February 1, 1996

# Fast Algorithms for $K_4$ Immersion Testing[*][†]

Heather D. Booth, Rajeev Govindan,
Michael A. Langston and Siddharthan Ramachandramurthi

Department of Computer Science
University of Tennessee
Knoxville, TN 37996–1301

*Abstract*

Many useful classes of graphs can in principle be recognized with finite batteries of obstruction tests. One of the most fundamental tests is to determine whether an arbitrary input graph contains $K_4$ in the immersion order. In this paper, we present for the first time a fast, practical algorithm to accomplish this task. We also extend our method so that, should an immersed $K_4$ be present, a $K_4$ model is isolated.

[*] This research has been supported in part by the National Science Foundation under grant CDA–9115428 and by the Office of Naval Research under contract N00014–90–J–1855.

[†] A preliminary version of a portion of this paper was presented at the Great Lakes Symposium on VLSI, held in Kalamazoo, Michigan, in February, 1992.

# Contents

# 1 Introduction

We restrict our attention to finite, undirected graphs. Multiple edges may be present, but loops are ignored. A pair of adjacent edges $uv$ and $vw$, with $u \neq v \neq w$, is *lifted* by deleting the edges $uv$ and $vw$, and adding the edge $uw$. A graph $H$ is *immersed* in a graph $G$ if and only if a graph isomorphic to $H$ can be obtained from $G$ by taking a subgraph and lifting pairs of edges.

The immersion order can be applied to a number of combinatorial problems. Consider, for example, the problem of deciding whether a graph satisfies a given width metric. The *cutwidth* of $G = (V, E)$ is the minimum, over all linear layouts of $V$, of the maximum number of edges from $E$ that must be cut if the layout is split between any two consecutive vertices. Although $\mathcal{NP}$-complete in general, cutwidth can, in principle, be decided in linear time for any fixed width using a finite but unknown list of immersion tests. Multidimensional generalizations of cutwidth, termed *congestion* problems, can likewise be solved in linear time if only one has the right collection of immersion tests available. These and other problems amenable to the immersion order arise during circuit fabrication, parallel computation, network design and many other processes.

The graphs required for the aforementioned tests are called *obstructions*. So, for example, when one knows all obstructions to cutwidth $k$, one knows a characterization for the family of graphs that have cutwidth $k$ or less. Given the right collection of obstructions, linear-time decidability is assured by bounding an input graph's treewidth [FL2], computing its tree decomposition [Bo], and applying dynamic programming to test each obstruction against the decomposition [RS]. We refer the reader to [FL1] for detailed information on this subject.

Unfortunately, little is known about immersion obstructions in general or about practical immersion tests in particular. Complete graphs are often obstructions. Testing for $K_1$ and $K_2$ are trivial. Detecting a $K_3$ is easy: $K_3$ is immersed in any graph of order three or more unless the graph is a tree with no pair of multiple edges incident on a common vertex.

The first really difficult test, and the one we devise here, is for $K_4$. Observe that $K_4$ is an obstruction for cutwidth three, because any arrangement of its vertices on a line will require a cut of four edges. Ours is the first practical linear-time algorithm known for this task.

If a graph contains a topological $K_4$, then it also contains an immersed $K_4$. Thus we consider only those graphs with no topological $K_4$. These are exactly the series-parallel graphs [Du]. But $K_4$ can be immersed in a series-parallel graph. As a simple example, consider the star graph with three rays, each ray with three edges, as shown in Figure 1. Clearly, multiple edges are critical, making immersion tests potentially more complicated than tests in the more-familiar minor and topological orders (see for example [LG]).

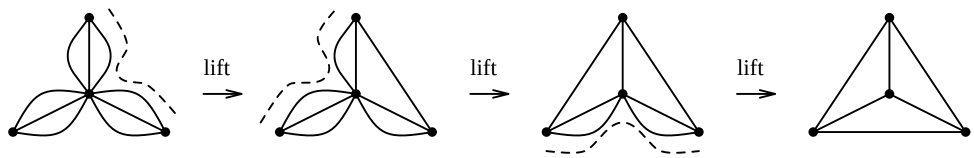


Figure 1: A series-parallel graph with an immersed $K_4$.

In the next section we state relevant definitions and derive a few useful technical lemmas. In Sections 3 and 4, we present algorithms for $K_4$ immersion testing and $K_4$ model finding, respectively. Although many of the explanatory details are tedious, especially the correctness proofs, the algorithms themselves are straightforward to implement. In a final section we discuss efficiency, applications and parallelization.

# 2 Preliminaries

We concentrate on edge-disjoint paths, which are relevant due to the following alternate characterization of immersion containment: $H = (V_H, E_H)$ is immersed in $G = (V_G, E_G)$ if and only if there exists an injection from $V_H$ to $V_G$ for which the images of adjacent elements of $V_H$ are connected in $G$ by edge-disjoint paths. Under such an injection, an image vertex is called a *corner* of $H$ in $G$; all image vertices and their associated paths are collectively called a *model* of $H$ in $G$. Our algorithms exploit the edge connectivity of the input graph.

## 2.1 Three-Edge Connectivity

A *cut point* of a connected graph $G$ is a vertex whose removal disconnects $G$. Two vertices are said to be *biconnected* if there are at least two vertex-disjoint paths between them. A *biconnected component* of $G$ is the subgraph induced by a maximal set of pairwise biconnected vertices.

A *cut edge* of $G$ is an edge whose removal disconnects $G$. A pair of edges, neither of which is a cut edge, is said to form a *cut edge pair* if removing both of them disconnects $G$. Two vertices are *three-edge-connected* if there are at least three edge-disjoint paths between them. $G$ is *three-edge-connected* if and only if it has no cut edges and no cut edge pairs.

A *three-edge-connected component* of $G = (V, E)$ is a graph $G' = (V', E')$ where $V' \subseteq V$ is a maximal set of vertices that are pairwise three-edge-connected in $G$. $E'$ contains all edges induced by $V'$ plus a (possibly empty) set of *virtual edges* defined as follows: for $\{u, v\} \subseteq V'$, a virtual edge $uv$ is added to $E'$ for each distinct $\{x, y\} \subseteq V - V'$ such that $ux$ and $vy$ form a cut edge pair in $G$. Note that, due to the possible presence of virtual edges, a three-edge-connected component will not necessarily be a subgraph.

**Lemma 1** *If $K_4$ is immersed in $G$, then $K_4$ is immersed in some three-edge-connected component of $G$.*

**Proof** Let $a$, $b$, $c$ and $d$ denote the corners of a $K_4$ model in $G$. These corners are joined (in $G$) by at least six edge-disjoint paths: $[ab]$, $[ac]$, $[ad]$, $[bc]$, $[bd]$ and $[cd]$. Thus $a$ and $b$ are connected by at least three edge-disjoint paths: $[ab]$, $[ac][cb]$ and $[ad][db]$. Maximality ensures that the three-edge-connected component containing $a$ also contains $b$ and, by symmetry, $c$ and $d$. Let $G_a$ denote this component. If $[ab]$ contains edges not in $G_a$, then $[ab]$ can be written as $[au]ux[xy]yv[vb]$, where $ux$ and $yv$ are a cut edge pair in $G$ and $uv$ is a virtual edge in $G_a$. Thus $a$ and $b$ are connected within $G_a$ by $[au]uv[vb]$, which is edge disjoint from the other five paths of the model. By symmetry, all pairs of corners are so connected within $G_a$. ∎

The proof of Lemma 1 can be generalized to any three-edge-connected graph immersed in another.

For our purposes, a multigraph is said to be *reduced* if all but four copies of any edge having multiplicity five or more are removed.

**Lemma 2** *If $K_4$ is immersed in $G$, then $K_4$ is immersed in the reduced graph of $G$.*

**Proof** Let $a$, $b$, $c$ and $d$ denote the corners of a $K_4$ model in $G$, and suppose five or more copies of the edge $uv$ are contained within its six edge-disjoint paths. Without loss of generality, assume these paths are simple. For some pair of corners, say $a$ and $b$, all five paths with an endpoint at $a$ or $b$ contain both $u$ and $v$. Either $u$ is a corner, or it can be made a corner by replacing $a$ with $u$ (deleting the three subpaths of the form $[au]$). Similarly, either $v$ is a corner or $b$ can be replaced with $v$. $G$ therefore contains a $K_4$ model with corners $u$, $v$, $w$ and $x$, where $\{w, x\} \subset \{a, b, c, d\}$. At most one of $[uw]$, $[vw]$ must contain $uv$; at most one of $[ux]$, $[vx]$ must contain $uv$. Thus, of the six edge-disjoint paths of this model, at least two need not contain $uv$, and all but four copies of $uv$ can be eliminated. This construction is iterated until a model is obtained whose edges each have at most four copies. Edges not in this model are now removed until $G$ is reduced. ∎

In the sequel, we assume that all graphs are reduced.

## 2.2 Series-Parallel Graphs

Series-parallel graphs have been widely studied, and are characterizable in several ways. As mentioned in Section 1, one such characterization relies on the absence of a topological $K_4$. Topological containment can be defined as a restricted form of immersion containment, with lifting permitted only at vertices of degree two. Alternately, topological containment can be viewed as an injection, but with vertex-disjoint rather than edge-disjoint paths.

**Lemma 3** *Each three-edge-connected component of a series-parallel graph is series-parallel.*

**Proof** The proof is straightforward, by noting that virtual edges introduce no additional vertex-disjoint paths. ∎

Another useful characterization is much older, and based on graphs that are said to be *two-terminal series-parallel* (henceforth 2TSP). A 2TSP graph is defined in terms of base graphs and two types of composition operators. A base graph is a copy of $K_2$, with vertices (terminals) labeled "source" and "sink." A series operator combines two graphs by identifying one's source with the other's sink. A parallel operator combines two graphs by identifying source with source and sink with sink. Hence the characterization: a graph is series-parallel if and only if its biconnected components are two-terminal series-parallel.

This characterization is often attractive because it prompts a natural "decomposition tree" $T$ whose labels indicate how a 2TSP can be broken back down into base graphs and operators. If a 2TSP graph is merely a base graph $e$, $T$ is a single vertex with label $e$. Otherwise, $T$ is formed from the decomposition trees, $T_1$ and $T_2$, of the pair of 2TSP graphs used in the composition. The roots of $T_1$ and $T_2$ are joined to the root of $T$, which is labeled $S$ in the case of a series composition and $P$ in the case of a parallel composition.

We conclude this section by noting from [Di] that if a simple graph $H$ is series-parallel, then $|E_H| \leq 2|V_H| - 3$. From this bound and Lemma 2, we know that all graphs of interest have at most a linear number of edges.

# 3 Testing for $K_4$

Let $G$ denote an arbitrary input graph with $n$ vertices and $m$ distinct edges. Without loss of generality, we assume $G$ has already been reduced and is input as a simple graph with integer weights indicating edge multiplicities.

Our method to test for the presence of an immersed $K_4$ proceeds in three steps. Algorithm decompose is first invoked to determine whether $G$ is series-parallel. If $G$ is series-parallel, then algorithm components is used to break $G$ into three-edge-connected components. Finally, algorithm test is employed to search each three-edge-connected component separately for an immersed $K_4$.

## 3.1 Algorithm decompose

Algorithm decompose is modeled on the method of [He]. It determines whether $G$ is series-parallel and, if so, computes a decomposition tree for each biconnected component. (Recall that a graph $G$ is series-parallel if and only if every biconnected component is 2TSP.)

To accomplish this, decompose makes use of the fact that for any edge $st$ in a biconnected graph $B$ with $p$ vertices, the vertices of $B$ may be numbered from 1 to $p$ so that vertex $s$ receives number 1, vertex $t$ receives number $p$, and every vertex except $s$ and $t$ is adjacent to both a higher-numbered vertex and a lower-numbered vertex [LEC]. Such a numbering is called an *st-numbering* for $B$.

```
algorithm decompose(G)
input: a multigraph G
output: a series-parallel decomposition tree for each biconnected component of G if G is
            series-parallel, NO otherwise
begin
    find all the biconnected components of G; call them B_1, ..., B_k
    for i = 1 to k do
        begin
            choose a pair of adjacent vertices to be the source s and sink t in B_i
            find an s,t-numbering of B_i
            let B̄_i be the directed graph obtained by orienting each edge in B_i from
                the end point with the lower s,t-number to the one with the higher number
            if B̄_i is a directed 2TSP graph
                then compute a series-parallel decomposition tree T_i for B_i
                else output NO and stop
        end
    for i = 1 to k do
        output T_i
end
```

The correctness of decompose is based on the observation that any $s,t$-numbering will suffice [He]. Efficient methods for finding biconnected components and computing $s,t$-numberings are known from [Ta,ET]. Techniques for determining whether directed graphs are 2TSP and finding decomposition trees can be found in [VTL]. All these algorithms are linear in $n$ and the number of edges; thus decompose runs in $O(n)$ time.

## 3.2 Algorithm components

Algorithm components finds the three-edge-connected components of a series-parallel multigraph in linear time. The input to components is a series-parallel graph and a series-parallel decomposition tree for each of its biconnected components. The output is its set of three-edge-connected components (including virtual edges).

We proceed by first removing all cut edges. These are easily found since each cut edge is contained in a biconnected component consisting only of that edge. Notice that each cut edge pair must be contained within some biconnected component. Thus it suffices to give an algorithm for computing the three-edge-connected components of a biconnected 2TSP graph.

Let $G$ be such a 2TSP graph with source $s$ and sink $t$. Let $e, f$ be a cut edge pair of $G$. Let $G_1$ and $G_2$ be the graphs left when $e$ and $f$ are deleted from $G$. We call this cut edge pair *s,t-non-separating* if $s$ and $t$ are both in $G_1$ or both in $G_2$. Otherwise we call the pair *s,t-separating*. We say an $s,t$-non-separating pair is *special* if its deletion, followed by the addition of virtual edges, results in two graphs such that one contains $s$ and $t$ and the other is three-edge-connected.

These definitions are illustrated in Figure 2. In this figure, edges $ab$ and $cd$ are a special pair of graph $G$. Deleting them and adding virtual edges $ad$ and $bc$ gives $G_1$, which contains both $s$ and $t$, and $G_2$, which is three-edge-connected. Edge $st$ and the virtual edge $ad$ together form the $s,t$-separating pair of $G_1$. $G_{11}$, $G_{12}$ and $G_2$ are the three-edge-connected components of $G$.

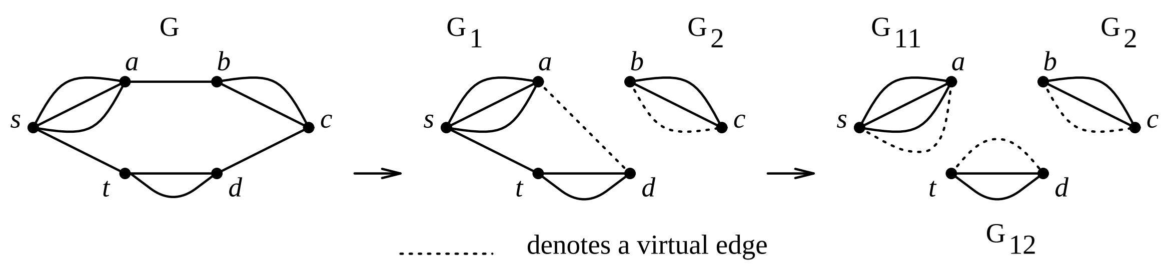


Figure 2: A two-terminal series-parallel graph with cut edge pairs.

For our purposes, the decomposition tree $T$ for a 2TSP graph $G$ must be *ordered*. That is, if $x$ is a tree node representing a graph formed by composing $G_1$ and $G_2$ in series such that the sink of $G_1$ is identified with the source of $G_2$, then the left child of $x$ must be the root of a decomposition tree for $G_1$ and the right child of $x$ must be the root of a decomposition tree for $G_2$. Thus the order among children of a series node is fixed. The children of a parallel node can be in any order. Additionally, we assume that an edge $uv$ stored at a leaf of a decomposition tree is represented by the ordered pair $(u, v)$, where $u$ has a smaller number than $v$ in the $s, t$-numbering used in decompose.

Our algorithm proceeds in two phases. In the first phase special pairs are found and deleted (and appropriate virtual edges are added) until no more are left. This leaves a collection of (isolated vertices and) 2TSP graphs, one of which contains both $s$ and $t$. We will call this graph $G_{s,t}$. All other graphs in the collection are three-edge-connected. Graph $G_{s,t}$ may contain at most one cut edge pair, since otherwise there would also be an $s, t$-non-separating pair. In the second phase the last remaining cut edge pair, if it exists, is found, removed, and virtual edges are added.

In order to find any of these cut edge pairs we use the *compressed* decomposition tree for the graph. A compressed decomposition tree is formed from a regular decomposition tree merely by identifying all pairs of adjacent nodes that are of the same type.

Let $G$ be a biconnected 2TSP graph with compressed decomposition tree $T$. Let $\hat{e}$ denote the leaf node in $T$ representing edge $e$ in $G$. Since $G$ is biconnected, the root of $T$ will be a P-node. Our algorithm is based on the following claims, whose correctness we will address later (see for example Lemmas 6 and 8).

**Claim 1** Edges $e$ and $f$ are an $s, t$-non-separating pair for $G$ if and only if $\hat{e}$ and $\hat{f}$ are siblings whose parent $x$ is an S-node. Furthermore, $e$ and $f$ are a special pair if and only if for every node $y$ that is a child of $x$ occurring between $\hat{e}$ and $\hat{f}$ in $T$, $y$ is not a leaf and the graph represented by the subtree of $y$ does not contain an $s, t$-non-separating pair.

**Claim 2** Edges $e$ and $f$ are an $s, t$-separating pair if and only if the root of $T$ has exactly two children and each of $\hat{e}, \hat{f}$ is either a child of the root or a child of a distinct S-node that is a child of the root.

Special pairs can be found by processing $T$ in a bottom-up fashion. When a special pair $e, f$ is removed, virtual edges are added and $T$ is modified to represent the graph $G'_{s,t}$, which is the graph containing $s$ and $t$ that is left after removing $e$ and $f$ from $G$ (the other graph left is a 3-edge-connected component).

The $s, t$-separating pair is easy to detect using Claim 2.

If $e$ and $f$ are an $s,t$-non-separating pair such that the vertex pair $(u,v)$ is stored with $\hat{e}$ and the pair $(y,z)$ is stored with $\hat{f}$, then the virtual edges to be added when $e$ and $f$ are removed are $uz$ and $vy$. See Lemma 7. We need to construct the compressed decomposition tree $T'$ representing $G'_{s,t}$. Let $x$ be the parent of $\hat{e}$ and $\hat{f}$. Let $g$ be the virtual edge $uz$. If $\hat{e}$ is the leftmost child of $x$ and $\hat{f}$ is the rightmost, then replace $x$ by $\hat{g}$; otherwise, replace $\hat{e}$ and $\hat{f}$ and all children of $x$ in between by $\hat{g}$. See Corollary 1 to Lemma 7.

Pseudo code for components is presented below. In a compressed tree, each internal node will have at least two children, stored in a linked list called *child list*. Stored long with each tree node is its type (P, S, or leaf), a pointer to its child list and, if it is a leaf node, and an ordered pair giving the endpoints of its associated edge.

The following functions are also used:

left_child($x$): for $x$ a tree node, if $x$ is not a leaf, this returns the leftmost child in $x$'s child list; otherwise, it returns the value NULL.

right_child($x$): for $x$ a tree node, if $x$ is not a leaf, this returns the rightmost child in $x$'s child list; otherwise, it returns the value NULL.

next_sibling($q$): for $q$ a non-root tree node, this returns the child following $q$ in the child list of the parent of $q$ or NULL if no such child exists.

left_leaf($x$): for $x$ a tree node, if $x$ is not a leaf, this returns the leftmost node in $x$'s child list that is a leaf or NULL if no such node exists.

**algorithm** components ($T$)
input: a binary series-parallel decomposition tree $T$ of a biconnected multigraph $G$
output: the three-edge-connected components of $G$
**begin**
    let $r$ be the root of $T$
    compress($r$)
    remove_non_sep($r$)
    remove_sep($r$)
**end**

**algorithm** compress($x$)
input: a node $x$ in a binary series-parallel decomposition tree $T$

```
output: the compressed form of the sub-tree rooted at x
begin
    if x is a leaf node
        then return
    compress(left_child(x))
    compress(right_child(x))
    if x and left_child(x) are of the same type
        then in the child list of x, replace left_child(x) by the child list of left_child(x)
    if x and right_child(x) are of the same type
        then in the child list of x, replace right_child(x) by the child list of right_child(x)
end
```

**algorithm** remove_non_sep($q$)
input: a node $q$ in a series-parallel decomposition tree $T$ of a multigraph $G$
output: the graph $G$, after deletion of all $s, t$-non-separating pairs that are contained in the sub-tree of $T$ rooted at $q$, and addition of virtual edges

```
begin
    ch = left_child(q)
    while ch is not NULL
        begin
            if ch is not a leaf node
                then remove_non_sep(ch)
            ch = next_sibling(ch)
        end
    if q is an S-node
        then while q has two children that are leaves
            begin
                let leaf1 and leaf2 be the first two leaf-node children of q
                let (u, v) be the ordered edge associated with leaf1
                let (w, x) be the ordered edge associated with leaf2
                delete uv and wx from the graph
                add edges ux and vw to the graph
                create tree node new representing the ordered edge (u, x)
                if q has more than 2 children
                    then replace all children of q between leaf1 and leaf2 (inclusive)
                        by new
                    else replace q by new
            end
    end
end
```

**algorithm** remove_sep($root$)
input: the root of a series-parallel decomposition tree $T$ of a multigraph $G$ without any $s, t$-non-separating pairs

```
output: the graph G after deletion of the s,t-separating pair, if present, and addition
            of a virtual edge
begin
    if root has exactly two children
        then begin
            let c1 and c2 be the children of root
            if c1 is not leaf node
                then set c1 = left_leaf(c1)
            if c2 is not leaf node
                then set c2 = left_leaf(c2)
            if c1 and c2 are both non-NULL
                then begin
                    let (u, v) be the ordered edge associated with c1
                    let (w, x) be the ordered edge associated with c2
                    delete edges uv and wx from the graph
                    add edges uw and vx to the graph
                end
        end
end
```

**Lemma 4** *Algorithm* components *runs in* $O(m+n)$ *time on a graph with* $m$ *edges and* $n$ *vertices.*

**Proof** The algorithm takes time proportional to the size of the binary decomposition tree, which is $O(m+n)$. ∎

Thus, in our setting, components takes $O(n)$ time. We note for completeness that a more complex linear-time approach may be viable [Ra], by modifying the ear decomposition techniques used to decide vertex connectivity in [FRT].

### 3.3 The Correctness of components

Neither the components driver nor algorithm compress require discussion.

Consider algorithm remove_non_sep. Note first that remove_non_sep cannot inadvertently remove an $s, t$-separating pair, because the edge $st$ must be a child of the root (which is a P-node), and remove_non_sep eliminates only edges that are children of S-nodes.

In order to find and remove all $s, t$-non-separating pairs, remove_non_sep exploits these facts:

- If a 2TSP graph has an $s,t$-non-separating pair, then it has a special pair.
- If a biconnected 2TSP graph has no $s,t$-non-separating pairs, then it is either three-edge-connected or it has one $s,t$-separating pair.
- If a special pair is removed from a biconnected 2TSP graph, then the resulting subgraph containing $s$ and $t$ will be biconnected when augmented with a virtual edge.

To proceed, we classify edges and pairs of edges in a 2TSP graph as follows. A single edge is called either a cut edge or a non-cut edge. A pair of edges can be: a pair of cut edges, an $s,t$-separating pair, an $s,t$-non-separating pair, or a *non-cut pair*.

Let $G_1$ and $G_2$ be 2TSP graphs such that $G_s$ is the graph formed by composing them in series and $G_p$ is the graph formed by composing them in parallel. Suppose $e$ is an edge in $G_1$ and $f$ is an edge in $G_2$. Table 1 shows the relation between the class of edge $e$ in $G_1$, edge $f$ in $G_2$ and the pair $e, f$ in $G_s$ and $G_p$. For example, if edges $e$ and $f$ are cut edges in $G_1$ and $G_2$ respectively, then $e$ and $f$ must be an $s,t$-separating pair in $G_p$.

| **class of edge $e$ in $G_1$** | **class of edge $f$ in $G_2$** | **class of $e$ and $f$ in $G_s$** | **in $G_p$** |
|---|---|---|---|
| non-cut edge | non-cut edge | non-cut pair | non-cut pair |
| non-cut edge | cut edge | $f$ a cut edge | non-cut pair |
| cut edge | non-cut edge | $e$ a cut edge | non-cut pair |
| cut edge | cut edge | cut edges | $s,t$-separating |

Table 1

Now suppose edges $e$ and $f$ are both in the 2TSP graph $G_1$, and $G_2$ is any other 2TSP graph. Graphs $G_s$ and $G_p$ are as defined above. Table 2 relates the class of $e$ and $f$ in $G_1$ to their class in $G_s$ and $G_p$.

| class of edges $e$ and $f$ | | |
|---|---|---|
| **in $G_1$** | **in $G_s$** | **in $G_p$** |
| cut edges | cut edges | $s,t$-non-separating |
| $s,t$-non-separating | $s,t$-non-separating | $s,t$-non-separating |
| $s,t$-separating | $s,t$-separating | non-cut pair |
| non-cut pair | non-cut pair | non-cut pair |

Table 2

Let $z$ be a non-leaf node in decomposition tree $T$ and let $T_z$ denote the subtree of $T$ rooted at $z$. The 2TSP graph $H$ that has $T_z$ as a decomposition tree is a *constituent graph* for $G$ with respect to $T$. If $e$ and $f$ are edges in $G$ then the *least constituent graph containing $e$ and $f$* is the smallest constituent graph of $G$ that contains both $e$ and $f$. This graph has as a decomposition tree $T_z$ where $z$ is the least common ancestor of $\hat{e}$ and $\hat{f}$ in $T$.

Let $G$ be a 2TSP graph containing edges $e$ and $f$ and let $H$ be the least constituent graph of $G$ that contains $e$ and $f$. Table 3 gives the relation between the class of an edge in a 2TSP graph and its class in a constituent of that graph.

| class of edges $e$ and $f$ | |
|---|---|
| **in $H$** | **in $G$** |
| cut edges | cut edges or $s,t$-non-separating |
| $s,t$-non-separating | $s,t$-non-separating |
| $s,t$-separating | $s,t$-separating or non-cut pair |
| non-cut pair | non-cut pair |

Table 3

The following lemmas are used to justify the correctness of the procedure for finding special pairs, removing special pairs, updating the decomposition tree for the connected component containing $s$ and $t$, and adding virtual edges.

**Lemma 5** *Let $G$ be a 2TSP graph, and let $e$ and $f$ be an $s,t$-non-separating pair for $G$. If $H$ is the least constituent graph of $G$ containing $e$ and $f$, then $e$ and $f$ are cut edges in $H$.*

**Proof** By the first two lines of Table 3, either $e$ and $f$ are cut edges in $H$, as claimed, or they form an $s,t$-non-separating pair. Since $H$ is a least constituent graph, $H$ must be formed by composing two 2TSP graphs $H_1$ and $H_2$ such that $H_1$ contains $e$ and $H_2$ contains $f$. According to Table 1, $e$ and $f$ cannot be an $s,t$-non-separating pair in $H$. Therefore they must be cut edges for $H$, as claimed. ∎

The following lemma is crucial. Its proof uses the following fact: if $G$ is a 2TSP graph and $T$ is a compressed decomposition tree for $G$, then edge $e$ is a cut edge of $G$ if and only if the root of $T$ is an S-node and $\hat{e}$ is a child of the root.

**Lemma 6** *If $e$ and $f$ are edges in a biconnected 2TSP graph $G$ with decomposition tree $T$, then $e$ and $f$ are an $s,t$-non-separating pair if and only if $\hat{e}$ and $\hat{f}$ are siblings whose parent is an S-node.*

**Proof** Let $z$ be the least common ancestor of $\hat{e}$ and $\hat{f}$ in $T$. Let $T_z$ be the subtree of $T$ rooted at $z$ and let $H$ be the 2TSP graph having $T_z$ as a decomposition tree. Note that $H$ is the least constituent of $G$ that contains $e$ and $f$.

Suppose $e$ and $f$ are $s,t$-non-separating. By Lemma 5, $e$ and $f$ are cut edges for $H$. Thus $\hat{e}$ and $\hat{f}$ are children of $z$ and $z$ is an S-node.

Now suppose that $\hat{e}$ and $\hat{f}$ are siblings whose parent is an S-node. This implies that $e$ and $f$ are cut edges for $H$. Then, by Table 3, $e$ and $f$ must be either cut edges or an $s,t$-non-separating pair in $G$. Since $G$ is biconnected, $e$ and $f$ must in fact be an $s,t$-non-separating pair. ∎

In what follows, we say that node $x$ in tree $T$ occurs "between" nodes $y$ and $z$ if $x$ occurs between $y$ and $z$ in the preorder traversal of $T$. Let $H_x$ denote the graph having $T_x$ as a decomposition tree.

**Lemma 7** *Let $G$ be a biconnected 2TSP graph and let $T$ be a compressed decomposition tree for $G$. In $G$, let $e = uv$ and $f = wx$ be an $s,t$-non-separating pair whose removal yields a graph $G_1$ containing $s$ and $t$, and another graph $G_2$. Let $(u,v)$ be the pair stored with $\hat{e}$ and $(w,x)$ be the pair stored with $\hat{f}$. Then the edges in $G_2$ are $\{g|\hat{g}$ occurs between $\hat{e}$ and $\hat{f}$ in*

*$T\}$, and the vertices in $G_2$ are the endpoints of these edges plus $\{v, w\}$.*

**Proof** Since $e$ and $f$ are $s,t$-non-separating, by Lemma 6, $\hat{e}$ and $\hat{f}$ are siblings whose parent $z$ is an S-node. Without loss of generality, assume $\hat{e}$ occurs before $\hat{f}$ in $T$. Since $G$ is biconnected, $z$ has a P-node parent which we denote by $y$.

Removal of $e$ and $f$ from $H_z$ leaves three graphs $H_1$, $H_2$, and $H_3$ such that: $H_1$ contains all edges represented by nodes occurring before $\hat{e}$ in $T_z$, their associated vertices, and vertex $u$; $H_2$ contains all edges represented by nodes occurring between $\hat{e}$ and $\hat{f}$ and associated vertices plus $\{v, w\}$; and $H_3$ contains all edges represented by nodes occurring after $\hat{f}$ in $T_z$ and associated vertices plus $x$. The source and sink of $H_z$ are in $H_1$ and $H_3$ respectively.

In $H_y$, edges $e$ and $f$ are an $s,t$-non-separating pair whose removal leaves $H_2$ and another graph containing $H_1$, $H_3$, and the portion of $H_y$ not in $H_z$. The source and sink of $H_y$ are the source and sink of $H_z$ and are not in $H_2$. Thus the claim holds for $H_y$.

Any graph formed by composing two 2TSP graphs, one of which has $H_y$ as a constituent, still has the claimed property because the paths in the new graph that are not in $H_y$ can only connect vertices not in $H_2$. Since no new paths are added from vertices in $H_2$ to vertices not in $H_2$, it is still the case that removal of $e$ and $f$ will separate the vertices in $H_2$ from the rest of the graph. Thus the claim also holds for any graph having $H_y$ as a constituent. ∎

**Corollary 1** *Let $G, T, e$, and $f$ be as defined in Lemma 7. Let $G_1{}'$ be the graph consisting of $G_1$ plus virtual edge $ux$ and let $G_2{}'$ be the graph consisting of $G_2$ plus virtual edge $vw$. Let $z$ be the parent of $\hat{e}$ and $\hat{f}$ in $T$ and let $r_1, \ldots, r_k$ be the children of $z$ in order from left to right such that $r_i = \hat{e}$ and $r_j = \hat{f}$. Let $\hat{g}$ be a tree node representing $g = ux$; the ordered pair stored with $\hat{g}$ is $(u, x)$. Let $\hat{h}$ be a tree node representing $h = vw$; the ordered pair stored with $\hat{h}$ is $(v, w)$.*

*A decomposition tree for $G_1{}'$ is formed by replacing $r_i, \ldots, r_j$ by node $\hat{g}$ if $i \neq 1$ or $j \neq k$ and replacing $T_z$ by $\hat{g}$ otherwise.*

*A decomposition tree for $G_2{}'$ is one of the following:*

*(a) empty, if $j = i + 1$;*

*(b) a P-node with children $\hat{h}$ and $r_{i+1}$, if $j = i + 2$;*

*(c) a P-node with two children $\hat{h}$ and an S-node, which in turn has children $r_{i+1}, \ldots, r_{j-1}$,*

*otherwise.*

**Proof** We know by Lemma 7 that ${G_1}'$ is formed by replacing the portion of $G$ represented by nodes in $T$ between $\hat{e}$ and $\hat{f}$ by a single edge $ux$, so the decomposition tree for $G'$ is as claimed. We also know that ${G_2}'$ consists of the edges represented by nodes in $T$ strictly between $\hat{e}$ and $\hat{f}$, their associated vertices and vertices $v$ and $w$, with the edge $vw$ composed in parallel. Since the nodes between $\hat{e}$ and $\hat{f}$ are children of an S-node, the decomposition tree for ${G_2}'$ is as claimed. ∎

**Lemma 8** *Let $G$ be a biconnected 2TSP graph and let $T$ be a compressed decomposition tree for $G$. A pair of $s,t$-non-separating edges, $e$, $f$, is a special pair for $G$ if and only if for every sibling $y$ of $\hat{e}$ and $\hat{f}$ in $T$ that occurs between $\hat{e}$ and $\hat{f}$, $y$ is not a leaf and $T_y$ does not represent a graph containing an $s,t$-non-separating pair.*

**Proof** Since $e$ and $f$ are $s,t$-non-separating, removal of $e$ and $f$ yields two graphs $G_1$ and $G_2$ such that $G_1$ contains $s$ and $t$. Let $G'$ be the graph $G_2$ plus the virtual edge. Edges $e$ and $f$ are special if and only if $G'$ is three-edge-connected. Let $T'$ be the decomposition tree for $G'$ as described in Corollary 1. Let $z$ be the parent of $\hat{e}$ and $\hat{f}$ in $T$.

Suppose $e$ and $f$ are special. We employ proof by contradiction and assume there exists a child $y$ of $z$ between $\hat{e}$ and $\hat{f}$ such that $y$ is a leaf or $T_y$ represents a graph containing an $s,t$-non-separating pair. Then $G'$ must be three-edge-connected, which implies $T'$ has no cut edges or cut edge pairs. If $y$ is a leaf then, by Corollary 1, $T'$ consists of a P-node with two children. One of them is a leaf (representing the virtual edge) and the other is either $\hat{y}$ or an S-node having $\hat{y}$ as a child. In either case the structure of $T'$ requires that the virtual edge and $y$ form an $s,t$-separating pair for $G'$, a contradiction. If, on the other hand, $y$ is non-leaf node whose subtree $T_y$ represents a graph having an $s,t$-non-separating pair, then $T_y$ contains an S-node with two leaves as children. These nodes also represent an $s,t$-non-separating pair for $G'$, again a contradiction.

Now suppose $\hat{e}$ and $\hat{f}$ satisfy the conditions of the lemma, but that $e$ and $f$ are not special. Then $G'$ must have a cut edge or a cut edge pair, and arguments analogous to those above yield a contradiction. ∎

Therefore, Lemmas 5 through 8 demonstrate that the algorithm remove_non_sep correctly finds special pairs, adds virtual edges, and updates the decomposition tree to represent the graph left after the edges are removed.

We suppress the analysis of remove_sep, which at this point is relatively straightforward.

## 3.4 Algorithm test

Algorithm test is the heart of our method. The input to test is a three-edge-connected series-parallel multigraph. In such a graph, suppose $v$ is a vertex with exactly two neighbors, $u$ and $w$, and suppose there is only one copy of the edge $vw$. (Thus there are at least two copies of $uv$ by three-edge-connectivity.) We say that $v$ is *pruned* if the multiplicity of $uv$ is set to two. Similarly, we say a graph is pruned if each vertex fitting the profile of $v$ is pruned.

```
algorithm test(G)
input: a three-edge-connected series-parallel multigraph G
output: YES, if G contains an immersed K4, NO otherwise
begin
    for each vertex v in G with exactly one neighbor
        delete all but three copies of edges incident on v
    if any cut point in G has degree seven or more
        then output YES and halt
    for each biconnected component B with four or more vertices
        for each vertex v in B
            prune v if possible
        if there is a vertex in B with degree five or more
            then output YES and halt
    output NO and halt
end
```

## 3.5 The Correctness of test

The correctness of test relies on a number of lemmas, which follow. Before proceeding, we make a few useful observations.

**Observation 1** *If $H'$ is immersed in $H$, and if $M'$ is a $K_4$ model in $H'$, then in $H$ there is a $K_4$ model $M$ with the same corners as $M'$.*

Observation 1 follows from noting that edges in $H'$ map to edge-disjoint paths in $H$, and that in $H$ a suitable $K_4$ model can be found merely by replacing the edges of $M'$ with their image paths in $H$.

Observation 2 is a well-known property of series-parallel graphs.

**Observation 2** *Every biconnected series-parallel multigraph with four or more vertices contains at least two non-adjacent vertices with exactly two neighbors.*

Suppose $v$ is a vertex with exactly two neighbors, $x$ and $y$. We say that $v$ is *shorted* if we lift all pairs of edges $vx$ and $vy$ and delete any remaining edges incident on $v$ along with $v$ itself.

**Observation 3** *Shorting preserves biconnectivity, three-edge-connectivity and series-parallelness.*

Observation 3 holds because shorting a vertex does not change the number of vertex-disjoint or edge-disjoint paths between any pair of remaining vertices.

**Observation 4** *A biconnected component of a three-edge-connected graph is three-edge-connected.*

Observation 4 follows from noting that edge-disjoint paths may as well be made simple and that, whenever a pair of vertices lies in the same biconnected component, all vertices along simple paths connecting them in the original graph must also lie in this component.

**Lemma 9** *Let $G$ denote a graph in which a vertex, $v$, has exactly one neighbor, $w$. Let $G'$ be obtained from $G$ by deleting all but three copies of the edge $vw$. Then $K_4$ is immersed in $G$ if and only if $K_4$ is immersed in $G'$.*

**Proof** If $K_4$ is immersed in $G$, then $G$ contains a $K_4$ model whose edge images are simple paths. Since $v$ cannot be an intermediate vertex in a simple path, at most three copies of the edge $vw$ are needed. Thus $K_4$ is also immersed in $G'$. If $K_4$ is not immersed in $G$, then neither is it immersed in $G'$ since $G'$ is a subgraph of $G$. ∎

**Lemma 10** *Let $G$ be three-edge-connected, with non-cut point vertices $u$ and $v$. Let $w$ denote* *any other vertex in $G$. Then there exist three mutually edge-disjoint paths, each beginning with $w$ and ending with either $u$ or $v$, such that at most two of these paths contain $u$, and at most two contain $v$.*

**Proof** The paths we seek to identify are illustrated in Figure 3, where the dashed lines denote edge-disjoint paths that do not contain $u$ or $v$ as an intermediate vertex. Consider three mutually edge-disjoint paths $P_1$, $P_2$ and $P_3$, each from $w$ to $\{u, v\}$. These paths exist because $G$ is three-edge-connected. Assume all three contain, say, $u$. Hence all three may as well be simple and end at $u$. Consider now some path $P$ between $w$ and $v$ that does not contain $u$ (such a path exists since $u$ is not a cut point). $P$ may contain vertices and edges in $P_1$, $P_2$ and $P_3$. Let $y$ be the last vertex in $P$ (counting from $w$) that is also in $P_1$ or $P_2$ or $P_3$. Without loss of generality, assume $y$ is in $P_1$. We can construct a path $P'$ from $w$ to $v$, by taking $P_1$ until we reach $y$, and using $P$ from there on. Thus $P'$, $P_2$ and $P_3$ are the desired edge-disjoint paths, with $P'$ not containing $u$. ∎

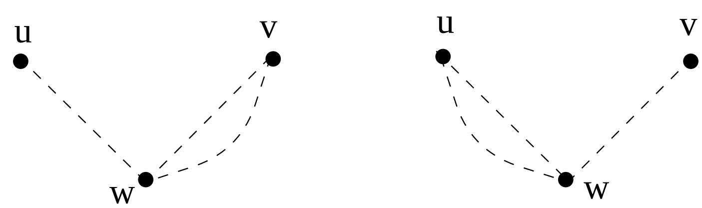


Figure 3: Edge-disjoint paths in a three-edge-connected graph.

Figure 4 depicts a graph we will discuss frequently, henceforth termed graph $M$.

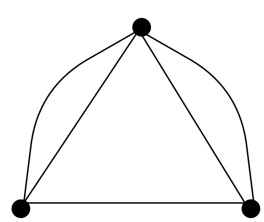

Figure 4: The graph $M$.

**Lemma 11** *Let $G$ be three-edge-connected. Let $v$ denote a non-cut point vertex in $G$ with degree at least four, let $u$ and $w$ be neighbors of $v$, and suppose $uv$ has multiplicity at least* *two. Then $G$ contains an $M$ model, with corners $u$, $v$ and $w$, and with $v$ the image of $M$'s degree-four vertex.*

**Proof** We restrict our attention to $G'$, the biconnected component of $G$ containing $v$. ($G'$ is three-edge-connected by Observation 4. Since $v$ is not a cut point, its neighborhood is unchanged in $G'$.) From Lemma 10, we know that there are three mutually edge-disjoint paths from $w$ to $\{u, v\}$ such that at most two of these paths contain $u$ and at most two contain $v$. One of these paths is the edge $wv$. If one of the other paths contains $v$ as well, the lemma holds. So suppose neither contains $v$. See Figure 5(a). To complete an $M$ model, we must find an edge-disjoint path $[vw]$. If $uv$ has multiplicity three or more, we can construct this path by combining one of the edges $vu$ and one of the paths $[uw]$. So assume $uv$ has multiplicity 2. Let $x$ denote a neighbor of $v$ other than $u$ or $w$. Since $G'$ is biconnected, there is a path $[xw]$ that does not contain any of the edges incident on $v$. Let $y$ denote the first vertex on this path (counting from $x$) common to either of the two paths $[uw]$. We combine the edge $vx$ with the paths $[xy]$ and $[yw]$ to get the desired path $[vw]$. See Figure 5(b). ∎


Figure 5: Graphs used in the proof of Lemma 11.

**Lemma 12** *Let $G$ be three-edge-connected and series-parallel, with at least three vertices. Let $v$ denote a vertex in $G$ with degree at least four. Then $G$ contains an $M$ model, with $v$ the image of $M$'s degree-four vertex, and corner $u$ adjacent to $v$ and corner $w$ adjacent to $u$ or $v$.*

**Proof** Suppose $v$ has only one neighbor, which must be $u$. Then edge $uv$ has multiplicity four. Let $w$ denote an arbitrary neighbor of $u$. Since $G$ is three-edge-connected, and since $u$

is a cut point, the graph depicted in Figure 6(a) is immersed in $G$, satisfying the statement of the lemma. Suppose $v$ has two or more neighbors, and $v$ is a cut point. Let $u$ and $w$ denote arbitrary neighbors of $v$. Now the graph in Figure 6(b) is immersed in $G$, again satisfying the statement of the lemma. Finally, suppose $v$ has two or more neighbors, and $v$ is not a cut point. For this case, we prove something slightly stronger, namely, that an $M$ model exists with $v$ the image of $M$'s degree-4 vertex, and corners $u$ and $w$ *both* adjacent to $v$.

We proceed by contradiction, and let $H = (V_H, E_H)$ denote a counterexample. Without loss of generality, we assume $H$ is minimal. That is, no counterexample exists with fewer than $|V_H|$ vertices and, for this number of vertices, no counterexample exists with fewer than $|E_H|$ edges. $H$ must be biconnected, else the biconnected component containing $v$ provides a smaller counterexample. Similarly, no three-edge-connected, series-parallel graph containing $v$ and all its incident edges can be properly immersed in $H$, else such a graph would again contradict minimality. This implies that any vertex with exactly two neighbors must be adjacent to $v$ (else the vertex could be shorted). So there must be some vertex, $x$, that is adjacent to $v$ and that has exactly one other neighbor, $y$. By Lemma 11, we know that $vx$ has multiplicity one. Three-edge-connectivity requires that $xy$ has multiplicity two or more. Now consider $H'$, obtained from $H$ by shorting $x$. $H'$ satisfies the conditions of the lemma and so (by the minimality of $H$) contains an $M$ model, with $v$ the image of the degree-4 vertex in $M$, and corners $u$ and $w$ both adjacent to $v$. The only edge in $H'$ not in $H$ is $vy$, implying that $y$ plays the role of $u$ (or, by symmetry in this case, $w$). But this means that, in $H$, we can replace the edge-disjoint paths $[vy]$, $[vy]$ and $[uy]$ with $vx$, $[vy]yx$ and $[uy]yx$ respectively, giving us an $M$ model with corners $v$, $x$ and $w$, a contradiction to the presumed existence of a counterexample. ∎

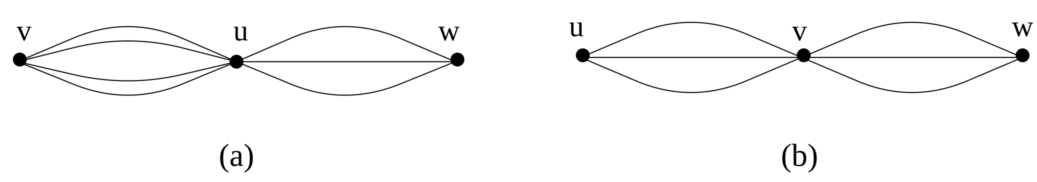


Figure 6: Graphs used in the proof of Lemma 12.

**Lemma 13** *Let $G$ be three-edge-connected and series-parallel, and let all vertices in $G$ with* *exactly one neighbor have degree three. Suppose $G$ has a cut point $v$ with degree seven or more. Then there is a $K_4$ model in $G$ with corners $u$, $v$, $w$ and $x$, where $u$ and $x$ are adjacent to $v$ and $w$ is adjacent to $u$ or $v$.*

**Proof** Let $C_1, \ldots, C_k$ denote the connected components of $G - \{v\}$. Let $A_i$ denote $C_i$ augmented with a copy of $v$ and the edges it induces. Each $A_i$ is three-edge-connected, and thus contains a model of the triple-edge shown in Figure 7(a), with any pair of vertices serving as the corners. Without loss of generality, assume $A_1$ contains the least number of edges incident on $v$, and let $H$ denote $G - C_1$. It follows that $v$ has degree four or more in $H$ and that $H$ has at least three vertices. Thus, by Lemma 12, there is an $M$ model in $H$ with $v$ the image of the degree-4 vertex in $M$, and with corner $u$ adjacent to $v$ and corner $w$ adjacent to $u$ or $v$. This $M$ model can be combined with a model of the triple-edge in $A_1$ to form in $G$ a model of the graph shown in Figure 7(b), which contains the desired $K_4$ model. ∎

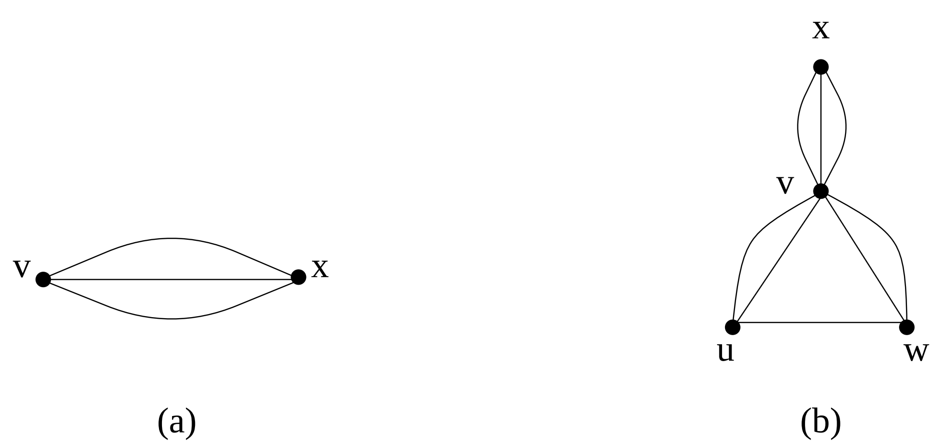


Figure 7: Graphs used in the proof of Lemma 13.

**Lemma 14** *Suppose $G$ has no cut point with degree exceeding six. Then $K_4$ is immersed in $G$ if and only if $K_4$ is immersed in a biconnected component of $G$.*

**Proof** If a biconnected component of $G$ contains $K_4$, then so does $G$, because a biconnected component is a subgraph. To prove the converse, consider a $K_4$ model in $G$ with the $K_4$ edges mapped to simple paths. Let $u$, $w$, $x$ and $y$ denote the corners of this model, and suppose there is a cut point $v$ that separates them. We know that $v$ cannot be one of the corners, else it would need degree seven or more (see Figure 8(a)). Nor can $v$ separate two corners from the others, else it would need degree eight or more (see Figure 8(b)). So it must be that $v$ separates just one corner, say $y$, from the others (see Figure 8(c)). Thus the edge-disjoint paths $[uy]$, $[wy]$ and $[xy]$ all pass through $v$, and we can construct another $K_4$ model in which $v$ replaces $y$ as a corner. By iterating this replacement, we eventually get a $K_4$ model all of whose corners (and paths) are in the same biconnected component. ∎

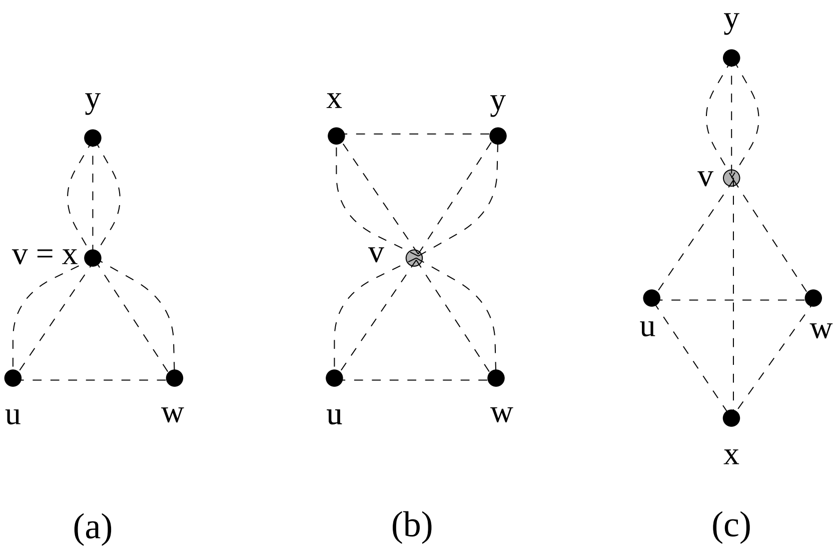


Figure 8: Models of $K_4$ that span a cut-point.

**Lemma 15** *Let $v$ denote a vertex with exactly two neighbors, $u$ and $w$, and suppose the edge $vw$ has multiplicity one. Then $u$ and $v$ can be corners of a given $K_4$ model only if degree$(u) \geq$ degree$(v) + 2$.*

**Proof** Let $x$ and $y$ denote the other corners of this model. Paths $[ux]$ and $[uy]$ need not contain $uv$. Either $[vx]$ or $[vy]$ has to pass through $u$. Thus at least three edges are incident on $u$ in addition to the copies of $uv$ (see Figure 9), and the lemma follows. ∎

**Lemma 16** *If $G$ is series-parallel and of maximum degree four, then $K_4$ is not immersed in $G$.*

**Proof** Suppose otherwise, and let $H$ denote a minimal counterexample. $H$ must be three-

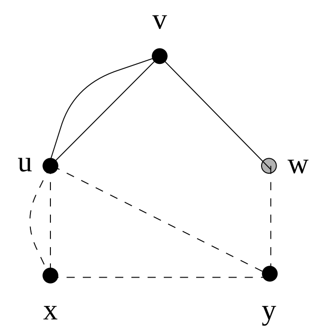


Figure 9: A model of $K_4$ with corners $u$, $v$, $x$ and $y$.

edge-connected by Lemma 1. $H$ must also be biconnected, since a cut point in a three-edge-connected graph has degree at least six. Thus, by Observation 2, $H$ contains a vertex, $v$, with exactly two neighbors, $u$ and $w$. It must be that $v$ is needed as a corner in every $K_4$ model, else we can short it, contradicting minimality. So $v$ has degree three and we assume, without loss of generality, that $uv$ has multiplicity two, $vw$ has multiplicity one. We now fix the remaining corners of some $K_4$ model. Vertex $u$ cannot be one of these corners, by Lemma 15. But now it is easy to see that $u$ can replace $v$ in this model, contradicting the fact that $v$ must be a corner. ∎

We henceforth use the term *candidate graph* to denote a biconnected, three-edge-connected, series-parallel multigraph with four or more vertices.

**Lemma 17** *In a candidate graph, $G$, suppose vertex $v$ has exactly two neighbors, $u$ and $w$, and suppose the multiplicity of $uv$ is greater than the multiplicity of $vw$. If degree$(u) -$ degree$(v) \geq 2$, then $K_4$ is immersed in $G$.*

**Proof** Suppose otherwise, and let $H$ denote a minimal counterexample. Let $x \neq v$ denote another vertex with exactly two neighbors. The edge $xu$ must exist and have multiplicity two or more, else we can short $x$, contradicting minimality. Consider the effect of shorting $v$, producing the graph $H'$. Since $u$ has degree at least four in $H'$, we know from Lemma 11 that the $M$ model illustrated in Figure 10(a) is immersed in $H'$. But this means that the graph shown in Figure 10(b), which contains $K_4$, is immersed in $H$, thereby contradicting the assumption that $H$ is a counterexample. ∎

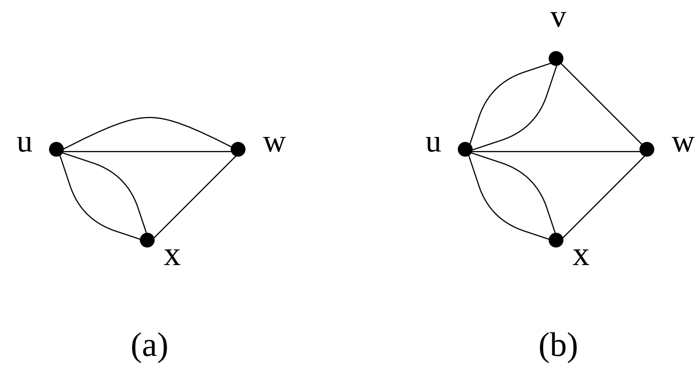


Figure 10: Graphs used in the proofs of Lemmas 17 and 19.

Recall pruning, as defined in Section 3.4.

**Lemma 18** *In a candidate graph, $G$, suppose vertex $v$ has exactly two neighbors, $u$ and $w$, and suppose $vw$ has multiplicity one. Letting $G'$ denote the graph resulting from pruning $v$, $K_4$ is immersed in $G$ if and only if it is immersed in $G'$.*

**Proof** If $K_4$ is immersed in $G'$, then it is immersed in $G$ as well, because $G' \subseteq G$. Suppose $K_4$ is immersed in $G$. If $G$ contains a $K_4$ model in which $v$ is not a corner, then so does $G'$, since pruning is irrelevant (at most one of the images of the $K_4$ edges in this model can pass through $v$.) So suppose $v$ is a corner in every $K_4$ model in $G$. Vertex $u$ must also be a corner in all these models, else we could replace $v$ with $u$, forming a model in which $v$ is not a corner. Now, by Lemma 15, $u$ has degree at least two more than $v$, a property unchanged by pruning. Thus, by Lemma 17, $K_4$ is immersed in $G'$. ∎

**Lemma 19** *In a pruned candidate graph, $G$, suppose vertex $v$ has exactly two neighbors, $u$ and $w$, and suppose $uv$ has multiplicity at least three, $vw$ has multiplicity at least two. Then there is a $K_4$ model in $G$ with corners $u$, $v$, $w$ and $x$, where $x \notin \{v, w\}$ is a neighbor of $u$.*

**Proof** The biconnectivity of $G$ ensures that $u$ has some neighbor other than $v$ (and possibly $w$) to play the role of $x$. If $v$ and $x$ are $u$'s only neighbors, then $ux$ must have multiplicity two or more ($G$ has been pruned and yet $uv$ has multiplicity three or more). Thus in $G'$, the graph that results from shorting $v$, the degree of $u$ is at least four. We conclude from Lemma 11 that the $M$ model illustrated in Figure 10(a) is immersed in $G'$, and the graph shown in Figure 10(b), which contains $K_4$, is immersed in $G$. ∎

**Lemma 20** *In a candidate graph, $G$, suppose vertex $v$ has exactly two neighbors, $u$ and $w$, suppose $uv$ and $vw$ each have multiplicity at least two, and suppose $uw$ exists. Then there is a $K_4$ model in $G$ with corners $u$, $v$, $w$ and $x$, where $x \notin \{v, w\}$ is a neighbor of $u$.*

**Proof** As in the last lemma, such an $x$ must exist. We apply Lemma 10, with $w$ playing the role of $v$ and $x$ playing the role of $w$. Thus at least one of the graphs shown in Figure 11, both of which contain $K_4$, is immersed in $G$. ∎

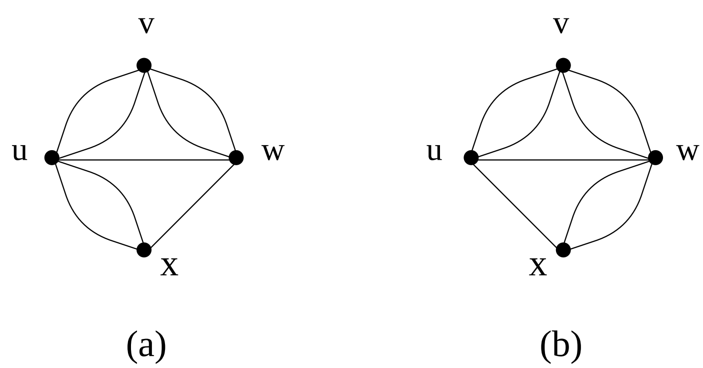


Figure 11: Graphs used in the proof of Lemma 20.

**Lemma 21** *Let $G$ denote a pruned candidate graph. $K_4$ is immersed in $G$ if and only if $G$ has a vertex of degree five or more.*

**Proof** We know from Lemma 16 that a candidate graph of maximum degree four contains no $K_4$. To prove the converse, we proceed by contradiction and assume $H$ denotes a minimal pruned candidate graph, with at least one vertex of degree five or more, but with no immersed $K_4$. It is easy to verify that $H$ has at least five vertices, a necessary property because we will use shorting to contradict minimality, and a candidate graph requires at least four vertices. Let $v$ denote a vertex in $H$ with exactly two neighbors, $u$ and $w$, and assume the multiplicity of $uv$ is at least that of $vw$. Lemma 19 guarantees that $v$ cannot have degree five or more. If $v$ has degree four, Lemma 20 and the fact that $H$ is pruned ensure that $uw$ does not exist. But now we can short $v$, obtaining a pruned candidate graph that contradicts minimality. So $v$ must have degree three and, by Lemma 17, $u$ has degree four or less. Biconnectivity requires that at most one copy of $uw$ exists. But now we can again short $v$ to obtain a pruned candidate graph, contradicting the presumed minimality of $H$. ∎

This completes the proof of the correctness of test. The work of the last two sections now provides the proof of the following principal result.

**Theorem 1** *Algorithms* decompose, components *and* test *correctly decide whether $K_4$ is immersed in an arbitrary input graph.*

# 4 Finding a Model

Once the presence of $K_4$ has been detected in a graph, our method to identify a $K_4$ model proceeds in two steps. Algorithm corners is first invoked to modify the input graph until an appropriate set of corners is isolated. Then algorithm paths is used to find the $K_4$ edge images.

## 4.1 Algorithm corners

Algorithm corners marks vertices in the input graph as part of the corner-finding process. All vertices are assumed to be unmarked initially. Algorithm corners also maintains a list for every copy of every edge, to store the sequence of edges that may have been eliminated by shorting. Each list is assumed to contain only the edge itself initially.

**algorithm** corners($G$)
input: a three-edge-connected series-parallel multigraph $G$ containing an immersed $K_4$
output: the four corners of a $K_4$ model in $G$
**begin**
    **for** each vertex with only one neighbor
        delete all but three copies of its incident edge
    **if** $G$ has a cut point $v$ of degree seven or more
        **then if** $G - \{v\}$ has three or more connected components
            **then** set $u, w$ and $x$ to neighbors of $v$ in $G$, each in a different connected
                component of $G - \{v\}$, and halt
            **else begin**
                let $C_1$ and $C_2$ denote the connected components of $G - \{v\}$
                let $A_1$ denote $C_1$ augmented with $v$ and the edges it induces

```
                let A_2 denote C_2 augmented with v and the edges it induces
                if v has degree four or more in A_1
                    then set A = A_1 and B = A_2
                    else set A = A_2 and B = A_1
                while v induces no edges of multiplicity two or more in A
                    if there is a vertex in A with only one neighbor
                        then delete this vertex and its incident edges
                        else short some vertex in A with only two neighbors
                set u to some vertex in A such that uv has multiplicity at least two
                if v has a neighbor other than u in A
                    then set w to any one of these neighbors
                    else set w to any neighbor of u in A other than v
                set x to any vertex in B that is a neighbor of v and halt
            end
    let C denote some biconnected component containing K_4, and discard G − C
    prune all vertices with exactly two neighbors
    while true
        begin
            set v to an unmarked vertex with exactly two neighbors u and w, with the
                multiplicity of uv at least that of vw
            if v has degree at least five, or v has degree four and uw exists
                then set x to any neighbor of u besides v or w and halt
                else if v or u has degree four
                    then short v
                    else if uw exists
                        then set x to any neighbor of u other than v or w and halt
                        else if there is an edge ua, a ≠ v, of multiplicity two or more
                            then set x to a and halt
                            else if there are two vertices of degree five or more
                                then short v
                                else mark v
        end
end
```

We address the correctness of corners. Suppose $G$ contains a cut point $v$ of degree at least seven. Lemmas 11 and 12 tell us how to find the corners of an $M$ model, and from this Lemma 13 tells us how to find the corners of a $K_4$ model, as long as either (1) $G - \{v\}$ has three or more connected components or (2) there is an augmented component in which $v$ has degree at least four and $uv$ has multiplicity at least two. If neither of these conditions is initially satisfied, condition (2) is easily forced with a series of vertex deletions and shorting operations.

So suppose no cut point of degree seven or more exists, and consider some biconnected component containing an immersed $K_4$. This component must also contain a vertex with exactly two neighbors (Observation 2) and a vertex of degree five or more (Lemma 21). In this event, we employ Lemmas 19, 20, and 21, plus Lemma 22, which follows.

**Lemma 22** *In a candidate graph, $G$, suppose vertex $v$ has exactly two neighbors, $u$ and $w$, and suppose $uv$ has multiplicity two, $vw$ has multiplicity one and $u$ has degree at least five. Let $x$ denote a neighbor of $u$ other than $v$ or $w$. If either $ux$ has multiplicity at least two or $uw$ exists, then there is a $K_4$ model in $G$ with $u$, $v$, $w$ and $x$ as corners.*

**Proof** In $G'$, the graph resulting from shorting $v$, $u$ has degree at least four, and either $ux$ has multiplicity at least two or $uw$ now does. Then by Lemma 11, there is an $M$ model in $G'$ with corners $u$, $w$ and $x$, and with $u$ the image of $M$'s degree-four vertex. Thus the graph in Figure 10(b), which contains the desired $K_4$ model, is immersed in $G$. ■

If an immersed $K_4$ cannot yet be identified, then a vertex, $v$, with exactly two neighbors is shorted as long as the resulting graph retains at least one vertex of degree at least five. Accordingly, if one of $v$'s neighbors, $u$, is the only vertex of degree at least five, $uv$ has multiplicity two, and all other edges incident on $u$ are simple, then $v$ cannot be shorted. It suffices in this case to mark $v$ as having been visited, since at most one vertex can be so marked and another candidate for $v$ is always available.

In each iteration, corners deletes, shorts or marks some vertex. Handling any of these operations and updating the appropriate edge list requires only a constant number of steps. Thus corners runs in linear time.

## 4.2 Algorithm paths

Algorithm paths uses the property that $k$ edge-disjoint paths exist between a pair of vertices if and only if a network flow of value $k$ is possible between them.

**algorithm** paths($G$, $s$, $t_1, \ldots, t_k$)
input: a multigraph $G$ and distinguished vertices $s$, $t_1, \ldots, t_k$
output: edge-disjoint paths $p_1, \ldots, p_k$, with $p_i$ connecting $s$ to $t_i$, if such paths exist

```
begin
    construct an edge-weighted digraph G', by replacing each edge uv of multiplicity m with
        the directed edges (u, v) and (v, u), each of capacity m
    add to G' a vertex t and the edges (t_1, t), ..., (t_k, t), each of capacity one
    find a flow of value k from s to t, if such a flow exists
    if there is no such flow
        then halt
        else for each edge (u, v) in G' do
            if both (u, v) and (v, u) have positive flow values
                then set flow((u, v)) = max{0, flow((u, v)) − flow((v, u))} and
                    set flow((v, u)) = max{0, flow((v, u)) − flow((u, v))}
    discard from G' any edge without a positive flow
    for i = 1 to k do
        begin
            set p'_i to a path in G' from s to t_i
            set p_i to the corresponding path in G
            decrement in G' the flow along each edge in p'_i by one
            delete from G one copy of each edge in p_i
        end
    output p_1, ..., p_k
end
```

We address the correctness and use of paths. In the following figures, paths that are mere edges are shown as solid lines. These edges are temporarily deleted so that paths can be employed to find additional paths with multiple edges, depicted with dashed lines. To illustrate, consider the case in which the $K_4$ model spans a cut-point $v$ and $G - \{v\}$ has three or more connected components. See Figure 12. Three calls are made to paths, each with $v$ playing the role of $s$ and $k$ set to two. (The first call uses $u = t_1 = t_2$; the second uses $w = t_1 = t_2$; the third uses $x = t_1 = t_2$.) If $G - \{v\}$ has two connected components, two calls suffice. See Figures 13 and 14. If the $K_4$ model is in a single biconnected component, one call is enough. See Figure 15.

Recall that the input to paths has at most a linear number of edges and no more than four copies of any edge. Thus it takes only linear time to construct $G'$ and to read off paths (using, for example, a shortest paths algorithm) after a flow of value $k$ has been found. The running time of paths is therefore dominated by the algorithm for finding network flows. So we employ a flow method such as Ford-Fulkerson, which runs in linear time as long as $k$ is bounded by an integer constant and all edge-capacities are integers, as is the case here.

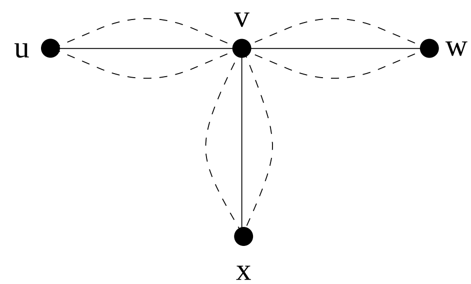


Figure 12: Paths to be found if $G - \{v\}$ has three or more connected components.

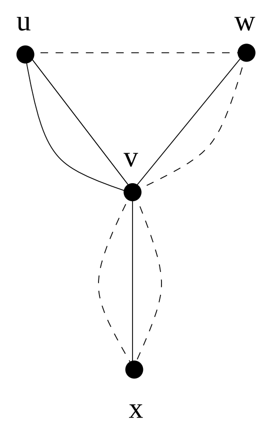


Figure 13: Paths to be found if $v$ has at least two neighbors in $u$'s component.

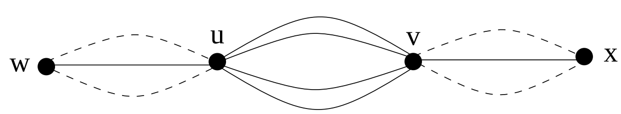


Figure 14: Paths to be found if $v$ has no neighbor besides $u$ in $u$'s component.

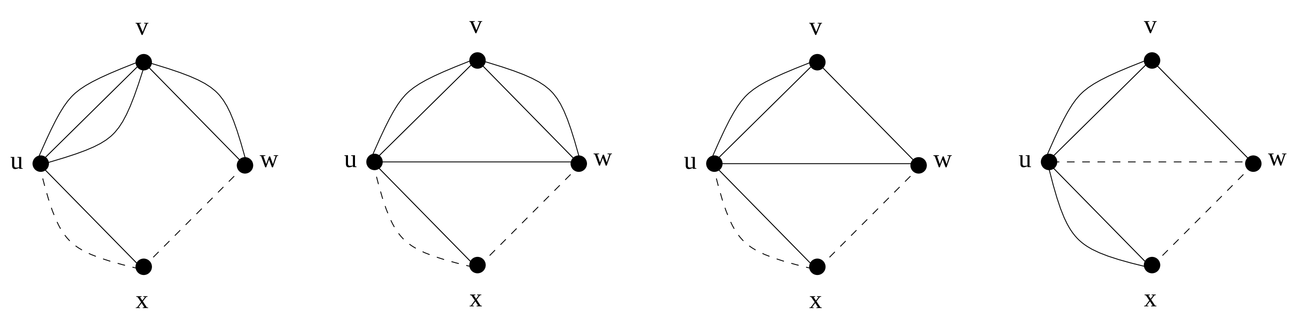


Figure 15: Paths to be found if the $K_4$ model lies in a biconnected component.

In summary, to find a $K_4$ model we invoke corners once and paths at most three times. The entire model-finding process is accomplished in linear time.

**Theorem 2** *Algorithms* corners *and* paths *correctly isolate a $K_4$ model if $K_4$ is immersed in an arbitrary input graph.*

# 5 Discussion

## 5.1 Computational Experience

We implemented our algorithms in C and ran them on a SUN SPARCstation 20. Representative results are listed in Table 4. Each execution time shown is in seconds, and was obtained by averaging the times observed in a dozen runs. The graphs employed are (pseudo) random, generated by fixing the number of vertices and then randomly adding edges until the desired average degree was reached. Each edge was added only after verifying that its addition maintained series-parallelness, since a quick test for a topological (and hence an immersed) $K_4$ suffices to eliminate non-series-parallel graphs.

It is clear that from these results that our algorithms are practical, not just asymptotically optimal. They take only seconds to process graphs with thousands of vertices. The running time of the detection algorithm is affected mainly by the size of the input graph. One might suspect that the distribution of edges over vertices might also have an effect, but we sampled several edge-probability distributions and could find no noticeable differences. On the other hand, the model-finding algorithm does appear to take slightly longer on graphs in which we have forced corners to be connected only by long paths. Even on such contrived instances, finding a model takes no more than twice the average time for random graphs of similar size.

| Average Degree | Number of Vertices | Detection Time | Percent with Immersed $K_4$ | Model-Finding Time |
|---|---|---|---|---|
| 1.0 | 200 | 0.01 | 0 | N/A |
| | 500 | 0.03 | 0 | N/A |
| | 1000 | 0.07 | 0 | N/A |
| | 2000 | 0.14 | 0 | N/A |
| | 5000 | 0.35 | 0 | N/A |
| | 10000 | 0.70 | 0 | N/A |
| 1.25 | 200 | 0.02 | 0 | N/A |
| | 500 | 0.05 | 0 | N/A |
| | 1000 | 0.09 | 0 | N/A |
| | 2000 | 0.16 | 0 | N/A |
| | 5000 | 0.41 | 17 | 0.72 |
| | 10000 | 0.80 | 42 | 1.42 |
| 1.5 | 200 | 0.02 | 8 | 0.03 |
| | 500 | 0.05 | 33 | 0.08 |
| | 1000 | 0.10 | 50 | 0.15 |
| | 2000 | 0.20 | 58 | 0.32 |
| | 5000 | 0.54 | 83 | 0.83 |
| | 10000 | 1.09 | 92 | 1.66 |
| 1.75 | 200 | 0.02 | 25 | 0.04 |
| | 500 | 0.05 | 67 | 0.09 |
| | 1000 | 0.11 | 83 | 0.19 |
| | 2000 | 0.24 | 100 | 0.37 |
| | 5000 | 0.61 | 100 | 0.88 |
| | 10000 | 1.19 | 100 | 1.75 |
| 2.0 | 200 | 0.02 | 67 | 0.05 |
| | 500 | 0.05 | 75 | 0.11 |
| | 1000 | 0.12 | 100 | 0.21 |
| | 2000 | 0.23 | 100 | 0.40 |
| | 5000 | 0.60 | 100 | 1.01 |
| | 10000 | 1.20 | 100 | 2.05 |
| 2.25 | 200 | 0.03 | 100 | 0.05 |
| | 500 | 0.07 | 100 | 0.11 |
| | 1000 | 0.13 | 100 | 0.22 |
| | 2000 | 0.26 | 100 | 0.43 |
| | 5000 | 0.64 | 100 | 1.13 |
| | 10000 | 1.27 | 100 | 2.24 |

Table 4

## 5.2 Applications Revisited

Fast immersion tests are of interest in their own right. In practice, they also have potential as indicators of graph width metrics. To illustrate, we return to the cutwidth problem, which has appeared in a wide variety of VLSI applications (see, as examples, [FHKY, HPK]). Deciding whether a graph has small cutwidth is an important part of many layout processes. Graphs representing circuits are frequently series-parallel. More generally, they tend to be sparse, with at most a linear number of edges, and of bounded degree due to limitations on porting and fan-in/out. Integer weights are used to model multiple edges in these applications, just as we have used them here. The presence of an immersed $K_4$ in such a graph guarantees that it cannot have cutwidth three. The absence of $K_4$, however, merely approximates its cutwidth at three. In particular, such an absence says nothing at all about how to find a layout of width three even if many should exist. To solve this problem, our algorithms can be used in conjunction with previously-studied "self reduction" techniques [BFL, FL2] to search for a layout in $O(n^2)$ time.

Many other combinatorial problems may benefit from fast immersion tests. For example, a variety of *load factor* [FL1] problems can be decided by a finite battery of immersion tests, including $K_4$. A problem indirectly approachable with this method is graph bisection. Bounded cutwidth is a sufficient, but not a necessary, condition for bounded bisection width. For problems such as these, there is interest in devising fast tests for other key graphs [LR, MK].

## 5.3 Parallelization

It is not difficult to devise parallel versions of decompose, components and test.

Biconnected components can be found in $O(\log n)$ time on a CRCW PRAM with $O((m+n)\alpha(m,n)/\log n)$ processors [FRT], where $\alpha(m,n)$ denotes the inverse of Ackermann's function. Deciding whether a graph is series-parallel can be done in $O(\log^2 n + \log m)$ time with $O(m+n)$ processors [He]. A parallel version of decompose therefore needs at most $O(\log^2 n)$ time with $O(n)$ processors.

The triconnected components algorithm of [FRT], modified slightly to find three-edge-connected components [Ra], yields a parallel version of `components` that runs in $O(\log n)$ time with $O(n \log\log n / \log n)$ processors. It is straightforward to parallelize `test` so that it takes constant time with $O(n)$ processors.

Thus, in principle, it is possible to determine whether a graph has an immersed $K_4$ in $O(\log^2 n)$ time with $O(n)$ processors on the CRCW PRAM model. We did not implement this scheme because many of the algorithms mentioned are highly impractical. The problem of devising an efficient parallel model-finding method remains open.

## References


[BFL] D. J. Brown, M. R. Fellows and M. A. Langston, "Polynomial-Time Self-Reducibility: Theoretical Motivations and Practical Results," *Int'l J. of Computer Mathematics* 31 (1989), 1–9.

[Bo] H. L. Bodlaender, "A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth," *Proceedings, 25th Annual ACM Symposium on Theory of Computing* (1993), 226–234.

[Di] G. A. Dirac, "In Abstrakten Graphen Vorhandene Vollstandige 4-Graphen und ihre Unterteilungen," *Mathematische Nachrichten* 22 (1960), 61–85.

[Du] R. J. Duffin, "Topology of Series-Parallel Networks," *Journal of Mathematical Analysis and Applications* 10 (1965), 303–318.

[ET] S. Even and R. E. Tarjan, "Computing an *st*-numbering," *Theoretical Computer Science* 2 (1976), 339-344.

[FHKY] T. Fujii, H. Horikawa, T. Kikuno and N. Yoshida, "A Heuristic Algorithm for Gate Assignment in One-Dimensional Array Approach," *IEEE Transactions on Computer-Aided Design* 6 (1987), 159–164.

[FL1] M. R. Fellows and M. A. Langston, "On Well-Partial-Order Theory and Its Application to Combinatorial Problems of VLSI Design," *SIAM Journal on Discrete Mathematics* 5 (1992), 117–126.

[FL2] M. R. Fellows and M. A. Langston, "On Search, Decision and the Efficiency of Polynomial-Time Algorithms," *Journal of Computer and Systems Sciences* 49 (1994), 769–779.

[FRT] D. Fussell, V. Ramachandran, R. Thurimella, "Finding Triconnected Components by Local Replacements," *Proceedings, 16th International Colloquium on Automata, Languages and Programming* 372 (1989), 379–393.

[He] X. He, "Efficient Parallel Algorithms for Series Parallel Graphs," *Journal of Algorithms* 12 (1991), 409–430.

[HPK] Y-S. Hong, K-H. Park and M. Kim, "Heuristic Algorithms for Ordering the Columns in One-Dimensional Logic Arrays," *IEEE Transactions on Computer-Aided Design* 8 (1989), 547–562.

[LEC] A. Lempel, S. Even, and I. Cederbaum, "An Algorithm for Planarity Testing of Graphs," in *Theory of Graphs: International Symposium* (P. Rosenstiehl, ed.), Gordon and Breach, New York, 1967, 215–232.

[LG] P. C. Liu and R. C. Geldmacher, "An O(max(m,n)) Algorithm for Finding a Subgraph Homeomorphic to $K_4$," *Congressus Numerantium* 29 (1980), 597–609.

[LR] M. A. Langston and S. Ramachandramurthi, "Dense Layouts for Series-Parallel Circuits," *Proceedings, First Great Lakes Symposium on VLSI* (1991), 14–17.

[MK] P. J. McGuinness and A. E. Kezdy, "An Algorithm to Find a $K_5$ Minor," *Proceedings, Third ACM-SIAM Symposium on Discrete Algorithms* (1992), 345–356.

[Ra] V. Ramachandran, private communication.

[RS] N. Robertson and P. D. Seymour, "Graph Minors XIII. The Disjoint Paths Problem," *Journal of Combinatorial Theory, Series B* 63 (1995), 65–110.

[Ta] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing* 1 (1972), 146–159.

[VTL] J. Valdes, R.E. Tarjan, and E. Lawler, "The Recognition of Series-Parallel Digraphs," *SIAM Journal on Computing* 11 (1982), 298–313.