Electronic Colloquium on Computational Complexity, Report No. 114 (2004)

An $O(\log n \log^{(2)} n)$ Space Algorithm for Undirected s,t-Connectivity*

Vladimir Trifonov†

Abstract

We present a deterministic $O(\log n \log^{(2)} n)$ space algorithm for undirected s,t-connectivity. It is based on the deterministic EREW algorithm of Chong and Lam [CL93] and uses the universal exploration sequences for trees constructed by Koucký [K01]. Our result improves the $O(\log^{4/3} n)$ bound of Armoni et al. [ATSWZ97] 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 recently by Reingold [R04].

1 Introduction

The problem we are concerned with is s,t-connectivity in an undirected graph G with n vertices, 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 s,t-connectivity plays an important role in complexity theory because directed s,t-connectivity is **NL**-complete [S70] and undirected s,t-connectivity is **SL**-complete [LP82].

Linear time and space sequential algorithm for solving even the harder directed s,t-connectivity problem have been known for a long time [T72]. The problem of developing more efficient space and parallel algorithms was poised.

If we allow one-sided randomness, the result of [AKL$^+$79] shows that undirected s,t-connectivity can be solved in $O(\log n)$ space. The starting point of deterministic space efficient sequential algorithms is the $O(\log^2 n)$ space algorithm for directed s,t-connectivity of Savitch [S70]. For a long time this was the best result even for undirected s,t-connectivity. The space bound for undirected graphs was first improved by Nisan et al. [NSW92] to $O(\log^{3/2} n)$ and then to $O(\log^{4/3} n)$ by Armoni et al. [ATSWZ97]. Both of these results depend on the efficient construction of universal traversal sequences by Nisan [N90]. Finally, simultaneous to our result, the space complexity of undirected s,t-connectivity was shown by Reingold [R04] to be $O(\log n)$ using a different set of techniques.

Developing efficient parallel algorithms for undirected s,t-connectivity 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

*Submitted to STOC 2005.

†Dept. of Computer Science, University of Texas at Austin, Austin, TX 78712-1188, USA. e-mail: vladot@cs.utexas.edu

ISSN 1433-8092

value. The results of [HCS79, SJ81, CLC82, JM91, KNP92] are concerned with the CREW PRAM model, [SV82, AS83, CV86, G86] with the CRCW PRAM model, and [HZ96, CL93, CHL99, PR99] with the EREW PRAM. The state of the art in parallel algorithms for undirected s,t-connectivity are the results of [CHL99], which shows that the problem can be solved on the EREW PRAM in $O(\log n)$ time with $O(m+n)$ processors, and [PR99], which demonstrates a randomized EREW PRAM algorithm running in time $O(\log n)$ with optimal number of processors.

The starting point of our algorithm is the $O(\log n \log^{(2)} n)$ time deterministic EREW PRAM algorithm with $O(m+n)$ processors of [CL93], which we call the CL algorithm. Using it we define a sequential algorithm with the same space requirements as the CL algorithm. We use the sequential algorithm to define configuration as a mathematical structure, 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 points of its execution. We use the sequence of configurations to trivially define a $O(\log^2 n)$ 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 $O(\log^2 n)$ algorithm into an algorithm which uses $O(\log n \log^{(2)} n)$ space.

The possibility of using parallel algorithms to define space efficient sequential algorithms was suggested by Prof. Vijaya Ramachandran in 2000. She conjectured an $O(\log n \log^{(2)} n)$ algorithm derived from [CL93] and an alternate simple $O(\log^{3/2} n)$ space algorithm derived from the algorithm of [JM91], by using the max-degree hooking scheme of [CL93]. She observed that the step needing derandomization in [NSW92] 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 [CL93] gives the additional benefit that trees with small neighborhoods are small. 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 $\Omega(\log^{(2)} n)$ and $\Theta(\log n)$, depending on the level. A key tool for our method are the exploration walks on trees defined in [K01]. Exploration walks on trees are similar to the Euler tour technique used by [TV85] 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 describes briefly the CL algorithm and the sequential algorithm derived from it. Section 3 gives the formal definition of a labeled multi-graph and operations on graphs, as well as the definition of configuration and sequence of configurations. Section 4 gives an overview of the space efficient algorithm for solving undirected s,t-connectivity. The appendix contains the complete pseudo-code of the algorithm and a discussion about its execution and translation to a Turing Machine.

2 The Chong-Lam algorithm

The CL algorithm uses a hook and contract approach. There are several stages of hooking and contraction. Before every stage every vertex of the original graph is in exactly one of three states – active, inactive, and done; all active and inactive vertices have non-zero degree and there are no multi-edges between active vertices; the inactive vertices are organized in a set of hooking trees.

In a hooking phase the active vertices in parallel choose to hook to one of their current neighbors and thus either become part of existing hooking trees or form new ones. The fact that the components formed by the hooking of vertices are trees is ensured by the special hooking scheme of the CL algorithm.

In the contraction phase some of the current hooking trees are contracted to a representative vertex. Which trees are contracted is determined by a parameter, which depends on the stage 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 vertices and the rest of its vertices become done and are removed from further consideration. Also all multi-edges between new active vertices are cleaned-up. Finally the vertices of every uncontracted tree become inactive.

The processing required by the hooking is performed in parallel time $O(\log d)$, where d is the degree of the active vertex, using pointer jumping. The important part of the contraction procedure is checking the degree of a hooking tree. In parallel this could be done in $O(\log c)$, where c is the value of the contraction parameter, by using a pointer jumping and 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(`$2^{2^{k}}$`);`
        `Connect(k` $-$ `1);`
        `Connect(k` $-$ `1);`
    `Contract(`$2^{2^{k+1}}$`);`

The correctness of the CL algorithm ensures that a call to `Connect`($\lceil \log^{(2)} n \rceil$) 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.

From the CL algorithm we extract trivially a sequential algorithm with the same linear space requirement as the CL algorithm. We fix an ordering on the edges incident on 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 details of the sequential algorithm are given in the following section.

3 Definitions

3.1 Graphs, trees, and exploration walks

An undirected multi-graph is a graph with possibly multiple edges between any 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 1. An *undirected multi-graph* G is a triple $\langle V, \delta, \mu \rangle$, where V is a set, $\delta : V \to \mathbb{N}$, and $\mu : E \to E$ is a bijection such that $\mu(\mu(e)) = e$ and $\mu(v, 0) = (v, 0)$, where $E = \{(v, i) : v \in V \text{ and } 0 \le i \le \delta(v)\}$.

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 $\eta : E \to V$ and $\beta : E \to \mathbb{N}$ be the first and the second component of μ. 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 2. A graph $G' = \langle V', \delta', \mu' \rangle$ is a subgraph of G, if $V' \subseteq V$ and for every $u, v \in V'$, $|\{i : \eta'(u, i) = v\}| \le |\{i : \eta(u, i) = v\}|$.

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

Definition 3. Let G be a graph. Let $\Delta : E \times \mathbb{Z} \to E$ 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 $\Gamma_{G,k}, \Gamma'_{G,k} : E \times \mathbb{Z}^k \to E$ inductively on $k \ge 0$. First $\Gamma_{G,0}(e) = \Gamma'_{G,0}(e) = e$. Now let $\Gamma_{G,k+1}(e, (j_1, \ldots, j_{k+1})) = \Delta(\mu(\Gamma_{G,k}(e, (j_1, \ldots, j_k))), j_{k+1})$ and $\Gamma'_{G,k+1}(e, (j_1, \ldots, j_{k+1})) = \mu(\Delta(\Gamma'_{G,k}(e, (j_1, \ldots, j_k))), j_{k+1})$.

$\Gamma_{G,k}(e, (j_1, \ldots, j_k))$ is called an *exploration walk* starting from the edge e and using *exploration sequence* $(j_1, \ldots, j_k)$. $\Gamma'_{G,k}$ is called *the reverse exploration walk.* Let $e_l = (v_l, i_l) = \Gamma_{G,l}(e, (j_1, \ldots, j_l))$, for $0 \le l \le k$. v_l and e_l are correspondingly the l-th vertex and the l-th edge visited by the exploration walk.

Exploration sequences were introduced in [K01]. The fact that they are reversible, more precisely that $\Gamma'_{G,k}(\Gamma_{G,k}(e, (j_1, \ldots, j_k)), (-j_k, \ldots, -j_1)) = e$, was noticed by Koucký and is what makes them so important to us. The only fact about exploration sequences that we use is the following proposition for exploration walks on trees.

Proposition 1 ([K01]). *Let $\mathbf{1}_k$ be the all-ones sequence of length k. Let $e = (v, i) \in E$, $i \neq 0$, and $e_k = (v_k, i_k) = \Gamma_{G,k}(e, \mathbf{1}_k)$. If G is a tree with at most one undirected edge between any two vertices and $k = 2(\text{size}(G) - 1)$, then $e_k = e$ and every edge of G which is not a self-loop appears exactly once in $e_0, \ldots, e_{k-1}$. Furthermore $2(\text{size}(G) - 1)$ is the smallest k such that v appears exactly $\delta(v) + 1$ times in $v_0, \ldots, v_k$.*

In what follows we reserve exploration walk to mean exploration walk with the all-ones sequence and a reverse exploration walk to mean a reverse exploration walk with the all-minus-ones sequence. We also reserve Γ and Γ' only for such walks, i.e. $\Gamma_{G,k}(v, i) = \Gamma_{G,k}((v, i), \mathbf{1}_k)$ and $\Gamma'_{G,k}(v, i) = \Gamma'_{G,k}((v, i), -\mathbf{1}_k)$

3.2 Operations on graphs

3.2.1 Configuration

A configuration is the state of the sequential algorithm described in section 2. Formally

Definition 4. 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 $n \in \mathbb{N}$, and $\delta(v) = 0$, for $v \in D$. A, I, and D form a partition of V. $H : V \to \mathbb{N}$ and $R : V \to V$ are such that $H(v) \leq \delta(v)$ and if $R(v) \neq v$, then $v \in D$. Furthermore H and R do not have non-trivial cycles in the following sense. Let $v_1, \ldots, v_k \in V$, $k \geq 2$. Then 1) if $v_{i+1} = \eta(v_i, H(v_i))$ and $v_1 = \eta(v_k, H(v_k))$, then $H(v_1) = 0$, and 2) if $v_{i+1} = R(v_i)$ and $v_1 = R(v_k)$, then $R(v_1) = v_1$.

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 *representative* of u.

A configuration $\mathcal{C}$ is *correct*, if there is at most one undirected edge between any two active vertices, i.e. if $u, v \in A$, then $|\{i : \eta(v, i) = u\}| \leq 1$.

Define $\text{rep}_R(v)$ to be v, if $R(v) = v$, and $\text{rep}_R(v) = \text{rep}_R(R(v))$. By 2) of Definition 4, this definition is correct.

Definition 5. 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 $0 < i_1 < \cdots < i_k$ be such that $\{i_1, \ldots, i_k\} = \{i : (v, i) = \mu(u, H(u))$, for some $u \in V\}$. Let ε be 1, if $H(v) \neq 0$, and 0, otherwise. 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, $\deg(T)$, 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 1) of Definition 4.

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 one hooking step.

Being elements of $\mathbb{N}$, the elements of V are ordered. Define the linear ordering $<_d$ on V so that $u <_d v$ iff $\delta(u) < \delta(v)$ or $\delta(u) = \delta(v)$ and $u < v$.

The result of the hooking operation $\text{Hook}(\mathcal{C})$ is the configuration $\mathcal{C}' = \langle G, A, I, D, H', R\rangle$ defined in the following way. If v is inactive, then $H'(v) = H(v)$. If v is active, let $v_1, \ldots, v_{\delta(v)}$ 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 fact that $\text{Hook}(\mathcal{C})$ is a configuration is proven as in [CL93]. Furthermore, by [CL93], if T is a hooking tree in $\text{Hook}(\mathcal{C})$ composed entirely of active vertices, then $\text{size}(T) \leq \deg^2(T)$.

3.2.3 Contraction

We will define Contract$(\mathcal{C}, d)$ so that, if $\mathcal{C}$ describes the state of the sequential algorithm, then Contract$(\mathcal{C}, d)$ is its state after one contraction step.

Let $d \in \mathbb{N}$. A hooking tree T in $\mathcal{C}$ is called *d-contractable*, if $\deg(T) \le d$.

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

First define $A'' = \{v : v \notin D \text{ and } \deg(T_v) \le d \text{ and } \mathrm{root}(T_v) = v\}$ and $D'' = \{v : v \in D \text{ or } \deg(T_v) \le d \text{ and } \mathrm{root}(T_v) \neq v\}$.

Now define $I' = \{v : v \notin D \text{ and } \deg(T_v) > d\}$; $H'(v)$ is $H(v)$, if $v \in I'$, and 0, otherwise; $R'(v)$ is $R(v)$, if $v \in D$, $\mathrm{root}(T_v)$, if $v \in D'' - D$, and v, otherwise.

Let T be a hooking tree in $\mathcal{C}$ and $s = \mathrm{size}(T)$. Let $v_1, \ldots, v_s$ be the enumeration of the vertices of T visited by the exploration walk starting from $(\mathrm{root}(T), 1)$, where we enumerate a vertex only the first time it is visited by the exploration walk. Let $e_1, \ldots, e_k$ be the enumeration of the edges of G incident to the vertices of T defined in the following way – enumerate 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 $l_u \in \mathbb{N}$ and the following enumeration of edges $e_{u,j}$, $1 \le j \le l_u$. First consider the enumeration $e_1, \ldots, e_k$ of the edges of T_u from the previous paragraph. Remove from this enumeration all the 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 $e_{i_1}, \ldots, e_{i_h}$ are all the edges in the sequence $e_1, \ldots, e_k$ such that $\eta(e_{i_j}) \in V(T)$, leave only e_{i_1}. Let l_u be the number of remaining edges in the enumeration and $e_{u,j}$, $1 \le j \le l_u$, be their enumeration. Naturally we call the edges $e_{u,j}$, the *remaining edges of* T_u.

Define $A' = A'' - \{v \in A'' : l_v = 0\}$, $D' = D'' \cup \{v \in A'' : l_v = 0\}$; $\delta'(v)$ is l_v, if $v \in A'$, $\delta(v)$, if $v \in I'$, and 0, if $v \in D'$.

We are left now 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 $w = \mathrm{root}(T_u)$ 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 $w = \mathrm{root}(T_u)$ and k is the only index such that $e_{w,k} = (u, j)$.

From the definition of $\mathcal{C}' = $ Contract$(\mathcal{C}, d)$ follows that $\mathcal{C}'$ is a correct configuration such that $\delta'(v) \le d$, for $v \in A'$, and $\deg(T_v) > d$, for $v \in I'$. Furthermore in the hooking forest F' every $v \in A' \cup D'$ is in a hooking tree which contains only v.

3.3 Sequence of configurations

For some $r \in \mathbb{N}$, we will define a sequence of configurations $\mathcal{C}_{r,l}$, $0 \le l \le r$. In the following we omit r from the subscripts of elements of the sequence.

First define recursively a rooted tree C_k (here a tree is used in the usual sense) of depth k whose leaves are labeled. The root and only leaf of C_0 is labeled with $(1, 0)$. The root of C_k, $k \ge 1$, has four descendants. From left to right they are: first a leaf labeled with $(0, k)$, second C_{k-1}, third C_{k-1}, and finally a leaf labeled with $(1, k)$. By induction it can be proven easily that C_k has $r(k) = 3 \cdot 2^k - 2$ leaves. Number starting from 1 the leaves of C_k as they appear in its left to right traversal. For $0 \le l \le r(k)$, let $\sigma(k, l)$ be $(0, 0)$, if $l = 0$, and the label of the l-th leaf of C_k, otherwise. Let $\sigma_{1,k}(l)$ and $\sigma_{2,k}(l)$ be correspondingly the first and the second component of $\sigma(k,l)$. In the following we omit k from the subscript of $\sigma_{1,k}$ and $\sigma_{2,k}$.

Let $\mathcal{C}_0$ be some configuration. Assume that we have already defined $\mathcal{C}_0,\ldots,\mathcal{C}_{l-1}$ for $1 \le l \le r(k)$. Let $\mathcal{C}'_l$ be $\mathrm{Hook}(\mathcal{C}_{l-1})$, if $\sigma_1(l)\sigma_2(l) = 0$, and $\mathcal{C}_{l-1}$, otherwise. Let $\mathcal{C}_l = \mathrm{Contract}(\mathcal{C}'_l, 2^{2^{\sigma_2(l)+\sigma_1(l)}})$.

Let C'_k be the labeled tree, which is obtained from C_k by substituting the label of the l-th leaf with $(\texttt{Hook}; \texttt{Contract}(2^{2^{\sigma_2(l)+\sigma_1(l)}}))$, if $\sigma_1(l)\sigma_2(l) = 0$, and $\texttt{Contract}(2^{2^{\sigma_2(l)+1}})$ otherwise. The structure of C'_k follows the structure of the recursive calls of $\texttt{Connect}(k)$ – the levels of C'_k are the levels of recursion of $\texttt{Connect}(k)$. Thus the configurations in the sequence defined in the previous paragraph are exactly the states of the sequential algorithm after consecutive operations, where $\mathcal{C}_0$ is its state initialized according to the input graph.

Define a configuration $\mathcal{C}_l$ to be *nice*, if it is correct and 1) $\mathrm{size}(T_v) > 2^{2^{h+1}}$ and $\deg(T_v) > 2^{2^{h+2}}$, for $v \in I_l$, and 2) $\delta_l(v) \le 2^{2^{h+2}}$, for $v \in A_l$, where h is k, if $l = 0$, and $\sigma_2(l) - 1 + \sigma_1(l)$, otherwise. By definition $\mathcal{C}_l$ is nice iff the state described by it fulfills the preconditions given in [CL93] for executing $\texttt{Connect}(h)$.

Considering the correspondence between the sequence $\mathcal{C}_0,\ldots,\mathcal{C}_{r(k)}$ and the $\texttt{Connect}(k)$ procedure of the sequential algorithm, the following theorem is a consequence of the results in [CL93].

Theorem 1. *If $\mathcal{C}_0$ is a nice configuration, then $\mathcal{C}_l$ is nice, for every $1 \le l \le r(k)$, $|A_{r(k)}| \le \max\{|A_0|/2^{2^k}, 1\}$, and $\mathrm{size}(T_v) > 2^{2^{k+1}}$, for $v \in I_{r(k)}$.*

Finally, again by [CL93], 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 1. *Let G be a graph with at most one undirected edge between any two vertices and $V = [n]$. Let $r = 3 \cdot 2^{\lceil \log^{(2)} n \rceil} - 2$ and $\mathcal{C}_0 = \langle G, A, I, D, H, R \rangle$, where $A = \{v : \delta(v) \ne 0\}$, $I = \emptyset$, $H(v) = 0$ and $R(v) = v$, for $v \in V$. Then u and v are connected in G iff $\mathrm{rep}_{R_r}(u) = \mathrm{rep}_{R_r}(v)$.*

4 Space efficient algorithm

4.1 An $O(\log^2 n)$ algorithm

Let G be a graph with $V = [n]$ and with at most one undirected edge between any two vertices. Let $r = 3 \cdot 2^{\lceil \log^{(2)} n \rceil} - 2$. Consider the sequence of configurations $\mathcal{C}_0,\ldots,\mathcal{C}_r$ from Corollary 1. The starting point for a space efficient algorithm comes directly from the definition of this sequence. More precisely we define functions $\texttt{Active}(l,v)$, $\texttt{Inactive}(l,v)$, $\texttt{Done}(l,v)$, $\texttt{Degree}(l,v)$, $\texttt{Neighbor}(l,v,i)$, $\texttt{BackLabel}(l,v,i)$, $\texttt{Hook}(l,v)$, and $\texttt{Rep}(l,v)$, where $0 \le l \le r$, v is a vertex, and i is the label of an edge incident to v, which return the corresponding component of $\mathcal{C}_l$.

Call the value of the parameter l, the *level of recursion*. Thus the levels of recursion of our algorithm correspond to the elements of the sequence $\mathcal{C}_0,\ldots,\mathcal{C}_r$. For $l = 0$ all of these functions just use the input graph G (this is the bottom of the recursion), and their output for level $l+1$ is determined from outputs for level l according to the definitions in section 3.3. Using these functions, to solve undirected s,t-connectivity we apply Corollary 1.

Consider the sequence $\mathcal{C}'_1,\ldots,\mathcal{C}'_{r-1}$ corresponding to $\mathcal{C}_0,\ldots,\mathcal{C}_r$ as defined in section 3.3. To obtain $\mathcal{C}'_l$ from $\mathcal{C}_{l-1}$ we define $\texttt{NewHook}(l,v)$, which returns $H'_l(v)$ as defined in section 3.2.2. $\mathcal{C}_l$ is obtained from $\mathcal{C}'_l$ by contracting some of the hooking trees in $\mathcal{C}'_l$ as defined in section 3.2.3. For a hooking tree T in $\mathcal{C}'_l$, by the definition of contraction from section 3.2.3, 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 `TreeSize`(l, v) and `TreeWalk`(l, v, i). Let T be the hooking tree in $\mathcal{C}'_l$ containing v. If s is the size of T, `TreeSize`(l, v) returns $2(s-1)$. Notice that $2(s-1)$ is the length of the exploration walk on T given in Proposition 1. `TreeWalk`(l, v, i) returns $\Gamma_{T,i}(v, 1)$.

The definition of $\mathcal{C}_l$ from $\mathcal{C}_{l-1}$ given in section 3.3 depends on two parameters – whether we perform hooking and a contraction parameter. Those two parameters are determined by $(\varepsilon, k) = \sigma(\lceil \log^{(2)} n \rceil, l)$ as given in section 3.3. We define functions `Sigma`(l), `ContractOnly`(l) and `ContractDegree`(l) which return correspondingly $2k + \varepsilon$, whether we perform hooking, and what is the contraction parameter.

The following claim can be proven by 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 takes $O(\log n)$ space.

Claim 1. *The functions* `Active`(l, v)*,* `Inactive`(l, v)*,* `Done`(l, v)*,* `Degree`(l, v)*,* `Neighbor`(l, v, i)*,* `BackLabel`(l, v, i)*,* `Hook`(l, v)*, and* `Rep`(l, v)*, correctly return the corresponding components of* $\mathcal{C}_l$*. The total space taken by the execution of each of these functions is* $O(l \log n)$*.*

4.2 The $O(\log n \log^{(2)} n)$ algorithm

The precise definition of all functions mentioned in this section and the next are given in the appendix.

The $O(\log n)$ space per level in Claim 1 comes mainly from having to store vertices in the local variables of the functions, since each vertex takes $\Theta(\log n)$ space. To take care of this bottleneck we define the functions so that they never have to keep 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 `currvertex` 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 some of our functions. Instead of `Neighbor`(l, v, i), we have `Neighbor`(l, i), which moves the current vertex to its i-th neighbor in $\mathcal{C}_l$. Let T be the hooking tree of cv in $\mathcal{C}'_l$. Instead of `TreeWalk`(l, v, i) we have `TreeForward`(l, i), which returns j and moves the current vertex to u, where $(u, j) = \Gamma_{T,i}(cv, 1)$. Similarly we have `TreeBack`$(l, i, j)$ which moves the current vertex to the end vertex of $\Gamma'_{T,i}(cv, j)$. Finally instead of `Rep` we have `Root` as described in Lemma 1 below.

The most important part of our idea to avoid storing vertices 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 `Move`(l, i) to return $\beta_{l-1}(cv, i)$ and move the current vertex to $\eta_{l-1}(cv, i)$. Let `Current` be the empty function (we just use it as the path description, defined later, of the current vertex). Call `Move`(l, i), `TreeForward`(l, i) and `Current` *forward moves.* For a forward move M, let `Reverse`(M, j) be its reverse, i.e. it is correspondingly `Move`(l, j), `TreeBack`(l, i, j), and `Current`, where j is the result of M. It is here that the reversibility of exploration walks comes into play, because it allows us to use `TreeBack` as the reverse of `TreeForward`.

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 reverse of the moves in `P` to restore the current vertex.

A simple example of the use of `after` is the 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 three 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 $\Theta(\log^{(2)} n)$ space necessary to store the index of a bit. This way the bottleneck of $\Theta(\log n)$ space is reduced to $\Omega(\log^{(2)} n)$.

For example, we use the `=` operator in the definition of `TreeSize` in the following way. Using Proposition 1, we can make steps from the exploration walk on a hooking tree T until we go back to the starting vertex v sufficiently many times. For this we have to store v, so that we can compare it with each new vertex of the walk. This takes $\Theta(\log n)$ space, independent of the degree of T. Instead, we incrementally find i with properties as in Proposition 1, where to check if the i-th vertex of the walk is equal to v, we keep the current vertex at v and use the `=` operator, as defined above, to compare it to the vertex with path description $\texttt{TreeForward}(i)$. Thus, if T has degree d, we can find its size in space $O(\max\{\log d, \log^{(2)} n\})$.

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 $v(l) = 2 \cdot 2^{2^{k+2}}$, where $k = \sigma_2(\lceil \log^{(2)} n \rceil, l)$. 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 only degrees of vertices, indices of neighbors, back-labels of edges, and lengths of exploration walks on hooking trees. For example, the information stored for reversing a sequence of forward moves are back-labels (for `Move`) and tree-edge labels (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 1. *1) 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. 2) All vertices which are active in $\mathcal{C}_{l-1}$ are valid for level l. 3) If a hooking tree T in $\mathcal{C}'_l$ is contractable, then all of its vertices are valid for level l.*

The first item of the observation is trivial, the second follows from Theorem 1, because all $\mathcal{C}_l$ are nice, and the third follows because $\texttt{Valid}(l) > \texttt{ContractDegree}(l)$.

One consequence of having an upper bound on the numerical values for a level is that we might not be able to process the result of a function, if it is invalid for the level requesting it. Actually, as can be seen from Lemma 1, some functions are specified to return `null` (a special constant different from all numerical values), if their result is invalid for the level requesting it. The only information we can derive from an invalid or `null` result is that either the current vertex or a neighbor of the current vertex is invalid, or that the current vertex is part of an uncontractable tree. This information is enough to define the functions as in Lemma 1. 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, vertices which are either invalid or part of an uncontractable tree are inactive and thus, by definition, inherit their properties 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.

Our goal has become to prove the following lemma. Let T be the hooking tree of cv in $\mathcal{C}'_l$. Let (cv, i) be an edge of T and $v = \eta_T(cv, i)$. We call a move along (cv, i) *possible for level* l, if v is valid for level l. T is *contractable for level* l, if $\deg(T) \leq d$, where d is the contraction parameter for level l.

Lemma 1. *1.* `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*.* `NewHook`(l) *returns* $H'_l(cv)$*, for* $l \geq 1$*.*

2. If T *is uncontractable or* $cv \in D_{l-1}$*, then* `Root`(l) *returns 0, otherwise it returns the index of the first occurrence of* $\mathrm{root}(T)$ *in the exploration walk on* T *starting from* $(cv, 1)$*.*

`TreeSize`(l) *returns* $2(\mathtt{size}(T) - 1)$*, if* T *is contractable for level* l*, and* `null` *otherwise.*

Assume that cv *is valid for level* l*. If all moves of* $\Gamma_{T,i}(cv, 1)$ *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* $\Gamma'_{T,i}(cv, j)$ *are possible for level* l *and it ends in* v*, then* `TreeBack`(l, i, j) *moves the current vertex to* v*.*

3. 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, and* `null` *otherwise.*
All local variables are valid.

The proof of the lemma is done by induction on the level of recursion. The base case could be easily verified from the definitions of the functions and the burden of the proof lies in making the inductive steps. 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`. This fact can be seen from the definitions of the functions.

4.3 Solving undirected s, t-connectivity

Using Lemma 1 we can prove the main theorem of this paper.

Theorem 2. *Undirected* s, t*-connectivity on a graph with* n *vertices can be solved in space* $O(\log n \log^{(2)} n)$*.*

By Corollary 1, s and t are connected iff $\mathrm{rep}_{R_r}(s) = \mathrm{rep}_{R_r}(t)$. Thus to solve undirected s, t-connectivity it is enough to define a function which returns $\mathrm{rep}_{R_l}(cv)$.

Let $m = \lceil \log^{(2)} n \rceil$. The space complexity of the algorithm is dominated by the space taken by the stack used in the execution of the pseudo-code. From Lemma 1 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 $O(\max\{\log v(l), m\})$ (the additional $O(m)$ space appears because of comparison of vertices). Since $\log v(l) = O(2^{k(l)})$, $1 \le l \le r$, where $k(l) = \sigma_2(m, l)$, we have to prove that $\sum_{l=1}^{r} \max\{2^{k(l)}, m\} = O(\log n \log^{(2)} n)$.

Consider the recurrence given by $S(0) = m$ and $S(k+1) = 2(S(k) + \max\{2^k, m\})$. From the recursive definition of C_r 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 that $S(m) = O(\log n \log^{(2)} n)$.

5 Conclusion

We demonstrated a $O(\log n \log^{(2)} n)$ space algorithm for undirected s,t-connectivity. An interesting question is can we use similar techniques to extract an even more space efficient algorithm from [CHL99]. Unfortunately we were not able to do so, the main obstacle being that in the PRAM models comparison of $O(\log n)$ bit numbers takes constant time and we need $O(\log^{(2)} n)$ space to do the same.

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

[AKL+79] R. Aleliunas, R. Karp, R. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, 1979, 218–223.

[AS83] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for ultracomputer and PRAM. *Proceedings International Conference on Parallel Processing*, 1983, 175–179.

[ATSWZ97] R. Armoni, A. Ta-Shma, A. Wigderson, and S. Zhou. $SL \subseteq L^{\frac{4}{3}}$. *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, 1997, 230–239.

[CHL99] K. Chong, Y. Han, and T. Lam. On the parallel time complexity of undirected connectivity and minimum spanning trees. *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999, 17–19. Journal version in *J. ACM* 48(2), 2001, 297–323.

[CL93] K. Chong and T. Lam. Finding connected components in $O(\log n \log^{(2)} n)$ time on the EREW PRAM. *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993, 11–20. Journal version in *J. Algorithms* 18(3), 1995, 378–402.

[CLC82] F. Chin, J. Lam, and I-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM* 25, 1982, 659–666.

[CV86] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986, 478–491.

[G86] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986, 492–501.

[HCS79] D. Hirschberg, A. Chandra, and D. Sarwate. Computing connected components on parallel computers. *Communications of the ACM* 22(8), 1979, 461–464.

[HZ96] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems. *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1996, 438–447.

[JM91] D. Johnson and P. Metaxas. Connected components in $O(\log^{3/2}|V|)$ parallel time for the CREW PRAM. *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, 1991, 688–697.

[K01] M. Koucký. Universal traversal sequences with backtracking. *Proceedings of the 16th Annual IEEE Conference on Computational Complexity*, 2001, 21–27.

[KNP92] D. Karger, N. Nisan, and M. Parnas. Fast connected components algorithm for the EREW PRAM. *Proceedings of the 4th ACM Symposium on Parallel Algorithms and Architectures*, 1992, 373–381.

[LP82] H. Lewis and C. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science* 19, 1982, 161-187.

[N90] N. Nisan. Psuedorandom generators for space-bounded computation. *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, 1990, 204–212.

[NSW92] N. Nisan, E. Szemeredi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, 1992, 24–29.

[PR99] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *Proceedings of 3rd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM) and 2nd International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*, 1999, 233-244. Journal version in *SIAM J. Comput.* 31(6), 2002, 1879–1895.

[R85] J. Reif. Depth-first search is inherently sequential. *Information Processing Letters* 20(5), 1985, 229–234.

[R04] O. Reingold. Undirected ST-Connectivity in Log-Space. Electronic Colloquium on Computational Complexity Report TR04-094, http://www.eccc.uni-trier.de/eccc/.

[S70] W. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences* 4(2), 1970, 177–192.

[SJ81] C. Savage and J. JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM Journal on Computing* 10(4), 1981, 682–691.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms* 3(1), 1982, 57-67.

[T72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1(2), 1972, 146–160.

[TV85] R. Tarjan and U. Vishkin. An Efficient Parallel Biconnectivity Algorithm. *SIAM Journal on Computing* 14(4), 1985, 862–874.

6 Appendix: Pseudo-code and Turing Machine

Since giving directly the Turing Machine which solves the problem is rather cumbersome, we define our algorithm using a pseudo-code and then translated it to a Turing Machine. The language of the pseudo-code is similar to Pascal, except that the blocks are marked by indentation rather than `begin` and `end`. We use fixed width font to denote the names of functions and variables from the pseudo-code, e.g. `FooBar` and `i`, and a roman font for mathematical functions and variables, e.g. FooBar and i. In this section we give the pseudo-code, discuss its execution and how to translate it to a Turing Machine.

6.1 Execution of the pseudo-code

The variable usage and execution of the pseudo-code are similar to a program written in Pascal. 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.* 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. During its execution, a function can call other functions either on the same level of recursion or on a lower level of recursion. 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.

6.2 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 `arg1F`, `arg2F`, 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 `null`.

The method which uses arrays is more subtle. Let `F` be a function which uses this method of passing arguments and returning values. We have two arrays – one, `argF`, for passing arguments to `F` and one, `retF`, 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.

Let `H` calls `F`. If `H` uses the value returned by `F`, then before the call to `F` the entry of `retF` for the current level of recursion is marked, otherwise it is left unmarked. When `F` produces a result, it finds in `retF` the first marked entry after the entry for the current level of recursion and tries to store its result there. If the value produced is too large for the corresponding entry, `F` writes `null`. After the call to `F` returns, `H` unmarks the entry of `retF` for the current level of recursion.

Similarly, if `H` provides arguments to `F`, then before the call to `F`, `H` marks the entry of `argF` for the current level of recursion and provides values for it. When `F` wants to access its arguments it looks up, starting from the current level, and finds the first entry of `argF` which is marked and uses the values stored in the entry as its arguments. After the call to `F` returns, `H` unmarks the entry of `argF` for the current level of recursion. In the code we denote with `argF` the argument to `F`, with the agreement that this is a call to a global function returning the argument rather than access to a global variable.

This method can be used, if `F` is always called on the previous level of recursion, i.e. all calls to `F` are $\mathtt{F}(l-1,\ldots)$, because then there is no danger of overwriting its arguments or returned value.

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 $\mathtt{F}(x_1,\ldots,x_k)$ to call a function `F` with arguments $x_1,\ldots,x_k$. 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`.

6.3 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 $O(\max\{\log v(l), \log^{(2)} n\})$. The $O(\log^{(2)} n)$ appears because of comparison of vertices, as explained in section 4.2 and the definitions of the comparison operators in section 6.4.2.

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.

6.4 Pseudo-code

6.4.1 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 $3 \cdot 2^{\lceil \log^{(2)} n \rceil} - 1$ 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 can change it. We denote the current level of recursion with cl.

In the following, T denotes the hooking tree of the current vertex in $\mathcal{C}'_{cl}$. A hooking tree in $\mathcal{C}'_{cl}$ is called contractable, if it is d-contractable, where d is the contraction parameter for level cl. A value is valid, if it is valid for level cl.

We use the following observation. It follows from Observation 1 and the correctness of the functions mentioned in it.

Observation 2. *1) If* `TreeSize` $\neq$ `null`*, then all moves of* `TreeForward`(i) *are possible, for* $i \geq 0$. *2) If* `MoveValid`(i) *is true, then the result of* `Move`(i) *is not* `null` *and valid, otherwise* $\eta_{cl-1}(cv, i)$ *is invalid and in* I_{cl-1}.

By this observation `TreeSize` and `MoveValid` serve as "safeguard" checks for forward moves. Thus before using a sequence of forward moves 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 all the forward moves of the sequence return valid results.

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

Every function is preceded by paragraphs which give its specification and describe its local variables. 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.

6.4.2 Important functions

`Sigma` is a global function, which takes one argument l and returns $2k + \varepsilon$, where $(\varepsilon, k) = \sigma(\lceil \log^{(2)} n \rceil, l)$.

global function `Sigma`
    $\mathtt{k} := \lceil \log^{(2)} \mathtt{n} \rceil$;
    while `true`
        if $\mathtt{k} = 0$ **then return** 1;
        else
            if $\mathtt{l} = 1$ **then return** $2 \cdot \mathtt{k}$;
            else
                if $\mathtt{l} = 3 \cdot 2^{\mathtt{k}} - 2$ **then return** $2 \cdot \mathtt{k} + 1$;
                else
                    if $\mathtt{l} \geq 3 \cdot 2^{\mathtt{k}-1}$ **then** $\mathtt{l} := \mathtt{l} - 3 \cdot 2^{\mathtt{k}-1} + 1$;
                    $\mathtt{k} := \mathtt{k} - 1$;

`Valid`, `ContractDegree`, and `ContractOnly` are global functions which do not take arguments and return correspondingly $v(cl)$, what is the contract parameter for level cl, and whether we perform hooking to define $\mathcal{C}'_{cl}$ (see section 3.3).

global function `Valid`
    return $2^{1+2^{2+\mathtt{Sigma(level)\ div\ 2}}}$;

global function `ContractDegree`
    return $2^{2^{\mathtt{Sigma(level)\ div\ 2\ +\ Sigma(level)\ mod\ 2}}}$;

global function `ContractOnly`
    return (`Sigma(level) div 2`) (`Sigma(level) mod 2`) $\neq 0$;

Let $\mathtt{M}_1, \ldots, \mathtt{M}_k$ be a path description relative to the current vertex (see section 4.2). **after** $\mathtt{M}_1, \ldots, \mathtt{M}_k$ **do** `B` changes the current vertex according to the moves in $\mathtt{M}_1, \ldots, \mathtt{M}_k$, executes `B`, and finally restores the original current vertex. Its local variables are $\mathtt{l}_1, \ldots, \mathtt{l}_k$. A function using `after` must make sure that the values of $\mathtt{l}_1, \ldots, \mathtt{l}_k$ are valid using "safeguard" checks as explained in section 6.4.1.

define after $\mathtt{M}_1$, $\mathtt{M}_2$, ..., $\mathtt{M}_k$ **do** `B`
    $\mathtt{l}_1 := \mathtt{M}_1$; $\mathtt{l}_2 := \mathtt{M}_2$; ...; $\mathtt{l}_k := \mathtt{M}_k$;
    `B`;
    Reverse($\mathtt{M}_k$, $\mathtt{l}_k$); ...; Reverse($\mathtt{M}_2$, $\mathtt{l}_2$); Reverse($\mathtt{M}_1$, $\mathtt{l}_1$);

The operators `<`, `=`, and $<_d$ take as arguments two path descriptions relative to the current vertex $\mathtt{P}_1$ and $\mathtt{P}_2$. Let v_1 and v_2 be the end vertex of $\mathtt{P}_1$ and $\mathtt{P}_2$, correspondingly. The three operators check correspondingly, whether $v_1 < v_2$, $v_1 = v_2$, and $v_1 <_d v_2$. The local variables of `<` and `=` are `i`, $\mathtt{b}_1$, and $\mathtt{b}_2$. The local variables of $\mathtt{<}_d$ are $\mathtt{d}_1$ and $\mathtt{d}_2$.

`Bit`(s,t) returns the s-th most significant bit of t. $\mathtt{b}_1$ and $\mathtt{b}_2$ are single bits, and the function using $\mathtt{<}_d$ must make sure that the values of $\mathtt{d}_1$ and $\mathtt{d}_2$ are valid using a corresponding "safeguard" check as explained in section 6.4.1. The `i` variables of `<` and `=` are the only local variable whose value is $\Theta(\log n)$. As given in Lemma 1, all other local variables are valid. Thus the space for level l is $O(\max\{\log v(l), \log^{(2)} n\})$.

```
define P1 < P2
    for i := 1 to ⌈log n⌉ do
        after P1 do b1 := Bit(i, currvertex);
        after P2 do b2 := Bit(i, currvertex);
        if b1 ≠ b2 then return b1 < b2;
    return false;

define P1 = P2
    for i := 1 to ⌈log n⌉ do
        after P1 do b1 := Bit(i, currvertex);
        after P2 do b2 := Bit(i, currvertex);
        if b1 ≠ b2 then return false;
    return true;

define P1 <_d P2
    after P1 do d1 ::= Prev(Degree);
    after P2 do d2 ::= Prev(Degree);
    return (d1 < d2) or (d1 = d2 and P1 < P2);
```

6.4.3 Status functions

`Done`, `Active`, and `Inactive` return correspondingly whether $cv \in D_{cl}$, $cv \in A_{cl}$, and $cv \in I_{cl}$. These functions do not have arguments and local variables.

```
function Done
    return (level = 0 and Degree = 0) or
        (level > 0 and (Prev(Done) or Root ≠ 0 or Degree = 0));

function Active
    return (not Done) and TreeSize ≠ null;

function Inactive
    return (not Done) and TreeSize = null;
```

6.4.4 Hooking

`NewHook` does not take arguments and returns $H'_{cl}(cv)$, $cl \geq 1$. Its local variables are $\mathtt{d}_1$, $\mathtt{d}_2$, `i`, `j`, and `m`.

`NewHook` is defined as given in section 3.2.2. In line 7 we use 1) of Observation 1 to deduce that $\eta_{cl-1}(cv, \mathtt{i}) \in I_{cl-1}$. At line 3 $cv \in A_{cl-1}$ and hence $\mathtt{d}_1$ is valid and non-`null`. So `i` and `m` are also valid. At line 10 we have that $\eta_{cl-1}(cv, \mathtt{i}), \eta_{cl-1}(cv, \mathtt{m}) \in A_{cl-1}$. At line 12 all neighbors of cv are in A_{cl-1}. Hence $\mathtt{d}_2$ and j are valid.

```
    function NewHook
1       if Prev(Inactive) then return Prev(Hook);
2       if Prev(Done) or ContractOnly then return 0;

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

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

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

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

Hook returns $H_{cl}(cv)$. It does not have arguments and local variables.

```
function Hook
    if level = 0 then return 0;
    if Inactive then return NewHook;
    return 0;
```

IsHooked takes one argument i. Let $(v, j) = \mu_{cl-1}(cv, i)$ and $h = H'_{cl}(v)$. If cv is valid or $\delta_{cl-1}(cv) \leq \delta_{cl-1}(v)$, then IsHooked is true iff $h = j$. The local variables of IsHooked are i, j, and h.

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 NewHook = 0;
2    if argIsHooked > Valid then return Prev(IsHooked(argIsHooked));
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 ::= NewHook;
10   return h = j;
```

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 1) of Observation 1. If `j` is invalid, then $v \in I_{cl-1}$, and it did not hook to cv (otherwise $cv \in I_{cl-1}$). 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 $\delta_{cl-1}(cv) \leq \delta_{cl-1}(v)$ and v valid. 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 $cl \geq 2$, $\delta_{cl-2}(cv) = \delta_{cl-1}(cv)$, $\delta_{cl-2}(v) = \delta_{cl-1}(v)$, $(v, \mathtt{j}) = \mu_{cl-2}(cv, \mathtt{i})$, and $\mathtt{h} = H'_{cl-1}(v)$. Thus the correctness in this case is ensured by the inductive hypothesis.

6.4.5 Exploration walk

The functions in this section come from the definition of a hooking forest of a configuration given in section 3.2.1. T is the hooking tree of cv in $\mathcal{C}'_{cl}$.

`TreeDegree` does not take arguments. If cv is valid for level l, it returns $\delta_T(cv)$, otherwise it returns `null`. Its local variables are `i`, `d`, and `td`.

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 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
    if NewHook ≠ 0 then td := td + 1;
    return td
```

TreeMove takes one argument i. Let $(v, j) = \mu_T(cv, i)$. Assume that cv is valid. If a move along (cv, i) is possible, i.e. v is valid, TreeMove returns j and moves the current vertex to v, otherwise it does not change the current vertex and returns null. The local variables of TreeMove are i, j, k, l, d, $\mathtt{d}_1$, and r.

Lines 2-7 convert from the label i of a tree-edge e 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 hoked 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). $\mathtt{d}_1$ is valid because of the assignment in line 9. k and r are valid because at line 14 v is valid.

```
function TreeMove
        i := argTreeMove;
1       if i = 0 then return 0;

        // convert from tree-edge label to graph-edge label
2       l := i;
3       d ::= Prev(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 ::= NewHook;

        // if the new vertex is invalid return null
8       if not MoveValid(j) then return null;
9       after Move(j) do d1 ::= Prev(Degree);
10      if d1 = null then return null;

11      if i < TreeDegree or (i = TreeDegree and NewHook = 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 d1 do
16          after Move(j) do fl := IsHooked(k);
17          if not fl then continue;
```

```
        // enumerate all the edges with which neighbors
        // of the new vertex hooked to it

        after Move(j) do fl := MoveValid(k);
        if not fl then
            // the k-th neighbor of the new vertex is invalid
            r := r + 1;
            continue;

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

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

`TreeForwardStep` takes one argument i. Let $(v, j) = \Gamma_{T,1}(cv, i)$. Assume that cv is valid. 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. The local variable of `TreeForwardStep` is `j`.

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

`TreeForward` takes one argument i. Assume that cv and i are valid. If all moves of $\Gamma_{T,i}(v, 1)$ are possible and it ends in (v, j), then `TreeForward` returns j and moves the current vertex to v. The local variables of `TreeForward` are `i`, `j`, and `k`.

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

`TreeBack` takes two arguments i and j. Assume that cv and i are valid. If all moves of $\Gamma'_{T,i}(v, j)$ are possible and it ends in v, then `TreeBack` does not return anything and moves the current vertex to v. `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);
```

`TreeSize` does not take arguments. It returns $2(\texttt{size}(T) - 1)$, if T is contractable, and `null`, otherwise. Its local variables are `i`, $\texttt{i}_1$, $\texttt{k}_1$, $\texttt{k}_2$, `d`, and `td`.

$2(\texttt{size}(T) - 1)$ is the length of the exploration walk given in Proposition 1. 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 makes 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 ($\texttt{Valid} > \texttt{ContractDegree}$). 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 $s \leq \deg(T)$ (at line 6, T has at least one edge) and on the other, by Proposition 1, exploration walk of length $2(s-1)$ visits all vertices of a tree of size s and returns to the starting vertex sufficiently many times. Since $\texttt{Valid} \geq 2\ \texttt{ContractDegree}$, if the length of the exploration walk becomes bigger than `Valid`, then $s > \texttt{ContractDegree}$, so $\deg(T) > \texttt{ContractDegree}$ and T is uncontractable.

`i` and $\texttt{i}_1$ are valid because of the condition of the loop in line 6. $\texttt{k}_1$ is valid because of the assumption that all vertices visited by the exploration walk of length $\texttt{i} - 1$ in line 7 are valid. $\texttt{k}_2$ 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 $\texttt{d}_1$ are at most `ContractDegree`, and since $\texttt{Valid} \geq 2\ \texttt{ContractDegree}$, 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
```

```
        k1 := TreeForward(i-1);
        k2 := TreeForwardStep(k1);
        if k2 = null then
            // if we cannot then the tree is uncontractable
            TreeBack(i-1, k1);
            return null;
        TreeBack(i, k2);

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

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

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

Root does not take arguments. If T is uncontractable or $cv \in D_{cl-1}$, then Root returns 0, otherwise it returns the index of the first occurrence of root(T) in the exploration walk on T starting from $(cv, 1)$. The local variables of Root are d and i.

According to the definition of 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 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 vertex in it which hooked to itself
3     for i := 0 to d-1 do
4         after TreeForward(i) do fl := (NewHook = 0);
5         if fl then return i;
```

6.4.6 Contraction

The definitions of the functions in this section come from the definition of the contraction operation given in section 3.2.3. T is the hooking tree of cv in $\mathcal{C}'_{cl}$.

`IsEdge` takes two arguments i and j. Assume that $cv = \mathrm{root}(T)$ and T is contractable. If v is the end vertex of $\Gamma_{T,i}(cv, 1)$, then `IsEdge` returns true iff the (v, j) is a remaining edge of T (see the definition in section 3.2.3). The local variables of `IsEdge` are `i`, `j`, $\mathtt{j}_1$, `k`, $\mathtt{k}_1$, `d`, $\mathtt{d}_1$, and $\mathtt{d}_2$.

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 $\mathcal{C}'_{cl}$. 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 $(u, \mathtt{j}_1)$ is an edge before e in the enumeration of the remaining edges of T given in section 3.2.3. Let $u' = \eta_{cl-1}(u, \mathtt{j}_1)$. Lines 14-16 check whether u' is in T'. In line 13 we use that, if the hooking tree of u' in $\mathcal{C}'_{cl}$ 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. $\mathtt{d}_2$ and $\mathtt{k}_1$ are valid, because at line 6 T' is contractable. Lines 9 and 10 ensure the validity of $\mathtt{d}_1$ and $\mathtt{j}_1$.

```
    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;
```

Let `P` be a path description relative to the current vertex, v be the vertex with path description `P`, and T' is its hooking tree in $\mathcal{C}'_{cl}$. Assume that T' is contractable and $v = \mathrm{root}(T')$. Let `B` be some instruction(s) which might depend on the variables `i` and `j`. **after** `P` **for every edge** `(i, j)` **do** `B` executes `B` for all possible values of `(i, j)` such that $(u, \mathtt{j})$ is a remaining edge of T', where u is the end vertex of $\Gamma_{T,\mathtt{i}}(v, 1)$. The local variables are $\mathtt{i}_1$, $\mathtt{d}_1$, $\mathtt{d}_2$. `i` and `j` are local to the function using **after** `P` **for every edge** `(i, j)` **do** `B`.

Lines 3-5 check, if this is the first time the exploration walk on T' 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.

```
define 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 fl := IsEdge(i, j);
 9            if not fl then continue;

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

`Degree` does not take arguments and returns $\delta_{cl}(cv)$. Its local variables are `i`, `j`, and `td`.

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 TreeSize = null then return Prev(Degree);
 3    if Prev(Done) or Root ≠ 0 then return 0;

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

`Neighbor` takes one argument i. It does not return anything, but moves the current vertex to $\eta_{cl}(cv, i)$. Its variables are `i`, `j`, `l`, and `d`.

The definition of `Neighbor` follows the definitions in section 3.2.3. First we make sure that T is contractable (lines 3 and 9). 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 13-19). 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 5, 11, and 18).

`l` is valid because at line 7 `argNeighbor` is valid, and the other local variables are valid because at line 13 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);
       // handle the self-loop case
2      if argNeighbor = 0 then return;

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

8      d := TreeSize;
9      if d = null then
           // T is uncontractable
10         Prev(Neighbor(l));
11         if TreeSize ≠ null then TreeForward(Root);
12         return;

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

           // move to e and then along e
16         TreeForward(i);
17         Prev(Neighbor(j));
           // move to the root of T'
18         if TreeSize ≠ null then TreeForward(Root);
19         return;
```

`BackLabel` takes one argument i and returns $\beta_{cl}(cv, i)$. `BackLabel` uses the array method described in section 6.1.2 of taking arguments and returning values. Its local variables are `i`, `j`, $\mathtt{j}_1$, `k`, $\mathtt{k}_1$, `l`, `d`, `nd`, and `r`.

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 11-13). Let $v = \eta_{cl-1}(e)$ and T' be the hooking tree of v in $\mathcal{C}'_{cl}$. If T' is uncontractable, then we call recursively, because in this case the back-label comes from the previous level of recursion (line 21). Otherwise we have to find the index `nd` of the first remaining edge e' of T' which goes from T' to T (lines 23-31). This is the new back-label. To find the index of e', first we find the root of T' (line 19) and then enumerate all remaining edges of T' (lines 24-31). For each remaining edge of T' we check if it goes to T (line 29-31). In lines 26-28, 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 4 and 9).

`l` is valid because of line 3. `d` is valid because of the assignment in line 7. `i`, `j`, and $\mathtt{k}_1$ are valid because at line 11 T is contractable. `r` is valid because of line 19. $\mathtt{j}_1$, `k`, and `nd` are valid because at line 23 T' is contractable.

The recursive call in line 21 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 argBackLabel = 0 then return 0;

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

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

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

14         after TreeForward(i) do
15             fl := MoveValid(j);
16             if fl then
```

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

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

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

29            for k1 := 0 to d-1 do
30                if (TreeForward(i), Move(j),
                             TreeForward(r), TreeForward(k), Move(j1)) =
                         TreeForward(k1) then
31                    return nd;
```

`BackLabelAux` takes one argument i. To take argument and return value `BackLabelAux` uses the arrays of `BackLabel`. `BackLabelAux` handles the T uncontractable case of `BackLabel`. Its local variables are `i`, `j`, `k`, `bl`, `nbl`, `r`, and `d`.

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 `bl` is the back-label of e (because of line 1). So we have to find the index of $(v, \texttt{bl})$ in T' ($(v, \texttt{bl})$ 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 $(v, \texttt{bl})$.

`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. `r`, `i`, `j`, `k`, and `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
    bl := Prev(BackLabel);
    if bl = null then
        // if T' is uncontractable call recursively
        Prev(BackLabel);
        return;

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

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

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

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

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

`Move` takes one argument i. Let $(v, j) = \mu_{cl-1}(cv, i)$. Assume that i and j are valid. Then `Move` returns j and moves the current vertex to v. Its local variables are `i` and `j`, which are valid by the assumption about the argument of `Move`.

function `Move`

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

MoveValid takes one argument i and returns true iff $\beta_{cl-1}(cv, i)$ is valid. It does not have local variables.

```
function MoveValid
    return Prev(BackLabel(argMoveValid)) ≠ null;
```

6.4.7 Solving undirected s, t-connectivity

MoveToRep moves the current vertex to $\mathrm{rep}_{R_{cl}}(cv)$. It does not have arguments and local variables.

```
procedure MoveToRep
    if level > 0 then
        Prev(MoveToRep);
        if TreeSize ≠ null then TreeForward(Root);
```

Connected is a global function which takes two arguments s and t and returns true iff s and t are connected in G.

```
global function Connected
    level := 3 · 2^⌈log^(2) n⌉ − 2;
    currvertex := s; MoveToRep;
    r := currvertex;
    currvertex := t; MoveToRep;
    return r = currvertex;
```


ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'