Theta(\log n)$ space, independent of the size of T . Instead, we incrementally find i with properties as in Proposition 3.4, where to check if the i -th vertex of the walk is equal to cv , we keep the current vertex at cv and use the $=$ operator, as defined above, to compare it to the vertex with path description $\text{TreeForward}(l, i)$. More precisely, we use the comparison $\text{Current}=\text{TreeForward}(l, i)$ to check whether the i -th vertex of the walk is equal to cv , where the semantics of the comparison in this case is explained in the previous paragraph. Thus, if $\text{size}(T) \leq s$, then we can find its size in space $O(\log s + \log \log n)$ (the $\log \log n$ appears because of counters used during comparison of vertices). Alternatively, if $\text{size}(T) > s$, we can still use only $O(\log s + \log \log n)$ space to learn this without actually computing $\text{size}(T)$, because it is enough to stop the exploration walk on T as soon as we learn that $\text{size}(T) > s$.

The second part of our idea to reduce the space of the algorithm is to have an upper bound $v(l)$ on the values which variables can take at level l , i.e. during the execution of all functions at level l the values of their local variables are at most $v(l)$. We set

$$v(l) = 2 \cdot \text{dexp}(\text{Arg}_l + 2).$$

Call a number x *valid* for level l , if it is at most $v(l)$. A vertex v is *valid* for level l , if its degree $\delta_{l-1}(v)$ is valid for level l .

Using the concept of a current vertex, we can eliminate the need to store a vertex in a local variable and thus our local variables contain essentially only degrees of vertices, labels and back-labels of edges, and lengths of exploration walks on hooking trees. For example, the information stored for reversing a forward move is a back-label (for Move) or a tree-edge label (for TreeForward). We still have to make sure that every time we store a value in a local variable it is valid. For this the following observation is helpful.

OBSERVATION 4.1.

- (i) *The labels of the edges incident to vertex v valid for level l are valid for level l . This is not necessarily true for their back-labels.*
- (ii) *All vertices which are active in \mathcal{C}_{l-1} are valid for level l .*
- (iii) *If $\text{Op}_l = \text{Contract}$, then if a hooking tree T in \mathcal{C}_{l-1} has degree at most $\text{dexp}(\text{Arg}_l + 1)$, then all of its vertices are valid for level l .*

The first item of the observation is trivial, the second follows from Theorem 3.12, because all \mathcal{C}_l are nice, and the third follows because $\text{dexp}(Arg_l + 1) < v(l)$.

Our goal has become to prove the following lemma. The functions mentioned in the lemma are the ones outlined previously in this section. Their precise definitions are given in section 4.4. Let T be a hooking tree in \mathcal{C}_{l-1} . T is *contractable for level l* , if $\text{deg}(T) \leq \text{dexp}(Arg_l + 1)$. Let (v, i) be an edge of T and $u = \eta_T(v, i)$. A move along (v, i) is *possible for level l* , if u is valid for level l .

LEMMA 4.2. *Let T be the hooking tree in \mathcal{C}_{l-1} of the current vertex cv .*

- (i) **Active**(l), **Inactive**(l), **Done**(l), **Hook**(l), and **Degree**(l), correctly return the value of the corresponding component of \mathcal{C}_l for cv .
- (ii) If T is uncontractable for level l or $cv \in D_{l-1}$, then **Root**(l) returns 0, otherwise it returns the index of the first occurrence of $\text{root}(T)$ in the exploration walk on T starting from $(cv, 1)$.
TreeSize(l) returns $2(\text{size}(T) - 1)$, if T is contractable for level l , and **null** otherwise.
Assume that cv is valid for level l . If all moves of $\Gamma_{T, \leq i}(cv, 1)$ are possible for level l and it ends in (v, j) , then **TreeForward**(l, i) moves the current vertex to v and returns j . If all moves of $\Gamma'_{T, \leq i}(cv, j)$ are possible for level l and it ends in v , then **TreeBack**(l, i, j) moves the current vertex to v .
- (iii) Let $(v, j) = \mu_l(cv, i)$.
Neighbor(l, i) moves the current vertex to v .
BackLabel(l, i) returns j , if j is valid for the level at which **BackLabel**(l, i) was called (see the discussion on returning values in section 4.4.2), and **null** otherwise.

All local variables are valid.

The proof of the lemma is done by induction on the level of recursion. For this we need the correctness of the functions which a given function calls. Sometimes we have to use correctness for the same level of recursion, but this does not result in a circular reasoning because for any two functions **F** and **G**, there are no chains of function calls within the same level of recursion both from **F** to **G** and from **G** to **F**.

The correctness essentially follows from the correctness of the CL algorithm (Corollary 3.13). It can be easily seen that the introduction of one global current vertex and always returning information about this vertex, maintains the faithfulness of our implementation to the CL algorithm – the current vertex is an implicit argument to all functions describing a configuration and calling it “current” just facilitates our intuition about how the algorithm proceeds.

The only real deviation from the definitions given in section 3.2 is that we have an upper bound on the numerical values which can be stored at a level, and so we might be unable to process the result of a function, if it is invalid for the current level of recursion. Actually, as can be seen from Lemma 4.2, some functions are specified to return **null**, if their result is invalid for the level requesting it. By Observation 4.1, the only information we can derive from an invalid or **null** result is that either the current vertex is invalid (if **Degree**(l) is invalid), a neighbor of the current vertex is invalid (if **BackLabel**(l, i) is **null**), or that the current vertex is part of an uncontractable tree (if **TreeSize** is **null**). This information is enough to define the functions as in Lemma 4.2. First, we never move to a vertex from which we cannot return, i.e. along an edge with an invalid back-label, so we never have to store an invalid back-label locally. Second, during contraction vertices which are either invalid or part of an uncontractable tree will become inactive and thus, by definition, will inherit

their properties, e.g. degree and hooking edge, from the previous level. In a hooking operation (section 3.2.2), we do not need to lookup the degree of an invalid (and even inactive) vertex. In a contraction operation (section 3.3.3), we can stop the exploration walk on a tree as soon as the walk runs into an invalid vertex, because then the tree is clearly uncontractable.

To ensure that all local variables are valid for the current level of recursion we use Observation 4.1 in the following way. First, notice that since the value returned from a function resides in a global variable we can check whether it is valid by simply inspecting this global variable. Furthermore, some functions (e.g. `TreeSize` and `BackLabel`) return `null`, if their result is invalid for the current level of recursion. In any case, we can easily learn when the return value of a function is invalid without having to store it locally. According to Observation 4.1(i), if the result of `BackLabel` is `null`, then the corresponding neighbor is invalid and we never move the current vertex along such edges. Also, Observation 4.1(ii) ensures that we can always process locally active vertices. Finally, according to Observation 4.1(iii), if the result of `TreeSize` is `null`, then the hooking tree of the current vertex is uncontractable.

4.3. Solving undirected st-connectivity. Using Lemma 4.2 we can prove the main theorem of this paper.

THEOREM 4.3. *USTCONN on a graph with n vertices can be solved in space $O(\log n \log \log n)$.*

Let $m = \lceil \log \log n \rceil$. Recall that $r = 5 \cdot 2^m - 3 = O(\log n)$. By Corollary 3.13, s and t are connected iff $\text{rep}_{R_r}(s) = \text{rep}_{R_r}(t)$. Thus to solve USTCONN it is enough to define a function `MoveToRep`(l) which moves the current vertex cv to its representative $\text{rep}_{R_l}(cv)$ in \mathcal{C}_l . Then, to check whether s and t are connected, we call `MoveToRep`(r) twice, once from s and once from t , and compare the two resulting final current vertices (see the definition of `Connected` in section 4.4.10). `MoveToRep` is defined by first calling itself recursively. This moves the current vertex cv to its representative $v = \text{rep}_{R_{l-1}}(cv)$ in \mathcal{C}_{l-1} . Then, if $Op_l = \text{Contract}$ and the hooking tree T in \mathcal{C}_l of the new current vertex v is contractable for level l , `MoveToRep` moves v to the root of T . We use here that the root of T will be the immediate representative of v after contracting T . See section 4.4.10 for a precise definition of `MoveToRep`.

The space complexity of the algorithm is dominated by the space taken by the execution stack. From Lemma 4.2 follows that each local variable at level l is at most $v(l)$. Since there are constant number of local variables per function and the length of every chain of function calls within the same level of recursion is bounded by a constant, the space taken by level l is $O(\log v(l) + m)$ (the additional $O(m)$ space appears because of the counters used during comparison of vertices). Since $\log v(l) = 4 \cdot 2^{Ar_{gl}} + 1$, $1 \leq l \leq r$, and $r \cdot m = O(\log n \log \log n)$, we have to prove that

$$\sum_{l=1}^r 2^{Ar_{gl}} = O(\log n \log \log n).$$

Consider the recurrence given by

$$S(k) = \begin{cases} 3, & k = 0 \\ 2S(k-1) + 2 \cdot 2^k + 2^{k+1}, & k > 0. \end{cases}$$

From the recursive definition of S_m given in section 3.3 follows that the left hand side of the equation which we want to prove is exactly $S(m)$. Finally it is not hard to prove by induction on k that $S(k) = 2^k(4k + 3)$. Hence $S(m) = O(\log n \log \log n)$.

4.4. Pseudo-code. The language of the pseudo-code is self-explanatory. The difference between a function and a procedure, marked with **function** and **procedure** correspondingly, is that the first returns a value and the second does not. In the following we will refer to both as functions. Operators, marked with **operator**, are functions for which a more traditional in-fix position of their usage is emphasized. The semantics of statements, marked with **statement**, is closer to what is traditionally known as macroses, than to procedures and functions, namely they do not allocate a new stack frame. Blocks are marked by indentation, **break** exits the closest surrounding loop, **continue** continues with the next cycle of the execution of the closest surrounding loop, and **return** exits the current function, returning a value, if necessary. We use fixed width font to denote the names of functions and variables from the pseudo-code, e.g. `F` and `i`, and a roman italics font for mathematical functions and variables, e.g. *f* and *i*.

4.4.1. Execution of the pseudo-code. The variable usage and execution of the pseudo-code is standard. During its execution a function can use only its own local variables and the global variables. For the execution of the functions we use a stack which contains the local variables of the functions executed at the moment. The part of the stack devoted to the execution of a function is called the *stack frame of the function*. Statements use the stack frame of their surrounding function. The top of this stack contains the stack frame of the function being executed at the moment. When the current function calls another function, first a new stack frame is allocated on the top of the stack and then the new function is executed. Part of the stack frame of a function contains information about the address (the place in the program) from which the function was called. Once the current function is finished its stack frame is removed from the top of the stack and the execution resumes from the address from which the current function was called.

In addition to functions which are executed using the stack we have *global functions*, which do not use the stack for their execution. Such functions use only global variables for their execution – i.e. their “local” variables are in fact global variables which are visible only from the particular global function. Since in what follows we are only concerned with the contents of the stack, we will concentrate on functions which use the stack.

The execution of the pseudo-code proceeds in the following way. Every function is executed at some level of recursion. Every function has an argument l , which contains the current level of recursion. During its execution, a function can call other functions either on the same level of recursion, i.e. calls like $F(l, \dots)$, or recursive calls to the previous level of recursion, i.e. calls like $F(l - 1, \dots)$. There is a constant c , such that the length of a chain of function calls within the same level of recursion is at most c . The stack frames of functions executed in the same level of recursion are consecutive on the stack. We call such a sequence of stack frames, the *stack frame for the level*. From the fact that any stack frame for a level contains at most c stack frames for functions and that each function uses a constant number of variables follows that there is a constant d such that the stack frame of every level contains a total of at most d variables.

The local and global variables contain only numerical and boolean values, and a special value `null`, different from any other value. For every level of recursion l we have an upper bound $v(l)$, computed by a global function, on the numerical values which are valid for this level, i.e. all numerical values at level l are at most $v(l)$. Boolean values and `null` are always valid. The counters used for comparison of

vertices are at most $\lceil \log n \rceil$.

4.4.2. Passing arguments and returning values. We have two methods of passing arguments to a function and returning a value. The first is through global variables and the second is through global arrays which contain an entry for every level of recursion.

The method which uses global variables is straightforward. We have global variables which are set to the arguments of the function before it is called. We denote the global variables for the arguments of a function F with $\mathbf{arg1}F$, $\mathbf{arg2}F$, and so on. During its execution a function can lookup its arguments from these global variables. It might decide to store them as local variables, but this might not always be possible, because the arguments might be invalid for the current level of recursion.

To return a value a function sets a global variable. After a return from a call of a function, the calling function decides whether it wants to store the returned value locally. We have a special assignment operator, $:=$, which assigns the return value r of a function returning through a global variable to a local variable in the following way: if r is valid for the current level of recursion, the result of the assignment is r , otherwise it is \mathbf{null} .

The method which uses arrays is more subtle. Let F be a function which uses this method of passing arguments and returning values. This method can be used only, if all calls to F at the current level of recursion l , are recursive, i.e. all calls to F look like $F(l-1, \dots)$. We have two global arrays – one, $\mathbf{arg}F$, for passing arguments to F and one, $\mathbf{ret}F$, for returning a value from it. Those arrays contain exactly one entry for every possible level of recursion and each entry could be marked. Also each entry holds values which are valid for the corresponding level of recursion, so the space taken by each such array is the same as the space taken by the execution stack.

Let H call F . If H uses the value returned by F , then before the call to F the entry of $\mathbf{ret}F$ for the current level of recursion is marked, otherwise it is left unmarked. When F produces a result at level l , it finds the smallest index $i > l$ such that the i -th entry of $\mathbf{ret}F$ is marked and tries to store its result there. If the value produced is too large for the corresponding entry, F writes \mathbf{null} . After the call to F returns, H unmarks the entry of $\mathbf{ret}F$ for the current level of recursion. The only time when H does not use the value returned by F is when H is actually F and the call to F is a recursive call whose result is passed back without modification, e.g. a call like $\mathbf{return} F(l-1, \dots)$ in the definition of F .

Similarly, if H provides arguments to F , then before the call to F , H marks the entry of $\mathbf{arg}F$ for the current level of recursion and provides values for it. When F wants to access its arguments at level l , it finds the smallest index $i > l$ such that the i -th entry of $\mathbf{arg}F$ is marked and uses the values stored in the entry as its arguments. After the call to F returns, H unmarks the entry of $\mathbf{arg}F$ for the current level of recursion. Again we have that the only time when H does not provide arguments to F is when H is F and the call to F is a recursive call whose arguments are the same as for the current call to F , e.g. a call like $F(l-1, y)$ in the definition of $F(l, x)$ at a place where it is always true that $x = y$.

Since this method is used only if all calls to F are recursive, there is no danger of overwriting F 's arguments or returned value. For passing arguments this method helps when we need to store an argument only at the level which generates it, where it is valid, but still allow for lower levels of recursion to access it, where it might be invalid. For returning values this method helps when we want to return a value exactly at the level which requested it originally. This method allows for nested calls

from different levels of recursion to the same function, which the ordinary method of passing variables through global variables does not always allow.

The restriction we have on the space of a level is the reason why we chose those methods of passing arguments and returning value. The method using arrays is more unusual, and it is used only in two functions, `BackLabel` and `BackLabelAux`. The reason why we introduced it is explained in the notes for those functions.

In the code, we use $F(l, x_1, \dots, x_k)$ to call a function F at level l with arguments x_1, \dots, x_k . If a function F takes arguments, but is called without ones, it uses the values currently located in the global variables (or the arrays) for F .

4.4.3. Translation of the pseudo-code to a Turing Machine. Let us address now the issue of translating the pseudo-code to a Turing Machine with a binary alphabet. Most of the details, like doing arithmetic and performing conditionals are rather straightforward, so we skip them and concentrate only on variable usage. Numerical values are represented in binary. To represent the `null` value, we use one additional bit to designate whether the value is `null` or not. We have a separate tape for each global variable (there are only constant number of them). The space taken by each global variable is $O(\log n)$.

There is a tape assigned for the stack and the head of this tape is positioned at the stack frame of the current function, i.e. at the top of the stack. The stack frame of a function contains the state to which the TM machine must return after the execution of the function (this takes a constant number of bits depending only on the TM) and the values of the local variables of the function.

The space taken by each local variable depends on the level of recursion at which the stack frame occurs. Since at a level l the value of every valid local variable is at most $v(l)$, the space $s(l)$ taken by such variable at level l is $O(\log v(l))$. This space is known to the current function, because it can be computed by a global function from the current level of recursion. So to use the i -th local variable the current function must move the head of the stack tape to the place where the variable is located. This place can be computed by the current function from i and $s(l)$. As discussed earlier there is a constant d such that the stack frame of the l -th level of recursion contains at most d variables. Thus the space taken by the stack frame of level l is $O(\log v(l) + \log \log n)$. The $O(\log \log n)$ appears because of comparison of vertices, as explained in section 4.2.

After the execution of the current function the state of the TM is restored to the value stored on the stack and the head of the stack tape is moved to the stack frame of the caller.

4.4.4. Preliminaries. To simplify the exposition of the algorithm, we remove the level of recursion from the argument lists of the functions. Instead we have one global variable, `level`, which contains the current level of recursion. `level` is set to $5 \cdot 2^{\lceil \log \log n \rceil} - 3$ initially. Let F be a function which calls a function G on the previous level of recursion. This task is performed by `Prev`, namely `Prev(G(...))` passes arguments to G , decreases the current level of recursion, calls G , and upon return from G increases the current level of recursion. This is the only way that the current level of recursion is changed – all functions can lookup the value of `level`, but none of them changes it. We denote the current level of recursion with cl .

The global variable `currvertex` contains the current vertex cv .

In the following, T denotes the hooking tree of the current vertex in \mathcal{C}_{cl-1} . Unless specified otherwise, the exploration walk refers to the exploration walk on T starting

from $(cv, 1)$. A hooking tree in \mathcal{C}_{cl-1} is called contractable, if its degree is at most $\text{dexp}(Arg_{cl} + 1)$. A value is valid, if it is most $v(cl)$.

We use the following observation. It follows from Observation 4.1 and the correctness of the functions mentioned in it. **MoveValid**(i) checks whether the back-label $\beta_{cl-1}(cv, i)$ of the i -th edge of cv in \mathcal{C}_{cl-1} is valid, i.e. whether $\beta_{cl-1}(cv, i) \leq v(cl)$.

OBSERVATION 4.4.

- (i) If **TreeSize** \neq null, then all moves of **TreeForward**(i) are possible for level cl .
- (ii) If **MoveValid**(i) is true, then the result of **Move**(i) is not null and valid for level cl , otherwise $\beta_{cl-1}(cv, i)$ is invalid for level cl .

By this observation **TreeSize** and **MoveValid** serve as “safeguard” checks for forward moves. Thus, before making a forward move to change the current vertex, if we want to be able to return, e.g. in an **after** statement, we always first make sure that the forward move returns a valid result. In this case we say that the forward move is valid.

All functions, except **BackLabel** and **BackLabelAux**, take arguments and return values through global variables.

Every function is preceded by paragraphs which give its specification. Also notes are made on the definition and correctness of the function, and on the validity of its local variables. In the notes we use interchangeably the name of a local variable, given in fixed font, and its value at a particular point of the execution of the function. To facilitate the reading of the pseudo-code we have annotated the meaning of the local variables of the non-global functions.

4.4.5. Important functions. Global function ArgOp. Input l . Output $2 \cdot Arg_l + \varepsilon_l$, where ε_l is 0, if $Op_l = Hook$, and 1, otherwise. Assumes $1 \leq l \leq 5 \cdot 2^{\lceil \log \log n \rceil} - 3$.

global function ArgOp

```

1 := argArgOp;
k := ⌈log log n⌉;
while true
  if k = 0 then return 2 · (1 - 1) + (1 - 1);
  else
    if 1 ≤ 2 then return 2 · k + (1 - 1);
    else
      if 1 = 5 · 2k - 3 then return 2 · (k + 1) + 1;
      else
        if 1 ≥ 5 · 2k-1 then 1 := 1 - 5 · 2k-1 + 1;
        else 1 := 1 - 2;
        k := k - 1;

```

Global functions Valid, ContractDegree, and Contraction. Input None. Output Correspondingly $2 \cdot \text{dexp}(Arg_{cl} + 2)$, $\text{dexp}(Arg_{cl} + 1)$, and whether $Op_{cl} = Contract$.

global function Valid

```

return 2 · dexp(2 + ArgOp(level) div 2);

```

global function ContractDegree

```

return dexp(1 + ArgOp(level) div 2);

```

global function Contraction

return ArgOp(level) mod 2 = 1;
Statement “after M_1, M_2, \dots, M_k do B”. **Input** M_1, \dots, M_k – path description relative to the current vertex, B some instruction(s). **Output** Moves the current vertex according to the forward moves in M_1, \dots, M_k , executes B, and finally restores the original current vertex. **Assumes** All forward moves are valid. **Local variables** l_1, \dots, l_k (results of the forward moves).

statement after M_1, M_2, \dots, M_k **do** B

$l_1 := M_1; l_2 := M_2; \dots; l_k := M_k;$
 B;

Reverse(M_k, l_k); ...; Reverse(M_2, l_2); Reverse(M_1, l_1);

Operators $<$, $=$, and $<_d$. **Input** P_1 and P_2 – path descriptions relative to the current vertex. **Output** Let v_1 and v_2 be the end vertex of P_1 and P_2 , correspondingly. The three operators check correspondingly, whether $v_1 < v_2$, $v_1 = v_2$, and $v_1 <_d v_2$. **Assumes** All forward moves are valid. $<_d$ assumes also that v_1 and v_2 are valid. **Local variables** i (counter for indices of bits), b_1 (i -th bit of v_1), and b_2 (i -th bit of v_2) for $<$ and $=$. d_1 (degree of v_1) and d_2 (degree of v_2) for $<_d$. **Notes** Bit(s, t) returns the s -th most significant bit of t . b_1 and b_2 are single bits.

operator $P_1 < P_2$

for $i := 1$ **to** $\lceil \log n \rceil$ **do**
after P_1 **do** $b_1 := \text{Bit}(i, \text{currvertex});$
after P_2 **do** $b_2 := \text{Bit}(i, \text{currvertex});$
if $b_1 \neq b_2$ **then return** $b_1 < b_2;$
return false;

operator $P_1 = P_2$

for $i := 1$ **to** $\lceil \log n \rceil$ **do**
after P_1 **do** $b_1 := \text{Bit}(i, \text{currvertex});$
after P_2 **do** $b_2 := \text{Bit}(i, \text{currvertex});$
if $b_1 \neq b_2$ **then return false;**
return true;

operator $P_1 <_d P_2$

after P_1 **do** $d_1 ::= \text{Prev}(\text{Degree});$
after P_2 **do** $d_2 ::= \text{Prev}(\text{Degree});$
return $(d_1 < d_2)$ **or** $(d_1 = d_2$ **and** $P_1 < P_2);$

4.4.6. State functions. Functions Done, Active, and Inactive. Input None. **Output** Correspondingly whether $cv \in D_{cl}$, $cv \in A_{cl}$, and $cv \in I_{cl}$. **Assumes** None. **Local variables** None.

function Done

return $(\text{level} = 0$ **and** $\text{Degree} = 0)$ **or**
 $(\text{level} > 0$ **and not** Contraction **and** $\text{Prev}(\text{Done}))$ **or**
 $(\text{level} > 0$ **and** Contraction **and**
 $(\text{Prev}(\text{Done})$ **or** $\text{Root} \neq 0$ **or** $\text{Degree} = 0));$

function Active

return $(\text{level} = 0$ **and** $\text{Degree} \neq 0)$ **or**
 $(\text{level} > 0$ **and not** Contraction **and** $\text{Prev}(\text{Active}))$ **or**
 $(\text{level} > 0$ **and** Contraction **and**
not Done **and** $\text{TreeSize} \neq \text{null});$

function Inactive

return $(\text{level} > 0$ **and not** Contraction **and** $\text{Prev}(\text{Inactive}))$ **or**
 $(\text{level} > 0$ **and** Contraction **and**

```
not Done and TreeSize = null);
```

4.4.7. Hooking. Function Hook. **Input** None. **Output** $H_{cl}(cv)$. **Assumes** None. **Local variables** d_1 (degree of cv), i (counter for neighbors of cv), d_2 (degree of the i -th neighbor of cv), j (counter for neighbors of the i -th neighbor of cv), and m (current candidate neighbor for hooking). **Notes** Hook is defined as given in section 3.2.2. In line 9 we use Observation 4.1(i) to deduce that $\eta_{cl-1}(cv, i) \in I_{cl-1}$. At line 5 $cv \in A_{cl-1}$ and hence d_1 is valid and non-null. So i and m are also valid. At line 12 we have that $\eta_{cl-1}(cv, i), \eta_{cl-1}(cv, m) \in A_{cl-1}$. At line 14 all neighbors of cv are in A_{cl-1} . Hence d_2 and j are valid.

```
function Hook
1   if level = 0 then return 0;
2   if not Contraction and Prev(Inactive) then return Prev(Hook);
3   if Contraction and Inactive then return Prev(Hook);
4   if Prev(Done) or Contraction then return 0;

5   d1 := Prev(Degree);
6   m := 0;
7   for i := 1 to d1 do
8     // if the i-th neighbor is inactive then hook to it
9     if not MoveValid(i) then return i;
10    after Move(i) do fl := Prev(Inactive);
11    if fl then return i;

    // otherwise check if it is bigger than the current biggest
    // active neighbor
12    if Move(m) <_d Move(i) then m := i;
13    if m > 0 then return m;

14  for i := 1 to d1 do
15    after Move(i) do d2 := Prev(Degree);
16    for j := 1 to d2 do
17      // if the j-th neighbor of the i-th neighbor
18      // is inactive, then hook to the i-th neighbor
19      after Move(i) do fl := MoveValid(j);
20      if not fl then return i;
21      after Move(i), Move(j) do fl := Prev(Inactive);
22      if fl then return i;

      // otherwise hook to the i-th neighbor,
      // if its j-th neighbor is bigger than currvertex
23      if Current <_d (Move(i), Move(j)) then m := i;
24  return m;
```

Function IsHooked. **Input** i . **Output** Let $(v, j) = \mu_{cl-1}(cv, i)$ and $h = H_{cl-1}(v)$. IsHooked is true iff $h = j$. **Assumes** $cl \geq 1$, $0 \leq i \leq \delta_{cl-1}(cv)$, and cv is valid or $\delta_{cl-1}(cv) \leq \delta_{cl-1}(v)$. **Local variables** i , j , and h (contain their equivalents from the definition of the output). **Notes** i is valid because of line 2, j is valid because of the assignment in line 5, and h is valid because of the assignment in line 9.

```
function IsHooked
1   if argIsHooked = 0 then return Prev(Hook) = 0;
```

```

2   if argIsHooked > Valid then return Prev(IsHooked);
3   i := argIsHooked;
4   if Prev(Degree) > Valid then return Prev(IsHooked(i));

5   j := Prev(BackLabel(i));
6   if j = null then
7       if Prev(Active) then return false;
8       return Prev(IsHooked(i));
9   after Move(i) do h ::= Prev(Hook);
10  return h = j;

```

Correctness We will prove the correctness of `IsHooked` by induction on cl . Notice that for $cl = 1$, the checks in lines 2, 4, and 6 all fail because at level 1 all vertices are valid, and we compare h and j in line 10.

First consider the case when $cv \in A_{cl-1}$. In this case cv is valid by Observation 4.1(ii). If j is invalid, then $v \in I_{cl-1}$, and it is not hooked to cv (otherwise $cv \in I_{cl-1}$ by Definition 3.5). We catch this in line 7. If j is valid, then in line 10 we check whether it is equal to h .

Let now $cv \in I_{cl-1}$ and $v \in A_{cl-1}$. Since v is valid, cv is valid also, because this follows from v valid, and cv valid or $\delta_{cl-1}(cv) \leq \delta_{cl-1}(v)$. So i , j and h are valid and we can compare h and j in line 10.

Assume now that $cv, v \in I_{cl-1}$. In this case the only way `IsHooked` returns an answer without calling recursively is in line 10, then j is valid and we have compared it to h . Notice now that, if `IsHooked` calls itself recursively then $\delta_{cl-1}(cv) \leq \delta_{cl-1}(v)$. This is true for the calls in lines 2 and 4, because then cv is invalid. For the call in line 8 this is true, because cv is valid and v is invalid. Since $cv, v \in I_{cl-1}$, we have that $cl \geq 2$, $\delta_{cl-2}(cv) = \delta_{cl-1}(cv)$, $\delta_{cl-2}(v) = \delta_{cl-1}(v)$, $(v, j) = \mu_{cl-2}(cv, i)$, and $h = H_{cl-2}(v)$. Thus the correctness in this case is ensured by the inductive hypothesis.

4.4.8. Exploration walk. The functions in this section come from the definition of a hooking forest of a configuration given in section 3.2.1. Throughout it is assumed that $cl \geq 1$ and $Op_{cl} = Contract$.

Function TreeDegree. **Input** None. **Output** If cv is valid, `TreeDegree` returns $\delta_T(cv)$, otherwise it returns `null`. **Assumes** None. **Local variables** i (counter for steps of the exploration walk), d (degree of cv), and td (output). **Notes** In line 5 we use that cv is valid to apply the correctness of `IsHooked`. d is valid because of the assignment in line 1, and i and td are valid because at line 3 cv is valid.

```

function TreeDegree
    // if currvertex is invalid return null
1   d ::= Prev(Degree);
2   if d = null then return null;

3   td := 0;
    // count the number of neighbors which are hooked to currvertex
4   for i := 1 to d do
5       if IsHooked(i) then td := td + 1;
    // add 1 if currvertex did not hook to itself
6   if Prev(Hook)  $\neq$  0 then td := td + 1;
7   return td

```

Function TreeMove. **Input** i . **Output** Let $(v, r) = \mu_T(cv, i)$. If a move along (cv, i) is possible, i.e. v is valid, `TreeMove` returns r and moves the current vertex

to v , otherwise it does not change the current vertex and returns `null`. **Assumes** $0 \leq i \leq \delta_T(cv)$, cv is valid. **Local variables** i (input), j (graph-edge label of the i -th T -edge of cv), k (counter for the neighbors of v), l (counter for vertices which hooked to cv), d (degree of cv), d_1 (degree of v), and r (back-label of the i -th T -edge of cv).

Notes Lines 2-7 convert from the label i of a T -edge $e = (v, i)$ to a label j of an edge in the graph. In line 5 we use that cv is valid to apply `IsHooked`. Lines 8-10 handle the case when v is invalid. Lines 11-26 compute the tree back-label r of e and move the current vertex to v . Lines 11-13 handle the case, when v hooked to the current vertex. Lines 14-26 handle the case when e is the hooking edge of the current vertex. In lines 19-21 we use that, if the k -th neighbor of v is invalid, then it is not cv , because cv is valid.

i , j , l , and d are valid, because cv is valid (the assignments in lines 3 and 7 are non-`null`). d_1 is valid because of the assignment in line 9. k and r are valid because at line 14 v is valid.

```

function TreeMove
     $i := \text{argTreeMove};$ 
1    if  $i = 0$  then return 0;

    // convert from T-edge label to graph-edge label
2     $l := i;$ 
3     $d := \text{Prev}(\text{Degree});$ 
4    for  $j := 1$  to  $d$  do
5        if IsHooked( $j$ ) then  $l := l - 1;$ 
6        if  $l = 0$  then break;
7    if  $j > d$  then  $j := \text{Prev}(\text{Hook});$ 

    // if the new vertex is invalid return null
8    if not MoveValid( $j$ ) then return null;
9    after Move( $j$ ) do  $d_1 := \text{Prev}(\text{Degree});$ 
10   if  $d_1 = \text{null}$  then return null;

11   if  $i < \text{TreeDegree}$  or ( $i = \text{TreeDegree}$  and Prev(Hook) = 0) then
        // e goes to a neighbor which hooked to currvertex
12       Move( $j$ );
13       return TreeDegree;

    // e is the hooking edge of currvertex

    // compute the tree back-label
14    $r := 1;$ 
15   for  $k := 1$  to  $d_1$  do
16       after Move( $j$ ) do  $f_1 := \text{IsHooked}$ ( $k$ );
17       if not  $f_1$  then continue;
        // enumerate all the edges with which neighbors
        // of the new vertex hooked to it

18       after Move( $j$ ) do  $f_1 := \text{MoveValid}$ ( $k$ );
19       if not  $f_1$  then

```

```

20           // the k-th neighbor of the new vertex is invalid
21           r := r + 1;
22           continue;

           // check if this is the edge with which currvertex
           // hooked to the new vertex
23           if (Move(j), Move(k)) = Current then break;
24           r := r + 1;

           // move to the new vertex
25           Move(j);
           // return the tree back-label
26           return r;

```

Function TreeForwardStep. **Input** i . **Output** Let $(v, j) = \Gamma_{T,1}(cv, i)$. If a move along (cv, i) is possible, i.e. v is valid, then **TreeForwardStep** returns j and moves the current vertex to v , otherwise it returns null and does not change the current vertex. **Assumes** $0 \leq i \leq \delta_T(cv)$, cv is valid. **Local variables** j (back-label of the i -th T -edge of cv).

```

function TreeForwardStep
    j := TreeMove(argTreeForwardStep);
    if j = null then return null;
    j := j + 1;
    if j > TreeDegree then j := 1;
    return j;

```

Function TreeForward. **Input** i . **Output** If $(v, j) = \Gamma_{T,i}(cv, 1)$, then the function **TreeForward** returns j and moves the current vertex to v . **Assumes** cv and i are valid, all moves of $\Gamma_{T,\leq i}(v, 1)$ are possible. **Local variables** i (input), j (output), and k (counter for steps of the exploration walk).

```

function TreeForward
    i := argTreeForward;
    j := 1;
    for k := 1 to i do
        j := TreeForwardStep(j);
    return j;

```

Function TreeBack. **Input** i, j . **Output** If v is the vertex of $\Gamma'_{T,i}(cv, j)$, then **TreeBack** moves the current vertex to v . **Assumes** $0 \leq j \leq \delta_T(cv)$, cv and i are valid, all moves of $\Gamma'_{T,\leq i}(v, j)$ are possible. **Local variables** i (input), j (input), k (counter for steps of the exploration walk). **Notes** **TreeBack** is defined in a way similar to **TreeForward** using a function **TreeBackStep**.

```

function TreeBackStep
    j := argTreeBackStep - 1;
    if j = 0 then j := TreeDegree;
    return TreeMove(j);

```

```

procedure TreeBack
    i := arg1TreeBack; j := arg2TreeBack;
    for k := 1 to i do
        j := TreeBackStep(j);

```

Function TreeSize. **Input** None. **Output** $2(\text{size}(T) - 1)$, if T is contractable, and `null`, otherwise. **Assumes** None. **Local variables** `i` (length of the exploration walk), `i1` (counter for steps of the exploration walk), `k1` (T -edge label of the $(i - 1)$ -st edge of the exploration walk), `k2` (T -edge label of the i -th edge of the exploration walk), `d` (T -degree of cv), `d1` (T -degree of the i -th vertex of the exploration walk), and `td` (degree of T).

Notes $2(\text{size}(T) - 1)$ is the length of the exploration walk given in Proposition 3.4. The method to compute it is provided by the same proposition, i.e. `TreeSize` incrementally finds (line 6-22) the length of a walk which visits the current vertex exactly the number of times equal to its tree-degree plus 1 (the check is done in lines 13 and 14). Before increasing the length of the walk, we first make sure that the next move is possible (lines 7-12). If it is not, `TreeSize` returns `null`. This is correct, because if T has an invalid vertex, it is not contractable (`Valid > ContractDegree`). Otherwise it checks, if the walk went back to the starting vertex and returns, if the starting vertex was visited sufficiently many times. Also when `TreeSize` visits a vertex for the first time (lines 15-17), it adds its degree to the current total degree of T and returns `null`, if the total degree becomes larger than `ContractDegree` (lines 18-21).

The condition of the loop in line 6, makes sure that the current length `i` of the exploration walk is valid. If it is not, line 23 returns `null` because T is uncontractable. This is correct because on one hand $\text{size}(T) \leq \text{deg}(T)$ (at line 6, T has at least one edge) and on the other, by Proposition 3.4, exploration walk of length $2(\text{size}(T) - 1)$ visits all vertices of a tree of size $\text{size}(T)$ and returns to the starting vertex sufficiently many times. Since `Valid` ≥ 2 `ContractDegree`, if the length of the exploration walk becomes bigger than `Valid`, then $\text{size}(T) > \text{ContractDegree}$, so $\text{deg}(T) > \text{ContractDegree}$ and T is uncontractable.

`i` and `i1` are valid because of the condition of the loop in line 6. `k1` is valid because of the assumption that all vertices visited by the exploration walk of length $i - 1$ in line 7 are valid. `k2` is valid because of the condition on the output of `TreeForwardStep` in line 8. `d` is valid because of the condition on the output of `TreeDegree` in line 1. `td` is valid because at line 20 both `td` and `d1` are at most `ContractDegree`, and since `Valid` ≥ 2 `ContractDegree`, the addition in line 20 produces a valid result.

```

function TreeSize
1   d := TreeDegree;
   // if currvertex is invalid, then the tree is uncontractable
2   if d = null then return null;
3   if d = 0 then return 0;

4   i := 1;
5   td := 0;
6   while i ≤ Valid do
   // check if we can make one more step
   // from the exploration walk
7   k1 := TreeForward(i-1);
8   k2 := TreeForwardStep(k1);
9   if k2 = null then
   // if we cannot then the tree is uncontractable
10  TreeBack(i-1, k1);
11  return null;

```

```

12     TreeBack(i, k2);

    // check if we have visited the starting vertex
    // sufficiently many times
13     if TreeForward(i) = Current then d := d - 1;
    // if yes, then return the current length of
    // the exploration walk
14     if d = 0 then return i;

    // check if the end of the current exploration walk
    // is visited for the first time
15     for i1 := 0 to i - 1 do
16         if TreeForward(i) = TreeForward(i1) then break;
17     if i1 = i then
    // if it is, add its degree to the total degree
18         after TreeForward(i) do d1 ::= Prev(Degree)
    // if the total degree becomes too large,
    // then the tree is uncontractable
19         if d1 > ContractDegree then return null;
20         td := td + d1;
21         if td > ContractDegree then return null;

    // increase the length of the exploration walk by 1
22     i := i + 1;
23     return null;

```

Function Root. **Input** None. **Output** If T is uncontractable or $cv \in D_{cl-1}$, then **Root** returns 0, otherwise it returns the index of the first occurrence of $\text{root}(T)$ in the exploration walk. **Assumes** None. **Local variables** d ($2(\text{size}(T) - 1)$) and i (counter for steps of the exploration walk).

Notes According to the definition of $\text{root}(T)$ given in section 3.2.1, **Root** enumerates the vertices of T using the exploration walk starting from $(cv, 1)$ (lines 3-5) and finds the first vertex which is hooked to itself (line 4). d is valid because of the assignment in line 1, and i is valid because at line 3 T is contractable.

```

function Root
    // check if T is contractable
  1   d := TreeSize;
  2   if d = null or Prev(Done) then return 0;

    // if it is, find the first vertex in it which is hooked
    // to itself
  3   for i := 0 to d-1 do
  4       after TreeForward(i) do f1 := (Prev(Hook) = 0);
  5       if f1 then return i;

```

4.4.9. Contraction. The definitions of the functions in this section come from the definition of the contraction operation given in section 3.2.3.

Function IsEdge. **Input** i and j . **Output** If v is the vertex of $\Gamma_{T,i}(cv, 1)$, then **IsEdge** returns true iff (v, j) is a remaining edge of T (see the definition in section 3.2.3). **Assumes** i is valid, $0 \leq j \leq \delta_{cl-1}(v)$, $cv = \text{root}(T)$ and T is contractable. **Local variables** i (input), j (input), k (counter for steps of the exploration walk),

j_1 (counter for neighbors of the k -th vertex of the exploration walk), k_1 (counter for steps of the exploration walk on the tree of the j -th neighbor of v), d ($2(\text{size}(T) - 1)$), d_1 (degree of the k -th vertex of the exploration walk), and d_2 ($2(s - 1)$, where s is the size of the tree of the j -th neighbor of v).

Notes The definition of `IsEdge` follows exactly the definition of the remaining edges of T given in section 3.2.3. Let $e = (v, j)$, $w = \eta_{cl-1}(e)$, and T' be the hooking tree of w in \mathcal{C}_{cl-1} . Lines 2-5 check, if T' is contractable. Line 7 checks, if e is internal. Let u be the k -th vertex in the exploration walk of T starting from cv . Because of lines 9 and 10, at line 12 (u, j_1) is an edge before e in the sequence of the remaining edges of T given in section 3.2.3. Let $u' = \eta_{cl-1}(u, j_1)$. Lines 14-16 check whether u' is in T' . In line 13 we use that, if the hooking tree of u' in \mathcal{C}_{cl-1} is uncontractable, then u' is not from T' , because at this point T' is contractable.

i , j , and d are valid because T is contractable. d_2 and k_1 are valid, because at line 6 T' is contractable. Lines 9 and 10 ensure the validity of d_1 and j_1 .

```

function IsEdge
  i := arg1IsEdge; j := arg2IsEdge;
1  d := TreeSize;
   // if T' is uncontractable, then e remains
2  after TreeForward(i) do fl := MoveValid(j);
3  if not fl then return true;
4  after TreeForward(i), Move(j) do d2 := TreeSize;
5  if d2 = null then return true;

   // T' is contractable
6  for k := 0 to d-1 do
   // e does not remain, if it is an internal edge
7  if TreeForward(k) = (TreeForward(i), Move(j)) then
   return false;
8  if k > i then continue;

9  if k = i then d1 := j - 1;
   else
10  after TreeForward(k) do d1 ::= Prev(Degree);

   // e does not remain, if it is not the first edge
   // from T to T'
11  for j1 := 1 to d1 do
12  after TreeForward(k) do fl := MoveValid(j1);
13  if not fl then continue;

14  for k1 := 0 to d2-1 do
15  if (Treeforward(i), Move(j), TreeForward(k1)) =
   (TreeForward(k), Move(j1)) then
16  return false;
17  return true;

```

Statement “after P for every edge (i, j) do B”. **Input** P a path description relative to the current vertex, i and j names of local variables using this statement, B instruction(s) which might depend on the variables i and j . **Output** Let v be the vertex with path description P and T' is its hooking tree in \mathcal{C}_{cl-1} . This statement

executes B for all possible values of (i, j) such that (u, j) is a remaining edge of T' , where u is the vertex of $\Gamma_{T', i}(v, 1)$. **Assumes** $cl \geq 1$, all forward moves in P are valid, T' is contractable and $v = \text{root}(T')$. **Local variables** i_1 (counter for steps of the exploration walk on T' starting from $(v, 1)$), d_1 ($2(\text{size}(T') - 1)$), d_2 (degree of the i -th vertex of the exploration walk on T' starting from $(v, 1)$).

Notes Lines 3-5 check, if this is the first time the exploration walk on T' starting from $(v, 1)$ visits the i -th vertex v . If so, lines 7-9 enumerate the remaining edges of T' incident to v . All local variables, and i and j , are valid because T' is contractable.

```

statement after P for every edge (i, j) do B
1   after P do d1 := TreeSize;
2   for i := 0 to d1 - 1 do
    // visit only once every vertex of T'
3   for i1 := 0 to i - 1 do
4   if (P, TreeForward(i)) = (P, TreeForward(i1)) then
    break;
5   if i1 < i then continue;

6   after P, TreeForward(i) do d2 ::= Prev(Degree);
7   for j := 1 to d2 do
8   after P do f1 := IsEdge(i, j);
9   if not f1 then continue;

// if (i, j) is a remaining edge, then execute B
10  B;

```

Function Degree. **Input** None. **Output** $\delta_{cl}(cv)$. **Assumes** None. **Local variables** i (counter for steps of the exploration walk), j (counter for neighbors of the i -th vertex of the exploration walk), and d (degree of cv). **Notes** To obtain the degree of the current vertex, we just enumerate all remaining edges of T . If T is not contractable, then, by definition, the degree comes from a previous level (line 2). Line 2 handles the case when $cv \in I_{cl}$, and line 3 the case when $cv \in D_{cl}$. All local variables are valid because at line 3 T is contractable.

GraphDegree returns the degree of the current vertex in the input graph G .

```

function Degree
1   if level = 0 then return GraphDegree;
2   if not Contraction or TreeSize = null then return Prev(Degree);
3   if Prev(Done) or Root  $\neq$  0 then return 0;

4   d := 0;
5   after Current for every edge (i,j) do d := d + 1;
6   return d;

```

Procedure Neighbor. **Input** i . **Output** Moves the current vertex to $\eta_{cl}(cv, i)$. **Assumes** $0 \leq i \leq \delta_{cl}(cv)$. **Local variables** i (input), l (counter for steps of the exploration walk), j (counter for neighbors of the l -th vertex of the exploration walk), and d ($2(\text{size}(T) - 1)$).

Notes The definition of **Neighbor** follows the definitions in section 3.2.3. First we make sure that T is contractable (lines 4 and 10). If not, then we call recursively. Otherwise, i is the index of a remaining edge e of T , and we locate e and move along it (lines 14-20). Once we move along e , we move the current vertex to the representative of the new current vertex, i.e. the root of the new current hooking tree

T' , if it is contractable (lines 6, 12, and 19).

i is valid because at line 8 `argNeighbor` is valid, and the other local variables are valid because at line 14 T is contractable.

`GraphNeighbor(i)` moves the current vertex to its i -th neighbor in the input graph G .

```

procedure Neighbor
1   if level = 0 then GraphNeighbor(argNeighbor);
   if not Contraction then Prev(Neighbor);
2   // handle the self-loop case
3   if argNeighbor = 0 then return;

4   if argNeighbor > Valid then
   // if  $T$  is uncontractable, call recursively
5   Prev(Neighbor);
   // if  $T'$  is contractable, move to its root
6   if TreeSize  $\neq$  null then TreeForward(Root);
7   return;
8    $i :=$  argNeighbor;

9    $d :=$  TreeSize;
10  if  $d =$  null then
   //  $T$  is uncontractable
11  Prev(Neighbor( $i$ ));
12  if TreeSize  $\neq$  null then TreeForward(Root);
13  return;

   //  $T$  is contractable
14  after Current for every edge ( $l, j$ ) do
15   $i := i - 1$ ;
   // check if ( $l, j$ ) is  $e$ 
16  if  $i > 0$  then continue;

   // move to  $e$  and then along  $e$ 
17  TreeForward( $l$ );
18  Prev(Neighbor( $j$ ));
   // move to the root of  $T'$ 
19  if TreeSize  $\neq$  null then TreeForward(Root);
20  return;

```

Function BackLabel. **Input** i . **Output** $\beta_{cl}(cv, i)$. Uses the array method described in section 6.1.2 of taking arguments and returning values. **Assumes** $0 \leq i \leq \delta_{cl}(cv)$. **Local variables** i (input), l (counter for steps of the exploration walk), k_1 (same as l), j (counter for the neighbors of the l -th vertex of the exploration walk), d ($2(\text{size}(T) - 1)$), nd (output), k (counter for steps of the exploration walk on T' starting from $(u, 1)$), j_1 (counter for neighbors of the k -th vertex of the exploration walk on T' starting from $(u, 1)$), and r (index of the first occurrence of the root of T' in the exploration walk on T' starting from $(u, 1)$), where u is the j -th neighbor of the l -th vertex of the exploration walk and T' is the hooking tree of u .

Notes The first case of `BackLabel` is when T is contractable. In this case we find the remaining edge e of T with index i (lines 12-14). Let $v = \eta_{cl-1}(e)$ and T'

be the hooking tree of v in C_{cl-1} . If T' is uncontractable, then we call recursively, because in this case the back-label comes from the previous level of recursion (line 22). Otherwise we have to find the index \mathbf{nd} of the first remaining edge e' of T' which goes from T' to T (lines 24-32). This is the new back-label. To find the index of e' , first we find the root of T' (line 20) and then enumerate all remaining edges of T' (lines 25-32). For each remaining edge of T' we check if it goes to T (line 30-32). In lines 27-29, we use that, if a remaining edge of T' goes to an uncontractable hooking tree, then it does not go to T , because at this point T is contractable. The case when T is uncontractable is handled by `BackLabelAux` (lines 5 and 10).

\mathbf{i} is valid because of line 4. \mathbf{d} is valid because of the assignment in line 8. \mathbf{l} , \mathbf{j} , and \mathbf{k}_1 are valid because at line 12 T is contractable. \mathbf{r} is valid because of line 20. \mathbf{j}_1 , \mathbf{k} , and \mathbf{nd} are valid because at line 24 T' is contractable.

The recursive call in line 22 does not assign the returned value to a local variable, i.e. this call returns a value at some higher level of recursion, depending on the array for returning values of `BackLabel`. This call is the reason why `BackLabel` returns through an array instead of a global variable. The conventional thing to do is to store the result of this call locally, and once the **after** statement has restored the original current vertex, return the stored value. This will not work for us, because the value returned from the recursive call might be invalid. Instead, using that the only reason why we store the returned value is to pass it back, when `BackLabel` produces a result we let it store the result at the level at which it is requested. This works because `BackLabel` is always called on the previous level of recursion.

`GraphBackLabel(i)` returns the back-label of the i -th edge incident to cv in the input graph G .

```

function BackLabel
1   if level = 0 then return GraphBackLabel(argBackLabel);
2   if not Contraction then return Prev(BackLabel);
3   if argBackLabel = 0 then return 0;

   // if currvertex is invalid call BackLabelAux
4   if argBackLabel > Valid then
5       BackLabelAux;
6       return;
7   i := argBackLabel;

   // if T is uncontractable call BackLabelAux
8   d := TreeSize;
9   if d = null then
10      BackLabelAux;
11      return;

   // T is contractable
12  after Current for every edge (l, j) do
13      i := i - 1;
   // find e
14      if i > 0 then continue;

15      after TreeForward(l) do
16          fl := MoveValid(j);

```

```

17         if fl then
18             after Move(j) do
19                 fl := (TreeSize ≠ null);
20                 if fl then r := Root;

21     if not fl then
22         // if T' is uncontractable call recursively
23         after TreeForward(l) do Prev(BackLabel(j));
24         return;

25     // T' is contractable
26     nd := 0;
27     // find the first edge of T' which goes to T and
28     // return its index
29     after TreeForward(l), Move(j), TreeForward(r)
30         for every edge (k, j1) do
31             nd := nd + 1;
32             after TreeForward(l), Move(j),
33                 TreeForward(r), TreeForward(k) do
34                 fl := MoveValid(j1);
35                 if not fl then continue;

36     for k1 := 0 to d-1 do
37         if (TreeForward(l), Move(j),
38             TreeForward(r), TreeForward(k), Move(j1)) =
39             TreeForward(k1) then
40             return nd;

```

Function BackLabelAux. **Input** i . **Output** $\beta_{cl}(cv)$. To take argument and return value `BackLabelAux` uses the arrays of `BackLabel`. **Assumes** $0 \leq i \leq \delta_{cl}(cv)$, T is uncontractable. **Local variables** l (counter for steps of the exploration walk on T' starting from $(v, 1)$), j (counter for neighbors of the l -th vertex of the exploration walk on T' starting from $(v, 1)$), k (same as l), \mathbf{bl} ($\beta_{cl-1}(cv, i)$), \mathbf{nbl} (output), \mathbf{r} (index of the first occurrence of the root of T' in the exploration walk on T' starting from $(v, 1)$), and d ($2(\text{size}(T') - 1)$), where T' and v are as in the note for `BackLabel`.

Notes The definition of `BackLabelAux` follows the definitions given in section 3.2.3 when T is uncontractable. Let v and T' be as in the note for `BackLabel`. If T' is uncontractable, the back-label is inherited from the previous level of recursion, so we call `BackLabel` recursively (lines 3 and 10). Otherwise at line 12, T' is contractable, the current vertex is v (because of line 5), and \mathbf{bl} is the back-label of e (because of line 1). So we have to find the index of (v, \mathbf{bl}) in T' ((v, \mathbf{bl}) is a remaining edge of T' because T is uncontractable). Line 12 finds the root of T' , and lines 13 and 14 find the index k of the first occurrence of v in the exploration walk of T' starting from its root. Lines 16-20 enumerate the remaining edges of T' until we find (v, \mathbf{bl}) .

\mathbf{bl} is valid by the assumption for the return convention of `BackLabel` for line 1. d is valid because of the assignment in line 6. \mathbf{r} , l , j , k , and \mathbf{nbl} are valid because at line 12 T' is contractable.

Just like for `BackLabel`, the calls to `BackLabel` in lines 3 and 10 return values at some higher level of recursion. The calls to `BackLabel` in lines 1, 3, and 10 do not have arguments – by convention this means that the argument to `BackLabel` comes

from a higher level of recursion.

The case when T is uncontractable is the reason why the argument to `BackLabel` is passed through an array instead of a global variable. More precisely, the problem is when the current vertex is invalid, then the argument i to `BackLabel`, which is the label of an edge incident to cv , might be invalid and storing it locally will be impossible. In this case we still want to be able to use the value of i after calling functions which can potentially change the value of a global argument to `BackLabel`. The decision is to let the value of the argument stay at the level which produced it, because it certainly is valid for this level. For this to work, it is important that the value of the argument stored in the array is not changed while processing the call to `BackLabel`. Fortunately this does not happen, because `BackLabel` is always called on the previous level of recursion.

```

function BackLabelAux
1   bl := Prev(BackLabel);
2   if bl = null then
      // if T' is uncontractable call recursively
3     Prev(BackLabel);
4     return;

      // move along e
5   Prev(Neighbor(argBackLabel));

6   d := TreeSize;
7   if d = null then
8     // if T' is uncontractable go back and call recursively
9     Prev(Neighbor(bl));
10    Prev(BackLabel);
11    return;

      // T' is contractable
12   r := Root;
      // find the index of the first occurrence of v in
      // the exploration walk on T' starting from (r,1)
13   for k := 0 to d - 1 do
14     if TreeForward(r), TreeForward(k) = Current then break;

      // compute the new back-label
15   nbl := 0;
16   after TreeForward(r) for every edge (l, j) do
      // increase the new back-label by one for every edge
      // that happens before e
17     nbl := nbl + 1;

18     if l = k and j = bl then
      // if we are at (v,bl) move back and return
      // the new back-label
19     Prev(Neighbor(bl));
20     return nbl;

```

Function Move. Input i . Output Let $(v, j) = \mu_{cl-1}(cv, i)$. Move returns j and

moves the current vertex to v . **Assumes** $cl \geq 1$, $0 \leq i \leq \delta_{cl-1}(cv)$, i and j valid. **Local variables** i (input) and j (output), which are valid by the assumption about the argument of Move.

function Move

```

    i := argMove;
    j := Prev(BackLabel(i));
    Prev(Neighbor(i));
    return j;

```

Function MoveValid. **Input** i . **Output** True iff $\beta_{cl-1}(cv, i)$ is valid. **Assumes** $cl \geq 1$, $0 \leq i \leq \delta_{cl-1}(cv)$, i valid. **Local variables** None.

function MoveValid

```

    return Prev(BackLabel(argMoveValid))  $\neq$  null;

```

4.4.10. Solving undirected st-connectivity. Procedure MoveToRep. **Input** None. **Output** Moves the current vertex to $\text{rep}_{R_{cl}}(cv)$. **Assumes** None. **Local variables** None.

procedure MoveToRep

```

    if level > 0 then
        Prev(MoveToRep);
        if Contraction and TreeSize  $\neq$  null then TreeForward(Root);

```

Global function Connected. **Input** s and t . **Output** True iff s and t are connected in G .

global function Connected

```

    level :=  $5 \cdot 2^{\lceil \log \log n \rceil} - 3$ ;
    currvertex := arg1Connected; MoveToRep;
    v := currvertex;
    currvertex := arg2Connected; MoveToRep;
    return v = currvertex;

```

5. Proofs of Theorem 3.12 and Corollary 3.13. The proofs in this section are a direct translation into our notation of equivalent statements from [6] and are given here only for completeness. We use the notation of section 3.3.

LEMMA 5.1. *Assume that \mathcal{C}_l is nice and $P_{l+1} = \langle \text{Hook}, k \rangle$.*

- (i) *If $v \in I_l$, then $v \in I_{l+r(k)}$, $\delta_{l+r(k)}(v) = \delta_l(v)$, and $H_{l+r(k)}(v) = H_l(v)$.*
- (ii) *If $v \in I_{l+r(k)} - I_l$, then $\delta_{l+r(k)}(v) \leq \text{dexp}(k+2)$.*
- (iii) *\mathcal{C}_{l+1} is nice. If $k > 0$, then \mathcal{C}_{l+2} is nice.*

Proof. (i) The state and degree of a vertex change only during a contraction operation. The same holds for the hooking edges of inactive vertices. Let T be the hooking tree of v in \mathcal{C}_l . Since \mathcal{C}_l is nice, $\text{deg}(T) > \text{dexp}(k+2)$. For $l+1 \leq i \leq l+r(k)$, we have that $\text{Arg}_i \leq k+1$, so trees of degree more than $\text{dexp}(k+2)$ are never contracted. Therefore the status, degree, and hooking edge of v is never changed.

(ii) Since $v \in I_{l+r(k)} - I_l$ and because a done vertex stays done, we have that $v \in A_l$, and so $\delta_l(v) \leq \text{dexp}(k+2)$, because \mathcal{C}_l is nice. Hooking does not change degrees. For $l+1 \leq i \leq l+r(k)$, we have that $\text{Arg}_i \leq k+1$, so any active vertex appearing after a contraction must have degree at most $\text{dexp}(k+2)$. Let i be the largest $l \leq i < l+r(k)$ such that $v \in A_i$. Then $\delta_i(v) \leq \text{dexp}(k+2)$ and its degree does not change afterwards. Hence $\delta_{l+r(k)}(v) \leq \text{dexp}(k+2)$.

(iii) We have that $P_{l+2} = \langle \text{Contract}, k \rangle$ or $P_{l+2} = \langle \text{Contract}, 1 \rangle$, if $k = 0$. Since a hooking operation only changes the hooking edges of active vertices, \mathcal{C}_{l+1} is nice. Furthermore, for a hooking tree T' in \mathcal{C}_{l+1} which contains an inactive vertex, we have that $\text{size}(T') > \text{dexp}(k+1)$

If $k > 0$, then $P_{l+3} = \langle \text{Hook}, k-1 \rangle$. Let T be a hooking tree in \mathcal{C}_{l+2} composed of inactive vertices. Because $P_{l+2} = \langle \text{Contract}, k \rangle$, we have that $\delta_{l+2}(v) \leq \text{dexp}(k+1)$, for $v \in A_{l+2}$, and $\deg(T) > \text{dexp}(k+1)$. There are two possibilities for T – either it was a hooking tree in \mathcal{C}_{l+1} composed of active vertices or it contains a hooking tree T' of \mathcal{C}_{l+1} which contained an inactive vertex. In the first case by Lemma 3.9, $\text{size}(T) > \text{dexp}(k)$. This is also true in the second case because, as mentioned at the end of the previous paragraph, $\text{size}(T) > \text{dexp}(k+1) > \text{dexp}(k)$. Therefore \mathcal{C}_{l+2} is nice. \square

LEMMA 5.2. *Assume that \mathcal{C}_l is nice and $P_{l+1} = \langle \text{Hook}, k \rangle$.*

(i) *If T is hooking tree in $\mathcal{C}_{l+r(k)}$ composed of inactive vertices, then $\text{size}(T) > \text{dexp}(k+1)$ and $\deg(T) > \text{dexp}(k+2)$.*

(ii) *If $k > 0$, then $\mathcal{C}_{l+2+r(k-1)}$ and $\mathcal{C}_{l+2+2r(k-1)}$ are nice.*

Proof. Let $l_0 = l+2$, $l_1 = l_0 + r(k-1)$, $l_2 = l_1 + r(k-1)$, and $l_3 = l_2 + 1 (= l+r(k))$.

(i) Let T be a hooking tree of \mathcal{C}_{l_3} composed of inactive vertices, i.e. $V_T \subseteq I_{l_3}$. Since $P_{l_3} = \langle \text{Contract}, k+1 \rangle$, we have that

$$(5.1) \quad \deg(T) > \text{dexp}(k+2),$$

which ensures the degree part of the first item of the lemma. We are left with proving the size part.

Case 1: $V_T \cap I_l \neq \emptyset$. By Lemma 5.1(i), the vertices in $V_T \cap I_l$ form a hooking tree T' of \mathcal{C}_l composed of inactive vertices. Since \mathcal{C}_l is nice, $\text{size}(T') > \text{dexp}(k+1)$, and hence $\text{size}(T) > \text{dexp}(k+1)$.

Case 2: $V_T \cap I_l = \emptyset$. We do induction on k . If $k = 0$, then by Lemma 3.9 and (5.1), $\text{size}(T) > \text{dexp}(1)$.

Assume that $k > 0$.

From \mathcal{C}_l nice and $P_{l_0} = \langle \text{Contract}, k \rangle$, follows that $I_l \subseteq I_{l_0}$. Also by Lemma 5.1(i), $I_{l_0} \subseteq I_{l_1} \subseteq I_{l_2}$. Finally $I_{l_3} \subseteq I_{l_2}$, because $Op_{l_3} = \text{Contract}$. Hence $V_T \subseteq I_{l_2}$.

For $l_0 + 1 \leq i \leq l_2$, we have that $Arg_i \leq k$. Therefore we can use the same argument as in the proof of Lemma 5.1(ii) to see that for every $v \in I_{l_2} - I_{l_0}$

$$(5.2) \quad \delta_{l_2}(v) \leq \text{dexp}(k+1).$$

Case 1.1: $V_T \cap I_{l_0} = \emptyset$. By (5.2), $\delta_{l_2}(v) \leq \text{dexp}(k+1)$, for every $v \in V_T$. Hence $\deg(T) \leq \text{size}(T)\text{dexp}(k+1)$. Thus, from (5.1), follows that $\text{size}(T) > \text{dexp}(k+1)$.

Case 1.2: $V_T \cap I_{l_0} \neq \emptyset$. From Lemma 5.1(i), follows that $V_T \cap I_{l_0}$ form a hooking tree T' in \mathcal{C}_{l_0} . Let $d = \deg(T')/\text{size}(T')$, the average degree of a vertex of T' .

Case 1.2.1: $d \leq \text{dexp}(k+1)$. We have that $\deg(T') \leq \text{dexp}(k+1)\text{size}(T')$. Also for every $v \in V_T - V_{T'} \subseteq I_{l_2} - I_{l_0}$, by (5.2), $\delta_{l_2}(v) \leq \text{dexp}(k+1)$. Hence

$$\deg(T) = \sum_{v \in V_{T'}} \delta_{l_2}(v) + \sum_{v \in V_T - V_{T'}} \delta_{l_2}(v) \leq \text{dexp}(k+1)\text{size}(T),$$

and so $\text{size}(T) > \text{dexp}(k+1)$, because of (5.1).

Case 1.2.2: $d > \text{dexp}(k+1)$. Since $V_{T'} \subseteq V_T$, we have that $V_{T'} \cap I_l = \emptyset$. Hence $V_{T'} \subseteq A_l$ and so, by Lemma 3.9, $\text{size}(T') > \deg(T')/\text{size}(T') = d > \text{dexp}(k+1)$. Therefore, by (5.2)

$$\begin{aligned} \deg(T) &\leq \deg(T') + \text{dexp}(k+1)(\text{size}(T) - \text{size}(T')) \\ &< \text{size}^2(T') + \text{size}(T')(\text{size}(T) - \text{size}(T')) = \text{size}(T')\text{size}(T) \leq \text{size}^2(T). \end{aligned}$$

Hence in this last case and using (5.1) again, $\text{size}(T) > \text{dexp}(k+1)$ as well.

(ii) By Lemma 5.1(iii), \mathcal{C}_{l_0} is nice. Hence, by the first item of the lemma (notice that $P_{l_0+1} = \langle \text{Hook}, k-1 \rangle$), for any hooking tree T in \mathcal{C}_{l_1} composed of inactive vertices, we have that $\text{size}(T) > \text{dexp}(k)$ and $\text{deg}(T) > \text{dexp}(k+1)$. Furthermore, since $P_{l_1} = \langle \text{Contract}, k \rangle$, for any $v \in A_{l_1}$, $\delta_{l_1}(v) \leq \text{dexp}(k+1)$. Therefore \mathcal{C}_{l_1} is nice. \mathcal{C}_{l_2} is nice for similar reasons.

□

LEMMA 5.3. *Assume that \mathcal{C}_l is nice and $P_{l+1} = \langle \text{Hook}, k \rangle$. Then for every $v \in A_{l+r(k)}$,*

$$|\{u \in A_l : \text{rep}_{l+r(k)}(u) = v\}| \geq \text{dexp}(k).$$

Proof. We do induction on k . Let $k = 0$. Consider $v \in A_{l+2}$. v must be a root of a hooking tree T in \mathcal{C}_{l+1} and $V_T \subseteq A_{l+1}$, because \mathcal{C}_l is nice and $P_{l+2} = \langle \text{Contract}, 1 \rangle$. Hooking does not change the states of vertices and so $A_l = A_{l+1}$. By Lemma 3.9, T has at least two vertices and therefore v represents at least two vertices from A_l .

Assume that $k > 0$. Let $l_0 = l + 2$, $l_1 = l_0 + r(k-1)$, $l_2 = l_1 + r(k-1)$, and $l_3 = l_2 + 1 (= l + r(k))$. As can be seen in the proof of Lemma 5.2, we have that $A_{l_2} \subseteq A_{l_1} \subseteq A_{l_0} \subseteq A_l$ and $A_{l_2} \subseteq A_{l_3}$. Let $v \in A_{l_3}$. We have that v is in either A_{l_2} or in I_{l_2} .

Assume first that $v \in I_{l_2}$. Since $v \in A_{l_3}$ and $P_{l_3} = \langle \text{Contract}, k+1 \rangle$, v is a root of a hooking tree T of \mathcal{C}_{l_2} composed of inactive vertices which contracts to v . $V_T \subseteq A_l$, because if there exists $u \in V_T \cap I_l$, then by Lemma 5.1(i) $u \in I_{l_3}$, but we have that $V_T \cap I_{l_3} = \emptyset$. By Lemma 5.2(ii) \mathcal{C}_{l_1} is nice and therefore by Lemma 5.2(i), $\text{size}(T) > \text{dexp}(k)$. Thus v represents at least $\text{dexp}(k)$ vertices from A_l .

Assume now that $v \in A_{l_2}$. Let

$$U_v = \{u \in A_{l_1} : \text{rep}_{l_2}(u) = v\}.$$

By the inductive hypothesis $|U_v| \geq \text{dexp}(k-1)$. Let $u \in U_v$. Again by the inductive hypothesis $|\{w \in A_{l_0} : \text{rep}_{l_1}(w) = u\}| \geq \text{dexp}(k-1)$. Hence

$$|\{w \in A_{l_0} : \text{rep}_{l_2}(w) = v\}| \geq \text{dexp}(k-1)\text{dexp}(k-1) = \text{dexp}(k).$$

Since $A_{l_0} \subseteq A_l$, the statement follows in this case as well.

□

We are ready to prove Theorem 3.12.

Proof. [Proof of Theorem 3.12] $|A_{r(k)}| \leq \max\{|A_0|/\text{dexp}(k), 1\}$ follows from Lemma 5.3 and the fact that the sets of vertices represented by different vertices from $A_{r(k)}$ are disjoint. $\text{size}(T_v) > \text{dexp}(k+1)$, for $v \in I_{r(k)}$, follows from Lemma 5.2(i).

We have to prove now that \mathcal{C}_l is nice, for $1 \leq l < r(k)$. We prove by induction on t that, if \mathcal{C}_l is nice and $P_{l+1} = \langle \text{Hook}, t \rangle$, then \mathcal{C}_i is nice for $l+1 \leq i < l+r(t)$ and if $0 < t < k$, then $\mathcal{C}_{l+r(t)}$ is also nice.

By Lemma 5.1(iii), \mathcal{C}_{l+1} is nice. This takes care of $t = 0$. Assume that $t > 0$. By Lemma 5.1(iii) and Lemma 5.2(ii), \mathcal{C}_{l+2} , $\mathcal{C}_{l+2+r(t-1)}$, and $\mathcal{C}_{l+2+2r(t-1)}$ are nice. By the inductive hypothesis applied twice \mathcal{C}_i is nice, for $l+3 \leq i < l+2+r(t-1)$ and $l+3+r(t-1) \leq i < l+2+2r(t-1)$. Since $P_{l+r(t)} = \langle \text{Contract}, t+1 \rangle$, we have that $\delta_{l+r(t)}(v) \leq \text{dexp}(t+2)$, for $v \in A_{l+r(t)}$. By Lemma 5.2(i), for any hooking tree of $\mathcal{C}_{l+r(t)}$ composed of inactive vertices $\text{size}(T) > \text{dexp}(t+1)$ and $\text{deg}(T) > \text{dexp}(t+2)$.

If $t < k$, then $P_{l+r(t)+1}$ is either $\langle \text{Contract}, t+2 \rangle$ or $\langle \text{Hook}, t \rangle$, and in both cases $\mathcal{C}_{l+r(t)}$ is nice. \square

Finally let us prove Corollary 3.13.

Proof. [Proof of Corollary 3.13] Let $k = \lceil \log \log n \rceil$. By Theorem 3.12,

$$|A_{r(k)}| \leq \max\{n/\text{dexp}(k), 1\} = 1.$$

On the other hand, again by Theorem 3.12, every hooking tree in $\mathcal{C}_{r(k)}$ composed of inactive vertices has size more than $\text{dexp}(k+2) \geq n^2$. Therefore $I_{r(k)} = \emptyset$.

Suppose that u and v are connected in G , but $r_1 = \text{rep}_{r(k)}(u) \neq \text{rep}_{r(k)}(v) = r_2$. As we saw $|A_{r(k)}| \leq 1$ and $I_{r(k)} = \emptyset$, so w.l.o.g. we can assume that $r_1 \in D_{r(k)}$. Let l be the largest such that $r_1 \notin D_l$. $\delta_{l+1}(r_1) > 0$, because $r_1 \neq r_2$, r_1 and r_2 are connected in G , and r_1 inherits the neighbors of the vertices it represents. This contradicts the fact that r_1 becomes done, if it is a root of a hooking tree and its degree is 0 (it can also become done if it is a non-root of a hooking tree which is contracted, but then it would not be a representative). \square

Acknowledgments The author is grateful to Prof. Anna Gál for help in the preparation of this paper and Prof. Vijaya Ramachandran for pointing out the problem and many helpful discussions.

REFERENCES

- [1] R. Aleliunas, R. Karp, R. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th IEEE Symposium on Foundations of Computer Science*, pages 218–223, 1979.
- [2] R. Armoni, A. Ta-Shma, A. Wigderson, and S. Zhou. $SL \subseteq L^{\frac{4}{3}}$. In *20th ACM Symposium on Theory of Computing*, pages 230–239, 1997.
- [3] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for ultracomputer and PRAM. In *International Conference on Parallel Processing*, pages 175–179, 1983.
- [4] F. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25:659–666, 1982.
- [5] K. Chong, Y. Han, and T. Lam. On the parallel time complexity of undirected connectivity and minimum spanning trees. *Journal of the ACM*, 48(2):297–323, 2001.
- [6] K. Chong and T. Lam. Finding connected components in $O(\log n \log \log n)$ time on the EREW PRAM. *Journal of Algorithms*, 18(3):378–402, 1995.
- [7] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. In *27th IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.
- [8] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *27th IEEE Symposium on Foundations of Computer Science*, pages 492–501, 1986.
- [9] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems. In *7th ACM-SIAM Symposium on Discrete Algorithms*, pages 438–447, 1996.
- [10] D. Hirschberg, A. Chandra, and D. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.
- [11] D. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} |V|)$ parallel time for the CREW PRAM. In *32nd IEEE Symposium on Foundations of Computer Science*, pages 688–697, 1991.
- [12] D. Karger, N. Nisan, and M. Parnas. Fast connected components algorithm for the EREW PRAM. In *4th ACM Symposium on Parallel Algorithms and Architectures*, pages 373–381, 1992.
- [13] M. Koucký. Universal traversal sequences with backtracking. In *16th IEEE Conference on Computational Complexity*, pages 21–27, 2001.
- [14] H. Lewis and C. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19:161–187, 1982.

- [15] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992.
- [16] N. Nisan, E. Szemerédi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *33rd IEEE Symposium on Foundations of Computer Science*, pages 24–29, 1992.
- [17] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM Journal on Computing*, 31(6):1879–1895, 2002.
- [18] O. Reingold. Undirected st-connectivity in log-space. In *37th ACM Symposium on Theory of Computing*, pages 376–385, 2005.
- [19] C. Savage and J. JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM Journal on Computing*, 10(4):682–691, 1981.
- [20] W. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.
- [21] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.
- [22] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [23] R. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.