An unambiguous class possessing a complete set

Klaus-Jörn Lange

Wilhelm-Schickard-Institut, Tübingen

Topics : Computational and structural complexity theory

Abstract

In this work a complete problem for an unambiguous logspace class is presented. This is surprising since unambiguity is a 'promise' or 'semantic' concept. These usually lead to classes apparently without complete problems.

1 Introduction

One of the most central questions of complexity theory is to compare determinism with nondeterminism. Our inability to exhibit the precise relationship between these two features motivates the investigation of intermediate features such as symmetry or unambiguity. In this paper we will concentrate on the notion of unambiguity.

Unfortunately, unambiguity of a device or of a language is in general an undecidable property. Unambiguous classes are not defined by a 'syntactical' machine property but rather by a 'semantical' restriction. A nasty consequence is the apparent lack of complete sets. In the case of time bounded computations there are relativizations of unambiguity which provably have no complete problem ([10]).

For space bounded computations the concept of unambiguity is not as uniform as in the time bounded case. There are several versions of it, which probably are different for space classes, while they coincide for time bounded computations. This is the case, since it is possible to keep track of the complete history of a computation without increasing the running time. It is remarkable, that this is precisely the same construction which shows the equivalence of nondeterminism with symmetry and of determinism with reversability in the time bounded case ([14,3]).

The main result of this work is to show that the unambiguous logspace class $RUSPACE(\log n)$ possesses a complete problem. The proof makes intensive use of the space-specific possibility to cycle through all configurations of a machine without increasing the resource bound. The proof doesn't seem to work for other versions of unambiguity or for other related semantic classes but nevertheless leaves the hope to exhibit complete problems for other semantic logspace classes defined by concepts like unambiguity and randomization.

As an interesting consequence of this result we get the first case of a single and explicit problem which can be solved by an unambiguous logspace algorithm but which is not known to be solvable in logarithmic space deterministically.

It was known before that the family *ULIN* of unambiguous linear context-free languages is contained in *USPACE*$(\log n)$ and it is still open whether *ULIN* $\subseteq$ *DSPACE*$(\log n)$ (see [4]). But there was no specific candidate known within *ULIN*, which was not known to be in *DSPACE*$(\log n)$.

2 Preliminaries

The reader is assumed to be familiar with the basic notions of complexity theory as they are contained in the standard text books on theoretical computer science.

2.1 Turing machines and configuration graphs

For a Turing machine T with input alphabet X we denote the graph of configurations over an input $w \in X^*$ by $G_T(w)$. If T is logarithmically space-bounded $G_T(w)$ has a size polynomial in the length of w. Without loss of generality we assume all machines considered here to be non-looping. Hence all configuration graphs are acyclic. In order to ease the presentation of our constructs we assume that in nondeterministic machines each non nonterminating configuration has exactly two successor configurations, which can be reached in one step. It should be remarked that this normal form can be reached without changing the number of accepting computations, i.e.: without changing properties like umambiguity.

Let $G = (V, E)$ be a directed acyclic graph. For two nodes x and y we denote by $N(x, y)$ the number of different paths leading from x to y in G. For $x = y$ we set $N(x, y) := 1$. For each pair of nodes (x, y) let $d(x, y)$ be the length of the shortest path between x and y. The length of a path is the number of its edges. If x and y are not connected, $d(x, y)$ is infinite. $d(x, x)$ is 0 for each $x \in V$. In the following we will work with complete binary graphs, that is, each node of G is either a leaf with no outgoing edges, or an inner node with two outgoing edges. This is determined by a mapping $\phi : V \longrightarrow \{i, l\}$, which takes the value l for leaves and i for inner nodes. The two successors of an inner node x will be denoted by $L(x)$ and $R(x)$. In the following we assume all graphs like $G_T(w)$ to be given in this form as (V, ϕ, L, R). Thus the edge set E would be $\{(x, L(x))|\phi(x) = i\} \cup \{(x, R(x))|\phi(x) = i\}$. Since G is acyclic, it contains no self-loop, i.e.: $L(x) \neq x \neq R(x)$ for $x \in V$. In addition, we assume without loss of generality that G contains no double edges (i.e.: $L(x) \neq R(x)$ for all x). If G has n nodes we assume V to be $\{v_1, v_2, \cdots, v_n\}$. We will be interested in the existence of paths between nodes v_1 and v_n. We will assume v_1 to be an inner node and v_n to be a leaf. Hence $n > 1$.

We call G *accepting* iff $N(v_1, v_n) > 0$, i.e.: if there is a path from v_1 to v_n. Otherwise we call G *rejecting*.

For $x \in V$ let $T(x) := \{y \in V|N(x, y) \geq 1\}$ be the set of all nodes reachable from x. For $d \geq 0$ we set $T_d(x) := \{y \in T(x)|d(x, y) \leq d\}$. Obviously, we have $T(x) = T_{n-1}(x)$ for every $x \in V$. Throughout the paper we will identify $T(x)$ and $T_d(x)$ with the subgraphs of G induced by $T(x)$ and by $T_d(x)$.

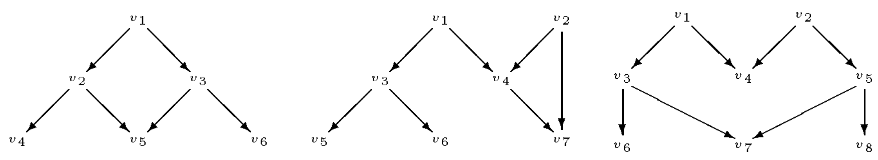
3 Unambiguity

A concept intermediate in power between determinism and nondeterminism is *Unambiguity.* A nondeterministic machine is said to be unambiguous, if for every input there exists at most one accepting computation. This leads for instance to the classes *UP* and *USPACE*$(\log n)$; we have $P \subseteq UP \subseteq NP$ and *DSPACE*$(\log n) \subseteq$ *USPACE*$(\log n) \subseteq$ *NSPACE*$(\log n)$ where none of these inclusions is known to be strict. The notion of unambiguity should be distinguished from that of *Uniqueness*, which uses the unique existence of an accepting path not as a restriction but as a tool. The resulting language classes 1*NSPACE*$(\log n)$ and 1*NP* consists of languages defined by machines that accept their inputs if there is exactly one accepting path. Thus, the existence of two or more accepting computations is not forbidden, but simply leads to rejection. In the polynomial time case we have *Co–NP* $\subseteq$ 1*NP*. In the logspace case inductive counting ([11,19]) shows 1*NSPACE*$(\log n) =$ *NSPACE*$(\log n)$.

3.1 Space bounded unambiguous classes

The concept of unambiguity of space bounded computations is not as uniform as that for time bounded classes. Instead we are confronted with a variety of probably different concepts of unambiguity. In the following we classify three notions of unambiguity for configuration graphs and complexity classes. For more varieties of unambiguities see [4].

A configuration graph $G = (V, \phi, L, R)$ is called *unambiguous* if there is at most one path from v_1 to v_n, i.e.: if $N(v_1, v_n) \leq 1$. G is called *reach-unambiguous* if for any x there is at most one path from v_1 to x, i.e.: if for each $x \in V\, N(v_1, x) \leq 1$. G is called *strongly unambiguous* or a *mangrove* if for any pair (x, y) of nodes there is at most one path leading from x to y, i.e.: if $\forall_{x,y \in V}\, N(x, y) \leq 1$. Every mangrove is reach-unambiguous and every reach-unambiguous graph is unambiguous. Although a mangrove does not need to be a tree, for each x the subgraph $T(x)$ is indeed a tree and the same is true for the set of all nodes from which x can be reached. Some examples:



The left configuration graph is accepting and unambiguous since there is exactly one path from v_1 to v_6. It is not reach unambiguous since there exist two different paths between nodes v_1 and v_5. The second one is reach-unambiguous and accepting, but not a mangrove since there are two different paths between nodes v_2 and v_7. The third example is a mangrove, which is rejecting since there is no path from v_1 to v_8.

Let T be a nondeterministic Turing machine with input alphabet X. T is called *unambiguous*, if for all inputs $w \in X^*$ the configuration graph $G_T(w)$

is unambiguous. T is *reach-unambiguous*, if for all $w \in X^*$ the configuration graph $G_T(w)$ is reach-unambiguous. Finally, T is *strongly unambiguous*, if for all $w \in X^*$ the configuration graph $G_T(w)$ is a mangrove. Of course, both for the language and Turing machines each of this properties is undecidable. These concepts lead to the following classes:

Definition 1. a) $USPACE(\log n)$ is the class of all languages accepted by unambiguous logspace machines.
b) $RUSPACE(\log n)$ is the class of all languages accepted by reach-unambiguous logspace machines.
c) $StUSPACE(\log n)$ is the class of all languages accepted by strongly unambiguous logspace machines.

We mention in passing that in the case of time bounded classes these three concepts coincide.

By definition, these three classes fulfill: $StUSPACE(\log n) \subseteq RUSPACE(\log n) \subseteq USPACE(\log n) \subseteq NSPACE(\log n)$. In addition, by [4] we know

Proposition 2. *i)* $RUSPACE(\log n) \subseteq LOG(DCFL)$
ii)Both $RUSPACE(\log n)$ *and* $StUSPACE(\log n)$ *are closed under complement.*

Here $LOG(DCFL)$ denotes the class of all languages reducible to deterministic context-free languages by deterministic many-one reductions. As a consequence of part i) both $RUSPACE(\log n)$ and $StUSPACE(\log n)$ are contained in $SC^2 := DTIMESPACE(pol, \log^2 n)$, the second level of the SC hierarchy ([6,7]).

The inclusion $StUSPACE(\log n) \subseteq DSPACE(\log^2 n / \log\log n)$ was shown in [2]. But as remarked there, the proof uses only the fact, that the unfolding of the reachability graph, i.e.: the number of all paths starting in the root, is of polynomial size. While this is not true for $USPACE(\log n)$, this property is fulfilled by reach-unambiguous graphs. Hence we have:

Proposition 3. $RUSPACE(\log n) \subseteq DSPACE(\log^2 n / \log\log n)$

It should be remarked, that nothing like Proposition 2 or 3 is known for $USPACE(\log n)$.

3.2 Sets of unambiguous reachability problems

The unambiguous classes defined in the previous subsection are semantic or promise classes. They are defined via machines which are subject to an undecidable restriction. As a consequence, they probably don't posses complete sets. There are relativizations in the time bounded case excluding the existence of complete sets ([10]).

For logarithmically space bounded classes the typical complete problems are reachability or connectivity problems in graphs. The question for existence of a connecting path in directed graphs is complete for $NSPACE(\log n)$, that in undirected graphs for $SymSPACE(\log n)$, and that in forests for $DSPACE(\log n)$.

In connection with the three unambiguous classes the corresponding connectivity problems would be:

$$L_u := \{G = (V, \phi, L, R) \,|N(v_1, v_n) = 1\},$$

$$L_{ru} := \{G = (V, \phi, L, R) \,|N(v_1, v_n) = 1 \;, \forall_{x \in V} \; N(v_1, x) \leq 1\},$$

$$L_{su} := \{G = (V, \phi, L, R) \,|N(v_1, v_n) = 1 \;, \forall_{x,y \in V} \; N(x, y) \leq 1\},$$

Obviously, these sets are hard for the corresponding complexity classes:

Proposition 4. *i)* L_u *is* $USPACE(\log n)$*–hard, ii)* L_{ru} *is* $RUSPACE(\log n)$*–hard, and iii)* L_{su} *is* $StUSPACE(\log n)$*–hard.*

But to show the completeness of these languages we have to exhibit unambiguous logspace algorithms. But while the uniqueness of a computational path is used as a restriction in the definition of the complexity classes, this uniqueness is used as a tool (i.e.: as an acceptance criterion) in the definition of the connectivity problems. In fact, it turns out, that L_u seems to be harder than $USPACE(\log n)$, since it is complete for $1NSPACE(\log n) = NSPACE(\log n)$, as mentioned above:

Proposition 5. L_u *is* $NSPACE(\log n)$*–complete.*

Proof: Obviously, $L_u \in NSPACE(\log n)$ by the closure of $NSPACE(\log n)$ under complement. On the other hand, the usual $NSPACE(\log n)$–complete GAP problem asking for the existence of a path from v_1 to v_n in an unrestricted graph is reduced to the complement of L_u by adding the edge (v_1, v_n). $\square$

This seems to indicate that $L_u$is probably not a member of $USPACE(\log n)$. It is tempting in this situation to increase this appearence by trying to derive structural upper bounds as they were obtained for $StUSPACE(\log n)$ and $RUSPACE(\log n)$ ([4,2]). But by a result of Wigderson ([20]) this would immediately carry over to nonuniform $NSPACE(\log n)$.

4 Main Result

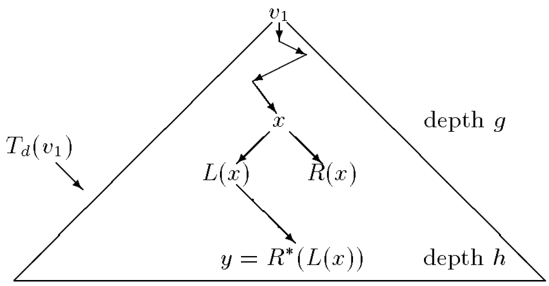
We will now show $L_{ru} \in RUSPACE(\log n)$. The idea of proof will be as follows: assume $T(v_1)$ to be a tree and to perform a breadth first search for violations of this assumption while inductively using the fact that the parts searched so far are indeed trees. If no violation exists $T(v_1)$ is a tree and hence the input graph is reach-unambiguous. The problem is to traverse the tree since logspace doesn't suffice to keep track of the whole path leading from v_1 to the currently visited node. To traverse $T(v_1)$ as a tree we use nondeterminism. If in fact $T(v_1)$ is a tree, all paths are unique and guessing paths turns out to be reach-unambiguous. If the input graph is not a tree we will find the smallest counterexample in an reach-unambiguous way.

We will use the procedure $NEXT(y, h, d)$, given in Table 1, which in case $T_d(v_1)$ is a tree computes the preorder successor of a node $y \in T_d(v_1)$ which

```
1  IF φ(y) = i AND h < d THEN
2        (y,h) := (L(y), h+1)
   ELSE
3        z := v1;
4        j := 1;
5        WHILE j < h AND φ(z) = i DO
6             z := R(z);
7             j := j + 1
         OD;
8        IF R(z) = y THEN
9             (y,h) := (v1, 0)
         ELSE
10            GUESSUNCLE(y,h,d)
```

Table 1. Procedure NEXT(y, h, d)

has depth h wrt v_1. Procedure NEXT first deterministically checks the easy cases that y is an inner node inside of $T_d(v_1)$ or that $y = R^h(v_1)$. If this is not successful NEXT uses the nondeterministic procedure GUESSUNCLE. It should be remarked, that in line 8 the condition $R(z) = y$ is not allways well-defined since not necessarily $\phi(z) = i$ holds. In this case we regard the condition as not fulfilled leading to a call of GUESSUNCLE. This procedure, given in Table 2 is invoked if on the path from v_1 to y an R-edge is used. Nondeterministically this path and in particular its last occurring R-edge are guessed. This situation is illustrated in Figure 1. Procedure GUESSUNCLE has two types of exits: either



On the unique path from v_1 to y, x is the least predecessor of y which is followed by an L-successor. GUESSUNCLE guesses x and the path from v_1 to x. The rest is checked deterministically. As a result y is replaced by $R(x)$.

Fig. 1. The work of procedure GUESSUNCLE

it successfully computes the successor of y in inorder traversal of $T_d(v_1)$ or it ends its computation in a state STOPi for $i = 1, 2, 3$, or 4. To reach one of the STOP states means that the previous nondeterministic computation was not able to find the node y. If $T(v_1)$ is not a tree, the behaviour of GUESSUNCLE is ambiguous. But if $T(v_1)$ is a tree, we can show that GUESSUNCLE works reach-unambiguously. Further on, if it avoids a STOP state, the computed successor of (y, h) is uniquely determined.

```
GUESS  g ∈ {0, 1, ···, h − 1};
z := v1;
FOR  j = 1, 2, ···, g DO
     GUESS  b ∈ {0, 1};
     IF  b = 0 THEN
          z := L(z)
     ELSE
          z := R(z);
     IF  φ(z) ≠ i THEN  STOP1(d, y, h, g, z, j)
OD;
x := z;
z := L(z);
IF  g + 1 = hTHEN
     IF  y ≠ z  THEN STOP2(d, y, h, x);
     (y, h) := (R(x), h)
ELSE
     FOR  j = g + 2, ···, h DO
          IF  φ(z) ≠ i THEN  STOP3(d, y, h, g, x, z, j);
          z := R(z)
     OD;
     IF  y ≠ z  THEN STOP4(d, y, h, g, x, z);
     (y, h) := (R(x), g + 1)
```

Table 2. Procedure GUESSUNCLE(y, h, d)

Proposition 6. *If for an input graph (V, ϕ, L, R) the subgraph $T_d(v_1)$ is a tree, then NEXT(y, h, d) works reach-unambiguously for each $0 \leq h \leq d$ and each $y \in T_d(v_1)$.*

Proof: NEXT is deterministic except for its subprocedure GUESSUNCLE, which is activated in the situation of Figure 1: There is a node x with $d(v_1, x)) = g$ for some $g < h$ such that $R^{h-g-1}(L(x)) = y$. Since $g < d$ and $T_d(v_1)$ is a tree, there is exactly one path leading from v_1 to x. First, this path is guessed nondeterministically. After that the condition $R^{h-g-1}(L(x)) = y$ can be checked deterministically.

During the nondeterministic process there are several possibilities to make wrong guesses which all end up in *STOP* states. STOP1 is reached if the current node is a leaf and thus has no outgoing edge. That is, while trying to guess a path of length g we got stuck after j steps. STOP2 indicates in the case $g = h-1$ that the guessed path didn't hit y. STOP3 is reached, if $R^{h-g-1}(L(x))$ does not exist. Finally, STOP4 means that we guessed a path of length h which doesn't lead to y.

In all these cases the actual values of the program variables uniquely determine the computational history which led to the STOP state, since $T_d(v_1)$ was assumed to be a tree. To make this more precise, with each STOP state the relevant variables are explicitly listed. Thus the computation is reach-unambiguous. $\square$

It should be remarked that we couldn't replace NEXT and GUESSUNCLE by something using the Immerman-Szelepcséyi procedure ([11,19]) since this inherently admits an exponential number of possible computation before reaching an ACCEPT, REJECT, or STOP state. Thus it could only be unambiguous but never reach-unambiguous. The advantage of NEXT and GUESSUNCLE is that they stop early enough such that the complete computational history is uniquely determined by the program variables and the input. In this way we are able to avoid a superpolynomial number of (rejecting) computations.

With the help of the previous proposition we are now able to prove our main result.

Theorem 7. $L_{ru} \in RUSPACE(\log n)$.

Proof: The logspace algorithm for checking that $T(v_1)$ is a tree for a given input graph $G = (V, \phi, L, R)$ is given in Table 3. Obviously, this algorithm uses

```
1  reached:= false;
2  FOR d = 1, 2, ···, n − 2 DO
3      (y, h) := (v1, 0);
4      just-begun := true;
5      WHILE (y, h) ≠ (v1, 0) OR just-begun = true DO
6          just-begun := false;
7          IF vn ∈ {L(y), R(y)} THEN reached:= true;
8          IF h = d AND φ(y) = i THEN
9              (y', h') := (v1, 0);
10             just-begun' := true;
11             WHILE (y', h') ≠ (v1, 0) OR just-begun' = true DO
12                 just-begun' := false;
13                 IF y' ∈ {L(y), R(y)} THEN REJECT1(d, y, y', h');
14                 IF h' = d AND y ≠ y' AND φ(y') = i THEN
15                     IF {L(y'), R(y')} ∩ {L(y), R(y)} ≠ ∅ THEN
16                         REJECT2(d, y, y');
17                 NEXT(y', h', d)
               OD;
18         NEXT(y, h, d)
       OD
   OD;
19 IF reached THEN ACCEPT ELSE REJECT3
```

Table 3. Program to check the tree property of $T(v_1)$

only $O(\log n)$ space. Further on, there exists an accepting computation if and only if $T(v_1)$ is a tree and $v_n \in T(v_1)$. That is, this a nondeterministic logspace algorithm recognizing L_{ru}. It is deterministic except for the calls of NEXT in lines 17 and 18. The cooperation of the main programm with its subroutines works in a way that the control is given back to the main program unless a STOP occurred. In that case the whole computation stops without acceptance.

If a REJECT is reached in the main program the computation is terminated as well, but with the knowledge that this input doesn't belong to L_{ru}.

We will now show that the algorithm sketched in the previous tables works reach-unambiguously. First, we may assume the graphs $T_0(v_1)$ and $T_1(v_1)$ to be trees, since by assumption our graphs contain neither double edges nor self-loops. (This could be tested deterministically in advance.)

The program searches in a breadth first way for ambiguities in the graphs $T_2(v_1), T_3(v_1), \cdots, T_{n-1}(v_1)$. It doesn't start to look over $T_{d+1}(v_1)$ unless $d \leq 1$ or $T_d(v_1)$ turned out to be a tree in the previous "stop-free" execution of the *FOR* loop in line 2. If this was the case, we traverse $T_d(v_1)$ with the help of NEXT in line 18. For every node $y \in T_d(v_1)$ on this tour with $d(v_1, y) = d$ and $\phi(y) = i$, that is for every leaf of $T_d(v_1)$ which is not a leaf in $T(v_1)$, $L(y)$ and $R(y)$ are compared with every $y' \in T_d(v_1)$. In case of any coincidence we end up in state REJECT1 which indicates the existance of two paths of different length from v_1 to y'. Otherwise, we compare $L(y)$ and $R(y)$ with $L(y')$ and $R(y')$ for every $y' \in T_d(v_1)$ with $d(v_1, y') = d, \phi(y') = i$, and $y \neq y'$. In case of any coincidence we end up in state REJECT2 which indicates the existance of two different paths of the same length leading from v_1 to either $L(y')$ or $R(y')$. If both REJECT1 and REJECT2 have been avoided, we know that $T_{d+1}(v_1)$ is a tree. After doing this for $d = 1, \cdots, n-2$ without reaching a STOP state we know that $T_{n-1}(v_1) = T(v_1)$ is a tree. Finally, we accept if a path from v_1 to v_n had been detected. Otherwise, we end up in state REJECT3.

Since procedure NEXT and hence GUESSUNCLE are only activated on subgraphs $T_d(v_1)$ which have been shown to be trees before, they work reach-unambiguously. Thus the whole program, being deterministic except for the call of NEXT, works reach-unambiguously, as well, since NEXT computes a single-valued function by either stopping or computing a successor value which is uniquely determined by the input. □

Corollary 8. *L_{ru} is $RUSPACE(\log n)$–complete.*

With [13] this implies that we can constructively enumerate the reach-unambiguous logspace languages:

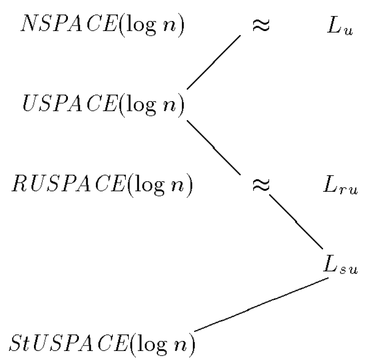
Corollary 9. *$RUSPACE(\log n)$ is recursively presentable*

By changing the main program to check the tree property of $T(x)$ not only for $x = v_1$ but for every $x \in V$ gives us a reach-unambiguous program to recognize mangroves:

Corollary 10. $L_{su} \in RUSPACE(\log n)$

We remark, that this program is not strongly unambiguous.

We summarize the relations between unambiguous classes and hardest languages in the Figure 2.



The three unambiguous classes seem to be weaker than the corresponding reachability problems. The $NSPACE(\log n)$-completeness of L_u indicates that we should not expect $USPACE(\log n)$-completeness. The more surprising is the $RUSPACE(\log n)$-completeness of L_{ru}

Fig. 2. Unambiguous logspace classes and reachability problems

5 Discussion and open questions

Considering the results of the previous section, it is natural to ask whether $StUSPACE(\log n)$ or even $RUSPACE(\log n)$ collapse down to $DSPACE(\log n)$. One fact speaking against equivalence, is that the unambiguous linear context-free languages are contained in $StUSPACE(\log n)$ but still are not known to be in $DSPACE(\log n)$.

Theorem 7 and Corollary 10 presented reach-unambiguous algorithms to recognize L_{ru} and L_{su}. None of the two works strongly unambiguous since the unreachable situation that GUESSUNCLE is activated on a nontree-like subgraph inherently results in an ambiguous behaviour. The open question here is, whether L_{su} could be $RUSPACE(\log n)$-complete, too.

The approach of Theorem 7 neither works for $USPACE(\log n)$ and L_u. But nevertheless we obtain with L_{ru} the first example of a single and explicit language which is known to be in $USPACE(\log n)$ and which is not known to be in $DSPACE(\log n)$. Compare this with the situation of the primality problem which is known to be in UP but not known to be in P ([9,15]). In view of the lack of complete problems for UP the primality problem and its relatives are the natural candidates for a problem in $UP \setminus P$. This role could now be played for $USPACE(\log n)$ vs. $DSPACE(\log n)$ by L_{ru}.

Finally, we would like to draw the attention of the reader to the structural similarity of reach-unambiguity and symmetry. Both $RUSPACE(\log n)$ and the symmetric logspace class $SymSPACE(\log n)$ possess complete problems and share nearly the same structural upper bounds, which seem to distinguish them from $NSPACE(\log n)$; they are contained in parity logspace, $DSPACE(o(\log^2 n))$, and SC^2 ([12,16,4,17,2]). Open questions here are: what is the relationship between $SymSPACE(\log n)$ and $RUSPACE(\log n)$. Can the inclusion of $SymSPACE(\log n)$ in randomized logspace ([1]) be extended to $RUSPACE(\log n)$? If so, the deterministic space bound of $O(\log^2 n / \log\log n)$ for $RUSPACE(\log n)$ could be improved to $O(\log^{1.5} n)$ ([18]). Can the inclusion $RUSPACE(\log n) \subseteq LOG(DCFL)$ be extended to $SymSPACE(\log n)$? If so, the currently best known

CREW-PRAM running time for $SymSPACE(\log n)$ of $O(\log n \cdot \log\log n)$ demonstrated in [5] would be improved to $O(\log n)$ and, in addition, the new algorithm would run on the simpler CROW-PRAM [8].

References

1. R. Aleliunas, R. Karp, R. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal sequences and the complexity of maze problems. In *Proc. of 20rd FOCS*, pages 218–223, 1979.
2. E. Allender and K.-J. Lange. StUSPACE(log n) in DSPACE($\log^2$ n/loglog n). In *Proc. of the 17th ISAAC*, 1996. In print.
3. C. Bennet. Time/Space trade-offs for reversible computation. *SIAM J. Comp.*, 18:766–776, 1989.
4. G. Buntrock, B. Jenner, K.-J. Lange, and P. Rossmanith. Unambiguity and fewness for logarithmic space. In *Proc. of the 8th Conference on Fundamentals of Computation Theory*, number 529 in LNCS, pages 168–179, 1991.
5. Ka Wong Chong and Tak Wah Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 11–20, 1993.
6. S. Cook. Deterministic CFLs are accepted simultaneously in polynomial time and log squared space. In *Proc. of the 11th Annual ACM Symp. on Theory of Computing*, pages 338–345, 1979.
7. S. Cook. A taxonomy of problems with fast parallel algorithms. *Inform. and Control*, 64:2–22, 1985.
8. P. Dymond and W. Ruzzo. Parallel RAMs with owned global memory and deterministic context-free language recoginition. In *Proc. of 13th International Colloquium on Automata, Languages and Programming*, number 226 in LNCS, pages 95–104. Springer, 1986.
9. M. Fellows and N. Koblitz. Self-witnessing polynomial-time complexity and prime factorization. In *Proc. of the 7th IEEE Structure in Complexity Conference*, pages 107–110, 1992.
10. J. Hartmanis and L. Hemachandra. Complexity classes without machines: on complete languages for UP. *Theoret. Comput. Sci.*, 58:129–142, 1988.
11. N. Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comp.*, 17:935–938, 1988.
12. M. Karchmer and A. Wigderson. On span programs. In *Proc. of the 8th IEEE Structure in Complexity Theory Conference*, pages 102–111, 1993.
13. W. Kowalczyk. Some connections between presentability of complexity classes and the power of formal systems of reasoning. In *Proc. of 10th Symposium on Mathematical Foundations of Computer Science*, number 201 in LNCS, pages 364–368. Springer, 1984.
14. P. Lewis and C.H. Papadimitriou. Symmetric space-bounded computation. *Theoret. Comput. Sci.*, 19:161–187, 1982.
15. G. Miller. Riemann's hypothesis and tests for primality. *J. Comp. System Sci.*, 13:300–317, 1976.
16. N. Nisan. $RL \subseteq SC$. In *Proc. of the 24th Annual ACM Symposium on Theory of Computing*, pages 619–623, 1992.
17. N. Nisan, E. Szemeredi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *Proc. of 33th Annual IEEE Symposium on Foundations of Computer Science*, pages 24–29, 1992.

18. M. Saks and S. Zhou. RSPACE(s) $\subseteq$ DSPACE($s^{3/2}$). In *Proc. of 36th FOCS*, pages 344–353, 1995.
19. R. Szelepcsényi. The method of forcing for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.
20. A. Wigderson. NL/poly $\subseteq \bigoplus$L/poly. In *Proc. of the 9th IEEE Structure in Complexity Conference*, pages 59–62, 1994.