Parallel Depth First Search in $K_{3,3}$-free Graphs and K_5-free Graphs

by

B. Narendran

Computer Sciences Technical Report #993

January 1991

Parallel Depth First Search in $K_{3,3}$-free Graphs and K_5-free Graphs

B. Narendran

Computer Sciences Department
University of Wisconsin-Madison
Madison, Wisconsin-53706.

Abstract

Parallel algorithms for depth first search on two classes of graphs are presented. A new algorithm for the class of graphs that do not contain subgraphs homeomorphic to $K_{3,3}$ is presented that takes $O(\log^2 n)$ time using $O(n)$ processors. For the class of graphs that do not contain a subgraph homeomorphic to K_5, we present an algorithm that takes $O(\log^3 n)$ time using $O(n^2)$ processors. The model of computation assumed is a CRCW PRAM.

1 Introduction

Efficient parallel depth first search has been an open problem for a long time. Reif [Rei85] showed that constructing a lexicographically first depth-first -search tree is P-complete. Aggarwal and Anderson [AA87] presented an RNC algorithm for DFS on general graphs that ran in $O(\log^5 n)$ time using $O(n^{4.5})$ processors.

Efficient algorithms are known for special cases of graphs. For planar graphs, DFS can be done in $O(\log^2 n)$ time [Smi86, HY88], with a linear number of processors. Khuller [Khu88a] has a parallel algorithm for graphs that do not contain subgraphs that are homeomorphic to $K_{3,3}$ ($K_{3,3}$-free graphs) which runs in $O(\log^3 n)$ time using a linear number of processors. Here, we present an improved algorithm for these graphs that takes $O(\log^2 n)$ time with the same processor bound. In addition to being more efficient asymptotically, our algorithm is somewhat simpler than the one in [Khu88a]. We also present a parallel algorithm for the depth-first search problem on

graphs that do not contain a subgraph that is homeomorphic to K_5 (K_5-free graphs). This is the first efficient parallel algorithm for this class of graphs.

Our algorithms use characterizations of $K_{3,3}$-free and K_5-free graphs due to Hall [Hal43] and Wagner [Wag85], that were also used in [Vaz89, Khu88a, Khu88b]. These characterizations allow the decomposition of such graphs into simpler structures which can be processed independently and the results are combined in an efficient manner.

In Section 2 , we define terms that are used in this paper. Section 3 discusses the characterization of $K_{3,3}$-free graphs. Section 4 discusses the algorithm for $K_{3,3}$-free graphs and analyzes its time and processor requirements. Sections 5 and 6 similarly deal with K_5-free graphs.

2 Definitions

Let $G(V, E)$ be a connected undirected simple graph. Let $S \subseteq V$. Define $G \setminus S$ to be the subgraph of G obtained by deleting from G all the vertices of S and all the edges incident on vertices in S.

A vertex u of G is said to be a *cut-vertex* of G if $G \setminus \{u\}$ is not connected. A connected graph is *biconnected* if it has no cut-vertices. A maximal biconnected subgraph of G is called a *biconnected component* of G.

Let $G(V, E)$ be a biconnected graph. A a pair of vertices $u, v \in V$ is a *cut-pair* of G if $G \setminus \{u, v\}$ is disconnected. In this case, let $\{G'_1, G'_2, G'_3, \ldots G'_k\}$ be the set of k connected subgraphs in $G \setminus \{u, v\}$. Let G_i be the subgraph induced on $V(G'_i) \cup \{u, v\}$, with the edge (u, v) added, if it doesn't already exist in G. The G_i are called the split components of G w.r.t. the cut-pair $\{u, v\}$.

A graph G is *triconnected* if it has no cut-pairs. For a graph G that is not triconnected, the *triconnected components* are defined as follows. Find a cut-pair {u,v} and split G into its split graphs w.r.t. {u,v}. Recursively continue to split the components obtained until no component has a cut-pair. The resultant triconnected graphs are the triconnected components of G. This decomposition is unique and any two triconnected components share at most two vertices [Tut66].

In a similar fashion, if $G(V, E)$ is a triconnected graph, a set of three vertices $\{u, v, w\} \subset V(G)$ is said to be a *cut-triplet* if $G \setminus \{u, v, w\}$ is disconnected. *Four-connected* graphs and *four-connected components* of a graph are defined similarly in the natural way.

Given a graph G, let E_1 and E_2 be a partition of the edge set $E(G)$. Let

V_1 and V_2 be the subsets of $V(G)$ induced naturally by the edge sets E_1 and E_2 respectively. Then $(G_1(V_1, E_1), G_2(V_2, E_2))$ is said to be a *separation* of G. Let (G_1, G_2) be a separation of G. Let H_i be the graph obtained from G_i by adding new edges between every pair of vertices in $V(G_1 \cap G_2)$. Then, G is said to be the *clique-sum* $H_1 \circ H_2$, of H_1 and H_2. If $|V(H_1) \cap V(H_2)| \leq k$, then $H_1 \circ H_2$ is said to be a $\leq k$ *clique-sum* of H_1 and H_2. If $\mathcal{C}$ is a class of graphs, the set of graphs obtained by repeatedly taking $\leq k$ *clique-sums* of graphs in $\mathcal{C}$ is denoted by $\langle \mathcal{C} \rangle_k$.

3 A Characterization of $K_{3,3}$-free Graphs

The following lemma is due to Hall [Hal43].

Lemma 3.1 (Hall) *A graph G has no subgraph homeomorphic to $K_{3,3}$ iff every triconnected component of G is either planar or the graph K_5.*

While Lemma 3.1 gives us a decomposition of $K_{3,3}$ free graphs into "simple" components, the following property of the decomposition says that the components are connected in a particularly simple structure that facilitates the components being handled in parallel.

Lemma 3.2 *Let G be a graph and let C be the set of triconnected components of G and P be the set of cut-pairs in this decomposition of G. Define the graph H on $C \cup P$ s.t. (c, p) is an edge of H iff $c \in C, p \in P$ and the pair of vertices p are in the component c. Then, H is a tree.*

Lemmas 3.1 and 3.2 were used in [Khu88a, Vaz89] to develop parallel algorithms for several graph problems on $K_{3,3}$ graphs.

We will call the tree H of Lemma 3.2 the *decomposition tree* of G. For purposes of description, we define another auxiliary tree H' of the triconnected components alone where, an edge between c_i and c_j exists iff they share a cut-pair of vertices. Clearly, H' is obtained from H by collapsing the nodes corresponding to cut-pairs into their parents. We will call H' the *component tree* of G. Each component c (except the root) in this component tree has a unique cut-pair that separates it from its parent. We will call this cut-pair the parent cut-pair of c.

4 The Algorithm for $K_{3,3}$-free Graphs

Miller and Ramachandran [MR87] give a parallel algorithm to obtain triconnected components of a graph that runs in $O(\log^2 n)$ time using $O(n)$

processors. This algorithm can be used to construct the tree of triconnected components and cut-pairs described in Lemma 3.2.

For planar graphs, He and Yesha [HY88] give an $O(\log^2 n)$ time parallel algorithm for depth first search that uses a linear number of processors. This algorithm proceeds by first finding a separating path in the planar graph, i.e. a path whose removal disconnects the graph into components which are smaller than the original graph by at least a constant factor. This path becomes a path in the DFS-tree and the DFS-trees of the remaining components are computed recursively and spliced onto this path. The algorithm of Khuller [Khu88a] works in a similar fashion for $K_{3,3}$ graphs and uses Lemmas 3.1 and 3.2 to efficiently compute a separating path in such graphs.

Our algorithm proceeds by generating the DFS trees for each of the triconnected components in parallel. This can be done efficiently, since, by Lemma 3.1, the triconnected components are either planar or exactly K_5. Then, these trees are put together by using the simple structure afforded by Lemma 3.2. This merging is done by merging adjacent pairs of levels in the tree of components H in parallel, i.e. in the first step, levels $2i-1$ and $2i$, $i = 1, 2, 3, \ldots$, are merged to obtain a tree of half the depth. This is continued recursively giving an $O(\log n)$ stage procedure. We show below that we can do each of these merging steps in constant time.

At each stage, each node of the current tree represents a cluster of the initial triconnected components, whose DFS tree has been generated. Actually, our algorithm computes and maintains four different DFS trees (with specific properties) for each node (except the root cluster) in the current component tree. This is to ensure that the next merging step can be performed efficiently. Thus, when a cluster $\mathcal{C}$ is merged with one of its children $\mathcal{D}$, each of $\mathcal{C}$'s DFS trees $T_{\mathcal{C}}^{(k)}, k = 1, \ldots, 4$ is merged with exactly one of $\mathcal{D}$'s DFS trees $T_{\mathcal{D}}^{(k)}$, the choice being made based on which $T_{\mathcal{D}}^{(k)}$ is "easy" to handle. At the end of this merging then, this again leaves behind four DFS trees of the newly formed cluster.

We now define the structure of these four DFS trees. Let $\mathcal{C}$ be a cluster. Recall that $\mathcal{C}$ represents a subtree of the original tree of triconnected components consisting of a root component C_r and all its descendants up to a particular level. Let (u, v) be the parent cut-pair of C_r. Define the following DFS trees :

i) $T_{\mathcal{C}}^{(-,u)}$, a DFS tree of $\mathcal{C} \setminus \{v\}$, rooted at u.

ii) $T_C^{(-,v)}$, a DFS tree of $C \setminus \{u\}$, rooted at v.

iii) $T_C^{(u,v)}$, a DFS tree of C with edge (u,v) removed, rooted at u.

iv) $T_C^{(v,u)}$, a DFS tree of C with edge (u,v) removed, rooted at v.

The structure of these trees is illustrated in Fig. 1.

If $T_C^{(x,y)}$ is one such DFS tree of C, we will call it a (x,y)–DFS tree of C. Note that $T_C^{(-,u)}$ and $T_C^{(-,v)}$ do not contain all the vertices of C. The fact that this definition is consistent follows from the fact that C is a biconnected component, and hence $C \setminus \{u\}$ and $C \setminus \{u\}$ are still connected graphs and can be covered by a single DFS tree.

The motivation for the above definitions is the following lemma, which essentially describes how to combine the DFS trees of two clusters.

Lemma 4.3 *Let C be a cluster and $\mathcal{D}$ its parent cluster in the current tree. Let (u,v) be the cut-pair that separates these clusters. Let (u_1,v_1) be the copies of (u,v) in $\mathcal{D}$ and (u_2,v_2) its copies in C. Let $T_{\mathcal{D}}^{(x,y)}$ be any (x,y)–DFS tree for $\mathcal{D}$, where $(x,y) \in \{(u_1,v_1),(v_1,u_1),(-,u_1),(-,v_1)\}$. Let $T_C^{(-,u_2)}$,$T_C^{(-,v_2)}$, $T_C^{(u_2,v_2)}$ and $T_C^{(v_2,u_2)}$ be the DFS trees of C as defined above. Then,*

i) *If u_1 and v_1 are both in $T_{\mathcal{D}}^{(x,y)}$, then u_1 and v_1 are on a common path from the root of $T_{\mathcal{D}}^{(x,y)}$.*

ii) *Suppose either $u_1 \notin T_{\mathcal{D}}^{(x,y)}$ or u_1 is an ancestor of v_1 in $T_{\mathcal{D}}^{(x,y)}$.*

- **(a)** *If either $(u_1,v_1) \notin T_{\mathcal{D}}^{(x,y)}$ or $(u_1,v_1) \in G$, then, $E(T_{\mathcal{D}}^{(x,y)}) \cup E(T_C^{(-,v_2)})$ forms an (x,y)–DFS tree for $C \cup \mathcal{D}$.*
- **(b)** *If $(u_1,v_1) \in T_{\mathcal{D}}^{(x,y)}$, but $(u_1,v_1) \notin G$, then, $E(T_C^{(u_2,v_2)}) \cup E(T_{\mathcal{D}}^{(x,y)}) \setminus \{(u_1,v_1)\}$ is an (x,y)–DFS tree for $C \cup \mathcal{D}$.*

Proof:

i) Since (u,v) is a cut-pair that separates C and $\mathcal{D}$, $\mathcal{D}$ contains the virtual edge (u_1,v_1). Thus, any DFS tree $T_{\mathcal{D}}$ of $\mathcal{D}$ that contains both u_1 and v_1 is constrained to have u_1 and v_1 on a common path from the root.

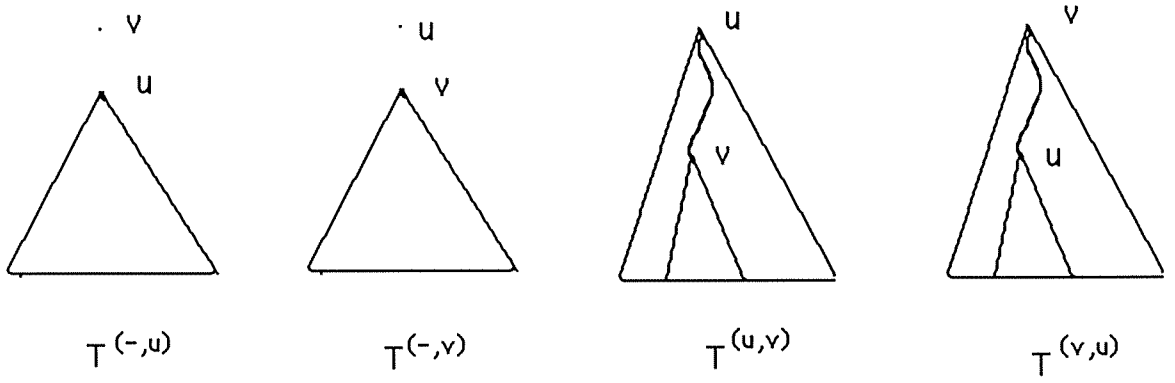

Fig. 1.

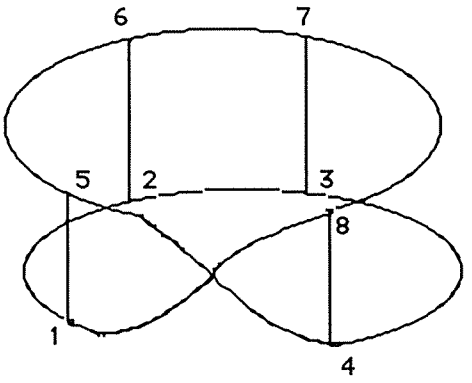

Fig. 2. The four-rung Mobius ladder

ii) **(a)** In this case, by identifying the vertices v_1 and v_2 and making $T_{\mathcal{C}}^{(-,v_2)}$ a subtree of $T_{\mathcal{D}}$ at v_1, we get a tree that spans $\mathcal{C} \cup \mathcal{D}$. Since the additional non-tree edges introduced by this operation go only from vertices of $T_{\mathcal{C}}^{(-,v_2)}$, to the vertex u_1, which is an ancestor of $T_{\mathcal{C}}^{(-,v_2)}$ (either currently or potentially), the DFS-tree property is retained. We will call this operation *grafting.*

(b) Here, the tree $T_{\mathcal{D}}$ contains the illegal edge (u_1, v_1), which can be replaced by a part of the path from the root of $T_{\mathcal{C}}^{(u_2,v_2)}$. Since in this *splicing* step, the relative ancestor–descendent relationships of the nodes are not disturbed, the DFS property is again retained in the merged tree.

■

The process of *grafting* and *splicing* can be done in constant time using a linear number of processors (linear in $|\mathcal{C}| + |\mathcal{D}|$). It remains to show how to efficiently decide the ancestor–descendent relationship between u_1 and v_1 in $T_{\mathcal{D}}$. Observe that the ancestor–descendent relationship between the nodes in a DFS tree within any triconnected component is an invariant and is unchanged by *grafting* or *splicing.* Thus it suffices to establish this relationship at the beginning for every DFS tree constructed and subsequently keep track of which trees are currently active at any stage.

Specifically, initially each cluster has four distinct DFS trees associated with it. At any stage, when two clusters are merged, an appropriate form of the child cluster's DFS tree is chosen and merged with each of the four types of the parent's DFS tree. Thus, at any stage, for every initial triconnected component, there are four versions of its DFS tree extant (perhaps in a form merged with other components to form a cluster). Each triconnected component thus maintains the following information about itself throughout the course of the algorithm : the identity of the cluster it is currently contained in, and the forms of its four DFS trees represented as an ordered list $\langle (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4) \rangle$, where each $(x_i, y_i) \in \{(u, v), (v, u), (-, u), (-, v)\}$. This list is easily updated on merges by observing which versions of its cluster were merged with the parent cluster.

The following segment of pseudo-code summarizes the above description of the merging of a cluster $\mathcal{C}$ into its parent cluster :

Procedure *Merge* ($\mathcal{C}$);
begin C_r := the root component of cluster $\mathcal{C}$;
$\mathcal{D}$:= parent cluster of $\mathcal{C}$;
C_p := the parent of C_r in the component tree H;
Let (u, v) be the cut-pair that separates C_r and C_p;
For each currently extant DFS tree $T_{\mathcal{D}}^{(x,y)}$ of $\mathcal{D}$ do
/* WLOG, assume that either u is a parent of v in $T_{\mathcal{D}}^{(x,y)}$ or $u \notin T_{\mathcal{D}}^{(x,y)}$ */
if $(u, v) \notin T_{C_p}^{(x,y)}$ or $(u, v) \in G$ **then**
graft $T_{\mathcal{C}}^{(-,v)}$ onto $T_{\mathcal{D}}$;
else
splice $T_{\mathcal{C}}^{(u,v)}$ onto $T_{\mathcal{D}}$;
end

We now consider how this merging can be done in parallel at each stage of the algorithm. Clearly, it suffices to look at merging two adjacent levels, since the other pairs of levels can be merged independently in parallel. Let l and $l + 1$ be the levels of the cluster tree that are to be merged. Let $\{\mathcal{C}_0, \mathcal{C}_1, \ldots\}$ be the set of clusters at level l and let $\{\mathcal{C}_{i,1}, \mathcal{C}_{i,2}, \ldots\}$ be the children of $\mathcal{C}_i$. Again, it clearly suffices to look at any one $\mathcal{C}_i$ and its children. Also, if two child clusters have distinct cut-pairs through which they are attached to the parent cluster, they can also be handled independently.

Therefore, we need to worry about only the case where more than one cluster has the same parent cut-pair, say (u, v). To handle this case, we initially choose, in the decomposition tree H, a distinguished child cluster as a *normalizer*. This cluster proceeds to merge into its parent as described before, and its siblings can then proceed by performing a *graft* operation, irrespective of the status of the edge (u, v) in the parent tree, since the state has been *normalized* by the distinguished child. Thus, we avoid the situation where more than one child may attempt a *graft* operation on the same edge o the parent tree. Clearly, the *normalizers* can be chosen in constant time and marked as such after the decomposition tree H has been constructed.

The running time of the above algorithm is as follows. The initial decomposition of the $K_{3,3}$-free graph into triconnected components and the construction of the decomposition tree can be done in $O(\log^2 n)$ time using

$O(n)$ processors using the algorithm in [MR87]. Generating DFS trees for the planar components is again possible in $O(\log^2 n)$ time by the algorithm in [HY88]. The labeling of the vertices to determine the initial ancestor–descendent relationships can also be done at the same time. As explained before, the individual DFS trees can then be merged in $O(\log n)$ stages, with each stage taking a constant time and using $O(n)$ processors. Thus, the overall running time is dominated by the preprocessing step, giving an $O(\log^2 n)$ algorithm.

5 A Characterization of K_5-free Graphs

The following lemma is due to Wagner [Wag85], and was used in [Khu88b].

Lemma 5.4 (Wagner) *Let C be the class of all planar graphs along with the "four-rung Möbius ladder" (see Fig. 2). Then, $\langle C \rangle_3$ is the class of all K_5-free graphs.*

Corollary 1 *Let G be a triconnected K_5-free graph. Then, its four-connected components are planar.*

Proof: Since G is triconnected, it was generated by taking 3-clique sums only, since $\leq$ 2-clique-sums would give rise to either a cut-vertex or a cut-pair. Since the "four-rung Möbius ladder" does not have K_3 as a subgraph, it could not have participated in a 3-clique-sum operation. Hence G must have been generated by 3-clique-sum operations on planar graphs only. The four-connected components of G are all subgraphs of these planar graphs. ∎

As in the case of the triconnected components of $K_{3,3}$-free graphs, we have the following lemma that says that the four-connected components are linked in a "nice" fashion.

Lemma 5.5 *Let G be a triconnected K_5-free graph and let C be the set of four-connected components of G and P be the set of cut-triples in this decomposition of G. Define the graph H on $C \cup P$ s.t. (c, p) is an edge of H iff $c \in C, p \in P$ and the triple of vertices p are in the component c. Then, H is a tree.*

Lemmas 5.4 and 5.5 were used in [Khu88b] to develop parallel algorithms for some graph problems on K_5-free graphs.

We will again refer to the tree H of Lemma 5.5 as the *decomposition tree* of G, and define the auxiliary *component tree* H' as in the case of $K_{3,3}$-free graphs. Each component c (except the root) in this component tree has a unique cut-triple that separates it from its parent. We will call this cut-triple the parent cut-triple of c.

6 The Algorithm for K_5-free graphs

We will assume that the graph G is biconnected, since otherwise we can solve the problem for each biconnected component and put the resulting DFS trees together. Since, by Lemma 3.2, the triconnected components of *any* graph form a tree, we can proceed essentially as in the previously described algorithm for $K_{3,3}$-free graphs, i.e., we decompose G into a tree of triconnected components, compute a set of DFS trees for each component and merge them in $O(\log n)$ steps. The only difference now is that the triconnected components may no longer be planar or "small". We can use the parallel planarity testing algorithm of [RR89] that runs in logarithmic time using a linear number of processors to identify the planar components and use the algorithm in [HY88] to compute the DFS trees for them. In what follows, we describe how the non-planar components are handled.

As in [Khu88b], we use Lemmas 5.4 and 5.5 to decompose the given triconnected non-planar graph into a tree of 4-connected components, which are now planar. This can be done in $O(\log n)$ time and $O(n^2)$ processors using the algorithm described in [KR87]. Recall that these 4-connected components will have some extra *virtual* edges that were not present in the original graph G. Having obtained this decomposition, we proceed in a manner similar to the previous case. As before, for each four-connected component, we generate a small set of DFS-trees constrained in a manner that enables easy merging. Then, the DFS-trees of components at adjacent odd-even levels are merged, giving a *cluster* tree of half the height. This process is repeated until we are left with a single cluster and a DFS-tree corresponding to it. In this case though, since the interface between two four-connected components is more complicated (cut-triples rather than cut-pairs), we need to generate a larger number of DFS-trees per component and, in addition, ensuring that the DFS-trees satisfy the necessary constraints requires more work than in the case of the triconnected components. We now describe this process.

Our first step is to augment each four-connected component C in the

following manner :

For each four-connected component C in the component tree H' **do**

Step 1 Delete any virtual edges induced on C's parent cut-triple.

Step 2 For each cut-triple $\{u, v, w\}$ that separates C from a child component, if there is is only one child component, linked to C through this triple, remove any virtual edges that were added to form the 3-clique $\{u, v, w\}$, and add a new vertex x and the edges $\{(u, x), (v, x), (w, x)\}$ to C.

Step 3 For each cut-triple $\{u, v, w\}$ that separates C from a child component, if there are two or more child components linked to C through this triple, then add the edges of the 3-clique $\{u, v, w\}$ in C.

Henceforth, by a *component* of G, we will mean a component that has been augmented as described above. Observe that this augmentation deletes some of the virtual edges and adds a few virtual vertices. We will call the original (non-virtual) vertices and edges of G as *real* vertices and edges. Each virtual vertex is said to be *associated* with the corresponding unique cut-triple of G and each virtual edge is *associated* with one or more cut-triples of G. The following lemma verifies that the above augmentation does not destroy the planarity and connectivity properties of the four-connected components.

Lemma 6.6 *Let C be a four-connected component of G that has been augmented as above. Then,*

i) *C is planar.*

ii) *If $\{u, v, w\}$ is the parent cut-triple of C, then $C \setminus \{u\}$ and $C \setminus \{u, v\}$ are both connected graphs.*

Proof:

i) Since the original four-connected graphs are planar, removing some of the virtual edges in Step 1 and adding some of them back in Step 3 clearly retains planarity. In Step 2, we replaced some 3-cliques with a "3-star" (see Fig. 3) with an additional virtual vertex. To see that this does not destroy planarity, observe that in the original planar 4-connected component, the edges of the 3-clique must have formed a face in any planar embedding. Otherwise, removing the vertices of the

3-clique would have disconnected the component, thus contradicting its four-connectivity. Thus, we have a face available to us in which we can embed the new virtual vertex and its associated edges.

ii) To see that removing two vertices of the parent cut-triple does not disconnect $\mathcal{C}$, observe that since the original (unmodified) component was 4-connected, removal of any two vertices still leaves it connected. The first step of the augmentation (removal of edges on $\{u, v, w\}$) is subsumed by removing this pair of vertices. Also, the second step is easily seen to retain any connectivity, since a deleted edge (x, y) is always replaced by a path (x, z, y), where z is a virtual vertex.

To see that $\mathcal{C} \setminus \{u\}$ is connected, we simply have to show that v has at least one neighbour in $\mathcal{C} \setminus \{u, v\}$. This is clearly true, as in the unmodified component, v had at least 4 neighbours, and Step 1 of the augmentation process removes at most two of them, and any modifications of step 2 replaces two neighbours with one virtual neighbour.

∎

We define an *aggregate* of G to be the union of some collection of components $\{C_1, C_2, \ldots C_k\}$ of G that are connected in H'. Every aggregate $\mathcal{A}$ thus has a unique root component, and the parent cut-triple of $\mathcal{A}$ is defined to be the parent cut-triple of this root component. An aggregate $\mathcal{A}_2$ is said to be a child of aggregate $\mathcal{A}_1$ if the root component of $\mathcal{A}_2$ is the child of some component of $\mathcal{A}_1$. A cut-triple of G that occurs in $\mathcal{A}$ is said to be an external cut-triple of $\mathcal{A}$ if the triple connects $\mathcal{A}$ to a component of G not in $\mathcal{A}$.

A *cluster* C of G is the natural maximal subgraph of some aggregate $\mathcal{A}$ such that every virtual vertex or edge that is in C is associated with some external cut-triple of $\mathcal{A}$. Thus, in particular, every component of G is a cluster of G and G itself is a cluster of G corresponding to the aggregate that includes all components of G. The parent and child cut-triples and clusters of a given cluster are defined naturally from those induced by the corresponding aggregates.

A DFS-tree $T_{\mathcal{C}}$ on a cluster $\mathcal{C}$ is said to be well-constrained if it satisfies the following properties :

i) If $\{u, v, w\}$ is an external cut-triple of $\mathcal{C}$ that connects $\mathcal{C}$ to a child cluster, then u, v and w are either all on a common root to leaf path of $T_{\mathcal{C}}$, or occur in $T_{\mathcal{C}}$ in the "3-star" configuration (Fig. 3).

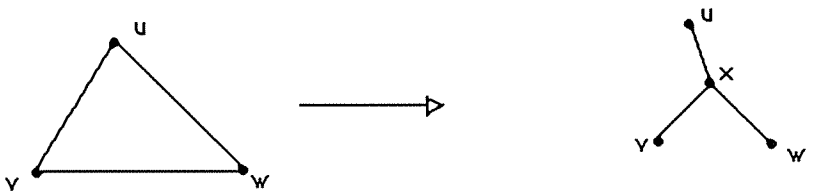

Fig. 3. Replacing the 3-clique with a 3-star

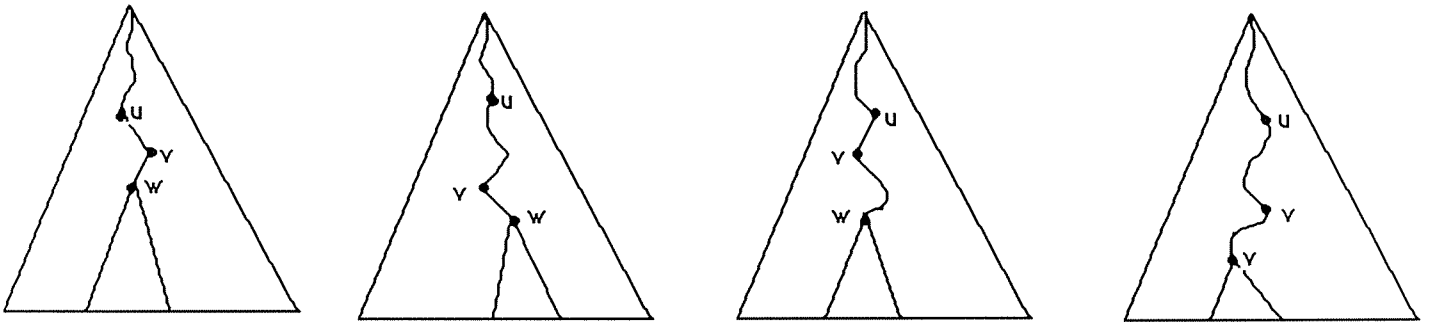

Fig. 4 (a)

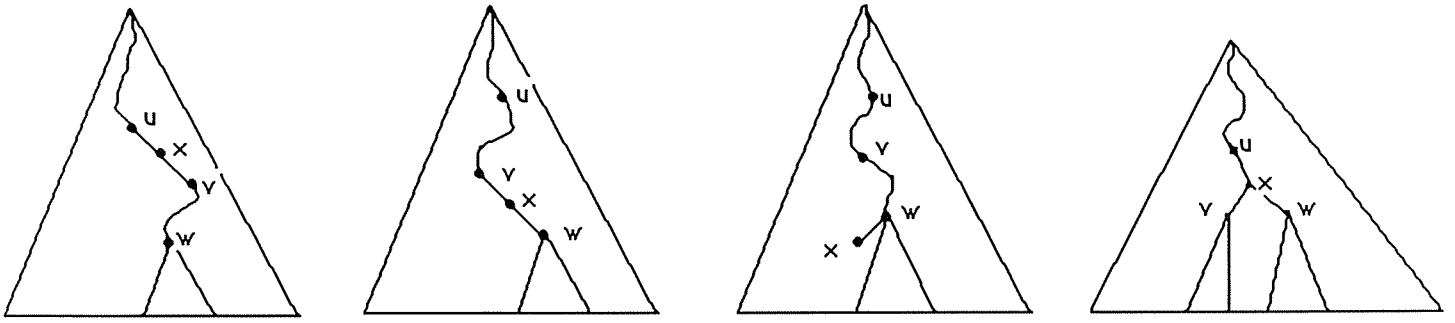

Fig. 4 (b)

Fig 4. The possible orientations of a cut-triple (u,v,w) in a well constrained dfs tree

ii) For every virtual edge (u, v) in $T_{\mathcal{C}}$ where u and v are real vertices, there exists a distinct component of G not contained in $\mathcal{C}$ that is connected to $\mathcal{C}$ through a cut-triple that contains $\{u, v\}$. Similarly, one can associate a distinct component not in $\mathcal{C}$ with each virtual vertex x in $T_{\mathcal{C}}$.

Intuitively, condition (i) above ensures that a potential path in some other component between a pair of vertices of the cut-triple will not affect the depth-first tree property. Condition (ii) on the other hand ensures that the virtual edges and vertices which were introduced (to ensure condition (i)) can be "replaced" by a suitable real path from some other component.

The motivation behind the augmentation of the components that was described earlier is given by the following lemma.

Lemma 6.7 *Let T_C be any DFS search tree on an augmented component C. Then, T_C is well-constrained.*

Proof:

i) Let $\{u, v, w\}$ be an external cut-triple that connects C to a child component. We have two cases to consider. If $\{u, v, w\}$ has two or more children, then the 3-clique on $\{u, v, w\}$ produces the desired constraint. If $\{u, v, w\}$ has only one child, then the "star" configuration constrains the possibilities to those shown in Fig. 4(b).

ii) Each virtual vertex introduced in C is associated with a distinct cut-triple and the single child component of that cut-triple. Virtual edges are introduced in C only as edges on cut-triples that have two or more child components. Since at most two of these 3 edges on any cut-triple can appear in any T_C, we can again associate a distinct external component to each virtual edge of T_C.

∎

Thus, we have ensured that any DFS trees we construct on each individual component are "nice" in the orientations of their child cut-triples. We now describe a small set of orientations of the *parent* cut-triples that can exploit this structure when merging components. The structure of the parent cut-triples will be described by using *path constraints*. Let $\mathcal{C}$ be a cluster and let $\{u, v, w\}$ be its parent cut-triple. A *path-constraint* is specified by describing a path on a subset of the vertices $\{u, v, w\}$ that specifies the order of occurrence of the vertices and, in addition specifies whether the

sub-path between each successive pair of vertices is composed of a single edge (denoted by $u \rightarrow v$) or a sequence of one or more edges of $\mathcal{C}$ (denoted by $u \stackrel{*}{\rightarrow} v$). Thus, for instance, $(u \stackrel{*}{\rightarrow} v \rightarrow w)$ denotes a path p from u to w containing v such that the sub-path of p from u to v is composed of one or more edges of $\mathcal{C}$, whereas the sub-path from v to w consists of a single edge (v, w) (which may or may not exist in $\mathcal{C}$). By this notation, (u) denotes the path consisting of the single vertex u.

A DFS tree of $\mathcal{C}$ that is path constrained by a given path-constraint p is a DFS tree that is rooted at the first vertex of p and contains a path from the root that is consistent with the constraints imposed by p. A p-path constrained DFS tree of $\mathcal{C}$ is denoted by $T_{\mathcal{C}}^{p}$. In particular, $T_{\mathcal{C}}^{(u)}$ denotes simply a DFS tree of $\mathcal{C}$ rooted at u. Observe that a path constraint may require an edge that is not present in $\mathcal{C}$. In this case, we will add the required edge as an additional virtual edge. Note that this addition does not detract from the resulting DFS tree being well-constrained as defined earlier. For each cluster, we will be interested in the following types of path-constrained DFS trees.

i) $T_{\mathcal{C}}^{(u \rightarrow v \rightarrow w)}$

ii) $T_{\mathcal{C}}^{(u \rightarrow v \stackrel{*}{\rightarrow} w)}$

iii) $T_{\mathcal{C}}^{(u \stackrel{*}{\rightarrow} v \rightarrow w)}$

iv) $T_{\mathcal{C}}^{(u)}$

Each of these trees will be called a *version* of a DFS tree of $\mathcal{C}$. The structure of these trees is illustrated in Fig. 5

We now show that constructing these versions of DFS trees for an initial cluster (i.e. component of G) can be done in $O(\log^2 n)$ time using a linear number of processors. We know that the algorithm of [HY88] generates an arbitrary DFS tree of a planar graph rooted at a particular vertex within these time and processor bounds. From Lemma 6.6, we know that each components C is planar. To obtain a path-constrained DFS tree, we simply modify the algorithm in [HY88] as follows. We first generate the path required by the path-constraint. The paths $(u \rightarrow v \rightarrow w)$ and (u) are trivial to generate. For the other two cases, observe that only one sub-path, say (u, v), is required to be a path in C. We first obtain this path by running the spanning tree algorithm of [SV82] on $C \setminus \{w\}$ (which is connected by Lemma

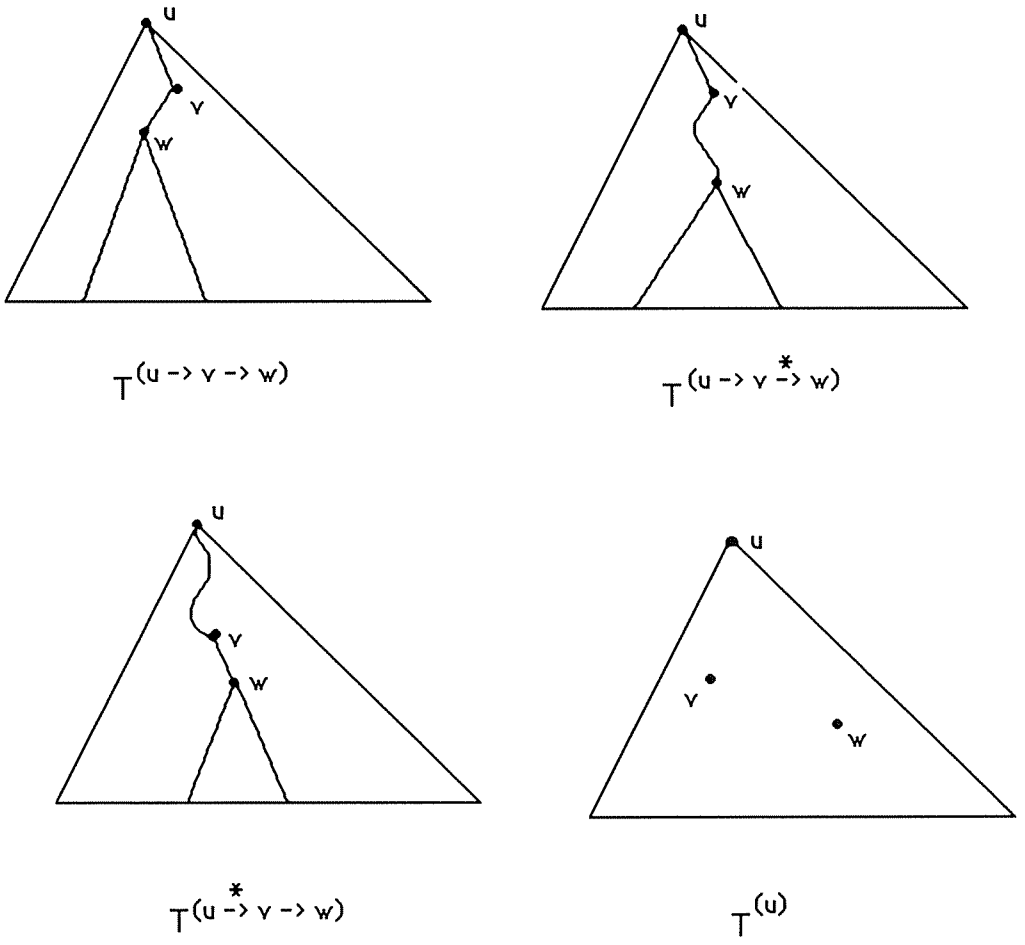

Fig. 5. The structure of the path constrained trees

6.6), and then append the edge (v, w) to it. Once we have the required path, we then proceed as if this was the first path generated by the planar depth first search algorithm. Since the algorithm to generate the path in C takes only $O(\log n)$ time with $O(n)$ processors, and the planar graph depth first search algorithm runs in $O(\log^2 n)$ time, we get the bound on the running time.

As remarked earlier, the reason for constructing these constrained DFS trees is that given a version of a well-constrained DFS tree $T^p_{\mathcal{D}}$ for a cluster $\mathcal{D}$ and a child cluster $\mathcal{C}$, we can always find a version of $\mathcal{C}$'s DFS tree that can be "easily" merged into the tree $T^p_{\mathcal{D}}$, as outlined in Procedure *Merge* in Fig. 6.

Lemma 6.8 *Let $\mathcal{D}$ be a cluster and $T^p_{\mathcal{D}}$ be a version of $\mathcal{D}$'s DFS tree Let $\mathcal{C}$ be a child cluster of $\mathcal{D}$. Then* Merge$(T^p_{\mathcal{D}}, \mathcal{C})$ *returns a well-constrained DFS tree $T^p_{\mathcal{C}\circ\mathcal{D}}$ of $\mathcal{C} \circ \mathcal{D}$.*

Proof: First, it is clearly the case, by the nature of the four-connected components, that all cut-triples in $\mathcal{C} \circ \mathcal{D}$ must have been contained entirely in either $\mathcal{C}$ or $\mathcal{D}$. Also, it is clear that Procedure *Merge* leaves the relative ancestor – descendent relationship between any pair of nodes within each of the trees undisturbed except in the case where the cut-triple $\{u, v, w\}$, across which $\mathcal{C}$ and $\mathcal{D}$ were just merged was oriented in a "star" configuration in $T^p_{\mathcal{D}}$ and the vertices v and w, which were originally the two unrelated vertices of the "star", fall into a different configuration dictated by $T_{\mathcal{C}}$. In this case, let us examine the potential external cut-triples of $\mathcal{C} \circ \mathcal{D}$ that may contain $\{v, w\}$. $\{u, v, w\}$ itself is no longer an external cut-triple of the new cluster, since the fact that it was oriented as a "star" implied that it had only one descendent cluster, namely $\mathcal{C}$ itself, which has already merged into the new cluster. For the same reason, $\{v, w\}$ could not have been part of another cut-triple of $\mathcal{D}$, since then, there would have been a path (either a single edge or a path consisting of a single virtual vertex) between v and w, and thus they could not have been unrelated vertices of the "star". v and w could be a part of some external cut-triple of $\mathcal{C}$, but since $T_{\mathcal{C}}$ was well-constrained to begin with, and v and w are now oriented according to $T_{\mathcal{C}}$, this preserves the well-constrained property.

It is also easily verified that property (ii) of being well-constrained is maintained, as each merge eliminates a virtual vertex or a virtual edge whenever it is possible to do so.

Since the merge procedure ensures that some virtual edge of the parent

Procedure *Merge* $(T^p_{\mathcal{D}}, \mathcal{C})$;
begin
C_r := the root component of cluster $\mathcal{C}$;
C_p := the parent of C_r in the component tree H;
Let (u, v, w) be the cut-triple that separates C_r and C_p;
Let $< u, v, w >$ be the permutation of (u, v, w) such that
$level(u) < level(v) \leq level(w)$ in $T^p_{\mathcal{D}}$;
if C_r is the only child of C_p in H'
then
begin
x := the virtual vertex associated with {u, v, w} in $\mathcal{C}$;
if $\{(x,u),(x,v),(x,w)\} \in T^p_{\mathcal{D}}$ **then**
return $((T^p_{\mathcal{D}} \setminus \{(x,u),(x,v),(x,w)\}) \cup T^{(u)}_{\mathcal{C}})$;
else if $\{(x,u),(x,v)\} \in T^p_{\mathcal{D}}$ **then**
return $((T^p_{\mathcal{D}} \setminus \{(x,u),(x,v)\}) \cup T^{(u \overset{*}{\rightarrow} v \rightarrow w)}_{\mathcal{C}})$;
else if $\{(x,v),(x,w)\} \in T^p_{\mathcal{D}}$ **then**
return $((T^p_{\mathcal{D}} \setminus \{(x,v),(x,w)\}) \cup T^{(u \rightarrow v \overset{*}{\rightarrow} w)}_{\mathcal{C}})$;
else
return $(T^p_{\mathcal{D}} \cup (T^{(u \rightarrow v \rightarrow w)}_{\mathcal{C}} \setminus \{(u,v),(v,w)\}))$;
end
else
begin
if $(u,v) \in T^p_{\mathcal{D}}$ **and** (u,v) is virtual **and** (u,v) is not required by p **then**
return $(T^p_{\mathcal{D}} \cup (T^{(u \overset{*}{\rightarrow} v \rightarrow w)}_{\mathcal{C}} \setminus \{(v,w)\}))$;
else if $(v,w) \in T^p_{\mathcal{D}}$ **and** (v,w) is virtual **and** (v,w) is not required by p **then**
return $(T^p_{\mathcal{D}} \cup (T^{(u \rightarrow v \overset{*}{\rightarrow} w)}_{\mathcal{C}} \setminus \{(u,v)\}))$;
else
return $(T^p_{\mathcal{D}} \cup (T^{(u \rightarrow v \rightarrow w)}_{\mathcal{C}} \setminus \{(u,v),(v,w)\}))$;
end
end.

Figure 6: Procedure *Merge* merges a suitable DFS tree of cluster $\mathcal{C}$ with the path-constrained DFS tree $T^p_{\mathcal{D}}$ of cluster $\mathcal{D}$. It is assumed that the DFS trees are represented as edge sets.

cluster is not required by the path constraint p before trying to splice over it, the resultant tree also satisfies the same path constraint p. ∎

If we are able to establish the relative orientation of the cut-triple (u, v, w) in T_D^p, the rest of the operations in Procedure *Merge* can be done in constant time with a linear number of processors. The previous Lemma implies that the orientation of the cut-triple in T_D^p is the same as it was in the version of T_{C_p} that had merged into T_D^p. Thus, it suffices to establish the orientations of the cut-triples for each version of the trees for each original component, and subsequently keep track of which version has been merged into each cluster.

The above description gives us a sequential algorithm to merge the DFS trees of the various components to obtain a single DFS tree of the entire graph. We start with some version of the root component's tree and successively merge an appropriate versions of adjacent components' trees with it to obtain DFS trees of successively larger clusters.

We now describe the parallel merge procedure. Essentially, our initial clusters are simply the original components of G in the tree H'. At each stage, we merge, in parallel, every cluster at an odd level in the tree with its parent cluster thus reducing the height of the tree of clusters by a factor of two. In $O(\log n)$ stages, we will be left with a single cluster and a DFS tree corresponding to it. By Lemma 6.8, this tree is well defined with respect to the entire cluster, and hence is a valid DFS tree of G. At each stage, a merge consists of merging an appropriate version of a child cluster's tree into each version of the parent cluster's tree as described earlier.

When merging several clusters in parallel, however, we need to ensure that two child clusters do not attempt to splice over the same virtual edge in a parent cluster. We informally describe a way of ensuring this, that is similar to the technique used in the case of $K_{3,3}$-free graphs. For each cut-triple of G that has two or more children in H', we will identify one of them as an "upper normalizer" and another as a "lower normalizer". This can be done in constant time; each cut-triple in the decomposition tree H simply picks two of its child components and labels them appropriately. The basic idea is that when it is time for a parent cluster and all child clusters to merge across a cut-triple (u, v, w) that has two or more children (so that (u, v, w) is on a common root to leaf path in th parent tree), exactly one of the children will splice over the "upper" pair (u, v) if it is a virtual edge and exactly one other child is similarly assigned to take care of the "lower" pair (v, w). In addition, we need to avoid the condition where a single edge

may be spliced over by children of two different cut-triples both containing the edge. Again, we handle this case by pre-assigning every virtual edge to a single cut-triple. This can again be done in constant time in a CRCW PRAM as follows. Each cut-triple "votes" for itself to be assigned to each of its associated virtual edges, by writing its own identifier into locations associated with those edges. For each edge, exactly one of the cut-triples it is contained in "wins" and only a child cluster associated with this cut-triple will ever try to splice onto that edge.

We now turn to the analysis of the running time of the parallel algorithm. Decomposing a triconnected component into the tree of four-connected components and cut-triples can be accomplished in $O(\log n)$ time with $O(n^2)$ processors using the algorithm of Kanevsky and Ramachandran [KR87]. Generating the different versions of the DFS trees for each planar component takes $O(\log^2 n)$ time with a linear number of processors using the algorithm in [HY88]. As was observed earlier, each of the $O(\log n)$ phases of merging can be done in constant time with a linear number of processors. This gives a total of $O(\log^2 n)$ time and $O(n^2)$ processor requirement to generate a DFS tree of a given nonplanar triconnected graph.

Theorem 1 *A depth first search tree of a triconnected K_5-free graph can be computed in $O(\log^2 n)$ time using $O(n^2)$ processors.*

Proof: The proof follows from the previous discussion. ∎

Theorem 2 *A depth first search tree of any K_5-free graph can be computed in $O(\log^2 n)$ time using $O(n^2)$ processors.*

Proof: As described earlier, given an arbitrary K_5-free graph, we first decompose it into its triconnected components and run the algorithm for $K_{3,3}$-free graphs on it. Whenever we are required to generate a DFS tree for a nonplanar triconnected component, we use the algorithm described above to obtain it. The time and processor bounds follow easily. ∎

References

[AA87] A. Aggarwal and R. J. Anderson. A random NC algorithm for depth first search. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 325–334, 1987.

[Hal43] D. W. Hall. A note on primitive skew curves. *Bulletin of the American Mathematical Society*, 49:935–937, 1943.

[HY88] X. He and Y. Yesha. A nearly optimal parallel algorithm for constructing depth first spanning trees in planar graphs. *SIAM Journal on Computing*, 17(3):486–491, June 1988.

[Khu88a] S. Khuller. Extending planar graph algorithms to $K_{3,3}$–free graphs. Technical report, Computer Science Department, Cornell University, March 1988.

[Khu88b] S. Khuller. Parallel graph algorithms for K_5-minor free graphs. Technical report, Computer Science Department, Cornell University, March 1988.

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

[MR87] G. L. Miller and V. Ramachandran. A new graph triconnectivity algorithm and its parallelization. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 335–344, 1987.

[Rei85] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20, 1985.

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

[Smi86] J. Smith. Parallel algorithms for depth first searches I : Planar graphs. *SIAM Journal on Computing*, 15:814–830, 1986.

[SV82] Y. Shiloach and U. Vishkin. An $o(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[Tut66] W. T. Tutte. *Connectivity in Graphs.* University of Toronto Press, 1966.

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

[Wag85] K. Wagner. Über eine Eigenschaft der ebenen Komplexe. *Math. Ann.*, 20, 1985.