The complexity of planarity testing

Eric Allender*[1] and Meena Mahajan**[2]

[1] Dept. of Computer Science, Rutgers University, Piscataway, NJ, USA.
allender@cs.rutgers.edu
[2] The Institute of Mathematical Sciences, Chennai 600 113, INDIA.
meena@imsc.ernet.in

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 $\mathsf{L} = \mathsf{SL}$ [25]. The upper bound of SL matches the lower bound of L in the context of (nonuniform) circuit complexity, since L/poly is equal to $\mathsf{SL}/\mathrm{poly}$.
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 [22], although planarity checking for degree-three graphs had been shown to be in SL [23, 20].

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 [10] and (via another approach) by combining the results of [16, 4, 8]. In the context of parallel computation, a logarithmic-time CRCW-PRAM algorithm was presented by Ramachandran and Reif [22] that performs almost linear work.

From the perspective of computational complexity theory, however, the situation has been far from clear. The best upper bound on the complexity of planarity 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 [22]. In a recent survey of problems in the complexity class SL [2], the planarity problem for graphs of *bounded degree* is listed as belonging to SL, but this is based on the claim in [23] that checking planarity for bounded degree graphs is in the "Symmetric Complementation Hierarchy", and on the fact that SL is closed under complement [20] (and thus this hierarchy collapses to SL). However, the algorithm presented in [23] actually works only for graphs of degree 3, and

* Supported in part by NSF grant CCR-9734918.
** 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.

no straightforward generalization to graphs of larger degree is known. (This is implicitly acknowledged in [22, pp. 518–519].) Interestingly, Mario Szegedy has pointed out to us (personal communication) that an algebraic structure proposed by Tutte [28], when combined with more recent results about span programs and counting classes [14], gives a $\oplus\mathsf{L}$ algorithm for planarity testing. It is listed as an open question by Ja'Ja' and Simon [13] if planarity is in NL, although the subsequent discovery that NL is closed under complementation [11, 27] allows one to verify that one of the algorithms of [12, 13] 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 observe that the algorithm of Ramachandran and Reif can be implemented in SL.

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

Recall that

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

$$\mathsf{SL} \subseteq \oplus\mathsf{L}.$$

(See [14].) L (respectively SL, NL) denotes deterministic (respectively symmetric, nondeterministic) logarithmic space, 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 nondeterministic logspace machines with an odd 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 $\oplus\mathsf{L}/\mathrm{poly}$ [9].

This essentially solves the question of planarity 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 $\log^2 n$ [26], through $\log^{1.5} n$ [19], to $\log^{4/3} n$ [3]. It is suspected that this trend will continue to eventually reach $\log n$.
- *UGAP* can be solved in randomized logspace [1]. Recent developments in derandomization techniques have led many researchers to conjecture that randomized logspace is equal to L [25].

In the context of nonuniform complexity theory (for example, as explored in [15, 5]), the corresponding nonuniform complexity classes L/poly and $\mathsf{SL}/\mathrm{poly}$ are equal.[1] Hence in this setting, the computational complexity of planarity 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

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

from [17] 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 NL $\subseteq$ UL/poly [24]), and then use the algorithm in [17] 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. In Section 3 we sketch the main parts of our analysis, showing that the algorithm of [22] can be implemented in SL.

2 Hardness of planarity testing

The following problem is known to be complete for L:

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.*

The hardness of this problem for L was shown in [6], where only an NC^1 reduction is claimed. However, it is easy to see that this problem is actually hard under uniform projections as well. (To see this, consider any L machine M; assume that configurations are time-stamped, that there is a unique accepting configuration c_h whose successor is itself (with the next time-stamp), and that M decides within n^k time for some constant k. Construct the polynomially sized computation graph G where time-stamped configurations are the vertices, and edges denoting machine moves are labeled by input bits or their negations. M accepts its input if and only if $(G, (c_0, 0), (c_h, n^k))$ is an instance of UFA, where c_0 is the initial configuration.)

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.

We have thus proved the following theorem:

Theorem 1. *Planarity testing is hard for* L *under projections.*

It is worth noting that planarity testing remains hard for L even for graphs of degree 3. This does not follow from the construction given above, but can be shown by modifying a construction in [7]. Details will appear in the final paper.

3 The SL algorithm for planarity testing and embedding

We describe here how the algorithm of Ramachandran and Reif [22] can be implemented in SL. The algorithm of Ramachandran and Reif is complex, and it

involves a number of fairly involved technical definitions. Due to space limitations, it is not possible to present all of the necessary technical definitions here. Therefore, we have written this section so that it can be read as a companion to [22]. We use the same notation as is used in [22], and we will show that each step of their algorithm can be computed in L^{SL}, or by NC^1 circuits with oracle gates for the reachability problem in undirected graphs. Since SL is closed under NC^1 reducibility and logspace-Turing reducibility [20], it follows that the entire algorithm can be implemented in SL.

Our approach will be as follows. First, we present some general-purpose algorithms for operating on graphs and trees. Next, we show how an open ear decomposition can be computed in SL; the parallel algorithm to perform this step is also fairly complex, and space limitations prevent us from presenting all of the details for this step. Therefore, we present this section so that it can be used as a companion to the presentation of the open ear decomposition algorithm of Ramachandran as given in [21]. Finally, we go through the other steps of the algorithm of [22].

3.1 Elementary graph computations in SL

Our method of exposition in this subsection 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.

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)? [23, 20, 2].
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)? [20]

Given a graph G and a spanning tree T, the following conditions can be checked in SL:

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, ET_r? Does $x \longrightarrow y$ precede $y \longrightarrow x$? (In

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? (See [22] for a definition of "bridge". 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 which 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? (See [22] for a definition of "interlacing". 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.)

3.2 Finding an open ear decomposition

We follow the exposition from [21]. The algorithm in Figure 1 finds an open ear decomposition of a graph: it labels each edge e by the number of the first ear containing e.

input: biconnected graph G; vertices v, r; edge e

1: Find a spanning tree T, and number the vertices in preorder from 0 to $n-1$ with respect to root r.

2: Label the non-tree edges:

- **2.1** For each vertex v other than r, find $low(v)$. Mark v if $low(v) < parent(v)$.
- **2.2** Construct multigraph H: $V_H = V \backslash \{r\}$;
  For each $e \notin T$ with distinct base vertices x, y, put edge (x, y) in E_H;
  For each $e \notin T$ with a single base vertex x, put edge (x, x) in E_H;
- **2.3** Find the connected components of H. Label a component C by $preorder(parent(a))$ for some (arbitrary) $a \in C$.
- **2.4** Within each component C, find a spanning tree T_C, root it at a marked vertex if one exists, and preorder the vertices $0, 1, \ldots, k$.
- **2.5** For $e = (parent(y), y) \in T_C$, label e with the pair $(label(C), y)$.
  For $e \notin T_C$, label e with the pair $(label(C), k+1)$.
- **2.6** For $e \notin T$, label e with the label of the corresponding edge in H.
- **2.7** Sort labels in non-decreasing order and relabel as $1, 2, \ldots$.

3: Label a tree edge $(parent(v), v)$ by the smallest label of any non-tree edge incident on a descendant of v (including v).

4: Relabel the non-tree edge labeled 1 by the label 0.

Fig. 1. Open Ear Decomposition Algorithm

In this procedure, most of the computations involve computing spanning trees, finding connected components, preordering a tree, and sorting labels, all of which can be performed in SL. The only new steps here are the computation of base vertices and low vertices. These are also easily seen to be computable in SL using the operations from Subsection 3.1; note that

1. z is a base vertex of non-tree edge $e = (x, y)$ if $parent(z) = lca(x, y)$ and either x or y is a descendant of z.
2. $low(v)$ is the smallest w such that $w \in C_e$ for a non-tree edge $e = (e_1, e_2)$ with e_1 or e_2 a descendant of v.

3.3 An overview of the algorithm

The planarity testing algorithm of Ramachandran and Reif [22] is outlined in Figure 2. If G^* is not 2-colorable (step 2.7), or if step 2.8 yields an embedding that is not planar, then the input graph is not planar. Otherwise, this procedure gives a planar combinatorial embedding of the input graph. For the complete algorithm and definitions of the terms used above, see [22].

The emphasis in [22] 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 biconnected component. Step 2.1 was addressed in subsection 3.2. Step 2.4 has been discussed in subsection 3.1. The remaining steps are discussed in the following subsections.

1: Decompose the input graph into its biconnected components.
2: For each biconnected 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 biconnected components.

Fig. 2. Planarity Testing Algorithm

3.4 Constructing the directed *st*-numbering graph G_{st}

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

- The edge (s, t) is oriented $s \longrightarrow t$.
- Let the two endpoints of an ear P_i be the vertices u and v.
  - If $ear(v) < ear(u)$, then all edges on P_i are oriented to form a path from v to u.
  - If $ear(u) < ear(v)$, then all edges on P_i are oriented to form a path from u to v.
  - If $ear(u) = ear(v)$, then all edges on P_i are oriented to form a path from u to v if u comes before v in the orientation on $P_{ear(v)}$, and are oriented to form a path from v to u otherwise.

G_{st} is acyclic, and every vertex lies on a path from s to t [18].

We show that G_{st} can be computed from G and D in logarithmic space.

Orienting the edges in ear P_i is easy if $ear(u) \neq ear(v)$. The routine shown in Figure 3 shows how to orient the edges if $ear(u) = ear(v) = i'$. It is clear that this can be implemented in L.

3.5 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

Input (D, i)
Find the endpoints u and v of P_i. Let $ear(u) = ear(v) = i'$.
Note that P_i is oriented from u to v iff u comes before v in $P_{i'}$.

1. Let u' and v' be the endpoints of $P_{i'}$. Compute the bit B such that
(u comes before v in P_i) iff [(u' comes before v' in i') $\oplus$ B].
(This can be done in logspace since $P_{i'}$ is a path. The routine can start at u' and see if it encounters u or v first.)

2. If $ear(v') \neq ear(u')$
Then we can compute the orientation directly.
Else
Let $i'' = ear(v') = ear(u')$
Let the endpoints of $P_{i''}$ be u'' and v''.
Find and remember the bit B' such that
(u' comes before v' in $P_{i''}$) iff [(u'' comes before v'' in $P_{i''}$) $\oplus B'$]
(At this point, we can forget about u' and v'.)
$u' := u''$; $v' := v''$; $i' := i''$; $B := B \oplus B'$
GO TO statement 2.
end.procedure

Fig. 3. Orienting an ear.

which 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 3.1, 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.

3.6 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 [22], 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 3.1), this classification is also in SL.

In the nomenclature of [22], 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 3.1.

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 3.1, so $H(B)$ can be computed in $\mathsf{FL}^{\mathsf{SL}}$.

3.7 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 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 which 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 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.

3.8 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 3.1, 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 [2].

3.9 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 $\phi^* \stackrel{\triangle}{=} \phi \circ R$ satisfies Euler's formula $n + f = m + 1 + c$. (n, m, c are the number of vertices, undirected edges, 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 [22] 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 3.1. 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 can be computed in SL [20], 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.

3.10 Merging embeddings of 2-components

It is well-known that a graph is planar iff its biconnected components are planar; see for instance [29]. 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 which 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 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:

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

4 Open problems

- 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.
- 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.

References

1. R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. 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.
2. C. Àlvarez and R. Greenlaw. A compendium of problems complete for symmetric logarithmic space. Technical Report ECCC-TR96-039, Electronic Colloquium on Computational Complexity, 1996.
3. R. Armoni, A. Ta-Shma, A. Wigderson, and S. Zhou. $SL \subseteq L^{\frac{4}{3}}$. In *Proceedings of the 29th Annual Symposium on Theory of Computing*, pages 230–239. ACM, 1997.
4. 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.
5. A. Chandra, L. Stockmeyer, and U. Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13(2):423–439, 1984.
6. S. A. Cook and P. McKenzie. Problems complete for L. *Journal of Algorithms*, 8:385–394, 1987.
7. K. Etessami. Counting quantifiers, successor relations, and logarithmic space. *Journal of Computer and System Sciences*, 54(3):400–411, Jun 1997.
8. S. Even and R. Tarjan. Computing an *st*-numbering. *Theoretical Computer Science*, 2:339–344, 1976.
9. A. Gál and A. Wigderson. Boolean vs. arithmetic complexity classes: randomized reductions. *Random Structures and Algorithms*, 9:99–111, 1996.

10. J. Hopcroft and R. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21:549–568, 1974.
11. N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, Oct 1988.
12. J. Ja'Ja' and J. Simon. Parallel algorithms in graph theory: Planarity testing. *SIAM Journal on Computing*, 11:314–328, 1982.
13. J. Ja'Ja' and J. Simon. Space efficient algorithms for some graph-theoretic problems. *Acta Informatica*, 17:411–423, 1982.
14. 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.
15. R. M. Karp and R. J. Lipton. Turing machines that take advice. *L' Ensignement Mathématique*, 28:191–210, 1982.
16. 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.
17. M. 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.
18. Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and *st*-numbering in graphs. *Theoretical Computer Science*, 47:277–296, 1986.
19. 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.
20. N. Nisan and A. Ta-Shma. Symmetric Logspace is closed under complement. *Chicago Journal of Theoretical Computer Science*, 1995.
21. V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity. In J. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaumann, 1993.
22. V. Ramachandran and J. Reif. Planarity testing in parallel. *Journal of Computer and System Sciences*, 49:517–561, 1994.
23. J. Reif. Symmetric complementation. *Journal of the ACM*, 31(2):401–421, 1984.
24. K. Reinhardt and E. Allender. Making nondeterminism unambiguous. In *38 th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 244–253, 1997. to appear in SIAM J. Comput.
25. M. 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.
26. W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, April 1970.
27. R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.
28. W. T. Tutte. Toward a theory of crossing numbers. *Journal of Combinatorial Theory*, 8:45–53, 1970.
29. H. Whitney. Non-separable and planar graphs. *Transactions of the American Mathematical Society*, 34:339–362, 1932.