The complexity of planarity testing*

Eric Allender†
Dept. of Computer Science,
Rutgers University,
Piscataway, NJ, USA.
Email: allender@cs.rutgers.edu

Meena Mahajan‡
The Institute of Mathematical Sciences,
Chennai 600 113, INDIA.
Email: meena@imsc.ernet.in

August 27, 2003

Abstract

We clarify the computational complexity of planarity testing, by showing that planarity testing is hard for L, and lies in SL. This nearly settles the question, since it is widely conjectured that $\mathsf{L} = \mathsf{SL}$ [Sak96]. The upper bound of SL matches the lower bound of L in the context of (nonuniform) circuit complexity, since L/poly is equal to SL/poly.

Similarly, we show that a planar embedding, when one exists, can be found in $\mathsf{FL}^{\mathsf{SL}}$.

Previously, these problems were known to reside in the complexity class AC^1, via a $O(\log n)$ time CRCW PRAM algorithm [RR94], although planarity checking for degree-three graphs had been shown to be in SL [Rei84, NTS95].

1 Introduction

The problem of determining if a graph is planar has been studied from several perspectives of algorithmic research. From most perspectives, optimal algorithms are already known. Linear-time sequential algorithms were presented by Hopcroft and Tarjan [HT74] and (via another approach) by combining the results of [LEC67, BL76, ET76]. In the context of

*An extended abstract of this paper appeared in the Proceedings of STACS 2000 [AM00].

†Supported in part by NSF grants CCR-9734918 and and CCR-0104823.

‡Part of this work was done when this author was supported by the NSF grant CCR-9734918 on a visit to Rutgers University during summer 1999.

parallel computation, a logarithmic-time CRCW-PRAM algorithm was presented by Ramachandran and Reif [RR94] that performs almost linear work.

From the perspective of computational complexity theory, however, the situation has been far from clear. In this paper we will focus on "small" subclasses of the complexity class NC. Recall that

$$\mathsf{L} \subseteq \mathsf{SL} \subseteq \mathsf{NL} \subseteq \mathsf{AC}^1$$
$$\mathsf{SL} \subseteq \oplus\mathsf{L}.$$

(See [KW93].) L (respectively SL, NL) denotes deterministic (respectively symmetric, nondeterministic) logarithmic space, and AC^1 denotes problems solvable by polynomial size AND-OR circuits of logarithmic depth, where the gates are allowed to have any number of inputs. The class $\oplus\mathsf{L}$ consists of problems solvable by determining if a nondeterministic logspace machine has an odd or even number of accepting paths. Although it is not known if NL is contained in $\oplus\mathsf{L}$, it is known that NL is contained in $\oplus\mathsf{L}/\text{poly}$ [GW96].

The best upper bound on the complexity of planarity testing that has been published so far is the bound of AC^1 that follows from the logarithmic-time CRCW-PRAM algorithm of Ramachandran and Reif [RR94]. In a recent survey of problems in the complexity class SL [AG00], the planarity testing problem for graphs of *bounded degree* is listed as belonging to SL, but this is based on the claim in [Rei84] that checking planarity for bounded degree graphs is in the "Symmetric Complementation Hierarchy", and on the fact that SL is closed under complement [NTS95] (and thus this hierarchy collapses to SL). However, the algorithm presented in [Rei84] actually works only for graphs of degree 3, and no straightforward generalization to graphs of larger degree is known. (This is implicitly acknowledged in [RR94, pp. 518–519].) Interestingly, Mario Szegedy has pointed out to us (personal communication) that an algebraic structure proposed by Tutte [Tut70], when combined with more recent results about span programs and counting classes [KW93], gives a $\oplus\mathsf{L}$ algorithm for planarity testing. It is listed as an open question by Ja'Ja' and Simon [JS82b] if planarity testing is in NL, although the subsequent discovery that NL is closed under complementation [Imm88, Sze88] allows one to verify that one of the algorithms of [JS82a, JS82b] can in fact be implemented in NL. It remains an open question if their algorithm can be implemented in SL, but in this paper we establish that the algorithm of Ramachandran and Reif can be implemented in SL.

We also show that the planarity testing problem is hard for L under projection reducibility.

This essentially solves the question of planarity testing from the complexity-theoretic point of view. To see this, it is sufficient to recall that it is widely conjectured that $\mathsf{SL} = \mathsf{L}$. This conjecture is based on the following considerations:

- The standard complete problem for SL is the graph accessibility problem for undirected graphs (UGAP). Upper bounds on the space complexity of UGAP have been dropping, from $\log^2 n$ [Sav70], through $\log^{1.5} n$ [NSW92], to $\log^{4/3} n$ [ATSWZ02]. It is plausible that this trend will continue to eventually reach $\log n$.

- UGAP can be solved in randomized logspace [AKL+79]. Recent developments in derandomization techniques have led many researchers to conjecture that randomized logspace is equal to L [Sak96].

In the context of nonuniform complexity theory (for example, as explored in [KL82, CSV84]), the corresponding nonuniform complexity classes L/poly and SL/poly are equal.[1] Hence in this setting, the computational complexity of planarity testing is resolved; it is complete for L/poly under projections.

One consequence of our result is that counting the number of perfect matchings in a planar graph is reducible to the determinant, when the graph is presented as an adjacency matrix. More precisely, it follows from this paper and from [MSV99] that there is a (nonuniform) projection that takes as input the adjacency matrix of a graph G, and produces as output a matrix M with the property that if G is planar then the absolute value of $det(M)$ is the number of perfect matchings in G. (*Sketch*: Given the proper advice strings, a GapL algorithm can take as input the matrix M, compute its planar embedding (since this is in L/poly), then compute its "normal form embedding" along a unique computation path (since $\mathsf{NL} \subseteq \mathsf{UL}$/poly [RA00]), and then use the algorithm in [MSV99] to compute a number whose absolute value is the number of perfect matchings in M. Since the determinant is complete for GapL under projections, the result follows.)

The paper is organized as follows. In Section 2 we present our hardness result for planarity testing. In Section 3, we briefly recapitulate the notation and definitions used in the subsequent algorithms, and show that some general-purpose algorithms for operating on graphs and trees can be performed in SL. In Section 4 we show that the algorithms of [MSV86] for finding an open ear decomposition of a biconnected graph and computing an st-numbering, essential pre-processing procedures for the planarity testing algorithm of [RR94], can be implemented in SL. In Section 5 we show that the algorithm of [RR94] can be implemented in SL. These sections, Section 4 and Section 5, are written so as to be read as companions to [MSV86] and [RR94] respectively.

2 Hardness of planarity testing

In this section we prove that planarity testing is hard for L, even for graphs of maximum degree three.

The following problems are known to be complete for L under uniform projections:

Definition 1 *Undirected Forest Accessibility (*UFA*): Given an undirected forest G and vertices u, v, decide if u and v are in the same tree.*

Definition 2 *Iterated Permutation Precedence (ΠS_n) Given a sequence of n permutations $\pi_1, \ldots, \pi_n \in S_n$ as a ternary relation $R(i, j, k)$ (meaning that the kth permutation takes element i to element j), and given an element u, decide if the product $\prod \pi_i$ takes element 1 to an element $v \geq u$.*

Definition 3 *Path Ordering (*ORD*): Given a directed path L, specified by giving for each vertex w its successor $s(w)$ along the path, and given vertices u, v, decide if u precedes v in L. (This representation of a directed path is called a directed line graph or a successor graph in [Ete97].)*

[1] That is, a *universal traversal sequence* [AKL+79] can be used as an "advice string" to enable a logspace-bounded machine to solve UGAP.

The hardness of the UFA problem for L was shown in [CM87], where only an NC^1 reduction is claimed. However it is easy to see that the hardness is under uniform projections as well; see for instance [Ete97] (Lemma 4.3).

The hardness of ΠS_n for L under uniform projections was shown in [CM87, Ete97].

The hardness of ORD for L under uniform projections is established in [Ete97] by a reduction from ΠS_n.

The UFA problem suffices to establish hardness of planarity testing for L. Let G' be the complete graph on 5 vertices, minus any one edge (p, q). The graph H is obtained by identifying vertices u and v of G (from the UFA instance) with vertices p and q of G'. Clearly, H is planar if and only if (G, u, v) is not in UFA.

By constructing a reduction from ΠS_n, we can prove something stronger – planarity testing is hard for L even when restricted to graphs of maximum degree 3. The reduction to a large extent mimics the reduction of [Ete97] from ΠS_n to ORD. Etessami's construction produces a line graph (a directed path) where the relative ordering of s and t depends on whether $\prod \pi_i[1] \geq u$. A modification of this construction gives the disjoint union of a line graph and a directed cycle, with two special vertices that lie in the two different components if $\prod \pi_i[1] \geq u$, and lie in the same component otherwise. By suitably attaching a copy of $K_{3,3} - e$ (the graph $K_{3,3}$ with any one edge deleted) to this graph, we have planarity iff $\prod \pi_i[1] \geq u$. The complete construction is described in the proof of the following theorem.

Theorem 4 *Planarity testing is hard for* L *under projections, even when restricted to graphs with maximum degree 3.*

Proof: We reduce ΠS_n to planarity testing. Recall the reduction from ΠS_n to ORD, from [Ete97]. Given the sequence of n permutations $\pi_1, \ldots, \pi_n \in S_n$, the directed graph $G = (V, E)$ is constructed as follows:

$V = \{\langle a, b, c\rangle \mid a \in \{1, \ldots, n+1\}, b \in \{1, \ldots, n\}, c \in \{f, r\}\}$

Vertices with third component f have edges tracing the permutations $\pi_1, \ldots, \pi_n \in S_n$; the first component describes which permutation is being traced out. Thus E includes edges

$$\langle a, b, f\rangle \longrightarrow \langle a+1, \pi_a(b), f\rangle.$$

Vertices with third component r have edges tracing the permutations $\pi_1, \ldots, \pi_n \in S_n$ in reverse; again, the first component describes which permutation is being traced out. Thus E includes edges

$$\langle a+1, \pi_a(b), r\rangle \longrightarrow \langle a, b, r\rangle.$$

At one end ($a = 1$), the two copies are connected matched. At the other end ($a = n+1$), they are connected matched with a cyclic shift, except at u. Thus E includes the following edges:

$$\begin{array}{rcll} \langle 1, b, r\rangle & \longrightarrow & \langle 1, b, f\rangle & \\ \langle n+1, b, f\rangle & \longrightarrow & \langle n+1, b+1, r\rangle & \text{for } b \neq n, b \neq u-1 \\ \langle n+1, n, f\rangle & \longrightarrow & \langle n+1, 1, r\rangle & \end{array}$$

Let σ denote the product permutation $\prod \pi_i$. The graph G is a line graph with source $\langle n+1, u, r\rangle$ and sink $\langle n+1, u-1, f\rangle$. It traces out vertices as follows:

$$
\begin{array}{cccccccc}
\langle n+1, u, r\rangle & \leadsto & \langle 1, \sigma^{-1}(u), r\rangle & \longrightarrow & \langle 1, \sigma^{-1}(u), f\rangle & \leadsto & \langle n+1, u, f\rangle & \longrightarrow \\
\langle n+1, u+1, r\rangle & \leadsto & & \ldots & & \leadsto & \langle n+1, u+1, f\rangle & \longrightarrow \\
\vdots & & & & & & & \vdots \\
\langle n+1, n, r\rangle & \leadsto & \langle 1, \sigma^{-1}(n), r\rangle & \longrightarrow & \langle 1, \sigma^{-1}(n), f\rangle & \leadsto & \langle n+1, n, f\rangle & \longrightarrow \\
\langle n+1, 1, r\rangle & \leadsto & & \ldots & & \leadsto & \langle n+1, 1, f\rangle & \longrightarrow \\
\langle n+1, 2, r\rangle & \leadsto & & \ldots & & & & \\
\vdots & & & & & & & \vdots \\
\langle n+1, u-1, r\rangle & \leadsto & \langle 1, \sigma^{-1}(u-1), r\rangle & \longrightarrow & \langle 1, \sigma^{-1}(u-1), f\rangle & \leadsto & \langle n+1, u-1, f\rangle &
\end{array}
$$

Consider vertices of the form $\langle 1, b, r\rangle$. These vertices appear in the second column above, and can precede $\langle n+1, n, f\rangle$ if and only if $\sigma(b) \geq u$. Thus the input is an instance of ΠS_n iff $(G, \langle 1, 1, r\rangle, \langle n+1, n, f\rangle)$ is an instance of ORD.

Now consider the graph G_1, that is essentially the same as G, except that edge $\langle 1, 1, r\rangle \longrightarrow \langle 1, 1, f\rangle$ is replaced by edge $\langle 1, 1, r\rangle \longrightarrow \langle n+1, u, r\rangle$. This disconnects the graph into two pieces: a cycle containing $\langle 1, 1, r\rangle$, and a line graph. The vertex $\langle n+1, n, f\rangle$ is on the cycle iff it precedes $\langle 1, 1, r\rangle$ in G, i.e. iff the input instance is not in ΠS_n. Note that the maximum degree in G_1 is 2.

Let G_2 be the graph $K_{3,3}$, minus any one edge (p, q).

The graph H is obtained by connecting vertices $\langle n+1, n, f\rangle$ and $\langle 1, 1, r\rangle$ of G_1 to vertices p and q respectively of G_2 via single edges. Clearly, H is planar iff the input instance is in ΠS_n. Furthermore, the maximum degree of any vertex in H is 3. ■

We remark that a similar proof shows that planar instances of UGAP are reducible to planarity testing.

3 Definitions and Basic Graph Operations

3.1 Notation and Definitions

For formal definitions of most graph-theoretic terms used below, see any standard text e.g. [Eve79, Gib85, Tut84]. We give a few definitions here for completeness.

A graph is *planar* if it can be drawn on the plane so that the edges intersect only at end vertices. Such a drawing is a *planar embedding*. A *combinatorial embedding* ϕ is a cyclic ordering of the edges around each vertex. Replace each edge (u, v) by directed arcs $\langle u, v\rangle$ and $\langle v, u\rangle$. Then ϕ is a cyclic ordering of the arcs leaving each vertex. Let R map each arc to its inverse. Then ϕ is a *planar combinatorial embedding* iff the number of orbits f in $\phi^* \triangleq \phi \circ R$ satisfies Euler's formula $n + f = m + 1 + c$. (Here, n, m, c are the number of vertices, undirected edges, connected components respectively.) For more background, see [Whi84, Section 6-6].

The *connected components* of a graph induce a partition on the vertex set. A *cut vertex* (or *separation vertex*) is a vertex with the property that $G-\{v\}$ has more connected components than G. A graph G is said to be biconnected if it has no cut-vertex. A *2-component* (also called a *biconnected component* or a *nonseparable component*) of a graph G is a maximal subgraph that is connected and that remains connected if any one vertex is deleted. 2-components can share vertices, but the edges of 2-components form a partition of the edge set of G.

Given a spanning tree T of a connected graph G, each edge $e \notin T$ completes a cycle in G. Such cycles, with exactly one non-tree edge, are called the *fundamental cycles* of G with respect to T.

Given a connected graph, a cycle C induces a partition of the edges not on C, where two edges are in the same class if one can be reached from the other without using any vertices of C except at the endpoints. The classes of this partition are called the *bridges* of C. The vertices of a bridge that lie on C are called *attachment points*. Two bridges B_1 and B_2 relative to C are said to *conflict*, or *overlap*, or *interlace*, if either they share 3 attachment points, or for some u, v, w, x occurring in that order on C, u, w are attachment points of B_1 and v, x are attachment points of B_2.

A tree is said to be *rooted at vertex* r if all edges are oriented to form paths from r towards the leaves. An undirected tree can be rooted at any vertex. Once the tree is rooted, the terms *children, parent, descendants, ancestors*, become well-defined in the obvious way. The root is an ancestor of all vertices in the tree.

An *Euler tour* of a graph is a closed path that uses every edge of the graph exactly once. For a rooted tree, by assuming that for each edge (u, v) the reverse edge (v, u) is also available, we get a directed graph in which an Euler tour exists; by convention, the tour is assumed to begin and end at r.

The *least common ancestor* (lca) of two vertices u, v in a rooted tree is a vertex w such that w appears on both the $r \rightsquigarrow u$ path and the $r \rightsquigarrow v$ path, and no descendant of w has this property. If the tree is a spanning tree of a graph, then for each edge $e = (u, v)$, we refer to $lca(u, v)$ as $lca(e)$.

3.2 Elementary graph computations in SL

In this subsection, we present some general-purpose algorithms for operating on graphs and trees. Our method of exposition is to give a statement of the subproblem to be solved, and then in parentheses give an indication of how this subproblem can be restated in a way that makes it clear that it can be solved using an oracle for undirected reachability, or by making use of primitive operations that have already been discussed.

Often, the upper bounds we show are easier to understand in terms of either L^{SL} or as NC^1 circuits with oracle gates for the reachability problem in undirected graphs. As usual when defining NC^1-reducibility, an oracle gate with fan-in m is considered to have depth $\log m$; see [Coo85, Wil85]. Since SL is closed under NC^1 reducibility and logspace-Turing reducibility [NTS95], these bounds coincide with SL.

Given a graph G, the following conditions can be checked in SL:

1. Are u and v in the same 2-component? (Algorithm: for each vertex x, check if the removal of x separates u and v. This can be tested using UGAP.)

2. Let each 2-component be labeled by the smallest two vertices in the 2-component. Is (u, v) the "name" of a 2-component? (First check that u and v are in the same 2-component, and then check that no $x < \max(u, v)$ with $x \notin \{u, v\}$ is in the same 2-component.)

3. Is u a cut-vertex? (Are there vertices v, w connected in G but not in $G - \{u\}$?)

4. Is there is a path (not necessarily simple) of odd length between vertices s and t? (Make two copies of each vertex. Replace edge (u, v) by edges $(u0, v1)$ and $(u1, v0)$. Check if $s0, t1$ are connected in this new graph.)

5. Is G bipartite (i.e. 2-colorable)? [Rei84, NTS95, AG00].

6. If G is connected, 2-colorable, and vertex 1 is colored 1, is vertex i colored 2? (Test if there is a path of odd length from 1 to i.)

7. Is edge e in the lexicographically first spanning tree T of G (under the standard ordering of edges)? [NTS95]

Given a graph G and a spanning tree T, the following conditions can be checked in SL. (Actually, items 1 through 7, and item 11 can be checked in L.)

1. For $e \in T$ with $e = (x, y)$, does $x \longrightarrow y$ occur at position i of the lexicographically first Euler tour rooted at r, (denoted ET_r)? Does $x \longrightarrow y$ precede $y \longrightarrow x$? (In deterministic logspace, one can compute the lexicographically-first Euler tour by starting at r and following the edge $r \longrightarrow x$, where x is the smallest neighbor of r in T. At any stage in the tour, if the most recent edge traversed was $u \longrightarrow v$, the next edge in the Euler tour is $v \longrightarrow z$ where z is the smallest neighbor of v greater than u in T if such a neighbor exists, and z is the smallest neighbor of v otherwise.)

2. Is $u = parent(v)$ when T is rooted at r? (Equivalently, is $u \longrightarrow v$ the first edge of ET_r to touch v? This can be checked in L.)

3. If T is rooted at r, is u a descendant of v? (Equivalently, does the first occurrence of u in ET_r lie between the first and last occurrences of v?)

4. Is z the least common ancestor (lca) of vertices x and y in T? (Check that x and y are both descendants of z, and check that this property is not shared by any descendant of z.)

5. Is i the preorder number of vertex u? (Count the number of vertices with first occurrence before that of u in ET_r.)

6. Is vertex u on the fundamental cycle C_e created by non-tree edge e with T? (Let $e = (p, q)$. Vertex u is on C_e iff the graph $T - \{u\}$ has no path from p to q.)

7. Is edge f on C_e? (This holds iff $f = e$ or $f \in T$ and both endpoints of f are on C_e.)

8. Are vertices u, v on the same bridge with respect to C_e? (Vertices u and v are on the same bridge iff there is a path from u to v in G, with no internal vertices of the path belonging to C_e.)

9. Are edges f, g on the same bridge with respect to C_e? (This holds if $f, g \notin C_e$, neither f nor g is a trivial bridge (i.e. a chord of C_e), and the endpoints of f, g that are not on C_e are on the same bridge with respect to C_e.)

10. Is vertex u a point of attachment of the bridge of C_e that contains edge f? (Let $f = (f_1, f_2)$. If both f_1 and f_2 are on C_e, then these are the only points of attachment of the trivial bridge $\{f\}$. Otherwise, if f_i is not on C_e, then u is a point of attachment iff $u \in C_e$ and u, f_i are on the same bridge with respect to C_e.)

11. Given vertices u, v, on C_e, and given a sequence $\langle w_1, w_2, \ldots \rangle$, is there a path from u to v along C_e avoiding vertices $\langle w_1, w_2, \ldots \rangle$? (This is simply the question of connectivity in $C_e - \{w_1, w_2, \ldots\}$.)

12. Relative to C_e, do the bridges containing edges f and g interlace? (Either there is a triple u, v, w where all three vertices are points of attachments of both bridges, or there is a 4-tuple u, v, w, x where (1) u, w are attachment points of the bridge containing f, (2) v, x are attachment points of the bridge containing g, and (3) u, v, w, x occur in cyclic order on C_e. To check cyclic order, use the previous test.)

4 Obtaining open ear decompositions and st-numberings in SL

An ear decomposition of a connected graph is a partition of its edge set E into simple paths $P_0, P_1, \ldots, P_{r-1}$, where

- P_0 is a single edge.
- For each $P_i, i > 0$, the endpoints of P_i appear in $\bigcup_{j<i} P_j$.
- For each $P_i, i > 0$, no internal vertex of P_i appears on any ear $P_j, j < i$.

The ear decomposition is said to be an open ear decomposition if, for each path P_i, the endpoints of P_i are distinct.

An st-numbering of a graph is a numbering of its n vertices from $s = 1$ to $t = n$ such that every vertex v other than s and t has adjacent vertices u, w satisfying $u < v < w$.

It is known that a graph has an st-numbering iff it is biconnected iff it has an open ear decomposition[LEC67, Whi32].

Given an open ear decomposition $D = [P_0, \ldots P_{r-1}]$ of a biconnected graph G, where P_0 consists of the edge (s, t), the graph G_{st} is the result of orienting each edge of G, so that

- Each ear is directed from one endpoint to the other.
- The edge (s,t) is oriented $s \longrightarrow t$.
- Every vertex lies on a path from s to t
- The resulting graph is acyclic.

Such an orientation is always possible.

In this section, we show that the parallel algorithms of [MSV86] for finding an open ear decomposition and an st-numbering of a biconnected graph can be performed in SL. We briefly describe the algorithms, using notation from [MSV86], and then elaborate on each step to show that it can be performed in SL. Again, recall that it suffices for us to show that the computation can be performed in L^{SL} or by NC^1 circuits with oracle gates for the reachability problem in undirected graphs. For proofs of correctness of the parallel algorithms, the reader is referred to [MSV86].

4.1 Finding an open ear decomposition

The algorithm for finding an open ear decomposition of a biconnected graph is described briefly in Figure 1, and the individual steps with their SL implementation are described in more detail below.

Input: biconnected graph $G = (V, E)$ where edges of E have distinct serial numbers from the set $\{1, 2, \ldots, |E|\}$; adjacent vertices s, t.

Output: For each edge e, a label ear(e) that is the number of the ear containing e.

Step 1. Compute the lexicographically first spanning tree T' of the subgraph induced on the vertex set $V - \{t\}$. Add vertex t and edge (s,t) to get a spanning tree T of G. Root it at t. Consider the partition of E into special edge (s,t), tree edges $TE = E_{T'}$, and non-tree edges $NTE = E - E_T$.

Step 2. For each vertex v, compute $F(v) =$ father of v in T, and level$(v) =$ number of edges in the unique $t \rightsquigarrow v$ path in T.

For each edge $e = (u,v) \in NTE$, find $lca(u,v)$.

For each edge $e = (u,v) \in NTE$, define $\textsc{Number}(e) = (level(lca(u,v)), serial(e))$. (The reader will note that $\textsc{Number}(e)$ is actually a pair of numbers.)

Step 3. For each edge $e = (u,v) \in TE$, let F_e be the set of edges f in NTE for which $e \in C_f$; i.e. e is in the fundamental cycle created by f and T. Define master(e) to be that edge $g \in F_e$ with the lexicographically smallest $\textsc{Number}$.

The remaining edges (i.e., the special edge (s,t), and NTE) are their own masters.

The partition of E according to master, with blocks of the partition ordered according to the $\textsc{Number}$ assigned to the master edge in that part, is an ear decomposition (not necessarily open) of G.

Input: biconnected graph $G = (V, E)$ where edges of E have distinct serial numbers from the set $\{1, 2, \ldots, |E|\}$; adjacent vertices s, t.

Output: For each edge e, a label ear(e) that is the number of the ear containing e.

1. Construct the lexicographically first spanning tree T.

2. Compute a NUMBER for each non-tree edge.

3. Find the master edge for each edge.

4. Construct auxiliary graphs H_x for each vertex x.

5. Find the lexicographically first spanning forest for each auxgraph.

6. Root each tree of the forests at a special vertex, preorder it, and so obtain NEWNUMBER for each non-tree edge of G.

7. Recompute master edges of tree edges using NEWNUMBER instead of NUMBER. The partition of edges into ears is done by having all edges having the same master edge constitute an ear. For each edge e, define ear(e) to be the number of its ear.

Figure 1: Finding an open ear decomposition.

Step 4. For each vertex x consider the bipartite undirected graph $H_x = (V_x, E_x)$ where

$$V_x = \{[u, v] \mid (u, v) \in NTE \text{ and } lca(u, v) = x\} \cup \{[u, F(u)] \mid F(u) = x\}$$

$$E_x = \left\{ ([u, v], [w, x]) \mid \quad \begin{array}{l} (u, v) \in NTE, u \text{ is not an ancestor of } v, (w, x) \in TE, \\ (x, w) \text{ is the first edge in the unique } x \rightsquigarrow u \text{ path of } T \end{array} \right\}$$

Note that the collection of vertices of all the H_x partitions E.

Step 5. Compute connected components and spanning forests of each H_x.

Step 6. For each spanning tree in the forest so obtained, find a $[w, x]$ where $F(w) = x$ such that $level(lca(master(w, x))) < level(x)$.

Root the tree at $[w, x]$.

Preorder the vertices of the tree. When this is done for each tree, all edges of G get a preorder number.

For each $e = (u, v) \in NTE$, define NEWNUMBER(e) to be $(level(lca(u, v)), preorder(e))$.

Step 7. Repeat Step 3 with NEWNUMBER instead of NUMBER. That is, for each edge $e \in TE$, define master(e) to be that edge $g \in F_e$ with the lexicographically smallest NEWNUMBER.

The partition of E according to master, with blocks of the partition ordered according to the NEWNUMBER assigned to the master edge in each part, is an open ear decomposition.

End

All the steps above involve one or more of the basic operations described in the previous section, and can thus be performed in SL.

4.2 Constructing the st-numbering and the directed st-graph G_{st}

In [MSV86], it is shown how, given an open ear decomposition of a graph G, the st-numbering of G as well as the directed graph G_{st} can be constructed. The steps of their algorithm are briefly described in Figure 2 below, after which we elaborate on each of these steps to show that all of these are implementable in SL[2].

Input $D = [P_0, \ldots P_{r-1}]$

1. a). Construct the ear tree ET.

   b). For each ear P_i, find its hinge H_i and determine the type of P_i.

   c). Construct the hinge tree HT.

2. Orient the ears based on HT.

3. Construct the numbering tree NT.

4. Preorder NT, using a specific ordering among children of the same node.

Figure 2: Orienting ears and finding an st-numbering.

The [MSV86] algorithm for st-numbering assumes that the open ear decomposition D is presented by giving, for each edge $e = (u, v)$,

(a) the number of the ear, say P_i, containing it. Define the ear number $\textsc{ear}(u)$ of a vertex u to be the smallest number i such that u appears on ear P_i. Let the end points of P_i have ear numbers j and k respectively, where $i > j \geq k \geq 0$. The endpoints of P_i on P_j and P_k are referred to as $L(P_i)$ and $R(P_i)$ respectively. (If $j = k$, choose the lexicographically first endpoint as $L(P_i)$.)

[2]The procedure for st-numbering as described in the preliminary version of this paper[AM00] is erroneous.

(b) a pointer to the next edge along P_i leading to $L(P_i)$ (if there is such an edge), and

(c) a pointer to the next edge along P_i leading to $R(P_i)$ (if there is such an edge).

The open ear decomposition algorithm described in the preceding subsection computes only the labeling of (a); however, it is straightforward to verify that given (a), (b) and (c) can be computed in L. Another labeling that is often required is an ear number for each vertex v other than s, t; ear(v) is the number of the ear on which v is an internal vertex. Given an open ear decomposition by an ear labeling of the edges, the ear labeling of the vertices can be computed in L.

While elaborating on the steps of the algorithm, we need some more definitions. For each ear P_i, the vertex $L(P_i)$ is called its anchor. The vertex of ear P_i that is adjacent to $L(P_i)$ (respectively $R(P_i)$) is called $LS(P_i)$ (respectively $RS(P_i)$). Again, it is straightforward to verify that given an open ear decomposition, the vertices $L(P), LS(P), RS(P), R(P)$ can be found for each ear P in L.

Step 1 a) The ear tree ET is defined as follows:
$V_T = \{P_i \mid P_i \text{ is an ear}\}$
$E_T = \{\ P_i \longrightarrow P_j \mid i \geq 1, \text{EAR}(L(P_i)) = j\}$[3].

Step 1 b) We find below the hinge and the type of each ear. The hinge of an ear is another ear, and the type of an ear is "same" or "opposite".

Case 1 The vertices $L(P_i)$ and $R(P_i)$ have the same ear number, say j. Then hinge(P_i) $= P_j$. Furthermore, if on P_j the relative ordering[4] is $L(P_j), L(P_i), R(P_i), R(P_j)$, then the type of P_i is "same". Otherwise, on P_j the relative ordering is $L(P_j)$, $R(P_i)$, $L(P_i)$, $R(P_j)$, and then P_i has type "opposite".

Case 2 The vertices $L(P_i)$ and $R(P_i)$ belong to ears P_j and P_k respectively, $j \neq k$. In ET, $lca(P_j, P_k) = P_\alpha$. Note that $j \geq k$ by definition of $L(P)$.

Case 2.1 $P_\alpha \neq P_j$, $P_\alpha \neq P_k$. Let v_j (respectively v_k) be the "anchor ancestor" of P_j (respectively P_k) in P_α. (See [MSV86] for a definition of anchor ancestor of an ear P in P'. Informally, it can be understood as follows: If P is a descendant of P'' that is a child of P', then $L(P'')$ is the anchor ancestor of P in P'.)

Case 2.1.1 $v_j \neq v_k$. Set hinge(P_i) to be P_α. Furthermore, P_i has type "same" iff on the hinge the relative ordering is $L(P_\alpha), v_j, v_k, R(P_\alpha)$.

Case 2.1.2 $v_j = v_k$. Let $(P_\alpha, P_{j'})$ be the first edge on the path in ET from P_α to P_j. The corresponding edge in G, e_j, has v_j as an endpoint. Similarly define $P_{k'}$ and e_k; e_k is also adjacent to v. If e_j precedes e_k in the adjacency list of v, then hinge(P_i) is $P_{j'}$ and P_i is of type "opposite", otherwise hinge(P_i) is $P_{k'}$ and P_i is of type "same".

[3]In [MSV86] E_T is said to be the set of edges $\{P_i \longrightarrow P_j \mid i \geq 1, L(P_i)$ is an internal vertex of $P_j\}$, but it is easy to see that this more general condition is really needed

[4]Here and elsewhere, note that some of these vertices may coincide. For instance, it is possible that $L(P_j) = L(P_i)$ and/or $R(P_i) = R(P_j)$.

Case 2.2 $P_\alpha = P_k$. Let v be the anchor ancestor of P_i on P_k.

Case 2.2.1 $v \neq R(P_i)$. Then hinge(P_i) is P_α, and P_i is of type "same" iff the relative ordering on P_α is $L(P_\alpha), v, R(P_i), R(P_\alpha)$.

Case 2.2.2 $v = R(P_i)$. Let $P_{j'}$ be the first ear on the path in T from P_α to P_j. Then hinge(P_i) is $P_{j'}$ and P_i is of type "opposite".

Step 1 c) The graph HT is defined as follows:
$V_{HT} = \{P_i \mid P_i \text{ is an ear}\}$
$E_{HT} = \{P_i \longrightarrow P_m \mid P_m \text{ is the hinge of } P_i\}$. This graph turns out to be a tree, and is referred to as the hinge tree.

Step 2 The hinge and the type of each ear determine its orientation: an ear P_i is oriented from $L(P_i)$ to $R(P_i)$ iff either its type is "same" and its hinge ear is oriented $L(.)$ to $R(.)$, or its type is "opposite" and its hinge ear is oriented $R(.)$ to $L(.)$. We explicitly compute the orientations as follows.

Orient the ear P_0 from s to t. For ear P_i where $i \geq 1$, consider the $P_i \rightsquigarrow P_0$ path in the hinge tree. Count the number of ears on this path with type "opposite". If this number is even, then orient P_i from $L(P_i)$ to $R(P_i)$ to get an *outgoing ear* of $L(P_i)$; otherwise orient it from $R(P_i)$ to $L(P_i)$ to get an *incoming ear* of $L(P_i)$.

Step 3 The numbering tree NT (an undirected tree) has two adjacent vertices v_{in} and v_{out} for each vertex v of G. Additional edges of NT are of the following forms:

Let edge (u, v) be in ear P_i and suppose that when P_i is traversed according to its orientation, we encounter u before v. Then edge (u_{out}, v_{in}) is in NT.

If P_i is an incoming ear with anchor v, then edge (v_{in}, w_{in}) is in NT, where $w = RS(P_i)$.

If P_i is an outgoing ear with anchor v, then edge (v_{out}, w_{in}) is in NT, where $w = LS(P_i)$.

Step 4 Perform a preorder traversal of NT beginning at s_{in}. Assign preorder numbers only to vertices of the form v_{out}. These numbers provide the st-numbering for the corresponding vertices of G.

End

All the above steps involving tracing ancestors in trees, determining ordering on a path and other basic operations described in the previous section. Thus they are all implementable in SL.

5 The SL algorithm for planarity testing and embedding

Finally we come to the the algorithm of Ramachandran and Reif [RR94] and describe how it can be implemented in SL. This algorithm is complex, and involves a number of fairly involved technical definitions. While we would like to make this presentation as self-contained as possible, it is not appropriate to reproduce the entire algorithm of [RR94] here. Instead, we have written this section so that each part can be read as a companion to [RR94]. We succinctly describe the main steps of the algorithm, using the same notation as in [RR94], and we show that each step of the corresponding algorithm can be computed in L^{SL}, or by NC^1 circuits with oracle gates for the reachability problem in undirected graphs. As argued earlier, it follows that the entire algorithm can be implemented in SL. For details of the algorithm and for the technical definitions involved, the reader is referred to [RR94].

5.1 An overview of the planarity testing algorithm

The planarity testing algorithm of Ramachandran and Reif [RR94] is outlined briefly in Figure 3, and works as follows: A graph is planar if and only if each of its 2-components is planar. To check if a 2-component is planar, first obtain an open ear decomposition and construct the directed (s,t)-numbering graph using the method from [MSV86]. A key idea used to obtain an embedding (originally from the Hopcroft-Tarjan sequential planarity testing algorithm [HT74]), is the following: consider any simple cycle C of a biconnected graph G embedded in the plane. The remaining edges of G are now partitioned into bridges. If two of these bridges interlace, then any embedding of G in the plane must place these bridges on opposite sides of C. If we were to proceed sequentially, we first embed P_0 and P_1 in any way we wish. These two paths together contain a closed cycle, so interlacing bridges of this cycle are now constrained to be on opposite sides. Subsequent paths of the open ear decomposition are embedded sequentially, while maintaining a list of constraints. Whole sections of the embedding may be flipped over, around a previously embedded path, if it turns out that some constraint cannot be met with the current embedding. Careful bookkeeping (of the partially constructed embedding and the constraints so far) allows for an efficient sequential algorithm.

Of course, things need substantial modification if efficient parallel testing is desired. In addition to the open ear decomposition and the directed (s,t)-numbering graph G_{st} of the biconnected graph G, we also construct the local replacement graph G_l (which is planar iff G is planar). Together, G_{st} and G_l uniquely specify a special spanning tree T_{st}. Now find all the fundamental cycles with respect to T_{st}. For each cycle, find approximations to all its bridges. (The bridges can be found exactly in SL, but the approximations allow for efficient testing.) These approximations are the bunches and bunch collections. Conflicts among interlacing bridges give rise to an interlacing parity graph on the bunches. A few additional constraints on the bunches give the constraint graph G^*.

If G^* is not bipartite, then G is not planar. Why? Intuitively, the constraints specify how a bridge (or approximation) can be embedded with respect to a previously embedded path. An edge between two bridges in G^* means that they must be on opposite sides of the concerned cycle. So an odd cycle implies that a bridge must simultaneously be inside and outside a cycle, an impossible situation.

If G^* is bipartite, it is still not clear that G is planar, since we used bunches instead of bridges. However, a 2-coloring of the vertices of G^* specifies, for each bunch, on which side of the fundamental cycle it should be embedded. This gives a combinatorial embedding of G_l, and this embedding is shown to be planar if G_l is planar.

The last steps are to recover an embedding of G from that of G_l, and to put together the embeddings of the 2-components.

For the complete algorithm and definitions of the terms used above, see [RR94].

1: Decompose the input graph into its 2-components.

2: For each 2-component G, do

- **2.1** Find an open ear decomposition $D = (P_0, P_1, \ldots, P_{r-1})$ with $P_0 = (s, t)$.
- **2.2** Direct the ears to get directed acyclic graph G_{st}.
- **2.3** Construct the local replacement graph G_l and the associated spanning tree T'_{st} and paths $D' = (P'_0, P'_1, \ldots, P'_{r-1})$.
- **2.4** Compute the bridges of each fundamental cycle C'_i.
- **2.5** Compute a bunch collection for each P'_i, and a hook for each bunch.
- **2.6** For each P'_i, construct its bunch graph and the corresponding interlacing parity graph.
- **2.7** Construct the constraint graph G^* and 2-color it, if possible.
- **2.8** From the 2-coloring, obtain a combinatorial embedding of G_l and hence G. Test if this embedding is planar.

3: Piece together the embeddings of the 2-components.

Figure 3: Planarity Testing Algorithm

The emphasis in [RR94] is to find a fast parallel algorithm that performs almost optimal work. However, for our purpose, any procedure that can be implemented in SL will do. Step 1 can be accomplished by determining, for each (u, v), if u and v are in the same 2-component; see Section 3.2. Steps 2.1 and 2.2 have been addressed in detail in Section 4. Step 2.4 has been discussed in subsection 3.2. The remaining steps are discussed in the following subsections.

5.2 Constructing the local replacement graph G_l

In G_l, each vertex v is replaced by a rooted tree T_v with $d(v) - 1$ vertices, one for each ear containing v. The construction exploits the fact that in the directed graph G_{st}, deleting the last edge of each path P_i for $i > 0$ gives a spanning tree T_{st}. The construction introduces new vertices, and maps each P_i to a path P'_i that is essentially the same as P_i, but has an extra edge involving a new vertex at each end.

The construction of G_l proceeds in 3 phases. In the first / second phase, the first / last edge of each ear is rerouted to a possibly new endpoint via one of the new vertices. In the last phase, some of the new edges are further rerouted to account for parallel ears.

The entire construction uses only the elementary operations described in subsection 3.2, and so can be implemented in $\mathsf{FL}^{\mathsf{SL}}$. The implementation immediately yields the new directed graph G'_{st}, and a listing of the new left and right endpoints $L(P'_i)$ and $R(P'_i)$ of each path.

5.3 Bunch Collections and Hooks

In the spanning tree T'_{st} of the graph G'_{st}, each path P'_i has a unique non-tree edge, which forms the fundamental cycle C'_i with respect to T'_{st}. In [RR94], each bridge of C'_i is classified as spanning, anchor or non-anchor depending on how the attachment points of C'_i are placed with respect to P'_i. Since bridges can be computed in $\mathsf{FL}^{\mathsf{SL}}$ (see subsection 3.2), this classification is also in SL.

In the nomenclature of [RR94], bunches are approximations to bridges: bunches contain only the attachment edges of bridges. A bridge is represented by at least one and possibly more than one bunch, subject to certain conditions. The conditions are: (1) A non-anchor bunch must be the entire bridge. (2) A spanning bunch must contain all attachment points of the corresponding bridge on internal vertices of P'_i and at least one edge attaching on $L(P'_i)$. (3) Edges within a bunch must be connected in G_l without using vertices from C'_i or from the other bunches. (4) The bunch collection for each P'_i must contain all attachments of bridges on its internal vertices and some attachment edges incident on $L(P'_i)$. Bunch collections are computed using operations described in Subsection 3.2.

A representative edge for each anchor bunch B is the hook $H(B)$, which also is used to determine a planar embedding if G turns out to be planar. $H(B)$ is usually an attachment on $C'_i - P'_i$ of the bridge of C'_i that contains B. The exception is when $L(P'_i)$ is the lca of the non-tree edge of P'_i, in which case $H(B)$ may be the incoming tree edge to $L(P'_i)$. Again, the entire procedure for computing hooks uses operations shown to be in SL in subsection 3.2, so $H(B)$ can be computed in $\mathsf{FL}^{\mathsf{SL}}$.

5.4 Bunch Graphs and Interlacing Parity Graphs

Once the bunch collections are formed, the bunch graphs are constructed as follows: extend each path P'_i to a path Q_i by introducing a new edge between $L(P'_i)$ and a new vertex $U(P'_i)$. Collapse each bunch B of P'_i to a single node v_B (which now has edges to some vertices of P'_i); thus B becomes a "star" S_B with center[5] v_B. Further, if B is an anchor bunch, include edge $(U(P'_i), v_B)$, and if B is a spanning bunch, include edge $(R(P'_i), v_B)$. This gives the so-called bunch graph $J_i(Q_i)$, which can clearly be constructed in $\mathsf{FL}^{\mathsf{SL}}$.

For each $J_i(Q_i)$, an interlacing parity graph $G_{i,I}$ is constructed as follows: There is a vertex v_B for each star S_B, and a vertex for each triple (u, v, B) where u, v are attachment vertices of S_B on Q_i, and u is an extreme (leftmost / rightmost) attachment. Edges connect (1) a bunch vertex v_B to all its chords (u, v, B), (2) bunch vertices v_S, v_T that share an internal (non-extreme) attachment vertex on Q_i, and (3) each chord to its left and right chords, when they exist. The left and right chords are defined as follows: For chord (u, v, B), consider the set of chords $\{(u', v', B') \mid B' \neq B, u' < u < v' < v\}$; intuitively, these are chords of other

[5] A *star* is a graph with vertices $V = V' \cup \{c\}$ where V' is an independent set and the *center* c is adjacent to each element of V'.

bunches that interlace with B. The left chord of (u, v, B) is the chord from this set with minimum u'; ties are broken in favor of largest v'. Right chords are analogously defined.

All the information needed to construct $G_{i,I}$ can be extracted from $J_i(Q_i)$ by a logspace computation.

5.5 The Constraint Graph G^*

The constraint graph contains two parts. One is the union over all i of the interlacing parity graphs $G_{i,I}$, and thus can be constructed in $\mathsf{FL}^{\mathsf{SL}}$. The other part accounts for the fact that more than one bunch may belong to the same bridge, and hence all such bunches must be placed consistently (on the same side) with respect to a path or fundamental cycle. This part has paths of length 1 or 2, called links, between anchor bunches and related bunches. Determining for each anchor bunch the length of the link, and its other endpoint, requires information about $G_{i,I}$ and computations described in subsection 3.2, and so the constraint graph G^* can also be constructed in $\mathsf{FL}^{\mathsf{SL}}$.

If G^* is not 2-colorable, then G is not planar. If G^* is 2-colorable, then the 2-coloring yields a combinatorial embedding of G_l. Testing whether G^* is 2-colorable (i.e. bipartite), and obtaining a 2-coloring if one exists, is known to be in $\mathsf{FL}^{\mathsf{SL}}$; see for instance [AG00].

5.6 The combinatorial embedding of G_l and of G

Given an undirected graph, a combinatorial embedding ϕ is a cyclic ordering of the edges around each vertex. Replace each edge (u, v) by directed arcs $\langle u, v\rangle$ and $\langle v, u\rangle$ to give the arc set A. Then ϕ is a permutation on A satisfying $\phi(\langle u, v\rangle) = \langle u, w\rangle$ for some w; i.e. ϕ cyclically permutes the arcs leaving each vertex. Let R be the permutation mapping each arc to its inverse. The combinatorial embedding ϕ is planar iff the number of cycles f in $\phi^* \triangleq \phi \circ R$ satisfies Euler's formula $n + f = m + 1 + c$. (Here, n, m, and c are the number of vertices, undirected edges, and connected components respectively.)

The 2-coloring of G^* partitions the non-P_i' edges with respect to P_i' in the obvious way (those that are to be embedded inside, and those that go outside). To further fix the cyclic ordering within each set, the algorithm of [RR94] computes, for each vertex v, a set of "tufts", which are the connected components of a graph that is easy to compute using the operations provided in subsection 3.2. Each tuft is labeled with a pair of vertices (again, these labels are easy to compute), and then the tufts are ordered by sorting these labels. (Sorting can be accomplished in logspace.) The cyclic ordering for tufts is either increasing or decreasing by labels, determined by the 2-coloring. This cyclic ordering then yields an ordering ϕ for all the arcs in G_l via a simple calculation.

To check planarity of ϕ, note that $c = 1$ (we are dealing with a 2-component), n and m are known, so the only thing left to compute is f. This can be computed in L as follows: Count the number of arcs a for which $a = c(a)$, where $c(a)$ is the lexicographically smallest arc on the cycle of ϕ^* containing a.

Since G_l is obtained from G by local replacements only, an embedding ϕ' of G can be easily extracted from the embedding ϕ of G_l: just collapse vertices of T_v back into v.

5.7 Merging embeddings of 2-components

It is well-known that a graph is planar iff its 2-components are planar; see for instance [Whi32]. To constructively obtain a planar combinatorial embedding of G from planar combinatorial embeddings of its 2-components, note that the ordering of edges around each vertex that is not a cut-vertex is fixed within the corresponding 2-component. At cut-vertices, adopt the following strategy: Let w be a cut-vertex present in d different 2-components $(u_1, v_1), (u_2, v_2), \ldots (u_d, v_d)$. The edges of w in each of these components are ordered according $\phi_1, \ldots, \phi_d$. Let x_i be the smallest neighbor of w in the 2-component (u_i, v_i). The orderings can be pasted together in $\mathsf{FL}^{\mathsf{SL}}$ as follows:

$$\begin{aligned} \phi(w, z) = {} & \phi_j(w, z) && \text{if } z \text{ is in the 2-component } (u_j, v_j) \text{ and } z \neq x_j \\ \phi(w, x_j) = {} & \phi_{j+1}^{-1}(w, x_{j+1}), && \\ \phi(w, x_d) = {} & \phi_1^{-1}(w, x_1). && \end{aligned}$$

Since the incidence of 2-components and cut-vertices in a connected graph gives rise to a tree, this strategy correctly embeds a connected graph. The planar combinatorial embedding of different connected components can be constructed independent of each other.

6 Open problems

- Given the complexity of the SL algorithm presented in this paper, the first question that comes to mind is: Is there a simpler SL algorithm for planarity? One of the main objectives in the algorithm of [RR94] was to minimize the amount of *work* done; their algorithm performs almost linear work on a CRCW PRAM with $O(\log n)$ processors. However, this low work does not immediately translate to any factor essential for computability within SL. Thus it is entirely plausible that a simpler SL algorithm exists. The algorithm of [JS82a, JS82b] is considerably simpler, but only yields an NL implementation. A simpler SL algorithm will go a long way in explaining the true connection between planarity testing and UGAP.

- Is planarity testing hard for SL? Is it in L? Until these classes are proved to coincide, there still remains some room for improvement in the bounds we present in this paper. For the class of series-parallel graphs, which forms a nontrivial subclass of planar graphs, this question has recently been completely resolved [JLR01]; it is shown that testing if a graph is a series-parallel graph is complete for L.

  There are a few other well-studied problems in SL that are not known to be in L or to be complete for SL. Among the most interesting of these is the undirected connectivity problem (called UCONN in [Wig92]). Some other problems in this situation are listed in the Open Problems section of [AG00], including the problems of recognizing interval graphs and permutation graphs.

- Can any of the techniques used here be extended to construct embeddings of small genus graphs? For instance, what is the parallel complexity of checking if a graph has genus 1, and if so, constructing a toroidal embedding?

Acknowledgments

We thank Pierre McKenzie for directing us to a simplification in our hardness proof. We also thank the anonymous referee for helpful comments.

References

[AG00] C Àlvarez and R Greenlaw. A compendium of problems complete for symmetric logarithmic space. *Computational Complexity*, 9:73–95, 2000.

[AKL+79] Romas Aleliunas, Richard M Karp, Richard J Lipton, Laszlo Lovász, and Charles Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pages 218–223. IEEE, 1979.

[AM00] Eric Allender and Meena Mahajan. The complexity of planarity testing. In *Proceedings of the 17th International Symposium on Theoretical Aspects of Computer Science STACS, LNCS Volume 1770*. Springer-Verlag, 2000.

[ATSWZ02] Roy Armoni, Amnon Ta-Shma, Avi Wigderson, and Shiyu Zhou. An $O((\log n)^{\frac{4}{3}})$ space algorithm for (s,t) connectivity in undirected graphs. *Journal of the ACM*, 47(2):294–311, March 2002.

[BL76] K Booth and G Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[CM87] S A Cook and P McKenzie. Problems complete for L. *Journal of Algorithms*, 8:385–394, 1987.

[Coo85] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[CSV84] Ashok Chandra, Larry Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13(2):423–439, 1984.

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

[Ete97] Kousha Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and System Sciences*, 54(3):400–411, Jun 1997.

[Eve79] S. Even. *Graph Algorithms.* Computer Science Press, 1979.

[Gib85] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[GW96] A. Gál and A. Wigderson. Boolean vs. arithmetic complexity classes: randomized reductions. *Random Structures and Algorithms*, 9:99–111, 1996.

[HT74] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM*, 21:549–568, 1974.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, Oct 1988.

[JLR01] Andreas Jakoby, Maciej Liskiewicz, and Rudiger Reischuk. Space efficient algorithms for series-parallel graphs. In *Symposium on Theoretical Aspects of Computer Science*, pages 339–352, 2001.

[JS82a] Joseph Ja'Ja' and Janos Simon. Parallel algorithms in graph theory: Planarity testing. *SIAM Journal on Computing*, 11:314–328, 1982.

[JS82b] Joseph Ja'Ja' and Janos Simon. Space efficient algorithms for some graph-theoretic problems. *Acta Informatica*, 17:411–423, 1982.

[KL82] Richard M Karp and Richard J Lipton. Turing machines that take advice. *L' Enseignement Mathématique*, 28:191–210, 1982.

[KW93] M. Karchmer and A. Wigderson. On span programs. In *Proceedings of the 8th Conference on Structure in Complexity Theory*, pages 102–111. IEEE Computer Society Press, 1993.

[LEC67] A Lempel, S Even, and I Cederbaum. An algorithm for planarity testing in graphs. In *Theory of Graphs: International Symposium*, pages 215–232, New York, 1967. Gordon and Breach.

[MSV86] Y Maon, B Schieber, and U Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theoretical Computer Science*, 47:277–296, 1986.

[MSV99] Meena Mahajan, P R Subramanya, and V Vinay. A combinatorial algorithm for Pfaffians. In *Proceedings of the Fifth Annual International Computing and Combinatorics Conference COCOON, LNCS Volume 1627*, pages 134–143. Springer-Verlag, 1999. DIMACS Technical Report 99-39.

[NSW92] N Nisan, E Szemeredi, and A Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *Proceedings of the 33rd Annual Smposium on Foundations of Computer Science*, pages 24–29. IEEE Computer Society Press, 1992.

[NTS95] N. Nisan and A. Ta-Shma. Symmetric Logspace is closed under complement. *Chicago Journal of Theoretical Computer Science*, 1995.

[RA00] K. Reinhardt and E. Allender. Making nondeterminism unambiguous. *SIAM Journal of Computing*, 29:1118–1131, 2000.

[Rei84] John Reif. Symmetric complementation. *Journal of the ACM*, 31(2):401–421, 1984.

[RR94] Vijaya Ramachandran and John Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49:517–561, 1994.

[Sak96] Michael Saks. Randomization and derandomization in space-bounded computation. In *Proceedings of the 11th Annual Conference on Computational Complexity*, pages 128–149. IEEE Computer Society, 1996.

[Sav70] Walter J Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, April 1970.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[Tut70] W. T. Tutte. Toward a theory of crossing numbers. *Journal of Combinatorial Theory*, 8:45–53, 1970.

[Tut84] W. T. Tutte. *Graph Theory*, volume 21 of *Encyclopedia of Mathematics and Its Applications*. Addison-Wesley, 1984.

[Whi32] H Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34:339–362, 1932.

[Whi84] A. White. *Graphs, Groups, and Surfaces*, volume 8 of *Mathematics Studies*. North-Holland, 1984.

[Wig92] A. Wigderson. The complexity of graph connectivity. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 629 of *Lecture Notes in Computer Science*, pages 112–132. Springer, 1992.

[Wil85] C. Wilson. Relativized circuit complexity. *Journal of Computer and System Sciences*, 31:169–181, 1985.