Optimal randomized EREW PRAM algorithms for finding spanning forests

Shay Halperin *   *Uri Zwick* *

April 13, 1998
Revised: February 6, 2000 and July 31, 2000

Abstract

We present the first randomized $O(\log n)$ time and $O(m+n)$ work EREW PRAM algorithm for finding a spanning forest of an undirected graph $G=(V,E)$ with n vertices and m edges. Our algorithm is optimal with respect to time, work and space. As a consequence we get optimal randomized EREW PRAM algorithms for other basic connectivity problems such as finding a bipartite partition, finding bridges and biconnected components, finding Euler tours in Eulerian graphs, finding an ear decomposition, finding an open ear decomposition, finding a strong orientation, and finding an *st*-numbering.

1 Introduction

Finding a spanning forest of an undirected graph $G=(V,E)$ is one of the most basic algorithmic graph problems. While the problem is trivial in the sequential setting, it seems far from being so in the parallel setting. A considerable number of researchers (see Tables 1 and 2 below) investigated the complexity of the problem in various parallel models including, in particular, various members of the PRAM family. In this work we consider the EREW PRAM model, the weakest member of this family, and obtain, for the first time, an optimal randomized algorithm for the problem. The algorithm presented runs in $O(\log n)$ time, with very high probability, using $O((m+n)/\log n)$ processors, where $n=|V|$ is the number of vertices and $m=|E|$ is the number of edges in the input graph.

A summary of previous PRAM algorithms for finding spanning forests is given in Table 1. A summary of PRAM algorithms for finding connected components is given in Table 2. An algorithm that finds a spanning forest of a graph also finds the connected components of this graph. The converse, however, is not necessarily true. The algorithms listed in Table 2, for example, find the connected components of a graph *without* finding a spanning forest of it.

All the $O(\log n)$ time EREW PRAM connectivity algorithms mentioned in Table 2 use *random walks* for defining, for every vertex v, a relatively large *virtual neighborhood* $C(v)$. All the vertices in the virtual neighborhood $C(v)$ of a vertex v are in the connected component of v. Vertices of $C(v)$, however, are not necessarily adjacent to v in the graph. The algorithms may then be seen to work on a virtual graph $G^*=(V,E^*)$ in which vertices u and v are connected, i.e., $(u,v)\in E^*$, if and only if u and v are virtual neighbors, i.e., if and only if $u\in C(v)$ or $v\in C(u)$. These algorithms produce, in a sense, a spanning forest of the virtual graph $G^*=(V,E^*)$ and not of the original graph $G=(V,E)$. There does not seem to be

*Department of Computer Science, School of Mathematical Sciences, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv 69978, Israel. E-mail addresses : {hshay,zwick}@math.tau.ac.il. Work supported in part by **The basic research foundation** administrated by **The Israel academy of sciences and humanities**.

CRCW PRAM

authors	ref.	time	processors	det./ran.
Shiloach, Vishkin	[SV82]	$O(\log n)$	$m+n$	deterministic
Awerbuch, Shiloach	[AS87]	$O(\log n)$	$m+n$	deterministic
Cole, Vishkin	[CV91]	$O(\log n)$	$O((m+n)\alpha(m,n)/\log n)$	deterministic
Iwama, Kambayashi	[IK94]	$O(\log n)$	$O((m+n)\alpha(m,n)/\log n)$	deterministic
Gazit	[Gaz91]	$O(\log n)$	$O((m+n)/\log n)$	randomized

CREW PRAM

authors	ref.	time	processors	det./ran.
Hirschberg, Chandra, Sarwate	[HCS79]	$O(\log^2 n)$	$O(n^2/\log n)$	deterministic
Chin, Lam, Chen	[CLC82]	$O(\log^2 n)$	$O(n^2/\log^2 n)$	deterministic
Han, Wagner	[HW90]	$O(\log^2 n)$	$O((m+n\log n)/\log^2 n)$	deterministic
Johnson, Metaxas	[JM97]	$O(\log^{3/2} n)$	$m+n$	deterministic

EREW PRAM

authors	ref.	time	processors	det./ran.
Nash, Maheshwari	[NM82]	$O(\log^2 n)$	$O(n^2/\log^2 n)$	deterministic
Phillips	[Phi89]	$O(\log^2 n)$	$O((m+n)/\log n)$	deterministic
Kruskal, Rudolph, Snir	[KRS90]	$O(\log^2 n)$	$O((m+n)/\log n)$	deterministic
Johnson, Metaxas	[JM95]	$O(\log^{3/2} n)$	$m+n$	deterministic
Chong, Lam	[CL95]	$O(\log n \log\log n)$	$m+n$	deterministic
Halperin, Zwick	$\star\star\star$	$O(\log n)$	$O((m+n)/\log n)$	randomized

Table 1: PRAM algorithm for finding spanning forests.

EREW PRAM

authors	ref.	time	processors	det./ran.
Nisan, Szemerédi, Wigderson	[NSW92]	$O(\log n)$	$n^{O(1)}$	randomized
Karger, Nisan, Parnas	[KNP99]	$O(\log n \log\log n)$	$O((m+n)/\log n)$	randomized
Karger, Nisan, Parnas	[KNP99]	$O(\log n)$	$O((m+n^{1+\epsilon})/\log n)$	randomized
Radzik	[Rad94]	$O(\log n)$	$m+n$	randomized
Halperin, Zwick	[HZ96b]	$O(\log n)$	$O((m+n)/\log n)$	randomized

Table 2: PRAM algorithm for finding connected components.

an easy way to augment these algorithms so that they could also produce a spanning forest of the input graph.

The random walks technique was first used, in the sequential setting, by Aleliunas, Karp, Lipton, Lovasz and Rackoff [AKL+79] to obtain a randomized LOGSPACE connectivity algorithm. It was first used in the parallel setting by Nisan, Szemerédi and Wigderson [NSW92]. A much more efficient implementation was then presented by Karger, Nisan and Parnas [KNP99]. Further improvements were then obtained, independently, by Radzik [Rad94] and Halperin and Zwick [HZ96b].

In this work we present the first efficient parallel algorithm based on the random walks method that *does* produce a spanning forest of the original graph. We do so while keeping the same resource bounds as in [HZ96b], i.e., $O(\log n)$ time and $O(m+n)$ work. In addition, the algorithm presented here uses linear space. The algorithm of [HZ96b] needs more space.

The fact that we can now find a spanning forest, and not just the connected components, of an undirected graph $G = (V, E)$ in $O(\log n)$ time and $O(m+n)$ work, has many consequences. Many parallel algorithms for solving basic connectivity related problems of undirected graphs use, as a subroutine, an algorithm for finding spanning forests. By plugging our new algorithm into these algorithms we can get optimal EREW PRAM algorithms for many of these problems. Perhaps the simplest example of this kind is the problem of deciding whether a graph $G = (V, E)$ is *bipartite* and finding a bipartite partition of it if it is.

A less trivial task is the task of finding the *biconnected components* and the *bridges* of an undirected graph. Tarjan and Vishkin [TV85] described a reduction from the problem of finding the biconnected components of a graph to the problem of finding a spanning forest of the graph. Though their algorithm was designed with the CRCW PRAM model in mind, it is not difficult to check that all the operations of their algorithm, except those used for finding a spanning forest (for which they used the algorithm of Shiloach and Vishkin [SV82]), can be implemented optimally also in the EREW PRAM model. By using our algorithm for finding the spanning forests we thus obtain an optimal randomized EREW PRAM algorithm for finding the biconnected components and bridges of a graph. Further applications are mentioned in Section 8.

The algorithm presented here is based on the connectivity algorithm presented in [HZ96b]. Many of the essential details however are completely different. A summary of the main differences between these algorithms is given in Section 3. We also present here a greatly simplified analysis of the first stages of both algorithms.

An interesting feature of our algorithm is that it uses techniques from almost all the previous connectivity and spanning forest algorithms. In the first stage, for example, we use the *growth control* mechanism and the idea of *edge list plugging* used by Johnson and Metaxas [JM97],[JM95] and by Chong and Lam [CL95], and a sampling technique of Gazit [Gaz91]. In the second phase, we use a sampling technique of Karger, Klein and Tarjan [KKT95] which is also used by Karger, Nisan and Parnas [KNP99]. In the third stage we use *short random walks*, as was done by Karger, Nisan and Parnas [KNP99].

An overview of the new algorithm for finding spanning forests is given in the next section. The main differences between the new algorithm and the connected components algorithm of [HZ96b] are described in Section 3. The three main stages of the spanning forest algorithm are then described in the next three sections. In Section 8 we describe some of the applications of our algorithm. We end, in Section 10 with some concluding remarks and open problems.

A preliminary version of this work appeared in [HZ96a]. The algorithms presented here slightly simpler than those presented there.

2 Overview of the algorithm

The algorithm described in this paper receives an undirected graph $G = (V, E)$ with n vertices and m edges, and finds a spanning forest of it. The algorithm, like that of [HZ96b], is composed of three main stages. The first stage of the algorithm finds a forest $F^{(1)}$ in $G = G^{(1)}$. The forest $F^{(1)}$ is not necessarily spanning but it contains only $O(m/\log^4 n)$ trees that do not span complete connected components. By contracting all the trees in $F^{(1)}$, and by removing all the trees that span complete connected components, we obtain a graph $G^{(2)}$ with $O(m/\log^4 n)$ vertices and at most m edges. The second stage finds a forest $F^{(2)}$ in $G^{(2)}$ such that the number of edges in $G^{(2)}$ that connect vertices contained in different trees of $F^{(2)}$ is at most $O(m/\log^2 n)$. By contracting the trees in $F^{(2)}$ we obtain a graph $G^{(3)}$ with $O(m/\log^4 n)$ vertices and $O(m/\log^2 n)$ edges. In the third stage we find a spanning forest $F^{(3)}$ of $G^{(3)}$. By combining the forests $F^{(1)}$, $F^{(2)}$ and $F^{(3)}$ we obtain a spanning forest of G.

Each stage runs in $O(\log n)$ time using $O((m+n)/\log n)$ processors. Each stage is randomized. The probability that each one of the stages takes more then $O(\log n)$ time to finish is at most $O(n^{-c})$, for any $c > 0$. In the sequel, we refer to events that occur with probability of at least $1 - O(n^{-c})$, for any $c > 0$, as events that occur *w.h.p*, i.e., with high probability, and to events that occur with probability of at least $1 - 2^{-\Omega(n^\epsilon)}$, for some $\epsilon > 0$, as events that occur *w.v.h.p*, i.e., with *very* high probability. Note that if each of a polynomial number of events occurs w.h.p., or w.v.h.p., then their intersection also occurs w.h.p., or w.v.h.p., respectively. Each stage of the algorithm takes $O(\log n)$ time w.h.p. It is shown in Section 4 that the first stage of the algorithm in fact runs in $O(\log n)$ time w.v.h.p. The probability that the second or the third stage of the algorithm will take more than $O(\log n)$ time can be reduced, as shown on Section 7, to $2^{-\Omega(2^{\sqrt{\log n}})}$, which is $o(n^{-c})$, for any $c > 0$. Hence. the probability that the whole algorithm takes more than $O(\log n)$ time is at most $2^{-\Omega(2^{\sqrt{\log n}})}$. (Note that this probability falls somewhere between w.h.p. and w.v.h.p. We do not claim that the whole algorithm runs in $O(\log n)$ time w.v.h.p.) The result returned by the algorithm is always correct.

The third stage of the algorithm is a complete algorithm for finding a spanning forest of a graph. This algorithm uses, however, $O((m+n)\log n)$ processors when working on a graph with n vertices and m edges. The number of processors used by the third stage can be easily reduced to $O(m+n)$, as described in section 7. The role of the first two stages of the algorithm is to reduce the number of edges and vertices in the graph by a logarithmic factor so that the third stage would become cheap enough. As the third stage of the algorithm is in itself a complete algorithm for finding spanning forests, and as the first two stages of the algorithm are technically involved, some readers may find it easier to concentrate at first on the third stage of the algorithm and only afterwards turn their attention to the first two stages of the algorithm. These readers are advised, however, to at least skim through the first two stages of the algorithm before moving on to the third stage.

3 Comparison with the connectivity algorithm of [HZ96b]

The algorithm of [HZ96b] starts by transforming the input graph G into a 3-regular graph. It then carefully controls the growth of the vertex degrees as the graph undergoes a series of contractions. The analysis and the implementation of the algorithm are heavily dependent on the controlled growth of the degrees. Here we find it more convenient to use a different strategy. Instead of transforming the graph G into a 3-regular graph, we mark vertices with low degrees as *active* and the rest as *inactive*. Only active vertices are then hooked on other vertices. (Hookings are explained in the next paragraph.) This simple measure gives us an alternative way of controlling the degrees in the graph. A similar idea is used by Chong and Lam [CL95].

The algorithm of [HZ96b] is composed of three stages, analogous to the three stages of the current algorithm. The first and third stages of both algorithms are composed of alternating *hooking* and *contraction* phases. In a hooking phase, some of the vertices are hooked on other vertices thus forming a forest. In contraction phases, the trees of these forests are contracted. In [HZ96b], vertices are hooked on vertices from their connected components, but necessarily on neighbors of theirs. Here we have to change most of the hooking operations to ensure that a vertex is always hooked on a *neighbor* of it.

The analysis of the first stage of [HZ96b], given in [HZ96b], is extremely complicated. An additional contribution of this work is a much simpler analysis of this stage. This new analysis also enabled us to simplify the first stage of the algorithm itself. The new analysis is described in Section 4.3.

The algorithm presented here uses only $O(m+n)$ *space*. The algorithm of [HZ96b] needs $O(m+n^{1+\epsilon})$

procedure *REDUCE-VERTEX*(G)

input: A graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges.

output: A forest F of G such that at most $O(m/\log^4 n)$ of its trees do not span complete connected components of G, and a graph $G' = (V', E')$ with at most $O(m/\log^4 n)$ non-isolated vertices obtained by contracting the edges of the forest F.

complexity: $O(\log n)$ time, w.v.h.p., using $O((m+n)/\log n)$ processors.

```
G_0 ← G.
for i ← 0 to k − 1 (where k = O(log log n)) do
{
    1. Let d_i = (4/3)^i.
    2. G_{i+1} ← DISTILL(G_i, d_i).
    3. Delete, w.v.h.p., at least 1/2 of all the white edges of G_{i+1}.
}
```

The forest F is composed of all the edges of G marked as forest edges during the calls to *DISTILL*. The graph G' is obtained by contracting the edges of F.

Figure 1: The first stage of the algorithm: reducing the number of vertices.

space. One of the means used to reduce the space required is the use of the linked-list based sorting algorithm of Goodrich and Kosaraju [GK96] instead of the array-based sorting algorithm of Cole [Col88]. This sorting method is used frequently in the third stage of the algorithm.

4 First stage: reducing the number of vertices

In this section we describe the first stage of our algorithm. The input to this stage is a graph G with n vertices and m edges. In this stage we construct a forest F composed of at most $O(m/\log^4 n)$ trees that do not span complete connected components. By contracting the trees of this forest, and then removing isolated vertices, we obtain a graph G' with $O(m/\log^4 n)$ vertices and at most m edges. The first stage may be skipped if $m = \Omega(n \log^4 n)$.

The input graph $G = (V, E)$ is specified using its adjacency lists. For each vertex $v \in V$, we have a linked list $N(v)$ that contains the edges $(v, u_1), (v, u_2), \ldots, (v, u_d)$ touching v. Each edge (v, u) in $N(v)$ has a pointer to its opposite edge (u, v) in $N(u)$.

The first stage of the algorithm consists of a call to procedure *REDUCE-VERTEX* whose description is given in Figure 1. The operation of *REDUCE-VERTEX* is composed of $k = O(\log \log n)$ iterations. Each iteration reduces the number of vertices by a constant factor. Each iteration involves a call to a procedure called *DISTILL*. A description of *DISTILL* is given in Figure 2.

Procedure *DISTILL* receives a graph G_i and returns a graph G_{i+1} which is a *minor* of G_i. (A minor of a graph G is a graph obtained from G by the contraction and the deletion of edges.) The first graph G_0 fed into *DISTILL* is the input graph G. The edges of G_i that are contracted in the transition from G_i to G_{i+1} form a forest F_i of G_i. Each vertex of G_i corresponds to a tree in G. Each edge of G_i corresponds to an original edge of G. The forest F_i may also be viewed, therefore, as a forest of G. The union of the forests

$F_0, F_1, \ldots, F_{k-1}$, which would also be a forest, is the forest F mentioned above. During its operation, *DISTILL* marks the edges of G that are to be included in the forest F. This will be explained in more detail when the operation of *DISTILL* is described.

Throughout the operation of *REDUCE-VERTEX* we maintain, w.v.h.p., the following invariant: The graph G_i contains at most $4m/d_i$ non-isolated vertices and at most $200m/d_i$ edges, where m is the number of edges in the input graph G, and $d_i = (4/3)^i$. Note that the number of non-isolated vertices of G_i is bounded in terms of the number of *edges* of the input graph. As some of the edges of G_i are deleted while forming G_{i+1}, the number of connected components of G_{i+1} may be larger than the number of connected components of G_i. The edges deleted by *DISTILL* are carefully chosen, however, so that the number of connected components of G_{i+1} is only slightly larger than the number of connected components of G_i. A connected component of G_{i+1} is said to be a *new* connected component if it does not correspond to a complete connected component of G_i. The number of new connected components in G_{i+1} is clearly an upper bound on increase in the number of connected components in the transition from G_i to G_{i+1}. While implementing *DISTILL* we will make sure that the number of new connected components in G_{i+1} is $O((m/d_i)/\log^4 n)$.

As mentioned to above, the graph G_{i+1} is obtained by the contraction of a forest of G_i and by the deletion of some edges. Such a contraction reduces the number of non-isolated vertices. It may, however, leave the number of edges almost unchanged. As the work performed in each stage is proportional to the number of edges left in the graph, and as we want the total number of operations performed in all iterations to be linear in the size of the input graph, we want the number of edges in the graphs G_i to decrease geometrically. This is why we are forced to delete some fraction of the edges in each iteration.

After $k = \lceil (4/\log(4/3)) \cdot \log\log n \rceil$ iterations, the number of non-isolated vertices is down to $O(m/\log^4 n)$. The forest F is then composed of $O(m/\log^4 n)$ trees that do not span complete connected components of G. The graph G' returned by *REDUCE-VERTEX* is obtained by contracting the edges of the forest F. Note that the graph G' is usually different from the graph G_k obtained after the last iteration. The graphs G' and G_k have the same set of vertices. Some of the edges of G' may be missing, however, from G_k, as the transition from each G_i to G_{i+1} involves the deletion of edges.

As we want the total number of operation performed by *REDUCE-VERTEX* to be $O(m+n)$, *DISTILL* is forced, as mentioned, to perform some edge deletions. As some of the vertices of the graphs G_i may have large degrees, compressing the adjacency lists of the graphs G_i after each phase of edge deletions, or even after each call to *DISTILL*, may be too time consuming. *DISTILL* adopts, therefore, the following strategy. Edges of G_i that are to be deleted are first marked for deletion. This is done by coloring these edges *white*. White edges are then gradually removed from the adjacency lists of the graph. Edges that are not colored white are said to be *black*. Initially, all the edges of the input graph are black. After each call to *DISTILL*, in step 3 of each iteration of *REDUCE-VERTEX*, we perform a clean-up stage in which at least 1/2 of all the white edges of G_{i+1} are physically removed from the adjacency lists. The way in which this is performed will be described in the next subsection.

We let n_i be the number of non-isolated vertices of G_i and m_i be the number of black (i.e., non-white) edges of G_i. The graph G_i is an undirected graph. Each edge (u, v) appears twice, however, in the representation of G_i, once as (u, v) in the adjacency list of u and once as (v, u) in the adjacency list of v. If the edge (u, v) is to be deleted, then both its copies are colored white. The two copies are treated independently from that point onward. A white edge (u, v) may be removed from the adjacency lists of the graph (in step 3), while its opposite edge (v, u) still remains, as a white edge, in the representation. We let $\bar{m}_i$ be the number of *directed* edges in the representation of G_i. (Note that m_i counts the number of undirected black edges in G_i while $\bar{m}_i$ counts directed edges.) We mentioned above that throughout the operation of *DISTILL* we maintain, w.v.h.p., the invariant $m_i \leq 200m/d_i$. We also make sure, w.v.h.p., that $\bar{m}_i \leq 600m/d_i$. This means that the number of white edges in the representation of G_i is, w.v.h.p., at most three times the bound that we maintain on the number of black edges.

The invariants maintained by *REDUCE-VERTEX* are summarized in the following lemma.

Lemma 4.1 *For every* $0 \le i \le k$ *we have, w.v.h.p.:*
(i) $n_i \le 4m/d_i$, *(ii)* $m_i \le 200m/d_i$, *(iii)* $\bar{m}_i \le 600m/d_i$,
(iv) The number of new connected components in G_i *is* $O((m/d_i)/\log^4 n)$.

Proof: Invariants (i), (ii) and (iv) follows immediately from the specifications of *DISTILL* (see Lemma 4.3 and Theorems 4.6 and 4.7). We are left, therefore, with the task of establishing the validity of (iii).

We know that $m_i \le 200m/d_i$, w.v.h.p. We prove by induction on i that $\bar{m}_i \le 600m/d_i$, w.v.h.p. The claim is certainly true for $i = 0$. Assume the claim is true for i. The representation of G_i contains $\bar{m}_i$ (directed) edges. After step 2 of the i-th iteration, the graph has only m_{i+1} black edges. The representation of the graph now contains $2m_{i+1}$ black edges (as each black edge appears twice in the representation) and at most $\bar{m}_i - 2m_{i+1}$ white edges. In step 3, we remove, w.v.h.p., at least a half of all the white edges from the representation of the graph. The size of the representation of G_{i+1} is then at most

$$\bar{m}_{i+1} \le 2m_{i+1} + \frac{\bar{m}_i - 2m_{i+1}}{2} = m_{i+1} + \frac{\bar{m}_i}{2} \le \frac{200m}{d_{i+1}} + \frac{300m}{d_i} = \frac{600m}{d_{i+1}} ,$$

as required. □

Finally, we bound the number of non-isolated vertices in the output graph of *REDUCE-VERTEX*.

Theorem 4.2 *The number of non-isolated vertices in the graph* G' *returned by REDUCE-VERTEX, and therefore the number of trees in the forest* F *that do not span complete connected components of* G*, is, w.v.h.p.,* $O(m/\log^4 n)$.

Proof: By Lemma 4.1 we know that the number of non-isolated vertices in G_k is at most $O(m/d_k)$, which is $O(m/\log^4 n)$, when $k = \lceil (4/\log(4/3)) \cdot \log\log n \rceil$.

The graph G_k is a subgraph of the output graph G'. The graph G' may therefore contain more non-isolated vertices than G_k. An isolated vertex in G_k which is not isolated in G' corresponds to a new connected component formed by one of the calls to *DISTILL*. By Lemma 4.1(iv), the i-th call to *DISTILL* generates, w.v.h.p., only $O((m/d_i)/\log^4 n))$ new connected components. As m/d_i is a geometrically decreasing sequence, the total number of new connected components generated by all the call to *DISTILL* is $O(m/\log^4 n)$, and the total number of non-isolated vertices in G' is also $O(m/\log^4 n)$. □

4.1 Implementation details of *REDUCE-VERTEX*

The input graph $G = (V, E)$, as well as the graphs G_i, are represented using adjacency lists. As mentioned above, some of the edges in the adjacency lists of the graphs G_i are colored white.

In the implementation of *REDUCE-VERTEX*, and of *DISTILL*, we use the processor allocation mechanism of [HZ96b]. As this mechanism is essential for the implementation of our algorithm, we briefly sketch it here.

Let ALG be a (possibly randomized) parallel algorithm that uses m processors. The execution of ALG is composed of K phases. All m processors are active at the beginning of the first phase. At the end of each phase some of the processors become inactive, and remain so until the end of the algorithm. The i-th phase of ALG takes $t_i \ge 1$ time units and m_i of the processors are active in it.

The total running time of ALG is clearly $T = \sum_{i=1}^{K} t_i$. The total amount of work performed by active processors is $W = \sum_{i=1}^{K} m_i t_i$. It is shown in [HZ96b] that ALG can be simulated, w.h.p., using P processors, in $O(W/P + T + K \log\log m + \log m)$ time. No concurrent read or write operations are introduced by this simulation.

We implement *REDUCE-VERTEX* in two stages. We assume, at first, that each vertex and each edge in the representation of G_i has a processor allocated to it. We refer to these processors as *virtual* processors. (Note that virtual processors are also allocated to white edges.) Each virtual processor becomes inactive when the vertex to which it is allocated becomes isolated, or when the edge to which it is allocated is physically removed from the adjacency lists. (In *DISTILL* we need to revive some of the virtual processors, this will be discussed there.) The operations performed by the active virtual processors are called *useful* operation. We then simulate these $m+n$ virtual processors with $(m+n)/\log n$ actual processors using the processor allocation scheme of [HZ96b].

Most of the work of *REDUCE-VERTEX* is performed in the calls to *DISTILL*. The operations of *DISTILL* will be described separately. All we have to do, therefore, to finish the description of *REDUCE-VERTEX* is to describe the way in which at least half of the white edges are removed at the end of each iteration, and the way in which the output forest F and the output graph G' are constructed.

The removal of at least a half of the white edges from the adjacency lists w.v.h.p. in step 3, is done using a simple randomized technique. Each one of the white edges chooses a spin 0 or 1. If a white edge (v, u) in the adjacency list of v chooses the spin 0, and its predecessor in the list chooses the spin 1 (or it is the first edge in the list, or its predecessor is not a white edge), then we remove the edge (v, u) from the list. The spins assigned to the edges ensure that no two consecutive white edges are deleted at the same time unit. We assume that each adjacency list is implemented as a doubly linked list. It is therefore easy to remove the selected white edges from the lists in constant time without concurrent read or write operations. This operation removes, on average, at least 1/4 of all the white edges. Using Chernoff's bound (see Alon and Spencer [AS92] or Motwani and Raghavan [MR95]), we get that, w.v.h.p., at least 1/5 of the white edges are removed. By repeating this operation three times, we get that, w.v.h.p., at least 1/2 of the white edges are removed. (This is essentially the random mating algorithm of [MR89].)

Before starting its operation, *REDUCE-VERTEX* makes a copy of the original input graph G. Each call to *DISTILL* marks some of the edges in this copy of G for inclusion in the forest F. After the $O(\log\log n)$ iterations, all the edges contained in the forest F are marked. The graph G' is easily obtained by contracting all the edges of F. This contraction can be easily implemented in $O(\log n)$ time using $O((m+n)/\log n)$ processors using the edge list plugging technique of Johnson and Metaxas [JM97],[JM95] and other basic EREW PRAM techniques (see the description of *CONTRACT* below).

4.2 The complexity of *REDUCE-VERTEX*

As mentioned, we assume at first that each non-isolated vertex and each non-removed edge has a virtual processor allocated to it. According to its specifications, the running time of the i-th call to *DISTILL* is $O((\log d_i)\cdot\log\log n + (\log\log n)^2)$ (see next subsection), in $O(\log\log n)$ phases. Step 3 of each iteration of *REDUCE-VERTEX* takes $O(1)$ time. The total running time, in this idealized model, is therefore

$$T = O(\sum_{i=0}^{k-1}(\log d_i)\cdot\log\log n + (\log\log n)^2) = O(\sum_{i=0}^{k-1}[i\cdot\log\log n + (\log\log n)^2]) = O((\log\log n)^3),$$

in $K = O((\log\log n)^2)$ phases. The number of *useful* operations performed by the all virtual processors that are still active in the i-th iteration is $O(\bar{m}_i \log d_i) = O((m\log d_i)/d_i)$ (see Lemma 4.1). The total number of *useful* operations is therefore

$$W = O(\sum_{i=0}^{k-1}\frac{m}{d_i}\log d_i) = O(m\cdot\sum_{i=0}^{k-1} i(\frac{3}{4})^i) = O(m).$$

Using the processor allocation scheme of [HZ96b], we can implement *REDUCE-VERTEX* using $P = (m+n)/\log n$ processors in $O(\log n)$ time, w.h.p., as required. Using a tuned up version of the processor allocation scheme of [HZ96b] we can implement *REDUCE-VERTEX* is such a way that the probability that its running time would exceed $O(\log n)$ is at most $2^{-\Omega(2^{\log n/(\log\log n)^2})}$.

As mentioned in the previous subsection, it is easy to implement the final step of *REDUCE-VERTEX*, that of contracting the edges of the forest F and the construction of the graph G', deterministically in $O(\log n)$ time using $O((m+n)/\log n)$ processors.

4.3 Procedure *DISTILL*

Procedure *DISTILL* is a simplified version of procedure *DILUTE* used in [HZ96b]. In [HZ96b], we gave a rather complicated analysis of *DILUTE*. Here we present a much simpler analysis.

A high level description of *DISTILL* is given in Figure 2. The general structure of *DISTILL* is similar to the structure of *REDUCE-VERTEX*. *DISTILL*, like *REDUCE-VERTEX*, performs $k = O(\log\log n)$ iterations. In each iteration we aim at reducing the number of non-isolated vertices by a constant factor, and the number of edges, again by a constant factor.

At the beginning of each iteration, we let $d_{i,j} = d_i \cdot (4/3)^j = (4/3)^{i+j}$. Non-isolated vertices of degree at most $d_{i,j}$ are marked as active. All other vertices are marked as inactive. (The degree of a vertex is the number of edges, including white edges, in its adjacency list.)

The reduction in the number of vertices is obtained by constructing a forest, using procedure *HOOK*, and then contracting it, using procedure *CONTRACT*. Procedure *HOOK*, which is described shortly in more detail, hooks active vertices on neighboring active and inactive vertices.

The reduction in the number of edges is obtained using more 'brutal' means. In step 4 of each iteration we simply delete (or more precisely, mark for deletion) about $1/5$ of all the surviving edges. At the beginning of *DISTILL*, all the black edges are marked as live. In step 4 of each iteration, each live edge is marked as dead with probability $1/5$. All decisions are made independently. The edges marked as dead are also colored white. The 'dead' marks are used to distinguish between edges that were already colored white in the graph fed into *DISTILL*, and those that were killed in one of the iterations of *DISTILL*. This distinction is important as we shall soon see.

The input to *DISTILL*, in the i-th iteration of the main loop of *REDUCE-VERTEX*, is the graph G_i. *DISTILL* constructs a sequence of graphs $G_{i,0}, G_{i,1}, \ldots, G_{i,k}$, where $G_{i,0} = G_i$, and $G_{i,j+1}$ is obtained from $G_{i,j}$ by contracting a forest and then deleting some edges. (Note that $G_{i,k}$ is *not* the output of *DISTILL*.)

We let $n_{i,j}$ be the number of non-isolated vertices of $G_{i,j}$, $m_{i,j}$ be the number of black edges of $G_{i,j}$, and $\bar{m}_{i,j}$ be the total number of (directed) edges in the representation of $G_{i,j}$. Note that $n_{i,0} = n_i$, $m_{i,0} = m_i$ and $\bar{m}_{i,0} = \bar{m}_i$.

Throughout the operation of *DISTILL* we maintain, w.v.h.p., the invariants $n_{i,j} \le 4m/d_{i,j}$, $m_{i,j} \le (5/6)^j \cdot 200m/d_i$, and $\bar{m}_{i,j} \le (5/6)^j \cdot 600m/d_i$.

As mentioned, the graph $G_{i,j+1}$ is obtained from $G_{i,j}$ by contracting a forest, and then deleting some edges. Vertices that are isolated after these deletions are said to be *introvert* (this term is taken from [Gaz91]). A vertex is isolated if all the edges in its adjacency list are colored white (white edges may also be dead). Checking whether a vertex with an adjacency list of length d is introvert requires $\Omega(\log d)$ time, when concurrent read and write operations are not allowed. We only check, therefore, whether active vertices become introvert. Those that do are marked as *introvert*. Note that only active vertices are marked as introvert.

The graph $G_{i,k}$ obtained after the $k = O(\log\log n)$ iterations of *DISTILL* is a minor of G_i, the input graph procedure *DISTILL*(G_i, d_i)

input: A graph $G_i = (V_i, E_i)$ with at most $4m/d_i$ non-isolated vertices and at most $200m/d_i$ black edges. The number of edges, including white edges, in the representation of G_i is at most $600m/d_i$.

output: A graph $G_{i+1} = (V_{i+1}, E_{i+1})$ which is a minor of G_i. It has at most $4m/d_{i+1}$ non-isolated vertices, at most $200m/d_{i+1}$ black edges, and at most $O((m/d_i)\log^4 n)$ new connected components.

complexity: $O((\log d_i)\log\log n + (\log\log n)^2)$ time, w.v.h.p., in $O(\log\log n)$ phases, and $O((m/d_i)\log d_i)$ useful operations using $O(m/d_i)$ virtual processors.

$G_{i,0} \leftarrow G_i$.
Mark all the edges of $G_{i,0}$ as live.

`for` $j \leftarrow 0$ `to` $k-1$ (where $k = O(\log\log n)$) `do`
{

1. Let $d_{i,j} = d_i{\cdot}(4/3)^j$. Mark non-isolated vertices of degree at most $d_{i,j}$ as active. Mark all other non-isolated vertices as inactive.
2. `parent` $\leftarrow$ *HOOK*$(G_{i,j})$.
3. $G_{i,j+1} \leftarrow$ *CONTRACT*$(G_{i,j},$ `parent`$)$.
4. Each live edge stays live with probability $\alpha = 4/5$, and becomes dead with probability $1-\alpha$. All decisions are made independently. Dead edges are colored white. (Do not perform this step if this is the last iteration.)
5. Mark active vertices that are now isolated (all their edges are dead and/or white) as *introverted*.
6. Delete at least $1/2$ of the white edges of $G_{i,j+1}$.

}

Remove all the edges of $G_{i,k}$ and revive the dead edges of active introvert vertices. (See Section 4.5 for an explanation of this step.) The obtained graph is the output graph G_{i+1}.

Figure 2: Procedure *DISTILL*.

of *DISTILL*. As mentioned, $G_{i,k}$ is *not* the graph returned by *DISTILL*.

Let $V_{i,k}$ be the vertex set of $G_{i,k}$. Each vertex of $V_{i,k}$ is a coalescence of possibly many vertices of G_i. Let $h_i : V_i \to V_{i,k}$ be the function that maps each vertex of G_i into the vertex of $G_{i,k}$ into which it was contracted. Let $G'_{i,k}$ be the graph $(V_{i,k}, E'_{i,k})$ whose vertex set is identical to that of $G_{i,k}$ and whose edge set is $E'_{i,k} = \{(h_i(u), h_i(v)) \mid (u,v) \in E_i \text{ and } h_i(u) \neq h_i(v)\}$, where E_i is the edge set of G_i. In other words, $G'_{i,k}$ is the graph obtained from G_i using the same sequence of edge contractions used to obtain $G_{i,k}$, but *without* the edge deletions. (We should point out here that the graph $G'_{i,k}$ is *not* constructed explicitly by *DISTILL*.)

procedure *HOOK*$(G_{i,j})$

input: A graph $G_{i,j} = (V_{i,j}, E_{i,j})$ with at most $4m/d_{i,j}$ non-isolated vertices and at most $(5/6)^j \cdot 200m/d_i$ black edges. The non-isolated vertices of degree at most $d_{i,j}$ are marked as active. All other non-isolated vertices are marked as inactive. The edges of $G_{i,j}$ are marked as live, dead or white.

output: An array parent of pointers. For each active vertex v, parent$[v]$ is either v or an active or inactive neighbor of v. For each inactive vertex u, parent$[u] = u$. The parent pointers define a tree in $G_{i,j}$.

complexity: $O(\log d_{i,j})$ time, in a single phase, and $O(\bar{m}_{i,j} \cdot \log d_{i,j})$ useful operations using $O(\bar{m}_{i,j})$ virtual processors, where $\bar{m}_{i,j}$ is the number of edges, including white edges, in the representation of $G_{i,j}$.

0. For every vertex v let parent$[v] \leftarrow v$.
1. For each active vertex v find the first live edge (v, u) in the adjacency list of v for which $u > v$. If there is such an edge then let parent$[v] \leftarrow u$.
2. For each active vertex v such that parent$[v] = v$, and $parent[u] \neq v$, for all the neighbors of v, find the first live edge (v, u) in the adjacency list of v. If there is such an edge then parent$[v] \leftarrow u$.

Figure 3: Procedure *HOOK*.

The graph $G_{i,k}$ obtained after the $O(\log\log n)$ iterations is a subgraph of $G'_{i,k}$. Active vertices that are isolated in $G_{i,k}$ are marked in step 5 of each iteration as *introverted*.

We are finally able to describe the graph $G_{i+1} = (V_{i+1}, E_{i+1})$ returned by *DISTILL*. The vertex set V_{i+1} of G_{i+1} is $V_{i,k}$, the vertex set of $G_{i,k}$. The edges of G_{i+1} are the edges of $G'_{i,k}$ that touch active introvert vertices.

The output graph G_{i+1} is described above in terms of the graph $G'_{i,k}$. As mentioned, however, the graph $G'_{i,k}$ is not constructed explicitly by *DISTILL*. Constructing the graph G_{i+1} in the actual implementation of *DISTILL* is a non-trivial task that will be described in Section 4.5.

The output graph G_{i+1} is clearly a minor of the input graph G_i. As some edges were deleted in the transition from G_i to G_{i+1}, each connected component of G_i may be broken into several connected components of G_{i+1}. Connected components of G_{i+1} that do not correspond to complete connected components of G_i are said to be *new* connected components. The hardest task of *DISTILL* is making sure that the number of new connected components of G_{i+1} is small enough, i.e., $O((m/d_i)/\log^4 n))$, where n and m are the number of vertices and edges, respectively, in the input graph G of *REDUCE-VERTEX*.

Procedure *HOOK*, called by *DISTILL*, is described in Figure 3. Procedure *HOOK* receives the graph $G_{i,j}$ and returns a pointer array parent. The pointer array parent represents a forest in $G_{i,j}$. Procedure *HOOK* uses a hooking technique used by Shiloach and Vishkin [SV82] and by Awerbuch and Shiloach [AS87]. Each vertex attempts to hook on a neighbor with a larger serial number. Vertices that did not find a larger neighbor to hook on, and that none of their neighbors hooked on them, hook on an arbitrary neighbor of theirs. It is not difficult to see, and it is shown in [AS87], that no cycles are formed.

procedure *CONTRACT*(G, parent)

input: A graph $G = (V, E)$ and a forest defined by the parent pointers.

output: A graph G' obtained by contracting all the trees of the forest defined by parent.

complexity: $O(1)$ time, assuming that a virtual processor is allocated to each vertex of G.

assumptions: Each vertex v as a linked list $N(v)$ of its edges and pointers first[v] and last[v] to the first and last edges in $N(v)$. Each vertex v which is not a root, i.e., parent[v] $\neq v$, has a pointer to the edge (v, parent[v]) in its adjacency list, and through it to the edge (parent[v], v) in the adjacency list of parent[v] (this pointer can be easily created by *HOOK*).

For each vertex v :
If parent[v] $\neq v$, insert the list $N(v)$ after the edge (parent[v], v) in the adjacency list of parent[v]. (This can be easily done using the pointers first[v] and last[v].

Figure 4: Procedure *CONTRACT*.

Checking whether a vertex with an adjacency list of length d has a neighbor with a larger serial number requires $\Omega(\log d)$ time in the EREW PRAM model. We cannot afford, therefore, to hook vertices with relatively large degrees. For this reason, only active vertices, i.e., vertices whose degrees are at most $d_{i,j}$, are hooked by *HOOK*.

The forest created by *HOOK* is contracted by *CONTRACT*. Procedure *CONTRACT* is described in Figure 4. Procedure *CONTRACT* contracts *all* the trees formed by *HOOK*. It uses the *edge list plugging* technique of [JM97]. If v is hooked on u, then the adjacency list of v is placed after the edge (u, v) in the adjacency list of u. The vertex u may, at the same time, be hooked on a vertex w, and so on. All these 'pluggings' are performed concurrently in *constant* time without concurrent read or write operations. Note, however, that the endpoint names of the edges in the adjacency lists formed may be out of date. The way in which this is dealt with is described in Section 4.5.

We should note here that *DISTILL*, as well as *DILUTE* of [HZ96b], are based on the *partitioning* procedure of Gazit [Gaz91]. The main difference between *DISTILL* (and *DILUTE*) and the partitioning procedure of Gazit is that in the i-th iteration of Gazit's procedure, each edge is live with probability α^{-i}, even if it were dead in one of the previous iterations. In the EREW PRAM implementation of *DISTILL* we cannot revive edges in that fashion.

4.4 Correctness of *DISTILL*

In this subsection we show that *DISTILL* does indeed meets its specifications. We begin by showing that w.v.h.p., $n_{i,j} \leq 4m/d_{i,j}$, $m_{i,j} \leq (5/6)^j \cdot 200m/d_i$ and $\bar{m}_{i,j} \leq (5/6)^j \cdot 600m/d_i$, for $0 \leq j \leq k$.

Lemma 4.3 *The number of non-isolated vertices in $G_{i,j}$ is at most $4m/d_{i,j}$.*

Proof: The proof is by induction on j. Let $n_{i,j}$ be the number of vertices of $G_{i,j}$. The graph $G_{i,0}$ is the input graph G_i of *DISTILL* and we are promised that $n_{i,0} \le 4m/d_i = 4m/d_{i,0}$. Assume now that $n_{i,j} \le 4m/d_{i,j}$. Let $ina_{i,j}$ be the number of inactive vertices in $G_{i,j}$. The number of active vertices in $G_{i,j}$ is therefore $n_{i,j} - ina_{i,j}$. As the multigraph $G_{i,j}$ contains at most m edges, and the degree of each inactive vertex is at least $d_{i,j}$, we get that $ina_{i,j} \le 2m/d_{i,j}$. The forest defined in $G_{i,j}$ by the parent pointers contains at most $ina_{i,j}$ trees whose roots are inactive vertices, and at most $\frac{1}{2}(n_{i,j} - ina_{i,j})$ trees whose roots are non-isolated active vertices (as each such tree contains at least two active vertices). Each non-isolated vertex in $G_{i,j+1}$ is obtained by contracting one of these trees. Thus,

$$n_{i,j+1} \;\le\; ina_{i,j} + \frac{1}{2}(n_{i,j} - ina_{i,j}) \;=\; \frac{1}{2}(n_{i,j} + ina_{i,j}) \;\le\; \frac{1}{2}(\frac{4m}{d_{i,j}} + \frac{2m}{d_{i,j}}) \;=\; \frac{3m}{d_{i,j}} \;=\; \frac{4m}{d_{i,j+1}} \;,$$

as required. □

Lemma 4.4 *The number of live edges in $G_{i,j}$ is, w.v.h.p., at most $(5/6)^j \cdot 200m/d_i$. The total number of edges in $G_{i,j}$, including white edges, is, w.v.h.p, at most $(5/6)^j \cdot 600m/d_i$.*

Proof: For brevity, we let $A = 200$. Let $m_{i,j}$ be the number of (undirected) live edges in $G_{i,j}$. Let $\bar{m}_{i,j}$ be the total number of (directed) edges in $G_{i,j}$. We are promised that $m_{i,0} \le Am/d_i$. In each iteration, each live edge stays live with probability 4/5. Using the Chernoff bound, we get that, w.v.h.p, at most 5/6 of the edges that are live at the beginning of the j-th iteration remain live at the end of the j-th iteration. This immediately implies that, w.v.h.p, $m_{i,j} \le (5/6)^j Am/d_i$, for every $1 \le j < k$.

Let $\bar{m}_{i,j}$ be the total number of edges in $G_{i,j}$. We prove the claim $\bar{m}_{i,j} \le (5/6)^j \cdot 3Am/d_i$ by induction on j. Again, we are promised that $\bar{m}_{i,0} \le 3A \cdot m/d_{i,0}$, w.v.h.p. We now suppose that the claim holds for j and prove it for $j+1$.

Consider the graph $G_{i,j+1}$ before the execution of step 6 of *DISTILL*. The number of black edges in $G_{i,j+1}$ is $m_{i,j+1}$. The number of (directed) white edges in $G_{i,j+1}$ is $\bar{m}_{i,j} - 2m_{i,j+1}$. In step 6 we delete, w.v.h.p., at least 1/2 of all the white edges. (This is done using the same technique used by *REDUCE-VERTEX*.) The number of the white edges after this step is, w.v.h.p., at most $(\bar{m}_{i,j} - 2m_{i,j+1})/2$. The total number of edges in $G_{i,j+1}$ is therefore

$$\bar{m}_{i,j+1} \le 2m_{i,j+1} + \frac{\bar{m}_{i,j} - 2m_{i,j+1}}{2} = m_{i,j+1} + \frac{\bar{m}_{i,j}}{2} \le [(5/6)^j + \frac{3}{2}(5/6)^j] \cdot Am/d_i \le (5/6)^{j+1} \cdot 3Am/d_i \;.$$

as required. □

We next prove the required properties of G_{i+1}: it has at most $4m/d_{i+1}$ non-isolated vertices, at most $200m/d_{i+1}$ black edges, and the number of new connected components in G_{i+1} is $O((m/d_i)/\log^4 n)$.

Lemma 4.5 *The number of non-isolated vertices in G_{i+1}, the graph returned by DISTILL, is at most $4m/d_{i+1}$.*

Proof: As edges are marked for deletion only at the end of each iteration of *DISTILL*, the number of non-isolated vertices in G_{i+1} is at most the number of non-isolated vertices in the graph $G'_{i,1}$ obtained after the calls to *HOOK* and *CONTRACT* in the first iteration. In Lemma 4.3 we argued that the number of non-isolated vertices in $G_{i,1}$ is at most $4m/d_{i+1}$. The same argument shows, however, that the number of non-isolated vertices in $G'_{i,1}$ is also at most $4m/d_{i+1}$. □

We now show that the number of edges in G_{i+1}, the output graph of *DISTILL* is at most $200m/d_{i+1}$. This is the 'heart' of the correctness proof of *DISTILL*.

Theorem 4.6 *The number of edges in G_{i+1} is, w.v.h.p., at most $200m/d_i$.*

Proof: We say that an edge is introvert if one of its enpoints is an intovert vertex. As can be seen from the pseudo-code of *DISTILL*, all the edges of G_{i+1} are edges that become introvert during one of the iterations of *DISTILL*. A vertex becomes introvert, i.e., isolated, when all its edges become dead (or white). Both active and inactive vertices may become introvert. In *DISTILL*, we only mark active vertices as introvert. Thus, G_{i+1} will only include introvert edges that touch active introvert vertices.

For the purpose of the proof, we classify the edges of the graphs $G_{i,j}$ as either *inspected* or *uninspected*. Initially, all edges are uninspected. Intuitively, an edge becomes inspected when its marking, live or dead, 'influences' the evolution of the sequence $G_{i,0}, G_{i,1}, \ldots$ (This will be explained below in more precise terms.)

Each iteration of *DISTILL* involves a call to *HOOK*. Procedure *HOOK* is the only place in *DISTILL* in which edges become inspected. Procedure *HOOK* scans the edges of each active vertex in two phases. In the first phase it looks for live edges that connect each vertex to a vertex with a larger serial number. In the second stage, if it is performed, it settles for any live edge touching the vertex.

Let $v_1, v_2, \ldots$ be an arbitrary ordering of the active vertices of $G_{i,j}$. For the purpose of the proof, we consider a sequential simulation of the operation of *HOOK*. We also assume, for simplicity, that the edges of each vertex v are organized in two adjacency lists, one containing edges that connect v with vertices with larger serial numbers, and the other containing the rest of the edges. (This separation of each adjacency list into two adjacency lists is again only for the purpose of the proof.)

The sequential simulation of *HOOK* considers each vertex of $G_{i,j}$ in turn according to the ordering chosen. For each vertex u, it scans the first adjacency list of u until the first live edge is found, or until all the edges of the first adjacency list are exhausted. All edges encountered in this scan are now considered to be inspected. When the first scan from each vertex is finished, we perform a second scan, from some of the vertices. In the second scan, edges from the second adjacency list of each vertex are considered, again until the first live edge is found, or until all the edges of the second adjacency list are exhausted. Note that an edge (u, v), where $u < v$, can be inspected for the first time either when the first adjacency list of u is scanned, or when the second adjacency list of v is scanned.

An active vertex u is introvert if and only if both its adjacency lists were exhausted before finding a live edge touching it. When a vertex u becomes introvert, all its edges are therefore inspected. The number of edges that are inspected, in all the calls to *HOOK*, supplies, therefore, an upper bound on the number of edges that become introvert, and therefore on the number of edges in G_{i+1}.

Let e be an edge which is inspected for the first time during the j-th iteration of *DISTILL*. We claim that the probability that e is live when inspected is exactly α^j, independently of all other events. This is because in each iteration a live edge remains live with probability α. As e was not inspected before the j-th iteration, its marking did not have any effect on the hookings performed so far. The event 'e is live' is therefore independent of all other events in the graph $G_{i,j}$.

Each uninspected edge has a probability of at least α^j of becoming the last edge inspected in its adjacency list. (Note that the last inspected edge is always a live edge.) As we are inspecting $n_{i,j}$ adjacency lists (we consider the second adjacency list of each vertex as being a continuation of the first), the total number of edges inspected in the j-th iteration is bounded by a *negative binomial* random variable with parameters $n_{i,j}$ and α^j. The expected number of edges inspected for the first time during the j-th iteration is therefore at most $\alpha^{-j} n_{i,j}$. Furthermore, w.v.h.p, the number of edges inspected in the j-th iteration is at most $2\alpha^{-j} n_{i,j}$.

As $n_{i,j} \leq 4(3/4)^j \cdot m/d_i$, w.v.h.p., we get that the total number of edges inspected during all the iterations, and therefore the total number of introvert edges (of active vertices), is, w.v.h.p., at most $8m/d_i \cdot \sum_{j \geq 0} (3/4)^j \cdot (5/4)^j \leq 200m/d_i$. □

We finally show that G_{i+1} does not contain too many new connected components.

Theorem 4.7 *The number of new connected components in G_{i+1} is $O((m/d_i)/\log^4 n)$.*

Proof: Consider again the graphs $G_{i,k}$ and $G'_{i,k}$. The graph $G_{i,k}$ is the graph obtained after the last iteration of *DISTILL*. The graph $G'_{i,k}$ is the graph obtained from G_i, the input graph to *DISTILL*, using the same sequence of edge contractions but without any edge deletions. The output graph G_{i+1} is the subgraph of $G'_{i,k}$ composed of the edges that touch active introvert vertices of $G_{i,k}$.

The graph $G'_{i,k}$ contains no new connected components. (Every connected component of $G'_{i,k}$ corresponds to a connected component of G_i.) The edges removed from $G'_{i,k}$ to form G_{i+1} are edges that touch non-isolated vertices of $G_{i,k}$, and edges that touch inactive introvert vertices of $G_{i,k}$. We get, therefore, that each new connected component of G_{i+1} must contain at least one non-isolated vertex of $G_{i,k}$ or at least one inactive introvert vertex of $G_{i,k}$. The number of non-isolated vertices of $G_{i,k}$ is at most $4m/d_{i,k}$. The number of inactive vertices in $G_{i,k}$ is at most $2m/d_{i,k}$. Since $d_{i,k} = d_i \cdot (4/3)^k$, after $k = \lceil \frac{4}{\log(4/3)} \log\log n \rceil$ iterations, the number of non-isolated vertices in $G_{i,k}$ and the number of inactive vertices in $G_{i,k}$ are both $O((m/d_i)/\log^4 n)$. Therefore the number of new connected components in G_{i+1} is $O((m/d_i)/\log^4 n)$, as required. $\square$

This completes the correctness proof of *DISTILL*.

4.5 Implementation details and complexity of *DISTILL*

As in *REDUCE-VERTEX*, we assume at first that a virtual processor is allocated to each non-isolated vertex and to each edge, including white edges, of $G_{i,j}$. We then use the processor allocation scheme of [HZ96b] to obtain a concrete implementation of *DISTILL*.

Procedure *HOOK*, described in Figure 3, uses a hooking technique similar to the one used by Shiloach and Vishkin [SV82] and by Awerbuch and Shiloach [AS87]. It can be easily implemented to run in $O(\log d_{i,j})$ time, when a processor is allocated to each vertex and each edge of $G_{i,j}$. The first live edge in each adjacency list, or the first live edge going into a vertex with a larger serial number, can be found using the *pointer jumping* technique (see JáJá [JáJ92]).

Procedure *CONTRACT* contracts *all* the trees formed by *HOOK*. It uses the *edge list plugging* technique of [JM97]. If v is hooked on u, then the adjacency list of v is placed after the edge (u, v) in the adjacency list of u. The vertex u may, at the same time, be hooked on a vertex w, and so on. All these 'pluggings' are performed concurrently in *constant* time. Note, however, that the endpoint names of the edges in the adjacency lists formed may be out of date. In the above example, for example, the adjacency list of u would include edges whose first end point is v and not u.

Before activating *HOOK*, we update the endpoint names of edges that belong to adjacency lists of active vertices. Using pointer jumping, it is easy to inform all the edges in the adjacency list of an active vertex u that their first end point is u. This takes $O(\log d_{i,j})$ time. Each edge (u, v) then updates its opposite edge (v, u). All active endpoints are now updated.

The contraction operations may create self-loops and parallel edges. After updating the active endpoints of the edges, we mark self-loops for deletion. We do not delete parallel edges.

The running time of the j-th iteration of *DISTILL*, when a virtual processor is allocated to each non-isolated vertex and to each edge, is therefore $O(\log d_{i,j})$.

After the $k = O(\log\log n)$ iterations of *DISTILL* are complete, we have to construct the output graph G_{i+1}. The vertices of G_{i+1} are simply the vertices of $G_{i,k}$. The edges of G_{i+1} are the edges of G_i that touch vertices that were contracted into active introvert vertices, with their endpoints updated, of course. We now describe how the edge set of G_{i+1} is constructed.

In each iteration of *DISTILL* we contract a forest and then delete some of the edges. In practice, the edges that we want to delete are not deleted immediately, they are simply colored white. Most of the white edges are eventually deleted, while some remain in the representation of $G_{i,k}$. There are, in fact, two types of white edges in $G_{i,k}$. White edges of the first type are white edges that were colored white in one of the previous calls to *DISTILL*. These edges are not edges of the graph G_i (the input graph to the current invocation of *DISTILL*). White edges of the second type are edges of the input graph G_i that were marked as dead, and consequently colored white, in step 4 of one of the iterations of the current invocation of *DISTILL*.

The edges that we want to revive are the dead edges of the active intovert vertices. While some of these edges may still be present in $G_{i,k}$ (marked as dead and colored white) most of these edges were probably removed from the graph in one of the previous iterations.

To get hold of all the edges of E_{i+1} we run *DISTILL* backward. We perform again $k = O(\log\log n)$ iterations. In the j-th iteration we revive the dead edges that were removed from the adjacency lists in step 6 of the $(k-j)$-th iteration. For this purpose, each actual processor keeps track of the edges that it removed in each iteration. Note, however, that because of the edge list pluggings performed, if (u', v') is an edge that was removed in some iteration of (the forward run) of *DISTILL*, and if u' and v' were eventually contracted into the vertices u and v of $G_{i,k}$, then the edge (u', v') would now appear in the adjacency list of u. As no concurrent read and write operations were used when the edge (u', v') was deleted, no such concurrent operations will be performed when the edge (u', v') is revived.

In $G_{i,k}$, it is easy to let all the edges that touch active introvert vertices know that they touch active introvert vertices and that they should therefore be revived. (Note that all these edges are marked as dead and are colored white.) As no two edges that appear consecutively in the adjacency lists were removed from the lists at the same time unit, no two consecutive edges would be woken up at the same time unit. When an edge (u', v') is woken up, it can therefore inquire with its predecessor whether it belongs to the adjacency list of an active introvert vertex. Its predecessor can also inform it that it now appears in the adjacency list of u. (We assume that each adjacency list contains a dummy first edge that is never deleted.) Each edge (u', v') then updates its opposite (v', u') and informs it whether it belongs to an active introvert vertex.

When the $O(\log\log n)$ backward iterations are completed, all the edges of G_{i+1} were revived. All edges that were not informed that they belong to G_{i+1} are marked for deletion.

We end this section by bounding the overall complexity of *DISTILL*. *DISTILL* performs $O(\log\log n)$ forward iterations and $O(\log\log n)$ backward iterations. The complexity of the backward iterations is bounded by the complexity of the forward iterations. The time required by the j-th forward iteration is $O(\log d_{i,j})$. The total running time of *DISTILL*, when a virtual processor is allocated to each vertex and edge, is therefore

$$O(\sum_{j=0}^{k-1} \log d_{i,j}) = O(\sum_{j=0}^{k-1}(j + \log d_i)) = O((\log\log n)^2 + (\log\log n)\log d_i).$$

The total number of ***useful*** operations performed by all the virtual processors in *DISTILL* is

$$O(\sum_{j=0}^{k-1} \bar{m}_{i,j} \log d_{i,j}) = O(\sum_{j=0}^{k-1}(5/6)^j \frac{m}{d_i}(j + \log d_i)) = O(\frac{m}{d_i}\log d_i).$$

Procedure *DISTILL* requires $O(\log\log n)$ phases of processors allocation. Using the processor allocation scheme of [HZ96b] we turn this formulation into a concrete implementation, as explained in *REDUCE-VERTEX*.

procedure *REDUCE-EDGE*(G)

input: A graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges.

output: A graph $G' = (V', E')$, with at most $O(n \log^2 n)$ edges, obtained by contracting a forest F of G.

complexity: $O(\log n)$ time using $O(m/\log n + n \log n)$ processors.

1. Let $H = (V, E')$ be a subgraph of G obtained by including each edge of G independently with probability $\log^{-2} n$.
2. Find a spanning forest F of H using the third stage of the algorithm.
3. Construct each tree of F in G and output the resulting graph.

Figure 5: The second stage of the algorithm: reducing the number of edges.

5 Second stage: reducing the number of edges

The second stage of the algorithm is the simplest stage of the algorithm. It uses a *sampling* technique of Karger [Kar94] whose analysis was tightened by Klein and Tarjan [KT94]. This technique forms the basis of the recent randomized linear time sequential algorithm of [KKT95] for finding *minimum weight* spanning forests.

As our graph is unweighted, we only need the following simplified version of the sampling Lemma of Karger, Klein and Tarjan which also appears in Karger, Nisan and Parnas [KNP99]. The reader is referred to [KKT95] or [KNP99] for a proof of the lemma. Throughout this section, all graphs may contain parallel edges.

Lemma 5.1 *Let $G = (V, E)$ be a graph with n vertices and m edges and let $0 < p < 1$. Let $H = (V, E')$ be a random subgraph of G obtained by including each edge of E in E' independently with probability p. Then, the probability that the number of edges of E that connect vertices in different connected components of the subgraph $H = (V, E')$ exceeds $2n/p$ is at most $e^{-n/4}$.*

The input to the second stage is a graph $G = (V, E)$ with n vertices and m edges. The second stage begins by constructing a random subgraph $H = (V, E')$ of G. Each edge of G is independently included in H with probability $p = \log^{-2} n$. The graph H has, w.v.h.p., at most $m' = O(m/\log^2 n)$ edges. It follows from lemma 5.1 that the number of edges of G that connect vertices from different connected components of H is, w.v.h.p., at most $O(n/p) = O(n \log^2 n)$. Using procedure *SPAN-FOREST*, that constitutes the third stage of our algorithm, we find a spanning forest F of H. The running time of this stage is $O(\log n)$ using $O((m/\log^2 n + n) \log n)$ processors. The forest F is a, not necessarily spanning, forest of G. By contracting each tree in F we thus obtain, w.v.h.p, a graph G' with at most n vertices and $O(n \log^2 n)$ edges. All these operations can be done in $O(\log n)$ time using $O(m/\log n + n \log n)$ processors.

Procedure *REDUCE-EDGE*, which constitutes the second stage of our algorithms is described in Figure 5. Note that in the complete algorithm, the input to *REDUCE-EDGE* is the graph $G^{(2)}$ produced by *REDUCE-VERTEX*. As $G^{(2)}$ contains only $O(m/\log^4 n)$ vertices, the number of edges in the graph returned is only $O(m/\log^2 n)$ and the number of processors required is only $O(m/\log n)$. The $O((m + n)/\log n)$ processors available are therefore enough.

procedure *SPAN-FOREST*(G)

input: A graph $G = (V, E)$ with n vertices and m edges.

output: A spanning forest of G.

complexity: $O(\log n)$ time using $O(m \log n)$ processors and $O(m \log n)$ space.

```
G_0 ← G ; d_0 ← 2
for i ← 0 to k − 1 (where k = O(log log n)) do
{
    1. Let d_{i+1} ← d_i^{4/3}.
    2. Mark non-isolated vertices of degree at most d_{i+1} as active.
       Mark all other non-isolated vertices as inactive.
    3. parent ← WALK-HOOK(G_i, d_i)
    4. G_{i+1} ← CONTRACT(G_i, parent)
}
```

The forest F is composed of all the edges of G marked as forest edges during the calls to *WALK-HOOK*.

Figure 6: The third stage of the algorithm: finding a spanning forest.

6 Third stage: finding a spanning forest

The third stage is a complete algorithm for finding a spanning forest of a graph $G = (V, E)$ with n vertices and m edges. It runs in $O(\log n)$ time using $O(m \log n)$ processors and $O(m \log n)$ space. (We assume, for simplicity, that the graph G does not contain isolated vertices; Isolated vertices can be easily removed in a preprocessing step.) The number of processors and the space can both be easily reduced to $O(m)$, as described in section 7, but the less efficient implementation is sufficient for our purposes. (Recall that this algorithm is going to be run on a graph with $O(m/\log^2 n)$ vertices and edges.) Procedure *SPAN-FOREST*, which constitutes the third stage of our algorithm, is described in Figure 6.

Procedure *SPAN-FOREST* is similar in its structure to procedure *REDUCE-VERTEX*. It also performs $k = O(\log \log n)$ iterations of hookings and contractions. We again define a sequence d_i of degree bounds. Only vertices of degree at most d_{i+1} are hooked in the i-th iteration. There are several important differences, however. *REDUCE-VERTEX* only had to reduce the number of vertices by a poly-logarithmic factor. *SPAN-FOREST* has to contract *all* the connected components of the graph, i.e., reduce the number of non-isolated vertices in the graph by a factor of n. As we would again like to finish in $O(\log \log n)$ iterations, the sequence d_i is now doubly exponential and each iteration now reduces the number of vertices in the graph by much more than a constant factor. On the other hand, we do not have to reduce the number of edges at each stage as we are allowed, this time, to perform $O(m \log^2 n)$ operations.

The hookings in *SPAN-FOREST* are performed using *WALK-HOOK*, a procedure that uses relatively short random walks to generate a forest that contains a relatively small number of trees. The trees in the forest constructed by *WALK-HOOK* are contracted using a procedure named *CONTRACT*. Procedure *CONTRACT* used here is basically identical to procedure *CONTRACT* used in the first phase of our algorithm.

It is convenient to assume that the degree of each non-isolated vertex in the input graph G is at least 2. To ensure it, we simply add a self-loop to all the vertices in G of degree 1. The obtained graph G_0 is the input to the first iteration of *SPAN-FOREST*.

The i-th iteration of *SPAN-FOREST* receives a graph G_i and returns a graph G_{i+1} obtained by contracting a forest in G_i. The forest in G_i is constructed by *WALK-HOOK* and it is contracted by *CONTRACT*. Each edge of G_i corresponds to an edge of the original graph G. After constructing a forest F_i in G_i, *WALK-HOOK* marks the edges in G that correspond to the edges of F_i as forest edges. The set of all edges in G marked as forest edges during the $O(\log\log n)$ iterations of *SPAN-FOREST* would constitute a spanning forest of G.

In the implementation of *SPAN-FOREST* we do not use the processor allocation mechanism of [HZ96b] which was used in the implementation of *REDUCE-VERTEX* and *DISTILL*. We will explicitly describe the allocation of processors to the various tasks.

Before the first iteration of *SPAN-FOREST*, we allocate $\log n$ processors to each edge of the input graph G. (Recall that we assume that the input graph G does not contain isolated vertices.) The $\log n$ processors allocated to an edge stay with that edge during all the $O(\log\log n)$ iterations. Thus, a vertex of G_i of degree d would have $d \log n$ processors allocated to it. Throughout the operation of *SPAN-FOREST* we do *not* remove self-loops and parallel edges. A contraction operation, therefore, does not reduce the number of edges in the graph and no processors are lost. (In the proposed implementation, we use the adjacency lists of the graphs G_i as a means for allocating processors to the vertices of the graph. We could have used separate linked lists for the processors allocated to each vertex, in which case we could have removed self-loops and parallel edges from the adjacency lists. We find it a bit simpler to use the adjacency lists for both purposes.) To avoid confusion, we mention here that self-loops and parallel edges (of active vertices) are temporarily removed by *WALK-HOOK*.

As $d_0 = 2$ and $d_{i+1} = d_i^{4/3}$, for $1 \le i < k$, it is easy to see that $d_i = 2^{(4/3)^i}$. Vertices whose degree, at the start of the i-th iteration, is at most d_{i+1} are marked as active. All the other vertices are marked as inactive (inactive).

Procedure *WALK-HOOK* hooks active vertices on neighboring active or inactive vertices. A description of procedure *WALK-HOOK* is given in Figure 7 (it is also described in detail in the next subsection). Each tree in G_i created by *WALK-HOOK* contains either an inactive vertex, or at least $d_i^{1/3}$ active vertices, or all the vertices of a complete connected component of G_i. The graph G_{i+1} is obtained by contracting this forest. We can now show:

Lemma 6.1 *The degree of each non-isolated vertex in G_i is at least d_i.*

Proof: We prove the claim using induction. The claim holds for $i = 0$, as each non-isolated vertex of G_0 has degree at least 2, after we added the self-loops, and $d_0 = 2$.

Suppose that the claim holds for G_i. The degree of each non-isolated active vertex in G_i is then at least d_i. The degree of each inactive vertex in G_i is, by definition, at least d_{i+1}. The call to *WALK-HOOK* creates a forest in G_i in which each tree contains, either an inactive vertex, or at least $d_i^{1/3}$ active vertices, or all the vertices of a complete connected component of G_i. The graph G_{i+1} is the contraction of this forest.

As we do *not* remove self-loops and parallel edges after the contraction, the degree of each vertex in G_{i+1} is equal to the sum of the degrees of the vertices that were contracted into it (each self-loop contributes 2 to the degree of the vertex it touches).

The vertices of G_{i+1} are of three types, corresponding to the three types of trees in the forest constructed by *WALK-HOOK*. If a tree contains an inactive vertex then the degree of this inactive vertex alone is at least d_{i+1}. If the tree contains at least $d_i^{1/3}$ active vertices, then as the degree of each one of these vertices is at least d_i, the sum of the degrees in this tree is at least $d_i^{1/3} \cdot d_i = d_i^{4/3} = d_{i+1}$. Finally, if a tree contains all the vertices of a complete connected component of G_i then the vertex in G_{i+1} obtained by contracting this tree is isolated. This completes the proof of the Lemma. □

Let k be the minimal integer for which $d_k = 2^{(4/3)^k} \geq 2m$. Note that $(4/3)^k = O(\log n)$ and $k = O(\log\log n)$. After k iterations, all the vertices of G_k are therefore isolated.

As mentioned, each edge of G_i corresponds to an edge of $G_0 = G$. The edges of G_0 that correspond to edges of $G_0, G_1, \ldots, G_{k-1}$ that were chosen to one of the forests constructed by *WALK-HOOK* form a spanning forest of G_0, as required.

As we shall see, the i-th iteration of *SPAN-FOREST* takes $O(\log d_i + \log\log n)$ time. The total running time of *SPAN-FOREST* is therefore:

$$O(\sum_{i=0}^{k-1}(\log d_i + \log\log n)) = O(\sum_{i=0}^{k-1}((4/3)^i + \log\log n)) = O((4/3)^k + k \cdot \log\log n) = O(\log n).$$

SPAN-FOREST uses $O(m \log n)$ processors and $O(m \log n)$ space.

In the next subsection we describe procedure *WALK-HOOK* and prove that it works properly.

6.1 Procedure *WALK-HOOK*

Procedure *WALK-HOOK* is described in Figure 7. It receives a graph $G' = (V', E')$ and a degree bound s. The graph G' is represented using adjacency lists. The degree of each non-isolated vertex in G' is at least s. Vertices of degree at most $d = s^{4/3}$ are marked as active. All other vertices are marked as inactive. Each edge of G' has $\theta(\log n)$ processors allocated to it. Each non-isolated vertex of G' has, as a result, at least $s' = c \cdot s \log n$ processors allocated to it. (The constant c will be discussed later.) The graph G' will usually contain self-loops and parallel edges.

The goal of *WALK-HOOK*, as mentioned before, is to construct a forest in G' in which each tree satisfies at least one of the following three conditions: (i) it contains an inactive vertex; (ii) it contains at least $s^{1/3}$ active vertices; (iii) it contains all the vertices of a complete connected component of G'. The forest will be constructed by hooking active vertices on active or inactive neighbors of theirs. (This is a major difference between our algorithm and previous algorithms that use random walks. Previous algorithms may have hooked vertices on vertices that are not their neighbors.) The forest constructed by *WALK-HOOK* is described using an array of pointers. For each vertex v of G', $\texttt{parent}[v]$ is the parent of v in the forest formed. If v is a root of a tree then $\texttt{parent}[v] = v$. (In the sequel, we will not always refer to the $\texttt{parent}$ pointers explicitly.)

To carry out its task, *WALK-HOOK* uses random walks. The graph G' may, as stressed, contain self-loops and parallel edges. Before starting the random walks, the parallel edges and self-loops of the active vertices are temporary removed, as the random walks are not guaranteed to have the desired properties when parallel edges and self-loops are present. The exact way in which this is done is described, along with many other implementation details, in Section 6.4. It is important to note, however, that each vertex has at least $s' = c \cdot s \log n$ processors allocated to it even when the parallel edges and self-loops are temporarily removed. These processors are used to construct the array `walk`, and the other arrays used by *WALK-HOOK* (see Figure 7).

Procedure *WALK-HOOK* starts by finding all the active vertices that have at least one inactive neighbor. Such vertices are marked as near-inactive and they can immediately be hooked on one of their inactive neighbors. As the degree of each active vertex is at most d, these operations can be easily performed in $O(\log d)$ time. All the other active vertices are marked as inner-active. (The need for differentiating between procedure *WALK-HOOK*(G', s)

input: A graph $G' = (V', E')$ and a degree bound s. All vertices of degree at most $d = s^{4/3}$ are marked as active. All other vertices are marked as inactive.

output: A pointer array parent that defines a forest in G' in which each tree contains, w.h.p., either at least $s^{1/3}$ vertices, or an inactive vertex, or all the vertices of a connected component of G'.

complexity: $O(\log s + \log\log n)$ time, w.h.p., when $\log n$ processors are allocated to each edge of G'.

Let $d = s^{4/3}$ and $s' = c \cdot s \log n$, where c is a constant (see text).

Each active vertex that has an inactive neighbor is marked as near-inactive. All other active vertices are marked as inner-active. Hook every near-inactive vertex v on one of its inactive neighbors.

Construct the walk array: For every inner-active vertex v and for each $1 \le t \le s'$, such that t is not divisible by s, let $\mathtt{walk}[v, t]$ point to a cell $[u, t+1]$, where u is a neighbor of v chosen uniformly at random. The tree containing the cell $[v, t]$ is denoted by $T_{[v,t]}$.

Identify all the small enough trees in the walk array, i.e., trees that contain at most $4s^2d$ cells. All subsequent operations are only performed on cells contained in small enough trees.

For every inner-active vertex v and every $1 \le t \le s'$, let $\mathtt{max}[v, t]$ be the maximal cell in the tree $T_{[v,t]}$. (The cells are ordered lexicographically, see text.)

Make $\mathtt{max}[v, t]$ the root of the tree $T_{[v,t]}$ by reversing the direction of some of the pointers in the tree. Let $\mathtt{walk}'$ be the array of the updated pointers.

Let $\mathtt{dist}[v, t]$ be the distance from the cell $[v, t]$ to the root $\mathtt{max}[v, t]$ of its tree.
Let $\mathtt{label}[v, t] = \langle \mathtt{max}[v, t], \mathtt{dist}[v, t] \rangle$.

Let $\mathtt{max}[v] = \max_{1 \le t \le s'} \mathtt{max}[v, t]$ and $\mathtt{label}[v] = \max_{1 \le t \le s'} \mathtt{label}[v, t]$.
Let $\mathtt{time}[v]$ be a time t for which $\mathtt{label}[v] = \mathtt{label}[v, t]$.

First hooking phase: Every inner-active vertex v is hooked on $\mathtt{ver}(\mathtt{walk}'[v, \mathtt{time}[v]])$, where $\mathtt{ver}([v, t]) = v$. (In other words, let $\mathtt{parent}[v] \leftarrow \mathtt{ver}(\mathtt{walk}'[v, \mathtt{time}[v]])$.) Each active vertex in G' now belongs to a tree in G' defined by the parent pointers. If r is the root of such a tree in G', we let T_r denote the tree rooted at r.

For each active vertex v, if v belongs after the first hooking phase to a tree that contains at most $s^{1/3}$ vertices, let $\mathtt{root}[v]$ be the root of the tree in G' to which v belongs. If v is contained in a larger tree, then $\mathtt{root}[v]$ is assigned a number larger than the serial numbers of all the vertices.

Second hooking phase: If T_r is a tree in G' formed by the first hooking phase which is composed of less than $s^{1/3}$ active vertices, then scan all the neighbors of the vertices of T_r and find among them the vertex u for which $\mathtt{root}[u]$ is maximal. If $u \notin T_r$, then let v be a neighbor of u in T_r. Make v the root of T_r, by reversing some of the pointers in the tree, and hook v on u.

Figure 7: Procedure *WALK-HOOK*.

near-inactive and inner-active vertices will become clear in Section 6.4 when the actual implementation of *WALK-HOOK* is discussed.)

Procedure *WALK-HOOK* proceeds by launching $\log n$ random walks, each of length s, from each one of the inner-active vertices of G'. The purpose of the random walks is the construction of large *virtual neighborhoods*. The virtual neighborhood of v is denoted by $C(v)$. All the vertices in $C(v)$ must lie in the connected component of v, and if $u \in C(v)$, then we must also have $v \in C(u)$. The virtual neighborhood $C(v)$ will contain the vertices reached by the random walks from v, as well as the vertices whose random walks reached v, and perhaps some other vertices. As we shall see, the size of each virtual neighborhood would be, w.h.p., at least $s^{1/3}$ (unless it contains an inactive vertex or all the vertices of a connected component of G').

Karger, Nisan and Parnas [KNP99] were the first to use random walks in parallel connectivity algorithms. In their algorithm, each vertex v is hooked on the largest vertex at virtual distance at most two from v, i.e., on the vertex with the largest serial number in the set $C(C(v)) = \cup_{u \in C(v)} C(u)$. It is not difficult to show that if each virtual neighborhood contains at least $s^{1/3}$ vertices, then the number of trees formed by these hookings is at most $n/s^{1/3}$. Some of the trees formed may, however, be quite small. In [HZ96b], we use a slight variation of this hooking method. A vertex v first attempts to hook on a vertex from $C(v)$ with a larger serial number. Only if such a vertex is not found, v attempts to hook on a larger vertex in $C(C(v))$. If such a vertex is also not found, then v becomes a root. It is not difficult to show that each tree formed by these hookings contains at least $s^{1/3}$ vertices. Both these hooking methods suffer from the problem that vertices are hooked on *virtual* neighbors, and not necessarily on *actual* neighbors of theirs. These hooking methods need to be modified, therefore, if a spanning forest is to be found.

Before describing the hooking method used by us here, we have to describe the way in which the random walks are implemented and explain how the virtual neighborhoods are constructed. This is done next. The hooking method employed by *WALK-HOOK* is then described in Section 6.3.

6.2 The random walks

A random walk of length s from a vertex $v = v_0$ is obtained by picking, uniformly at random, an edge (v_0, v_1) adjacent to v_0, then an edge (v_1, v_2) adjacent to v_1 and so on, until a path of length s is obtained. A random walk is not necessarily simple as it may return to a previously visited vertex.

Let $H = (V, E)$ be a *simple* connected graph on n vertices, let $v \in V$ be a vertex of H and let $w \leq n$. Barnes and Feige [BF96], proving a conjecture of Linial, showed that the expected number of steps taken by a random walk that starts at v until w distinct vertices are visited is $O(w^3)$. The following Lemma follows immediately from this result using Markov's inequality.

Lemma 6.2 *There exists a constant $a > 0$ such that the probability that a random walk of length s that starts at some vertex v of a simple graph $H = (V, E)$ contains at least $a \cdot s^{1/3}$ distinct vertices, or all the vertices in the connected component of v, is at least $1/2$.*

Proof: Suppose that the expected number of steps needed to encounter w distinct vertices is at most bw^3. Consider a random walk of length s. If the probability that this path encounters at least $(s/2b)^{1/3}$ distinct vertices is less than $1/2$, then the expected length of a path needed to visit $(s/2b)^{1/3}$ vertices is greater than $s/2$. But this is a contradiction as this expectation is at most $b \cdot (s/2b) = s/2$. The Lemma therefore holds with $a = (1/2b)^{1/3}$. □

To obtain, with high probability, neighborhoods of size at least $s^{1/3}$, *WALK-HOOK* starts $\Theta(\log n)$ random walks of length $\Theta(s)$ from each inner-active vertex v in G'. A random walk from an inner-active vertex v is stopped if it reaches a near-inactive vertex. The random walks are therefore performed on the subgraph of G' induced by the active vertices. As self-loops and parallel edges touching active vertices are removed, the random walks are performed, as required, on a simple graph.

To be more precise, we start from each inner-active vertex $c \log n$ random walks of length $a^{-3}s$, where a is the constant of Lemma 6.2 (as mentioned, a random walk is stopped if it reaches a near-inactive vertex). The probability that none of the walks initiated from a particular vertex reaches a near-inactive, nor visits at least $s^{1/3}$ vertices or all the vertices in the connected component of the start vertex, is at most n^{-c}. This failure probability can be reduced to $2^{-\Omega(2^{\sqrt{\log n}})}$, which is $o(n^{-c})$, for any $c > 0$, as explained in Section 7. For simplicity, we do not mention the constants a^{-3} and c explicitly from now on.

The parallel implementation of the random walks follows the work of Karger *et al.* [KNP99]. The random walks from all the inner-active vertices are chosen simultaneously. For each inner-active vertex v and each time $1 \leq t \leq s$, we choose the vertex that will *follow* v in the random walks if v is reached at time t. To implement this idea we use, following Karger *et al.* [KNP99], an array of pointers called `walk`.

The `walk` array contains a cell $[v, t]$ corresponding to each active vertex v and each time $1 \leq t \leq s$. For every inner-active vertex v and every $1 \leq t < s$, $\texttt{walk}[v, t]$ is set to point to a cell $[u, t + 1]$, where u is a randomly chosen neighbor of v. If v is a near-inactive vertex, or $t = s$, then $\texttt{walk}[v, t]$ points to the cell $[v, t]$ itself. The actual construction of the array `walk` is described in Section 6.4.

The random walk $v = v_1, v_2, \ldots, v_s$ started at an inner-active vertex v is defined as follows: $v_{t+1} = \texttt{ver}(\texttt{walk}[v_t, t])$, where $\texttt{ver}([v, t]) = v$ is the vertex to which the cell $[v, t]$ belongs. The random walk is effectively halted if v_t, for some $1 \leq t < s$ is a near-inactive vertex. If that happens then $v_t = v_{t+1} = \ldots = v_s$.

It is easy to see that the random walk started from an inner-active vertex v is indeed an unbiased random walk (subject to the condition that it is halted when it reaches a near-inactive vertex). At each time unit, the next vertex to be visited is chosen uniformly at random from among all the neighbors of the current vertex visited and the decisions at different time units are independent.

It should also be clear, however, that the random walks chosen from two different inner-active vertices v_1 and v_2 are *not* necessarily independent. If, for example, both these random walks reach the same vertex at the same time unit, then they would coalesce into the same path. Luckily, the random walks are not required to be independent.

We in fact use $L = \Theta(\log n)$ `walk` arrays to implement $\Theta(\log n)$ independent random walks of length s from each inner-active vertex. As each `walk` array is chosen independently, random walks defined by different `walk` arrays are clearly independent. In the actual implementation, we find it more convenient to have just one `walk` array in which each row is of length $s' = \Theta(s \log n)$. The cells $[v, t]$, where t is divisible by s are set to point to themselves. (Previous algorithms could have replaced the $\log n$ random walks of length s by one random walk of length $s \log n$. We, however, cannot do the same.)

The pointers of the walk array define a forest whose vertices are the cells $[v, t]$. We let $T_{[v,t]}$, where v is an active vertex and $1 \leq t \leq s'$, denote the tree in the `walk` array that contains the cell $[v, t]$. We would have liked to define the virtual neighborhood $C(v)$ of an active vertex v as the union of the vertices of the cells contained in the trees $T_{[v,t]}$, for $1 \leq t \leq s'$. The virtual neighborhood of an inner-active vertex v would then contain vertices from more than $\Theta(\log n)$ independent random walks starting at v and would therefore satisfy, w.h.p., at least one the the three required conditions. This definition also satisfies the requirement that $v \in C(u)$ if and only if $u \in C(v)$. The problem is that virtual neighborhoods defined in this way may be ***too big*** and finding the largest vertex in each neighborhood would be too time consuming. Luckily, we can show that most of the trees $T_{[v,t]}$ are not too big. The virtual neighborhood of v is therefore defined to be the union of all the trees $T_{[v,t]}$ that are ***small enough***. A more precise description follows.

Let $[v, t]$ be a cell of the `walk` array. We let $\mathcal{T}_{[v,t]}$ be the random variable giving the size of the tree $T_{[v,t]}$. We let $\mathcal{T}'_{[v,t]}$ be the random variable giving the size of the subtree $T'_{[v,t]}$ of the tree $T_{[v,t]}$ rooted at $[v, t]$. For convenience, the size of a tree is taken to be the number of edges in it.

Let $p = \langle [v_t, t], [v_{t+1}, t+1], \ldots, [v_{t'}, t'] \rangle$, where $v_t = v$, and $t' = s$ or $v_{t'}$ is a near-inactive, be a possible path in a `walk` array of length s that corresponds to the graph G' (this simply means that $v_t, v_{t+1}, \ldots, v_{t'}$ is a path in G'). Let $[u, r]$ be a cell in this `walk` array. We let $E[\mathcal{T}_{[u,r]} \mid p]$ denote the expected value of $\mathcal{T}_{[u,r]}$ given that p is the path chosen from $[v_t, t]$, i.e., the expected value of $\mathcal{T}_{[u,r]}$ given that $\mathtt{walk}[v_i, i] = v_{i+1}$ for $t \le i < t'$. The conditional expectation $E[\mathcal{T}'_{[u,r]} \mid p]$ is defined similarly.

The following two lemmas also appear in [HZ96b]. Their proofs are brought here for the sake of completeness. They would be used to show that many trees in the walk array would be small enough.

Lemma 6.3 *Let p be a possible path in the* `walk` *array and let $[u, r]$ be a cell which is either the first cell in p, or not on p. Then, $E[\mathcal{T}'_{[u,r]} \mid p] \le \deg(u) \cdot r$.*

Proof: We prove the Lemma by induction on r. If $r = 1$ then the claim is obvious. Suppose therefore that the claim holds for $r - 1$. We show that it also holds for r.

Let $w_1, \ldots, w_l$ be the active neighbors of u in G', where $l \le \deg(u)$. The subtree $T'_{[u,r]}$ is composed of the subtrees $T'_{[w_j, r-1]}$, where $1 \le j \le l$, for which $\mathtt{walk}[w_j, r-1] = u$. The probability that $\mathtt{walk}[w_j, r-1] = u$ is 0 if w_j is a near-inactive vertex or if it is contained in the path p and is $1/\deg(w_j)$ otherwise. We get therefore that

$$E[\mathcal{T}'_{[u,r]} \mid p] \le \sum_{j=1}^{l} \frac{1}{\deg(w_j)} \cdot (E[\mathcal{T}'_{[w_j, r-1]} \mid p] + 1) \; .$$

Recall that the size of a tree is taken to be the number of edges in it. Using the induction hypothesis we get that $E[\mathcal{T}'_{[w_j, r-1]} \mid p] \le \deg(w_j) \cdot (r-1)$ and therefore

$$E[\mathcal{T}'_{[u,r]} \mid p] \le \sum_{j=1}^{l} \frac{1}{\deg(w_j)} \cdot (\deg(w_j) \cdot (r-1) + 1) \le l \cdot r \le \deg(u) \cdot r \; .$$

This completes the proof of the Lemma. □

We are now able to bound the expected size of the trees of the `walk` array.

Lemma 6.4 *Let v be an active vertex in G', and let $[v, t]$ be a cell in a* `walk` *array of length s that corresponds to the graph G'. Then, $E[\mathcal{T}_{[v,t]}] \le ds^2$.*

Proof: By definition, the degree of the active vertex v is at most d. Let $p = \langle [v_t, t], [v_{t+1}, t+1], \ldots, [v_{t'}, t'] \rangle$, where $v_t = v$, be a possible path in the `walk` array, i.e., $\mathtt{walk}[v_j, j] = v_{j+1}$, for every $t \le j < t'$. This means that $\langle v_t, v_{t+1}, \ldots, v_{t'} \rangle$ is a path in the graph G'. We show that the expected size of the tree $T_{[v,t]}$ given that p is the path chosen from $[v, t]$ is at most ds^2. As this holds for any such path p, the unconditional expectation is also at most ds^2.

Let d_j be the number of the inner-active neighbors of the active v_j in the path p, $d_j \le \deg(v_j) \le d$, $t \le j \le t' \le s$. Let $u_{j,1}, \ldots, u_{j, d_{j+1}-1}$, for $t \le j < t'$, be the inner-active neighbors of v_{j+1}, other than v_j. It is easy to see that the tree $T_{[v,t]}$ is composed of the path p, the subtree $T'_{[v,t]}$ rooted at $[v, t]$, and from the subtrees $T'_{[u_{j,k}, j]}$, where $t \le j < t'$ and $1 \le k < d_{j+1}$, for which $\mathtt{walk}[u_{j,k}, j] = v_{j+1}$. Note that $\mathtt{walk}[u, j]$ can be v_{j+1} only if u is an inner-active vertex. The probability that $\mathtt{walk}[u_{j,k}, j] = v_{j+1}$ is $1/\deg(u_{j,k})$, since $u_{j,k}$ is an inner-active. We get therefore that

$$E[\mathcal{T}_{[v,t]} \mid p] \le (t' - t) + E[\mathcal{T}'_{[v,t]} \mid p] + \sum_{j=t}^{t'-1} \sum_{k=1}^{d_{j+1}-1} \frac{1}{\deg(u_{j,k})} \cdot (E[\mathcal{T}'_{[u_{j,k}]} \mid p] + 1) \; .$$

As $[v,t]$ is the first vertex of the path p, we get, by the last remark in the proof of Lemma 6.3 that $E[\mathcal{T}'_{[v,t]}|p] \le \deg(v)\cdot t \le d\cdot t$. As each of the vertices $u_{j,k}$ is not on the path p, we get, again using Lemma 6.3 that $E[\mathcal{T}'_{[u_{j,k},j]} \mid p] \le \deg(u_{j,k})\cdot j$ and therefore

$$
\begin{aligned}
E[\mathcal{T}_{[v,t]} \mid p] &\le (t'-t) + d\cdot t + \sum_{j=t}^{t'-1} \sum_{k=1}^{d_{j+1}-1} \frac{1}{\deg(u_{j,k})}\cdot(\deg(u_{j,k})\cdot j + 1) \\
&\le (t'-t) + d\cdot t + (d-1)\cdot \sum_{j=t}^{t'-1} (j+1) \le dt'^2 \le ds^2 .
\end{aligned}
$$

This completes the proof of the Lemma. □

It immediately follows from this Lemma, using Markov's inequality, that the probability that the size of a tree $T_{[v,1]}$ in the `walk` array is greater than $4ds^2$, is at most $1/4$. Note, on the other hand, that such a tree $T_{[v,1]}$ contains a random walk of length s that starts at v (which is halted if it reaches a near-inactive vertex). We saw earlier that the probability that this walk fails to visit at least $s^{1/3}$ vertices, does not reach a near-inactive vertex, and does not visits all the vertices in the connected component of v is at most $1/2$. The probability that either one of these bad events happens is therefore at most $3/4$. The same holds of course for the trees $T_{[v,s+1]}, T_{[v,2s+1]}, \ldots, T_{[v,(L-1)s+1]}$, where $L = \Theta(\log n)$. Thus, w.h.p., at least one of the trees $T_{[v,t]}$, for $1 \le t \le s'$ is small enough, yet it contains enough vertices.

The virtual neighborhood $C(v)$ of an inner-active vertex v is now defined as follows: $u \in C(v)$ if and only if there exist $1 \le t_1, t_2 \le s'$ such that $[u,t_1]$ and $[v,t_2]$ belong to the same tree $T_{[u,t_1]} = T_{[v,t_2]}$, and this tree is of size at most $4s^2d$. We have proved:

Lemma 6.5 *With high probability, the virtual neighborhood $C(v)$ of each active vertex v satisfies at least one of the following three conditions: (i) $C(v)$ contains at least $s^{1/3}$ vertices; (ii) $C(v)$ contains a near-inactive vertex; (iii) $C(v)$ contains all the vertices of the connected component of v.*

6.3 The hookings performed by *WALK-HOOK*

After describing the way the random walks are performed, we can now describe the hookings performed by *WALK-HOOK*. These hookings try to emulate the hookings performed in [HZ96b]. There, every active vertex v tries to hook itself on a virtual neighbor, i.e., on a vertex from $C(v)$, with a larger serial number. If such a neighbor does not exist then v tries to hook itself on a larger vertex from $C(C(v))$. If such a vertex does not exist either, then v becomes a root of a tree. The hookings performed by *WALK-HOOK* here try to achieve the same effect, though here each vertex can only be hooked on one of its actual neighbors.

The first step is easy. Each near-inactive vertex v is hooked on one its inactive neighbors. As the degree of each active vertex is at most d, this can be easily performed in $O(\log d)$ time. We are now left with the task of hooking the inner-active vertices. For this purpose we use the random walks and the `walk` array.

For each active vertex v, the `walk` array contains the cells $[v,t]$, for $1 \le t \le s'$, where $s' = \Theta(s \log n)$. Each such cell has a processor allocated to it. The `walk` pointers define a forest whose vertices are these cells. The tree containing the cell $[v,t]$ is denoted by $T_{[v,t]}$. The roots of the trees $T_{[v,t]}$ are cells corresponding to near-inactive vertices and cells with $t = s, 2s, \ldots s'$. Trees $T_{[v,t]}$ of size at most $4ds^2$ are said to be *small enough*. As will be explained in the next subsection, all the small enough trees can be identified in $O(\log s)$ time.

Each vertex in G' has an associated serial number. We want the serial numbers of the near-inactive vertices to be larger than those of the inner-active vertices. To achieve this we temporarily add n to the serial number of each near-inactive vertex.

We define a lexicographic ordering on the cells as follows: $[v_1, t_1] \le [v_2, t_2]$ iff $v_1 < v_2$ or $v_1 = v_2$ and $t_1 < t_2$. For every cell $[v,t]$ in a small enough tree, we let $\texttt{max}[v,t]$ be the maximal cell in the tree $T_{[v,t]}$. As near-inactive vertices are considered to be larger than all the inner-active vertices, the maximum cell in a tree whose root is a cell of a near-inactive vertex, is the root. (Also note that if a tree contains a cell that belongs to a near-inactive vertex, then this cell is the root of the tree.) The description of the way in which the maximal cell in each small enough tree is found, as well as many other implementation details, is deferred to the next subsection.

We now make $\texttt{max}[v,t]$ the root of the tree $T_{[v,t]}$. This is done by reversing the direction of some of the walk pointers defining the tree. The pointers whose direction should be reversed are precisely the pointers on the path from the cell $\texttt{max}[v,t]$ to the current root of the tree $T[v,t]$. We denote by $\texttt{walk}'[v,t]$ the new pointer attached to the cell $[v,t]$. Note that if $\texttt{walk}'[v,t] = [v',t']$ then either $t' = t+1$, if $\texttt{walk}[v,t] = [v',t']$, or $t' = t-1$, if $\texttt{walk}[v',t'] = [v,t]$ and the direction of the pointer from $[v',t']$ to $[v,t]$ was reversed. We retain the notation $T_{[v,t]}$ for the tree in the array walk' that contains the cell $[v,t]$.

We let $\texttt{dist}[v,t]$ be the distance from a cell $[v,t]$ to the root $\texttt{max}[v,t]$ of its tree. We also let $\texttt{label}[v,t] = \langle \texttt{max}[v,t], \texttt{dist}[v,t] \rangle$. We say that $\langle [v_1,t_1], d_1 \rangle < \langle [v_2,t_2], d_2 \rangle$ if and only if $[v_1,t_1] < [v_2,t_2]$ or $[v_1,t_1] = [v_2,t_2]$ and $d_1 > d_2$ (note that a label becomes *larger* as its distance component becomes *smaller*).

The above definitions apply to cells contained in small enough trees. If the tree in which the cell $[v,t]$ is contained is not small enough, we define $\texttt{max}[v,t] = 0$ and $\texttt{label}[v,t] = 0$, where 0 is considered to be less than any cell or label.

Finally, we let $\texttt{max}[v] = \max_{1 \le t \le s'} \texttt{max}[v,t]$ and $\texttt{label}[v] = \max_{1 \le t \le s'} \texttt{label}[v,t]$. We also let $\texttt{time}[v]$ be a time t for which $\texttt{label}[v] = \texttt{label}[v,t]$ (this t is not necessarily unique). It is not difficult to see that $\texttt{label}[v] = \langle \texttt{max}[v], \texttt{dist}[v, \texttt{time}[v]] \rangle$. It will be shown later that all these operations can be performed in $O(\log s + \log\log n)$ time using the processors allocated to the cells.

The virtual neighborhood $C(v)$ of an active vertex v is defined to contain all the vertices that own at least one of the cells in $\cup_{1 \le t \le s'} T_{[v,t]}$, where the union here is only on small enough trees. It is easy to check that $\texttt{ver}(\texttt{max}[v])$ is the largest vertex in the virtual neighborhood $C(v)$ of v.

If $u = \texttt{ver}(\texttt{max}[v]) > v$, we would have liked to hook v on u. We are not allowed to do it here as u is not necessarily a neighbor of v. Instead, we hook v on $w = \texttt{ver}(\texttt{walk}'[v, \texttt{time}[v]])$. In other words, we look at the cell $[v,t]$ which is *closest*, among all the cells of v, to a cell $[u,t']$ belonging u. The cell $[w,t''] = \texttt{walk}'[v,t]$ is then one step closer to the cell $[u,t']$. We hook v on w, i.e., we let $\texttt{parent}[v] \leftarrow w$. It is easy to see that w is a neighbor of v, this follows from the fact that v chose w at time t or w chose v at time t''. To show that no cycles are formed during the the first hooking phase, we show that $\texttt{label}[v] < \texttt{label}[w]$.

Lemma 6.6 *If a vertex v is hooked on a vertex w in the first hooking phase, then* $\texttt{label}[v] < \texttt{label}[w]$.

Proof: Recall that $\texttt{label}[v] = \texttt{label}[v,t] = \langle \texttt{max}[v], \texttt{dist}[v,t] \rangle$, where $t = \texttt{time}[v]$. This means that the cell $[v,t]$ is at distance $d = \texttt{dist}[v,t]$ from a cell $[\texttt{max}[v], t']$ that belongs to $\texttt{max}[v]$. Consider the cell $[w,t''] = \texttt{walk}'[v, \texttt{time}[v]]$ (here $t'' = t+1$ or $t'' = t-1$). This cell is one step closer to the cell $[\texttt{max}[v], t']$ which is the root of the tree to which the cells $[v,t]$ and $[w,t'']$ belong. Therefore $\texttt{label}[w,t''] = \langle \texttt{max}[v], d-1 \rangle$ and therefore $\texttt{label}[w] \ge \langle \texttt{max}[v], d-1 \rangle > \texttt{label}[v]$. □

As each vertex is hooked on a vertex with a larger label, no cycles are formed by the first hooking phase. This completes the description of the first hooking phase.

Consider now the trees formed in G' by the first hooking stage. A tree in G' is considered to be big enough if it contains either a near-inactive vertex, or a whole connected component, or at least $s^{1/3}$ active vertices. (Note that a tree that contains a near-inactive vertex also contains an inactive vertex, as each near-inactive vertex is hooked on an inactive vertex). If a tree T_r, rooted at r, is big enough, no additional actions are necessary. If, however, T_r is not big enough, we have to hook T_r on some other tree so that the resulting tree would be big enough.

The only trees that might not be big enough are trees that only contain inner-active vertices. Consider a tree T_r that only contains inner-active vertices. If T_r contains all the vertices in the virtual neighborhood $C(r)$ of the root r, then T_r is big enough. This follows from the fact that each virtual neighborhood either contains a near-inactive vertex, or a whole connected component, or at least $s^{1/3}$ active vertices. Thus, if T_r is not big enough, then T_r does not contain all the vertices of $C(r)$.

We next establish the following lemma:

Lemma 6.7 *Let T_r be a tree in G' formed by the first hooking phase. If T_r does not contain all the vertices of $C(r)$, then there exist vertices $v \in T_r$ and $u \in C(r) \setminus T_r$ such that $(v,u) \in E'$ and $u \in T_{r'}$ where $r < r'$.*

Proof: Let $w \in C(r) \setminus T_r$ be a vertex contained in the virtual neighborhood $C(r)$ of r but not in the tree T_r rooted at r. As $w \in C(r)$, we get, by the definition of virtual neighborhoods, that there exist $1 \le t_1, t_2 \le s'$ such that the cells $[r, t_1]$ and $[w, t_2]$ are contained in the same tree $T_{[r,t_1]} = T_{[w,t_2]}$. Furthermore, this tree is small enough. As r is not hooked in the first hooking phase, it is not difficult to see that the largest cell in the tree $T_{[r,t_1]}$ must be a cell of the form $[r, t'_1]$, where $1 \le t'_1 \le s'$ (it may be the case, of course, that $t'_1 = t_1$). This cell $[r, t'_1]$ is then made to be the root of the tree $T_{[r,t_1]}$. We may assume, therefore, without loss of generality that $[r, t_1]$ is the root of the tree $T_{[r,t_1]}$.

Consider the path $[w, t_2] = [w_0, t'_0], [w_1, t'_1], \ldots, [w_d, t'_d] = [r, t_1]$ in the tree $T_{[r,t_1]}$ from the cell $[w, t_2]$ to the root $[r, t_1]$. Let $[w_i, t'_i]$ be the first cell on this pass such that $w_i \in T_r$. We know that $0 < i \le d$. Let $u = w_{i-1}$ and $v = w_i$. Clearly $(v,u) \in E'$. As $[u, t'_{i-1}]$ belongs to the tree $T_{[r,t_1]}$, we get that $\mathtt{max}[u] \ge [r, t_1]$. Let r' be the root of the tree to which u belongs after the first hooking stage. As we assume that $u \notin T_r$, we get that $r \ne r'$. By Lemma 6.6, we get that $\mathtt{label}[u] < \mathtt{label}[r']$ and therefore $\mathtt{max}[u] \le \mathtt{max}[r']$. As we already know that $[r, t_1] \le \mathtt{max}[u]$, we get that $[r, t_1] \le \mathtt{max}[r']$. As r' is a root of a tree, we get that $r \le r'$. As $r \ne r'$, we get that $r < r'$, as required. $\square$

If an inner-active vertex v is contained after the first hooking stage in a tree T_r that is not big enough, we let $\mathtt{root}[v] \leftarrow r$ be the root of the tree containing it. If v is contained in a big enough tree in G', we assign $\mathtt{root}[v]$ a number larger than the serial number of all the vertices in G'.

Lemma 6.7 implies that if T_r is not big enough, then there is a vertex $v \in T_r$ and a vertex $u \notin T_r$ such that $(v,u) \in E'$ and $\mathtt{root}[u] > \mathtt{root}[v] = r$.

In the second hooking phase each inner-active vertex v finds its neighbor u for which $\mathtt{root}[u]$ is maximal. For each tree T_r which is not large enough, we then find a vertex u which is adjacent to a vertex in T_r and for which $\mathtt{root}[u]$ is maximal (note that each such tree contains at most $s^{1/3}$ vertices). Let $v \in T_r$ be a neighbor of this vertex u. We make v the root of the tree T_r, by reversing some of the pointers in the tree, and hook v on u.

It is easy to see that the hookings in the second phase do not create any cycles. This follows from the fact that each tree is hooked on a tree whose root has a larger serial number. It is also easy to see that each tree formed in G' is now big enough. Let T_r be a tree in G' formed by the two hooking phases. Then, r must have been a root of a tree also after the first hooking phase. Denote this tree by T'_r. Clearly $T'_r \subseteq T_r$. As the tree T'_r was not hooked during the second hooking phase, we get that T'_r was already big enough (if T'_r was not big enough, then it did not contain all the vertices of the virtual neighborhood $C(r)$, and T'_r would have been hooked on another tree.)

We have thus established the following theorem:

Theorem 6.8 *Procedure WALK-HOOK creates a forest in G' in which each tree contains, w.h.p., either at least $s^{1/3}$ vertices, or an inactive vertex, or all the vertices of a connected component of G'.*

In the next subsection we show that *WALK-HOOK* can be implemented to run in $O(\log s + \log\log n)$ time when $\Theta(\log n)$ processors are allocated to each edge of G'.

6.4 The implementation of *WALK-HOOK*

The graph $G' = (V', E')$ is specified using its adjacency lists. Each edge (u, v) has a pointer to its opposite edge (v, u). Each edge of G' has $\Theta(\log n)$ processors allocated to it. The degree of each non-isolated vertex in G' is at least s. Vertices of degree at most $d = s^{4/3}$ are marked as active.

The active vertices of G' are easily identified by running the first $\log d$ iterations of the simple list ranking algorithm on each adjacency list. We would then know which vertices have adjacency lists of length at most d.

Before starting the random walks, each active vertex is allocated with at least $\Theta(s \log n)$ processors. The processors allocated to a vertex v are simply the processors allocated to the edges that touch v. (A processor may therefore be allocated to two vertices. This only slows down the execution of the algorithm by a constant factor.) The processors allocated to a vertex v are given consecutive serial numbers staring from 1. As the degree of each active vertex is at most d, these operations can be easily carried out in $O(\log d + \log\log n)$ time using simple list ranking algorithms.

We next remove, temporarily, self-loops and parallel edges that connect two active vertices. The removal of self-loops is immediate. To remove parallel edges we need to sort the adjacency lists of the active vertices. The sorting is done using the linked-list based parallel sorting algorithm of Goodrich and Kosaraju [GK96]. When the adjacency lists are sorted, it is easy to mark parallel edges for deletion and then to delete them using a simple list ranking algorithm.

We next identify all the active vertices that have an inactive neighbor and mark them as *near-inactive*. All other active vertices are marked as *inner-active*. This can again be done in $O(\log d)$ time as follows. We first copy the marking of the active vertices to all the edges that touch them. Thus, if u is active, then all the edges $(u, v) \in E'$ and $(v, u) \in E'$ will carry an indication that u is active. (Each edge would carry up to two such indications.) An active vertex u is then marked as *near-inactive* if its adjacency list contains an edge (u, v) in which it is *not* indicated that v is active. Such a near-inactive vertex is immediately hooked on one of its inactive neighbors. As this only requires the scanning of the adjacency lists of the active vertices, and the degree of each active vertex is at most d, this can easily be done in $O(\log d)$ time.

After these preliminary operations we move to the more challenging task of constructing the `walk` array. To appreciate the difficulty of this task, note that when *WALK-HOOK* is called, we only have a linked list with the vertices of G'. Assigning the vertices of G' consecutive serial numbers is, therefore, too time consuming. It is possible to implement the `walk` array using an actual array, i.e., a matrix whose entries occupy consecutive memory locations. This approach was used in [KNP99] and by us in [HZ96b]. The drawback of this approach is that it is a bit wasteful in terms of work space as, at least in the naive implementation, we have to allocate to each vertex v of the original graph G a row in the matrix, even though we would eventually use only a small fraction of these rows. As we are aiming here for an algorithm that uses linear space, we use a different approach. We implement each row `walk`$[v]$ of the `walk` array using a *linked list* of length $s' = \Theta(s \log n)$. The length of the linked list `walk`$[v]$ that corresponds to an active vertex v is at most the number of processors allocated to v. Each one of the $m \log n$ processors used by the algorithm is alloted a constant number of memory cells. The t-th cell $[v, t]$ of the linked list `walk`$[v]$ resides in a memory cell allocated to the t-th processor allocated to v.

We next have to let the pointer `walk`$[v, t]$ that corresponds to the cell $[v, t]$ of an inner-active vertex v point to a cell $[u, t + 1]$, where u is a random neighbor of v in G'. (We assume here for simplicity that t is not divisible by s.) We achieve this goal by performing the following steps.

At first, we distribute the degree $\deg(v)$ of each inner-active vertex v to the s' processors allocated to its row. This takes $O(\log s') = O(\log s + \log\log n)$ time. The t-th processor allocated to v then puts in the cell $[v, t]$ a triple $(i, t, 0)$, where i is selected, uniformly at random, from $\{1, 2, \ldots, \deg(v)\}$. In other words, we start by choosing the indices of the random neighbors of v.

Next, using the processors allocated to the adjacency list of v, we create a copy `adj`$[v]$ of the adjacency list of v. The i-th element in the list is then replaced by the triple $(i, u_i, 1)$, where u_i is the i-th neighbor of v. We now concatenate the linked lists `walk`$[v]$ and `adj`$[v]$, and sort the concatenated list according to the first component of the triples. The sorting is again performed using the linked-list based sorting algorithm of Goodrich and Kosaraju [GK96] in $O(\log d + \log s')$ time. It is then easy to replace the triple $(i, t, 0)$ that we have placed in the cell $[v, t]$ by a pair $(u_i, t+1)$, where u_i is the i-th neighbor of v. We will shortly replace this pair by a pointer to the cell $[u_i, t+1]$.

The above sorting operation also gives us, for every inner-active vertex v and each vertex u which is a neighbor of v, a linked list `times`$[v, u]$ that contains the times in which v chose u. Each one of the elements in these linked lists has a processor allocated to it (note that the total length of all these lists is equal to the number of cells in the walk array). We establish links between each edge (v, u) of G' and the first element of the linked list `times`$[v, u]$.

As we saw in the previous subsection, the cells in the `walk` array are organized in trees. To avoid concurrent read and write operations, we set up for each cell $[u, t]$ a linked list `prev`$[u, t]$ of the cells that point to $[u, t]$. In other words, a cell $[v, t-1]$ appears in the list `prev`$[u, t]$ if and only if `walk`$[v, t-1] = [u, t]$. To obtain the lists `prev`$[u, t]$, we concatenate, for each vertex u, the linked lists `times`$[v_i, u]$ of all the neighbors v_i of u. The length of this concatenated list is at most $s'd$. Each time t in a list `times`$[v_i, u]$ is replaced before this concatenation by the pair (v_i, t)). We then sort the elements of these concatenated liked lists, again using the algorithm of Goodrich and Kosaraju [GK96]. We obtain a concatenation of the lists `prev`$[u, t]$, for $1 \le t \le s'$. It is easy to set up links from each cell $[u, t]$ to the first element in the list `prev`$[u, t]$.

Finally, using the `prev`$[u, t]$ lists, we can distribute the address of the cell $[u, t]$ to all the cells contained in the list `prev`$[u, t]$, thereby finishing the construction of the `walk` array.

Using the pointers of the `walk` array and the `prev` lists, we can easily define an Euler tour in each one of the trees of the forest formed by `walk`. By running the first $O(\log s)$ iterations of the simple list ranking algorithm on these Euler tours, we can identify the small enough trees, i.e., the trees whose sizes are at most $4s^2d$, and then compute `max`$[v, t]$, `label`$[v, t]$ for all the cells contained in small enough trees in $O(\log s)$ time. The computation of `max`$[v]$ and `label`$[v]$ then takes $O(\log s') = O(\log s + \log\log n)$ time.

The total running time of *WALK-HOOK* is therefore $O(\log s + \log\log n)$, when $\Theta(\log n)$ processors are allocated to each vertex. The amount of space needed is linear in the number of processors, i.e., $O(m \log n)$, where m is the number of edges of G'.

7 An easy m processor algorithm

In this section we describe a simple way of reducing the number of processors required by the third stage of our algorithm from $O((m+n)\log n)$ to $O(m+n)$. Although this reduction is not required for the purpose of obtaining our optimal $O(\log n)$ time spanning forest algorithm, we think that it is of interest. In particular, it supplies a relatively simple algorithm for finding a spanning forest in $O(\log n)$ time using $O(m+n)$ processors without using the techniques developed in the first and the second stages of the complete algorithm. The technique used to reduce the number of processors can also be used to reduce the failure probability of the algorithm.

To reduce the number of processors required from $O((m+n)\log n)$ to $O(m+n)$, we start the algorithm with $k' = \Theta(\log\log n)$ iterations in which we use the actual neighborhoods of the vertices of the graph and do not perform any random walks. In each iteration we use the standard hooking method of Shiloach and Vishkin

[SV82] and Awerbuch and Shiloach [AS87]. In these hooking methods each non-isolated vertex is either hooked on one of its neighbors, or one of its neighbors is hooked on it. Thus, each tree formed is of size at least 2. During these iterations we use the exponential sequence $d_0 = 1$ and $d_{i+1} = 2d_i$ to define the active and the inactive vertices, and not the doubly exponential sequence described earlier. It is easy to show that the degree of each non-isolated vertex before the i-th iteration is at least d_i. (Again, we are not removing self-loops and parallel edges.) The time spent on these k' iterations is $O(\sum_{i=0}^{k'} \log d_{i+1}) = O((\log\log n)^2)$ time. After these k' iterations $d_{k'} = \Omega((\log n)^2)$. In the i-th iteration, for $i > k'$, we then start $\Theta(\log n)$ random walks of length $\Theta(d_i/\log n) \geq \Theta(d_i^{1/2})$ from each inner-active vertex. We have enough processors for this as the adjacency list of each active vertex is of length at least d_i and has at least d_i processors allocated to it. The virtual neighborhood of each non-isolated vertex will be, with very high probability, of size at least $(d_i^{1/2})^{1/3} = d_i^{1/6}$. We will henceforth continue with the sequence $d_{i+1} = d_i^{7/6}$.

To reduce the failure probability we start with $k' = 2\sqrt{\log n}$ iterations in which we use the actual neighborhoods of the vertices. These iterations take $O(\log n)$ time. After these k' iterations $d_{k'} = 2^{2\sqrt{\log n}}$. In the i-th iteration, for $i > k'$, we start $\Theta(2^{\sqrt{\log n}})$ random walks of length $\Theta(d_i/2^{\sqrt{\log n}}) \geq \Theta(d_i^{1/2})$ from each active vertex. Again, we have enough processors as each active vertex has an adjacency list length of at least d_i and has at least d_i processors allocated to it. The probability that the virtual neighborhood of an active vertex will be of size less than $d_i^{1/6}$ is now at most $2^{-\Omega(2^{\sqrt{\log n}})}$. The probability that the third stage fails to complete in $O(\log n)$ time is now at most $2^{-\Omega(2^{\sqrt{\log n}})}$. We can adjust the parameters of the load balancing scheme used in the first stage so that the probability that the first stage would fail to complete in $O(\log n)$ time would be at most $2^{-\Omega(2^{\log n/(\log\log n)^2})}$ which is at most $2^{-\Omega(2^{\sqrt{\log n}})}$. The second stage uses the third stage as a subroutine. The probability that the whole algorithm would fail to complete in $O(\log n)$ time is then at most $2^{-\Omega(2^{\sqrt{\log n}})}$.

8 Applications

Many algorithms for solving basic graph connectivity problems use an algorithm for finding a spanning forest as a subroutine. In many cases, the running time of the whole algorithm is dominated by the spent in finding a spanning forest. By plugging our connectivity algorithm into these algorithms we can obtain optimal randomized EREW PRAM algorithm for many other basic connectivity problems.

By plugging our algorithm into an algorithm of Tarjan and Vishkin [TV85] we obtain an optimal randomized EREW PRAM algorithm for finding the *bridges* and *biconnected components* of an undirected graph. By using our algorithm in conjunction with the algorithms of Awerbuch, Israeli and Shiloach [AIS87] and Atallah and Vishkin [AV84] we obtain optimal randomized EREW PRAM algorithm for finding *Euler tours* in both directed and undirected Eulerian graphs.

Our algorithm can be plugged into the algorithm of Maon, Schieber and Vishkin [MSV86] and of Miller and Ramachandran [MR86] (see also Ramachandran [Ram92],[Ram93]) for finding *ear decompositions* and *open ear decompositions* yielding optimal randomized EREW PRAM algorithms for these problems and, as a consequence, for the *st-numbering* problem. To obtain these algorithms we have to be able to find the LCA's of all the non-forest edges in the graph in $O(\log n)$ time using $O((m + n)/\log n)$ processors. The way this is done is described in the next section.

Further applications of our algorithm may, in the future, include triconnectivity [FRT93],[Ram93] and planarity testing [RR94].

9 Finding LCA's

We are given a connected undirected graph $G = (V, E)$ and a rooted spanning tree T of it. The graph is represented using adjacency lists and the tree is represented using an array of parent pointers. For each non-tree edge $e = (u, v)$ we have to find the *least common ancestor* (LCA) of e, denoted by $\texttt{lca}(e)$, which is the the common ancestor of u and v furthest from the root.

We describe here a method of accomplishing that in $O(\log n)$ time using $O((m+n)/\log n)$ processors in the EREW PRAM model. This method was communicated to us by Vijaya Ramachandran [Ram96].

The problem of finding LCA's in a tree was considered by Schieber and Vishkin [SV88]. They describe a preprocessing algorithm that runs in $O(\log n)$ time using $O((m+n)/\log n)$ processors in the EREW PRAM model. After this preprocessing, each LCA query can be answered in $O(1)$ *sequential* time. In the CREW PRAM model we can therefore perform this preprocessing and then find the LCA's of all the non-tree edges in $O(1)$ time using $O(m)$ processors, or in $O(\log n)$ time using $O((m+n)/\log n)$ processors. We need concurrent-read capability in this approach as while finding the LCA of two non-tree edges we may try to access the same location in the data structure prepared by the preprocessing stage.

There is, however, an easy way around this problem. Note that all we need is a way of finding the LCA's of all the non-tree edges in $O(\log n)$ time using $O((m+n)/\log n)$ processors. We are *not* required to find all LCA's in $O(1)$ using $O(m)$ processors, after a preprocessing stage. Furthermore, in our case, all the required LCA's queries are known in advance, before the preprocessing stage.

There is a well known reduction from the LCA problem to the range minima problem (see, e.g., [JáJ92]). Construct an Euler tour of the tree and number the vertices in preorder. Let $e = (u, v)$ be a non-tree edge. Assume, without loss of generality, that $u < v$ (according the the preorder numbering of the vertices). It is easy to see that the LCA of the edge $e = (u, v)$ is the vertex w with minimal preorder number that appears on the portion of the tour that begins with the edge $(u, \texttt{parent}[u])$ and ends with the edge $(\texttt{parent}[v], v)$. The LCA problem is thus reduced to the range minima problem.

Suppose, therefore, that we are given an array X of length n and m pairs (a_i, b_i), for $1 \le i \le m$. For every $1 \le i \le m$ we have to find the minimal element in the segment $X[a_i..b_i]$. We show how to solve this problem in $O(\log n)$ time using $(m+n)/\log n)$ processors in the EREW PRAM model. (We assume that along with the queries (a_i, b_i), we are also given a sorted list of the a_i's and a sorted list of the b_i's.)

Assume, at first, that we have $m + n$ processors. For every $1 \le i \le n$ and every $1 \le j \le \lceil \log n \rceil$, we compute the minimum in the segment $X[i..i+2^j-1]$ and in the segment $X[i-2^j+1..i]$. This can be easily done in $O(\log n)$ time using n processors. Each query can now be answered by computing by computing the minimum of two precomputed values. This follows from the fact that the segment $X[a_i..b_i]$ is covered by the segments $X[a_i..a_i + 2^j - 1]$ and $X[b_i - 2^j + 1..b_i]$, where $j = \lfloor \log(b_i - a_i) \rfloor$. Thus, all the queries can now be answered in $O(1)$ time using m processors, if concurrent read operations are allowed.

We have two obstacles to overcome. The first is that above solution requires $O(m + n \log n)$ work, and not $O(m + n)$ work as desired. The second is that it requires concurrent read operations. To reduce the work to $O(m + n)$ we use the following approach. Partition the array into $n/\log n$ segments of length $\log n$. Let $X'[i]$ be the minimal element in the i-th segment. Process each segment of length $\log n$ using an optimal sequential preprocessing algorithm so that each query in the interval could be answered in $O(1)$ sequential time. This can be done in $O(\log n)$ time with a single processor allocated to each segment using an algorithm of Gabow, Bentley and Tarjan [GBT84]. Now apply the preprocessing described above to the array X'. As X' is of length $n/\log n$, the total work needed now is $O(n + m)$. For each $1 \le i \le m$, we can now find the minimal element in the segment $X[a_i..b_i]$ as follows. Let a'_i be the smallest index larger than a_i in which a block of length $\log n$ ends. Let b'_i be the largest index smaller than b_i in which such a segment begins. The minimal elements in the segments $X[a_i..a'_i]$ and $X[b'_i..b_i]$ can be found in $O(1)$ time using the data structure set up by the preprocessing done on the segments of length $\log n$. The minimal element in $X[a'_i+1..b'_i]$ can be found using the values computed during the preprocessing of X'. This gives us a solution that runs in $O(\log n)$ time using $(m+n)/\log n$ processors in the CREW PRAM model.

It is not difficult to see that the concurrent reads in the above solution can be easily avoided if all the endpoints of the intervals are distinct. As we assume that the a_i's and the b_i's are also given to us in sorted order, we can easily count the number of queries in which each element in the array X participates. We then *duplicate* each element the required number of times. The total length of the array is now at most $2m+n$. We can now run the algorithm described above on the EREW PRAM model.

10 Concluding remarks and open problems

The main remaining open problem is whether there exists a *deterministic* optimal speed-up $O(\log n)$ time EREW PRAM algorithm for finding spanning forests. Such a deterministic algorithm was not obtained yet even in the much stronger CRCW PRAM model.

Another interesting open problem is whether there exists a randomized linear work $O(\log n)$ time EREW PRAM algorithm for finding a *minimum* spanning forest in a *weighted* graph. Cole, Klein and Tarjan [CKT96] recently obtained such an algorithm in the CRCW PRAM model. Karger [Kar94] obtained, for any $\epsilon > 0$, a randomized $O(n^{1+\epsilon}+m)$ work $O(\log n)$ time EREW PRAM algorithm for finding minimum spanning forests. Recently, King *et al.* [KPRS97] obtained an optimal deterministic EREW PRAM algorithm for minimum spanning tree *verification*. Poon and Ramachandran [PR97] obtained a randomized linear work EREW PRAM algorithm for finding a minimum spanning tree whose running time is $o(\log^{1+\epsilon} n)$ for any $\epsilon > 0$.

Acknowledgment

We would like to thank Vijaya Ramachandran for discussions that led to the simplification of our algorithm. We would also like to thank her for allowing us to include the material of Section 9 in this paper. We would also like to thank an anonymous referee for numerous comments on two previous drafts of this paper that helped improve its presentation.

References

[AIS87] B. Awerbuch, A. Israeli, and Y. Shiloach. Finding Euler circuits in logarithmic parallel time. In F. Preparata, editor, *Advances in Computing Research, Volume 4*, pages 79–86. JAI Press, 1987.

[AKL+79] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal sequences and the complexity of maze problems. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico*, pages 218–223, 1979.

[AS87] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36:1258–1263, 1987.

[AS92] N. Alon and J.H. Spencer. *The probabilistic method.* Wiley, 1992.

[AV84] M.J. Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29:330–337, 1984.

[BF96] G. Barnes and U. Feige. Short random walks on graphs. *SIAM Journal on Discrete Mathematics*, 9:19–28, 1996.

[CKT96] R. Cole, P.N. Klein, and R.E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proceedings of the 8th Annual ACM Symposium on Parallel algorithms and architectures, Padua, Italy*, 1996.

[CL95] K.W. Chong and T.W. Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. *Journal of Algorithms*, 18:378–402, 1995.

[CLC82] F.Y. Chin, J. Lam, and I.N. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, 1982.

[Col88] R. Cole. Parallel merge-sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[CV91] R. Cole and U. Vishkin. Approximate parallel scheduling. Part II: Applications to logarithmic-time optimal graph algorithms. *Information and Computation*, 92:1–47, 1991.

[FRT93] D. Fussell, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacement. *SIAM Journal on Computing*, 22:587–616, 1993.

[Gaz91] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM Journal on Computing*, 20(6):1046–1067, 1991.

[GBT84] H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, Washington, D.C.*, pages 135–143, 1984.

[GK96] M.T. Goodrich and S.R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *Journal of the ACM*, 43:331–361, 1996.

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

[HW90] Y. Han and R.A. Wagner. An efficient and fast parallel connected component algorithm. *Journal of the ACM*, 37(3):626–642, 1990.

[HZ96a] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, Atlanta, Georgia*, pages 438–447, 1996.

[HZ96b] S. Halperin and U. Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. *Journal of Computer and System Sciences*, 53:395–416, 1996.

[IK94] K. Iwama and Y. Kambayashi. A simpler parallel algorithm for graph connectivity. *Journal of Algorithms*, 16:190–217, 1994.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[JM95] D.B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. *Journal of Algorithms*, 19:383–401, 1995.

[JM97] D.B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for CREW PRAM. *Journal of Computer and System Sciences*, 54:227–242, 1997.

[Kar94] D.R. Karger. *Random Sampling in Graph Optimization Problems.* PhD thesis, Department of Computer Science, Stanford University, 1994.

[KKT95] D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328, 1995.

[KNP99] D.R. Karger, N. Nisan, and M. Parnas. Fast connected components algorithms for the EREW PRAM. *SIAM Journal on Computing*, 28:1021–1034, 1999.

[KPRS97] V. King, C.K. Poon, V. Ramachandran, and S. Sinha. An optimal EREW PRAM algorithm for minimum spanning tree verification. *Information Processing Letters*, 62:153–159, 1997.

[KRS90] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5:43–64, 1990.

[KT94] P.N. Klein and R.E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Montréal, Canada*, pages 9–15, 1994.

[MR86] G. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Unpublished manuscript, MSRI, Berkeley, CA, 1986.

[MR89] G. L. Miller and J.H. Reif. Parallel tree contraction, Part 1: Fundamentals. In S. Micali, editor, *Advances in Computing Research, Volume 5, Randomness and Computation*, pages 47–72. JAI Press, 1989.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[MSV86] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and *st*-numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.

[NM82] D. Nash and S.N. Maheshwari. Parallel algorithms for the connected components and minimal spanning trees. *Information Processing Letters*, 14:7–11, 1982.

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

[Phi89] C.A. Phillips. Parallel graph contraction. In *Proceedings of the 1st Annual ACM Symposium on Parallel algorithms and architectures*, pages 148–157, 1989.

[PR97] C.k. Poon and V. Ramachandran. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *Proceedings of the 8th Annual International Symposium on Algorithms and Computation*, 1997.

[Rad94] T. Radzik. Computing connected components on EREW PRAM. Technical Report 94/02, King's College London, 1994.

[Ram92] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. Technical Report UTEXAS.CS//CS-TR-92-02, University of Texas at Austin, Department of Computer Sciences, 1992.

[Ram93] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In J.H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 1, pages 275–340. Morgan Kaufmann, 1993.

[Ram96] V. Ramachandran. Personal communication. 1996.

[RR94] V. Ramachandran and J. Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49:517–561, 1994.

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

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17:1253–1262, 1988.

[TV85] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14:862–874, 1985.