Space Efficient Algorithms for Series-Parallel Graphs

Andreas Jakoby[1,2] Maciej Liśkiewicz[1] Rüdiger Reischuk[1]
[1]Universität zu Lübeck [2]University of Toronto

Abstract

The subclass of directed *series-parallel graphs* plays an important role in computer science. To determine whether a graph is series-parallel is a well studied problem in algorithmic graph theory. Fast sequential and parallel algorithms for this problem have been developed in a sequence of papers. For series-parallel graphs methods are also known to solve the reachability and the decomposition problem time efficiently. However, no dedicated results have been obtained for the the space complexity of these problems – the topic of this paper.

For this special class of graphs, we develop deterministic algorithms for the *recognition, reachability, decomposition* and the *path counting problem* that use only logarithmic space. Since for arbitrary directed graphs reachability and path counting are believed not to be solvable in log-space the main contribution of this work are novel deterministic path finding routines that work correctly in series-parallel graphs, and a new characterization of series-parallel graphs by forbidden subgraphs.

We then sketch how these results can be generalised to extension of the series-parallel graph family: to graphs with multiple sources or multiple sinks and to the class of *minimal vertex series-parallel graphs.*

It it also shown that the space bounds are best possible, i.e. the decision problems are $\mathcal{L}$-complete with respect to $\mathcal{AC}^0$-reductions.

Finally, we discuss implications of theses small space bounds for the parallel time complexity of series-parallel graphs.

CLASSIFICATION: efficient algorithms, algorithmic graph theory, computational complexity

[1]Institut für Theoretische Informatik (`http://www.tcs.mu-luebeck.de/`),
email:`jakoby/liskiewi/reischuk@tcs.mu-luebeck.de`
[2]Department of Computer Science, `email: jakoby@cs.toronto.edu`

1 Introduction

All graphs $G = (V, E)$ that will be considered in this paper are directed. n denotes the number of vertices V of G and m the number of edges E. A well studied subclass of graphs are the *series-parallel graphs,* for which different definitions and characterizations have been given [6]. We will consider the basic class, sometimes also called *two terminal series-parallel graphs,* that are most important for applications in program analysis.

Definition 1 *$G = (V, E)$ is a series-parallel graph if either G is a line graph of length at most 1, that is a single node or a pair of nodes connected by a single edge, or there exist two series-parallel graphs $G_i = (V_i, E_i)$, $i = 1, 2$, with sources $v_{in,i}$, and sinks $v_{out,i}$ such that $V = V_1 \cup V_2$, $E = E_1 \cup E_2$, and either*

(A) **parallel composition**: *$v_{in} = v_{in,1} = v_{in,2}$ and $v_{out} = v_{out,1} = v_{out,2}$, or*

(B) **series composition**: *$v_{in} = v_{in,1}$ and $v_{out} = v_{out,2}$ and $v_{out,1} = v_{in,2}$.*

Note that such a series-parallel graph has exactly on source and one sink. We will assume that the graphs are specified by a list of their edges, but we put no restrictions on their ordering. In particular, we do not require that this ordering in any way reflects the structure of the series-parallel composition operations. Otherwise, recognizing and handling series-parallel graphs becomes quite easy. The correctness and efficiency of the algorithms presented below will not depend on the representation of the input graphs. For example, one could use adjacency-matrices as well.

Series-parallel graphs are suitable to describe the information flow within a program that is based on sequential and parallel composition. The graphical description of a program helps to decide whether it can be parallelized and to generate schedules for a parallel execution.

To determine if a given graph G belongs to the class of series-parallel graphs is a basic problem in algorithmic graph theory. An optimal linear time sequential algorithm for this problem has been developed by Valdes, Tarjan, and Lawler in [15] long time ago. Also, fast parallel algorithms have been published. He and Yesha have presented an EREW PRAM algorithm working in time $O(\log^2 n)$ time while using $n + m$ processors [12]. Eppstein has reduced the time bound constructing an algorithm that takes only $O(\log n)$ steps on the stronger CRCW PRAM model (concurrent instead of exclusive read and write) that requires $C(m, n)$ processors [11]. Here $C(m, n)$ denotes the number of processors necessary to compute the connected components of a graph in logarithmic time. Finally, Bodleander and Fluiter have presented an EREW PRAM algorithm using $O(\log n \cdot \log^* n)$ time and $O(n + m)$ operations [5].

The space complexity of this problem, however, has not been investigated successfully so far. In this paper we give an answer to this question.

The *decompositon of a series-parallel graph* is quite useful to decide other graph properties. Hence, an important task is also to compute such a decomposition efficiently. In [15] a linear-time sequential algorithm for decomposing series-parallel graphs has been given. We will show that this task can be done in small space as well.

For general graphs, the *reachability problem,* that is the question whether there exists a path between a given pair of nodes, is the classical $\mathcal{NL}$-complete problems. By well known simulations, for the parallel time complexity one can infer a logarithmic upper bound on CRCW PRAMs. The reachability problem restricted to series-parallel graphs, however, can be solved in logarithmic time already by an EREW PRAM using the minimal number $(n + m)/\log n$ of processors [15]. Certain graph properties like acyclicity are also complete for $\mathcal{NL}$, while for other problems their computational complexity is still unsolved. Recently, Allender and Mahajan have made a major step in classifying the computational complexity of planarity testing showing that this problem is hard for $\mathcal{L}$ and belongs to $\mathcal{SL}$ (symmetric Logspace) [3]. They leave as an open problem to close the gap between the lower bound and the upper bound. In this paper we determine the computational complexity of a nontrivial subproblem of planarity testing precisely, namely for series-parallel graphs it will be proven to be $\mathcal{L}$-complete.

For $\mathcal{L}$ several simple graph problems are known to be complete with respect to $\mathcal{AC}^0$-reductions: for example, whether a graph is a forest or even a tree, or whether in a given forest G a pair of nodes belongs to the same tree (for a comlete list see [9, 13]). In this paper we will prove three problems for series-parallel graphs as $\mathcal{L}$-complete as well: the *recognition problem,* the *reachability problem,* and *counting the number of paths mod 2.* While the hardness of these problems can be obtained in a straightforward way, it requires a lot of algorithmic effort to prove that the lower bound can actually be achieved. Thus, the main technical contribution of this paper are new graph-theoretical notions and algorithmic methods that allow us to solve these problems using only logarithmic space.

Furthermore, not only decision problems for series-parallel graphs turn out to be tractable. A decomposition of such graphs can be computed within the same space bound as well. For general graphs counting the number of paths is one of the generic complete problems for the class $\#\mathcal{L}$ [2]. Thus, this problem is not computable in $\mathcal{FL}$, the functional deterministic log-space complexity class, unless certain hierarchies collapse. We will prove that restricting to series-parallel graphs the counting problem can be solved in $\mathcal{FL}$. This will be achieved by combining our space efficient reachability decision procedure with a modular representation of numbers requiring only little space, and the recent result that a chinese remaindering representation can be converted to the standard binary one in logarithmic space [8].

Because of the relation between $\mathcal{L}$ and parallel time complexity classes defined by the EREW PRAM model (see [14]) these new algorithms can be modified to solve these problems in logarithmic time on EREW PRAMs as well. Finally, these results can also be extended to generalizations of series-parallel graphs: multiple source or multple sink, and minimal vertex-series-parallel graphs.

This paper is organized as follows. In Section 2 we will prove the $\mathcal{L}$-hardness of the reachability and the recognition problem. Procedures solving these problems within logarithmic space will be described in detail in Section 3. Section 4 outlines an algorithm that generates an edges-ordering that reflects the structure of a given series-parallel graph. Based on this ordering we sketch a decomposition algorithm in Section 5. In Section 6, we combine the methods presented so far to solve the path counting problem. Finally, in Section 7 it will be indicated how this results can be extended to generalizations of series-parallel graphs. The paper ends with some conclusions and open problems.

2 Hardness Results

To establish meaningful lower bounds for the deterministic space complexity class $\mathcal{L}$ one has to restrict the concept of polynomial time many-one reductions to simpler functions. We consider the usual requirement that the reducing function f can be computed in $\mathcal{AC}^0$. The $\mathcal{L}$-hardness for series-parallel graphs can be shown in a direct way.

Theorem 1 *The following problems are hard for $\mathcal{L}$ under $\mathcal{AC}^0$ reducibility: (i) recognition of series-parallel graphs, and (ii) reachability in series-parallel graphs.*

Proof: Let L be a language in $\mathcal{L}$ and M a logarithmic space-bounded deterministic Turing machine accepting L and using at most n^k steps on inputs X of length n, where k is a fixed exponent. We may assume that M has unique final configurations C_{acc}, the accepting one, and C_{rej}, the rejecting one. In addition, all configurations C of M on X are time-stamped, that means are actually tuples (C,t) with $0 \leq t \leq n^k$. Then the successor configuration of (C,t) is $(C',t+1)$ if $t < n^k$ and M can move in one step from C to C'. If C is a final configuration and $t < n^k$ then $(C,t+1)$ is the successor of (C,t). For input X we construct a directed graph G_X, where the time-stamped configurations (C,t) are the vertices of G_X and edges represent (the inverse of) the successor relation: G_X contains the edge $((C',t+1),(C,t))$ iff $(C',t+1)$ is a successor of (C,t). Obviously, G_X is a forest consisting of trees with roots of the form (C,n^k).

To prove the hardness of the rechability problem we augment G_X by two new nodes u and v. For every configuration (C,n^k) the edge $(u,(C,n^k))$ is added, and for every leaf (C,t) the edge $((C,t),v)$. It is easy to see that the resulting graph is series-parallel with source u and sink v. Furthermore, it contains a path from (C_{acc},n^k) to $(C_{init},0)$, where $(C_{init},0)$ represents the starting configuration of M iff M accepts X. The reduction itself can be computed in $\mathcal{AC}^0$.

In order to prove the hardness of the recognition problem we modify G_X as follows. For every configuration $C \notin \{C_{acc},C_{rej}\}$ add an edge $((C_{acc},n^k),(C,n^k))$ to G_X. Also add the edge $((C_{init},0),(C_{rej},n^k))$. Augment the graph by a new node v, and for every leaf $(C,t) \neq (C_{init},0)$ add the edge $((C,t),v)$. If M accepts X then the new graph is series-parallel with source (C_{acc},n^k) and sink v. Otherwise, it contains a cycle connecting $(C_{init},0)$ and (C_{rej},n^k). ∎

3 Recognition and Reachability in Logspace

Establishing corresponding upper bounds is not obvious at all. We will give a new characterization of series-parallel graphs by forbidden subgraphs and exploit the structure of internal paths very thoroughly.

Assume that the nodes of the input graph G are represented by the set of numbers $\{1, 2, \ldots, n\}$. G is given by a list of edges $(i_1, j_1), (i_2, j_2), \ldots (i_m, j_m)$, where i_k, j_k are binary representations of the names of the nodes. Let $\mathbf{pred(v)}$ denote the set of direct predecessors of v, and $\mathbf{pred(v, i)}$ the i-th direct predecessor of v according to the ordering implicitly given by the specification of G. Similarly, let $\mathbf{succ(v)}$ and $\mathbf{succ(v, i)}$ be the set of direct successors of v, resp. its i-th direct successor. $\mathrm{pred}(v, i)$ and $\mathrm{succ}(v, i)$ can be computed deterministically in logarithmic space: the Turing machine searches through the list of edges looking for the i-th entry that starts with v.

Define $\mathbf{pred^+(v)}$ (resp. $\mathbf{succ^+(v)}$) as the transitive closure of $\mathrm{pred}(v)$ (resp. $\mathrm{succ}(v)$) not containing v, and $\mathbf{pred^*(v)}$:= $\mathrm{pred}^+(v) \cup \{v\}$ and $\mathbf{succ^*(v)}$:= $\mathrm{succ}^+(v) \cup \{v\}$. To shorten the notation, let us introduce the predicate $\mathbf{PATH(u, v)}$ being true iff the given graph G possesses a path from node u to v. Thus,

$$\mathtt{PATH}(u, v) \quad \Longleftrightarrow \quad u \in \mathrm{pred}^*(v) \quad \Longleftrightarrow \quad v \in \mathrm{succ}^*(u) \ .$$

Remember that deciding PATH for arbitrary graphs is $\mathcal{NL}$-complete. To construct a deterministic space efficient algorithm solving this problem for series-parallel graphs we introduce the following concepts:

$$\begin{array}{lll}
\text{lm-down}(v) & := & \text{the maximal acyclic path } u_1, u_2, u_3, \ldots, u_l \text{ with } u_1 = v, \text{ and } u_{i+1} = \mathrm{succ}(u_i, 1), \\
\text{lm-up}(v) & := & \text{the maximal acyclic path } u_1, u_2, u_3, \ldots, u_l \text{ with } u_1 = v, \text{ and } u_{i+1} = \mathrm{pred}(u_i, 1), \\
\text{lm-pred}^*(v) & := & \{u \mid \text{lm-down}(u) \cap \text{lm-up}(v) \neq \emptyset\}, \quad \text{and} \\
\text{lm-succ}^*(v) & := & \{u \mid \text{lm-down}(v) \cap \text{lm-up}(u) \neq \emptyset\} \ .
\end{array}$$

Here, "lm" stands for *left-most*, that means in each node u_i the path follows the first edge as specified by the representation of G. A path being acyclic requires that all its nodes u_i are different. Thus, a maximal acyclic down-path either ends in a sink or stops immediately before hitting a node as the left-most successor a second time. These sets can be decided by the following simple procedure.

```
procedure  membership-test1[u ∈ lm-down(v)]
let n be the number of nodes in G
let x := v;  i := 1;
while  x ≠ u  and  |succ(x)| > 0  and  i ≤ n  do
   let x := succ(x, 1);  i := i + 1  od
if x = u then return TRUE else return FALSE
```

The algorithm for testing whether $u \in \text{lm-up}(v)$ is just the symmetric dual. To check if $u \in \text{lm-pred}^*(v)$ one can use the following algorithm calling the procedure above for the appropriate membership problems.

```
procedure  membership-test2[u ∈ lm-pred*(v)]
result := FALSE
forall nodes x in G do
   if  x ∈ lm-down(u)  and  x ∈ lm-up(v) then let  result := TRUE  od
return result
```

In the dual way we can test whether $u \in \text{lm-succ}^*(v)$. Hence it follows:

Lemma 1 *For an arbitrary graph G and node v the membership problem for the sets* $\text{lm-down}(v)$, $\text{lm-up}(v)$, $\text{lm-pred}^*(v)$, *and* $\text{lm-succ}^*(v)$ *can be solved deterministically in logarithmic space.*

A graph G is called ***st*-connected** if G is has a unique source named s and a unique sink named t, and for every node v it holds: $\mathtt{PATH}(s, v)$ and $\mathtt{PATH}(v, t)$.

We start with the following simple procedure, called preliminary-test, which for an acyclic graph G returns TRUE iff G is *st*-connected. Otherwise, if G contains a cycle $C = (v_1, v_2, \ldots, v_l)$ the procedure will detect the cycle if it is on a left-most path. In such a case the procedure outputs FALSE. Cycles which are not of this form will be not detected, and in such cases the procedure erroneously may output TRUE.

`procedure` preliminary-test(G)
`if not` [G has a unique source and a unique sink] `then return FALSE` and `exit`
`else let` s be the source and t the sink
`let` *result* :=`TRUE`
`forall` nodes v in G `do`
    `if` $t \notin$ lm-down(v) `or` $s \notin$ lm-up(v) `then let` *result* :=`FALSE od`
`return` *result*

Lemma 2 *The procedure* preliminary-test *can be implemented deterministically in logarithmic space. Moreover, if* preliminary-test(G) *outputs* `TRUE` *then G is st-connected. If it outputs* `FALSE` *then at least one of the following conditions holds:*
(*i*) *G has more then one source or more than one sink, or*
(*ii*) *G is st-connected, but it has a cycle.*

The proof of this lemma is straightforward and we omit it. Note that a graph G with output `TRUE` can still have a cycle. To detect this property is difficult for deterministic machines since this question can easily be shown to be $\mathcal{NL}$-complete. Therefore, we lock for a simpler task.

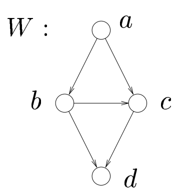

Figure 1: The forbidden minor W.

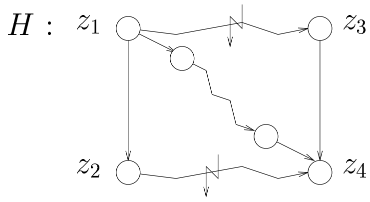

Figure 2: The forbidden induced subgraph H.

Let W denote the graph shown in Fig. 1. A graph W' is *homeomorphic* to W if it contains four distinct vertices a, b, c, d and pairwise internally vertex disjoint paths $P_{ab}, P_{ac}, P_{bd}, P_{cd}$ and P_{bc}. If G contains a homeomorphic image of W as a subgraph then W is called a *minor* of G. The following characterization of series-parallel graphs by forbidden minors has been known for long.

Lemma 3 ([10],[15]) *Let G be an st-connected acyclic graph. Then G is series-parallel iff W is not a minor of G.*

To make series-parallel graph recogniton space efficient, instead of searching for the forbidden minor W we will use the following characterization. Let H be a graph with four distinct nodes z_1, z_2, z_3, z_4 such that

1. $(z_1, z_2), (z_3, z_4)$ are edges of H and $\texttt{PATH}(z_1, z_4)$,
2. $\neg\ \texttt{PATH}(z_1, z_3)$ and $\neg\ \texttt{PATH}(z_2, z_4)$.

These condition are illustrated in Figure 2. In the following we will show how H can be used to determine whether a graph is series-parallel. We say that H is an *induced subgraph* of G if G contains four nodes z_1, z_2, z_3, z_4 which fulfill these connectivity conditions.

Theorem 2 *Let G be an st-connected acyclic graph. Then G is series-parallel iff it does not contain H as an induced subgraph.*

Proof: Let G be acyclic and st-connected with source s and sink t. We will show that G contains W as minor iff it contains H as an induced subgraph.

First assume that W is a minor and denote the corresponding nodes of G by a, b, c, d. Because of the acyclicity G cannot contain a path from d to c. Hence, on the path P_{bd} there must be an edge (b_1, b_2) such that

$$\texttt{PATH}(b_1, c) \quad \wedge \quad \neg\ \texttt{PATH}(b_2, c)\ .$$

Similarly, on the path P_{ac} one can find an edge (c_1, c_2) such that

$$\texttt{PATH}(b_1, c_2) \quad \wedge \quad \neg\, \texttt{PATH}(b_1, c_1)\ .$$

$\neg\, \texttt{PATH}(b_2, c)$ implies $\neg\, \texttt{PATH}(b_2, c_2)$. Hence, the nodes $z_1 := b_1$, $z_2 := b_2$, $z_3 := c_1$, and $z_4 := c_2$ fulfill the conditions from above.

To prove the opposite implication assume that nodes z_1, z_2, z_3, z_4 form the basis of H as an induced subgraph of G. Choose $b := z_1$ and $c := z_4$. Moreover, let a be the first common predecessor of b and z_3 according to left-most paths from b and z_3 back to the source of G. Since G is st-connected such a node a exists. Hence, we have found internally vertex disjoint paths P_{ab} and P_{az_3} from a to b and from a to z_3, respectively. Analogously, let d be the first common successor of z_2 and c according to the left-most path order which gives rise to vertex disjoint paths P_{z_2d} and P_{cd}. The acyclicity of G and $\neg\, \texttt{PATH}(z_1, z_3)$ and $\neg\, \texttt{PATH}(z_2, z_4)$ imply that the paths P_{ab}, $P_{ac} = P_{az_3}\ (z_3, c)$, $P_{bd} = (b, z_2)P_{z_2b}$, and P_{cd} have also to be internally vertex disjoint from the path $P_{bc} = P_{z_1z_4}$, in other words W is a minor of G. ∎

We will now deduce the key property that makes reachability in series-parallel graphs easier compared to arbitrary graphs. Although the parallel composition operator introduces a lot of nondeterminism into the structure of these graphs when trying to find a path from a node u to a node v this question can be solved by considering the unique lm-down-path starting at u and the unique lm-up-path starting in v and deciding whether these two intersect. In other words, it holds:

Theorem 3 *If G is series-parallel then for every node v it holds* $\mathrm{pred}^*(v) = \text{lm-pred}^*(v)$.

Proof: Assume that $\mathrm{pred}^*(v) \neq \text{lm-pred}^*(v)$ for some node v of G. We will show that then H has to be an induced subgraph of G – a contradiction to Theorem 2.

Obviously, v cannot be the source s of G. Since G is st-connected and acyclic every lm-down-path from an arbitrary node u has to terminate in the sink t. Thus, for t holds $\mathrm{pred}^*(t) = \text{lm-pred}^*(t) = V$, and hence $v \neq t$.

Let $u_1 \in \mathrm{pred}^*(v) \setminus \text{lm-pred}^*(v)$. Since every lm-up-path terminates in the source s we can conclude $u_1 \neq s$. Let $u_1, u_2, \ldots, u_k = t$ be the leftmost down-path $\text{lm-down}(u_1)$. $u_1 \notin \text{lm-pred}^*(v)$ implies that $u_i \neq v$ for all $i \in [1..k]$. Furthermore, let $v_1 = v, v_2, \ldots, v_\ell = s$ be the leftmost up-path $\text{lm-up}(v)$ from v.

Since $u_1 \in \mathrm{pred}^*(v_1)$ there exists a non-trivial path from u_1 to v_1. On the other hand, because of $v_\ell = s$ and $v_1 = v \neq s$ it holds $\neg\, \texttt{PATH}(v_1, v_\ell)$, and similarly because of $u_k = t$ and $v_1 \neq t$, $\neg\, \texttt{PATH}(u_k, v_1)$. Hence, there exist $i \in [1..k-1]$ and $j \in [1..\ell-1]$ such that

$$\texttt{PATH}(u_i, v_j), \quad \neg\, \texttt{PATH}(u_i, v_{j+1}), \quad \text{and} \quad \neg\, \texttt{PATH}(u_{i+1}, v_j).$$

The nodes $z_1 := u_i$, $z_2 := u_{i+1}$, $z_3 := v_{j+1}$, and $z_4 := v_j$ prove that H is an induced subgraph of G. ∎

Thus, if for some node v of G the relation $\mathrm{pred}^*(v) = \text{lm-pred}^*(v)$ is violated one can conclude that G is not series-parallel. This equality, however, can be tested space efficiently.

Lemma 4 *There exists a deterministic logarithmic space-bounded Turing machine that for arbitrary $v \in G$ decides whether* $\mathrm{pred}^*(v)$=$\text{lm-pred}^*(v)$.

Proof: Assume that $\mathrm{pred}^*(v) \neq \text{lm-pred}^*(v)$. First, we claim that there has to be an edge $(u, w) \in E$ such that $u \in \mathrm{pred}^*(v) \setminus \text{lm-pred}^*(v)$ and $w \in \text{lm-pred}^*(v)$. To see this, let $x \in \mathrm{pred}^*(v) \setminus \text{lm-pred}^*(v)$ and $u_1 = x, u_2, u_3, \ldots, u_k = v$ be a down-path from x to v. Obviously, $u_1, u_2, \ldots, u_k \in \mathrm{pred}^*(v)$, $u_1 \notin \text{lm-pred}^*(v)$, and $u_k \in \text{lm-pred}^*(v)$. Therefore, there exists an index $i \in [1..k-1]$ such that $u_i \in \mathrm{pred}^*(v) \setminus \text{lm-pred}^*(v)$ and $u_{i+1} \in \text{lm-pred}^*(v)$. This proves our claim.

Now it is easy to see that the following algorithm answers the question whether $\mathrm{pred}^*(v)$=$\text{lm-pred}^*(v)$:

```
procedure  equality-test[pred*(v) = lm-pred*(v)]
  result := TRUE
  forall edges (u,w) in G do
     if (u ∉ lm-pred*(v)) and (w ∈ lm-pred*(v)) then let result := FALSE od
  return result
```

∎

Corollary 1 *Let G be an st-connected graph with* $\mathrm{pred}^*(v) = \text{lm-pred}^*(v)$ *for every node v. Then reachability within G can be decided in $\mathcal{L}$.*

Proof: The identity $\text{pred}^*(v) = \text{lm-pred}^*(v)$. guarantees that for all w in G holds:

$$v \text{ is reachable from } w \iff \texttt{PATH}(w,v) \iff w \in \text{pred}^*(v) \iff w \in \text{lm-pred}^*(v).$$

Lemma 1 has shown that the membership problem for $\text{lm-pred}^*(v)$ can be solved space efficiently. ∎

We now define an algorithm that decides for an arbitrary graph G whether it is series-parallel.

```
procedure  SERIES-PARALLEL(G)
 1   call preliminary-test(G)
 2   if preliminary-test(G) returns FALSE then return FALSE and exit
 3   forall nodes v in G do
 4      if pred*(v) ≠ lm-pred*(v) then return FALSE and exit  od
 5   forall pairs of nodes x, y in G do
 6      if  x ∈ lm-pred*(y)  and  y ∈ lm-pred*(x)  then return FALSE and exit  od
 7   forall pairs of edges (z1, z2), (z3, z4), with z1 ≠ z4 do
 8      if  z1 ∈ lm-pred*(z4)  and  z1 ∉ lm-pred*(z3)  and  z2 ∉ lm-pred*(z4)
 9        then return FALSE and exit  od
10   return TRUE
```

To prove its correctness we argue as follows. From Lemma 2 one can conclude that the algorithm stops at line 2 and outputs `FALSE` if G has more then one source or more than one sink, or G is *st*-connected, but it has a cycle. Hence, G is not series-parallel and the answer `FALSE` is correct. On the other hand, if the procedure does not stop at line 2 then G is *st*-connected.

Furthermore, if SERIES-PARALLEL(G) outputs `FALSE` in line 4 then $\text{pred}^*(v) \neq \text{lm-pred}^*(v)$ for some node v. By Theorem 3 it follows that this answer is correct, too. If the algorithm continues, we can presuppose at the beginning of line 5 that G is *st*-connected and for any v it holds $\text{pred}^*(v) = \text{lm-pred}^*(v)$. In lines 5 and 6 we check whether G is acyclic, and stop if not. The answer will be correct since $\text{lm-pred}^*(y)$ contains all predecessors of a node y.

Let us recapitulate the conditions a graph G has to fulfill such that SERIES-PARALLEL(G) does not stop before line 7: G has to be *st*-connected, acyclic and for every pair of nodes x, y in G it holds: $\texttt{PATH}(y,x) \iff y \in \text{lm-pred}^*(x)$. This guarantees that in lines 7-9 the existence of H as an induced subgraph is tested correctly. Finally, since all tests applied can be performed in deterministic logarithmic space we can conclude.

Theorem 4 *The question whether a graph is series-parallel can be decided in $\mathcal{L}$.*

4 An Edge Ordering Algorithm

When specifying a graph as a list of edges we make no assumptions about their ordering. In particular, this ordering is not required to reflect the construction process of the series-parallel graph in any way. Otherwise, most questions for series-parallel digraph (e.g. recognition and reachability) could be solved very easily. In this section we will address the problem to compute such an ordering. We will construct a log-space algorithm that given a series-parallel graph G outputs a special ordering called **SP-ordering**. The crucial property of this ordering is that for any series-parallel component C with source v all direct successors of v in C are enumerated with consecutive integers. Speaking formally, for a node v the sequence **SP-succ(v)** is a permutation of $\text{succ}(v)$ such that the following condition holds:

$$\forall\, u \in \text{succ}^+(v) \qquad \{\, i \mid \texttt{SP-succ}(v,i) \in \text{pred}^*(u) \,\} \quad \text{is a subset of consecutive integers,} \qquad (1)$$

where for with $1 \leq i \leq |\text{succ}(v)|$ the value $\texttt{SP-succ}(v,i)$, denotes the i'th vertex in the SP-ordering of $\text{succ}(v)$. Recall that $\text{succ}(v,i)$ is the i'th direct successor of v according to the ordering implicitly given by the input ordering. Hence, in general $\texttt{SP-succ}(v,i)$ will be different from $\text{succ}(v,i)$. To compute the SP-ordering we will use $\text{succ}(v,i)$ and the following functions for a nonterminal node v:

$\texttt{FINAL}(v)$:= the nearest successor u of v such that any path from v to the sink of G contains u;

$\texttt{START}(v)$:= the nearest predecessor u of v such that any path from the source of G to v contains u.

Hence, $\texttt{FINAL}(v)$ is the sink of a smallest series-parallel component of G containing v and all direct successors of v. Similarly, $\texttt{START}(v)$ is the source of a smallest series-parallel component containing v and its predecessors $\text{pred}(v)$.

If $\mathrm{succ}(v)$ contains only a single node u then by definition $\mathtt{FINAL}(v) = u$. Otherwise, $\mathtt{FINAL}(v)$ can be computed by finding the nearest common successor of the left-most paths from any direct successors of v to the sink of G. $\mathtt{START}(v)$ can be computed in an analog way. Hence, both functions $\mathtt{FINAL}(v)$ and $\mathtt{START}(v)$ can be computed in logarithmic space.

Let us now introduce an important notion, which arises from our analysis of series-parallel graphs in the previous section. Let $v_1 \neq v_2$ be two arbitrary nodes of G. Then we define the set of **bridge nodes** between v_1 and v_2 as follows: $\mathtt{BRIDGES}(\boldsymbol{v_1, v_2})$ is the unique subset of $\mathrm{succ}^+(v_1) \cap \mathrm{pred}^+(v_2)$ such that

(1) $v_1 = \mathtt{START}(v)$ for every $v \in \mathrm{bridges}(v_1, v_2)$,
(2) $\forall\, v \in \mathrm{succ}^+(v_1) \cap \mathrm{pred}(v_2)$: every path from v_1 to v goes through a node of $\mathtt{BRIDGES}(v_1, v_2)$, and
(3) $\forall\, v_3 \neq v_4 \in \mathtt{BRIDGES}(v_1, v_2)$: v_2 ist the nearest common successor of v_3 and v_4.

From the definition it is not at all clear that $\mathtt{BRIDGES}(v_1, v_2)$ is well defined, whether these conditions can be fulfilled simultaneously and characterize this set of nodes uniquely. For general graphs, this it is indeed not the case. For series-parallel graphs, however, using the characterization by forbidden graphs of Theorem 2 it can be shown that this set is well defined.

Using the functions $\mathrm{succ}(v, i)$, $\mathtt{FINAL}(v)$ and $\mathtt{START}(v)$, the set $\mathtt{BRIDGES}(v_1, v_2)$ can be computed deterministically in logarithmic space. Furthermore, given a node $v_3 \in \mathtt{BRIDGES}(v_1, v_2)$ and v_1, the node v_2 can be determined in logarithmic space, as well. Let $\mathtt{BRIDGES}(v_1, v_2, i)$ be the i-th bridge node between v_1 and v_2 (we assume that this procedure outputs every element of the resulting set exactly once). Using $\mathrm{succ}(v, j)$ the values $\mathtt{BRIDGES}(v_1, v_2, i)$ can be computed in log-space.

We will now describe a recursive procedure that outputs the sequence of direct successors of a node v in SP-ordering. This procedure will initially be called with v and $u := \mathtt{FINAL}(v)$.

procedure SP-sequence(v, u)
  if $u \in \mathrm{succ}(v)$ **then output** u
  for $i = 1$ **to** $|\,\mathit{bridges}(v, u)|$ **do** SP-sequence$(v, \mathtt{BRIDGES}(v, u, i))$ **od**

It is easy to verify that this procedure computes a correct SP-ordering. On the other hand, using a log-space algorithm to compute $\mathtt{BRIDGES}(v, u, i)$ one can implement this procedure with logarithmic space as well. Furthermore, $\mathtt{SP\text{-}succ}(v, i)$ can be computed by counting the nodes in the output of the procedure SP-sequence$(v, \mathtt{FINAL}(v))$.

5 The Decomposition in Log-space

The *decomposition tree* of a series-parallel graph provides information how this graph has been built using the parallel and serial constructors.

Definition 2 *A binary tree $T = (V_T, E_T)$ with a labeling function $\sigma : V_T \to \{\mathtt{p}, \mathtt{s}\} \cup E$ is called a* **decomposition tree** *of a series-parallel graph $G = (V, E)$ iff*

1. *leaves of T are labelled with elements of E, internal nodes with* p *or* s*;*
2. *G can be generated recursively using T as follows:*
   (a) *if $V_T = \emptyset$ then G consists of a single node;*
   (b) *if T is a single node v then G consists of the single edge $\sigma(v)$ and its associated nodes;*
   (c) *let T_1 (resp. T_2) the right (resp. left) subtree of T and G_i be series-parallel graphs with decomposition tree T_i: if $\sigma(v) = \mathtt{p}$ (resp. $\mathtt{s}$) then G is the parallel (resp. serial) composition of G_1 and G_2.*

Our algorithm to generate a decomposition tree is based on the functions described in the previous section: $\mathtt{FINAL}(v)$, $\mathtt{START}(v)$, and $\mathtt{BRIDGES}(u, v, i)$. The procedure SP-DECOMPOSE(s, t) outputs the root of a decomposition tree of G, where s, t are the source (resp. sink) of G. The root will be called a. In the program below the semantic of an assignment $a := b$ is that node b now takes the role of the root. As an example for a decomposition tree generated by SP-DECOMPOSE(v_1, v_2) see Figure 3.

```
procedure SP-DECOMPOSE(u,v)
 1  if |BRIDGES(u,v)| ≥ 1 then
 2      get_node(a);  σ(a) :=s
 3      left(a) := SP-DECOMPOSE(u,BRIDGES(u,v,1))
 4      right(a) := SP-DECOMPOSE(BRIDGES(u,v,1),v)
 5      for i := 2 to |BRIDGES(u,v)| do
 6          get_node(b); get_node(c);  σ(b) :=p;  left(b) := a;  right(b) := c;  σ(c) :=s;
 7          left(c) := SP-DECOMPOSE(u,BRIDGES(u,v,i))
 8          right(c) := SP-DECOMPOSE(BRIDGES(u,v,i),v)
 9          a := b  od
10  if (u,v) ∈ E then
11      get_node(c);  σ(c) := (u,v)
12      if |BRIDGES(u,v)| = 0 then a := c
13      else  get_node(b);  σ(b) :=p;  left(b) := a;  right(b) := c;  a := b
14  return a.
```

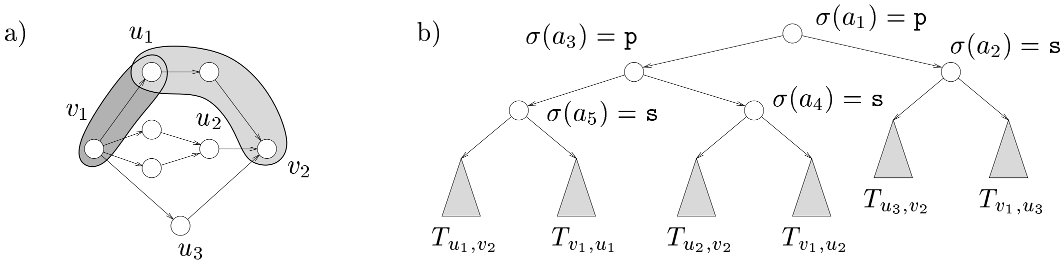

Figure 3: An example of a decomposition tree generated by SP-DECOMPOSE(v_1, v_2) where $T_{v,u}$ are the subtrees generated by SP-DECOMPOSE(v, u).

Theorem 5 *The decomposition tree of a series-parallel graph can be computed by a deterministic Turing machine in logarithmic space.*

Proof: The correctness of the procedure above can be proven inductively. To achieve space efficiency a procedure BACK-DECOMPOSE(x, y) will be used that for two nodes x, y returns the pair u, v and an integer i with the following properties: if SP-DECOMPOSE is called recursively for nodes (x, y) then u, v and i are the current values for this recursive call in lines 3, 4, 7, or 8. It is easy to see that it requires only logarithmic space, thus one can implement SP-DECOMPOSE without any stack for the recursion. ∎

```
procedure BACK-DECOMPOSE(x,y)
1  u := source(G);  v := sink(G);
2  i := 1
3  if (x = u ∧ y = BRIDGES(u,v,i)) or (x = BRIDGES(u,v,i) ∧ y = v) then return (u,v,i)
4  if x,y ∈ succ*(u) ∩ pred*(BRIDGES(u,v,i)) then v := BRIDGES(u,v,i);   goto 2
5  if x,y ∈ succ*(BRIDGES(u,v,i)) ∩ pred*(v) then u := BRIDGES(u,v,i);   goto 2
6  i := i + 1;   goto 3.
```

6 Path Counting Problems

In this section we show that for series-parallel graphs the classical problem to count the number of paths can be solved in $\mathcal{FL}$. For general graphs counting the number of paths is not solvable in $\mathcal{FL}$ – unless certain hierarchies collapse – since this problem is one of the generic complete problems for the class $\#\mathcal{L}$ [2]. Speaking more precisely, let us define the functional problem #PATH as follows: given a graph G and nodes a, b estimate the number of different paths from a to b in G.

Theorem 6 *Restricted to series-parallel graphs* #PATH *can be computed in* $\mathcal{FL}$.

Proof: Consider the subgraph G_{ab} of G induced by $V' := \text{succ}^*(a) \cap \text{pred}^*(b)$. It is either empty (and then $\#\texttt{PATH}(a,b) = 0$), or it is a series-parallel graph with source a and sink b. This follows from the fact that the predicate PATH restricted to nodes in V' is identical on G and G_{ab}. Furthermore, all paths from a to b in G occur in G_{ab} as well, thus the number of paths is identical. A simple induction shows that $\#\texttt{PATH}(a,b)$ can be bounded by 2^{n+m}. Using the reachability algorithm presented in Section 3 we can also decide in log-space whether an arbitrary edge of G belongs to G_{ab}.

Let $T = (V_T, E_T)$ be the decomposition tree of G_{ab} and $z \leq n+m$ be its size. We interpret the tree as an arithmetic expression as follows. Every leaf represents the integer 1. An internal node v of T labeled by $\sigma(v) = \texttt{s}$ (resp. $\texttt{p}$) corresponds to a multiplication (resp. addition) of the expressions given by the sons of v. It is easy to see that the value ρ of the root of T equals $\#\texttt{PATH}(a,b)$.

Below we sketch how ρ can be computed in logarithmic space. Let $p_1 < p_2 < \ldots$ be the standard enumeration of primes. The prime number theorem implies

$$\prod_{p_i \leq n+m} p_i \;=\; e^{(n+m)(1+o(1))} \;>\; \#\texttt{PATH}(a,b)\ . \tag{2}$$

Using the log-space algorithm of [7] one can tranform T into a binary tree T' of depth $O(\log z)$ representing an arithmetic expression with the same value ρ as T.

We evaluate $T' \bmod p_i$ using the algorithm in [4]. For $p_i \leq n+m$ this algorithm works in space $O(\log z + \log(n+m)) \leq O(\log n)$. By inequality (2), taking all $p_i \leq n+m$ the values $\rho \mod p_i$ give a *Chinese Remainder Representation* of ρ. Using the recent result of Chiu, Davida, and Litow [8] that such a representation can be converted to the ordinary binary representation in log-space, finishes the proof. ∎

On the other hand, we can show that counting the number of path is hard for $\mathcal{L}$ by using a reduction similarly to the one in Section 2.

Theorem 7 *Restricted to series-parallel graphs counting* $\#\texttt{PATH} \bmod 2$ *is* $\mathcal{L}$*-complete.*

Using the techniques presented so fare one can also solve some other counting problems space efficiently. For example, the size of G_{ab} can also be determined in $\mathcal{FL}$.

7 Generalisations of Series-Parallel Graphs

First we will consider graphs with several sources, but still with a unique sink.

Definition 3 *The family of multiple source series-parallel graphs, MSSP-graphs for short, are an extension of series-parallel graphs, adding the following constructor:*

(C) **In-Tree composition**: *a graph $G = (V,E)$ is generated from MSSP-graphs $G_i = (V_i, E_i)$ for $i = 1,2$ by selecting a node $\hat{v}$ in G_1, identifying it with the sink $v_{out,2}$ of G_2 and forming the union of both graphs: $V := V_1 \cup V_2$ and $E := E_1 \cup E_2$.*

An in-tree composition may be applied several times, but only at the end. As soon as a graph has several sources the series and the parallel constructor can no longer be used. *Multiple sink series-parallel graphs* with a unique source can be defined in the dual way. In the following, we will restrict ourselves to the first extension – the main results hold for both classes. Unlike ordinary series-parallel graphs, the reachability problem for MSSP-graphs cannot be solved by the following leftmost path. $\texttt{PATH}(x,y)$ not longer implies $\text{lm-down}(x) \cap \text{lm-up}(y) \neq \emptyset$ as illustrated in Figure 4a.

To solve this problem we have to use a more sophisticated strategy. We need a bunch of new definitions:

$$\text{elude}(v) \;:=\; \{\ u \mid \exists\ w_1, w_2 \in \text{succ}(u)\ :\ v \in \text{lm-down}(w_1)\ \setminus\ \text{lm-down}(w_2)\ \}\ ,$$

$$\text{min}-\text{elude}(v) \;:=\; \{u \in \text{elude}(v) \mid \forall\ w \in \text{elude}(v) \setminus \{u\} \quad u \in \bigcup_{x \in \text{succ}(w)} \text{lm-down}(x)\}\ ,$$

$$\text{if elude}(v) \neq \emptyset,$$

$$\{v\} \qquad \text{else},$$

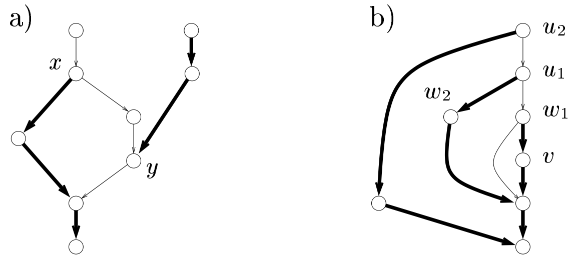

Figure 4: a) left-most paths that do not meet, b) computing elude(v)

$$\begin{aligned}
\min\text{-}\mathrm{elude}(v,i) &:= \begin{cases} v & \text{if } i = 0, \\ \min\text{-}\mathrm{elude}(\min\text{-}\mathrm{elude}(v, i-1)) & \text{if } i \geq 1, \end{cases} \\
\min\text{-}\mathrm{elude}^*(v) &:= \bigcup_i \min\text{-}\mathrm{elude}(v,i) \ .
\end{aligned}$$

These notions may be less intuitive. An illustration of this sets is is given in Figure 4b: $\mathrm{elude}(v) = \{w_1, u_1\}$, $\min\text{-}\mathrm{elude}(v) = w_1$, and $\min\text{-}\mathrm{elude}^*(v) = \{w_1, u_1, u_2\}$. One can show:

Lemma 5 *The function* min−elude *is well defined, that means if* $\mathrm{elude}(v) \neq \emptyset$ *then there is always a unique* $u \in \mathrm{elude}(v)$ *fulfilling the conditions in the definition of* min−elude. *It is deterministically computable in log-space.*

Analogously to lm-pred* we define

$$\mathrm{elude}\text{-}\mathrm{pred}^*(v) \ := \ \{ \ u \mid \exists w \in \min\text{-}\mathrm{elude}^*(v) \ : \ w \in \text{lm-down}(u) \ \} \ .$$

Similarly to Lemma 4 it can be shown:

Lemma 6 *Let G be a MSSP-graph, then* $\mathrm{pred}^*(v) = \mathrm{elude}\text{-}\mathrm{pred}^*(v)$ *for all v. Furthermore, the question whether* $\mathrm{pred}^*(v) = \mathrm{elude}\text{-}\mathrm{pred}^*(v)$ *can be decided in $\mathcal{L}$.*

Theorem 8 *The reachability problem for multiple source single sink series-parallel graphs can be decided in $\mathcal{L}$.*

To verify whether a given garph is a MSSP we can make use of the forbidden subgraph H again. However, a set of nodes fulfilling its `PATH` conditions can occur in MSSP-graph G, but only if z_3 belongs to another component G_2 of G than z_1 and z_2, which is connected to the rest graph of G via an in-tree composition step at z_4. Since this can also be verified in log-space we obtain:

Theorem 9 *The question whether a graph is MSSP can be decided in $\mathcal{L}$.*

The notion of decomposition trees can be extended to this graph family: first generate nodes that represent the in-tree composition steps, then the subtrees that describes the decomposition of the basic series-parallel graphs.

Theorem 10 *A decomposition tree of a MSSP-graph can be computed in $\mathcal{FL}$.*

Let us finally investigate another generalisation of series-parallel graphs: the minimal vertex-series-parallel graphs.

Definition 4 *A graph is a* **minimal vertex-series-parallel graph** *(MVSP for short) if it can be constructed using the following operations:*

Initial Step: *a single node is a MVSP-graph*

Parallel Composition *Let* $G_i = (V_i, E_i)$, $i = 1, 2$ *be two node disjunct MVSP-graphs. Then* $G = (V_1 \cup V_2, E_1 \cup E_2)$ *is a MVSP as well.*

Series Composition *Let* G_i *be as above and* $\text{sinks}(G_i)$ *(resp.* $\text{sources}(G_i)$*) denote the sinks (resp. sources) of* G_i. *Then* $G = (V_1 \cup V_2,\ E_1 \ \cup\ E_2 \ \cup\ \text{sinks}(G_1) \times \text{sources}(G_2)$ *is a MVSP-graph.*

For this family of graphs we use the following modification of the left-most path concept: $\text{lm-pred}_1^*(v) := \{\ u \mid (\text{lm-up}(v) \cup \text{pred}(\text{lm-up}(v))) \cap \text{lm-down}(v) \neq \emptyset\ \}$. We can show:

Lemma 7 *For every node* v *of a MVSP-graphs it holds* $\text{pred}^*(v) = \text{lm-pred}_1^*(v)$. *Furthermore, the question whether* $\text{pred}^*(v) = \text{lm-pred}_1^*(v)$ *can be decided in* $\mathcal{L}$.

This implies

Theorem 11 *The reachability problem for MVSP-graphs can be decided in* $\mathcal{L}$.

It has been shown that the *line graphs* of MVSP-graphs are closely related to series-parallel graphs (see [15] for the definition of line graphs and further details). Using an slight modification of the notion of line graphs we can also recognize MVSP-graphs space efficiently.

Theorem 12 *The decision problem for MVSP-graphs is in* $\mathcal{L}$.

Decomposition trees for MVSP-graphs can be defined in the same way as for series-parallel graphs. Again, using the decomposition tree of the modified line graph of a MVSP-graph we can generate the decomposition tree of the MVSP-graph. It follows:

Theorem 13 *Given a MVSP-graph* G, *the decomposition tree of* G *can be computed in deterministic log-space.*

Finally, we remark that the counting algorihm for series-parallel graphs can be extended to these classes as well.

Theorem 14 *For MSSP- and MVSP-graphs the function* #`PATH` *can be computed in* $\mathcal{FL}$. *The same holds for the size of subgraphs of the form* G_{ab}.

8 Conclusions and Open Problems

A deterministic Turing machine working in space $S \geq \log$ can be simulated by an EREW PRAM in time $O(S)$ (see e.g. [14]). The machine may use an exponential number of processors with respect to S. Therefore, we get immediately that the graph problems investigated in this paper can be solved in logarithmic parallel time. The simulation of space-bounded Turing machines by PRAMs can even be performed by the EROW model (exclusive-read owner-write). Hence we can deduce

Corollary 2 *For series-parallel graphs and their extensions considered above, recognition, reachability, decomposition, and path counting can be done in logarithmic time on EROW PRAMs with a polynomial number of processors.*

The exact number of processors depends on the time complexity of the Turing machine. Since our basic log-space algorithms require time $O(n^c)$ for some constant c significantly larger than 1, we probably will not achieve a linear number of processors this way. For the reachability problem in series-parallel graphs it is known that an EREW PRAM can solve it in logarithmic time using $n/\log n$ processors [15]. But it is still open whether also recognition and decomposition can be done in logarithmic time using at most a linear number of processors.

If we switch to *undirected* graphs the problems considered here seem to be inherently more difficult. In the undirected case series-parallel graphs can be characterized as the set of graphs containing no clique of size 4 as a minor [10]. To illustrate the difference between a directed and an undirected series-parallel graph consider the example given in Figure 5. The first graph a) shows a series-parallel directed graph, whereas b) is not series-parallel. The third graph c) illustrate the undirected graph corresponding to b). Note that c) is series-parallel, while d) is a 4-clique and thus not series-parallel.

In contrast to the series-parallel graph family, the reachability problem for arbitrary graphs seems to be easier in the undirected case than in the directed case. From [1] we know that the undirected version can be solved by a randomized log-space bounded machine, whereas no randomized algorithm is known for the directed case. Are there other distinctions of this kind?

Acknowledgment: Thanks are due to Eric Allender and Markus Bläser for helpful comments and pointers to the literature. Furthermore we would thank Faith Fich, Pierre McKenzie, and Christian Schindelhauer for fruitful discussions.

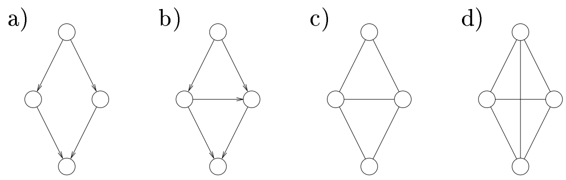

Figure 5: comparing directed and undirected series-parallel graphs

References

[1] R. Aleliunas, R. Karp, R. Lipton, L. Lovasz, C. Rackoff, *Random Walks, Universal Sequences and the Complexity of Maze Problems,* Proc. 20. FOCS, 1979, 218-223.

[2] C. Álvarez, B. Jenner, *A Very Hard Log-space Counting Classes,* Theo. Comp. Science 107, 1993, 3-30.

[3] E. Allender, M. Mahajan *The Complexity of Planarity Testing,* Proc. 17. STACS, LNCS 1770, 2000, 87-98.

[4] M. Ben-Or, R. Cleve *Computing Algebraic Formulas Using a Constant Number of Registers,* SIAM J. Comput. 21, 1992, 54-58.

[5] H. Bodleander, B. de Fluiter, *Parallel Algorithms for Series Parallel Graphs,* Proc. 4. ESA, LNCS 1136, 1996, 277-289.

[6] A. Brandstädt, V. Bang Le, J. Spinrad, *Graph Classes: A Survey,* SIAM 1999.

[7] S. Buss, S. Cook, A. Gupta, V. Ramachandran, *An Optimal Parallel Algorithm for Formula Evaluation,* SIAM J. Comput. 21, 1992, 755-780.

[8] A. Chiu, G. Davida, B. Litow, *NC^1 Division,* unpublished manuscript, November 1999.

[9] S. Cook, P. McKenzie, *Problems Complete for Deterministic Logarithmic Space*, J. Algo. 8, 1987, 385-394.

[10] R. Duffin, *Topology of Series-Parallel Networks*, J. Math. Analysis Appl. 10, 1965, 303-318.

[11] D. Eppstein, *Parallel Recognition of Series-Parallel Graphs*, Inf. & Comp. 98, 1992, 41-55.

[12] X. He, Y. Yesha, *Parallel Recognition and Decomposition of Two Terminal Series Parallel Graphs*, Inf. & Comp. 75, 1987, 15-38.

[13] B. Jenner, K.-J. Lange, P. McKenzie, *Tree Isomorphism and Some Other Complete Problems for Deterministic Logspace*, publication #1059, DIRO, Université de Montréal, 1997.

[14] R. Karp, V. Ramachandran, *Parallel Algorithms for Shared-Memory Machines*, in: J. van Leeuwen (Ed.): Handbook of Theoretical Computer Science, Volume A, 1990, 869-941.

[15] J. Valdes, R. Tarjan, E. Lawlers *The Recognition of Series Parallel Digraphs,* SIAM J. Comput. 11, 1982, 298-313.