Sub-Linear Distributed Algorithms for Sparse Certificates and Biconnected Components.*

Ramakrishna Thurimella
Department of Mathematics and Computer Science
University of Denver, Denver, CO 80208. U.S.A.

Abstract A *certificate* for the k connectivity† of a graph $G = (V, E)$ is a subset E' of E such that (V, E') is k connected iff G is k connected. Let $n = |V|$ and $m = |E|$. A certificate is called *sparse* if it has size $O(kn)$. We present a distributed algorithm for computing sparse certificate for k connectivity whose time complexity is $O(k(D+n^{0.614}))$ where D is the diameter of the network. A new algorithm for identifying biconnected components is also presented. This algorithm is significantly simpler than many existing algorithms and can be implemented in a distributed environment to run in $O(D+n^{0.614})$ time. Both algorithms improve on the previous best known time bounds. Our main focus in this paper is the time complexity. However, no more than a polynomial number of messages, each of size $O(\log n)$, are generated by the algorithm.

1 Introduction

Connectivity is an important property of graphs with many applications in computer science. We study the distributed time complexity of questions pertaining to vertex and edge connectivity of graphs. A connected graph is said to be *k-vertex* (resp. *k-edge*) connected if it has at least $(k+1)$ vertices (resp. edges), and the deletion of any $(k-1)$ vertices (resp. edges) leaves the graph connected. The vertex connectivity when $k = 2$ is also known as *biconnectivity. Biconnected components* or *blocks* are equivalence classes induced on the edge set by relating two edges e_1, e_2 if and only if there exists a simple cycle containing e_1 and e_2. The edge and vertex connectivity determine respectively the number of link and node failures that can be tolerated by a distributed network.

A *certificate* for the k connectivity of a graph $G = (V, E)$ is a subset E' of E such that the subgraph (V, E') is k connected if (and only if) G is k connected. The *size* of a certificate is $|E'|$. Let n and m denote $|V|$ and $|E|$. There is a trivial lower bound of $kn/2$ on the size of a certificate for k connectivity because the degree of every vertex in a k connected graph is at least k. An example of a certificate for the 1-connectivity of a connected graph G is a spanning tree of G. Moreover, this is a minimum-size certificate. However, for $k > 1$, the problem of finding minimum-size certificates for k connectivity is NP-complete[GJ 79]. Therefore, certificates whose size is within a constant factor of minimum are of interest. Call a certificate for k connectivity *sparse* if it has size $O(kn)$.

*Research supported in part by National Science Foundation grant CCR-9210604.

†k connectivity refers to both k-edge connectivity and k-vertex connectivity unless explicitly stated otherwise.

An area of research interest in distributed computing is to design fault tolerant protocols [Ha 87], [IR 88]. To this end, sparse certificates are useful in serving as a reliable means of doing a message-efficient broadcast on a distributed network. They are preferable to a spanning tree because the latter can withstand no failures. Efficient sequential, parallel and distributed algorithms for computing sparse certificates are given by [CKT 93] (see also [NI 90] for a sequential algorithm). Their distributed algorithm is rather simple: Find a breadth-first search (BFS) spanning tree F_1 in G with an arbitrary vertex as the root. Delete the edges of F_1 from G and denote the resulting graph by $G - F_1$. Find another BFS spanning forest F_2 in $G - F_1$, again rooting each tree at an arbitrary vertex in each component. Repeat this process k times. It is shown in [CKT 93] that $F_1 + F_2 + \cdots + F_k$ is a sparse certificate for k connectivity of G. The best known distributed algorithm for BFS takes time $O(D \log^3 n)$ with $O(m + n \log^3 n)$ messages[AP 90] where D is the diameter of the network. Nonetheless, the time to compute a sparse certificate in the worstcase is $O(kn \log^3 n)$ (and not $O(kD \log^3 n)$) as the diameter of $G - \Sigma F_i$ for some i, $1 \leq i \leq (k-1)$, could be as high as n.

The question of identifying inherent graph parameters that govern the distributed complexity of various fundamental network problems has been raised by [GKP 93]. This issue is important because many previous algorithms that achieved a time bound of $O(n)$ claimed optimality under the assumption that the solution is optimal for *some* networks of n vertices. A more desirable optimality, as argued in [GKP 93], is achieved when an algorithm solves a problem optimally on *every* instance. They further make a case for the diameter of a network D as being one of the important parameters that is inherent in the distributed complexity of algorithms. Under this paradigm, they present a novel distributed algorithm for computing a minimum spanning tree (MST) that runs in time $O(D + n^{0.614})$. The chief tools employed in achieving this bound are *fragments* – connected subgraphs of limited diameter and a BFS tree that spans the entire network. The computation is part distributed and part central. Each fragment achieves a subgoal distributedly and independently in parallel. The results of the subgoals are "put together" centrally at the root of a BFS tree. The final results are broadcast over the entire network using pipelining. To get good performance, the number as well as the diameter of the fragments is controlled and the volume of the final broadcast is kept small.

In this paper, the technique of dividing the computation into part distributed and part central (using fragments and BFS) is shown to be useful in other contexts. In particular, we show that a sparse certificate for k connectivity can be found by similar methods in time $O(k(D + n^{0.614}))$. This improves the previous best known bound of $O(kn \log^3 n)$. We also present a new algorithm for computing biconnected components. This algorithm is extremely simple and is suitable for implementation on various models of computation in addition to the distributed model. On a distributed network, it runs in $O(D + n^{0.614})$ time. This is the first sublinear-time distributed algorithm for biconnected components. There are two known distributed algorithms for biconnected components both of which take at least linear time [Hu 89], [Ho 90]. As in [GKP 93], the main focus here is the time complexity, and we ignore the communication costs.

The *model* of computation is a point-to-point communication network represented by an undirected graph $G(V, E)$ where the set of vertices V stands for processors and the set of edges E stands for bidirectional communication channels. There is no shared memory and processors may communicate only by transmitting messages. Typically, there are two complexity measures that are used to analyze distributed algorithms: the *communication* complexity and the *time* complexity. The communication complexity is the total number of *elementary messages* generated in the worstcase during the execution of the algorithm where each elementary message consists of length $O(\log n)$ bits. The time complexity is the number of *rounds* in the worstcase where in each round, each processor may send out at most one elementary message per incident edge, receive all messages sent to it during that round from its neighbors, and carry out some local computation. As mentioned before, we will not try to minimize the number of messages generated in this paper. Therefore, for a slight overhead in communication complexity, we can employ a synchronizer and assume the computation to be synchronous.

The rest of the paper is organized as follows. The next section describes scan-first search and its relation to finding sparse certificates. Section 3 presents a distributed algorithm for edge and vertex certificates. An algorithm for finding biconnected components is presented in Section 4. Concluding remarks are given in Section 5.

2 Scan-First Search

Scan-first search (SFS) was first introduced in [CKT 93]. It proved to be a useful tool in obtaining sparse certificates on the parallel model of computation. In this paper, we show that it is valuable on the distributed model as well.

A *scan-first search* in a connected undirected graph G starting from a specified vertex r is a systematic way of visiting the vertices of G. To *scan* a vertex is to visit all previously unvisited neighbors of that vertex. At the beginning of the search, only r is visited. Then, the search iteratively scans an already visited but unscanned vertex until all vertices are scanned.

For an undirected graph that is not connected, a scan-first search can be performed on each connected component by starting from an arbitrary vertex and applying the above procedure. The search produces a spanning forest with a spanning tree in each connected component.

Notice that SFS is less restrictive than sequential BFS. In other words, all sequential BFS trees are SFS trees but some SFS trees are not BFS trees. For example, in any odd cycle of length n with a specified root r, there is only one BFS tree: the tree obtained by removing the edge at distance $(n-1)/2$ from r. However, there are $n-2$ SFS trees: the trees obtained by removing any edge except the two edges incident on r.

The following result was proved in [CKT 93].

Theorem 1 (Cheriyan, Kao, Thurimella) *Assume that the vertices of a connected graph G are labeled with preorder labels using an arbitrary tree rooted at r. For each v, $v \neq r$, let $n(v)$ denote the neighbor of v with the smallest preorder number. Then, the subgraph formed by the edges $(v, n(v))$, for all $v \neq r$, is an SFS spanning tree of G.*

The above theorem can be stated in a slightly more general form:

Theorem 2 *Let G be a graph with c components. Add any $c-1$ edges that make G connected and denote the resulting graph by G'. Assume that the vertices are labeled using a preorder traversal of a spanning tree T of G'. Consider a component C of G. Let r_c be the vertex with smallest preorder label in C. For each $v \in V(C)$, $v \neq r_c$, let $n(v)$ be the neighbor of v in C with the smallest preorder label. Then, the subgraph formed by the edges $(v, n(v))$ is an SFS spanning tree of C.*

Proof: Denote the preorder that is used in the theorem by Z. Consider another preorder Q on the vertices of C using the subtree of T restricted to C with r_c as the root. Now, apply the method of Theorem 1 on C using Q. For any two vertices, w and x of C, if x is visited after w in Q, then x will be visited after w in Z, possibly after visiting vertices of other components. In fact, $pre(w) < pre(x)$ in Z if and only if $pre(w) < pre(x)$ in Q. That is, for each v, $v \neq r_c$, the relative ordering of the neighbors of v in C is the same in Z and Q. Therefore, for every vertex v, $n(v)$ is the same regardless of whether Z or Q is used. ∎

3 Sparse Certificates

A sparse certificate for k-edge connectivity can be computed as shown below.

Algorithm 1 *Edge Certificate*

Input: k-edge connected $G = (V, E)$.
Output: A sparse certificate C.

1. Let G_0 be G with edge weights equal to 0.
2. **For** $j \leftarrow 1, \ldots, k$ **do**
   (a) Find an MST T in G_{j-1}.
   (b) If the weight of an edge is 0 in G_{j-1} and if it belongs to T, then change it to 1 and call the new graph G_j.
3. C_k, the sparse certificate for k-edge connectivity, is the subgraph of G consisting of edges whose weight is 1.

End

Lemma 1 *Let H be a subgraph of G. Let G_w be a weighted version obtained from G by assigning weight one (resp. zero) to the edges of H (resp. $G - H$). Then, the zero weight edges of an MST of G_w constitute a maximal spanning forest of $G - H$.*

Proof: Denote the MST by T and its subgraph of zero weight edges by F. Clearly, F has no cycles and $E(F) \subseteq E(G - H)$. If F is not maximal, then there exists an edge e of weight 0 in $G - H$ whose addition to F will not create a cycle. But adding e to T will create a cycle. Therefore, this cycle must have an edge f whose weight is 1. That is, we can trade e for f and keep T connected. The total weight of this new spanning tree is one less than that of T, contradicting that T is an MST. ∎

Theorem 3 *The distributed time complexity of the above algorithm is $O(k(D + n^{0.614}))$. The above algorithm is correct.*

Proof: Since the above algorithm can be implemented by invoking the distributed algorithm of [GKP 93] k times, the time complexity follows.

The sequential algorithm proposed by [T 89] for edge certificates is similar to the above algorithm with the following difference. The definition of G_j is $(G_{j-1} - T)$, where T is a maximal spanning forest in G_{j-1}. By Lemma 1, finding a maximal spanning forest in ($G - \{$the edges of current certificate$\}$) is equivalent to finding the zero-weighted edges of an MST in an appropriately weighted version of the graph. This proves the correctness. ∎

Now consider finding a certificate for k-vertex connectivity. To preserve vertex connectivity, it is not enough to take any spanning forest in G_{j-1}. However, taking an SFS spanning forest will suffice, as shown in [CKT 93]. Actually, any BFS tree is an SFS tree. However, finding BFS trees in subgraphs in sublinear time seems to be difficult. Instead, we show how to compute preorder numbers of a tree, thus enabling us to convert an arbitrary tree into an SFS one as pointed out in Theorem 2.

Algorithm 2 *Vertex Certificate*

Input: A k-vertex connected $G = (V, E)$.
Output: A sparse certificate C.

1. Let G_0 be G with edge weights equal to 0.
2. **For** $j \leftarrow 1, \ldots, k$ **do**
   (a) Find an MST T in G_{j-1}. Denote the zero-weighted subgraph of T by F. (F is a maximal spanning forest in the zero-weighted subgraph of G by Lemma 1.)
   (b) Preorder label the vertices of G using T. Convert F into an SFS spanning forest F' by the method described in Theorem 2. If the weight of an edge is 0 in G_{j-1} and if it belongs to F', then change it to 1 and call the new graph G_j.
3. C_k, the sparse certificate for k-vertex connectivity, is the subgraph of G consisting of edges whose weight is 1.

End

In the rest of this section, we show how to fold the preorder computation into the algorithm for MST from [GKP 93].

Let us recapitulate some of their terminology and results. The result of Lemma 2.6 of their paper states that the input graph can be partitioned into at most $N = n/2^I$ *fragments* (each fragment $\mathcal{F}$ is a connected subgraph that is induced by a subset of vertices) where each fragment has a diameter, denoted by d, at most 3^I. Here I is a parameter that can be chosen so as to minimize the total running time. It is pointed out in Section 5 of [GKP 93] that the best choice of I is the one for which $N = d$, i.e. $n/2^I = 3^I$.

Their final part of the algorithm can be summarized as follows. Each fragment $\mathcal{F}$ contains a rooted spanning tree, the root r being referred to as the *center* of that fragment. In order to find an MST, they first build a BFS spanning tree T_b. Then, using the center of each fragment r, the list of all edges incident on $\mathcal{F}$ that are remaining at the end of Phase III is sent up (at most $O(N)$, by Corollary 3.11) to T_b's root in time $O(N + D)$. The root of T_b computes centrally the list of edges of an MST that connect different fragments and broadcasts this list over the network.

Our strategy is to use their framework of fragments and centers. In other words, we will make the center of each fragment r "responsible" for preorder labeling the vertices of the subtree that belongs to $\mathcal{F}$. We will also assume that there is a rooted BFS tree T_b superimposed over the fragments to facilitate the computations that need to be performed centrally. Refer to Figure 1(a).

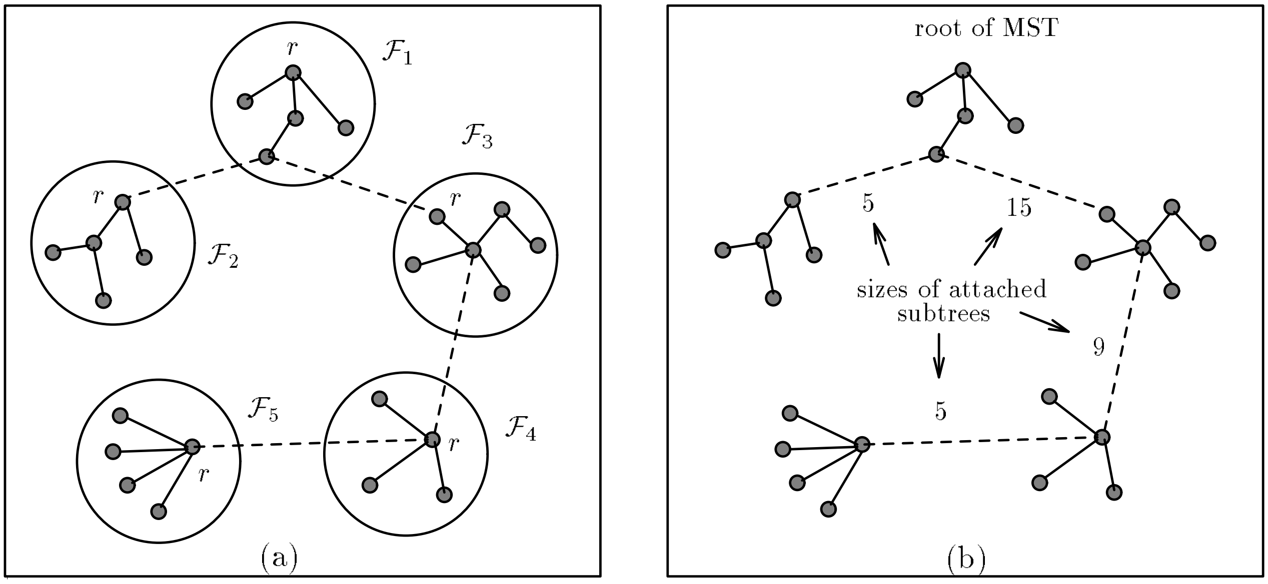

Figure 1: Centers and fragments of a minimum spanning tree (MST).

Each fragment is shown in a circle. To keep the figures simple, BFS tree T_b is not shown. The dashed edges, the edges connecting different fragments, will be referred to as *inter-fragment edges*, following the terminology of [GKP 93].

Algorithm 3 *Distributed Preorder*

Input: A $G = (V, E)$, an MST T with edges directed away from the root $R(T)$, a BFS tree T_b with root $R(T_b)$, fragments $\mathcal{F}$ and their roots r.
Output: A preorder number $pre(v)$, for every v of T.

1. Compute *size*, the number of vertices, of each fragment $\mathcal{F}$ at r.
2. Collect size information at $R(T_b)$. For each inter-fragment edge $u \to v$ of T, compute centrally at $R(T_b)$, the size of the subtree of T rooted at v. Associate this size with $u \to v$. See Figure 1(b).
3. Broadcast these sizes over the entire network.
4. At each r, using the sizes associated with inter-fragment edges and assuming the $pre(r)$ to be 0, compute the preorder number P of all vertices of $\mathcal{F}$. Also, compute the preorder number P of the roots of fragments adjacent to $\mathcal{F}$. Associate the P values of roots with the inter-fragment edges going out of $\mathcal{F}$. (In Figure 2(a), the P values of roots are the first components of the closed intervals associated with the dashed edges.)
5. Send up the P labels of inter-fragment edges to $R(T_b)$. Compute centrally, the final preorder number for each r by summing the P labels of the inter-fragment edges from r to the root of the MST $R(T)$.

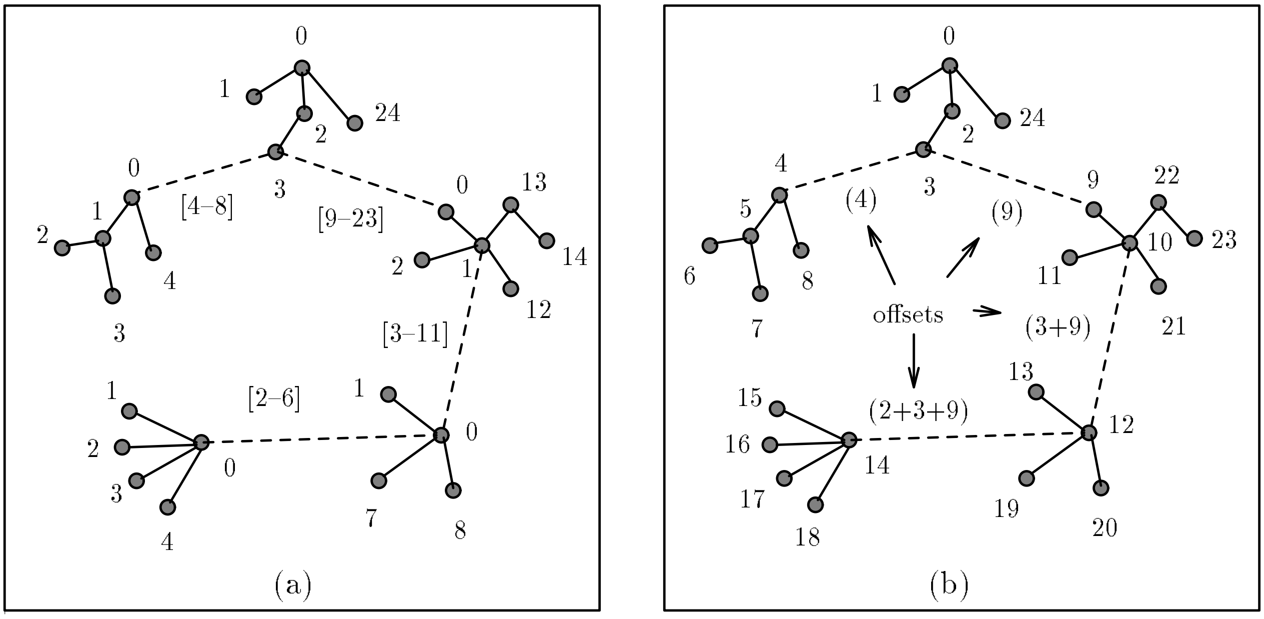

Figure 2: Distributed preorder computation.

6. Broadcast the final preorder numbers of each r over the network.
7. Each r computes the preorder number for the vertices of $\mathcal{F}$ by adding its own preorder number to P – the preorder number computed assuming $pre(r)$ to be 0 in Step 4.

End

Theorem 4 *The distributed time complexity of the above algorithm is* $O(D + n^{0.614})$. *The above algorithm is correct.*

Proof: First consider preparing the input for the algorithm, i.e. directing the tree edges away from the root $R(T)$. Start by directing the inter-fragment edges of the tree centrally at $R(T_b)$ and broadcast these directions over T_b. The computation is central and takes constant time. Broadcasting the direction of $N-1$ edges can be done in $O(N+D)$ by pipelining. Each fragment now has exactly one incoming edge $u \rightarrow r$ except for one – the fragment containing $R(T)$ which has zero incoming edges. We will refer to the head of this incoming edge as the *root* r of the fragment. (Assume the *root* r of the fragment containing $R(T)$ to be the root of MST $R(T)$.) The edges in different fragments are directed in simultaneously in parallel. This can be done by a broadcast from r in $O(d)$ time.

Computing the size of a fragment $\mathcal{F}$ at its root r can be done in time proportional to the diameter of $\mathcal{F}$. In fact, we can compute for each node, the size of the subtree attached to it within $\mathcal{F}$ by starting with leaves and sending up the sizes to their ancestors. Each node, after receiving the sizes from all its children, adds them up and sends the total to its parent. After d rounds, the root r will have the size of its fragment.

Steps 3 and 6 are similar and can be done by pipelining in time $O(N + D)$.

Step 4 can be implemented to run in time $O(d)$ as follows. Assume at each node v, we have the size of the subtree rooted at v and the sizes of the subtrees rooted at the children of v. Compute centrally at v, the prefix sums of the sizes of the subtrees rooted at the children of v. Once $pre(v)$ is known, add the $(i-1)$th prefix sum to $pre(v)+1$ and send it to the ith child. The value received from the parent at a node is the preorder number of that node. The whole computation is initiated at r by setting $pre(r) = 0$. Similarly, Step 7 can be implemented by doing a broadcast of $pre(r)$ from r which takes time $O(d)$.

Finally, consider Step 5. Sending up the P labels takes time $O(N + D)$. Once these values are available, the rest of the computation is local and takes constant time.

The distributed time complexity of the above steps, and that of the entire algorithm, is $O(N + D)$. According to [GKP 93], the total time is minimized when the value of N is $n^{0.614}$. That establishes the time bound.

The correctness can be seen from the following observations. Notice that if y is descendant of x, then $pre(y) \in [pre(x), pre(x) + size(x) - 1]$. If x is a tree vertex with $x_1, x_2, \ldots$ as its children in some order, then one possible preordering of the children of x after labeling x is $pre(x_i) = pre(x) + 1 + \Sigma_{j=1}^{i-1} size(x_j)$. Consider an inter-fragment edge $u \to v$. Let u and v belong to $\mathcal{F}_i$ and $\mathcal{F}_j$ respectively. Let $[x, y]$ be the interval associated with $u \to v$. Then, there are x vertices of $\mathcal{F}_i$ that occur before v in the preorder of T. Now, consider an r, the center of a fragment, and the tree path from r to $R(T)$ – the root of the MST T. Let $[x_1, y_1], [x_2, y_2], \ldots, [x_k, y_k]$ be the intervals associated with the inter-fragment edges encountered on this path. Clearly, the number of vertices of T that occur before r in the preorder of T, and hence $pre(v)$, is equal to $x_1 + x_2 + \cdots + x_k$. ∎

4 Biconnectivity

We first present an algorithm to identify biconnected components that is significantly simpler than many existing algorithms. We then describe its implementation on a distributed network.

The distributed algorithm of Huang [Hu 89] is based on the parallel algorithm proposed by Tarjan and Vishkin [TV 84]. The main idea behind both these algorithms is to reduce the problem of computing blocks to that of computing connected components. The algorithm of Tarjan and Vishkin constructs an auxiliary graph whose connected components correspond to the blocks of the original graph. In contrast, Huang's algorithm avoids building this auxiliary graph but constructs a graph that is a locally modified subgraph of the original graph. In this paper, we show that blocks can be retrieved from the connected components of a subgraph. Thus our algorithm builds neither an auxiliary graph nor a locally modified subgraph.

To identify the subgraph from which blocks can be retrieved, the following definitions are needed. Let T be a rooted spanning tree with root r. Denote the parent of u in T by $p(u)$. Assume each vertex v has been identified with a preorder number $pre(v)$, and $size(v)$ – the number of vertices in the subtree of T rooted at v. For each v, let $high(v)$ (resp. $low(v)$) denote the vertex with the highest (resp. lowest) preorder number that is either a descendant of v or adjacent to a descendant of v by a non-tree edge. Figure 4(a) shows a BFS tree with preorder labels. The *size* of the subtree rooted at vertex with label 5 is 8. The *high* value of the vertex with label 1 is 13.

For an articulation point x, $x \neq r$, classify (x, y) – an edge incident on x – as a *back* edge if

y is in the same connected component as $p(x)$ in $G - \{x\}$; otherwise, it is a *forward* edge. Define all edges incident on r to be *forward* edges whether or not r is an articulation point. Figure 3 illustrates these definitions.

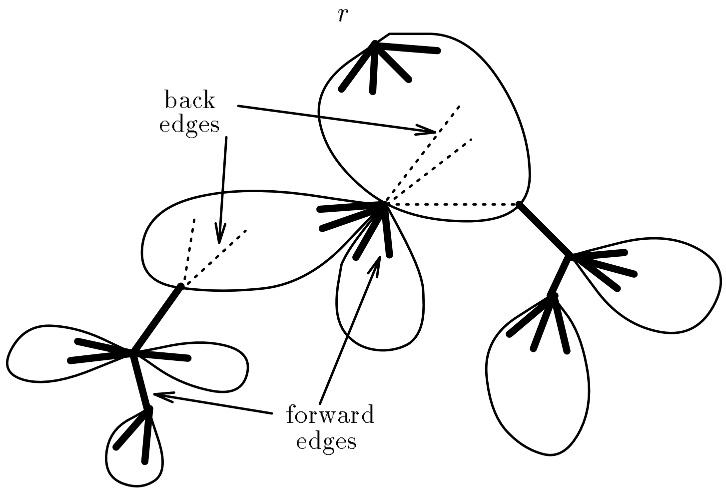

Figure 3: Forward (bold) edges and back (dotted) edges.

Our algorithm finds connected components of the subgraph obtained by deleting all forward edges from G. If two edges are in the same connected component in this subgraph, then clearly they are in the same block in G. Thus, all edges that are not forward edges can be labeled easily with block labels. To label forward edges, we take advantage of the inherent "tree" structure by which blocks are connected. To keep the algorithm simple, we ensure that all forward edges are tree edges by picking T to be an SFS tree. Finally, forward edges are identified using the pre, $high$, low and $size$ values of its end points.

Algorithm 4 *Biconnected Components*

Input: A $G = (V, E)$, a rooted SFS tree T with root $R(T)$, $pre(v)$, $size(v)$, $low(v)$ and $high(v)$.
Output: A label $\beta(u, v)$ for each edge (u, v). Two edges get the same β label iff they are in the same block.

1. Identify a subgraph H of G whose connected components are useful in finding biconnected components of G and represent it by associating a weight of 0 with its edges:
   (a) Assign a weight of 0 to all edges in E.
   (b) For an edge (u, v) of T, where $u = p(v)$, if $pre(u) \leq low(v)$ and $high(v) < pre(u) + size(u)$, change the weight of (u, v) to 1.
2. For every connected component C of H, label every $v \in V(C)$ with $l(v) = \min\{pre(u) \mid u \in V(C)\}$
3. Find the biconnected component labels for all edges in E: For every edge (u, v), $\beta(u, v)$ is the maximum of $l(u)$ and $l(v)$.

End

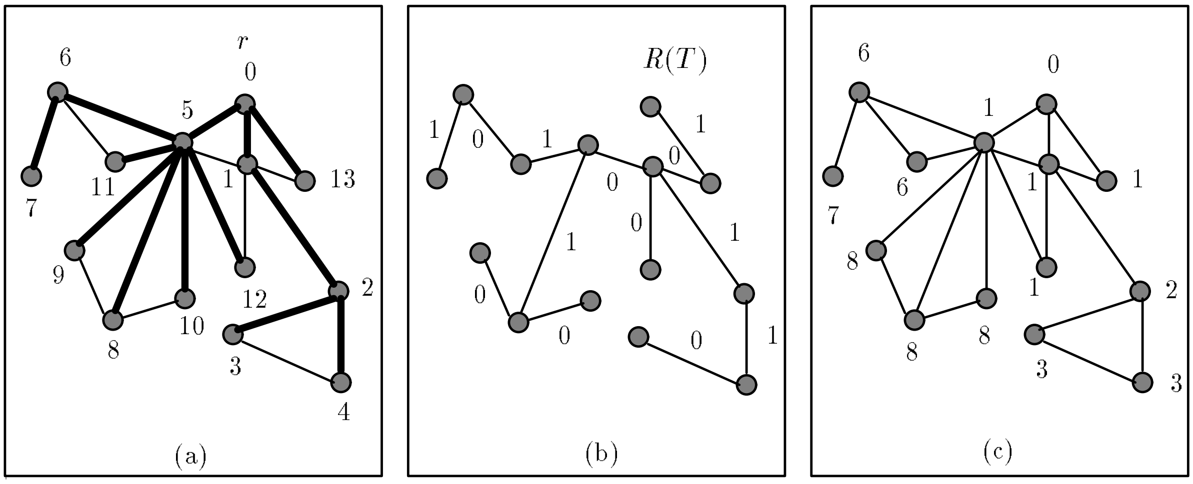

Figure 4: Computing biconnected connected components: (a) A graph with a rooted SFS tree (bold) and a preorder numbering of vertices. (b) A rooted MST T for the weighted version of the graph. (c) The l labels for the vertices.

Lemma 2 *Let B_i, $1 \leq i \leq k$, be the blocks of G. Let u_i be the vertex with the smallest preorder label in B_i. Then, for all i, $1 \leq i \leq k$, an edge of B_i belongs to H if and only if that edge is not incident on u_i. In other words, $H = G - \{forward\ edges\}$.*

Proof : Consider a B_i and corresponding u_i. If $u_i = r$, then clearly all edges incident on r pass the test in Step 1(b) and hence do not belong to H. Let $u_i \neq r$. We claim that none of the children of u_i that belong to B_i have descendants that are adjacent to a non-descendant of u_i. Assume not. Then, there is a child x of u_i and y a descendant of x that is adjacent to a non-descendant z of u_i. That is, including (y, z) in T creates a cycle that includes the edges $(p(u_i), u_i)$ and (u_i, x). Hence $(p(u_i), u_i)$ belongs to B_i. Then, the smallest preorder label in B_i is less than or equal to $pre(p(u_i))$ which is strictly less than $pre(u_i)$, contradicting that u_i is the vertex with the smallest preorder label in B_i. Given the claim, it is easy to see that the children of u_i that belong to B_i must have their low and $high$ values in the closed interval $[pre(u_i), pre(u_i) + size(u_i) - 1]$. Now observe that since T an SFS tree and u_i is visited before any other vertices of B_i, all vertices of B_i incident on u_i become children of u_i in T when u_i is scanned. In other words, if (u_i, v) is an edge of B_i, then it belongs to T and $pre(u_i) \leq low(v)$ and $high(v) < pre(u_i) + size(u_i)$. Therefore, (u_i, v) does not belong to H.

Next, we show that if an edge (a, b) is not incident on u_i, for some i, then it belongs to H. Assume not. Clearly, (a, b) must come from T as only tree edges undergo a change in their weights (see Step 1(b)). Let (a, b) belong to B_1 without loss of generality and that a is the parent of b. If a is articulation point that separates $(p(a), a)$ from the edges of B_1, then a must have the smallest preorder label in B_1 as it is the only entry point into B_1 from $p(a)$. That is $a = u_1$. Since we assumed (a, b) is an edge that is not incident on u_i, for any i, a cannot be an articulation point that separates $(p(a), a)$ and from the edges of B_1. That is $(p(a), a)$ and (a, b) are in the same block. Hence there is a simple cycle containing them. This cycle contains an edge that connects a descendant of b to a non-descendant of a. Therefore, either $low(b) < pre(a)$ or $high(b) \geq pre(a) + size(a)$. ∎

Corollary 1 *Assume that two edges (a, b) and (c, d) belong to two different blocks in G and that $a \neq d$. Then, a and d belong to different components in H.*

Proof : Let (a, b) and (c, d) belong to B_i and B_j. Since (a, b) and (c, d) belong to different blocks, there exists a vertex u in B_j, an articulation point, that is present on every path between a and d. Assume, without loss of generality, that a vertex of B_i is visited during the preorder before any of the vertices of $B_j - \{u\}$. In that case, u must be the vertex with the smallest preorder label in B_j. By the previous lemma, all edges of B_j incident on u are absent in H. Therefore, there is no path between a and d in H. ∎

Theorem 5 *Algorithm 4 is correct.*

Proof : To prove that all edges of a block get the same label, consider a block B of G with more than one edge. Let u and v be the vertices of B with the smallest and the next smallest preorder labels. By Lemma 2, H contains all edges of B except the ones incident on u. Now, consider any two vertices a and b of B different from u. As B is biconnected, there are at least two vertex-disjoint paths between a and b in G. At most one them contains u. Since the only edges of B that are missing in H are the ones incident on u, a and b are connected in H. That is, all vertices of B except u are in the same connected component in H. From Corollary 1, we know that if two vertices are in the same connected component in H, then there is a block in G that contains both those vertices. Therefore, all vertices of B except u get the same l label which, by Step 2, is $pre(v)$. The β label for all edges (x, y) of B not incident on u would be $\max(l(x), l(y)) = \max(pre(v), pre(v)) = pre(v)$. The edges of B that are incident on u are labeled with a β label equal to $\max(l(u), pre(v))$. Since $l(u) \leq pre(u)$, this is simply $pre(v)$. Therefore, all edges of B get labeled with the same β label which is equal to $pre(v)$.

Next we will prove that if two edges get the same label, then they are in the same block in G. Assume not. Consider two edges (a, b) and (c, d), $a \neq d$, that belong to different blocks B_i and B_j in G but which get the same β label. Since a and d are in different components in H by Corollary 1, $l(a) \neq l(d)$ by Step 2. Consequently, by Step 3, it is not possible that $l(a) \geq l(b)$ and $l(c) \leq l(d)$. There are three other cases.

1. $l(a) \geq l(b)$ and $l(c) > l(d)$. Then, $\beta(a, b) = \beta(c, d) = l(a) = l(c)$. In other words, a and c are in the same connected component in H. As a and d are in different components in H, c must be an articulation point in G. Since $l(c) > l(d)$, the vertices d and c are visited before the vertices of $B_i - \{c\}$ during the preorder traversal. Then, c must be vertex with the smallest preorder label in B_i. But then c and a must be in different components in H by Lemma 2, contradicting that $l(a) = l(c)$.

2. $l(a) < l(b)$ and $l(c) \leq l(d)$. Similar to the previous case.

3. $l(a) < l(b)$ and $l(c) > l(d)$. Then, $\beta(a,b) = \beta(c,d) = l(b) = l(c)$. In other words, b and c are in the same connected component in H. Since $l(a) < l(b)$, a and b are in different components in H. If $pre(a) < pre(d)$, then c is visited before any vertex from $B_j - \{c\}$ contradicting that $l(c) > l(d)$. The possibility $pre(a) > pre(d)$ can be ruled out similarly.

Therefore, two edges (a,b) and (c,d) are in the same block if and only if (a,b) and (c,d) get assigned the β same label. ∎

Consider implementing Algorithm 4 in a distributed environment so that it runs in sublinear time. We can assume without loss of generality that T is a BFS tree as all BFS trees are also SFS trees. Section 3 shows implementation for $pre(v)$ and $size(v)$. The same distributed time can also be achieved for $high(v)$ as follows. In a given round, each node computes the maximum of its preorder number, preorder number of its neighbors, and the values received, if any, from its children. This computed value is passed up to the parent, if one exists, in the next round. By D rounds, each node v would have its $high(v)$ value. Computation of $low(v)$ is similar. That only leaves the computation of connected components – Step 2. It is shown below that by employing techniques that were used to do distributed preorder, we can achieve sublinear time for this as well.

Algorithm 5 *Distributed Connected Components*

Input: A $G = (V,E)$ with a subgraph H whose edges are marked with a weight of 1. The edges not belonging to H have a weight of 0. A BFS tree T_b.
Output: A label $l(u)$ for each vertex u. Two vertices get the same l label iff they are in the same connected component in H.

1. Divide G into fragments $\mathcal{F}$ and find a rooted MST T of G with root $R(T)$. Label each vertex v of T with a preorder label $pre_t(v)$. Designate the root r of each fragment $\mathcal{F}$ to be the vertex of $\mathcal{F}$ with the smallest pre_t value.
2. Compute the connected components of H restricted to each fragment by finding a label $l_f(v)$ for each vertex v of $\mathcal{F}$.
3. For each v, compute $l(v)$ using $l_f(v)$.
   (a) For each inter-fragment edge (u,v) of T, send the weight of the edge, $l_f(u)$ and $l_f(v)$ to $R(T_b)$ – the root of T_b.
   (b) At $R(T_b)$ build a graph J whose vertices are vertices of T that have an inter-fragment edge (either incoming or outgoing) incident on them. Add an edge between u and v in J if either u and v are in the same fragment and $l_f(u) = l_f(v)$, or u and v are in different fragments but are connected by an edge of weight 0.
   (c) Compute the connected components of J and label each component with the smallest l_f label that belongs to that component. These labels are the l labels for all vertices incident on inter-fragment edges.
   (d) Extend l labels to all the vertices of G: Broadcast down $l(r)$ within each fragment along the zero-weighted tree edges from the root r. If a node u receives a label during this broadcast, $l(u)$ is the value received; otherwise $l(u)$ is $l_f(u)$ computed in Step 2 above.

End

Theorem 6 *The distributed time complexity of the above algorithm is $O(D + n^{0.614})$. The above algorithm is correct.*

Proof : Recall that D, N and d represent, respectively, the diameter of G, the number of fragments and the diameter of the largest fragment. It follows from [GKP 93] and Theorem 4 that Step 1 takes $O(D + n^{0.614})$ time. Step 2 can be implemented to run in $O(d)$ time as follows. Find the minimum of $pre_t(v)$ and the preorder numbers received, if any, from its children. If the edge connecting v to its parent (in T) belongs to $\mathcal{F}$ and has weight 0, then send this value up. Otherwise, denote it $l_f(v)$. Next, send this value down to all the children of v in $\mathcal{F}$ that are connected by zero-weighted edges. For a node u that has the parent in $\mathcal{F}$ connected by an edge of weight 0, $l_f(u)$ is the value u receives from its parent. Consider Step 3. Part (a) takes $O(N + D)$ time. Parts (b) and (c) are computed centrally and take constant time. Part (d) takes at most $O(d)$ time. That establishes the time bound. The correctness can be seen from the following facts. The connected component computation is performed hierarchically. That is, first components within each fragments are labeled, and then the labeling is extended to the whole graph. The connected components of H and that of the subgraph of zero-weighted edges of T are identical by Lemma 1. In other words, for any two vertices u and v, u and v are in the same connected component in H if and only if there is a path of zero-weighted edges connecting u and v in T. ∎

5 Conclusion

We presented sublinear time algorithms for finding sparse certificates and biconnected components. The reduction in time was achieved using the paradigm introduced by [GKP 93]. We demonstrated that this methodology of dividing a problem into subtasks, solving them in parallel and combining the results centrally at the root of a BFS tree is useful in contexts other than MST computation. It would be nice to see more applications of this technique.

If certificate algorithms are to be implemented, we suggest that the presented algorithms be modified so that the first spanning tree is a BFS tree (instead of an SFS tree). This way one can preserve not only connectivity but also the diameter of the original graph.

References

[AP 90] B. Awerbuch and D. Peleg, "Network synchronization with polylogarithmic overhead," *Proc. 31st Symp. Found. Comp. Sci.* (1990), pp. 514–522.

[CKT 93] J. Cheriyan, M. Kao and R. Thurimella, "Scan-first search and sparse certificates: an improved parallel algorithm for k-vertex connectivity," *SIAM J. on Computing* **22**:1, pp. 157–174.

[GKP 93] J. Garay, S. Kutten and D. Peleg, "A sub-linear time distributed algorithm for minimum-weight spanning trees," *Proc. 34th Symp. Found. Comp. Sci.* (1993), pp. 659–668.

[GJ 79] M. R. Garey and D. S. Johnson, "Computers and intractability: A guide to the theory of NP-completeness", W. H. Freeman, San Francisco, 1979.

[Ha 87] V. Hadzilacos, "Connectivity requirements for Byzantine agreement under restricted types of failures," *Distributed Computing* **2** (1987), pp. 95–103.

[Ho 90] W. Hohberg, "How to find biconnected components in distributed networks," *J. Parallel and Dist. Computing* **9**:4 (1990), pp. 374–386.

[Hu 89] Huang, S.T., "A new distributed algorithm for the biconnectivity problem," *Proc. 1989 Int. Conf. on Parallel Processing* (1989), Vol. III, pp. 106–113.

[IR 88] A. Itai and M. Rodeh, "The multi-tree approach to reliability in distributed networks," *Information and Computation* **79** (1988), pp. 43–59.

[NI 90] H. Nagamochi and T. Ibaraki, "Linear time algorithms for finding k-edge-connected and k-node-connected spanning subgraphs", *Algorithmica*, **7** (1992), pp. 583–596.

[T 89] R. Thurimella, *Techniques for the design of parallel graph algorithms*, Ph.D. Thesis, The University of Texas at Austin, August 1989.

[TV 84] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. Computing* **14** (1984), pp. 862–874.