Efficient Parallel Algorithms for Testing k–Connectivity and Finding Disjoint s-t Paths in Graphs*

Samir Khuller †
Computer Science Department
Upson Hall
Cornell University
Ithaca, NY 14853

Baruch Schieber
IBM Research Division
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

We present an efficient parallel algorithm for testing whether a graph G is k-vertex connected. The algorithm runs in $O(k^2 \log n)$ time and uses $nk^2C(n,m)$ processors on a CRCW PRAM, where n and m are the number of vertices and edges of G, and $C(n,m)$ is the number of processors required to compute the connected components of G in logarithmic time. For fixed k, the algorithm runs in logarithmic time and uses $nC(n,m)$ processors. To develop our algorithm we design an efficient parallel algorithm for the following *disjoint s-t paths* problem: Given a graph G, and two specified vertices s and t, find k vertex disjoint paths between s and t, if they exist. If no such paths exist, find a set of at most $k-1$ vertices whose removal disconnects s and t. Our parallel algorithm for this problem runs in $O(k^2 \log n)$ time and uses $kC(n,m)$ processors. We show how to modify the algorithm to find k-edge disjoint paths, if they exist. This yields an efficient parallel algorithm for testing whether a graph G is k-edge connected. The algorithm runs in $O(k^2 \log n)$ time and uses $nkC(n,kn)$ processors on a CRCW PRAM. Finally, we describe more applications of the disjoint s-t paths algorithm.

*An extended summary of this paper appears in *Proc. 30th IEEE Symp. on Foundations of Computer Science*, October 1989.

†The research of this author is supported by an IBM Graduate Fellowship, NSF grant DCR 85-52938, and funds from AT&T Bell Labs and Sun Microsystems. Part of this research was done while this author was visiting the IBM T.J. Watson Research Center.

1. Introduction

Graph Connectivity is considered as one of the classical subjects in Graph Theory [Har69, Ber76, Eve79], and has many practical applications, e.g., in chip and circuit design, reliability of communication networks and cluster analysis. Designing efficient parallel algorithms for graph connectivity is clearly a basic problem in parallel computation. Efficient parallel algorithms for testing connectivity [SV82, CV86], biconnectivity [TV85], triconnectivity [FT88, FRT89], and four-connectivity [KR87] of graphs have been developed. In this paper we present an efficient parallel algorithm for testing whether a graph is k-vertex connected, for any fixed k. The problem is formally stated as follows: Given an undirected graph $G(V, E)$ and a fixed integer k, test whether G is k-vertex connected. If the graph is not k-vertex connected, find a set of at most $k-1$ vertices whose removal disconnects G.

In order to solve the connectivity problem, we solve the following *disjoint s-t paths* problem: Given an undirected graph G, and two specified vertices s and t, find k-vertex disjoint paths between s and t. If no such paths exist, obtain a set of at most $k-1$ vertices whose removal disconnects s and t. Even [Eve75] observed that in order to test for k-vertex connectivity it is sufficient to check whether there are k-vertex disjoint paths between kn pairs of vertices.

The model of parallel computation used is the Concurrent-Read Concurrent-Write (CRCW) Parallel Random Access Machine (PRAM) [Wyl79]. A PRAM employs p synchronous processors all having access to a shared memory. A CRCW PRAM allows simultaneous access by more than one processor to the same memory location for both read and write purposes. In case several processors attempt to write simultaneously in the same memory location, an arbitrary one succeeds in doing the write. We remark that our algorithm can also be implemented in weaker PRAM models. The complexity of our algorithm when implemented on such models is the same as the complexity of finding connected components. However, to make the presentation simpler we concentrate on the CRCW PRAM model.

A parallel algorithm is said to have *optimal speedup* if its time-processor product is the same as the time complexity of the best known sequential algorithm for the same problem.

Our parallel algorithm for testing k-vertex connectivity runs in $O(k^2 \log n)$ time using $nk^2C(n, m)$ processors, where $n = |V|$, $m = |E|$ and $C(n, m)$ is the number of processors required to compute the connected components of a graph with n vertices and m edges in $O(\log n)$ time. For fixed k, our algorithm runs in logarithmic time using $nC(n, m)$ processors. From now on, throughout the paper we assume that k is fixed, and all factors of k will be elided from the complexity bounds.

Using the logarithmic time parallel connectivity algorithm [CV86], the bound for $C(n, m)$

is $(m+n)\alpha(m,n)/\log n$ processors, where $\alpha(m,n)$ is the functional inverse of Ackermann's function, as defined in [Tar75]. Recall that the time complexity of the best known deterministic sequential algorithm for testing k-vertex connectivity is $O(mn)$, for any fixed $k > 4$ [Eve75, Gal80, BDD$^+$82]. Thus, the existence of an optimal logarithmic time parallel algorithm for connectivity would make our k-vertex connectivity algorithm achieve optimal speedup, for any fixed $k > 4$. The main component of our parallel k-connectivity algorithm is an efficient parallel algorithm for the disjoint s-t paths problem. This algorithm runs in $O(\log n)$ time using $C(n,m)$ processors. Again, the existence of an optimal logarithmic time parallel algorithm for connectivity would make our parallel disjoint s-t paths algorithm achieve optimal speedup.

The previous best deterministic parallel algorithm for testing k-vertex connectivity, for $k > 4$, required $O(\log^2 n)$ time and $nM(n)$ processors (where $M(n) = \Omega(n^2)$ is the number of arithmetic operations required to multiply two $n \times n$ matrices). Similar to our algorithm, this algorithm tests k-vertex connectivity by finding k-vertex disjoint paths between pairs of vertices. The disjoint paths are found using a straightforward parallelization of the sequential algorithm [KMV89]. The parallelization suffers from the problem of being inefficient, requiring $O(\log^2 n)$ time and $M(n)$ processors. A randomized parallel algorithm for testing k-vertex connectivity is given in [LLW86]. It runs in $O(\log^2 n)$ time and uses $n^{2.5}$ processors. Compared to our algorithm, this algorithm uses fewer processors for dense graphs but has the disadvantage of using randomization, and takes $O(\log^2 n)$ time compared to our $O(\log n)$ time bound.

The sequential algorithm for testing connectivity reduces the problem to several flow problems in appropriately defined networks. This reduction yields an efficient sequential algorithm. However, the flow problem does not seem amenable to efficient parallelism since it involves computing reachability in directed graphs. Thus, obtaining an efficient parallel algorithm requires further insights into the problem. In our algorithm we reduce the connectivity problem to a reachability problem in directed graphs, called *bridge graphs*. We are able to exploit the structure of these *bridge graphs* to obtain an efficient parallel reachability algorithm.

This is not the first time that the techniques used for efficient sequential algorithms are not adequate for the respective parallel algorithms. For example, the sequential Depth First Search (DFS) algorithm is very efficient, and can be used to obtain efficient sequential algorithms for many other problems, such as: connectivity, biconnectivity, st-numbering. However, it seems that DFS is not amenable to efficient parallelism. In order to obtain efficient parallel algorithms for each of these problems, new techniques have to be developed. Examples of these techniques are the Euler tour technique [TV85] and Ear Decomposition Search [MSV86].

Our k-vertex disjoint paths algorithm can be extended to obtain k-edge disjoint paths with the same complexity. The algorithm finds either k-edge disjoint paths, or a set of at most $k-1$ edges whose removal disconnects s and t. This yields a parallel algorithm to test a graph

for k-edge connectivity that runs in $O(\log n)$ time using $nC(n,n)$ processors, improving on the previous parallel algorithms in both time and number of processors. The best sequential algorithm for testing k-edge connectivity runs in $O(n^2)$ time, for any fixed $k > 2$ [Mat87]. Thus, the existence of an optimal logarithmic time parallel algorithm for connectivity would make our k-edge connectivity algorithm achieve optimal speedup, for any fixed $k > 2$.

In addition to its application to connectivity algorithms, the disjoint s-t paths algorithm has several other applications. It can be used in protocols for reliable communication over networks as described in [IR84]. In [KMV89] it is used to obtain efficient parallel algorithms for the two paths problem ([Sey80, Shi80]). We use it to obtain efficient parallel algorithms for the subgraph homeomorphism problem for some fixed pattern graphs.

We remark that for the case $k = 2$, the disjoint s-t paths problem can be solved using an st-numbering of the graph. Unfortunately, the technique does not seem to generalize to solving the problem for any $k > 2$. For completeness, we describe this solution.

The paper is organized as follows. In Section 2 we give some preliminary definitions and background. Before giving a description of the entire algorithm, we describe the simpler case when $k = 2$, in Section 3. We also describe how to solve the problem when $k = 2$ using st-numbering. In Section 4, we describe the algorithm for finding k vertex disjoint s-t paths. In Section 5, we show how to modify the algorithm to get k edge disjoint s-t paths. Finally, Section 6 describes some more applications of the disjoint paths algorithm.

Remark: The algorithm for $k = 2$, given in Section 3, is a special case of the algorithm described in Section 4, and is included only to make the presentation clearer. Since Section 4 is self-contained, some of the readers might find it useful to skip over Section 3.

2. Connectivity and disjoint paths

Let $G(V,E)$ be a simple undirected graph. Without loss of generality we will assume that G is connected. Let s and t be distinct vertices in V that are not connected by an edge. An (s,t) vertex separator is a set of vertices $S_V \subset V$, such that every path from s to t contains at least one vertex from S_V. Define $N(s,t)$ to be the minimum cardinality of an (s,t) vertex separator. By Menger's theorem [Eve79, BM77] there are exactly $N(s,t)$ vertex disjoint paths between s and t. The vertex connectivity λ_V of G is defined as $\lambda_V = \min \{ N(s,t) \mid \{s,t\} \subset V, s \neq t, (s,t) \notin E\}$. In other words, the vertex connectivity of G is the cardinality of the smallest set of vertices whose removal disconnects G. Note that there are λ_V vertex disjoint paths between any pair of vertices in G.

We define an *articulation vertex* to be a vertex whose removal from G (together with its incident edges) disconnects G. A graph is said to be *biconnected* if its vertex connectivity is at least two. The biconnected components of a graph are its maximally biconnected subgraphs. A graph whose vertex connectivity is at least k, is called *k-vertex connected.*

From the definition of vertex connectivity, it follows that testing whether G is k-vertex connected can be done by checking if $N(s,t) \geq k$, for all n^2 pairs of vertices.

In [DF56, ET75] it was observed that $N(s,t)$ can be computed using max-flow techniques. Given the graph $G(V,E)$, construct a network $N_{s,t}(V,E)$, where each vertex (excluding s and t) and edge has unit capacity. It is not difficult to see that $N(s,t) \geq k$, if and only if the value of the max-flow from s to t in $N_{s,t}$ is at least k.

In [Eve75] it was observed that to test whether G is k-vertex connected it is sufficient to check whether $N(s,t) \geq k$, for only kn pairs.

Let $V_k = \{v_1, \ldots, v_k\}$ be a set of k vertices of G. (W.l.o.g. assume that $|V| > k$.)

Claim: If $N(v_i, u) \geq k$ for all $i = 1, \ldots, k$ and all $u \in V$, then G is k-vertex connected.

Proof: Suppose that the graph is not k-vertex connected. This implies that there exists a set of at most $k-1$ vertices whose removal disconnects G into at least two components C_1 and C_2. Since the size of the separating set is $< k$, at least one vertex v_i is in one of the components. W.l.o.g. assume that v_i is in C_1. Let u be a vertex in C_2. Clearly, $N(v_i, u) < k$, yielding a contradiction to the assumption. □

We conclude that we can test whether G is k-vertex connected by running in parallel kn copies of an algorithm for obtaining k-vertex disjoint paths. We will describe an algorithm for obtaining k-vertex disjoint paths that takes $O(\log n)$ time and $C(n,m)$ processors. This yields an $O(\log n)$ time algorithm that uses $nC(n,m)$ processors for testing whether a graph is k-vertex connected.

The edge connectivity of G is defined in a similar way. Let s and t be distinct vertices in V. An (s,t) edge separator is a set of edges $S_E \subset E$, such that every path from s to t uses at least one edge from S_E. Define $M(s,t)$ to be the minimum cardinality of an (s,t) edge separator. By Menger's theorem [Eve79, BM77], there are exactly $M(s,t)$ edge disjoint paths between s and t. The edge connectivity λ_E of G is defined as $\lambda_E = \min\{\, M(s,t) \mid \{s,t\} \subset V,\, s \neq t\}$. In other words, the edge connectivity of G is the cardinality of the smallest set of edges whose removal disconnects G. Note that there are λ_E edge disjoint paths between any pair of vertices in G.

To test whether a graph is k-edge connected we can check whether $M(s,t) \geq k$, for all n^2 pairs of vertices. As in the vertex case, $M(s,t)$ can also be computed using max-flow techniques

([DF56, ET75]). In the corresponding network $M_{s,t}$ each edge has unit capacity.

It is easy to see that it is sufficient to check whether $M(s,t) \geq k$, for only n pairs.

Claim: Let v be a vertex in G. If $M(v,u) \geq k$ for all $u \in V - \{v\}$, then G is k-edge connected.

Proof: Suppose that the graph is not k-edge connected. This implies that there exists a set of at most $k-1$ edges whose removal disconnects G into at least two components C_1 and C_2. W.l.o.g. assume that v is in C_1. Let u be a vertex in C_2. Clearly, $M(v_i, u) < k$, yielding a contradiction to the assumption. □

Recently, Thurimella [Thu89] observed that to test whether G is k-edge connected it is sufficient to consider a sparse spanning subgraph of G.

Let F_1 be a maximal spanning forest (tree) of G. For $i = 2, \ldots, k$, let F_i be a maximal spanning forest of $G - \bigcup_{j=1}^{i-1} F_j$. Define $H = \bigcup_{i=1}^{k} F_i$. Notice that H has $O(kn)$ edges.

Theorem 2.1 (Thurimella): *The graph G is k-edge connected if and only if its subgraph H is k-edge connected.*

Proof: The *if* direction is trivial. We prove the *only if* direction by contradiction. Assume that G is k-edge connected and H is not. This implies that there exists a set of at most $k-1$ edges whose removal disconnects H into at least two components (say C_1 and C_2). On the other hand, there are at least k edges between C_1 and C_2 in G. Since the edges of the spanning forests are pairwise disjoint, and there are k such forests, at least one such forest, say F_i, does not contain any edge between C_1 and C_2. Thus, in F_i, the components C_1 and C_2 are disconnected. However, in $G - \bigcup_{j=1}^{i-1} F_j$, which is spanned by F_i, there is at least one edge between C_1 and C_2; a contradiction. □

It follows that we can test whether G is k-edge connected by running n copies of an algorithm for obtaining k-edge disjoint paths in parallel on a sparse subgraph H of G, that has $O(kn)$ edges. We construct H by k applications of the parallel connectivity algorithm. This takes $O(\log n)$ time and $C(n,m)$ processors. We will describe a parallel algorithm for obtaining k-edge disjoint paths in a graph with n vertices and m edges, that runs in $O(\log n)$ time and $C(n,m)$ processors. This yields a $O(\log n)$ time algorithm that uses $nC(n,n)$ processors for testing whether a graph is k-edge connected.

We conclude the preliminaries with the following definition of *bridges*. Given a graph $G(V,E)$, let H be a subgraph of G, and let e and f be edges of G not in H. Define the equivalence relation $=_H$ by $e =_H f$, if and only if there is a path in G that includes e and f and has no internal vertices in common with $V(H)$. The subgraphs induced by the edges of the equivalence classes of $E(G) - E(H)$ under $=_H$ are called the *bridges* of G relative to H. The

attachment vertices of a bridge B relative to H are the vertices in $V(B) \cap V(H)$.

In Fig. 1, H is a simple cycle in G. The bridge B_1 is a trivial bridge consisting of only one edge, the two other bridges B_2 and B_3 are non-trivial.

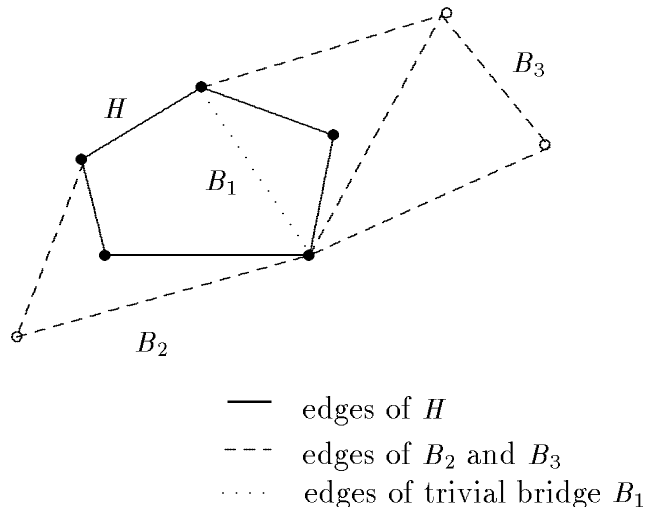


Figure 1: An example illustrating *bridges*

3. Finding two vertex disjoint paths

Let $G(V, E)$ be an undirected graph, with s and t vertices that are not connected by an edge. In this section, we develop a parallel algorithm for finding either two vertex disjoint paths between s and t, or an articulation vertex separating s and t. We will subsequently show how this algorithm can be generalized to finding k-vertex disjoint s-t paths, with the same complexity bounds.

High level description:

The algorithm consists of five steps.

Step 1. Find a path P_1 from s to t.

Step 2. Decompose G into its bridges relative to the path P_1.

If there is a single bridge that has both s and t as its attachment vertices, then a path P_2 between s and t in this bridge is vertex disjoint from P_1, yielding the two disjoint paths. Suppose that there is no such bridge.

Define a linear order on the attachment vertices according to their position on P_1. An attachment vertex a is said to be less than an attachment vertex b, if a is to the left of b on P_1. (We assume that s is the leftmost vertex on P_1 and t is the rightmost.) For each bridge B_i, define l_i to be its leftmost attachment vertex, i.e., the attachment vertex that is closest to s on P_1. Similarly, define r_i to be its rightmost attachment vertex; i.e., the attachment vertex that is furthest from s on P_1.

Step 3. Construct a new directed graph $G_B(V_B, E_B)$, called the *bridge graph*, as follows:

$$V_B = \{\beta_i \mid B_i \text{ is a bridge}\} \cup \{s, t\}$$

The set of edges E_B is defined as follows. The vertex s has an outgoing edge to a vertex β_j, if $l_j = s$, and r_j is furthest from s, among all bridges B_z, with $l_z = s$. A vertex β_i has an outgoing edge to t if $r_i = t$. If $r_i \neq t$, then we define an outgoing edge from β_i as follows: each vertex β_i has an outgoing edge to a vertex β_j, if r_j is the rightmost vertex on P_1, among the attachment vertices of bridges whose leftmost attachment vertex is to the left of r_i, and $r_j > r_i$.

Remark: When the maximum is achieved for more than one bridge, any one is chosen arbitrarily. This implies that the outdegree of every vertex of G_B is at most one.

Step 4. Find a (directed) path from s to t in G_B, if one exists.

We prove below that such a path in G_B exists if and only if there are two vertex disjoint paths between s and t. We also show how to construct these two paths, given the path in G_B.

Step 5. If a path from s to t in G_B was found, construct two vertex disjoint paths between s and t. (The construction is given below.) Otherwise (there is no path from s to t in G_B), if there is no bridge with s as an attachment vertex, then the neighbor of s on P_1 separates s and t, else, consider all the vertices reachable from s in G_B. Recall that each such vertex corresponds to a bridge of G relative to P_1. Let w be the attachment vertex of one of these bridges which is furthest from s. The vertex w separates s from t.

Correctness: To prove the correctness of the algorithm we prove the following theorem.

Theorem 3.1: *There are two vertex disjoint paths between s and t in G if and only if there is a directed path from s to t in G_B.*

Proof: The *if* direction: Suppose that there is a directed path $P_B = s, \beta_{x_1}, \ldots, \beta_{x_a}, t$ from s to t in G_B. To prove that there are two vertex disjoint paths from s to t in G, we show that the value of the max-flow in the network $N_{s,t}$ is at least two.

Observe that the path P_1 corresponds to a path in $N_{s,t}$ from s to t. Using P_1 we can push one unit of flow from s to t. Our goal is to show that P_B defines an augmenting path in $N_{s,t}$ given the flow corresponding to P_1.

For each vertex β_{x_i} on P_B, let R_{x_i} be the path in B_{x_i} from l_{x_i} to r_{x_i}. (Note that $l_{x_1} = s$ and $r_{x_a} = t$.)

Given a path P, let $P[l;r]$ denote its segment from l to r. We define P_2 to be a path from s to t in G as follows.

$$P_2 = R_{x_1}; P_1[r_{x_1}; l_{x_2}]; R_{x_2}; \ldots; R_{x_{a-1}}; P_1[r_{x_{a-1}}; l_{x_a}]; R_{x_a}.$$

Path P_2 consists of two kinds of segments: segments belonging to bridges and segments belonging to P_1 that "reverse" on P_1 (see Fig. 2). (We view P_1 as directed from s to t.)

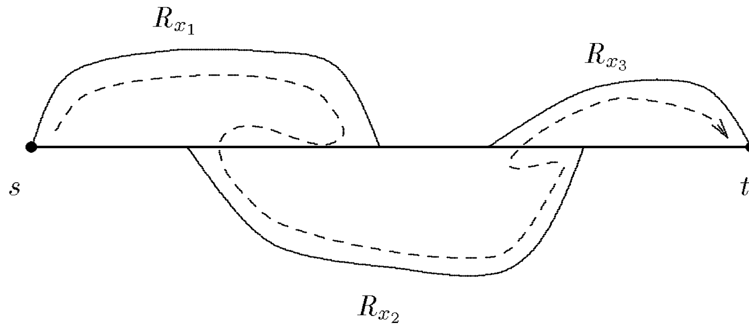


Figure 2: Path P_2

We claim that P_2 corresponds to an augmenting path in $N_{s,t}$, yielding a flow of two units (see Fig. 3). This follows from the following lemma.

Lemma 3.2: *After pushing one unit of flow along P_2, the outgoing flow from each vertex (excluding s) and the incoming flow to each vertex (excluding t) is at most one.*

Proof: Before pushing the flow along P_2 we had a legal flow of value one. Clearly, by pushing flow along P_2 we change the flow only for the vertices on P_2, thus we may consider only these vertices. Consider an internal vertex v on P_2. If v is not on P_1 (that is, it is an internal vertex on some R_{x_i}), then its incoming and outgoing flow is one.

Suppose that v is a vertex on $P_1[r_{x_i}; l_{x_{i+1}}]$. From the definition of G_B it follows that $l_{x_{i+1}} < r_{x_i}$, for $1 \leq i < a$. Clearly, $l_{x_{i+2}} \geq r_{x_i}$, for $1 \leq i \leq a-2$, since otherwise β_{x_i} would have an outgoing edge to $\beta_{x_{i+2}}$. We observe that the path P_2 has the following *monotonicity*

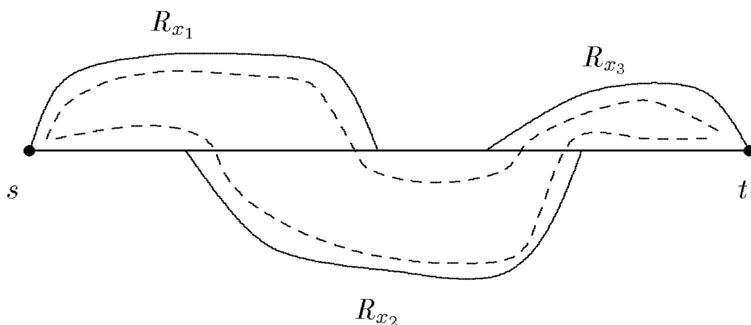


Figure 3: Obtaining two vertex disjoint s-t paths

property: $l_{x_2} < r_{x_1} \leq l_{x_3} < r_{x_2} \leq \ldots < r_{x_{a-2}} \leq l_{x_a} < r_{x_{a-1}}$. This implies if v appears more than once on P_2, then it appears twice, and $v = r_{x_i} = l_{x_{i+2}}$, for some $1 \leq i \leq a-2$ (see Fig. 4). Also note that P_2 is (edge) simple (i.e., it does not repeat any edges).

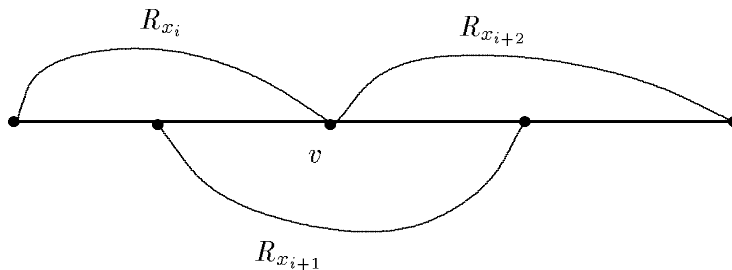


Figure 4: Vertex v occurs on P_2 twice

We distinguish between three cases: (1) The vertex v is not an endpoint of any path R_{x_i}. (2) The vertex v is a right endpoint of R_{x_i} and a left endpoint of $R_{x_{i+2}}$ (i.e., $v = r_{x_i} = l_{x_{i+2}}$). (3) The vertex v is an endpoint of one path R_{x_i} (i.e., either $v = l_{x_i}$ or $v = r_{x_i}$). (The figures describing all the cases are given in Fig. 9.) The proofs for each of these cases are given in the proof of Lemma 4.2 in the next section. (Lemma 4.2 is the version of this lemma for an arbitrary k.) □

The *only if* direction (of Theorem 3.1): Assume that there is no path from s to t in G_B. If there is no bridge with s as an attachment vertex, then the neighbor of s on P_1 separates s and t. Consider all the vertices reachable from s in G_B. Recall that each such vertex corresponds to a bridge of G relative to P_1. Let w be the attachment vertex of one of these bridges, say B_i, which is furthest from s. We claim that w separates s and t. Remove w and all its incident edges from G, and assume that there exists a path Q from s to t in the remaining graph.

Consider the vertices of Q that are also on P_1 in the order of their appearance on Q. Observe that there must be two such successive vertices w_1 and w_2 such that $w_1 < w$ and $w_2 > w$. Clearly, the sub-path of Q from w_1 to w_2 does not contain any vertex of P_1. Hence, there exists a bridge B_j relative to P_1 such that $l_j < w$ and $r_j > w$, contradicting the definition of w. □

Implementation and complexity: We show how to implement the algorithm in $O(\log n)$ time and $C(n,m)$ processors on a CRCW PRAM. In the following discussion we will assume that G is represented by its adjacency list.

Step 1. We find the path P_1 from s to t, by computing a spanning tree of G, and following the unique path from s to t in this tree. This is done in $O(\log n)$ time and $C(n,m)$ processors using a logarithmic time parallel graph connectivity algorithm (e.g., [CV86]).

Steps 2,3. The implementation of these steps is analogous to the implementation of Steps 2 and 3 of the k disjoint paths algorithm given in the next section. The detailed description of these steps is postponed to that section.

Step 4. Observe that the outdegree of each vertex in G_B is at most one. Observe also that G_B is acyclic, since $(\beta_x \rightarrow \beta_y)$ in G_B implies that $r_x < r_y$. We conclude that G_B is a rooted forest (where the edges are directed towards the root). Since the outdegree of t is zero, t is the root of some tree in the forest. Thus, there exists a path from s to t iff s is in the tree rooted at t.

We can check if s is in the tree rooted at t by applying the Euler tour technique of [TV85]. However, to apply this technique we need the full adjacency list of the forest, i.e., we need to compute the list of incoming edges to each vertex β_i. To do this we use the following observation. Consider two edges $(\beta_x \rightarrow \beta_i)$ and $(\beta_y \rightarrow \beta_i)$ incoming to β_i. Suppose that $r_x < r_y$. Then, all the bridges with rightmost attachment vertex between r_x and r_y have an outgoing edge to β_i.

For a bridge B_i, define the *rightmost edge* to be the edge of B_i whose endpoint is the rightmost attachment of B_i. Consider the concatenation of the adjacency lists (in G) of all the vertices on P_1. Compact this list to include only the rightmost edges of bridges. It follows from the above observation that the rightmost edges of all the bridges whose corresponding outgoing edges in G_B point to the same vertex in G_B are consecutive in this concatenated list. Thus, the list of incoming edges of all the vertices in G_B can be computed using the algorithms for list ranking and prefix sums in $O(\log n)$ time and $(m+n)/\log n$ processors [CV86, AM88, LF80].

Step 5. If there is no path from s to t in G_B, the separating vertex w is the rightmost attachment vertex of the bridge corresponding to the root of the tree containing s. This rightmost attachment can be found in $O(\log n)$ time with $n/\log n$ processors using the same technique used in Step 4.

If there is a path from s to t in G_B, the two vertex disjoint paths Q_1 and Q_2 can be constructed by following the two flow paths from s to t. We define Q_1 and Q_2 recursively. First, we define the prefix of each of these paths and then for each edge on these paths we define its successive edge. The prefix of Q_1 is $P_1[s; l_{x_2}]$. The prefix of Q_2 is R_{x_1}. The successor edge for an edge $(u \rightarrow v)$, $v \neq t$, is the edge along which the unit flow leaves v. Updating the incoming and outgoing flow edges through each vertex is easily accomplished after computing P_2. Each vertex of P_2 that is also on P_1, independently adjusts its incoming and outgoing flow edges. For some vertices the flow through them is completely cancelled, and they are not on either Q_1 or Q_2. In this way, we obtain the two paths as linked lists, that can be ranked in $O(\log n)$ time using $n/\log n$ processors [CV86, AM88, CV88].

To conclude this section, we remark that two disjoint s-t paths can be obtained from an st-numbering of the graph. This alternative algorithm has the advantage that after computing this numbering once, we can find two vertex disjoint paths between *any* pair of vertices. However, it seems that this algorithm cannot be generalized to obtain k-vertex disjoint paths, for $k > 2$.

For completeness we describe this algorithm briefly. Recall the definition of an st-numbering ([LEC67, ET76]). Let $G(V, E)$ be an undirected graph and let (s^*, t^*) be an edge in G. An st-numbering of G is a 1-1 function $f : V \rightarrow \{1, \ldots, n\}$ with the following properties: (1) $f(s^*) = 1$; (2) $f(t^*) = n$; (3) any vertex $v \in V - \{s^*, t^*\}$ has at least one adjacent vertex u with $f(u) < f(v)$ and at least one adjacent vertex w with $f(w) > f(v)$. It is not difficult to see that a graph is biconnected if and only if it has an st-numbering starting from any edge (s^*, t^*) ([LEC67]).

The algorithm consists of two stages: a preprocessing stage, in which an st-numbering is computed, and a processing stage which uses this numbering to obtain two vertex disjoint paths.

In the preprocessing stage we decompose G into its biconnected components, and compute an st-numbering for each biconnected component.

We now show how to find two vertex disjoint paths between s and t. First, we check to ensure that s and t are in the same biconnected component.

We construct a simple cycle C_s that contains both s and s^* by concatenating two vertex disjoint paths from s to s^*. One path is computed by starting at s and following a sequence of vertices with decreasing st numbers, until we reach s^*. We are guaranteed to reach s^* since each vertex (excluding s^*) has an adjacent vertex with a smaller number. Similarly, we construct a second path by starting at s and following a sequence of vertices with increasing st numbers, until we reach t^*, and then use the edge (s^*, t^*) to reach s^*. If t is on C_s then the two vertex disjoint paths from s to t are obtained from C_s. Otherwise, we compute a simple cycle C_t that contains both t and s^* in a similar way. Note that both C_t and C_s contain the edge (s^*, t^*).

Let w_1 be the first vertex on C_s to the left of s that is also on C_t, and let w_2 be the first vertex on C_s to the right of s that is also on C_t. It is easy to see that the parts of C_s and C_t from w_1 to w_2 define a simple cycle that contains both s and t, yielding two vertex disjoint paths between s and t. (See Fig. 5.)

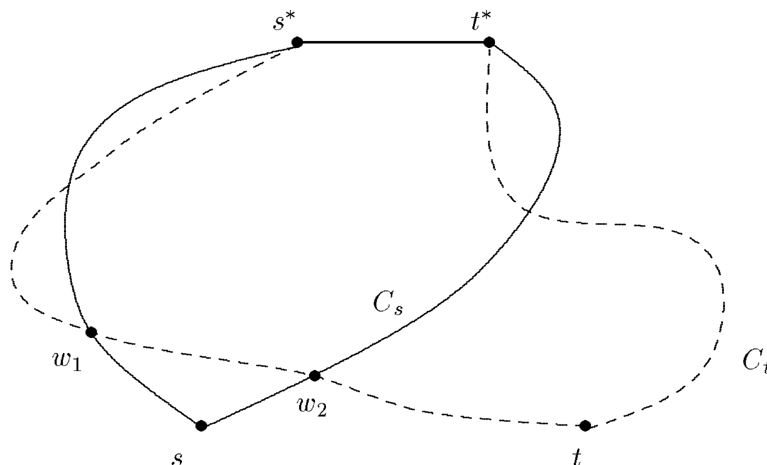


Figure 5: Obtaining two vertex disjoint s-t paths

The preprocessing stage of the algorithm can be done in $O(\log n)$ time and $C(n, m)$ processors on a CRCW PRAM, using the parallel algorithms for testing biconnectivity [TV85] and computing an st-numbering [MSV86]. Given this preprocessing the rest of the computation can be done in $O(\log n)$ time with $n/\log n$ processors, using the optimal logarithmic time parallel algorithm for list ranking [CV86, AM88].

4. Finding k vertex disjoint paths

Let $G(V, E)$ be an undirected graph, with s and t two specified vertices not connected by an edge. In this section we describe a parallel algorithm for finding either k-vertex disjoint paths between s and t, or a set of at most $k - 1$ vertices whose removal separates s from t. This algorithm is a generalization of the algorithm for obtaining two disjoint s-t paths given in the preceding section. It runs in $O(\log n)$ time and uses $C(n, m)$ processors.

High level description:

The algorithm consists of five steps.

Step 1. If $k = 1$, find a path from s to t using a spanning tree of the graph. Otherwise,

recursively find $k-1$ vertex disjoint paths from s to t, if they exist, if not, output a separating set of size $\leq k-2$.

Suppose that $k-1$ paths, $P_1, \ldots, P_{k-1}$ were found.

Step 2. Decompose the graph into its bridges relative to the subgraph $P_1 \cup P_2 \cup \ldots \cup P_{k-1}$.

If there is a bridge that has both s and t as its attachment vertices, then a path P_k between s and t in this bridge is vertex disjoint from $P_1, \ldots P_{k-1}$, yielding the k disjoint paths. Suppose that there is no such bridge.

For Step 3 we need the following definitions.

Definition 1: *Suppose that both attachment vertices a and b are on a path P_j. The vertex a is said to be* less than *b, if a is to the left of b on P_j. (We assume that s is the leftmost vertex on P_j, and t the rightmost.)*

This defines a linear order on the attachment vertices lying on a path P_j.

Definition 2: *For a bridge B_i and a path P_j, let l_i^j be the leftmost attachment vertex of B_i on P_j; and r_i^j be the rightmost attachment vertex of B_i on P_j. If B_i has no attachment vertices on P_j, l_i^j and r_i^j are undefined.*

Step 3. Construct a new directed graph $G_B(V_B, E_B)$, called the *bridge graph*, defined as follows.

$$V_B = \{\beta_i \mid B_i \text{ is a bridge}\} \cup \{s, t\}$$

The edges in E_B are the union of $k+1$ sets.

1. The set $S = \{e_1, \ldots, e_{k-1}\}$ of edges outgoing from s:

   $$e_j = \{s \rightarrow \beta_x \mid r_x^j = \max_z r_z^j \text{ s.t. } (l_z^j = s) \wedge r_x^j > s\}$$

   In words, the edge e_j is $(s \rightarrow \beta_x)$ if r_x^j is the rightmost vertex on P_j, to the right of s, among the attachment vertices of bridges whose leftmost attachment is s. Notice that if $(s \rightarrow \beta_x)$ is an edge then $l_x^j = s$, for *all* $1 \leq j \leq k-1$.

2. The set T of edges incoming to t:

   $$T = \{\beta_x \rightarrow t \mid r_x^j = t, \text{ for some } 1 \leq j \leq k-1\}$$

In words, the set T consists of edges $(\beta_j \rightarrow t)$, for all bridges B_j whose rightmost attachment is t. Notice that if $(\beta_x \rightarrow t) \in T$ then $r_x^j = t$, for *all* $1 \leq j \leq k-1$.

3. The edges between vertices corresponding to bridges. These edges are partitioned into sets $D_1, \ldots, D_{k-1}$. We add an edge $(\beta_i \rightarrow \beta_x)$ to D_j, if it is possible to "move" from bridge B_i to B_x by "reversing" along some path P_y, and r_x^j is the furthest we can move (from B_i) along path P_j (see Fig. 6). More formally, the edge $(\beta_i \rightarrow \beta_x) \in D_j$ if

   (a) For some path P_y $(1 \leq y \leq k-1)$, we have $(s < l_x^y < r_i^y)$.

   (b) We have $(r_x^j > r_i^j)$. (This condition is considered satisfied in case B_i has no attachment vertex on P_j.)

   (c) There is no β_z, such that β_z satisfies the above two conditions, and $r_z^j > r_x^j$.

   In words, to determine the outgoing edge from β_i in the set D_j, consider all bridges whose leftmost attachment vertex on some P_y is to the left of r_i^y (and is not s). The edge $(\beta_i \rightarrow \beta_x)$ is added to D_j if B_x has the rightmost attachment on P_j, among all bridges in the set, and this attachment vertex r_x^j, is to the right of r_i^j. Ties are broken lexicographically, i.e., if β_x and β_y are both candidates for the outgoing edge from β_i, we choose β_x if $x < y$. (Each bridge can be uniquely labeled by the index of the lowest numbered edge it contains.)

The graph G_B is illustrated by an example in Fig. 7.

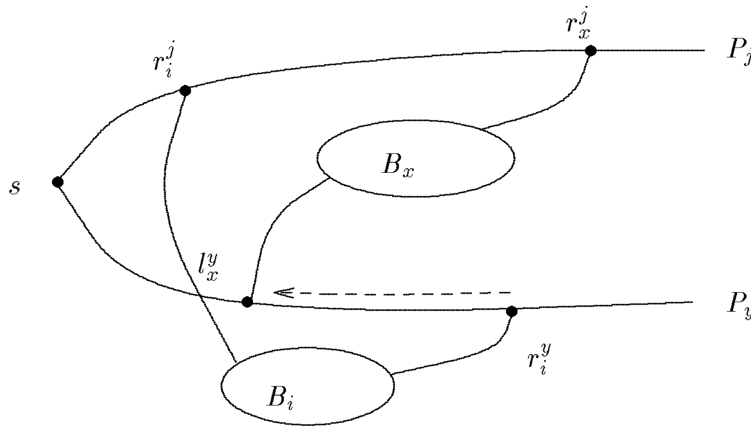


Figure 6: The edge $(\beta_i \rightarrow \beta_x)$ is added to D_j

Notice that each vertex, except s and t, has at most one outgoing edge belonging to each set D_j. Since there are $k-1$ such sets, the outdegree of each such vertex is at most $k-1$. It is easy to see also that the outdegree of s is at most $k-1$ and the outdegree of t is zero.

Our algorithm is based on two ideas. First, we show that there exists a directed path from s to t in G_B if and only if there are k vertex disjoint paths between s and t in G. Moreover, given the path in G_B, the k disjoint s-t paths in G can be constructed. Second, by exploiting the structure of G_B, we show how to find a path from s to t (if one exists) efficiently.

Step 4. Find a (directed) path from s to t in G_B, if one exists.

Step 5. If a path from s to t in G_B was found, construct the k vertex disjoint paths between s and t. Suppose that there is no path from s to t in G_B. Consider all the vertices β_i reachable from s in G_B. Recall that each such vertex corresponds to a bridge B_i of G. Let w_j be the attachment vertex of one of these bridges which is furthest from s on P_j. In case there is no attachment vertex of any of these bridges on P_j, then w_j is chosen to be the the neighbor of s on P_j. The set $\{w_1, \ldots, w_{k-1}\}$ separates s from t.

Correctness. Suppose that there are $k-1$ vertex disjoint paths between s and t. By our induction hypothesis these paths are found in Step 1. The base case is when $k = 1$. In this case the path P_1 from s to t is found by computing a spanning tree of G, and following the unique path from s to t in this tree. To prove the inductive step we prove the following theorem.

Theorem 4.1: *There are k vertex disjoint paths between s and t in G if and only if there is a directed path from s to t in G_B.*

Proof: The *if* direction: Suppose that there is a directed path $P_B = s, \beta_{x_1}, \ldots, \beta_{x_a}, t$ from s to t in G_B. Assume that P_B is (vertex) simple (i.e., no vertex appears twice in P_B).

To prove that there are k vertex disjoint paths from s to t in G, we show that the value of the max-flow in the network $N_{s,t}$ is at least k. Observe that the paths $P_1, \ldots, P_{k-1}$ correspond to $k-1$ vertex disjoint paths in $N_{s,t}$ from s to t. Using these paths we can push $k-1$ flow units from s to t. Our goal is to show that P_B defines an augmenting path in $N_{s,t}$, given this flow.

For an edge $(\beta_i \to \beta_x)$ in G_B, define the *type* of $(\beta_i \to \beta_x)$ to be y, if $r_i^y > l_x^y$. For example, in Fig. 6, the type of the outgoing edge from β_i is y (since the "reversal" is done on path P_y). Notice that by the definition of G_B, the type is always defined. In case there are several such y's any one may be chosen to be the type of the edge. Given the path $P_B = s, \beta_{x_1}, \ldots, \beta_{x_a}, t$, let i_b be the type of the outgoing edge from β_{x_b}, for $1 \le b \le a-1$.

For each vertex β_{x_b} on P_B, define a path R_{x_b} in B_{x_b} as follows. For B_{x_1}, R_{x_1} is from s to $r_{x_1}^{i_1}$, where i_1 is the type of the outgoing edge from β_{x_1}. For B_{x_a}, R_{x_a} is from $l_{x_a}^{i_{a-1}}$ to t, where i_{a-1} is the type of the outgoing edge from $\beta_{x_{a-1}}$. For B_{x_b}, $1 < b < a$, R_{x_b} is from $l_{x_b}^{i_{b-1}}$ to $r_{x_b}^{i_b}$, where i_{b-1} is the type of the outgoing edge from $\beta_{x_{b-1}}$ and i_b is the type of the outgoing edge from

β_{x_b} in P_B. See Fig. 7 for an example. In this example $a = 5$, and the paths $P_{i_1}, P_{i_2}, P_{i_3}, P_{i_4}$ are P_1, P_3, P_1, P_2 respectively. The values of x_1, x_2, x_3, x_4, x_5 are $1, 2, 3, 5, 6$ respectively.

Define a path P_k from s to t in G

$$P_k = R_{x_1}; P_{i_1}[r^{i_1}_{x_1}; l^{i_1}_{x_2}]; R_{x_2}; \ldots; R_{x_{a-1}}; P_{i_{a-1}}[r^{i_{a-1}}_{x_{a-1}}; l^{i_{a-1}}_{x_a}]; R_{x_a}.$$

Path P_k consists of the following two kinds of segments: segments belonging to bridges, and segments belonging to paths P_i that "reverse" on P_i. (We view each P_i as directed from s to t.)

The path P_k is not necessarily (edge) simple as shown in Fig. 7. (The figure illustrates only the leftmost and rightmost attachment vertices of each bridge.) This happens when some segments of P_k that "reverse" on some P_i, overlap. We claim that, given P_k we can construct an (edge) simple path from s to t that consists of two kinds of segments: segments belonging to bridges, and segments that "reverse" on some P_i. This is achieved by a "pruning" step on the path P_k. Moreover, we can obtain a "pruned" path P_k with the following *monotonicity property.* If $b < c$, and both $P_i[r^i_{x_b}; l^i_{x_{b+1}}]$ $P_i[r^i_{x_c}; l^i_{x_{c+1}}]$ are segments of the edge simple path P_k, then $l^i_{x_{b+1}} < r^i_{x_b} \leq l^i_{x_{c+1}} < r^i_{x_c}$. (In other words, as we move on the "pruned" P_k from s to t the segments of P_k that are "reverse" segments are ordered on P_i.)

The "pruning" step on P_k is done as follows. Consider the segment $P_i[r^i_{x_b}; l^i_{x_{b+1}}]$. Let d be the maximum index such that $d > b$, and $l^i_{x_d} < r^i_{x_b}$ on P_i. (Note that d is always defined, since $d = b+1$ is a possible candidate.) In this case we "prune" the path P_k, and replace the subpath of P_k from $r^i_{x_b}$ to $l^i_{x_d}$ by the segment $P_i[r^i_{x_b}; l^i_{x_d}]$ (See Fig. 8). The simple path is constructed by following the chain from s to t in the resulting graph. Let

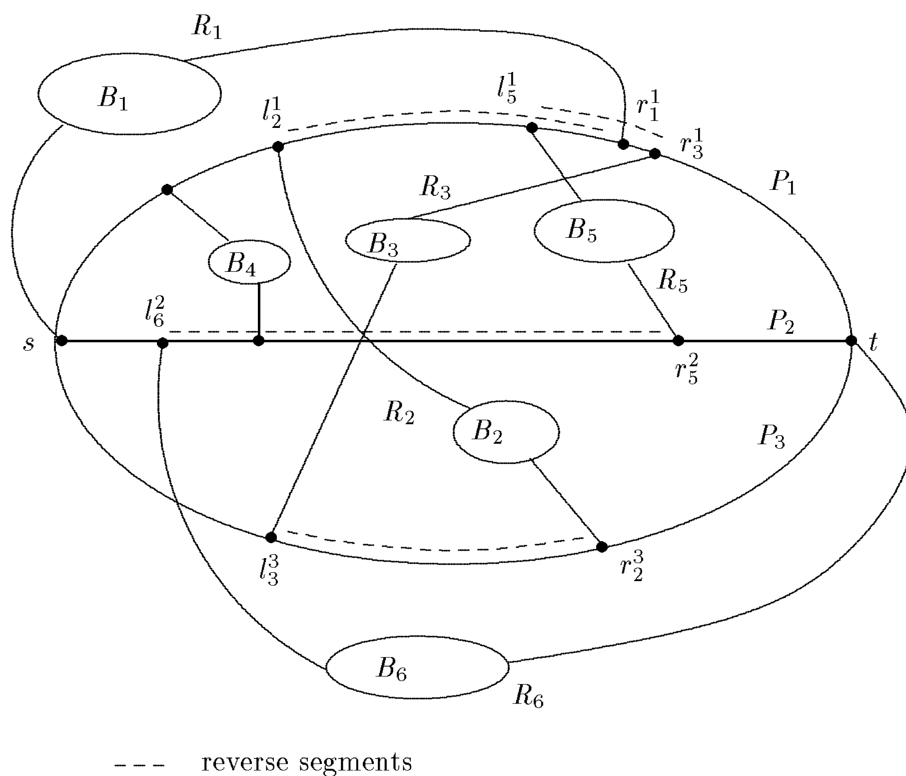
$$P_k = R_{x_1}; P_{i_1}[r^{i_1}_{x_1}; l^{i_1}_{x_2}]; R_{x_2}; \ldots; R_{x_{a-1}}; P_{i_{a-1}}[r^{i_{a-1}}_{x_{a-1}}; l^{i_{a-1}}_{x_a}]; R_{x_a}$$

denote the resulting (edge) simple path.

We claim that P_k corresponds to an augmenting path in $N_{s,t}$. This follows from the following lemma.

Lemma 4.2: *After pushing one unit of flow along P_k, the outgoing flow from each vertex (excluding s) and the incoming flow to each vertex (excluding t) is at most one.*

Proof: Before pushing the flow along P_k we had a legal flow of value $k-1$. Clearly, by pushing flow along P_k we change the flow only for the vertices on P_k, thus we may consider only these vertices. Consider an internal vertex v on P_k. If v is not on any P_i, $i < k$ (that is, it is an internal vertex on some R_{x_i}), then its incoming and outgoing flow is one.



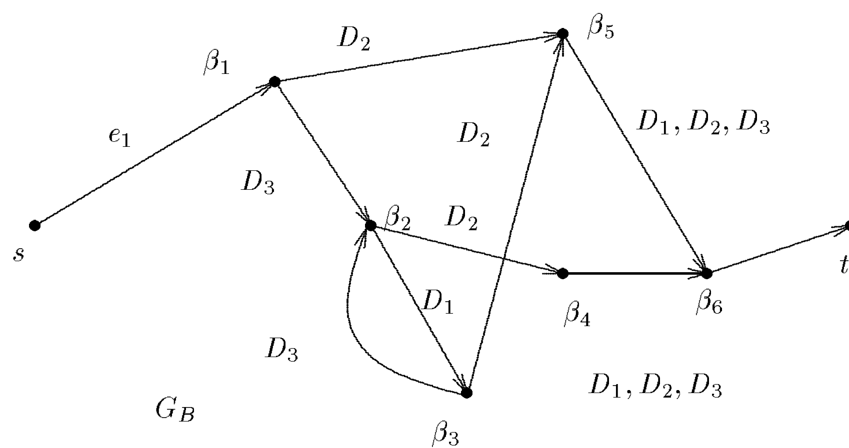


Figure 7: The path P_k corresponding to $P_B = s, \beta_1, \beta_2, \beta_3, \beta_5, \beta_6, t$ (in G_B).

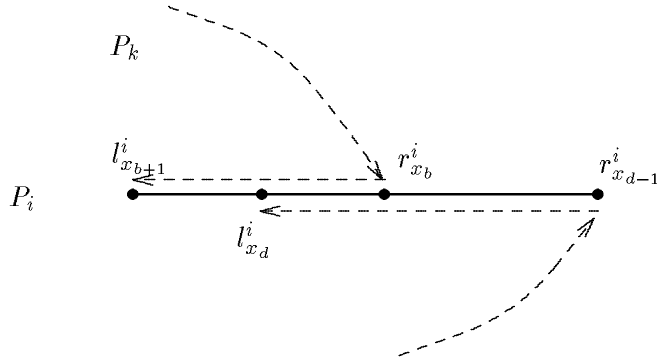


Figure 8: We prune the path so that P_k goes from B_{x_b} to B_{x_d}

Suppose that v is an internal vertex on some P_i. Let $(u \to v)$, and $(v \to w)$ be the incoming and outgoing edges of v on P_i respectively. We have three cases. (The figures describing these cases are given in Fig. 9.)

Case 1. The vertex v is not an endpoint of any path R_{x_b}.

In this case v appears only once on P_k. Observe that on P_k we have the edges $(w \to v)$ and $(v \to u)$ (that is, the reverse of the corresponding edges on P_i). After pushing one unit of flow along P_k, the incoming and outgoing flow of v is zero.

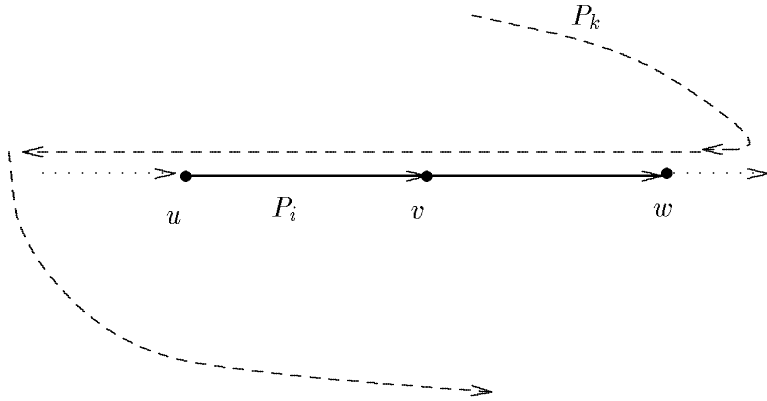
Case 2. The vertex v is a right endpoint of R_{x_b} and a left endpoint of $R_{x_{c+1}}$ (i.e., $v = r^i_{x_b} = l^i_{x_{c+1}}$), for some $b < c$.

Both $P_i[r^i_{x_b}; l^i_{x_{b+1}}]$ and $P_i[r^i_{x_c}; l^i_{x_{c+1}}]$ are segments of the edge simple path P_k. Sinc $r^i_{x_b} = l^i_{x_{c+1}}$, by the monotonicity property there is no d, $(b < d < c)$ such that R_{x_d} has an endpoint on P_i. It follows that the vertex v appears exactly twice on P_k. In the first appearance, the incoming edge of P_k is the last edge on R_{x_b}, and the outgoing edge is $(v \to u)$ (reverse of the edge on P_i). In the second appearance on P_k, the incoming edge is $(w \to v)$ and the outgoing edge is the first edge on $R_{x_{c+1}}$. After pushing one unit of flow along P_k, the incoming flow to v is one (from R_{x_b}), and the outgoing flow from v is one (towards $R_{x_{c+1}}$).

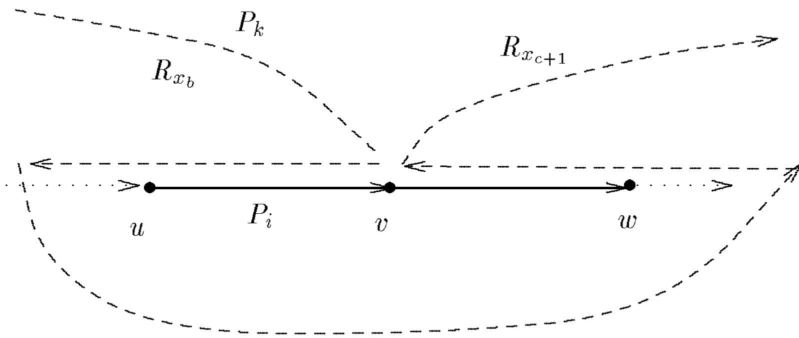
Case 3. The vertex v is an endpoint of one path R_{x_b} (i.e., either $v = l^i_{x_b}$ or $v = r^i_{x_b}$).

We prove it only for the case $v = l^i_{x_b}$, the proof for the case $v = r^i_{x_b}$ is analogous. The vertex v appears only once on P_k. Observe that on P_k we have the edges $(w \to v)$ and the outgoing edge from v on R_{x_b}. After pushing a unit flow along P_k the incoming flow to v is one (from u) and the outgoing flow from v is one (towards R_{x_b}).

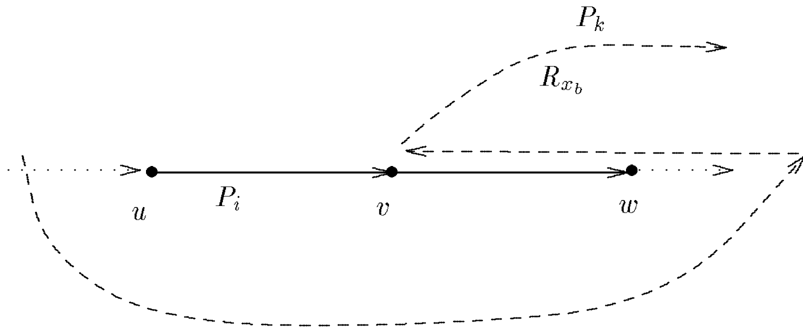
□



(a) Case 1.



(b) Case 2.



(c) Case 3.

Figure 9: Figures to illustrate all three cases

This concludes the proof of the *if* direction of Theorem 4.1.

The *only if* direction: Assume that there is no path from s to t in G_B. Consider all the vertices β_i reachable from s in G_B. Recall that each such vertex corresponds to a bridge B_i of G. Let w_j be the attachment vertex of one of these bridges which is furthest from s on P_j. In case there is no attachment vertex of any of these bridges on P_j, then w_j is chosen to be the the neighbor of s on P_j. We claim that $\{w_1, \ldots, w_{k-1}\}$ separate s from t. To see that, remove the vertices $\{w_1, \ldots, w_{k-1}\}$ and all their incident edges from G, and assume that there exists a path Q from s to t in the resulting subgraph.

Consider the vertices of Q that are also on one of the paths P_j, $1 \leq j \leq k-1$, in the order of their appearance on Q. Observe that there must be two successive vertices x_1 and x_2 such that: (i) x_1 is on some P_j and $x_1 < w_j$, (ii) x_2 is on some $P_{j'}$ and $x_2 > w_{j'}$. Clearly, the sub-path of Q from x_1 to x_2 does not contain any vertex of $P_1, \ldots, P_{k-1}$. Hence, there exists a bridge B_i such that $l_i^j < w_j$ and $r_i^{j'} > w_{j'}$. This implies that β_i is reachable from s in G_B, contradicting the definition of $w_{j'}$. □

Implementation and Complexity: We show how to implement the algorithm in $O(\log n)$ time using $C(n,m)$ processors on a CRCW PRAM, for any fixed k. In the following discussion we will assume that G is represented by its adjacency list.

Step 1. This is the recursive step.

Step 2. The bridges of G relative to the subgraph $P_1 \cup P_2 \cup \ldots \cup P_{k-1}$, and the attachment vertices of each bridge are obtained by computing the spanning forest of the subgraph of G induced by the vertices in $V(G) - V(P_1 \cup P_2 \cup \ldots \cup P_{k-1})$. By adding the edges incident to the vertices in $V(P_1 \cup P_2 \cup \ldots \cup P_{k-1})$ (that are not in $P_1 \cup P_2 \cup \ldots \cup P_{k-1}$) we can obtain all the trivial bridges, as well as the edges that go from vertices on the P_i paths to internal vertices of bridges. This is done in $O(\log n)$ time and $C(n,m)$ processors using a logarithmic time parallel graph connectivity algorithm (e.g., [CV86]).

Step 3. We show how to compute the edges of G_B. Recall that $(\beta_x \rightarrow \beta_y) \in D_i$, if r_y^i is the rightmost vertex on P_i, to the right of r_x^i, among the rightmost attachments of bridges whose leftmost attachment vertex on some P_z is to the left of r_x^z (and is not s).

We do the computation in $k-1$ phases. Phase i consists of three substeps, as follows.

1. Each vertex w on $P_1 \cup P_2 \cup \ldots \cup P_{k-1}$, considers all the bridges that are attached to it. Among them it selects the bridge whose rightmost attachment vertex on P_i is furthest from s. Denote this bridge by $M_i(w)$. This can be done, for all the vertices on $P_1 \cup P_2 \cup \ldots \cup P_{k-1}$, in $O(\log n)$ time and $(m+n)/\log n$ processors using the optimal logarithmic

time algorithm for finding the maximum [SV81].

2. Associate with each vertex w, the rightmost attachment vertex of $M_i(w)$ on P_i. (For convenience we refer to this attachment vertex as $M_i(w)$ as well.) For each vertex v on a path P_j, compute the prefix maximum $M_i^*(v) = \max_{w<v} M_i(w)$ (i.e., the maximum of $M_i(w)$ over all vertices w to the left of v on P_j). This can be done, for all the vertices in $O(\log n)$ time with $m/\log n$ processors, using the optimal logarithmic time algorithm for computing prefix maxima. This algorithm can be deduced from the general parallel algorithm for prefix computations [LF80].

3. Given the prefix maxima, the outgoing edge from the set D_i of a bridge B_x can be computed by taking the maximum over the set $\{M_i^*(r_x^1), \ldots, M_i^*(r_x^{k-1})\}$. This can be done, for all the bridges in $O(\log n)$ time with $mk/\log n$ processors, using the optimal logarithmic time algorithm for finding the maximum [SV81].

Step 4. We have to find a path from s to t in the directed graph G_B. In general, it is not known if s-t paths in directed graphs can be obtained in $O(\log n)$ time and $C(n,m)$ processors. We show how to find such a path in $O(k \log n)$ time using $k(m+n)/\log n$ processors, by exploiting the structure of G_B.

For $1 \leq j \leq k-1$, let F_j be the subgraph of G_B induced by the edges from the set $D_j \cup \{e_j\}$. (Recall that e_j is an outgoing edge from s.) Observe that the outdegree of each vertex in F_j is at most one. Observe also that F_j is acyclic, since $(\beta_x \rightarrow \beta_y)$ in F_j implies that $r_x^j < r_y^j$ (and also, there is no edge incoming to s). We conclude that F_j is a rooted forest (where the edges are directed towards the root).

Define a "shortcutting" operation *over* the edges from $D_j \cup \{e_j\}$. In the "shortcutting" the outgoing edges of all vertices with outgoing edges in $D_j \cup \{e_j\}$ are updated. Consider such a vertex β_y. (We assume that this vertex is not s. Later we describe how the updating is done for s.)

First, the outgoing edge from β_y from the set D_j is deleted. Let β_z be the root of the tree in F_j containing β_y. Notice that β_z has no outgoing edge from D_j. If β_z has no outgoing edges in G_B, then this completes the "shortcutting". Suppose that β_z has outgoing edges. If β_z has an outgoing edge to t, then we add an edge from β_y to t (in case such an edge does not exist). Suppose that β_z has no outgoing edge to t. For $a = 1, \ldots, k-1$, $a \neq j$, define β_{z_a} to be the vertex whose attachment point on P_a is the rightmost among the attachment points of all vertices with incoming edges from β_z. That is, $r_{z_a}^a \geq r_{z'}^a$, for all $\beta_{z'}$ such that $(\beta_z \rightarrow \beta_{z'}) \in E_B$. Observe that β_{z_a} is defined if β_z has outgoing edges, and that usually $(\beta_z \rightarrow \beta_{z_a})$ is the outgoing edge from the set D_a. This is not the case only when $r_{z_a}^a \leq r_z^a$. If β_y has an outgoing edge to

t then no update is done. Otherwise, for $a = 1, \ldots, k-1$, $a \neq j$, the outgoing edge from β_y, from the set D_a is updated as follows:

Case 1. The vertex β_y has no outgoing edge from the set D_a. If $r^a_{z_a} > r^a_y$ then the outgoing edge from β_y from D_a is taken to be $(\beta_y \rightarrow \beta_{z_a})$.

Case 2. The vertex β_y has an outgoing edge $(\beta_y \rightarrow \beta_{y_a})$ from the set D_a. If $r^a_{z_a} > r^a_{y_a}$ then the outgoing edge from β_y from D_a is updated to be $(\beta_y \rightarrow \beta_{z_a})$.

The "shortcutting" operation from s is defined similarly, where the sets $\{e_a\}$ play the role of the sets D_a.

The example in Fig. 10 shows a shortcutting step in F_1 (the subgraph of G_B induced by $D_1 \cup \{e_1\}$).

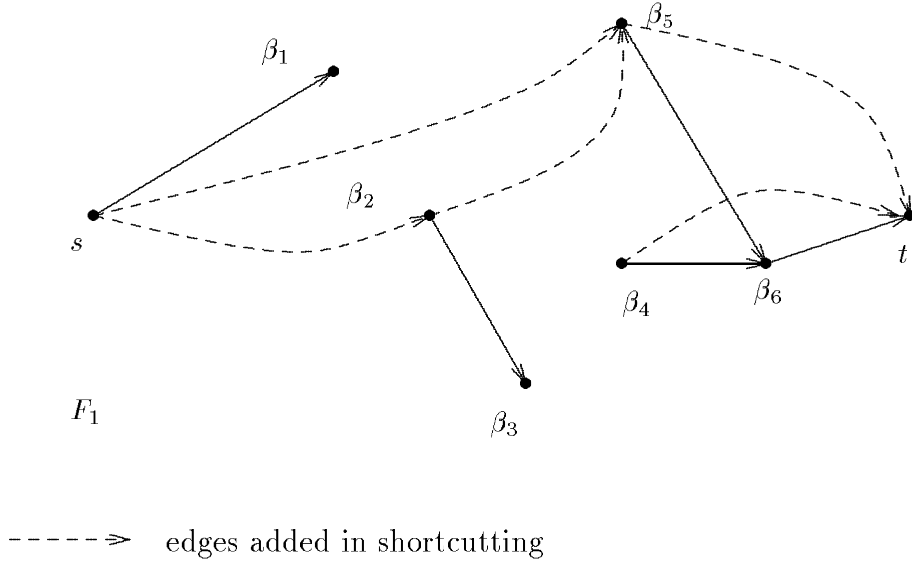


Figure 10: Shortcutting in F_1

Lemma 4.3: *Let G'_B be the graph resulting from the "shortcutting" over the edges of $D_j \cup \{e_j\}$. Then, there exists a path from s to t in G_B, if and only if there exists a path from s to t in G'_B.*

Proof: The *if* direction: Suppose that there is a path P'_B in G'_B, then it is easy to reconstruct a path P_B in G_B. If P'_B uses an edge $(\beta_y \rightarrow \beta_{z'})$ in G'_B (that is not an edge in G_B), then we replace it by the path from β_y to β_z and the edge $(\beta_z \rightarrow \beta_{z'})$.

The *only if* direction: For this direction we first prove the following claim.

Claim: If there exists a path from s to t in G_B then there exists a path $P_B = s, \beta_{x_1}, \ldots, \beta_{x_a}, t$, with the following property: The edge incoming to β_{x_1} is e_{i_1}, and for each $1 < c < a$, the edge incoming to β_{x_c} is from the set D_{i_c}, where i_c is the *type* of the outgoing edge from β_{x_c}.

Proof: Suppose that there exists a path $s, \beta_{y_1}, \ldots, \beta_{y_b}, t$, from s to t in G_B that does not satisfy the above property. We show how to construct a path $P_B = s, \beta_{x_1}, \ldots, \beta_{x_a}, t$, with this property. This is done by replacing the edges of the original path, one by one. The edge $(s \rightarrow \beta_{y_1})$ is replaced by the edge $e_{i_1} = (s \rightarrow \beta_{x_1})$, where i_1 is the type of the outgoing edge from β_{y_1}. If β_{x_1} has an outgoing edge to t then this completes the construction. Otherwise, we add the outgoing edge from β_{x_1} from the set D_{i_2} (if such exists). Notice that β_{x_1} can reach β_{y_2} by "reversing" on P_{i_1}. Thus, if β_{x_1} has no outgoing edge from D_{i_2}, then it must be the case that $r^{i_2}_{x_1} > r^{i_2}_{y_2} > l^{i_2}_{y_3}$. In this case β_{x_2} is taken to be β_{x_1}. (That is, for the proof, we allow the repetition of vertices in P'_B, the actual path is given by omitting these repetitions.) We continue in the same manner. Observe that we are guaranteed to get a path from s to t since we consistently move to a vertex whose attachment point on the path on which the next "reversal" is done is further to the right. □

We now return to the proof of Lemma 4.3. Suppose that there is a path $P_B = s, \beta_{x_1}, \ldots, \beta_{x_a}, t$, in G_B with the above property. If P_B consists only of edges from G'_B then the same path is also in G'_B. Suppose that P_B contains some edges that are not in G'_B. Let $(\beta_{x_b} \rightarrow \beta_{x_{b+1}})$ be the first such edge. We show how to construct a path $P'_B = s, \beta_{x_1}, \ldots, \beta_{x_b}, \beta_{x'_{b+1}}, \ldots, t$ in G'_B. The prefix of the path upto β_{x_b} is the same as the prefix of P_B upto β_{x_b}. If β_{x_b} has an outgoing edge to t then we add it to P'_B to complete the path. Otherwise, we distinguish between two cases:

Case 1. The edge $(\beta_{x_b} \rightarrow \beta_{x_{b+1}})$ is not from the set D_j. The outgoing edge from β_{x_b} in P'_B is taken to be its outgoing edge $(\beta_{x_b} \rightarrow \beta_z)$ from the set $D_{i_{b+1}}$, where i_{b+1} is the *type* of the outgoing edge from $\beta_{x_{b+1}}$ in P_B. From the definition of the "shortcutting" operation and the property of P_B it follows that such an outgoing edge exists and that $r^{i_{b+1}}_z > l^{i_{b+1}}_{x_{b+2}}$.

Case 2. The edge $(\beta_{x_b} \rightarrow \beta_{x_{b+1}})$ is from the set D_j. Let $(\beta_{x_c} \rightarrow \beta_{x_{c+1}})$ be the first edge in P_B following $(\beta_{x_b} \rightarrow \beta_{x_{b+1}})$ that is *not* from the set D_j. Notice that the *type* of this edge is j. In this case the vertices $\beta_{x'_{b+1}}, \ldots, \beta_{x'_c}$ are defined to be β_{x_b}. (That is, for the proof, we allow the repetition of vertices in P'_B, the actual path is given by omitting these repetitions.) There are two subcases for the outgoing edge from β_{x_c}.

Case 2.1. The vertex β_{x_c} is the root of the tree (in F_j) containing β_{x_b}. In this case the outgoing edge from $\beta_{x'_c} = \beta_{x_b}$ is taken to be its outgoing edge $(\beta_{x_b} \rightarrow \beta_z)$ from the set $D_{i_{c+1}}$ if such exists, where i_{c+1} is the *type* of the outgoing edge from $\beta_{x_{c+1}}$ in

P_B. From the definition of the "shortcutting" operation it follows that if such an outgoing edge exists then $r_z^{i_c+1} > l_{x_{c+2}}^{i_c+1}$. Otherwise, i.e., if there is no such edge, it must be the case that $r_{x_b}^{i_c+1} > l_{x_{c+2}}^{i_c+1}$. In this case, $\beta_{x'_{c+1}}$ is also defined to be β_{x_b}.

Case 2.2. The vertex β_{x_c} is not the root of the tree (in F_j) containing β_{x_b}. In this case, the next vertex is determined as follows. Let β_y be the root of the tree in F_j containing β_{x_b}. We have three possibilities:

1. If $r_{x_b}^{i_c+1} > l_{x_{c+2}}^{i_c+1}$ then the next vertex is also taken to be β_{x_b}.
2. If not (1) and $r_y^{i_c+1} > l_{x_{c+2}}^{i_c+1}$ then let $d > c$ be the minimum index such that $r_y^{i_d+1} < l_{x_{d+2}}^{i_d+1}$. (Notice that d is always defined. The only case it may be undefined is when β_y has an outgoing edge to t in G_B. However, in this case β_{x_b} would have also an outgoing edge to t in G'_B; a contradiction.) The vertices $\beta_{x'_{c+1}}, \ldots, \beta_{x'_d}$ are defined to be β_{x_b}. The outgoing edge from $\beta_{x'_d} = \beta_{x_b}$ is taken to be its outgoing edge $(\beta_{x_b} \rightarrow \beta_z)$ from the set $D_{i_{d+1}}$ if such exists. From the definition of the "shortcutting" operation it follows that if such an outgoing edge exists then $r_z^{i_d+1} > l_{x_{d+2}}^{i_d+1}$. Otherwise, i.e., if there is no such edge, it must be the case that $r_{x_b}^{i_d+1} > l_{x_{d+2}}^{i_d+1}$. In this case, $\beta_{x'_{d+1}}$ is also defined to be β_{x_b}.
3. Consider the remaining possibility. Notice that β_y can reach $\beta_{i_{c+1}}$ by "reversal" on P_j. Since $r_y^{i_c+1} \leq l_{x_{c+2}}^{i_c+1}$ and $r_{x_{c+1}}^{i_c+1} > l_{x_{c+2}}^{i_c+1}$, β_y must have an outgoing edge $(\beta_y \rightarrow \beta_z)$ from the set $D_{i_{c+1}}$, and clearly, $r_z^{i_c+1} > l_{x_{c+2}}^{i_c+1}$. After the "shortcutting" operation β_{x_b} must also have an outgoing edge $(\beta_y \rightarrow \beta_z)$ from the set $D_{i_{c+1}}$, where $r_z^{i_c+1} > l_{x_{c+2}}^{i_c+1}$. This outgoing edge is taken to be the next edge in P'_B.

On obtaining the outgoing edge from β_{x_b}, we continue the above process of modifying path P_B to obtain an s-t path in G'_B. Observe that, we may actually replace a vertex β_{x_c} in P_B by a vertex $\beta_{x'_c}$ in P'_B, for some $c > b$. However, we always have the property that $r_{x'_c}^{i_c} \geq r_{x_c}^{i_c}$, for all $c > b$. In words, we consistently move to a vertex whose attachment point on the path on which the next "reversal" is done is further to the right. We continue modifying P_B in this manner until we reach t. □

The path from s to t in G_B is computed in $k-1$ phases. In phase j we perform the "shortcutting" operation over the edges of $D_j \cup \{e_j\}$. From the above lemma it follows that there exists a path from s to t in G_B, if and only if s is connected to t by an edge in the graph resulting after these $k-1$ phases. If such a path exists, it can be reconstructed by adding the edges deleted in the "shortcutting", starting from phase $k-1$ down to phase 1.

Next, we describe how to implement each phase in $O(\log n)$ time using $km/\log n$ processors. The implementation consists of two stages: (1) For each vertex β_y identify the root of the tree of F_j containing it. (2) Given the root, update the outgoing edges of β_y.

It is not difficult to see that Stage (2) can be implemented in constant time using km processors and hence in $O(\log n)$ time using $km/\log n$ processors. The computation of Stage (1) can be done by applying the Euler tour technique of [TV85]. However, to apply this technique we need the full adjacency list of the forest F_j, i.e., we need to compute the list of edges from the set D_j incoming to each vertex β_x. For this we use the following observation. Consider two edges of type i: $(\beta_y \to \beta_x)$ and $(\beta_z \to \beta_x)$ from the set D_j. (Since both edges are of type i we can reach β_x from both β_y and β_z, by a "reversal" over P_i.) Suppose that $r_y^i < r_z^i$. Then, all the bridges whose outgoing edge from D_j is of type i and whose rightmost attachment vertex on P_i is between r_y^i and r_z^i have an outgoing edge to β_x.

For a bridge B_x and $1 \leq i \leq k-1$, define the *rightmost edge* on P_i to be the edge of B_x, whose endpoint is the rightmost attachment of B_x on P_i. Consider the concatenation of the adjacency lists (in G) of all the vertices on P_i. Compact this list to include only rightmost edges of bridges whose outgoing edge from D_j is of type i. It follows from the above observation that the rightmost edges of all the bridges that (i) their outgoing edge from D_j is of type i, and (ii) these edges point to the same vertex in G_B, are consecutive in this concatenated list. Thus, the list of incoming edges of all the vertices in G_B can be computed using the algorithms for list ranking and prefix sums in $O(\log n)$ time and $(m+n)/\log n$ processors [CV86, AM88, LF80].

Step 5. If there is no path from s to t in G_B, the separating set $\{w_1, \ldots, w_{k-1}\}$ can be found from the construction of Step 4 in $O(k \log n)$ time and $m/\log n$ processors. After doing the "shortcutting" assume that there is no edge from s to t in G'_B. We add the deleted vertices (that are "shortcutted" over) to G'_B, in the reverse order of deletion. At each stage of adding the vertices, we consider the set of edges which were deleted when the vertices were "shortcutted" over. For each deleted vertex, we test whether it was reachable from s when the deleted edges are added. In this way we are able to obtain the set of vertices reachable from s, and taking their rightmost attachment vertices on each of the paths yields the separating set.

If a path P_B was found, then following the proof of Theorem 4.1 we construct P_k. Recall, that to make the path P_k (edge) simple we have to perform a "pruning" step. This is implemented by constructing a linked list. For each rightmost attachment vertex $r_{x_b}^i$ on P_i, define its successor $succ(r_{x_b}^i)$ to be $l_{x_d}^i$, if (i) $d = \max\{z \mid l_{x_z}^i < r_{x_b}^i\}$, and (ii) there is no $c < b$ such that $l_{x_d}^i < r_{x_c}^i$. We add condition (ii) to avoid the situation where one attachment vertex is the successor of more than one vertex. For each leftmost attachment vertex $l_{x_b}^{i_{b-1}}$, $succ(l_{x_b}^{i_{b-1}}) = r_{x_b}^{i_b}$. Since each attachment vertex has at most one successor and is a successor of at most one vertex, the resulting graph is a set of chains. A list ranking step from s, yields the "pruned" path. The "pruning" is implemented optimally in logarithmic time using the parallel prefix computations algorithm of [LF80] and the parallel list ranking algorithm of [CV86, AM88].

Finally, the k vertex disjoint paths can be constructed following the resulting k flow units

from s to t, in $O(\log n)$ time and $n/\log n$ processors using the optimal list ranking algorithm ([CV86, AM88]). We conclude

Theorem 4.4: *The described algorithm finds k vertex disjoint s-t paths in $O(k^2 \log n)$ time using $kC(n,m)$ processors.*

5. Finding edge disjoint paths

In this section we describe a parallel algorithm for finding k-edge disjoint s-t paths. The algorithm uses the same techniques as the algorithm for finding vertex disjoint paths and has the same complexity; that is, the algorithm either finds k-edge disjoint paths or a set of at most $k-1$ edges whose removal separates s and t, in $O(\log n)$ time and $C(n,m)$ processors.

The parallel algorithm is similar to the algorithm of Section 4. The only difference is in the definition of the bridge graph G_B. Recall that edge disjoint s-t paths can be computed using a max-flow computation in the network $M_{s,t}$ in which each edge has unit capacity, i.e., we relax the restriction of unit vertex capacities and impose only the edge capacity constraints. Because of this relaxation the flow augmenting path P_k in $M_{s,t}$ is permitted to transfer itself from one bridge to another by using zero or more reverse edges of P_i, unlike the condition imposed earlier that required that at least one edge of P_i be traversed in the reverse direction. To capture this, the definition of the sets D_j in the bridge graph G_B is modified. Specifically, the edge $(\beta_i \rightarrow \beta_x)$ is added to the set D_j if r_x^j is to the right of r_i^j, and is the rightmost vertex on P_j among the rightmost attachments of all bridges whose leftmost attachment on some path P_y, is to the left of or *the same as* r_i^y. More formally, the edge sets $D_1, \ldots, D_{k-1}$ are defined as follows. For $1 \leq j \leq k-1$, the edge $(\beta_i \rightarrow \beta_x) \in D_j$ if

1. For some path P_y $(1 \leq y \leq k-1)$, we have $(s < l_x^y \leq r_i^y)$.
2. We have $(r_x^j > r_i^j)$. (This condition is considered satisfied in case B_i has no attachment vertex on P_j.)
3. There is no β_z, such that β_z satisfies the above two conditions, and $r_z^j > r_x^j$.

Note that the strict inequality in (1) was replaced by an inequality.

The correctness of the algorithm is implied by the following theorem.

Theorem 5.1: *There are k edge disjoint paths between s and t in G if and only if there is a directed path from s to t in (the modified) G_B.*

The proof of the theorem is similar to the proof of Theorem 4.1: We show that a path in G_B corresponds to a flow augmenting path P_k in the network $M_{s,t}$. Using the flow of value k we can construct k-edge disjoint paths. If there is no path from s to t in G_B, then the edges on the paths P_i outgoing from the furthest attachment vertices of the bridges reachable from s in G_B, separate s from t.

6. Applications

In this section we describe some more applications of our parallel algorithm for finding disjoint s-t paths.

Constructing a cycle through three specified vertices

Problem: Given an undirected graph G, and three specified vertices a, b, c of G determine whether the three vertices lie on a common simple cycle and construct such a cycle if one exists.
Below, we show how to derive an efficient logarithmic time parallel algorithm for this problem.

The parallel algorithm is a parallelization of the sequential algorithm of [LR80]. We assume that G is biconnected since a, b and c must be in the same biconnected component if the cycle exists. The main idea of the sequential algorithm of [LR80] is to decompose G into pieces and to look for paths which must exist in these pieces if the cycle is to exist in G. All the steps in this algorithm are easy to parallelize efficiently, except for the step that involves computing three vertex disjoint $s-t$ paths. Using the k-disjoint paths algorithm we can compute the three paths (if such exist) and either construct the desired cycle from these paths, or conclude that such a cycle does not exist.

The algorithm can be implemented in logarithmic time using the algorithm of Section 4, together with parallel logarithmic time connectivity (e.g., [CV86]), biconnectivity (e.g., [TV85]) and triconnectivity (e.g., [FRT89]) algorithms. The number of processors used in the algorithm is the same as the number of processors needed by the parallel triconnectivity algorithm. (The algorithm of [FRT89] appears to use $(n+m)\log\log n/\log n$ processors.)

The two paths problem

Problem: Given an undirected graph G, and two pairs of vertices a_1, a_2 and b_1, b_2 find two disjoint paths, one from a_1 to a_2 and one from b_1 to b_2.
Our algorithm for finding three disjoint s-t paths is used as a subroutine in the parallel algorithms of [KMV89] for the two paths problem to yield an $O(\log n)$ time, n^2 processors parallel algorithm for solving the two paths problem on general graphs, and a logarithmic time parallel

algorithm for solving the problem on planar graphs that uses the same number of processors as required for the triconnectivity algorithm.

Testing and finding subgraph homeomorphism for some fixed pattern graphs

Problem: Given a graph G, test if G has a subgraph homeomorphic to some fixed pattern graph H. If G has such subgraph find it.
We show how to solve this problem for the pattern graphs: K_4 and $K_{2,3}$. As a corollary this gives an efficient algorithm to test whether a graph is outer-planar.

First, we consider the testing problem. The following lemmas are from [Asa85].

Lemma 6.1: *For a triconnected graph H, a graph G has a subgraph homeomorphic to H if and only if there is a triconnected component of G that has a subgraph homeomorphic to H.*

Lemma 6.2: *If a simple graph G with two or more vertices has no subgraph homeomorphic to K_4, then $m \leq 2n - 3$.*

Lemma 6.3: *A graph G has a subgraph homeomorphic to K_4 if and only if there is a triconnected component of G with four or more vertices.*

Lemma 6.4: *If a simple graph G with two or more vertices has no subgraph homeomorphic to $K_{2,3}$, then $m \leq 2n - 2$.*

Lemma 6.5: *A simple graph G has a subgraph homeomorphic to $K_{2,3}$, if and only if there is a triconnected component of G satisfying one of the following:*

(i) It has five or more vertices.

(ii) It is the graph K_4 with at least one virtual edge.

(iii) It is a triple bond of three virtual edges.

Lemmas 6.2 and 6.3 imply a simple algorithm for testing whether a given graph G has a subgraph homeomorphic to K_4. Similarly, Lemmas 6.4 and 6.5 imply a simple algorithm for solving the same problem for $K_{2,3}$. Implementing both algorithms using a logarithmic time triconnectivity algorithm (e.g., [FRT89]) yields logarithmic time algorithms that use the same number of processors as required for the triconnectivity algorithm.

A planar graph is *outer-planar* if it can be embedded in the plane so that all its vertices lie on the same face. The following theorem is an easy corollary of Kuratowski's theorem.

Theorem 6.6: *A graph is outer-planar if and only if it has no subgraph homeomorphic to K_4 or $K_{2,3}$.*

We conclude that we can test whether a graph is outer-planar in $O(\log n)$ time using the same number of processors as required for the triconnectivity algorithm. Our algorithm is simpler than the algorithm given in [RR89]. However, it does not yield the (outer-)planar embedding of the graph in case it is outer-planar. To obtain an outer-planar embedding we add one artificial vertex v to G, and an edge from v to every original vertex in the graph. The new graph is planar if and only if G is outer-planar, since the addition of v ensures that all the original vertices are on the same face in the obtained embedding. Using the logarithmic time algorithm in [RR89] we can obtain an outer-planar embedding for G.

Finally, we note that in a similar way we can test subgraph homeomorphism for the pattern graphs: C_4, C_5 and $K_{2,p}$, for any fixed p.

We use the k-vertex disjoint paths algorithm to obtain homeomorphs of K_4 and $K_{2,3}$.

Finding K_4 homeomorphs: Our algorithm is based on the proof of Lemma 6.3 in [Asa85]. W.l.o.g. assume G has $2n$ edges (if not, choose any subgraph of G with $2n$ edges), and that G has a triconnected component G_1 with four or more vertices. Choose any vertex v of G_1 and consider the subgraph $G_1 - \{v\}$ (which is biconnected). In $G_1 - \{v\}$, find a cycle C of length ≥ 3, by obtaining a spanning tree of $G_1 - \{v\}$ and adding one non-tree edge. Find three vertex disjoint paths from v to three distinct vertices on the cycle C, such paths exist since G is triconnected. The three paths are found as follows. First, introduce a new vertex t, and add edges from t to all vertices on C. Then, find three vertex disjoint v-t paths. The required paths are the segments of these v-t paths upto their first point of intersection with C. It is easy to see that the graph obtained from these three disjoint paths from v to C, together with C is homeomorphic to K_4. To get the K_4 homeomorph in G, we may need to replace virtual edges by paths in G. It follows that using our algorithm, together with parallel logarithmic time connectivity and triconnectivity algorithms, a K_4 homeomorph can be found in $O(\log n)$ time using the same number of processors as required for the triconnectivity algorithm.

Finding $K_{2,3}$ homeomorphs: Our algorithm is based on the proof of Lemma 6.5 in [Asa85]. Again, we assume that G has $2n$ edges and a triconnected component G_1 satisfying one of the conditions in Lemma 6.5. We consider all the three cases which G_1 may satisfy.

Case 1. G_1 has five or more vertices: Find G', a subgraph homeomorphic to K_4. Let the vertices in G' of degree three be called v_1, v_2, v_3, v_4. If G' is exactly the graph K_4 then G_1 must have another vertex $u \notin G'$. Find three disjoint paths from u to any three vertices of the K_4. It is easy to see that a subgraph homeomorphic to $K_{2,3}$ can be extracted from

these three paths and the K_4.

If G' is a subdivision of K_4, then consider a vertex u on the path $P[v_1; v_2]$. Since G_1 is triconnected, there must be a path from u to some other vertex w of G' in $G_1 - \{v_1, v_2\}$. Using this path and the K_4 it becomes easy to extract the subgraph homeomorphic to $K_{2,3}$.

Case 2. G_1 is K_4 with a virtual edge: Replace the virtual edge by a path of length ≥ 2 and obtain a subgraph homeomorphic to $K_{2,3}$.

Case 3. G_1 is a triple bond of virtual edges: Replace all three virtual edges by paths of length ≥ 2 and obtain a subgraph homeomorphic to $K_{2,3}$.

Again, the implementation of this algorithm can be done in $O(\log n)$ time using the same number of processors as required for the triconnectivity algorithm.

Acknowledgements. We are grateful to Steve Mitchell and Vijay Vazirani for useful discussions and encouragement. We are also grateful to the referees for many useful suggestions.

References

[AM88] R.J. Anderson and G.L. Miller. Deterministic parallel list ranking. In *Proc. of AWOC 88, Lecture Notes in Computer Science* No. 319, pages 81–90. Springer-Verlag, 1988.

[Asa85] T. Asano. An approach to the subgraph homeomorphism problem. *Theoretical Computer Science*, 38:249–267, 1985.

[BDD+82] M. Becker, W. Degenhardt, J. Doenhardt, S. Hertel, G. Kaninke, W. Keber, K. Mehlhorn, S. Näher, H. Rohnert, and T. Winter. A probabilistic algorithm for vertex connectivity of graphs. *Information Processing Letters*, 15(3):135–136, October 1982.

[Ber76] C. Berge. *Graphs and Hypergraphs.* North-Holland, Amsterdam, 1976.

[BM77] J.A. Bondy and U.S.R. Murty. *Graph Theory with applications.* American Elsevier, New York, NY, 1977.

[CV86] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proc. 27th IEEE Symp. on Foundation of Computer Science*, pages 478–491, October 1986.

[CV88] R. Cole and U. Vishkin. Optimal parallel algorithms for expression tree evaluation and list ranking. In *Proc. of AWOC 88, Lecture Notes in Computer Science* No. 319, pages 91–100. Springer-Verlag, 1988.

[DF56] G.B. Dantzig and D.R. Fulkerson. On the max-flow min-cut theorem of networks. In *Linear Inequalities and Related Systems, Annals of math. Study* No. 38, pages 215–221. Princeton University Press, 1956.

[ET75] S. Even and R.E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4:507–518, 1975.

[ET76] S. Even and R.E. Tarjan. Computing *st*-numbering. *Theoretical Computer Science*, 2:339–344, 1976.

[Eve75] S. Even. An algorithm for determining whether the connectivity of a graph is at least k. *SIAM Journal on Computing*, 4:393–396, 1975.

[Eve79] S. Even. *Graph algorithms*. Computer Science Press, Rockville, MD, 1979.

[FRT89] D. Fussell, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacements. In *Proc. 16th ICALP, Lect. Notes in Comp. Sci.* No. 372, pages 379–393. Springer-Verlag, July 1989.

[FT88] D. Fussell and R. Thurimella. Separation pair detection. In *Proc. of AWOC 88, Lecture Notes in Computer Science* No. 319, pages 149–159. Springer-Verlag, 1988.

[Gal80] Z. Galil. Finding the vertex connectivity of graphs. *SIAM Journal on Computing*, 9:197–200, 1980.

[Har69] F. Harary. *Graph Theory*. Addison Wesley, Reading, Ma, 1969.

[IR84] A. Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. In *Proc. 25th IEEE Symp. on Foundation of Computer Science*, pages 137–147, October 1984.

[KMV89] S. Khuller, S.G. Mitchell, and V.V. Vazirani. Processor efficient parallel algorithms for the two disjoint paths problem and for finding a Kuratowski homeomorph. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 300–305, October 1989.

[KR87] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. In *Proc. 28th IEEE Symp. on Foundation of Computer Science*, pages 252–259, October 1987.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Proc. Int. Symp. on Theory of Graphs; P. Rosenstiehl Ed.*, pages 215–232. Gordon and Breach, 1967.

[LF80] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[LLW86] N. Linial, L. Lovász, and A. Wigderson. A physical interpretation of graph connectivity, and its algorithmic applications. In *Proc. 27th IEEE Symp. on Foundation of Computer Science*, pages 39–48, October 1986.

[LR80] A.S. LaPaugh and R.L. Rivest. The subgraph homeomorphism problem. *Journal of Computer and System Sciences*, 27:133–149, 1980.

[Mat87] D. Matula. Determining edge connectivity in $O(mn)$. In *Proc. 28th IEEE Symp. on Foundation of Computer Science*, pages 249–251, October 1987.

[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.

[RR89] V. Ramachandran and J.H. Reif. An optimal parallel algorithm for graph planarity. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 282–287, October 1989.

[Sey80] P.D. Seymour. Disjoint paths in graphs. *Discrete Mathematics*, 29:293–309, 1980.

[Shi80] Y. Shiloach. A polynomial solution to the undirected two path problem. *Journal of the ACM*, 27:445–456, 1980.

[SV81] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–63, 1982.

[Tar75] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal on the ACM*, 22:215–225, 1975.

[Thu89] R. Thurimella. *Techniques for the design of parallel graph algorithms.* PhD thesis, Dept. of Computer Science, The University of Texas at Austin, Austin, TX, 1989.

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

[Wyl79] J.C. Wyllie. *The complexity of parallel computations.* PhD thesis, Dept. of Computer Science, Cornell University, Ithaca, NY, 1979.