Journal of Algorithms **30,** 344–378 (1999)
Article ID jagm.1998.0991, available online at http://www.idealibrary.com on IDEAL®

Fast Algorithms for K_4 Immersion Testing*,†

Heather D. Booth

Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996

Rajeev Govindan

Qualcomm Incorporated, San Diego, California 92121

Michael A. Langston

Department of Computer Science, University of Tennessee, Knoxville, Tennessee 37996

and

Siddharthan Ramachandramurthi

LSI Logic Corporation, Waltham, Massachusetts 02451

Received February 13, 1996; revised August 15, 1998

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. © 1999 Academic Press

Contents.

* 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, MI in February, 1992.

0196-6774/99 $30.00
Copyright © 1999 by Academic Press
All rights of reproduction in any form reserved.

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 by 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 by 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, over all pairs u and v of consecutive vertices, of the number of edges from E that must be cut to split the layout between u and v. 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 [9], computing its tree decomposition [2], and applying dynamic programming to test each obstruction against the decomposition [19]. We refer the reader to [8] 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 3 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 3, because any arrangement of its vertices on a line requires 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 [4]. 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 Fig. 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, [15]).

In the next section we state relevant definitions and we 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.

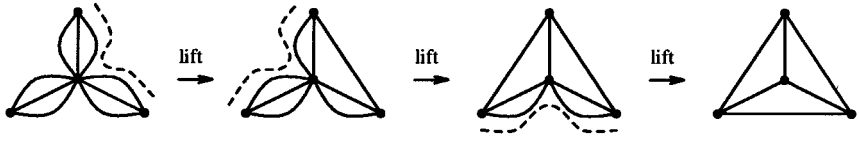

FIG. 1. A series-parallel graph with an immersed K_4.

2.1. *Three-Edge Connectivity*

A *cut point* in a connected graph G is a vertex whose removal disconnects G. Two vertices in G are said to be *biconnected* if they cannot be disconnected by the removal of any cut point. A *biconnected component* of G is the subgraph induced by a maximal set of pairwise biconnected vertices.

A *cut edge* in 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 preference 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 5 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 2. 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 graph 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. Nodes labeled S and P are termed S-nodes and P-nodes, respectively.

We conclude this section by noting from [3] 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 [11]. It determines whether G is series-parallel and, if so, computes a decomposition tree for each biconnected component. 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 [14]. Such a numbering is called an *s,t-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 the biconnected components $B_1, \ldots, B_k$ of G
  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
        B̄_i := the directed graph obtained by orienting each edge in B_i from
               the endpoint 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 halt
      end
    output T_1, ..., T_k
end
```

The correctness of decompose is based on the observation that any pair of adjacent vertices may be chosen as s and t [11]. Efficient methods for finding biconnected components and computing s, t-numberings are known [20, 5]. Techniques for determining whether directed graphs are 2TSP and finding decomposition trees can be found in [21]. All these algorithms are linear in n and m; 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 because 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 in 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-*nonseparating* 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-nonseparating 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 Fig. 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 an s, t-separating pair in G_1. G_{11}, G_{12}, and G_2 are the three-edge-connected components of G.

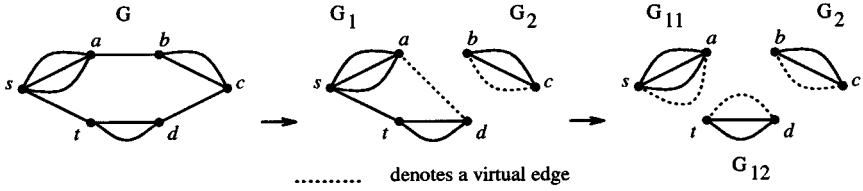

FIG. 2. A two-terminal series-parallel graph with cut edge pairs.

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. We can show that $G_{s,t}$ contains at most one cut edge pair. In the second phase the last remaining cut edge pair, if it exists, is found and 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 binary decomposition tree merely by identifying all pairs of adjacent nodes that are of the same type.

Let G be a biconnected 2TSP graph with a compressed decomposition tree T. 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 $G'_{s,t}$, the graph containing s and t that is left after removing e and f from G (the other graph left is a three-edge-connected component).

Pseudo-code for components is presented in the following text. In a compressed tree, each internal node will have at least two children, stored in a linked list called *child list*. Stored along with each tree node is its type (P, S, or leaf), a pointer to its child list and, if it is a leaf node, 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 nonroot tree node, this returns the child following q in the child list of the parent of q or null if no such child exists.

leftmost_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
  $r :=$ 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 subtree 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 compressed series-parallel decomposition tree T of a multigraph G
output: the graph G, after deletion of all s, t-nonseparating pairs that are contained in the subtree 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
        leaf1, leaf2 := the first two leaf-node children of q
        (u, v) := the ordered edge associated with leaf1
        (w, x) := the ordered edge associated with leaf2
        delete edges uv and wx from G
        add edges ux and vw to G
        create tree node new representing the ordered edge (u, x)
        if q has more than two children
          then replace all children of q between leaf1 and leaf2 (inclu-
            sive) by new
          else replace q by new
      end
end
```

```
algorithm remove_sep(root)
input: the root of a compressed series-parallel decomposition tree T of a
       multigraph G without any s, t-nonseparating pairs
output: the graph G after deletion of the s, t-separating pair, if present,
        and addition of virtual edges
begin
  if root has exactly two children
    then begin
      c1, c2 := the children of root
      if c1 is not a leaf node then c1 := leftmost_leaf(c1)
      if c2 is not a leaf node then c2 := leftmost_leaf(c2)
      if c1 and c2 are both nonNULL
        then begin
          (u, v) := the ordered edge associated with c1
          (w, x) := the ordered edge associated with c2
          delete edges uv and wx from G
          add edges uw and vx to G
          end
      end
end
```

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

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 [18], by modifying the ear decomposition techniques used to decide vertex connectivity in [10].

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 because G is biconnected), and remove_non_sep eliminates only edges that are children of S-nodes.

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

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 previously. Table 2 relates the class of e and f in G_1 to their class in G_s and G_p.

TABLE 1

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
noncut edge	noncut edge	noncut pair	noncut pair
noncut edge	cut edge	f a cut edge	noncut pair
cut edge	noncut edge	e a cut edge	noncut pair
cut edge	cut edge	cut edges	s, t-separating

TABLE 2

Class of edges e and f		
In G_1	In G_s	In G_p
cut edges	cut edges	s,t-nonseparating
s,t-nonseparating	s,t-nonseparating	s,t-nonseparating
s,t-separating	s,t-separating	noncut pair
noncut pair	noncut pair	noncut pair

Let G be a 2TSP graph with compressed decomposition tree T. Let x denote an arbitrary internal node in T, and let T_x denote the subtree of T rooted at x. Then the 2TSP graph that has T_x as a decomposition tree is a *constituent graph* for G with respect to T. For any edge e in G, let $\hat{e}$ denote the node in T with label e. If e and f are edges in G then the *least constituent graph containing e and f* is the smallest constituent graph H of G that contains both e and f. This graph has a decomposition tree T_z, where z is the least common ancestor of $\hat{e}$ and $\hat{f}$ in T. Table 3 gives the relation between the class of an edge in H and its class in G.

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-nonseparating pair in 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-nonseparating pair. Because 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-nonseparating pair in H. Therefore they must be cut edges in H, as claimed. ▮

TABLE 3

Class of edges e and f	
In H	In G
cut edges	cut edges or s,t-nonseparating
s,t-nonseparating	s,t-nonseparating
s,t-separating	s,t-separating or noncut pair
noncut pair	noncut pair

LEMMA 6. *If G is a 2TSP graph and T is a compressed decomposition tree for G, then edge e is a cut edge in G if and only if the root of T is an S-node and $\hat{e}$ is a child of the root.*

Proof. Suppose e is a cut edge in G. Then no ancestor of e in T is a P-node, because subtrees rooted at P-nodes represent biconnected graphs. Hence the path from the root of T to e includes only S-nodes. But because T is compressed, e must be a child of the root, which must be an S-node. Conversely, if the root of T is an S-node and $\hat{e}$ is a child of the root, then every path from s to t in G must include e. Therefore e is a cut edge. ∎

LEMMA 7. *If e and f are edges in a biconnected 2TSP graph G with a compressed decomposition tree T, then e and f are an s, t-nonseparating 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 H be the 2TSP graph having T_z as a decomposition tree. Note that H is the least constituent graph of G that contains e and f.

Suppose e and f are s, t-nonseparating. By Lemma 5, e and f are cut edges in H. Then, by Lemma 6, $\hat{e}$ and $\hat{f}$ are children of z and z is an S-node.

Now suppose $\hat{e}$ and $\hat{f}$ are siblings whose parent is an S-node. This implies that e and f are cut edges in H. Then, by Table 3, e and f must be either cut edges or an s, t-nonseparating pair in G. Because G is biconnected, e and f must in fact be an s, t-nonseparating 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 pre-order traversal of T. Let H_x denote the graph having T_x as a decomposition tree.

LEMMA 8. *Let G be a biconnected 2TSP graph and let T be a compressed decomposition tree for G. In G, let e and f be an s, t-nonseparating pair whose removal yields a graph G_1 containing s and t, and another graph G_2. Suppose $\hat{e}$ occurs before $\hat{f}$ in T, and let (u, v) and (w, x) be the pairs stored with $\hat{e}$ and $\hat{f}$, respectively. Then the edges in G_2 are $\{g \mid \hat{g}$ 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. Because e and f are s, t-nonseparating, by Lemma 7, $\hat{e}$ and $\hat{f}$ are siblings whose parent z is an S-node. Because G is biconnected, z's parent, y, is a P-node.

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, e and f are an s,t-nonseparating 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. Because 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 separates 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 8. 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 8 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. Because the nodes between $\hat{e}$ and $\hat{f}$ are children of an S-node, the decomposition tree for G_2' is as claimed. ▮

LEMMA 9. *Let G be a biconnected 2TSP graph and let T be a compressed decomposition tree for G. A pair of s,t-nonseparating edges, e, f, is a special pair in 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-nonseparating pair.*

Proof. Because e and f are s,t-nonseparating, 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 we 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-nonseparating pair. 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 in G', a contradiction. If, on the other hand, y is a nonleaf node whose subtree T_y represents a graph having an s,t-nonseparating pair, then T_y contains an S-node with two leaves as children. These nodes also represent an s,t-nonseparating pair in 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. According to Lemma 7, z is an S-node. Because T is compressed and no y is a leaf, each y is a P-node. The root of T' is a P-node, implying that G' has no cut edge. Moreover, the root has exactly two children, one of which is a virtual edge. Let x denote the other child. If there is only one node y between e and f in T, then x is a P-node with $T_x = T_y$; else x is an S-node with all the y as children. Because each T_y is rooted at a P-node, no T_y has a cut edge and neither does T_x. Therefore G' has no s,t-separating pairs. If T' has an s,t-nonseparating pair, then T_x must have either an s,t-nonseparating pair or a pair of cut edges. This in turn means that some T_y contains a cut edge or an s,t-nonseparating cut edge pair, a contradiction. ∎

COROLLARY 2. *If a biconnected* 2*TSP graph contains an* s,t*-nonseparating pair, then it contains a special pair.*

Proof. Let G be a biconnected 2TSP graph that contains one or more s,t-nonseparating pairs. Let T denote a compressed decomposition tree for G. By Lemma 6, T must contain an S-node whose children include at least two leaf nodes. Among all such S-nodes, choose one, say x, such that no other S-node in T_x has two or more leaf nodes as children. Lemma 6 implies that for each nonleaf child y of x, T_y cannot contain an s,t-nonseparating pair. Now from among all the leaf nodes that are children of x, choose two, say $\hat{e}$ and $\hat{f}$, such that no child of x that occurs between $\hat{e}$ and $\hat{f}$ is a leaf. By Lemma 9, e and f form a special pair. ∎

LEMMA 10. *Let G be a biconnected 2TSP graph that contains no special pairs. Then G contains at most one s,t-separating pair.*

Proof. We use contradiction. Suppose $\{e_1, e_2\}$ and $\{e_3, e_4\}$ are two distinct s,t-separating pairs in G. Assume, without loss of generality, that $e_1 \neq e_3$ and $e_2 \neq e_3$. Let G_1 and G_2 be 2TSP graphs such that G is the result of their parallel composition. From Tables 1 and 2, we see that any s,t-separating pair in G consists of a cut edge in G_1 and a cut edge in G_2. Thus each of e_1, e_2, and e_3 is a cut edge in either G_1 or G_2. Without loss of generality, assume that two of these three edges are in G_1. But now, by Table 2, these two edges constitute an s,t-nonseparating pair in G, contradicting Lemma 9. ∎

Recall that it suffices to find three-edge-connected components of biconnected graphs, because each cut edge and cut edge pair in any graph is contained within one of its biconnected components. Lemmas 5 through 9 demonstrate that 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. By Lemma 10, at most one s,t-separating pair remains in the graph. This pair, if it exists, is found and removed using remove_sep. We omit the analysis of this last step, 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 2. 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 do
    delete all but three copies of edges incident on v
  if any cut point in G has degree 7 or more
    then output YES and halt
  for each biconnected component B with four or more vertices do
    begin
      prune B
```

```
        if there is a vertex in B with degree 5 or more
          then output YES and halt
      end
    output NO and halt
end
```

THEOREM 1. *Algorithm* test *runs in* $O(n)$ *time on a graph with* n *vertices*.

Proof. Biconnected components and cut points can be found in linear time using a depth-first search. Operations such as deleting edges incident on vertices with only one neighbor and pruning vertices with two neighbors can be accomplished in constant time. The existence of a vertex with degree 5 or more can be confirmed simply by scanning the list of edges. Thus the theorem holds. ▮

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 (see [13], for example, for a proof).

OBSERVATION 2. *Every biconnected series-parallel multigraph with four or more vertices contains at least two nonadjacent 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 if we 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.

The next lemma justifies the first step in test, in which edges incident on vertices with only one neighbor have their multiplicity reduced to 3.

LEMMA 11. *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. Because 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' because G' is a subgraph of G. ▮

Under certain conditions, it is possible to detect an immersed K_4 by checking whether the graph in Fig. 3, henceforth termed graph M, is immersed in the input graph. The next four lemmas are useful in finding M-models.

LEMMA 12. *Let G be three-edge-connected, with noncut point vertices u and v. Let $w \notin \{u, v\}$ denote a 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, at most two contain v, and none contains both u and v.*

Proof. The paths we seek to identify are illustrated in Fig. 4, with dashed lines denoting 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 because 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

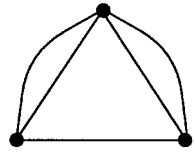

FIG. 3. The graph of M.

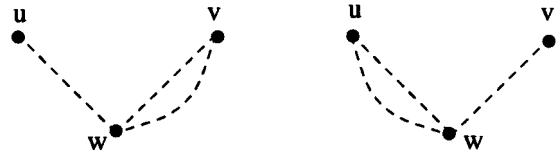

FIG. 4. Edge-disjoint paths in a three-edge-connected graph.

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. ▮

LEMMA 13. *Let G be three-edge-connected. Let v denote a noncut point vertex in G with degree at least* 4, *let u and w be neighbors of v, and suppose uv has multiplicity at least* 2. *Then G contains an M model, with corners u, v, and w, and with v the image of M's degree-*4 *vertex.*

Proof. We restrict our attention to G', the biconnected component of G containing v. (G' is three-edge-connected by Observation 4. Because v is not a cut point, its neighborhood is unchanged in G'.) From Lemma 12, 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 of these paths 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. This situation is illustrated in Fig. 5a. To complete an M model, we must find an edge-disjoint path $[vw]$. If uv has multiplicity 3 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, and let x denote a neighbor of v other than u or w. Because 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 can combine the edge vx with the paths $[xy]$ and $[yw]$ to get the desired path $[vw]$, as is clear from Fig. 5b. ▮

FIG. 5. Graphs used in the proof of Lemma 13.

LEMMA 14. *Let G be three-edge-connected, with at least three vertices. Let v denote a vertex in G with only one neighbor u, and let $w \neq v$ denote a neighbor of u. Suppose v has degree at least 4. Then G contains an M model, with corners u, v, and w, and with v the image of M's degree-4 vertex.*

Proof. The graph depicted in Fig. 6 is immersed in G. A satisfactory model of M can be obtained by lifting two pairs of edges in this graph. ▮

LEMMA 15. *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 4. Then G contains an M model in which v is the image of M's degree-4 vertex.*

Proof. If v has only one neighbor, then the model exists by Lemma 14. Otherwise let u and w denote arbitrary neighbors of v. If v is a cut point, then the graph in Fig. 7 is immersed in G, satisfying the statement of the lemma. So suppose v is not a cut point. If any vertex in G is adjacent to v by two or more edges, then Lemma 13 applies, and the model exists. So suppose there are no edges of multiplicity greater than 1 incident on v. Then we may force this condition by iteratively deleting vertices with only one neighbor and shorting vertices with only two neighbors. Neither operation changes the degree of v, or affects the three-edge-connectivity and series-parallelness of G. Thus, by the time the order of G is reduced to 3 (or before), some edge incident on v must have multiplicity 2 or more. Lemma 13 and Observation 1 then imply that the lemma holds. ▮

The preceding lemmas enable us to detect K_4 models that span cut points.

LEMMA 16. *Let G be three-edge-connected and series-parallel, and let all vertices in G with exactly one neighbor have degree 3. Suppose G has a cut point v with degree 7 or more. Then K_4 is immersed in G.*

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 Fig. 8a, 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 4 or more in H and that H has at least three vertices. Thus, by Lemma 15, there is an M model in H with v the image of the degree-4

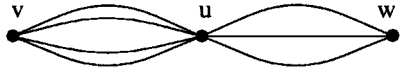

FIG. 6. Graph used in the proof of Lemma 14.

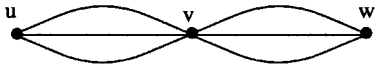

FIG. 7. Graph used in the proof of Lemma 15.

vertex in M. 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 Fig. 8b, which contains an immersed K_4. ▮

K_4 models that span cut points may exist even if no cut points have degree 7 or more. Nevertheless, we can restrict our search to biconnected components, as we show in the following text.

LEMMA 17. *Suppose G has no cut point with degree exceeding* 6. *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 7 or more (see Fig. 9a). v cannot separate two corners from the others, else it would need degree 8 or more (see Fig. 9b). So it must be that v separates just one corner, say y, from the others (see Fig. 9c). 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. ▮

FIG. 8. Graphs used in the proof of Lemma 16.

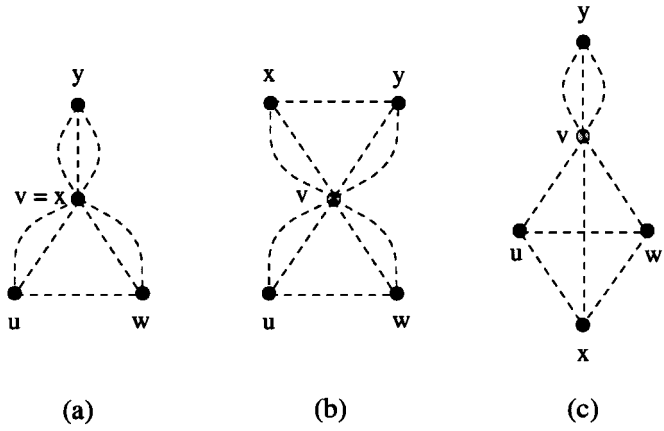

FIG. 9. Models of K_4 that span a cut point.

The remaining lemmas in this section deal with detecting an immersed K_4 in a biconnected, three-edge-connected, series-parallel graph. First we show that such a graph must have at least one vertex of degree 5 or more if it is to contain an immersed K_4. Then we present a series of lemmas that lead up to a proof of the converse; that is, if even one vertex has degree 5 or more, then an immersed K_4 exists.

LEMMA 18. *Let v denote a vertex with exactly two neighbors, u and w, and suppose the edge vw has multiplicity* 1. *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 Fig. 10), and the lemma follows. ▌

LEMMA 19. *If G is series-parallel and of maximum degree* 4, *then K_4 is not immersed in G.*

Proof. Suppose otherwise, and let H denote a minimal counterexample. H must be three-edge-connected by Lemma 1. H must also be biconnected, because a cut point in a three-edge-connected graph has degree at least 6. 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 3 and we assume, without loss of generality, that uv has multiplicity 2, vw has multiplicity 1. We now fix the remaining corners of some K_4 model. Vertex u cannot be one of these corners, by Lemma 18. But now it

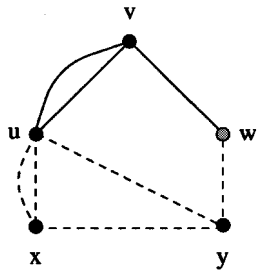

FIG. 10. A model of K_4 with corners u, v, x, and y.

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 20. *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 2 or more, else we can short x without affecting the degree of u or v, thus contradicting the minimality of G. Consider the effect of shorting v, producing the graph H'. Because u has degree at least 4 in H', we know from Lemma 13 that the M model illustrated in Fig. 11a is immersed in H'. But this means that the graph shown in Fig. 11b, which contains K_4, is immersed in H, thereby contradicting the assumption that H is a counterexample. ▮

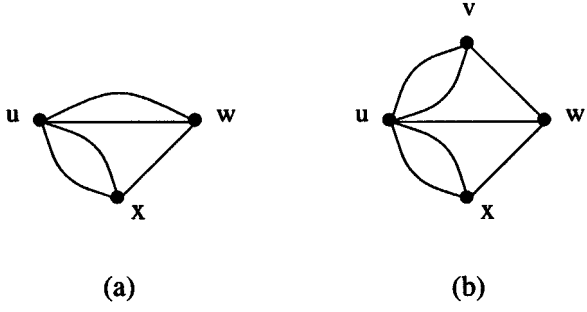

FIG. 11. Graphs used in the proofs of Lemmas 20 and 22.

Recall pruning, as defined in Section 3.4.

LEMMA 21. *In a candidate graph, G, suppose vertex v has exactly two neighbors, u and w, and suppose vw has multiplicity* 1. *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', because 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 18, u has degree at least 2 more than v, a property unchanged by pruning. Thus, by Lemma 20, K_4 is immersed in G'. ▮

LEMMA 22. *In a pruned candidate graph, G, suppose vertex v has exactly two neighbors, u and w, and suppose uv has multiplicity at least* 3. *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. Vertex u must have some neighbor not in $\{v, w\}$ to play the role of x, else w would be a cut point, contradicting the biconnectivity of G. Because G is pruned, edge vw must have multiplicity at least 2, and the degree of u must be at least two more than the multiplicity of uv. Thus in G', the graph that results from shorting v, the degree of u is at least 4. We conclude from Lemma 13 that the M model illustrated in Fig. 11a is immersed in G', and the graph shown in Fig. 11b, which contains K_4, is immersed in G. ▮

LEMMA 23. *In a candidate graph, G, suppose vertex v has exactly two neighbors, u and w, suppose uv and vw each have multiplicity at least* 2, *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 12, with w playing the role of v and x playing the role of w. Thus at least one of the graphs shown in Fig. 12, both of which contain K_4, is immersed in G. ▮

LEMMA 24. *Let G denote a pruned candidate graph. K_4 is immersed in G if and only if G has a vertex of degree* 5 *or more.*

Proof. We know from Lemma 19 that a candidate graph of maximum degree 4 contains no K_4. To prove the converse, we proceed by contradiction and we assume H denotes a minimal pruned candidate graph, with at

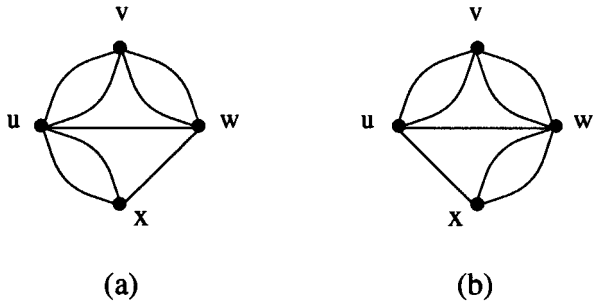

FIG. 12. Graphs used in the proof of Lemma 23.

least one vertex of degree 5 or more, but with no immersed K_4. We observe that H must contain more than four vertices. Otherwise, let a denote a vertex in H with degree 5 or more. If a has only two neighbors, then the graph of Fig. 13a, which contains K_4, is a subgraph of H, contradicting our assumption that H contains no K_4. If a has three neighbors, then the graph of Fig. 13b, which also contains K_4, is a subgraph of H, leading once more to a contradiction. Thus 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 22 guarantees that v cannot have degree 5 or more. If v has degree 4, Lemma 23 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 3 and, by Lemma 20, u has degree 4 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 provides the proof of the following principal result.

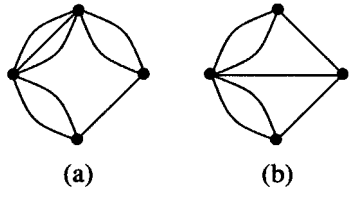

FIG. 13. Pruned four-vertex graphs that contain a vertex of degree 5.

THEOREM 2. *Algorithms* decompose, components *and* test *correctly decide in linear time 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 7 or more
    then $u, v, w, x :=$ spanning-corners(G, v)
    else begin
      prune each biconnected component of G
      $C :=$ a biconnected component with four or more vertices of which at least one has degree 5 or more
      $u, v, w, x :=$ biconnected-corners(C)
      end
  output u, v, w, x
end

We address the correctness of corners. Suppose G contains a cut point v of degree at least 7, after redundant edges incident on vertices with only neighbor are deleted. In this case, we use algorithm spanning-corners to locate the corners.

algorithm spanning-corners(G, v)
input: a three-edge-connected series-parallel multigraph G in which each vertex with exactly one neighbor has degree 3 and a cut point v with degree 7 or more

```
output: the four corners of a K_4 model in G
begin
  if G − {v} has three or more connected components
    then /* Case 1a */
      u, w, x := neighbors of v in G, each in a different connected
                 component of G − {v}
    else begin /* Case 1b */
      C_1, C_2 := the connected components of G − {v}
      A_1 := C_1 augmented with v and the edges it induces
      A_2 := C_2 augmented with v and the edges it induces
      if v has degree 4 or more in A_1
        then A := A_1 and B := A_2
        else A := A_2 and B := A_1
      while v induces no edge of multiplicity 2 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
      u := some vertex in A such that uv has multiplicity at least 2
      if v has no neighbor other than u in A
        then w := any neighbor of u in A other than v
        else w := any neighbor of v in A other than u
      x := any neighbor of v in B
      end
  output u, v, w, x
end
```

If $G - \{v\}$ has three or more connected components (Case 1a), it follows from the three-edge-connectivity of G that a model of the star graph shown in Fig. 1 exists in G, with v playing the role of the central vertex. Any three vertices in $G - \{v\}$ can serve as the remaining corners, as long as no two of them are in the same connected component of $G - \{v\}$. Thus the vertices returned are the corners of a K_4 model.

If $G - \{v\}$ has only two components (Case 1b), then we apply Lemmas 13 and 14. If v has only one neighbor u in the augmented component A (see algorithm spanning-corners), then by Lemma 14, we may choose v, u, and an arbitrary neighbor w of u as the corners of an M model. If v has two or more neighbors, then the corners of M can be found using Lemma 13 as long as an edge of multiplicity 2 or more is incident on v in A (note that v cannot be a cut point in A). As observed in the proof of Lemma 15, if this condition is not initially satisfied, it can easily be forced by deleting and shorting vertices. The corners of the M model serve as three of the four corners of a K_4 model. The vertex x chosen as the fourth corner belongs to the connected component of $G - \{v\}$ that does

not contain the first three corners. By the three-edge-connectivity of G, a model of the triple-edge exists in G, with v and x as corners. Combining this model with the M model, we obtain a model of the graph shown in Fig. 8b. Lifting two pairs of edges in this graph gives us K_4.

If G contains no cut point of degree 7 or more, then biconnected-corners is invoked on some pruned biconnected component of G that contains at least one vertex of degree 5 or more. Lemma 24 implies that such a biconnected component exists, and moreover, that it contains an immersed K_4. This component must contain a vertex with exactly two neighbors (Observation 2). In this event, we employ Lemmas 22 and 23, plus Lemma 25, which follows.

algorithm biconnected-corners(G)
input: a three-edge-connected biconnected pruned series-parallel multigraph G with at least four vertices and with at least one vertex of degree 4 or more
output: the four corners of a K_4 model in G
begin
  while x has not been assigned a value **do**
    begin
      $v :=$ an unmarked vertex with exactly two neighbors
      $u, w :=$ the neighbors of v, with the multiplicity of uv at least that of vw
      if v has degree at least 5, or v has degree 4 and uw exists
        then /* Case 2a */
          $x :=$ any neighbor of u besides v or w
        else if v or u has degree 4
          then short v
          else if uw exists
            then /* Case 2b */
              $x :=$ any neighbor of u other than v or w
            else if there is an edge ua, $a \neq v$, of multiplicity 2 or more
              then /* Case 2c */
                $x := a$
              else if there are two vertices of degree 5 or more
                then short v **else** mark v
    end
  output: u, v, w, x
end

LEMMA 25. *In a candidate graph, G, suppose vertex v has exactly two neighbors, u and w, and suppose uv has multiplicity 2, vw has multiplicity 1, and u has degree at least 5. Let x denote a neighbor of u other than v or w. If*

ux has multiplicity at least 2 *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 4, and either ux has multiplicity at least 2 or uw now does. Then by Lemma 13, there is an M model in G' with corners u, w, and x, and with u the image of M's degree-4 vertex. Thus the graph in Fig. 11b, which contains the desired K_4 model, is immersed in G. ∎

Let v be defined as an algorithm biconnected-corners. The corners of an immersed K_4 can be found if one of the following conditions holds:

- either v has degree at least 5 or v has degree 4 and edge uw exists (Case 2a). Lemma 22 applies in the former situation, and Lemma 23 in the latter situation.
- v has degree 3, u has degree 5 or more, and either edge uw exists or there is an edge ua, $a \neq v$, of multiplicity 2 or more (Cases 2b and c). Lemma 25 applies.

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 5. Accordingly, if one of v's neighbors, u, is the only vertex of degree at least 5, uv has multiplicity 2, 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, because at most one vertex can be so marked and another candidate for v is always available. Finally, we note that continued shorting will never result in a graph with fewer than four vertices. This is because, as observed in the proof of Lemma 24, one of the two graphs shown in Fig. 13 must be a subgraph of any candidate graph that contains exactly four vertices of which at least one has degree 5 or more. Because one of Cases 2a–c apply to any vertex with exactly two neighbors in these two graphs, corners are identified at this point without any vertex being marked or shorted. Of course, it is possible that corners are found before the graph is reduced to four vertices.

Biconnected components and cut points can be found (using a depth-first search) and vertices pruned, in linear time. It takes only linear time to check whether a biconnected component contains a vertex of degree 5 or more. In each iteration of spanning-corners, some vertex with two or fewer neighbors is deleted or shorted. In each iteration of biconnected-corners, some vertex with exactly two neighbors is shorted or marked. Vertices with two or fewer neighbors can be maintained using a queue. Deleting, shorting, or marking such a vertex, and updating the queue and the appropriate edge list require only a constant number of steps. Thus both spanning-corners and biconnected-corners run in linear time, and hence, so does corners.

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
  $G' :=$ the edge-weighted digraph obtained 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 $(t_1, t), \ldots, (t_k, t)$, each of capacity 1
  find a flow of value k from s to t, if such a flow exists
  if there is such a flow **then**
    begin
      for each edge (u, v) in G' **do**
        if both (u, v) and (v, u) have positive flow values
          then $flow((u, v)) := \max\{0, flow((u, v)) - flow((v, u))\}$ and
              $flow((v, u)) := \max\{0, flow((v, u)) - flow((u, v))\}$
      discard from G' any edge without a positive flow
      for $i = 1$ **to** k **do**
        begin
          $p_i' :=$ a path in G' from s to t_i
          $p_i :=$ 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 $p_1, \ldots, p_k$
    end
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. We consider various cases.

Case 1*a*. The K_4 model spans a cut point v, and $G - \{v\}$ has three or more connected components.

Corners u, w, and x are in different connected components of $G - \{v\}$, and each corner is adjacent to v in G. Two paths need to be found between each corner and v, to complete a model of the star graph in Fig. 1 and hence a K_4 model. So three calls are made to paths, each with v

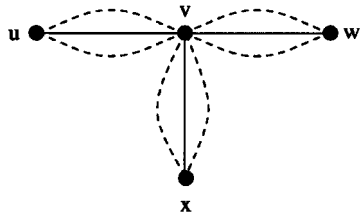

FIG. 14. Paths to be found if $G - \{v\}$ has three or more connected components.

playing the role of s and k set to 2. See Fig. 14. The first call uses $u = t_1 = t_2$; the second call uses $w = t_1 = t_2$; the third call uses $x = t_1 = t_2$.

Case 1*b*. The K_4 model spans a cut point v, and $G - \{v\}$ has only two connected components.

See Fig. 15. Figure 15a depicts the case when the biconnected component of G containing both u and v has at least one corner. In this case, corners u, w, and x are all adjacent to v. Moreover, edge uv has multiplicity 2 or more. Note that v separates x from u and w. To complete a model of the graph shown in Fig. 8b, two paths between v and x in the biconnected component containing v and x, and two paths between w and $\{u, v\}$ in the biconnected component containing u, v, and w must be found. Thus two calls to paths are required. The first call uses $w = s$, $u = t_1$, and $v = t_2$; the second call uses $v = s$ and $w = t_1 = t_2$. Figure 15b depicts the case when u and v are in a biconnected component by themselves. Corner x is adjacent to v and corner w is adjacent to u. Again, two calls to paths are required. The first call uses $u = s$, $w = t_1 = t_2$; the second call uses $v = s$ and $x = t_1 = t_2$.

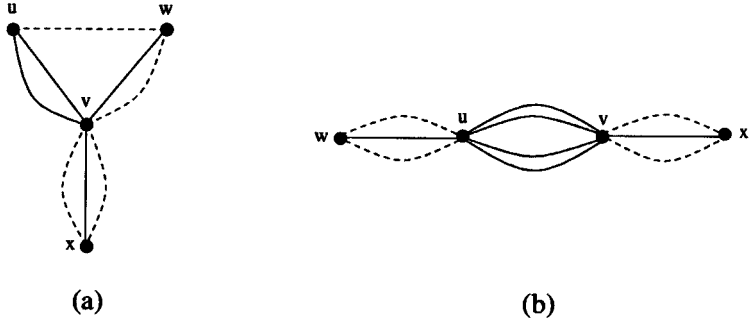

FIG. 15. Paths to be found if $G - \{v\}$ has only two connected components.

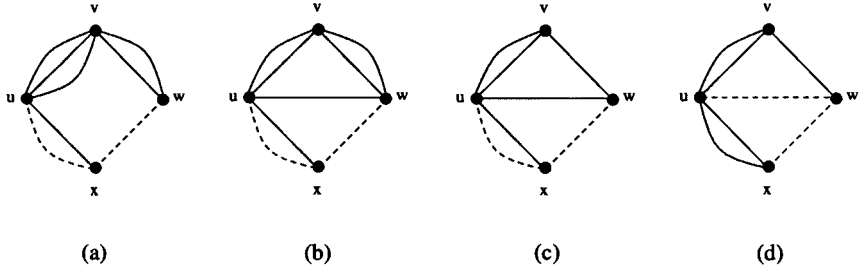

FIG. 16. Paths to be found if the K_4 model lies in a biconnected component.

Cases 2*a*–*c*. The K_4 model is in a single biconnected component. See Fig. 16. Case 2a of algorithm corners is illustrated in Figs. 16a and b. Cases 2b and c are illustrated in Figs. 16c and d, respectively. In each case, one call to paths, with k set to two, suffices.

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 breadth-first search) 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 [6], 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.

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

THEOREM 3. *If K_4 is immersed in an arbitrary series-parallel graph, algorithms* corners *and* paths *correctly isolate a K_4 model in linear time*.

5. DISCUSSION

We have presented linear-time methods to detect if K_4 is immersed in an input graph, and to isolate a K_4 model if any exist. We implemented our algorithms in C and we ran them on a SUN SPARCstation 20. Experiments on randomly generated graphs indicate that our algorithms are practical, taking 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 we 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.

We note that the detection algorithm can be efficiently parallelized. Biconnected components and cut points can be found in $O(\log n)$ time on a CRCW PRAM with $O((m + n)\alpha(m, n)/\log n)$ processors [10], 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 [11]. Recalling that graphs of interest have at most a linear number of edges, a parallel version of decompose thus needs at most $O(\log^2 n)$ time with $O(n)$ processors. The triconnected components algorithm of [10], modified slightly to find three-edge-connected components [18], 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. Besides finding cut points, which are available from components, the only operations in test are pruning vertices with just two neighbors and deleting edges incident on vertices with only one neighbor. Both can be accomplished in constant time with $O(n)$ processors. Thus 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.

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, [7, 12]). 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 3. The absence of K_4, however, merely approximates its cutwidth at 3. In particular, such an absence says nothing at all about how to find a layout of width 3 even if many should exist. To solve this problem, our algorithms can be used in conjunction with previously studied "self-reduction" techniques [1, 9] 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* [8] 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 [16, 17].

ACKNOWLEDGMENT

We thank an anonymous referee, whose careful review of our original submission helped us to clarify the presentation of these results.

REFERENCES

1. D. J. Brown, M. R. Fellows, and M. A. Langston, Polynomial-time self-reducibility: Theoretical motivations and practical results, *Internat. J. Comput. Math.* **31** (1989), 1–9.
2. H. L. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth, *in* "Proceedings Twenty-Fifth Annual ACM Symposium on Theory of Computing," 1993, pp. 226–234.
3. G. A. Dirac, In abstrakten graphen vorhandene vollstandige 4-graphen und ihre unterteilungen, *Math. Nachr.* **22** (1960), 61–85.
4. R. J. Duffin, Topology of series-parallel networks, *J. Math. Anal. Appl.* **10** (1965), 303–318.
5. S. Even and R. E. Tarjan, Computing an *st*-numbering, *Theoret. Comput. Sci.* **2** (1976), 339–344.
6. L. R. Ford and D. R. Fulkerson, "Flows in Networks," Princeton Univ. Press, Princeton, NJ, 1974.
7. T. Fujii, H. Horikawa, T. Kikuno, and N. Yoshida, A heuristic algorithm for gate assignment in one-dimensional array approach, *IEEE Trans. Computer-Aided Design* **6** (1987), 159–164.
8. M. R. Fellows and M. A. Langston, On well-partial-order theory and its application to combinatorial problems of VLSI design, *SIAM J. Discrete Math.* **5** (1992), 117–126.
9. M. R. Fellows and M. A. Langston, On search, decision and the efficiency of polynomial-time algorithms, *J. Comput. Syst. Sci.* **49** (1994), 769–779.
10. D. Fussell, V. Ramachandran, and R. Thurimella, Finding triconnected components by local replacements, *in* "Proceedings Sixteenth International Colloquium on Automata, Languages and Programming," vol. 372, 1989, pp. 379–393.
11. X. He, Efficient parallel algorithms for series parallel graphs, *J. Algorithms* **12** (1991), 409–430.
12. Y.-S. Hong, K.-H. Park, and M. Kim, Heuristic algorithms for ordering the columns in one-dimensional logic arrays, *IEEE Trans. Computer-Aided Design* **8** (1989), 547–562.
13. E. Korach and A. Tal, General vertex disjoint paths in series-parallel graphs, *Discrete Appl. Math.* **41** (1993), 147–164.

14. A. Lempel, S. Even, and I. Cederbaum, An algorithm for planarity testing of graphs, *in* ''Theory of Graphs: International Symposium'' (P. Rosenstiehl, Ed.), pp. 215–232, Gordon and Breach, New York, 1967.
15. P. C. Liu and R. C. Geldmacher, An $O(\max(m, n))$ algorithm for finding a subgraph homeomorphic to K_4, *Congr. Numer.* **29** (1980), 597–609.
16. M. A. Langston and S. Ramachandramurthi, Dense layouts for series-parallel circuits, *in* ''Proceedings, First Great Lakes Symposium on VLSI,'' 1991, pp. 14–17.
17. P. J. McGuinness and A. E. Kezdy, An algorithm to find a K_5 minor, *in* ''Proceedings, Third ACM–SIAM Symposium on Discrete Algorithms,'' 1992, pp. 345–356.
18. V. Ramachandran, private communication.
19. N. Robertson and P. D. Seymour, Graph minors XIII. The disjoint paths problem, *J. Combin. Theory Ser. B* **63** (1995), 65–110.
20. R. E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.* **1** (1972), 146–159.
21. J. Valdes, R. E. Tarjan, and E. Lawler, The recognition of series-parallel digraphs, *SIAM J. Comput.* **11** (1982), 298–313.