Efficient Parallel Algorithms for Disjoint Paths and Connectivity

Samir Khuller

Ph.D Thesis

TR 90-1142
July 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

EFFICIENT PARALLEL ALGORITHMS FOR DISJOINT PATHS AND CONNECTIVITY

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Samir Khuller
August 1990

EFFICIENT PARALLEL ALGORITHMS FOR DISJOINT PATHS AND CONNECTIVITY

Samir Khuller, Ph.D.
Cornell University 1990

This thesis is concerned with the problem of designing *efficient* parallel algorithms for various graph-theoretic problems. Our larger goal is to identify "tools" that would be useful in designing parallel algorithms for various graph-theoretic problems. We show that the concept of *bridges* plays a crucial role in the design of algorithms for the kinds of problems we consider. The specific problems we consider are the following.

We give an *NC* algorithm for finding vertex-disjoint s_1, t_1 and s_2, t_2 paths in an undirected graph. An important step in solving the general problem is solving the planar case. A new structural property yields the parallelization, as well as a simpler linear time sequential algorithm for this case. We extend the algorithm to the non-planar case by giving an *NC* algorithm for finding a Kuratowski homeomorph, and. in particular, a homeomorph of $K_{3,3}$, in a non-planar graph.

We also present an efficient parallel algorithm for testing whether a graph is k vertex-connected. To develop our algorithm we design an efficient parallel algorithm for the following *disjoint s-t paths* problem: Given a graph 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. 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 is k edge-connected. Finally, we describe more applications of the disjoint s-t paths algorithm. This algorithm is also used as a subroutine in the algorithm for the two paths problem mentioned earlier.

Biographical Sketch

Samir was born in Hubli, Karnataka, India on a warm spring day, March 29th, 1965. After spending his childhood in Madras and Lucknow, he went to High School at La Martiniere College (notwithstanding the name, it is still a High School !) in Lucknow. After completing his schooling, he proceeded to the Indian Institute of Technology at Kanpur, to study Computer Science in 1982.

Having successfully completed his requirements for a Bachelor's degree in Computer Science, he flew half way across to world to begin his studies for a PhD at Cornell. Samir arrived in Ithaca in the Fall of 1986, and, having gotten tired of writing programs at I.I.T, he started working on Graph Algorithms and Computational Geometry. After spending one summer playing tennis, and two summers at the IBM Thomas J. Watson Research Center, he finds himself ready to graduate from Cornell in the summer of 1990. In the Fall of 1990 he joins The University of Maryland's Institute for Advanced Computer Studies as a Research Associate.

To my parents, and my sister Alpana.

Acknowledgements

I would like to thank my parents and family, for always encouraging me to strive to achieve in my endeavours. If not for their encouragement and support, I would not be at a great place like Cornell.

I would like to give a special thanks to my advisor Vijay Vazirani, for first teaching me "how" to ask the right questions, and then teaching me "how" to attack them. He has always encouraged me to work on my ideas and has provided me with constant support and valuable time to discuss almost anything. I would also like to thank Joe Mitchell for valuable discussions, and encouragement at times when I needed it.

There are so many people I worked with at different points of time, all of whom provided me with new ideas, new perspectives and new results. All of them have contributed to my knowledge about Algorithms, Graph Theory and Computational Geometry. In particular, I would like to thank Estie Arkin, Steve Mitchell, Seffi Naor, and Baruch Schieber (toda raba !). I would also like to thank Prof. Mohanty for sparking off my interest in Graph Theory during the wonderful course he taught me many years ago.

I would like to thank IBM for supporting me with a fellowship, and Jean-Louis Lassez, Krishna Palem, and Ken Perry for inviting me to spend two summers at the Watson Research Labs, where I spent a good deal of time working on my own research (with Baruch Schieber).

I would like to thank David Gries and Dexter Kozen for serving on my committee.

My stay at Cornell was made more than memorable due to the many friends that I made here. For the great time I had in Ithaca, I would like to thank Ajei, Amy, Cathy, Chitra & Vasant, Mukta & Rakesh, Rohit, Sanjay, Sanjeev, Shyam, Sonali and all my other friends.

Table of Contents

List of Figures

Chapter 1

Introduction

This thesis is concerned with the design of *efficient* parallel algorithms for various graph-theoretic problems. The main problems that we study are:

- Given two pairs of vertices s_1, t_1 and s_2, t_2, finding two vertex-disjoint paths connecting s_i with t_i. (Joint work with Stephen Mitchell and Vijay Vazirani.)
- Finding k vertex-disjoint paths between two given vertices. (Joint work with Baruch Schieber.)
- Testing vertex and edge connectivities of graphs. (Joint work with Baruch Schieber.)
- Finding subgraphs homeomorphic to fixed pattern graphs.

Our larger goal is to identify "tools" that would be useful in designing parallel algorithms for various graph-theoretic problems. In the past, several tools, such as *depth first search* and *breadth first search*, have been developed for designing efficient sequential algorithms for graph-theoretic problems. Several of these techniques, including the two mentioned above, are not known to have efficient parallel algorithms. Therefore it is important to identify alternative techniques that would be of use in designing parallel algorithms. Our work identifies one such tool – the notion of *bridges*. Recently, this notion has been useful for designing parallel algorithms for other problems as well, e.g. planarity testing [RR89].

The study of parallel computation is becoming increasingly dominant in many areas of computer science. We expect that by the end of this century, fast parallel machines will be readily available, and that most algorithm design will be taking place in the context of parallel computation. So, it is important to understand parallel computation and to develop a framework for the designing parallel algorithms.

Designing a parallel algorithm that can run on some realistic machine model involves two steps. The first step is concerned with studying the inherent parallelism in the problem and developing an algorithm for it. The second step is to implement this algorithm efficiently on a specific parallel architecture. In this step, the actual architecture becomes rather important and one has to take into account the various aspects of the model and its special features. Our work is related to the first step of designing a parallel algorithm on the PRAM model. The PRAM model (Parallel Random Access Memory) consists of independent synchronized processors that share a common global memory. Despite being removed from physical reality, the PRAM model has proven to be extremely useful for studying parallel computation. This convenient and easy to work with model provides us with a setting divorced from communication issues.

Often, the PRAM algorithms can be modified easily to run efficiently on many real machine architectures. Of course, it may be necessary to change the algorithm completely to utilize the special features of a particular architecture. In the justification of PRAM's we would also like to note that a variety of good simulation results are also known, i.e., simulations of PRAM's on realistic machine models. In particular, Ranade [Ran87] showed how to simulate an N–processor PRAM on an N–node Butterfly architecture very efficiently, with a slowdown of an $O(\log N)$ factor. This result enables one to develop algorithms in the PRAM model and then simulate the algorithm efficiently on a network of processors.

1.1 Specific Graph Problems and Results

We now discuss some of the problems that we study in this thesis and outline the results obtained. These results appeared in [KMV89], [KS89] and [Khu89].

1.1.1 The Two Disjoint Paths Problem

Given a graph G and two pairs of vertices s_1, t_1 and s_2, t_2, the two disjoint paths problem asks for vertex-disjoint paths connecting s_i with $t_i, i = 1, 2$. This problem is a special case of undirected two-commodity integral flow in which every edge has unit capacity (the problem is NP-complete if the capacities are arbitrary) and has obvious applications in certain routing situations. It has been well-studied from the point of view of sequential computation [PS78], [Shi80], [Sey80]; we give a fast parallel algorithm for it. In case G is non-planar, our algorithm finds a Kuratowski homeomorph in G (i.e., a subgraph homeomorphic to $K_{3,3}$ or K_5). This complements the known NC planarity algorithms that give a planar embedding in the positive case: our algorithm provides a certificate of non-planarity in the negative case. In each case, the processor-time product of our algorithms is within a polylogarithmic factor of the best known sequential algorithm.

An important step in solving the general problem is solving the planar case. A linear time algorithm for this case was given by Perl and Shiloach [PS78]. Shiloach [Shi80] showed how to solve the non-planar case as follows: Find a Kuratowski homeomorph— if it is a $K_{3,3}$ homeomorph, then use it as a high connectivity "switch" to find the two disjoint paths; if it is a K_5 homeomorph, use the algorithm of [Wat68]. Independently, Seymour [Sey80] gave a polynomial time algorithm for the decision problem for general graphs; by self reducibility, this yields an algorithm for the search problem.

Using theorems from [PS78] and [Shi80], it is easy to show that the decision version of the two paths problem is in *NC* (see Chapter 3). Our main contribution is to give an *NC* algorithm for the search version, i.e., for actually finding the two disjoint paths. As pointed out in [KUW85], an *NC* algorithm for the decision version does not directly yield an *NC* algorithm for the search version, since ordinary self-reducibility is a sequential process. We present algorithmically relevant structure (the *pq-graph* theorem) that makes the parallelization possible. Our algorithm requires $O(n \log\log n/\log n)$ processors and $O(\log n)$ time. This structure also yields a *much simpler* linear time sequential algorithm for the planar case.

Our proof of the *pq-graph* theorem is patterned on the familiar "crossing lemma", which states that a cycle C with four vertices a, b, c and d in this order cannot have a-c and b-d paths lying inside C. (This lemma follows from the Jordan Curve Theorem.) Both the crossing lemma and our *pq*-graph theorem relate the combinatorics of certain planar graphs to the geometry of their embeddings.

Our planar graph algorithm extends to $K_{3,3}$-free graphs using a theorem of Hall [Hal43], as in [Vaz89]. The main difficulty in dealing with graphs containing a $K_{3,3}$ homeomorph is finding such a homeomorph in *NC*. The decision version of this problem was shown to be in *NC* in [Vaz89], and parallelizing the search version was left as an open problem. We solve this problem as follows: find any Kuratowski homeomorph in the graph, and in case this is a K_5, use it to find a $K_{3,3}$ homeomorph.

The Kuratowski homeomorph is found as follows: Given a non-planar graph G, use a parallel planarity testing algorithm to obtain a subgraph G' of G and an edge (u, v), such that G' is planar and $G' + (u, v)$ is non-planar. Obtain a maximal cycle containing vertices u and v in G', and, using ideas from Kuratowski's theorem, find a Kuratowski homeomorph in $G' + (u, v)$. This algorithm requires $O(n \log\log n/\log n)$ processors and $O(\log^2 n)$ time; we leave open the problem of reducing this to the complexity of planarity testing, which is currently $O(n \log\log n/\log n)$ processors and $O(\log n)$ time [RR89].

Once the $K_{3,3}$ homeomorph is found, we also use it as a high connectivity "switch" to find the two disjoint paths; however, additional ideas are required to make this algorithm processor efficient. Our bounds are $O(n^2)$ processors and $O(\log^2 n)$ time, compared to the $O(n^2)$ sequential running time for the non-planar case (using the algorithm of [Shi80], together with the $O(n^2)$ algorithm of [KR87] for four-connected components).

The k disjoint paths problem (where we need to find k vertex-disjoint paths connecting specified pairs of vertices) is NP-complete if k is part of the input [Kar75]. For fixed $k \geq 3$, the problem was open for a long time. Recently, Robertson and Seymour gave polynomial–time algorithms for this problem for any fixed k, using their extensive theory of graph minors [RS86a]. The following problem is central to the Robertson-Seymour theory: for a fixed graph H, decide whether a given graph contains H as a minor. This problem has closely related structure to the k disjoint paths problem. In fact, the minor problem polynomial–time reduces (even NC reduces) to the k disjoint paths problem [RS85]. Robertson and Seymour's polynomial–time algorithm for minor testing [RS86a] together with their proof of Wagner's conjecture [RS86b] has yielded non-constructive polynomial–time algorithms for testing membership in any minor-closed family of graphs. (Wagner's conjecture states that every minor-closed family of graphs has a finite list of forbidden minors.) For certain families (i.e., when the list of forbidden minors is known), this result gives explicit polynomial–time algorithms as well (but, because of large multiplicative factors in the running time of the minor testing algorithm, these algorithms are not practical).

A natural next question is whether the Robertson-Seymour theory can be used to obtain NC algorithms for the k disjoint paths problem, for fixed k. Robertson and Seymour concentrate on the decision version of their problems and rely on self-reducibility for solving the search version. One may need more structural properties in order to find parallel algorithms for the search versions. An NC algorithm for the k disjoint paths problem will immediately imply that testing for membership in any minor-closed family of graphs is in NC, without actually producing such an algorithm. As above, if the list of forbidden minors is known, this result will give an explicit NC algorithm as well; several natural problems fall under this category (see e.g., [MRS88]). We leave open the problem of testing for graph minors in NC.

1.1.2 Graph Connectivity and Disjoint s-t Paths

Graph connectivity is a classical problem in graph theory [Har69,Ber76,Eve79] and has many practical applications, e.g., VLSI design, reliability of communication networks etc. Efficient parallel algorithms for testing connectivity [SV82,CV86], biconnectivity [TV85], triconnectivity [FRT89], and four-connectivity [KR87] of graphs have been developed. 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 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 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.

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, 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. 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.

Our k vertex-disjoint paths algorithm can be extended to obtain k edge-disjoint paths with the same complexity. The algorithm either finds 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]. It is also used as a subroutine in the parallel algorithms for the two paths problem. We also use it to obtain efficient parallel algorithms for the subgraph homeomorphism problem for some fixed pattern graphs. In particular, we can test if a given graph contains subgraphs homeomorphic to the pattern graphs K_4 and $K_{2,3}$. This gives us an almost optimal parallel algorithm to test whether a graph is outer-planar.

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 as well.

1.2 A Few Words about PRAM's

A PRAM consists of several independent processors, each with its own private memory, communicating with the other processors through a globally shared memory. In one time unit, a processor can either read or write a memory location, or execute a single RAM operation. We assume that the processors are all synchronized.

There are several ways of resolving read and write conflicts in a PRAM, when various processors try to access the same memory location. These conflicts can be handled in different ways, and thus give variants of the PRAM. The weakest of the PRAM models is the EREW model (Exclusive Read Exclusive Write), in which simultaneous access by two or more processors to the same memory location is dissallowed. The most powerful variant is the CRCW model (Concurrent Read Concurrent Write), in which many processors are permitted to read and write the same memory location simultaneously. When many processors are attempting to do a write into a single location, we assume that an arbitrary one succeeds. In the CREW model, we permit simultaneous reads from a memory location, but disallow simultaneous writes. The model we will be using is the CRCW PRAM.

Even though there are many PRAM models, it has been shown that they do not differ much in their computational power. For example, it is known that any algorithm for the CRCW model can be simulated by the EREW model with the same number of processors and with the parallel time being increased by a factor of $O(\log p)$ where p is the number of processors.

Often, the study of parallel algorithms focusses on the trade-off between the time for an algorithm and the number of processors it uses. Despite the fact that the number of available processors is fixed, we continue to design algorithms in which the number of processors grows as a function of the size of the problem instance. After an algorithm is designed, one can always simulate it on a fixed number of processors.

Of particular interest to us are the so–called *efficient* algorithms, which run in *polylog time* and in which the processor-time product (or total work done) exceeds the number of steps in an optimal sequential algorithm by at most a polylog factor. A major goal in the design of parallel algorithms is to find efficient algorithms with as small a running time as possible. We note that simulation results between the various PRAM models make the notion of an *efficient* algorithm invariant with respect to the particular model being used.

For the problems we try to solve efficiently on PRAM's, the bounds we are usually interested in are polylog time using a polynomial number of processors. The class *NC* is defined to be the class of problems that can be solved within these resource bounds by deterministic algorithms.

1.3 Overview of Thesis

In Chapter 2, we give some preliminary defintions and basic notation used in the rest of the thesis. In Chapters 3 and 4 we give an algorithm for the two paths problem.

The first chapter presents the algorithm for the case when the graph is planar, and the second chapter gives an algorithm for finding a $K_{3,3}$ in a graph (if it contains one) and shows how to use the $K_{3,3}$ to solve the problem for general graphs.

In Chapter 5 we present the parallel algorithm for finding k vertex-disjoint s-t paths, and in Chapter 6 outline its applications to graph connectivity and various other problems. Finally, Chapter 7 ends with the conclusions and some open problems left by this work.

Chapter 2

Basic Terminology

2.1 Connectivity

Let $G = (V, E)$ denote 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.*

2.2 Triconnected Components

Let $\{a,b\}$ be a pair of vertices in a biconnected simple graph G. Suppose that the edges of G are divided into equivalence classes $E_1, \ldots, E_n$ such that two edges which lie on a common path not containing any vertex of $\{a,b\}$ except as an endpoint are in the same class. The classes E_i are called the *separation classes* of G with respect to $\{a,b\}$. If there are at least two separation classes, then $\{a,b\}$ is a *separation pair* of G unless there are exactly two separation classes, and one class consists of a single edge.

If G is biconnected such that no pair $\{a,b\}$ is a separation pair of G, then G is *triconnected*. Let $\{a,b\}$ be a separation pair of G. Let the separation classes of G with respect to $\{a,b\}$ be $E_1, \ldots., E_n$. Let $E' = \cup_{i=1}^{k} E_i$ and $E'' = \cup_{i=k+1}^{n} E_i$ be such that $\mid E' \mid \geq 2$, $\mid E'' \mid \geq 2$. Let $G_1 = (V(E'), E' \cup \{(a,b)\})$, $G_2 = (V(E''), E'' \cup \{(a,b)\})$.

The Graphs G_1 and G_2 are called the *split* graphs of G with respect to $\{a,b\}$. Replacing a graph G by two split graphs is called *splitting* G. A new edge (a,b) called a *virtual edge* is added in the split components if such an edge (a,b) is not already present in the graph G. Suppose G is split, the split graphs are split, and so on, until no more splits are possible (each remaining graph is triconnected). The graphs constructed in this way are called the *split components* of G.

2.3 On Homeomorphs and Minors

A planar embedding of a graph G, is a drawing of G in the plane such that no two edges intersect each other in the drawing (except at vertices). A graph is said to be planar if it has a planar embedding. A graph is said to be outer-planar if it has a planar embedding in which all the vertices are on a single face.

We now define what it means for G to contain a *subgraph homeomorphic* to H. If we can obtain H from G by either deleting some edges, or by performing contractions of edges that have one endpoint of degree 2, we say that G contains H as a homeomorph. Alternatively, we could think of G containing a subgraph H' that is a "subdivision" of H. (H' is a subdivisdion of H, if we can obtain H' from H by simply introducing vertices on edges.) A graph is said to be H-free if it does not contain H as a homeomorph. The following theorem that characterizes planar graphs is well known (see [Har69,BM77] for a proof).

Theorem 2.3.1 (Kuratowski) *A graph is planar if and only if it does not contain a subgraph homeomorphic to either K_5 or $K_{3,3}$.*

From this theorem it is easy to show the following.

Theorem 2.3.2 *A graph is outer-planar if and only if it does not contain a subgraph homeomorphic to either K_4 or $K_{2,3}$.*

A graph G is said to contain H as a *minor*, if we can obtain H from G, by either deleting some edges, or by performing contractions of edges. Note that there is no requirement that the edges that are contracted have one endpoint of degree 2 (as in the case of homeomorphs). So, it is possible that G may contain H as a minor but not as a homeomorph. For example, Peterson's graph contains K_5 as a minor but not as a homeomorph. There is an extensive literature on graph minors [RS86a].

2.4 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. 2.1, 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.

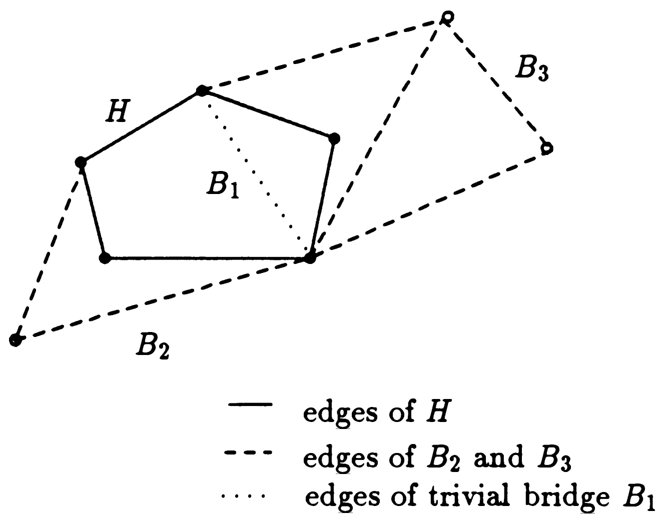


Figure 2.1: An example illustrating *bridges*

A bridge with k vertices of attachment is called a k-bridge. Two k-bridges with the same vertices of attachment are *equivalent* k-bridges. The vertices of attachment of a k-bridge B with $k \geq 2$ effect a partition of C into edge-disjoint paths, called the *segments* of B. Two bridges *avoid* one another if all the vertices of attachment of one bridge lie in a single segment of the other bridge; otherwise they *overlap*. Two bridges B and B' are *skew* if there are four distinct vertices u, v, u', v' of C such that u and v are vertices of attachment of B, u' and v' are vertices of attachment of B', and the four vertices appear in the order u, u', v, v' on C. It is shown in [BM77] that if two bridges overlap, then they are either skew or equivalent 3-bridges.

Chapter 3

The Two Paths Problem: Planar case

We first outline the algorithm for the decision version of the problem (for general graphs), and then proceed to give the algorithm for actually constructing the two paths in planar graphs. The algorithm for finding the paths in general graphs will be given in Chapter 4.

3.1 *NC* Algorithm for the Decision Problem

Lemma 3.1.1 *Solving the two disjoint paths (search or decision) problem for a graph G, NC-reduces to solving it for the triconnected components of G.*

Proof: Decompose G into its "tree" T' of triconnected components [Vaz89]. The lemma is obvious if s_i, t_i $(i = 1, 2)$ are all in the same triconnected component. Suppose s_i, t_i $(i = 1, 2)$ are in components S_i and T_i of T' respectively (see Fig. 3.1). Let N_i be the path from S_i to T_i in T'. If N_1 and N_2 do not intersect the disjoint paths are easy to obtain. Suppose N_1 and N_2 intersect in a path $P_0, P_1, .., P_k$ in T'. Let (a_i, b_i) be the separating pair between P_i and P_{i+1}. In each triconnected component P_i find "parallel" disjoint paths $p(a_{i-1}, a_i)$, $q(b_{i-1}, b_i)$ and "cross" disjoint paths $p'(a_{i-1}, b_i)$, $q'(b_{i-1}, a_i)$. (We use the notation $P(v_i, v_j)$ for a path from vertex v_i to v_j.) If any component allows both parallel and cross paths, then using one of these we can always find the disjoint paths. Otherwise, there is a unique way of connecting $\{a_0, b_0\}$ with $\{a_{k-1}, b_{k-1}\}$. We can now determine whether this allows disjoint s_i, t_i paths. □

Henceforth we will assume that G is triconnected.

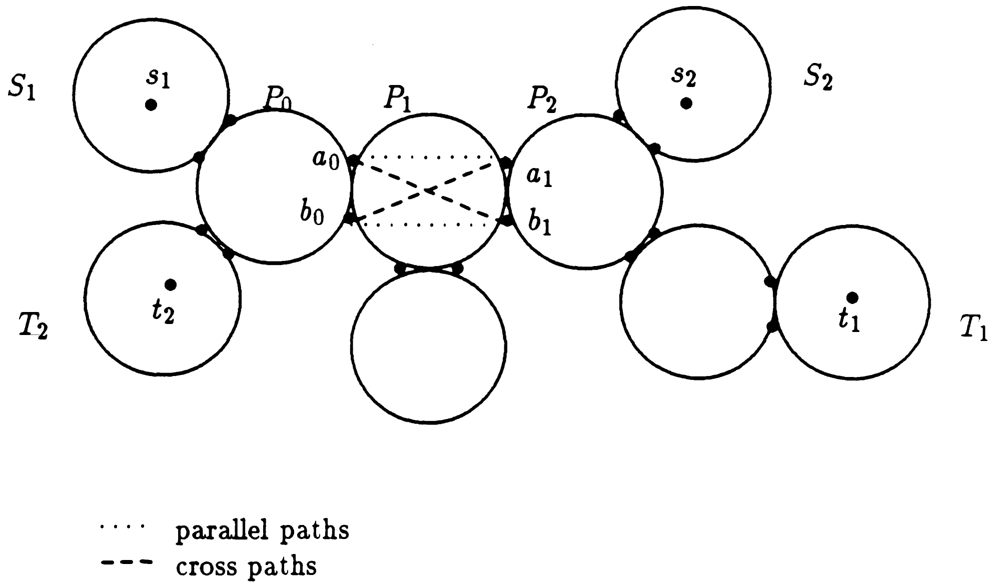


Figure 3.1: Decomposition tree T' of triconnected components

The decision algorithm:
Step 1. Find the separating triples in G [KR87]. For each triple $\{a, b, c\}$ discard the components of $G - \{a, b, c\}$ that do not contain a vertex from $\{s_1, s_2, t_1, t_2\}$. Add virtual edges between the vertices in each triple. Let G' be the new graph.
Step 2. If G' is planar then the answer to the decision version is "no" iff there is a face containing all four vertices s_1, s_2, t_1, t_2 in this order (by Theorem 3.1.2 below and the fact that a triconnected graph has a unique planar embedding [Whi33]).
Step 3. If G' is non-planar then the answer is "yes" [Shi80].

Theorem 3.1.2 (Perl-Shiloach) *Let G be a triconnected planar graph, and let s_1, t_1, s_2, t_2 be four vertices of G. Two disjoint paths from s_1 to t_1 and from s_2 to t_2 exist if and only if the vertices s_1, s_2, t_1, t_2 are not on a common face in this order.*

3.2 Two Disjoint Paths–the Planar case

Our algorithm for finding disjoint paths in case G is planar and triconnected relies on the structure of "pq-graphs", described below. Since G is triconnected, we can find three vertex-disjoint paths p_a, p_b, and p_c from s_1 to t_1, and similarly three vertex-disjoint paths q_a, q_b, and q_c from s_2 to t_2. (In Chapter 5, we will give a fast parallel algorithm to solve this problem.) We denote by G_{pq} the subgraph of G consisting of the p_i and q_j paths. We denote by $p[v_i; v_j]$ the segment of the path p from v_i to v_j.

Definition 1 *G_{pq} will be called a pq-graph if there is a numbering for the p_i and q_j paths such that when the p_i's are ordered from s_1 to t_1, and the q_j's from s_2 to t_2, the following six conditions are satisfied.*

1. *The first intersection of each q_j with any p-path is a vertex of p_1.*
2. *The last intersection of each q_j with any p-path is a vertex of p_3.*
3. *Let $v_1 \in q_j \cap p_1$, $v_3 \in q_j \cap p_3$ $(v_1, v_3 \neq s_1, t_1)$. Then $q_j[v_1; v_3]$ intersects p_2.*
4. *The first intersection of each p_i with any q-path is a vertex of q_1.*
5. *The last intersection of each p_i with any q-path is a vertex of q_3.*
6. *Let $v_1 \in q_1 \cap p_i$, $v_3 \in q_3 \cap p_i$ $(v_1, v_3 \neq s_2, t_2)$. Then $p_i[v_1; v_3]$ intersects q_2.*

Lemma 3.2.1 *If G_{pq} is not a pq-graph, then there exist disjoint s_1,t_1 and s_2,t_2 paths in G_{pq}. These paths can be obtained in $O(\log n)$ time using $O(n \log\log n / \log n)$ processors.*

The proof builds on ideas from [PS78], where they impose conditions 1–3. In fact, as can be seen from their solution; one of the paths can always be taken to be either a p path or q path. We now show the following theorem.

Theorem 3.2.2 *If G contains vertex-disjoint s_i, t_i paths, $i = 1, 2$, and if G_{pq} is a pq-graph, then one of these paths can be taken to be either p_2 or q_2.*

A complete proof is given in the next section.

The search algorithm for planar graphs:
Step 1. Find a planar embedding of G. If in the planar embedding s_1, s_2, t_1 and t_2 are vertices on some face F of G in this order, then there is no solution.
Step 2. Find three vertex-disjoint p-paths from s_1 to t_1 and three vertex-disjoint q-paths from s_2 to t_2. Let G_{pq} be the graph consisting of these six paths.
Step 3. Now either,

1. s_2, t_2 are in the same connected component in $G - p_i$ (for some i), or
2. s_1, t_1 are in the same connected component in $G - q_i$ (for some i).

Step 3 works because, if G_{pq} is not a pq-graph then using Lemma 3.2.1 one of the paths can be taken to be either a p path or a q path. If G_{pq} is a pq-graph then by Theorem 3.2.2, one of the two paths can be taken to be either p_2 or q_2.

Theorem 3.2.3 *For planar graphs, there is an $O(\log n)$ time algorithm for the two disjoint paths search problem using $O(n \log\log n / \log n)$ processors.*

Using the *NC* algorithms of [TV85], [FRT89], and [RR89] we can implement the above algorithm in the stated time and processor bounds. The sequential complexity of the above algorithm matches that of [PS78] and is $O(n)$.
Remark: The main bottleneck is the triconnectivity and planarity algorithms of [FRT89,RR89] that use $O(n \log\log n/\log n)$ processors. All the other algorithms used as subroutines are "almost" optimal.

3.3 Proof of the pq-graph Theorem

The proof of Theorem 3.2.2 uses the Jordan regions of a cycle in a plane embedded graph to derive results about the ordering of vertices on the cycle. The "crossing lemma", stated below, is prototypical of such an argument.

Let $G = (V, E)$ be a plane embedded graph. A cycle C partitions the plane into itself and two connected open *Jordan regions* (disjoint from C). A *face* of G is a cycle one of whose Jordan regions is empty, that is, contains no vertices or edges of G.

Theorem 3.2.2 will be presented here as a corollary to the following theorem.

Theorem 3.3.1 *In a pq-graph, vertices s_1, s_2, t_1, and t_2 appear in that order on a face which is vertex disjoint from p_2 and q_2 (except at the endpoints).*

Corollary 3.3.2 *If G contains vertex-disjoint s_1, t_1 and s_2, t_2 paths and if G_{pq} is a pq-graph, then one of these paths can be taken to be either p_2 or q_2.*

Proof: By Theorem 3.3.1 the vertices s_1, s_2, t_1, and t_2 appear in order on a face C of G_{pq}, where C is disjoint from p_2 and q_2 (except at s_1, s_2, t_1, and t_2). Define the four *sections* of C to be the subpaths from s_1 to s_2, from s_2 to t_1, from t_1 to t_2, and from t_2 to s_1. Call the empty region of C the *inside.* If G has no path inside of C between two different sections, then the vertices s_1, s_2, t_1, and t_2 lie on a face of G. (Recall that G_{pq} is a subgraph of G.)

Therefore, if the two disjoint paths exist, there is some path r in $G - G_{pq}$ inside C connecting two different sections of C. Then we can take one path to include r and edges of C, and the other can be taken to be either p_2 or q_2. □

Some geometrical lemmas

Before we prove Theorem 3.3.1, we present a few useful lemmas. The first is an old friend, and needs no proof.

Lemma 3.3.3 (Crossing lemma) *If C is a cycle in the plane containing distinct vertices a, b, c, and d in that order, then there do not exist disjoint paths p joining a and c, and q joining b and d, both lying inside C.*

(We have already used the crossing lemma implicitly in the proof of Corollary 3.3.2.)

Lemma 3.3.4 (Endpoint lemma) *In a pq-graph,*

(i) $s_1 \notin q_2$; *(iii)* $t_1 \notin q_1$; *(v)* $s_2 \notin p_2$; *(vii)* $t_2 \notin p_1$;
(ii) $s_1 \notin q_3$; *(iv)* $t_1 \notin q_3$; *(vi)* $s_2 \notin p_3$; *(viii)* $t_2 \notin p_2$.

Proof: Vertex s_1 cannot lie on q_2, since any path p_i must hit q_1 first before hitting q_2. The other assertions follow similarly. □

Let C be a cycle in a plane embedded graph. If vertices s and t do not lie on C, then they are *separated* by C if they lie in distinct Jordan regions of C.

Given a cycle C and a path p, a *touching* of p and C is a maximal common subpath (with one or more vertices). A *crossing* of p and C is a touching by which p crosses from one Jordan region of C to the other.

Given a path p and vertex-disjoint paths q_1, q_2, and q_3, a *segment* of p is a maximal subpath s which is disjoint from the q_i paths, except possibly at the endpoints of s. A q_i-q_j *hit* is a segment of p whose endpoints lie in q_i and q_j.

Lemma 3.3.5 (Cycle lemma) *Let G be a plane embedded graph, and suppose that vertices s, t are joined by vertex-disjoint paths q_1, q_2, and q_3. Suppose that C is a cycle separating s and t. Then C has a q_1-q_3 hit.*

Remark: Symmetrically, C must also have a q_1-q_2 hit and a q_2-q_3 hit.
Proof: Since s and t lie in opposite Jordan regions of C, each path q_i must contain an odd number of C crossings. Since there are three q paths, C must contain an odd number of q path crossings.

Suppose that C has no q_1-q_3 hit. Denote by $\mathbf{Q_{13}}$ the region bounded by the cycle q_1q_3, not containing q_2. Similarly define $\mathbf{Q_{12}}$ and $\mathbf{Q_{23}}$. Denote

$$\begin{aligned} \tilde{\mathbf{Q}}_{\mathbf{12}} &= \mathbf{Q_{12}} \cup q_1 \\ \tilde{\mathbf{Q}}_{\mathbf{23}} &= \mathbf{Q_{23}} \cup q_2. \end{aligned}$$

Orient the cycle C. Let v be a vertex on C, and let w be the vertex preceding v in the orientation of C. Assign to v a state $Q(v)$ as follows:

$$Q(v) = \begin{cases} \tilde{\mathbf{Q}}_{\mathbf{12}} & \text{if} \quad v \in \tilde{\mathbf{Q}}_{\mathbf{12}}, \\ \tilde{\mathbf{Q}}_{\mathbf{23}} & \text{if} \quad v \in \tilde{\mathbf{Q}}_{\mathbf{23}}, \\ \mathbf{Q_{13B}} & \text{if} \quad v \in q_3, \\ \mathbf{Q_{13A}} & \text{if} \quad v \in \mathbf{Q_{13}} \quad \text{and} \quad Q(w) = \mathbf{Q_{13A}} \\ & \text{or} \quad v \in \mathbf{Q_{13}} \quad \text{and} \quad Q(w) = \tilde{\mathbf{Q}}_{\mathbf{12}}, \\ \mathbf{Q_{13B}} & \text{if} \quad v \in \mathbf{Q_{13}} \quad \text{and} \quad Q(w) = \mathbf{Q_{13B}} \\ & \text{or} \quad v \in \mathbf{Q_{13}} \quad \text{and} \quad Q(w) = \tilde{\mathbf{Q}}_{\mathbf{23}}. \end{cases}$$

Then it is easy to verify that, traveling along C from an arbitrary initial vertex c_0. the possible state transitions are exactly those indicated by the diagram in Fig. 3.2.

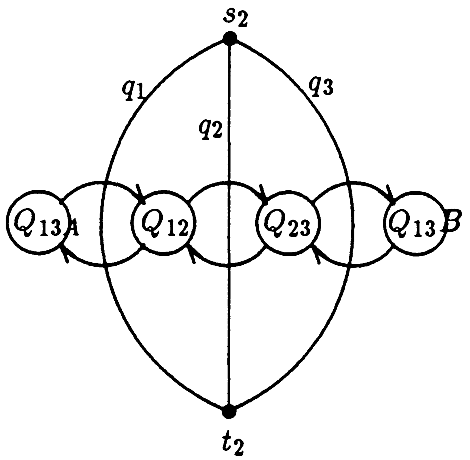


Figure 3.2: Possible state transitions for C.

Also, it is easy to see that a cycle whose state transitions obey this diagram must have an even number of q path crossings. But we have argued that if C separates s and t then C has an odd number of crossings. Hence C must have a q_1-q_3 hit. □

Suppose that G is a pq-graph. Define the regions $\mathbf{Q_{ij}}$ and $\mathbf{P_{ij}}$ as in the proof of the cycle lemma. Note that $\mathbf{Q_{12}}$, $\mathbf{Q_{23}}$, and $\mathbf{Q_{13}}$ are pairwise disjoint, as are P_{12}, $\mathbf{P_{23}}$, and $\mathbf{P_{13}}$.

Lemma 3.3.6 (Region lemma) *In a pq-graph*

(i) $s_1 \in \mathbf{Q_{13}} \cup q_1$; *(iii)* $t_1 \in \mathbf{Q_{13}} \cup q_3$;

(ii) $s_2 \in \mathbf{P_{13}} \cup p_1$; *(iv)* $t_2 \in \mathbf{P_{13}} \cup p_3$.

Proof: We will show that $s_2 \in \mathbf{P_{13}} \cup p_1$. The other assertions follow similarly.

By the endpoint lemma, if s_2 lies on some p path, it lies on p_1. Suppose, therefore, that s_2 does not lie on any p path.

Since the first segment of q_1 is a path from s_2 to p_1, s_2 cannot lie in $\mathbf{P_{23}}$. Similarly, $t_2 \notin \mathbf{P_{12}}$; by the endpoint lemma, $t_2 \notin p_1 \cup p_2$. Suppose $s_2 \in \mathbf{P_{12}}$. Then p_1p_2 is a cycle separating s_2 and t_2. By the cycle lemma, p_1p_2 must have a q_1-q_3 hit, contradicting the definition of pq-graphs. Therefore, $s_2 \in \mathbf{P_{13}}$. □

Let a be a vertex on p_1. Then we say that a path q_i *straddles* a if q_i has a p_1-p_1 segment lying in $\mathbf{P_{13}}$ whose endpoints lie on p_1 on either side of a. If, furthermore, a is the first hit of some path q_j with p_1, then we say that q_i straddles s_2. We make similar definitions for s_1, t_1, and t_2.

Lemma 3.3.7 (Straddle lemma) *No q path straddles s_2 or t_2. No p path straddles s_1 or t_1.*

Proof: Refer to Fig. 3.3. Suppose that q_i straddles a, the first hit of q_j on p_1. Let r

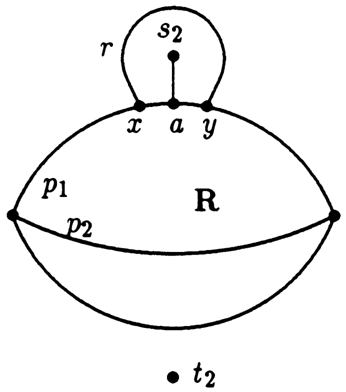


Figure 3.3: q_i cannot straddle s_2.

be a segment of q_i lying in $\mathbf{P_{13}}$ with endpoints $x, y \neq a$ lying on p_1 on either side of a. Let C be the cycle formed by r and the section of p_1p_2 containing p_2. Then from the definition of a pq graph it follows that C cannot have a q_1-q_3 hit.

Let C' be the cycle formed by r and the p_1 subpath xy. Then one region of C' is $\mathbf{R}' \subseteq \mathbf{P_{13}}$, and one region of C is $\mathbf{R} = \mathbf{R}' \cup \mathbf{P_{12}} \cup xy$. Because $a \in \mathbf{R}$ and q_j does not hit C before hitting a, $s_2 \in \mathbf{R}$.

By the region lemma $t_2 \notin \mathbf{P_{12}}$; by the endpoint lemma $t_2 \notin xy$. Also, $t_2 \notin \mathrm{R}'$, since the last hit of any q path is with p_3, not p_1. Therefore $t_2 \notin \mathbf{R}$, and C separates s_2 and t_2, contrary to the cycle lemma.

Therefore q_i cannot straddle s_2. The other assertions follow similarly. □

The proof of theorem 3.3.1

Let G be a pq graph. We start by constructing a *shell* C around $G - q_2$. Eventually, we show that C is the face required by the theorem.

Proposition 3.3.8 *$G - q_2$ has a directed cycle C (the* shell*) so that*

(i) *$G - q_2$ lies on the inside (right side) of C;*
(ii) *the outside (left side) of C lies within $\mathbf{P_{13}}$;*
(iii) *C is disjoint from p_2 (except at s_1 and t_1); and*
(iv) *C contains s_1, s_2, t_1, and t_2 in that order.*

Proof: We give the construction in three stages. (See Fig. 3.4.)
Stage I:

Consider first the subgraph consisting of paths p_1, p_2, and p_3, embedded so that these paths appear in clockwise order around s_1. Take initially for C the cycle p_1p_3. oriented so that p_1 is directed from s_1 to t_1. Then p_2 lies on the right of C, the outside of C is $\mathbf{P_{13}}$, and C is disjoint from p_2 except for the vertices s_1 and t_1, which appear on C.

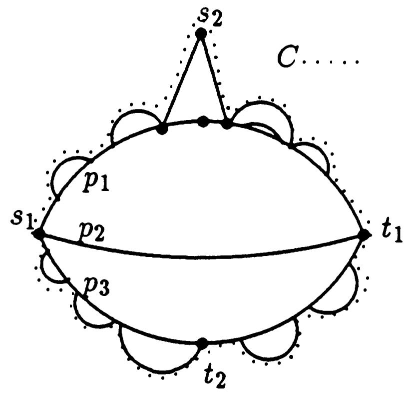


Figure 3.4: The shell C.

Stage II:

If $s_2 \notin p_1$, then by the region lemma $s_2 \in \mathbf{P_{13}}$. In this case, add to our subgraph the segments of q_1 and q_3 from s_2 to their first hit on p_1, and reroute C to take these q segments. Do similarly if $t_2 \notin p_3$. Then the enlarged subgraph still lies inside C, the outside of C lies within $\mathbf{P_{13}}$, C is disjoint from p_2, and the vertices s_2, t_1, t_2, and s_1 appear on C in that order.
Stage III:

Since every q path now begins and ends on C, any q subpath which escapes C must return to C. Add q_1 to our subgraph segment by segment, being careful to reroute C along every segment which would otherwise escape. Let r be such a segment. The endpoints of r must lie on a p path. Since r cannot have a p_1-p_3 hit, r does not straddle s_1 or t_1. By the straddle lemma, r does not straddle s_2 or t_2. Therefore C still contains s_2, t_1, t_2, and s_1 in order. Also, C continues to avoid p_2 and enclose the entire subgraph. In a similar manner, add q_3. □

The remainder of the proof is devoted to showing that q_2 also lies inside C. The next two propositions show that C is well behaved.

Denote by C^+ the oriented subpath of C from s_1 to s_2 to t_1. Similarly define C^- from t_1 to t_2 to s_1.

Proposition 3.3.9 *(i)* C^+ *contains the first hit of* p_1 *with* q_1*;*
(ii) C^+ *contains the last hit of* p_1 *with* q_3*;*
(iii) C^- *contains the first hit of* p_3 *with* q_1*;*
(iv) C^- *contains the last hit of* p_3 *with* q_3*;*
(v) C^+ *contains the first hit of* q_1 *with* p_1*;*
(vi) C^+ *contains the first hit of* q_3 *with* p_1*;*
(vii) C^- *contains the last hit of* q_1 *with* p_3*;*
(viii) C^- *contains the last hit of* q_3 *with* p_3*;*

Proof: (i) If $s_1 \in q_1$ then we need only observe that $s_1 \in C^+$. Otherwise, observe that the first hit c of p_1 with q_1 lies on C^+ in stage I of the above construction, and cannot be straddled by any q_1 or q_3 segment in stages II or III. (ii)–(iv) follow similarly.

(v) If $s_2 \in p_1$ then we need only observe that $s_2 \in C^+$. Otherwise, observe that the first hit d of q_1 with p_1 lies on C^+ in stage II of the above construction, and cannot be straddled by any q_1 or q_3 segment in stage III. (vi)–(viii) follow similarly. □

Proposition 3.3.10 *Suppose* $s_2 \notin p_1$*, and denote by* a *and* b *the first vertices of* q_1 *and* q_3 *upon* p_1*. Then these vertices appear on* C^+ *in the order* s_1*,* a*,* s_2*,* b*,* t_1*. Analogous statements hold at* t_1*,* t_2*, and* s_1*.*

Proof: Let c denote the first hit of p_1 on q_1, and let d denote the last hit of p_1 on q_3. By the previous proposition a, b, c, and d lie on C^+, with c being the first, and d being the last. By the crossing lemma applied to C and c, b, a, and d, b cannot precede a. By another application of the crossing lemma to c, b, s_2, and d, b cannot precede s_2. Similarly, s_2 cannot precede a. □

We now consider a cycle inside C, made up mostly of q_1 and q_3, which we will show contains q_2 on its inside.

Let g be the path inside C from s_1 to t_1 constructed by following p_1 from s_1 to q_1, q_1 to s_2, q_3 from s_2 to the last hit of q_3 along p_1, and p_1 to t_1. Similarly define g' from t_1 to t_2 to s_1. Then by the preceding proposition and an application of the crossing lemma g and g' are vertex-disjoint (except for s_1 and t_1). We will show that q_2 lies inside gg'. First we prove the following.

Proposition 3.3.11 *Consecutive hits of* g *with* C^+ *appear in forward order along* C^+*. Similarly, consecutive hits of* g' *with* C^- *appear in forward order along* C^-*.*

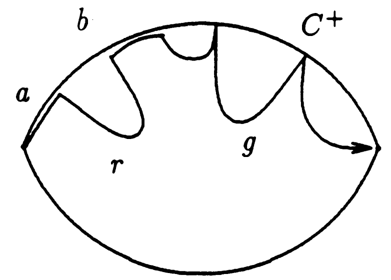


Figure 3.5: Consecutive g hits appear in order along C^+.

Proof: We show that g cannot pass through a vertex a on C^+ and subsequently hit a vertex b which precedes a on C^+. We proceed by induction on a along C^+. Initially, we take $a = s_1$. (See Fig. 3.5.)

If $a \neq t_1$ then g must proceed to hit C^+ at some vertex b which, by the induction hypothesis, succeeds a along C^+. If $b = t_1$, then g does not proceed beyond b. Suppose that $b \neq t_1$, and let r denote the a-b subpath of g. Since g does not self-intersect, g will hit neither a nor b again. Suppose that g hits some vertex $c \in C^+$ lying strictly between a and b. Let r' denote the subpath of g from c to t_1. Then r and r' violate the crossing lemma applied to C. Therefore g subsequently never hits any vertex preceding b on C^+. □

So far we have shown that the shell C is a face of the graph $G - q_2$ on which the vertices s_2, t_1, t_2, and s_1 appear in the given order, and that C is disjoint from p_2 (except for s_1 and t_1). We note that q_2 is disjoint from the cycle gg' (except at s_2 and t_2). All that remains to be shown is that q_2 lies inside the cycle gg', implying that q_2 (and hence all of G) lies inside C, and that q_2 is vertex-disjoint from C (except at s_2 and t_2).

We shall need the following. We say that gg' has four *sections*, meaning the t_2-s_1 subpath (section I), the s_1-s_2 subpath (section II), the s_2-t_1 subpath (section III), and the t_1-t_2 subpath (section IV).

Proposition 3.3.12 *No gg' segment of p_2 lying outside (to the left of) gg' has endpoints in two different sections.*

Proof: Suppose that p_2 has a gg' segment xy lying to the left of gg', with x and y in different sections of gg'. By the endpoint lemma $s_2, t_2 \notin p_2$, so neither x nor y is equal to s_2 or t_2. Therefore both gg' paths from x to y contain an s or t vertex, and in particular both paths hit C at some point strictly between x and y. If $x \neq s_1$ or t_1 let a be the last vertex of gg' (strictly) before x which lies on C, and let b be the next vertex of gg' (strictly) after x which lies on C. (See Fig. 3.6.) If $x = s_1$ or t_1, the choice of a and b depends on the direction of the p_2 segment. gg' hits y strictly after hitting b (and before returning again to a), so the cycle formed by the ab subpath of gg' and C encloses y in the region to its left. But gg' hits C after hitting y, so that gg' hits C after hitting b at a vertex between a and b. This contradicts the previous proposition. $\square$

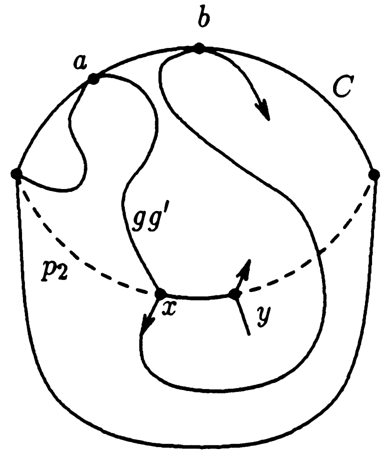


Figure 3.6: gg' cannot reach y.

Proposition 3.3.13 *Path q_2 lies inside (to the right of) gg'.*

Proof: Partition p_2 into its gg' segments. Since s_1 lies in sections I and II, and t_1 lies in sections III and IV, some gg' segment r of p_2 crosses from one of sections I or II to one of III or IV. By the previous proposition, r lies inside gg'. Segment r must hit q_1 first, q_3 last, and in between r must hit an internal vertex of q_2. Since q_2 is disjoint from gg', all of q_2 lies inside gg'. $\square$

This concludes the proof of theorem 3.3.1.

Chapter 4

The General Graph Case

In this chapter, we develop the algorithm to solve the two paths problem for the case of general graphs. When the graph is planar, we can use the algorithm outlined earlier. If the graph is a $K_{3,3}$-free graph, then using a theorem by Hall it is easy to solve the problem by reducing it to the planar graph case. Henceforth, we will assume that the graphs are non-planar and contain homeomorphs of $K_{3,3}$.

The algorithm for the case of general graphs proceeds by first finding a Kuratowski homeomorph in a non-planar graph. In case the obtained homeomorph is of a K_5, we show how to convert it to a $K_{3,3}$. Using the $K_{3,3}$ it becomes easy to obtain a parallel algorithm using the approach by [Shi80].

4.1 Extracting a Kuratowski Homeomorph

If C is a Jordan curve in the plane, then the rest of the plane is partitioned into two disjoint open sets called the *interior* and *exterior* of C. We denote the closures of the regions by *IntC* and *ExtC* respectively. In a plane graph G, each bridge of G relative to C is entirely contained in *IntC* or *ExtC*. A bridge in *IntC* (*ExtC*) is called an *inner* (*outer*) bridge.

Lemma 4.1.1 *If G is planar, and $G + e$ is non-planar and triconnected, then we can find a Kuratowski homeomorph in $G + e$ in $O(\log n)$ time with $O(n \log\log n / \log n)$ processors.*

Proof: Let u and v be the endpoints of e. The proof of Kuratowski's theorem in [BM77] relies on finding a cycle C of G that contains u and v and is such that the set *IntC* is maximal. We give a parallel algorithm for finding such a *maximal cycle*. The rest of the proof follows by a case analysis given in [BM77]. Since G is biconnected, we can find a cycle C in G containing u and v by finding two disjoint u-v paths (see

Chapter 5). Now consider the set of bridges of G with respect to the cycle C. Given a planar embedding of G, the bridges may be partitioned into two sets:
$Outer_C = \{B_i \mid B_i \text{ is embedded in } ExtC\}$
$Inner_C = \{B_j \mid B_j \text{ is embedded in } IntC\}$

A bridge $B_i \in Outer_C$ is in the set $Outer\text{-}skew_C$ if B_i is skew to (u, v); B_i is in the set $Outer\text{-}non\text{-}skew_C$ if it is not skew to (u, v). Since $G + e$ is non-planar, there is at least one bridge in the set $Outer\text{-}skew_C$ (otherwise e can be embedded in $ExtC$ in a planar embedding of G).

We first modify the cycle C to obtain a cycle C^1 containing u and v so that there are no bridges in the set $Outer\text{-}non\text{-}skew_{C^1}$ with respect to the cycle C^1. We then show how to modify C^1 to obtain cycle C^3 containing u and v such that the bridges in set $Outer\text{-}skew_{C^3}$ are single edges skew to (u, v).

Since G is biconnected, each bridge has at least two vertices of attachment on C. Let the attachment vertices of B_i on C be $x_i^1, x_i^2, ..., x_i^{k_i}$ (considered in clockwise order on C). We call $x_i^1(x_i^{k_i})$ the first (last) attachment vertex of B_i on C. The segment $C[x_i^1; x_i^{k_i}]$ is called the *attachment bar* of B_i on C. It was shown in [BM77] that outer bridges avoid one another. Hence, the attachment bar of each outer bridge can overlap with the attachment bar of another outer bridge only at an end vertex, and not at any internal vertex. Each bridge in the set $Outer\text{-}non\text{-}skew_C$ has all of its attachment vertices (and thus its attachment bar) on the segment $C[v; u]$ or $C[u; v]$.

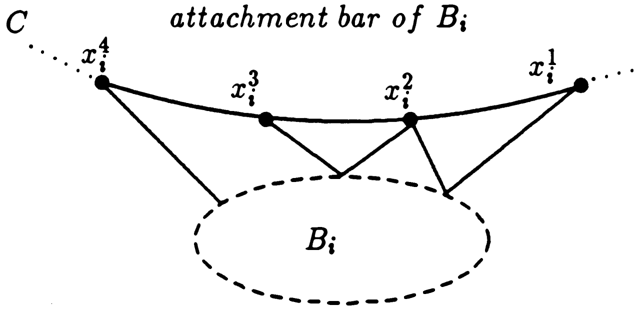


Figure 4.1: The graph B_i'

Consider the planar embedding of B_i and its attachment bar, and call the resultant graph B_i' (see Fig. 4.1). Note that B_i' is biconnected and planar. In B_i' the vertices x_i^1 and $x_i^{k_i}$ are on the outermost face in the planar embedding. Since B_i' is biconnected, the outer face of B_i' is a simple cycle C_i'. Let $P_i(x_i^1, x_i^k)$ be the subpath of C_i' from x_i^1 to $x_i^{k_i}$ avoiding the the attachment bar of B_i.

In C, replace all the segments which are attachment bars of some bridge $B_i \in$ *Outer-non-skew*$_C$ by the path $P_i[x_i^1; x_i^{k_i}]$. The new cycle C^1 contains both u and v, and all its outer bridges are skew to (u, v).

We perform the transformation from C^1 to C^3 in two stages. Consider the cycle C^1 and its bridges B_i in *Ext*C^1 which are skew to (u, v). If B_i has only two attachment vertices then clearly the bridge is only a path, and in fact, since the graph is triconnected, it is only a single edge. Assume that B_i has more than two attachment vertices on C^1. Let the attachment vertices on $C^1[u; v]$ be $x_i^1, x_i^2, \ldots, x_i^{k_i}$ (considered in order). The attachment vertices on $C^1[v; u]$ are $y_i^1, \ldots, y_i^{l_i}$ (see Fig. 4.2). We will replace $C^1[x_i^1; x_i^{k_i}]$ by a path P' in B_i to obtain C^2; and subsequently replace $C^2[y_i^1; y_i^{l_i}]$ by a path P'' in B_i to obtain C^3, the maximal cycle.

Obtain the subgraph B_i^1 of B_i by deleting the attachment vertices y_i on $C^1[v; u]$ (see Fig. 4.3). Consider the external face in a planar embedding of B_i^1. Find a path P from x_i^1 to $x_i^{k_i}$ along the external face. This path may not be simple as shown in Fig. 4.4. Find the articulation vertex closest to x_i^1 on P, and obtain a simple path P' by "short-circuiting" P. Replace the segment $C^1[x_i^1; x_i^{k_i}]$ by the path P' to obtain the cycle C^2. Do this step in parallel for every bridge in *Ext*C^1 skew to (u, v). Now identify the subgraphs B_i of G which form bridges with respect to the new cycle C^2. Let x_i^j and y_i^j be the attachment vertices of B_i on the upper and lower chains $C^2[u; v]$ and $C^2[v; u]$. Obtain the subgraph B_i^2 of B_i by deleting the attachment vertices x_i^j of B_i on $C^2[u; v]$. In a planar embedding of B_i^2 replace the segment $C^2[y_i^1; y_i^{l_i}]$ by the path P'' from y_i^1 to $y_i^{l_i}$ in the external face. If G is triconnected it is easy to see that the path P'' is simple (see Fig. 4.5). After the replacement we obtain cycle C^3; the only bridges in *Ext*C^3 are single edges skew to (u, v).

Consider the set *Inner*$_{C^3}$ of inner bridges of C^3. Since $G + e$ is non-planar, the set *Inner-skew*$_{C^3}$ is non-empty (else a planar embedding for $G + e$ can be obtained). If each bridge in *Inner-skew*$_{C^3}$ avoids every bridge in *Outer*$_{C^3}$, then each such bridge can be transferred to *Ext*C^3, yielding a planar embedding for $G + e$. Hence, there must be an inner bridge B_1 which overlaps an outer bridge B_2 and which is skew to (u, v).

At this point, by considering various cases for the possible configurations of attachment vertices of B_1 and B_2 on C^3, we can obtain a Kuratowski homeomorph as in [BM77]. The parallel algorithm takes $O(\log n)$ time using $O(n \log\log n / \log n)$ processors using [RR89] and [CV88]. □

From the proof of Lemma 4.1.1, we also conclude the following:

Lemma 4.1.2 *In a planar graph G, we can a find a maximal cycle C that contains u and v in $O(\log n)$ time using $O(n \log\log n / \log n)$ processors.*

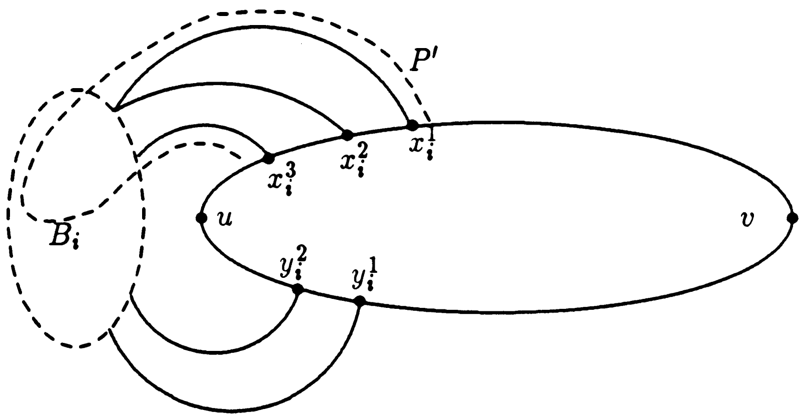


Figure 4.2: Bridge with attachment vertices skew to (u, v)

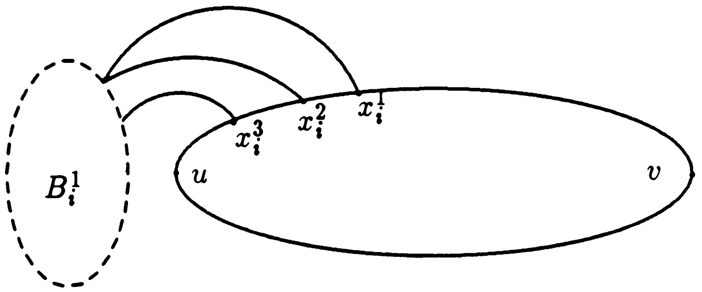


Figure 4.3: Subgraph B_i^1

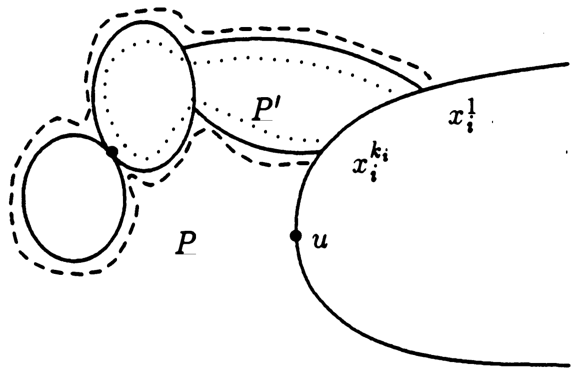


Figure 4.4: General form of path P'

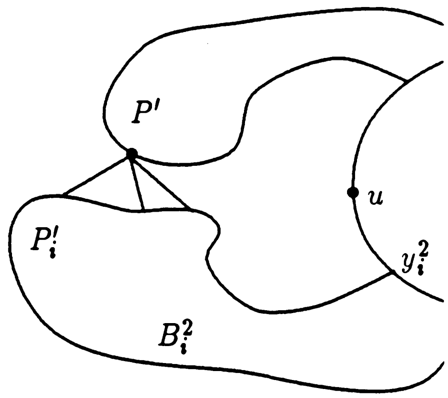


Figure 4.5: Path P_i' is s

Lemma 4.1.1 yields the parallel algorithm descr

Algorithm for finding a Kuratowski homeomc
Step 1. Order the edges in E arbitrarily as e_1, e_2.
consisting of the first i edges . Clearly, G_1 is plana
Step 2. Find the smallest index k such that G_1
$G_k (= G_{k-1} + e_k)$ is not, using the planarity testing
Step 3. Find a triconnected non-planar componen
$\in G'_k$ and that $G'_k - e_k$ is planar and biconnected.
$G'_k - e_k$ and edge e_k to find a Kuratowski homeom
Remarks: We can assume that G has $O(n)$ edges;
them, keeping G non-planar. Step 2 can be impleme
graphs G_i in parallel for planarity, or by doing a bi
Using the algorithm of [RR89], that takes $O(\log n)$ t
processors, we can implement the above algorithm
$O(n^2 \log\log n / \log n)$ processors, or in $O(\log^2 n)$ tim
cessors. We leave open the problem of finding a
same time and processor bounds as for planarity
processors and $O(\log n)$ time.

Theorem 4.1.3 *In a non-planar graph G, a Kurato in $O(\log n)$ time using $O(n^2 \log\log n / \log n)$ proces $O(n \log\log n / \log n)$ processors.*

4.2 Finding a $K_{3,3}$ Homeomorph

Our algorithm for finding a $K_{3,3}$ homeomorph is a parallelization of the sequential algorithm given in [Asa85]. This algorithm is based on the following theorem of Hall [Hal43].

Theorem 4.2.1 (Hall) *Each triconnected component of a $K_{3,3}$-free graph (i.e., a graph not containing a $K_{3,3}$ homeomorph) is either planar or exactly the graph K_5.*

Theorem 4.2.2 *There is an $O(\log n)$ time algorithm using $O(n^2 \log\log n/\log n)$ processors, or an $O(\log^2 n)$ algorithm using $O(n \log\log n/\log n)$ processors which finds a $K_{3,3}$ homeomorph (if one exists).*

Proof: First decompose G into its triconnected components. By Hall's theorem, one of the components is non-planar and contains at least six vertices. Use the algorithm of Theorem 4.1.3 to find any Kuratowski homeomorph G'. In case it is a homeomorph of K_5, call the vertices in G' of degree four v_1, v_2, v_3, v_4, and v_5. If G' is exactly the graph K_5, then the triconnected component containing G' must contain another vertex v not in G'. Obtain three vertex-disjoint paths from v to any three vertices of the K_5, say v_1, v_2, and v_3. These can be found by introducing an artificial sink vertex u and three edges from u to v_1, v_2 and v_3, and then finding three vertex-disjoint v-u paths. From the three paths and the K_5 it is easy to find a subgraph homeomorphic to $K_{3,3}$ by putting v_1, v_2, and v_3 in one partition of the $K_{3,3}$ and v, v_4, and v_5 in the other.

If the homeomorph G' is a subdivision of K_5, then consider a vertex u on the path $P(v_1, v_2)$, where v_1 and v_2 are vertices of degree four in G'. (P is a sub-division of an edge of the K_5.) Since G is triconnected, there must be a path in $G - \{v_1, v_2\}$ from u to some other vertex w of G'. Use this path to extract a $K_{3,3}$ homeomorph by considering where the path first hits G' (details in [Asa85]). □

4.3 Two Disjoint Paths–the non-Planar case

The two paths problem for $K_{3,3}$-free graphs reduces to the planar case via Hall's theorem as in [Vaz89]. For the remaining case when G contains a subgraph $G_{3,3}$ homeomorphic to $K_{3,3}$, we modify Shiloach's algorithm [Shi80]. Call the nine paths of $G_{3,3}$ representing the edges of $K_{3,3}$ *p-edges*, and call the six vertices of $G_{3,3}$ representing the six vertices of $K_{3,3}$ *p-vertices*.

Algorithm for triconnected graphs having a $K_{3,3}$ homeomorph:
Step 1. Find the separating triples in G [KR87]. For each triple $\{a, b, c\}$ discard the components of $G - \{a, b, c\}$ which do not contain a vertex from $\{s_1, s_2, t_1, t_2\}$. Add virtual edges between the vertices in each triple. Let G' be the new graph.
Step 2. Find a $K_{3,3}$ homeomorph in G' (else G' is $K_{3,3}$-free); call it $G_{3,3}$.
Step 3. Modify $G_{3,3}$ to include s_1 as one of the "corner" vertices. This is done by finding three-disjoint paths from s_1 to $G_{3,3}$. Call the modified homeomorph $G'_{3,3}$.
Step 4. Modify $G'_{3,3}$ further to include t_1 either as a corner vertex or on a p-edge incident to s_1. Call this homeomorph $G''_{3,3}$.
Step 5. Find four vertex-disjoint paths π_1, π_2, π_3 and π_4 connecting s_1, t_1, s_2 and t_2 with four p-vertices of $G''_{3,3}$ different from s_1 and t_1. These paths together with the $K_{3,3}$ homeomorph yield the two disjoint s_i, t_i, $i = 1, 2$, paths.

Step 1 can be parallelized using the algorithm in [KR87]. Theorem 4.2.2 yields a parallel algorithm for step 2. For steps 3 and 4, Shiloach gives a sequential path extension algorithm which modifies $G_{3,3}$. We parallelize this by working with *prefixes.* We describe the procedure in detail for step 3; the idea for step 4 is similar. Our case analyses is made more complicated by the fact that we *do not* do the preprocessing done by Shiloach, to obtain the "W-assumption" about the nature of the paths.

We make s_1 a p-vertex by constructing three-disjoint paths P_1, P_2, and P_3 from s_1 to three p-vertices on the same side of $G_{3,3}$. Let P_i have $v_i^1, v_i^2, ..., v_i^{k_i}$ as its vertices of intersection with the p-edges (considered in order from s_1). Let $P_i^1, ..., P_i^{k_i}$ denote the segments $P_i[s_1; v_i^1], ..., P_i[s_1; v_i^{k_i}]$ respectively. We define the *set of prefixes* (i, j, k) *of* P as $\{P_1^i, P_2^j, P_3^k\}$.

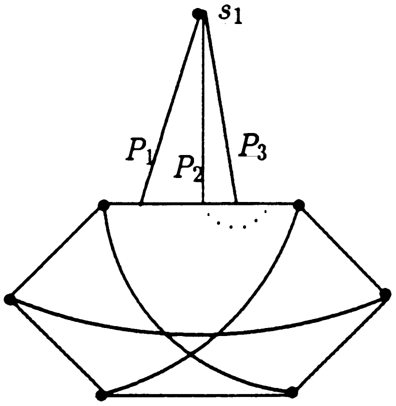
Lemma 4.3.1 *The subgraph $G_{3,3}$ and some set of prefixes yields a subgraph $G'_{3,3}$ of G homeomorphic to $K_{3,3}$, with s_1 as a p-vertex.*

Proof: The proof is based on an exhaustive case analysis. First consider the set $(1, 1, 1)$ of P. In case the prefixes are incident on different p-edges or on p-vertices, use one of the base cases from [Shi80], for which no extension of any prefix is required to get $G'_{3,3}$. The non-trivial case is that in which all three prefixes are incident on the same p-edge, say e_1, with endpoints v_1 and v_2. In this case extend the middle prefix to its next intersection with a p-edge (or a p-vertex). There are several cases to consider:

1. The extended segment hits e_1.

2. The extended segment hits a p-edge different from e_1.

3. The extended segment hits a p-vertex different from v_1 and v_2.

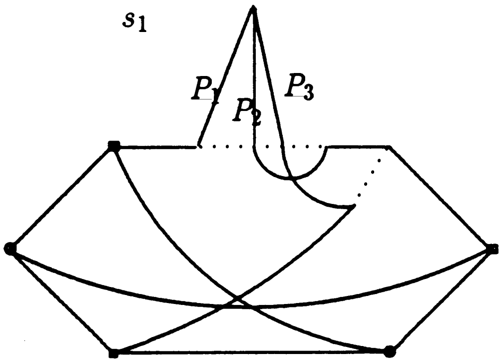


prefix (1,1,1)

Figure 4.6: Extend the middle prefix P_2

In the first case, continue extending the segments by extending the middle prefix at each step. Since the P_i-paths end at different p-vertices, this construction eventually falls into case two or three. By extending only the middle prefix, we ensure that the middle prefix does not intersect the two end segments $[v_1; x_1]$ and $[v_2; x_2]$ of e_1, where x_1 and x_2 are the end vertices of the non-middle prefixes. With this condition, in the last two cases the solution follows from Shiloach's base cases.

We illustrate this construction with an example. In Fig. 4.6 the middle prefix is of path P_2, which we extend to its second hit with a p-edge. If it hits a p-edge other than e_1, we immediately apply case two. Assume that it is e_1. Now the prefix of P_3 becomes the middle prefix, which we extend to its second hit with a p-edge. In this case it is a p-edge different from e_1; apply a base case to obtain $G'_{3,3}$ (see Fig. 4.7).



prefix (1,2,2)

Figure 4.7: Extending P_3 yields a base case

We illustrate a sequence of extensions of the middle prefixes in Fig. 4.8; the sequence of extensions terminates when P_2 hits a p-edge different from e_1. □

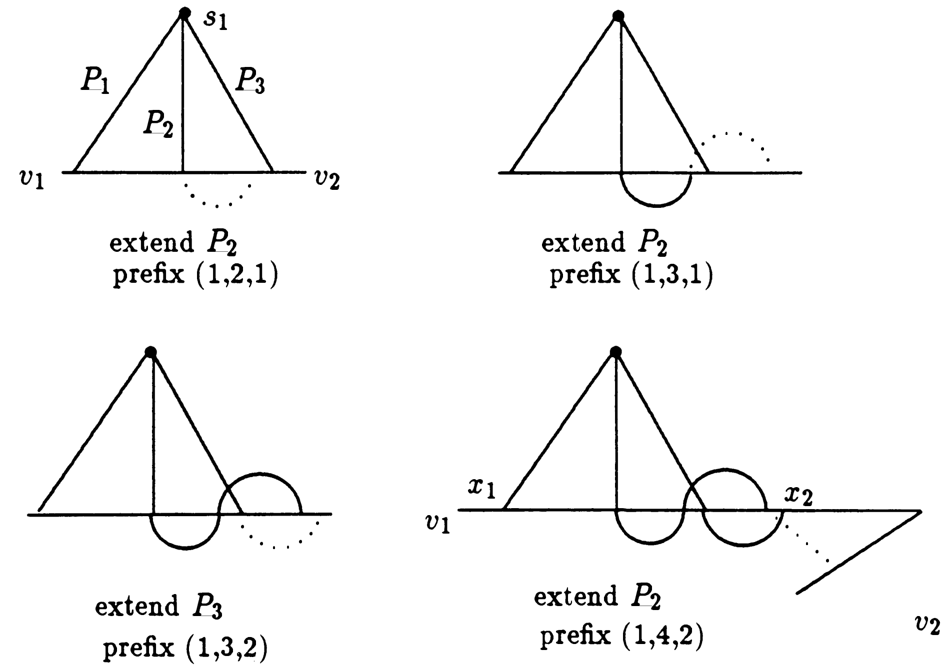


Figure 4.8: A sequence of extensions of prefixes of the P-paths

We now develop a parallel algorithm for finding the subgraph $G'_{3,3}$. First check all the base cases (when the first hits of the P_i paths yield $G'_{3,3}$). The only non-trivial case is when the first hits of all three P_i paths are on p-edge e_1. Now "guess" the ending configuration (there are only 6 of them) of the prefixes of the paths P_1, P_2 and P_3 on e_1. Extend the middle path (say P_2) of the guessed configuration until it first hits a p-vertex or an edge different from e_1. This can be done in $O(log\ n)$ time using $O(n)$ processors. After extending P_2, compute the leftmost and rightmost hits (x_l and x_r) of P_2 with p-edge e_1. The segments $e_1[v_1; x_l]$ and $e_1[x_r; v_2]$ are free of intersection with P_2. Extend paths P_1 and P_3 independently until they land on these segments in the same order as in the guessed configuration. We check to ensure that they do not have intersections with any p-edge other than e_1. (If they do, then we guessed the configuration incorrectly.)

Shiloach showed further that we can obtain $G''_{3,3}$ with s_1 as a p-vertex and with either:

1. t_1 also as a p-vertex, or

2. t_1 lying on a p-edge incident with s_1.

The construction is similar to that given above. Find disjoint paths Q_1, Q_2, and Q_3 connecting t_1 with three distinct p-vertices of $G'_{3,3}$; now there are more cases to consider, since the prefixes of the Q_i paths may land on p-edges incident to s_1 or on p-edges not incident to s_1.

Lemma 4.3.2 *The subgraph $G'_{3,3}$ and some set of prefixes yields $G''_{3,3}$ satisfying conditions 1 and 2.*

This construction is similar to the above construction and can be carried out in NC.

For Step 5 we introduce artificial source and sink vertices to construct the four π_i-paths connecting s_1, t_1, s_2 and t_2 with four p-vertices on $G''_{3,3}$ which are different from s_1 and t_1 (these are guaranteed to exist by Step 1). Shiloach showed how $G''_{3,3}$ could be used to make two disjoint connections, one between π_1 and π_2 and the other between π_3 and π_4, to yield the desired disjoint s_i, t_i paths.

There are three cases to consider:

1. s_1 and t_1 are p-vertices not connected by a p-edge.

2. s_1 and t_1 are p-vertices connected by a p-edge.

3. t_1 is on a p-edge incident with s_1.

Shiloach showed that by considering only the first hits of π_3 and π_4 we can obtain a solution for case 1.

The solution for case 2 is obtained as follows: the only non-trivial subcase (the others are symmetric) is as shown in Fig. 4.9. We first describe an $O(n)$ sequential "game" played by paths π_1, π_3, and π_4 on p-edges e_2 and e_3. (Our game is a modification of Shiloach's sequential algorithm.) We say that a prefix of path π_i *covers* a prefix of π_j if they land on the same p-edge with the endpoint of π_j closer to s_1 than the endpoint of π_i. The game starts with π_3. Inductively, we extend the current π path until it lands uncovered on p-edge e_2 or e_3, or until it leaves the game altogether by landing on a p-edge different from e_1, e_2, or e_3. Ultimately, one of the three paths leaves the game and we obtain prefixes allowing disjoint π_1-π_2 and π_3-π_4 connections.

We give a parallel algorithm to obtain such a prefix. First guess which p-edge each path ends on, and which path has left the game. There are only six distinct configurations and we check all of them. Suppose we guess that the first path to leave is π_1. Extend π_1 to its first hit on some edge other than e_1, e_2 or e_3. (The appropriate prefix of π_1 can be obtained by a prefix computation in $O(\log n)$ time using $O(n)$ processors.) Extend π_3 and π_4 to their shortest prefixes on edges e_2 and e_3 which are not covered by the extended prefix of π_1. If our guess was correct, then

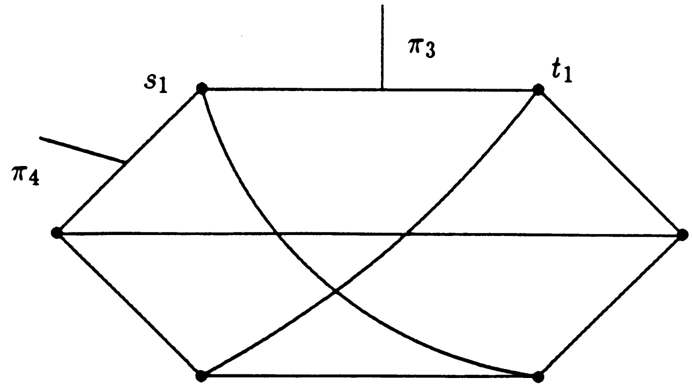


Figure 4.9: The only non-trivial subcase for case 2

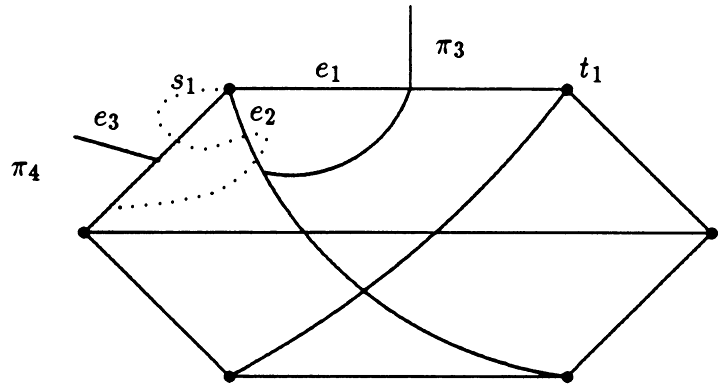


Figure 4.10: Paths π_3 and π_4 cover π_1

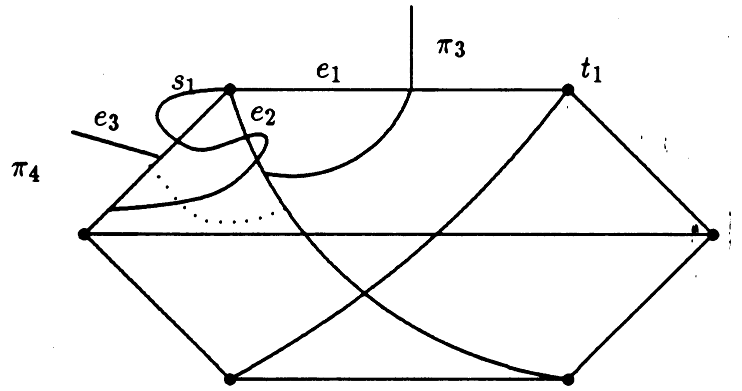


Figure 4.11: Path π_1 covers π_4

from the success of the sequential game we know that the extensions of π_3 and π_4 hit no p-edges besides e_1, e_2, and e_3. It is now easy to obtain the required disjoint connections.

The solution for case 3 is very similar to the solution for case 2. The only difference is that all four π_i paths participate in the game on all three edges e_i. Ultimately one of the four paths leaves the game to hit another edge. We guess which path is the first to leave and extend the other paths until they are not covered by the extended prefix of the first one. A solution can now be obtained (see [Shi80] for details).

Lemma 4.3.3 *One of the sets of prefixes of π_i along with the modified $G_{3,3}$ provides the disjoint π_1-π_2 and π_3-π_4 connections.*

Remark: Our Lemmas 4.3.1, 4.3.2, and 4.3.3 avoid Shiloach's "W-assumption", which relies on a sequential process, at the expense of more complicated case analyses.

The bottlenecks in the algorithm are finding four-connected components [KR87] and finding a $K_{3,3}$ homeomorph.

Theorem 4.3.4 *Given a graph G and vertices s_1, t_1, s_2 and t_2, we can solve the two paths problem in $O(\log^2 n)$ time using $O(n^2)$ processors.*

Chapter 5

Finding k disjoint paths

The algorithm for the two paths problem presented in Chapters 3 and 4, use as a subroutine an algorithm that computes vertex-disjoint paths between a pair of vertices. (Three paths for the planar case, and four for the general graph case.) In this chapter we will develop a parallel algorithm for finding k vertex-disjoint paths between a pair of vertices s and t (if they exist).

Our parallel algorithm for finding k vertex-disjoint s-t paths runs in $O(k^2 \log n)$ time using $kC(n,m)$ processors. For fixed k, our algorithm runs in logarithmic time using $nC(n,m)$ processors. From now on, we assume that k is fixed, and all factors of k will be elided from the complexity bounds.

Before we describe the general algorithm, we will explain the main idea of the algorithm for the simple case when $k = 2$. We also give a different way of computing two vertex-disjoint s-t paths using an st-numbering of a graph. Unfortunately, the technique does not seem to generalize to solving the problem for any $k > 2$. For completeness, we will describe this solution.

5.1 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.

5.1.1 Algorithm and Proof

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 5.1.1 *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. 5.1). (We view P_1 as directed from s to t.)

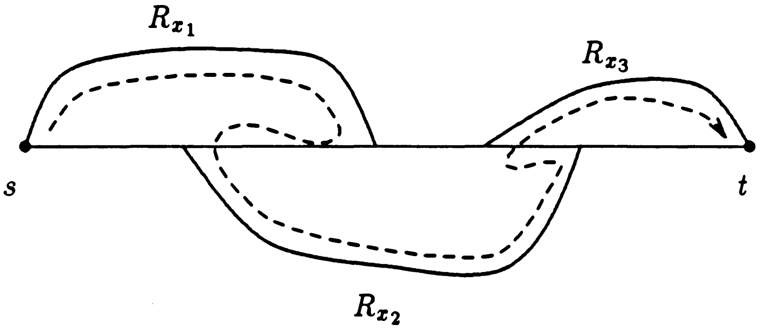


Figure 5.1: Path P_2

We claim that P_2 corresponds to an augmenting path in $N_{s,t}$, yielding a flow of two units (see Fig. 5.2). This follows from the following lemma.

Lemma 5.1.2 *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.

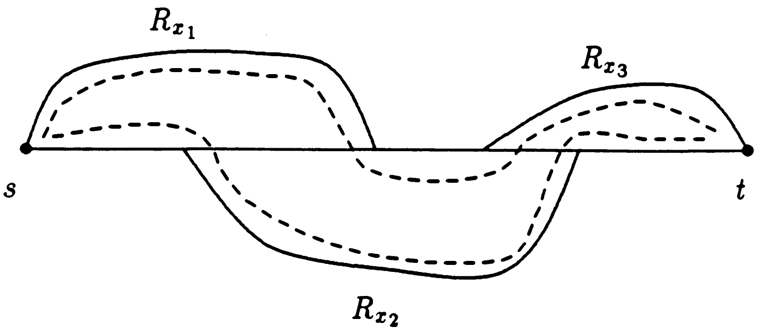


Figure 5.2: Obtaining two vertex-disjoint s-t paths

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 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. 5.3). Also note that P_2 is (edge) simple (i.e., it does not repeat any edges).

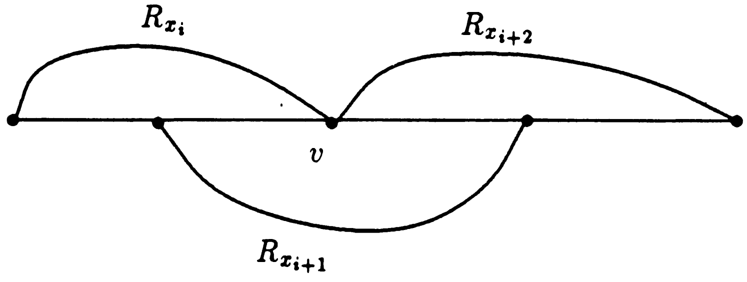


Figure 5.3: Vertex v occurs on P_2 twice

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. 5.9.) The proofs for each of these cases are given in the proof of Lemma 5.2.2 in the next section. (Lemma 5.2.2 is the version of this lemma for an arbitrary k.) □

The *only if* direction (of Theorem 5.1.1): 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. □

5.1.2 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].

5.1.3 Alternate Scheme using an st-numbering

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. 5.4.)

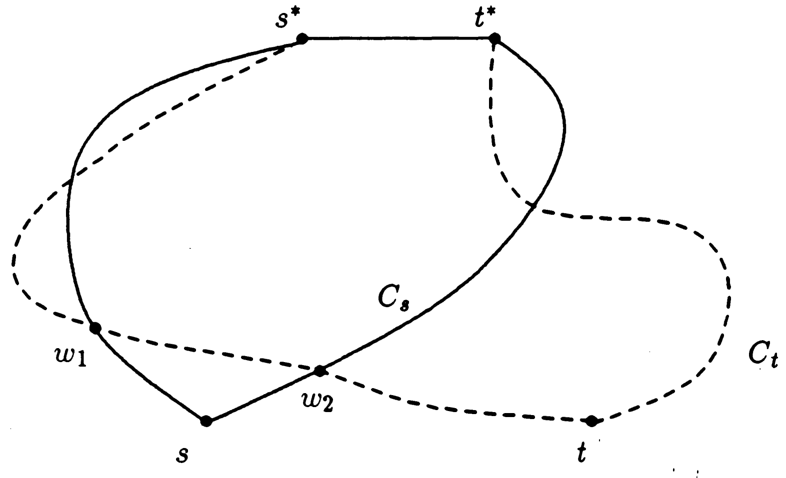


Figure 5.4: Obtaining two vertex-disjoint s-t paths

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].

5.2 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.

5.2.1 Algorithm and Proof

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 2 *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 3 *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:

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

In words, the edge e_j is $(s \to \beta_x)$ 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 $(s \to \beta_x)$ 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:

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

In words, the set T consists of edges $(\beta_j \to t)$, for all bridges B_j whose rightmost attachment is t. Notice that if $(\beta_x \to t) \in T$ 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 $(\beta_i \to \beta_x)$ 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. 5.5). More formally, the edge $(\beta_i \to \beta_x) \in D_j$ 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 $(\beta_i \to \beta_x)$ 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. 5.7.

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.

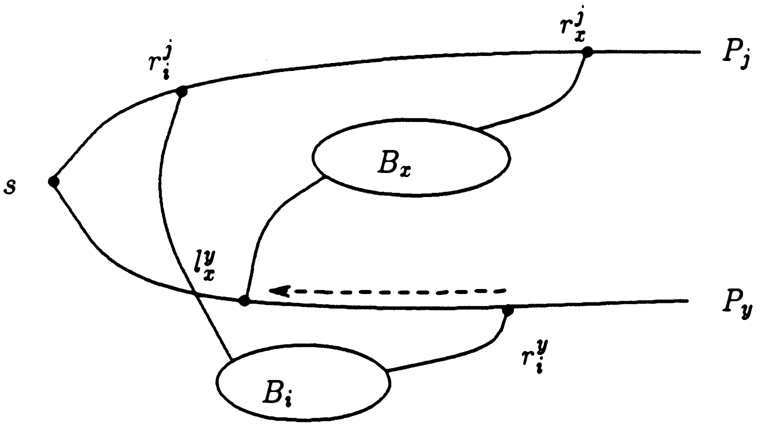


Figure 5.5: The edge $(\beta_i \to \beta_x)$ is added to D_j

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 5.2.1 *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 $(\beta_i \rightarrow \beta_x)$ in G_B, define the *type* of $(\beta_i \rightarrow \beta_x)$ to be y, if $r_i^y > l_x^y$. For example, in Fig. 5.5, 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 $1 \leq b \leq a-1$.

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. 5.6 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.

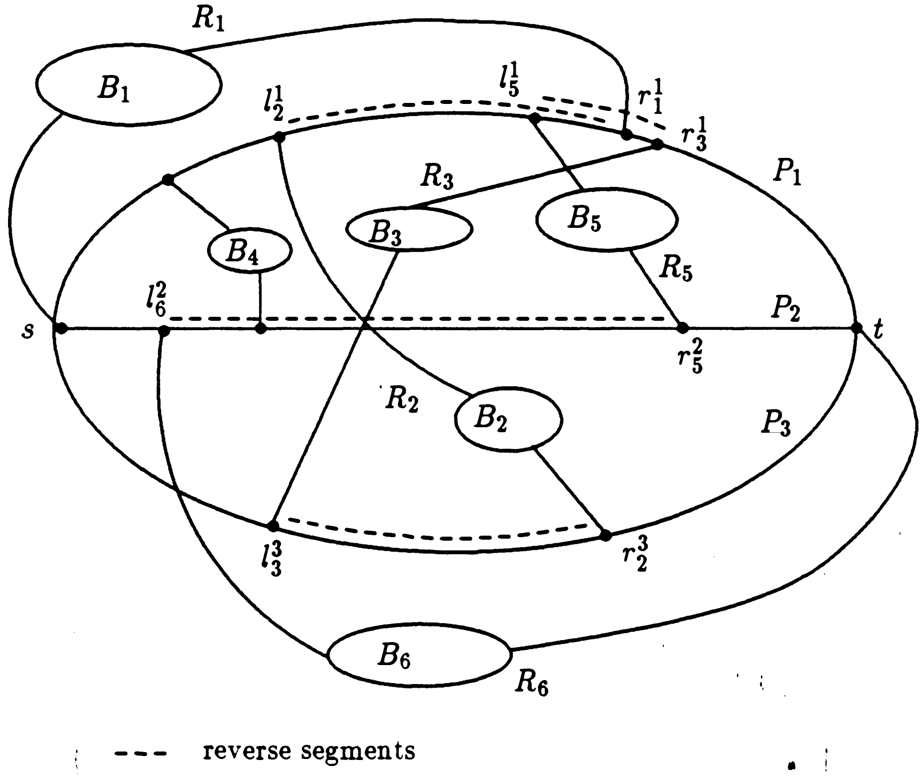


Figure 5.6: An example showing G and its bridges

Define a path P_k from s to t in G

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

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.)

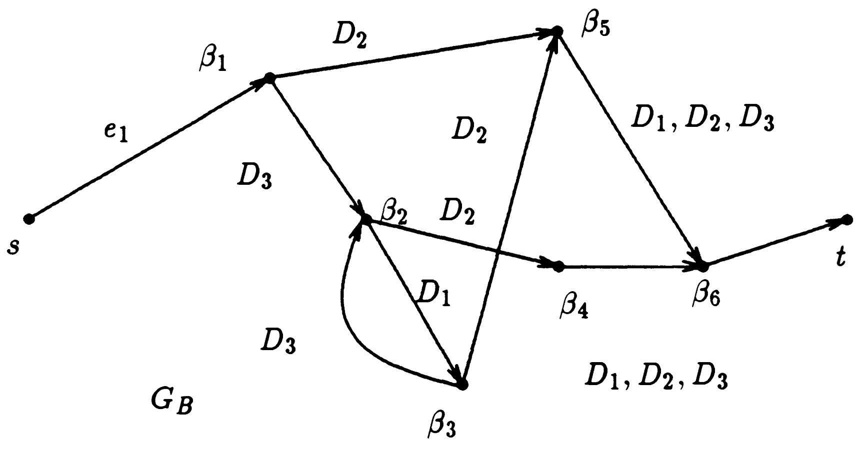


Figure 5.7: The path P_k corresponding to $P_B = s, \beta_1, \beta_2, \beta_3, \beta_5, \beta_6, t$ (in G_B).

The path P_k is not necessarily (edge) simple as shown in Fig. 5.6. (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. 5.8). 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.

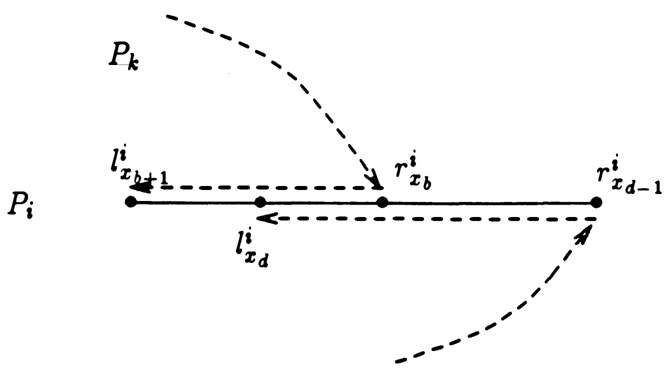


Figure 5.8: We prune the path so that P_k goes from B_{x_b} to B_{x_d}

Lemma 5.2.2 *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.

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. 5.9.)

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 $(w \rightarrow v)$ 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}).

□

This concludes the proof of the *if* direction of Theorem 5.2.1.

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.

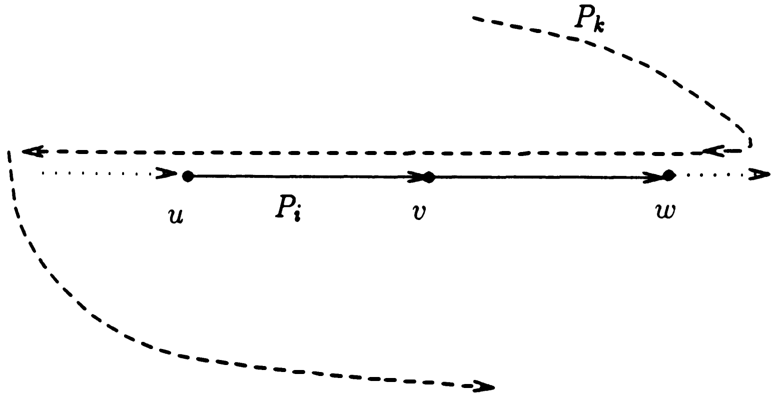
Consider the vertices of Q that are also on one of the paths P_j, $1 \leq j \leq k-1$, in the order of their appearance on Q. Observe that there must be two successive vertices x_1 and x_2 such that: (i) x_1 is on some P_j and $x_1 < w_j$, (ii) x_2 is on some $P_{j'}$ and $x_2 > w_{j'}$. Clearly, the 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 $l^j_i < w_j$ and $r^{j'}_i > w_{j'}$. This implies that β_i is reachable from s in G_B, contradicting the definition of $w_{j'}$. □

5.2.2 Implementation and Complexity:

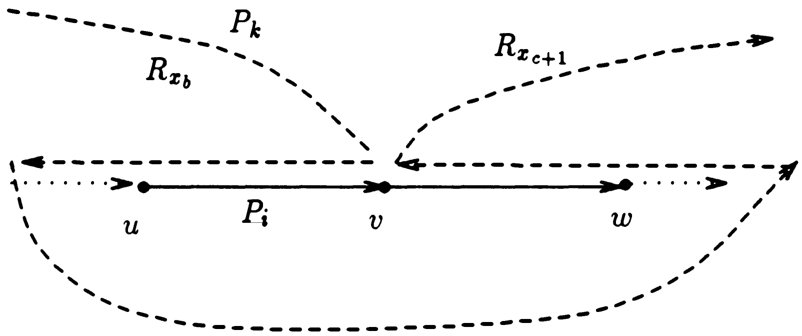
We show how to implement the algorithm in $O(\log n)$ time using $C(n, m)$ processors on a CRCW PRAM, for any fixed k. In the following discussion we will assume that G is represented by its adjacency list.

Step 1. This is the recursive step.

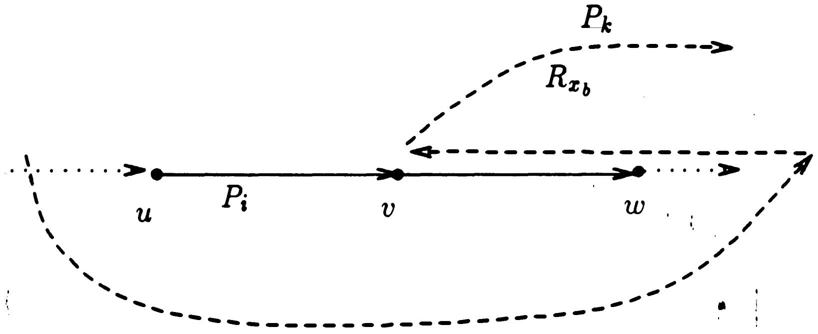
Step 2. The bridges of G relative to the subgraph $P_1 \cup P_2 \cup \ldots \cup P_{k-1}$, and the attachment vertices of each bridge are obtained by computing the spanning forest of the subgraph of G induced by the vertices in $V(G) - V(P_1 \cup P_2 \cup \ldots \cup P_{k-1})$. By adding the edges incident 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]).



(a) Case 1.



(b) Case 2.



(c) Case 3.

Figure 5.9: Figures to illustrate all three cases

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, $r^a_{z_a} \geq r^a_{z'}$, 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 $r^a_{z_a} \leq r^a_z$. 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. 5.10 shows a shortcutting step in F_1 (the subgraph of G_B induced by $D_1 \cup \{e_1\}$).

Lemma 5.2.3 *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

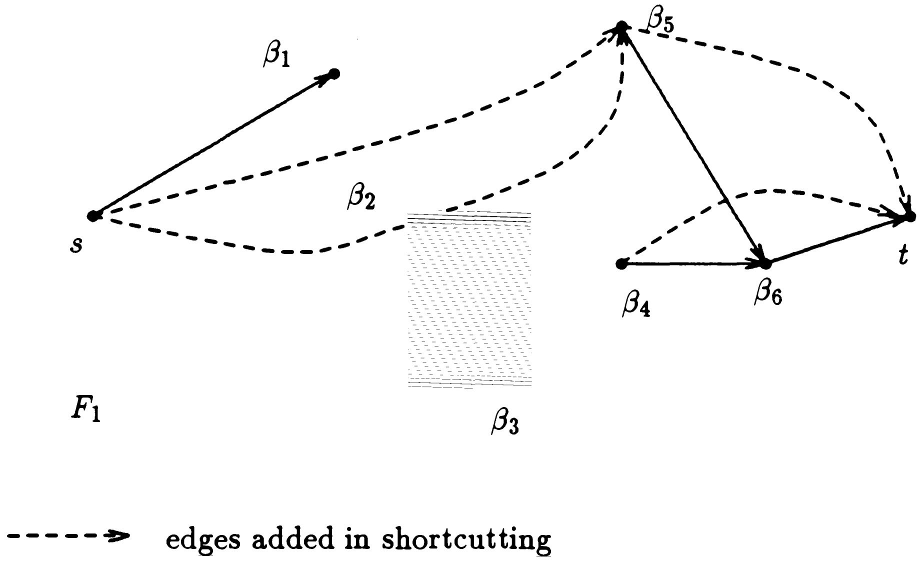


Figure 5.10: Shortcutting in F_1

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 5.2.3. 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. 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 $r^{i_c}_{x'_c} \geq r^{i_c}_{x_c}$, 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: $(\beta_y \rightarrow \beta_x)$ and $(\beta_z \rightarrow \beta_x)$ 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 $r^i_y < r^i_z$. 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^i_y and r^i_z 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 5.2.1 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 $r^i_{x_b}$ on P_i, define its successor $succ(r^i_{x_b})$ to be $l^i_{x_d}$, if (i) $d = \max\{z \mid l^i_{x_z} < r^i_{x_b}\}$, and (ii) there is no $c < b$ such that $l^i_{x_d} < r^i_{x_c}$. 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 $l^{i_{b-1}}_{x_b}$, $succ(l^{i_{b-1}}_{x_b}) = r^{i_b}_{x_b}$. 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 5.2.4 *The described algorithm finds k vertex-disjoint s-t paths in $O(k^2 \log n)$ time using $kC(n,m)$ processors.*

5.3 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 5.2. 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 $(\beta_i \to \beta_x)$ 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 $(\beta_i \to \beta_x) \in D_j$ 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 $r_z^j > r_x^j$.

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

Theorem 5.3.1 *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 5.2.1: 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.

Chapter 6

Connectivity Testing and Other Applications

In this chapter we outline various applications of the k disjoint s-t paths algorithm. The particular applications described are that of testing edge and vertex connectivities, and finding homeomorphs of fixed pattern graphs.

6.1 Connectivity and Disjoint Paths

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 6.1.1 (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.

6.2 Other Applications

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

6.2.1 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. This problem can also be thought of as finding a homeomorph of a K_3, where the mapping from the vertices of the triangle to the graph have been specified.

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 5.2, together with parallel logarithmic time connectivity [CV86], biconnectivity [TV85] and triconnectivity [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.)

For the triangle problem when the graph is planar, an elegant parallel algorithm was developed in [Khu89] to solve the problem. Incidentally, this algorithm also uses the concept of bridges.

6.2.2 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 presented in Chapter 3 and Chapter 4 for the two paths problem to yield an $O(\log^2 n)$ 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.

6.2.3 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 6.2.1 *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 6.2.2 *If a simple graph G with two or more vertices has no subgraph homeomorphic to K_4, then $m \leq 2n - 3$.*

Lemma 6.2.3 *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 6.2.4 *If a simple graph G with two or more vertices has no subgraph homeomorphic to $K_{2,3}$, then $m \leq 2n - 2$.*

Lemma 6.2.5 *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 6.2.2 and 6.2.3 imply a simple algorithm for testing whether a given graph G has a subgraph homeomorphic to K_4. Similarly, Lemmas 6.2.4 and 6.2.5 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 6.2.6 *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 6.2.3 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 6.2.5 in [Asa85]. Again, we assume that G has $2n$ edges and a triconnected component G_1 satisfying one of the conditions in Lemma 6.2.5. 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.

Chapter 7

Conclusions

The most important problem that is still left unsolved is that of testing for minor containment in NC. Of course, a parallel algorithm for the k disjoint paths problem would also yield such an algorithm. Here, we believe that a lot of work needs to be done to simplify the considerably complex algorithm developed by Robertson and Seymour. Their algorithm, as mentioned earlier, not only suffers from the problem of having very large constants in the running time but is also considerably complex. Work has continued in developing efficient parallel algorithms for the many sequential algorithms developed by Robertson and Seymour. Recently, [Lag90] has developed a fast parallel algorithm for decomposing a graph into its "tree decomposition" with constant treewidth using our parallel algorithms for k vertex-disjoint s-t paths.

We suspect that it is possible to obtain a simple algorithm for the the k paths problem when $k = 3$ and the graph is planar. It is easy to show that there are some "forbidden" embeddings (similar to the one in the case when $k = 2$, with some more cases). We suspect that with the assumption of higher connectivity (perhaps 4), one can simply test for the forbidden embeddings; if the graph satisfies all the "topological" requirements the paths will always exist.

We believe that the algorithm that finds a Kuratowski homeomorph in a non-planar graph can do with an improvement of a $\log n$ factor in the running time (the time to do the binary search). In fact, one should be able to adapt the planarity algorithm due to Ramachandran and Reif to find a Kuratowski homeomorph in the case when the graph is non-planar.

When the graphs are planar, we suspect that k vertex-disjoint s-t paths can be found in NC, for arbitrary k. Note that k edge-disjoint s-t paths can be found in NC, since it is known how to compute a max-flow in NC for planar graphs. However, this algorithm does not yield an algorithm for the vertex disjoint paths problem, since obtaining vertex disjoint paths requires introducing vertex capacities in the network.

Vertex capacities in planar graphs cannot be reduced to edge capacities simply, since the usual reduction does not maintain planarity in the graph [KN90].

Bibliography

[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. of 27th IEEE Symp. on Foundations 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. of 16th ICALP, LNCS* No. 372, pages 379–393. Springer-Verlag, July 1989.

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

[Hal43] D.W.Hall. A note on primitive skew curves. *Bull. Amer. Math. Soc.*, 49:935–937, 1943.

[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. of 25th IEEE Symp. on Foundations of Computer Science*, pages 137–147, October 1984.

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

[Kar75] R. M. Karp. On the complexity of combinatorial problems. *Networks* 5:45–68, 1975.

[KUW85] R. M. Karp, E. Upfal and A. Wigderson. Are search and decision problems computationally equivalent. In *Proc. of 17th ACM Symp. on Theory of Computing*, pages 464–475, May 1985.

[Khu89] S. Khuller. Parallel algorithms for the subgraph homeomorphism problem. In *Proc. of Workshop on Algorithms and Data Structures, LNCS* No. 382, pages 303–315, August 1989.

[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. of 30th IEEE Symp. on Foundations of Computer Science*, pages 300–305, October 1989.

[KN90] S. Khuller and J. Naor. Flow in planar graphs with vertex capacities. In *Proc. of Integer Programming and Combinatorial Optimization*, pages 367–383, May 1990.

[KS89] S. Khuller and B. Schieber. Efficient parallel algorithms for testing k-connectivity and finding disjoint s-t paths in graphs. In *Proc. of 30th IEEE Symp. on Foundations of Computer Science*, pages 288–293, October 1989.

[Lag90] J. Lagergren. Efficient parallel algorithms for tree-decomposition and related problems. In *Proc. of 31st IEEE Symp. on Foundations of Computer Science*, to appear, October 1990.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Proc. of 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. of 27th IEEE Symp. on Foundations 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. of 28th IEEE Symp. on Foundations 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.

[MRS88] R. Motwani, A. Raghunathan and H. Saran. Constructive results from graph minors: linkless embeddings. In *Proc. of 29th IEEE Symp. on Foundations of Computer Science*, pages 398–409, October 1988.

[PS78] Y. Perl and Y. Shiloach. Finding two disjoint paths between two pairs of vertices in a graph. *Journal of the Association for Computing Machinery*, 25:1–9, 1978.

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

[Ran87] A. G. Ranade. How to emulate shared memory. In *Proc. of 28th IEEE Symp. on Foundations of Computer Science*, pages 185–194, October 1987.

[RS85] N. Robertson and P. D. Seymour. Disjoint paths – a survey. *SIAM Journal of Algebraic and Discrete Methods*, 6 (2):300–305, 1985.

[RS86a] N. Robertson and P. D. Seymour. Graph Minors XIII: The disjoint paths problem. *manuscript*, 1986.

[RS86b] N. Robertson and P. D. Seymour. Graph Minors XVI: Wagner's Conjecture. *manuscript* (1986).

[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.

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

[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.

[Vaz89] V. V. Vazirani. *NC* Algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems. *Information and Computation*, 80, 2:152–164, 1989.

[Wat68] M. Watkins On the existence of certain disjoint arcs in graphs. *Duke Math. Journal*, 35:321–346, 1968.

[Whi33] H. Whitney. 2-isomorphic graphs. *American Journal of Math.*, 55:245–254, 1933.

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