Theoretical Computer Science 130 (1994) 203–236
Elsevier

Complexity models for incremental computation*

Peter Bro Miltersen**

Computer Science Department, DAIMI, Aarhus University, Ny Munkegade, Bldng. 540, DK-8000 Aarhus C, Denmark

Sairam Subramanian***

Department of Computer Science, Brown University, P.O. Box 1910, Providence, RI 02912-1910, USA

Jeffrey Scott Vitter***

Department of Computer Science, Duke University, P.O. Box 90129, Durham, NC 27708-0129, USA

Roberto Tamassia†

Department of Computer Science, Brown University, P.O. Box 1910, Providence, RI 02912-1910, USA

Abstract

Miltersen, P.B., S. Subramanian, J.S. Vitter and R. Tamassia, Complexity models for incremental computation, Theoretical Computer Science 130 (1994) 203–236.

We present a new complexity theoretic approach to incremental computation. We define complexity classes that capture the intuitive notion of incremental efficiency and study their relation to existing complexity classes. We show that problems that have small sequential space complexity also have small incremental time complexity.

Correspondence to: P.B. Miltersen, Computer Science Department, DAIMI, Aarhus University, Ny Munkegade, Bldng. 540, DK-8000 Aarhus C, Denmark.

* This paper reports the combined work of research efforts that have appeared in shortened form in "A complexity theoretic approach to incremental computation", by S. Sairam, J.S. Vitter, and R. Tamassia (STACS '93), and "On-line reevaluation of functions", by P.B. Miltersen (Tech. Report, Aarhus Univ.).

** Supported in part by the ESPARIT II Basic Research Actions Program of the European Community under contract No. 3075 (project ALCOM).

*** Supported in part by NSF PYI award CCR-9047466 with matching funds from IBM, by NSF research grant CCR-9007851, by Army Research Office grant DAAL03-91-G-0035 and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052, ARPA order 8225.

† Supported in part by an NSF research grant CCR-9007851, by Army Research Office grant DAAL03-91-G-0035 and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052, ARPA order 8225.

SSDI 0304-3975(93)E0214-O

We show that all common LOGSPACE-complete problems for P are also *incr*-POLYLOGTIME-complete for P. We introduce a restricted notion of completeness called NRP-completeness and show that problems which are NRP-complete for P are also *incr*-POLYLOGTIME-complete for P. We also give incrementally complete problems for NLOGSPACE, LOGSPACE, and nonuniform NC^1. We show that under certain restrictions problems which have efficient dynamic solutions also have efficient parallel solutions. We also consider a nonuniform model of incremental computation and show that in this model most problems have almost linear complexity. In addition, we present some techniques for lower bounding the complexity of explicitly defined problems.

We also look at the time complexity of circuit-value and network-stability problems restricted to comparator gates. We show that the comparator-circuit value problem and the "lex-first maximal matching" problem are in *incr*-LOGSPACE while the comparator-network stability and the "man-optimal stable marriage problem" are in *rincr*-LOGSPACE(NLOGSPACE). This shows that the dynamic versions of these problems are solvable quickly in parallel even though there are no known NC algorithms to solve them from scratch.

1. Introduction

What is the "best" algorithm for a given problem? There is obviously more than one answer, depending on what we mean by "best." If we consider worst-case time behavior, traditionally an algorithm is considered the best possible if it meets the information theoretic lower bound. There is another way of looking at the situation, wherein we are not worried about the time taken to evaluate every instance from scratch. We ask that once the algorithm has preprocessed an instance of the problem, it should handle any changes to the instance very fast.

In Section 2 we formalize these notions in terms of the classes *incr*-POLYLOGTIME, the class of problems whose dynamic versions are solvable in poly-logarithmic time, and *incr*-POLYLOGSPACE, the class whose dynamic versions can be solved with poly-logarithmic work space. We also give a restricted notion of nondeterministic incremental computation. We show that if a problem can be solved sequentially with a small amount of work space then there exists an efficient dynamic algorithm to solve the same problem. We then introduce the concept of an *incremental reduction* between problems, and we present problems that are incrementally complete for other existing complexity classes under such a reduction.

In Section 3 we show that commonly known P-complete problems (under LOGSPACE reductions) are *incr*-POLYLOGTIME-complete for the class P. We further prove that all *nonredundant* P-complete problems are *incr*-POLYLOGTIME-complete for P. This suggests that problems that are hard to parallelize are hard to dynamize. We also show that a variant of transitive closure is *incr*-CONSTANTTIME-complete for NLOGSPACE and give similar problems which are *incr*-CONSTANTTIME-complete for LOGSPACE and NC^1. In Section 5 we demonstrate that under certain restrictions, problems which have efficient dynamic solutions also have efficient parallel solutions.

In Section 4 we consider a nonuniform model of incremental computation and show that in this model most problems have almost linear complexity. In addition, we present some techniques for lower bounding the complexity of explicitly defined problems.

In Section 6 we look at *incr*-LOGSPACE; the class of problems that can be "updated" using only logarithmic work space. This is an interesting class because problems in this class are amenable to fast parallel updates (since LOGSPACE $\subseteq$ NC^2). We show that the circuit-value problem for comparator gates, discussed in Mayr and Subramanian [15] is in *incr*-LOGSPACE while the network stability problem is in relativized *incr*-LOGSPACE with respect to an NLOGSPACE oracle. Since these problems have no known efficient parallel solutions, we get a new class of problems which seem to be somewhat more sequential in nature than the problems in NC but nonetheless exhibit a degree of parallelism absent in P-complete problems. In Section 7 we present the conclusions and some open problems.

2. Preliminaries

Informally, we want a decision problem π to belong to the class *incr*-TIME$[f(n)]$ if we can "react" to the changes in a given instance of the problem in time proportional to $f(n)$ where n is the size of the problem statement. First we take some time to build a data structure for an initial instance of the problem. After that point we are required to react to any one-bit change in the current instance in time $O(f(n))$; i.e. we should calculate the solution for the new instance and update the current data structure all in time $O(f(n))$.

Let π be a decision problem. An n-bit instance I of π is any n-bit string of zeros and ones. An instance I is said to be a positive instance if a decision procedure for π outputs a 1 given I as the input. A single bit change to an instance I would involve flipping some bit of I from 0 to 1 or vice versa. We define the size of an incremental change Δ to an instance I (where Δ changes I to I') to be the number of bits of I that are flipped. Another reasonable model of incremental change would be one that allows one to insert and delete "zeros" and "ones" at various points in I. Ideally we would like to use a definition that measures an incremental change by the minimum amount of information needed to convert I to I', but that would involve issues of Kolmogorov complexity, which would not allow concrete time or space complexity results.

Our *replacement model* that we define above, where we can replace various bits of I by "zeros" or "ones," is a reasonable measure of incremental change and at the same time a sufficiently simple model for developing a theory of incremental complexity. In the remainder of the paper we only deal with the replacement model of incremental computation.

A dynamic algorithm for solving a problem π will have two stages: the *preprocessing stage* and the *update stage*. Given an initial instance I^0, in the preprocessing stage, the algorithm constructs an internal data structure D_{I^0} associated with the instance I^0. The data structure D_{I^0} is just the portion of the internal state of the algorithm A (after it processes I^0) that is carried over to the next stage. In the update stage, given an incremental change Δ to the current instance I, the algorithm computes the answer to

the new instance I' and makes changes to the internal data structure D_I to get $D_{I'}$. The data structure D_I is the portion of the internal state of the algorithm (after the instance I has been processed) that is carried over to the next stage. We will use the term *current instance* to denote the instance I that was encountered last by the algorithm, and the term *current data structure* to refer to the associated internal data structure D_I constructed by A when it processes the instance I.

Strictly speaking the notation D_I is not entirely appropriate since the relevant portions of the internal state (after I has been processed) depend not only on I but also on the algorithm A, the initial instance I^0, and the sequence of updates that were used to arrive at I. We will use the notation D_I^A when we want to disambiguate between two algorithms but wherever there is no ambiguity we will continue using the notation D_I in the interest of not overloading the symbols with too many subscripts and superscripts.

Definition 2.1. A decision problem π belongs to the class *incr*-TIME $[f(n)]$ if there are RAM programs P_1 and P_2 such that for all $n \in \mathbb{N}$ we have

(1) Given any initial n-bit instance I^0 of π, P_1 efficiently precomputes some internal data structure D_{I^0} associated with instance I^0.

(2) Given the current input instance I, the incremental change Δ to I and the current data structure D_I in the random access memory, P_2 determines $\pi(I')$, and modifies D_I into the new data structure $D_{I'}$ in $O(|\Delta| f(n))$ time.

Our notion of "efficient" in condition 1 is somewhat flexible; we could require that the precomputation be doable in polynomial time or logarithmic space, for example. The RAM programs are allowed to operate on $O(\log n)$ length words in constant time, where n is the size of the problem instance. They are therefore allowed to read, write, add and subtract $O(\log n)$ length words. The program counter is also allowed to transfer control to an arbitrary address (e.g. to execute a jump).

We denote by *incr*-POLYLOGTIME the class

$$\bigcup_{k \geq 0} \textit{incr}\text{-TIME}[\log^k n].$$

We believe that such a definition captures the intuitive notion of incremental computation; e.g. the problem of maintaining an ordered set through a set of updates of queries is in *incr*-TIME$[\log n]$.

Given the technical inconvenience in making the data structure an explicit part of the definition one might argue that we do not mention the data structure explicitly but simply state that the updating program P_2 be allowed $O(|\Delta| f(n))$ time to modify its internal state. We have chosen to make the data structure an explicit part of the definition because doing so allows us to control the kind of access that we allow the updating procedure P_2 to have on the current data structure. For example, in defining the class *incr*-SPACE $[f(n)]$ we allow P_2 a read-only-access to the current data structure thus limiting the amount of work space that it can have.

Definition 2.2. A decision problem π belongs to the class *incr*-SPACE$[f(n)]$ if there are RAM programs P_1 and P_2 such that for all $n \in \mathbb{N}$ we have

(1) Given any initial n-bit instance I^0 of π, P_1 efficiently precomputes some internal data structure D_{I^0} associated with instance I^0.

(2) Given the current input instance I, the incremental change Δ to I and the current data structure D_I in a read-only memory, P_2 determines $\pi(I')$, and constructs the new data structure $D_{I'}$ on a write-only memory while using only $O(|\Delta| f(n))$ work space.

We denote by *incr*-LOGSPACE the class *incr*-SPACE$[\log n]$, and define *incr*-POLYLOGSPACE in a manner similar to *incr*-POLYLOGTIME.

To extend these notions to nondeterminism we allow the update procedure P_2 access to some nondeterministic decision oracle S. The decision oracle S can be used by P_2 to ask queries regarding the current input I and the current data structure D_I. P_2 is allowed to make a polynomial number of calls to S. By restricting P_2's access to decision oracles that are allowed to operate in NLOGSPACE we get the following definition for *rincr*-LOGSPACE(NLOGSPACE).

Definition 2.3. A decision problem π belongs to the class *rincr*-LOGSPACE(NLOGSPACE) (read as π is in *incr*-LOGSPACE relative to the class NLOGSPACE) if there are RAM programs P_1 and P_2 such that for all $n \in \mathbb{N}$

(1) Given any initial n-bit instance I^0 of π, P_1 efficiently precomputes some auxiliary data structure D_{I^0} associated with instance I^0.

(2) Given the current input instance I, the incremental change Δ to I and the current data structure D_I in a read-only random access memory, P_2 determines $\pi(I')$, and constructs the new data structure $D_{I'}$ (output to a write-only memory) while using $O(|\Delta| \log(n))$ work space.

The update procedure P_2 is allowed to make a polynomial number of calls to some nondeterministic decision oracle S that uses $O(\log n)$ work space. P_2 can take the help of the decision oracle S in determining $\pi(I')$ and for making decisions about how to construct $D_{I'}$.

We now show that problems which can be solved sequentially using small amounts of work space can be incrementally solved using small amounts of time per update.

2.1. Incremental execution of space bounded computations

Standard techniques give that a nondeterministic $s(n)$-space bounded computations can be simulated from scratch on a RAM in time $O(n2^{O(s(n))})$. In an incremental setting, we show that for small (sublogarithmic) space bounds the cost of updating

a solution is much smaller than the cost of constructing it from scratch. For instance, as a corollary to the following theorem, NSPACE[$\log\log n$] $\subseteq$ *incr*-POLYLOG-TIME.

Theorem 2.4. *Let $s(n)$ be a space bound that can be computed in $n^{O(1)}$ time such that $s(n)=O(\log n)$. Then,* NSPACE[$s(n)$] $\subseteq$ *incr*-TIME[$(\log n)2^{O(s(n))}$].

Proof. Consider the computation of a nondeterministic Turing machine M on a fixed input $x=x_1x_2\ldots x_n$. We assume that the input is given on a read only input tape with a blank symbol # at the beginning and the end. Put $x_0=x_{n+1}=\#$. We can assume that the head on the input tape does not leave the segment $x_0x_1\ldots x_{n+1}$ during the computation. Furthermore, by changing the finite control, we can assume that M accepts by letting the head on the input tape leave the segment at the end of the computation, and rejects by staying inside the segment. By a *semi-configuration* of M we mean a description of the content of each work tape and the position of each work tape head, but with content and head position of the input tape omitted. Let S denote the set of semi-configurations of M. Given a segment $x_i\ldots x_j$ of the input tape, consider the following binary relation $R_{i,j}$ between $S\times\{\mathrm{l},\mathrm{r}\}$ and $S\times\{\mathrm{L},\mathrm{R}\}$ think of l and r as denoting "enter from the left" and "enter from the right", respectively, and L and R as denoting "exit to the left" and "exit to the right"):

- $(u,\mathrm{l})R_{i,j}(v,\mathrm{R})$ iff when M is started with the input head in cell x_i while in semi-configuration u there is a computation where the input head leaves the segment $x_i\ldots x_j$ for the first time by moving from x_j to x_{j+1} while the machine enters semi-configuration v.
- $(u,\mathrm{r})R_{i,j}(v,\mathrm{L})$ iff when M is started with the input head in cell x_j while in semi-configuration u there is a computation where the input head leaves the segment $x_i\ldots x_j$ for the first time by moving from x_i to x_{i-1} while the machine enters semi-configuration v.
- $(u,\mathrm{l})R_{i,j}(v,\mathrm{L})$ iff when M is started with the input head in cell x_i while in semi-configuration u there is a computation where the input head leaves the segment $x_i\ldots x_j$ for the first time by moving from x_i to x_{i-1} while the machine enters semi-configuration v.
- $(u,\mathrm{r})R_{i,j}(v,\mathrm{R})$ iff when M is started with the input head in cell x_j while in semi-configuration u there is a computation where the input head leaves the segment $x_i\ldots x_j$ for the first time by moving from x_j to x_{j+1} while the machine enters semi-configuration v.

Suppose we know $R_{i,k}$ and $R_{(k+1),j}$ for some $i\leqslant k<j$ and we want to compute $R_{i,j}$. Let $R_1=R_{i,k}$ and $R_2=R_{(k+1),j}$. Consider the directed graph $G=(V,E)$ with nodes

$$V=S\times\{\mathrm{l},\mathrm{r},\mathrm{L},\mathrm{R},m_1,m_2\}$$

and edges given by the following rules:

$$
\begin{aligned}
(u, \mathrm{l}) R_1 (v, \mathrm{L}) &\Rightarrow \langle (u, \mathrm{l}), (v, \mathrm{L}) \rangle \in E \\
(u, \mathrm{l}) R_1 (v, \mathrm{R}) &\Rightarrow \langle (u, \mathrm{l}), (v, m_1) \rangle \in E \\
(u, \mathrm{r}) R_1 (v, \mathrm{L}) &\Rightarrow \langle (u, m_2), (v, \mathrm{L}) \rangle \in E \\
(u, \mathrm{r}) R_1 (v, \mathrm{R}) &\Rightarrow \langle (u, m_2), (v, m_1) \rangle \in E \\
(u, \mathrm{l}) R_2 (v, \mathrm{L}) &\Rightarrow \langle (u, m_1), (v, m_2) \rangle \in E \\
(u, \mathrm{l}) R_2 (v, \mathrm{R}) &\Rightarrow \langle (u, m_1), (v, \mathrm{R}) \rangle \in E \\
(u, \mathrm{r}) R_2 (v, \mathrm{L}) &\Rightarrow \langle (u, \mathrm{r}), (v, m_2) \rangle \in E \\
(u, \mathrm{r}) R_2 (v, \mathrm{R}) &\Rightarrow \langle (u, \mathrm{r}), (v, \mathrm{R}) \rangle \in E
\end{aligned}
$$

Let R be the binary relation from $S \times \{\mathrm{l}, \mathrm{r}\}$ to $S \times \{\mathrm{L}, \mathrm{R}\}$ defined by taking the transitive closure of G and restricting it to this domain. We refer to R as the concatenation $R_1 * R_2$ of R_1 and R_2. It is easily seen that if $i \leqslant k < j$ then $R_{i,j} = R_{i,k} * R_{(k+1),j}$. This suggests the following data structure: Given a tape segment $x_i \ldots x_j$, we maintain a representation of the relation $R_{i,j}$ by recursively maintaining $R_{i,k}$ and $R_{k+1,j}$ where $k = \lfloor \frac{i+j}{2} \rfloor$. The segments $x_i \ldots x_j$ for which the relation $R_{i,j}$ is kept form a binary tree of height $O(\log n)$. Since we keep a representation of $R_{0,n+1}$ we can decide membership of L in constant time. When a letter x_i in the input is changed, we only have to recompute each of the $R_{j,k}$ in the data structure for which $j \leqslant i \leqslant k$, i.e. $O(\log n)$ updates on relations on sets of size $2^{O(s(n))}$. Each update is a transitive closure which can be done in polynomial time, i.e. the entire operation takes time $O((\log n) 2^{O(s(n))})$. Furthermore, by the assumption on $s(n)$, the data structure can be initialized in time $n^{O(1)}$. □

2.2. *Incremental reductions*

To compare the "hardness" of solving two problems in this incremental sense, we need the notion of incremental reduction. It is clear that the program doing the reduction from problem π_1 to problem π_2 has to be incremental in the same sense as the RAM program P_2 in Definition 2.1. In the same way as before we allow some computation to construct a data structure to an initial instance of π_1 and to find the corresponding instance of π_2. We then require that any change to the current instance of π_1 be reflected as a change to the corresponding instance of π_2, within a stated time limit. There are however two differences since we are now dealing with functions (in the previous case the solution was always a "yes" or a "no") which map instances of π_1 to instances of π_2. We need to quantify the relative sizes of the corresponding instances as well as the amount of change the mapping undergoes when the input changes by a specified amount.

We take care of these by introducing two more parameters to quantify a reduction.

Definition 2.5. A decision problem π_1 is *incrementally reducible* to another decision problem π_2 with time and size bounds $[f(n), g(n), p(n)]$, denoted $\pi_1 \leqslant_{incr[f(n),g(n),p(n)]} \pi_2$, if

(1) There is a transformation $T: \pi_1 \to \pi_2$ that maps instances of π_1 to instances of π_2 such that for any n-bit instance I of π_1 the size of the corresponding instance $T(I)$ of π_2 is bounded above by $p(n)$. Furthermore, I is a positive instance of π_1 if and only if $T(I)$ is a positive instance of π_2.

(2) There are RAM programs P and Q such that for all $n \geqslant 0$ we have

(a) Given any n-bit initial instance I^0 of π_1, P efficiently computes $T(I^0)$ and some auxiliary data structure S_{I^0} associated with I^0.

(b) Given the current instance I, the incremental change Δ_1 to I (where Δ_1 changes I to I' of π_1) and the current data structure S_I in the random access memory, Q constructs the incremental change Δ_2 to $T(I)$ (where Δ_2 changes $T(I)$ to $T(I')$), such that $|\Delta_2| \leqslant g(n)|\Delta_1|$, and modifies S_I into the new data structure $S_{I'}$ in $O(|\Delta_1| f(n))$ time.

Note that $g(n)$ is always $O(f(n))$.

We use the symbol S_I to denote the data structure maintained by the transformation T so as to differentiate it from D_I which is the data structure associated with an algorithm that solves a given problem π.

Theorem 2.6. *If π_1 is incrementally reducible to π_2 in time and size bounds $[f(n), g(n), p(n)]$, and if π_2 is in incr-TIME$[h(n)]$, then π_1 is in incr-TIME$[f(n)+h(p(n))g(n)]$.*

Proof. We now give a dynamic algorithm B to solve π_1 that is a composition of the transformation T (from π_1 to π_2) and the dynamic algorithm A that solves π_2 in incremental time $f(n)$. The basic idea is as follows: We run the transformation algorithm T in the background and whenever the current instance I of π_1 changes, we use T to compute the incremental change to the corresponding instance $T(I)$ of π_2 and feed that change to algorithm A. By the definition of the transformation the answer given by A is also the answer for the modified instance I' of π_1.

More specifically, in the preprocessing stage, given the initial instance I^0, we start the background transformation T that maps I^0 of π_1 to the corresponding instance $T(I^0)$ of π_2. We now also start the dynamic algorithm A, for solving π_2, with $T(I^0)$ as the initial instance. The internal data structure of B is therefore a combination of the internal data structures S_{I^0} of the transformation algorithm T and T_{I^0} of the algorithm A. In other words $D^B_{I^0} = S_{I^0} \cup D^A_{T(I^0)}$.

By the definition of the transformation T, $\pi_2(T(I^0))$ is the same as $\pi_1(I^0)$. Therefore the initial computation on $T(I^0)$ computes $\pi_1(I^0)$.

Given an incremental change Δ_1 to the instance I, we first use the dynamic transformation algorithm T to compute the incremental change Δ_2 to $T(I)$ in time $O(|\Delta_1| f(n))$ time. At this time the data structure S_I is modified to reflect the change in the input. Then, we give Δ_2 as input to the dynamic algorithm A, which computes

$\pi_2(T(I'))$ and modifies the data structure $D^A_{T(I)}$ to get $D^A_{T(I')}$. By definition $\pi_2(T(I'))=\pi_1(I')$ therefore the answer returned by A is correct.

By construction $|\Delta_2| \leqslant |\Delta_1| g(n)$. Also, since the size of $T(I)$ is no more than $p(n)$ and π_2 is in *incr*-TIME$[h(n)]$, algorithm A takes $O(|\Delta_2| h(p(n)))=O(|\Delta_1| g(n)h(p(n)))$ time to modify $D^A_{T(I)}$ and to compute $\pi_2(T(I'))$. Therefore, the time taken to compute $\pi_1(I')$ and to modify D^B_I is the time for computing Δ_2 summed with the time required by algorithm A to compute $\pi_2(T(I'))$. Thus the total time for the updating process is $O(|\Delta_1| f(n)+|\Delta_1| h(p(n))g(n))$. Hence, π_1 is in *incr*-TIME$[f(n)+h(p(n))g(n)]$. □

3. Incrementally complete problems

We now turn to the notion of incremental completeness. The idea is to find a problem which is hard to solve incrementally as any other problem in a given class. In this section we use incremental reductions to get natural problems incrementally complete for P, NLOGSPACE, LOGSPACE, and NC^1.

Definition 3.1. A problem π is said to be *incr*$[f(n), g(n), p(n)]$-*complete* for a class C if

(1) π is in the class C.

(2) For all π_1 in C, π_1 is incrementally reducible to π in time and size bounds $[f(n), g(n), p(n)]$.

We call a problem π *incr*-POLYLOGTIME-complete for a class C if π is *incr*$[f(n), g(n), p(n)]$-*complete* for C, where $p(n)$ is bounded above by some polynomial in n, and $f(n)$ and $g(n)$ are $O(\log^k n)$ for some k. We define *incr*-CONSTANT-TIME-completeness in an analogous fashion.

The obvious question to explore is how the complexity classes *incr*-POLYLOG-TIME, *incr*-POLYLOGSPACE, and *incr*-LOGSPACE are related to the well known sequential complexity classes? Our first intriguing result is that the commonly known P-complete problems listed in [18, 10] are *incr*-POLYLOGTIME-complete for the class P. These problems include the circuit-value problem (CV), the solvable path system problem (SPS), propositional horn satisfiability, and a host of other well known P-complete problems. These problems are therefore as hard to solve incrementally as any other problem in P.

Theorem 3.2. *All* P-*complete problems in* [18, 10] *are incr*-POLYLOGTIME-*complete for the class* P.

Proof. We present the proof for the case of circuit-value problem. In this problem we are given a directed acyclic graph as the input. A node in this graph is either labeled an *input node* or an *output node*, or it corresponds to a gate in the circuit. A gate could be AND, OR, or NOT gate. The edges in the graph correspond to wires of the circuit. All the input nodes have in-degree zero and out-degree one and all the output nodes have

in-degree one and out-degree zero. The AND and OR gates have in-degree at least two and out-degree at least one while the NOT gate has both in-degree and out-degree equal to one. Given an assignment of zeros and ones to the input nodes and a specific output node X, our aim is to find the value of the output node. In other words our aim is to find the value of the output wire attached to X given a particular assignment to the input wires of the circuit.

To prove that the circuit-value problem is *incr*-POLYLOGTIME-complete for P we proceed as follows: Given any problem π in P and an initial instance I^0 with $|I^0| \leqslant n$, we use the standard LOGSPACE reduction [14] to create a circuit of size $t(n)$ by $t(n)$ (where $t(n)$ is a time bound for some polynomial-time turing machine M to solve instances of π of size at most n) that simulates the turing machine M used to solve the problem π.

The inputs to this circuit are the bits of the initial instance I^0. This gives us the initial transformation of I^0. A one bit change to I^0 results in a one bit change to the corresponding instance of the CV (the input variable in the circuit corresponding to the changed bit is changed to reflect the new input). All this can be done in constant time, and thus the circuit-value problem is *incr*-POLYLOGTIME-complete for P.

We can provide similar proofs for the other problems listed in [18, 10]. The reductions needed to show this are sometimes different from those given in the literature. The modifications are minor and are needed to ensure the following property: For the reduction to be incremental we require that when a general problem π in P is changed by one bit, input to the incrementally complete problem under consideration changes only by a polylog number of bits. The use of preprocessing plays a significant role, since the size of the instance of the incrementally complete problem may be a polynomial factor larger than the problem π in P. □

Corollary 3.3. *If the* P*-complete problems in* [18, 10]*, like circuit-value problem, are in incr-*POLYLOGTIME *then all of* P *is in incr-*POLYLOGTIME.

Corollary 3.4. *If there are* NC *algorithms to update dynamically the* P*-complete problems in* [18, 10]*, then we automatically get* NC *algorithms to update dynamically all of* P.

These corollaries suggest that it is highly unlikely that we can use polylogarithmic time sequential or parallel algorithms to incrementally update these problems.

Delcher and Kasif [5] propose other notions of completeness. They define the incremental version of a function f as follows: Let f be a function that maps input X of length $|X|$ to output $f(X)$. Then Inc f is the function that given inputs of form $(X, f(X), X')$, computes $f(X')$, where X and X' differ in at most $\log|X|$ bits. A function f is Inc P-complete iff for every function g in P, Inc g is LOGSPACE-reducible to Inc f. With this definition they argue that the "all-outputs monotone circuit value problem" is Inc-P-complete. The drawback in this approach is that it disallows the use

of dynamic data structures. In other words their incremental reductions are not strong enough to conclude that it is unlikely that one will get good algorithms (meaning NC algorithms) for the incremental versions of Inc-P-complete problems irrespective of the size of the auxiliary storage. They try to overcome this drawback by proving that the incremental versions of many P-complete problems are also P-complete. Here they completely ignore the issue of preprocessing. Although their results are interesting, they are somewhat weak in that the issues of data structures and preprocessing are overlooked. They conjecture that the incremental versions of all P-complete problems are P-complete.

Interesting notions of completeness are sketched by Reif [21]. He shows that some problems are unlikely to have efficient incremental solutions, but he does not develop a comprehensive theory or consider the necessary details of preprocessing. Some interesting techniques are explored in [2, 6, 20] to derive lower bounds for incremental algorithms. Their main idea is to construct an algorithm for the batch version of the problem with input I by computing a dynamic data structure for some easily computed instance I', whose Hamming distance from I is small, and then applying the incremental algorithm repeatedly while I' is successively modified to I. This gives lower bounds for the incremental problem in terms of that for the batch problem. The drawbacks of this approach are that it limits the amount of preprocessing available and is problem specific.

Theorem 3.2 would seem to suggest that all P-complete problems are *incr*-POLYLOGTIME-complete for P. Unfortunately that is not the case, since one can take any *incr*-POLYLOGTIME-complete problem π and create another problem π' by duplicating the input, so that π' is no longer *incr*-POLYLOGTIME-complete even though it is still P-complete. In fact we prove that some P-complete problems can be solved effectively incrementally.

Theorem 3.5. *There are* P*-complete problems that are in incr-*POLYLOGTIME.

Proof. Let L be some P-complete language over the alphabet $\Sigma = \{0, 1\}$ such that it takes linear sequential time to determine whether a given word w is in L or not (the restriction to languages that can be recognized in linear time is only made for the sake of convenience, the same can be done with any P-complete language). Let us consider the language $L' = \{w^{|w|} | w \in L\}$. Obviously L' is also P-complete. We claim that membership queries for L' can be performed dynamically. We will abuse notation slightly by using the symbol L' both for the language and the associated membership problem. Consider an initial n^2-bit instance I^0 of the problem L' (instances which are not of the form n^2 for some $n \in N$ can be handled without too much trouble). We spend polynomial time to see if it is of the form $w^{|w|}$ for some $w \in L$. This, we can do by running a subroutine S_L to check membership in L and some associated linear time work to see if I^0 is of the form $w^{|w|}$. At any point in time let the current instance I be a concatenation of n bit strings $a_1, a_2, \ldots, a_n$. The idea behind our dynamic algorithm is as follows.

As long as less than $n/2$ of the a_i's are equal (i.e., less than $n/2$ of the strings are the copy of the same string w) we answer "no" for every instance. If at some point more than $n/2$ of the a_i's are all equal to the same w, we start a background process which executes the subroutine S_L on the string w. As long as more than half the a_i's are equal to w we spend a constant amount of time on the background computation for every update to the current instance. Therefore, by the time I becomes equal to $w^{|w|}$ we would have run S_L for a linear number of steps and thus will know whether w is in L or not. We can therefore answer the membership query of I in L' at that point.

It seems hard to dynamically figure out whether more than half the a_i's are all equal to the same w. Therefore, we perform the following easier computation.

(1) *The data structure.* We divide each a_i into $k = n/\log n$ words a_i^1–a_i^k, each of which is composed of $\log n$ consecutive bits from a_i. We now construct k sets $S_1, S_2, \ldots, S_k$ such that the set S_j contains the jth word from each a_i. In other words $S_j = \{a_1^j, a_2^j, \ldots, a_n^j\}$. With each set S_j we maintain two flags f_j and g_j. The flag f_j is set to 1 if more than half the words in that set become equal to the same word w^j, otherwise it is set to 0. The flag g_j is set to 1 if all the words in S_j are the same, otherwise it is set to 0. The word w^j associated with S_j is called the *majority word* of S_j. If the flag f_j of S_j is 0 then w_j is set to a string of $\log n$ 0's. We also maintain a string w that is the concatenation of $w^1, w^2, \ldots, w^k$ in that order.

(2) *The algorithm*

- As long as at least one of the flags $g_1, g_2, \ldots, g_k$ is equal to 0 we answer "no" for every instance.
- If at some point all of the f flags $f_1, f_2, \ldots, f_k$ are equal to 1, we start a background process which executes the subroutine S_L on the string w maintained by the data structure; and from then on we simulate a constant number of steps of S_L for every update operation (the constant depends on the sequential time complexity of L).
- It at some point one of the f flags becomes equal to 0 we abort the background process that runs the subroutine S_L.
- If all of the g flags g_1–g_k are equal to 1 we get the answer to the background computation on w from S_L and return that as the answer.

By definition when all the g flags g_1–g_k are equal to 1 the input is of the form $\bar{w}^{|w|}$, for some $\bar{w}$. Therefore, to prove our algorithm correct we need to prove that by the time all the g flags become 1 the background process already has the answer to whether $\bar{w}$ is in L or not.

To prove the correctness of our algorithm let us consider the sequence of instances starting at I^0, going through $I^1, I^2, \ldots$, all the way up to the current instance $I = I^l$ for some l, where every instance I^x is derived from the previous instance I^{x-1} by updating the bit that was changed at time $x-1$. Let I^h be the instance closest to I such that it is a concatenation of the strings $a_1, a_2, \ldots a_n$ and $n/2+1$ of the a_i's are equal to $\bar{w}$. Since at instance h only $n/2+1$ of the a_i's are equal to $\bar{w}$, $l-h \geq n/2-1$. Suppose we start the

background process S_L on the string $\bar{w}$ at time h and execute a constant number of its operations for every subsequent update. Since $l-h \geq n/2-1$ and S_L requires linear time to recognize $\bar{w}$, at instance l we would know whether $\bar{w}$ is in L or not.

We now show that the algorithm outlined above will start the background process at time h. Consider a division of $\bar{w}$ into $n/\log n$ words $\bar{w}^1, \bar{w}^2, \ldots, \bar{w}^k$ each of which consists of $\log n$ consecutive bits from $\bar{w}$. Since at time h more than half of the strings are equal to $\bar{w}$, for each set S_j the flag f_j is set and the majority word w^j of the set is equal to $\bar{w}^j$. Therefore, the string formed by concatenating all the majority words is equal to $\bar{w}$. Hence, our algorithm will start the background computation S_L on $\bar{w}$ at the right time thus ensuring the availability of the answer when all the g flags become 1.

We now show in detail how the sets S_1–S_k can be maintained dynamically as the input I undergoes changes. Each set S_j is maintained at the leaves of a balanced binary search tree T_j. The words $a_1^j, a_2^j, \ldots, a_n^j$ are stored at its leaves in sorted order (by treating each a_i^j as $\log n$ bit integer). Note that since the words are sorted all the words that are equal to each other are stored in contiguous leaves.

Each internal node i in T_j maintains three numbers max_i, $right_i$, and $left_i$ and three associated words max_word_i, $left\_word_i$ and $right\_word_i$. The quantity max_i indicates the maximum number of words in the subtree rooted at i that are equal to the same word max_word_i which is the word stored at the "*left-most*" leaf in the subtree rooted at i, while $right_i$ is equal to the number of words that are equal to the word $right\_word_i$ which is the word stored at the "right-most" leaf in the subtree rooted at i. We also store a value $total_i$ that is equal to the total number of leaves in the subtree rooted at i.

Let z be an internal node in T_j, and x and y be, respectively, the left and right child of z. We can then use the following program to derive the values of $total_z$, $left_z$, $right_z$, max_z, $left\_word_z$, and max_word_z from corresponding values at x and y:

procedure update
begin
[The default case]
  $total_z \leftarrow total_x + total_y$
  $left\_word_z \leftarrow left\_word_x$
  $left_z \leftarrow left_x$
  $right\_word_z \leftarrow right\_word_y$
  $right_z \leftarrow right_y$
  if $(max_x \geq max_y)$ then
    $max_z \leftarrow max_x$
    $max\_word_z \leftarrow max\_word_x$
  else
    $max_z \leftarrow max_y$
    $max\_word_z \leftarrow max\_word_y$

```
[when the boundary words of x and y are the same]
  if (right_word_x = left_word_y) then
    if (left_x = total_x) then
      left_z ← left_x + left_y
    if (right_y = total_y) then
      right_z ← right_x + right_y
    if (max_z < (right_x + left_y)) then
      max_z ← right_x + left_y
      max_word_z ← right_word_x
  end
end [procedure update]
```

Given the values at the nodes x and y these operations can be performed in constant time to derive the values at node z. Whenever the current instance I changes by a single bit, exactly one of the words a_x^j (for some x and j) changes. Therefore only one of the sets S_j needs to be updated. The elements of S_j are stored in sorted order at the leaves of a balanced binary search tree. Therefore, a change in one of the a_x^j's implies we have to delete it from its current location in tree and reinsert it in another location so as to preserve the sorted order. A delete or an add operation causes changes along a leaf-to-root path (starting at the leaf-node that was deleted or added). However, since the tree is balanced there are only $O(\log n)$ nodes along any leaf-to-root path. This implies that all the values along this path can be appropriately changed in $O(\log n)$ time to reflect the change at the leaf node. Thus the data structure can be modified in $O(\log n)$ time to reflect a single-bit change in the current instance I.

The above construction works by taking an *incr*-POLYLOGTIME-complete problem and introducing redundancy by copying, and in fact all the ways to generate incrementally tractable problems from P-complete problems seem to involve redundancy in some form or the other. We restrict ourselves therefore to P-complete problems that are in some sense nonredundant. To do that we first look at a much stricter definition of P-completeness; one in terms of projections, introduced by Skyum and Valiant [22].

Definition 3.6. A decision problem π_1 is *projection reducible to another decision problem* π_2 ($\pi_1 \leqslant_{proj} \pi_2$) if there is a function $p(n)$ bounded above by a polynomial in n, and a polynomial time computable family of mappings $\sigma = \{\sigma_n\}_{n \geqslant 1}$ where

$$\sigma_n : \{y_1, \ldots, y_{p(n)}\} \to \{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}, 0, 1\},$$

such that for any n-bit instance I of π_1 we can derive a $p(n)$-bit instance I' of π_2 using the mapping σ_n. The mapping has the property that I is a positive instance of π_1 if and only if I' is a positive instance of π_2. To derive I' we give x_i the same value as the ith bit of I and use $\sigma_n(y_i)$ as the bit of I'. We say that σ is bounded above by p and π_1 is *σ-reducible* to π_2.

Definition 3.7. A problem π is said to be $<_{proj}$-*complete* for a class C if π is in C and there is a function $p(n)$ bounded above by a polynomial in n such that every problem $\pi_1 \in C$ is projection reducible to π by a projection $\sigma = \{\sigma_n\}_{n \geq 1}$ bounded above by p.

Problems like the circuit-value problem are $\leqslant_{proj}$-complete for P. Even under this restricted setting it is possible to create problems which are $\leqslant_{proj}$-complete for P but are in *incr*-POLYLOGTIME. To remedy that we introduce the following notion of *nonredundancy*.

Definition 3.8. Let π_1 be a decision problem and π be another problem such that $\pi_1 \leqslant_{proj} \pi$. We say that π is *nonredundant with respect to* π_1 if there exists a polynomial time computable family $\sigma = \{\sigma_n\}_{n \geq 1}$ of mappings and a number $k \in \mathbf{N}$ such that π_1 is σ-reducible to π (where σ_n is a mapping from the set $\{y_1, \ldots, y_{p(n)}\}$ to $\{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}, 0, 1\}$), and for all symbols x_i, $|\sigma_n^{-1}\{x_i, \overline{x_i}\}| = O(\log^k n)$.

Intuitively the reduction from π_1 to π is nonredundant if in the projection mapping a single bit x_i of π_1 does not affect more than O(polylog(n)) bits of π.

We now define the notion of nonredundancy with respect to a class C.

Definition 3.9. Let π be a problem which is $\leqslant_{proj}$-complete for C. We say that π *is nonredundant with respect to* C if it is nonredundant with respect to every problem in C. We then call π a *nonredundant projection complete problem* (or a NRP-complete problem) for the class C.

Lemma 3.10. *Let C be a class of decision problems and let π_1 be a NRP-complete problem for C. If π is another decision problem in C such that $\pi_1 \leqslant_{proj} \pi$ and π is nonredundant with respect to π_1 then π is NRP-complete for C.*

Proof. The proof follows from the definitions in a straightforward manner. □

Th P-complete problems listed in [18, 10] are all NRP-complete for P. The following theorem shows that this large class of P-complete problems are difficult to make efficient incrementally.

Theorem 3.11. *Let C be a class of decision problems and π be an NRP-complete problem for C, then π is incr*-POLYLOGTIME*-complete for C.*

Proof. The proof follows from noting the fact that given any problem $\pi_1 \in C$, there is projection mapping from π to π_1 (that can be computed in the preprocessing stage)

such that a one bit change in an input instance of π_1 causes at most polylog number of bits to change in the corresponding instance of π. □

We now look at the classes NLOGSPACE, LOGSPACE, and NC^1.

Theorem 3.12. *The following variant of transitive closure is incr-*CONSTANTTIME*-complete for* NLOGSPACE:

Input: *A directed graph* $G=(V,E)$ *such that each* $e \in E$ *is colored with* 0 *or* 1; *a pair of vertices* v_1 *and* v_2; *a partitioning of* V *into sets* $V_1, V_2, \ldots, V_k$; *and a color vector* C *of length* k.

Output: *Is there a path* p *from* v_1 *to* v_2 *such that all edges in* p *emanating from vertices in* V_i *have color* $C[i]$?

Proof. The reduction from any problem in NLOGSPACE to this variant is straightforward. The trick that makes the reduction go in constant time and constant storage is the use of the sets $V_1, V_2, O, \ldots, V_k$ and the color vector. We group all intermediate states that read the input bit i into the class V_i; $C[i]$ is assigned the same value as input bit i. Now a change in some input bit i changes $C[i]$. □

A restriction of the above problem is *incr*-CONSTANTTIME-complete for LOGSPACE while a similar variant of bounded width polynomial size branching programs [1] gives us an *incr*-CONSTANTTIME-complete problem for nonuniform NC^1.

Surprisingly, however, there are problems which are NC^1-complete for LOGSPACE, but are nevertheless in *incr*-POLYLOGTIME. We consider the problem of undirected forest accessibility (UFA). The UFA problem is defined as follows: Given a forest of undirected trees (there should be at least two trees) and given two nodes u and v, determine if they are in the same tree. This was shown to be NC^1-complete for LOGSPACE by Cook and McKenzie [4]. Thus from the point of view of parallel computation UFA is as hard as any problem in LOGSPACE. However, as the following theorem shows, UFA is in *incr*-TIME$[\log n]$ while we do not know whether all problems in LOGSPACE can be efficiently dynamized. Thus in the dynamic setting UFA is not the hardest problem in LOGSPACE.

Theorem 3.13. *The undirected forest accessibility problem is in incr-*TIME$[\log n]$, *under additions, deletions, linking and cutting of trees.*

Proof. The dynamic maintenance is done by using balanced trees such as red-black trees to maintain ordered sets with insert, delete, split and join [11] (see also [23]).

- *Preprocessing*: An Euler tour of each tree in the forest is maintained in a red-black tree. Each edge of the tree occurs twice in its Euler tour.
- *Query*(n_1, n_2): Let e be an edge in the adjacency list of n_1 and f an edge in the list of n_2. We follow pointers in the red-black tree to determine if they both have the same root.

- Linking and cutting are implemented using the split and join operations of red-black trees, with $O(\log n)$ time per operation. □

4. Nonuniform complexity

In traditional, nonincremental complexity, it has proven useful to consider computing in *nonuniform* models, e.g. to consider the circuit complexity of decision problems. In this section, we consider the nonuniform complexity of incremental problems. Given a decision problem π, we may consider it as a subset of $\{0,1\}^*$ and consider its restriction π_n to $\{0,1\}^n$, i.e. we consider π as a family of Boolean functions. The nonuniform model in which we consider implementing this restricted problem is the *cell probe* or *decision assignment tree* model, previously considered by Fredman [7, 8] and Fredman and Saks [9]. In this model, the complexity of a computation is the number of cells accessed in the random access memory containing the data structure during the computation, while the computation itself is for free (and information about which operation to perform is also given for free). The number of bits B in a cell is a parameter of the model. Formally, the model is as follows: For each of the $2n$ possible incremental changes (corresponding to changing each bit to either 0 or 1), we assign *a decision assignment tree*, i.e. a rooted tree containing *read* nodes and *write* nodes. When performing an operation we proceed from the root of its tree to one of the leaves. The read nodes are labeled with a location of the random access memory. Each has 2^B sons, one for each possible content of the memory location. The write nodes which are unary are labeled with a memory location and a value between 0 and 2^B-1. When such a node is encountered, the value is written in the memory location. In the leaf finally encountered, the answer to the decision problem is found. The complexity of an implementation is the depth of its deepest tree, initialization of the data structure is for free.

Definition 4.1. If a problem π_n can be solved with cell size B and a system of trees of depth at most d, we say that $\pi_n \in \text{CPROBE}[B,d]$.

The model is very general, focusing on the *storage and access aspect* of data structuring. We consider two natural choices of B, namely, $B=1$ and $B=\log n$. We note that if a decision problem π is in *incr*-TIME$[f(n)]$, then $\pi_n \in \text{CPROBE}[\log n, O(f(n))]$. Thus, lower bounds in the $B=\log n$ model is also lower bounds for the complexity of solving the problem on a random access machine.

The following theorem gives tight bounds on the complexity of almost all functions.

Theorem 4.2. *For any $B \geq 1$, the following holds. For any function $f: \{0,1\}^n \to \{0,1\}$,*

$$f \in \text{CPROBE}[B, \lceil n/B \rceil].$$

Furthermore, for almost all functions f on this domain,

$$f \notin \text{CPROBE}[B, (n - 2\log n - 4)/B].$$

Proof. The upper bound follows from the data structure consisting of the string $x \in \{0,1\}^n$ to be maintained itself. For the lower bound we only have to consider the case $B = 1$. We can determine f from the system of $2n$ decision assignment trees implementing the incremental version of f and the state of the data structure when the value to maintain is initialized to $x = 0^n$. However, note that we may without loss of generality assume that the initial state of the data structure consists entirely of cells containing 0's, since we can convert an algorithm with a different initial state into such an algorithm by "hardwiring" information about the initial state into the algorithm. Thus, we only have to give an upper bound on the number of systems of trees of depth d. The total number of nodes and leaves in the trees in such a system is at most $2n(2^{d+1} - 1) \leq 4n2^d$. Thus, there are at most $4n2^d$ cells in the data structure, we may assume without loss of generality that these are numbered $1, \ldots, 4n2^d$. Each node/leaf is either a "answer 0" leaf, an "answer 1" leaf, one of $4n2^d$ possible read nodes or one of $8n2^d$ possible write nodes. There are thus at most $(1n2^d)^{4n2^d}$ systems. Therefore, there are at most $2^{2^{n-1}}$ different systems of depth $d = \lfloor n - 2\log n - 4 \rfloor$. There are, however, 2^{2^n} Boolean functions on n inputs. □

Thus, almost all decision problems have nonuniform complexity $\Theta(n/B)$ and are thus not in "nonuniform *incr*-POLYLOGTIME". However, in order to make progress on the P vs. *incr*-POLYLOGTIME question, we have to give lower bounds for easily computed functions.

We will present two methods giving such bounds: The first gives bounds of the type $f \notin \text{CPROBE}[1, o(\log n)]$ for some special, but easily computed, functions. The idea of the proof is very similar to Nečiporuk's method for proving lower bounds on formula size (see Boppana and Sipser [3]): If a function has many subfunctions on a small set of variables, it must have high complexity.

Let f be a Boolean function on the set of variables $X = \{x_1, x_2, \ldots, x_n\}$. A subfunction of f on $Y \subseteq X$ is a function obtained from f by setting the variables of $X - Y$ to constants.

Theorem 4.3. *Let $f: \{0,1\}^n \to \{0,1\}$ be a function that supports s different subfunctions on a set of variables Y of size m. Then*

$$f \notin \text{CPROBE}[1, \log\log s - \log\log\log s - \log m - 4].$$

Proof. Let a system of decision assignment trees of depth d for the incremental version of f be given. We can get an algorithm for the incremental problem for each of the subfunctions on Y by performing a sequence of operations from the initial state changing the value of the variables in $X - Y$ to the appropriate values and letting the resulting state be the initial state of the data structure. As in the proof of Theorem 4.2,

there are at most $(13m2^d)^{4m2^d}$ different functions on m variables in CPROBE$[1, d]$, so we must have $(13m2^d)^{4m2^d} \geqslant s$. This implies

$$d > \log\log s - \log\log\log s - \log m - 4. \quad \square$$

Let the *storage access* function $\text{ACCESS}_{n+\lceil\log n\rceil}: \{0,1\}^{n+\lceil\log n\rceil} \to \{0,1\}$ be the function which takes as input a bit vector $x = x_0 x_1 \ldots x_{n-1} \in \{0,1\}^n$ and a binary vector $y \in \{0,1\}^{\lceil\log n\rceil}$ denoting an integer i and outputs x_i. Let the *element distinctness* function $\text{DISTINCT}_{n\lceil 2\log n\rceil}: \{0,1\}^{n\lceil 2\log n\rceil} \to \{0,1\}$ be the function which takes as input n Boolean strings of length $\lceil 2\log n\rceil$ and outputs 1 if and only if all strings are distinct. Clearly, $\text{DISTINCT}_{n\lceil 2\log n\rceil}$ and $\text{ACCESS}_{\lceil\log n\rceil+n}$ are both in CPROBE$[1, O(\log n)]$. Theorem 4.3 provides matching lower bounds for these functions.

Corollary 4.4. *Neither* $\text{ACCESS}_{n+\lceil\log n\rceil}$ *nor* $\text{DISTINCT}_{n\lceil 2\log n\rceil}$ *are in* CPROBE$[1, o(\log n)]$.

It is however, easy to see that Theorem 4.3 is unable to provide larger bound on Boolean functions on n variables than $\log n - O(\log\log n)$.

Lower bounds in the $B = \log n$ model can be derived from a lower bound by Fredman and Saks for the complexity of maintaining an array during changes and prefix queries. Fredman and Saks [9] show that any cell probe algorithm for the problem of maintaining a vector $(x_0, \ldots, x_{n-1}) \in \{0,1\}^n$ during `change`(i) operations which flip the value of x_i and `prefix`(j) queries returning $x_0 \oplus x_1 \oplus \cdots \oplus x_j$ uses $\Omega(\log n/\log\log n)$ probes per operation ($\oplus$ denotes exclusive or). From this result, we get lower bounds on decision problems like the undirected forest accessibility problem. Consider the UFA problem in which we have to determine whether the nodes 1 and n are in the same tree or not:

Theorem 4.5. $\text{UFA} \notin \text{CPROBE}[\log n, o(\log n/\log\log n)]$

Proof. Assume that the incremental version of UFA can be solved in $O(\log n/\log\log n)$ probes. We show that the same holds for the prefix problem. We maintain the vector $(x_0, x_1, \ldots, x_{n-1})$ by maintaining the graph $G = (V, E)$ with $V = \{1, \ldots 2n+3\}$ and

$$E = \bigcup_{x_i = 0} \{(2i+1, 2i+3), (2i+2, 2i+4)\} \cup \bigcup_{x_i = 0} \{(2i+1, 2i+4), (2i+2, 2i+3)\}.$$

A change of an x_i corresponds to a constant number of insertions and deletion of edges. In order to compute $x_0 \oplus x_1 \oplus \cdots \oplus x_j$, we insert the edge $(2j+3, 2n+3)$. If vertex 1 and vertex $2n+3$ are in the same tree, the answer is 0, otherwise it is 1. After getting the answer we remove the edge $(2j+3, 2n+3)$. $\square$

Corollary 4.6. UFA$\notin$*incr*-TIME[$o(\log n/\log\log n)$]

However, for no polynomial-time computable function do we know any cell probe lower bounds better than $\Omega(\log n)$ (for $B=1$) or $\Omega(\log n/\log\log n)$ (for $B=\log n$). Once again, we see a similarity between incremental and parallel complexity: A nonuniform measure of parallel time is *depth*, and for no problem in P do we know a better lower bound on its depth than $\Omega(\log n)$, although we know that most problems require linear depth, which is also an upper bound.

5. Incremental computation in restricted realms

The class of *Z-stratified trees* was introduced by Overmars [19] in an effort to form a general theory of balancing in search trees. It subsumes a large portion of the search structures used to design dynamic algorithms. In this section we show that under reasonable assumptions problems which have incremental solutions based on these structures also have parallel solutions.

The class of Z-stratified trees generalizes (among others) the following classes of balanced search trees: AVL-trees, generalized AVL-trees, one-sided height-balanced trees, power trees, 2-3 trees, B-trees, symmetric binary B-trees, height-balanced 2-3 trees, k-neighbor trees and $BB[\alpha]$ trees. Thus Z-stratified trees generalize a large class of structures that are used in designing dynamic algorithms.

For the sake of completeness we now give some definitions from [19] that characterize the class of Z-stratified trees. For a more complete treatment the reader is referred to [19].

Definition 5.1. Let X be a class of balanced binary trees and for each $k\geqslant 0$ let X_k denote the subclass of trees in X that have height k (the height of a tree is the longest path from the root to a leaf). X is called α-proper (for $\alpha\geqslant 0$) if and only if for each $t\geqslant\alpha$ there is a tree in X with t leaves.

Definition 5.2. Let Z be a set of trees all of the same height β (Z need not be a subset of X_β). Also, let l_Z be the smallest and the largest number of leaves (respectively) that any tree in Z can have. Z is a β-variety iff the following conditions hold:

(1) $1<l_z<h_Z$

(2) For each t such that $l_Z\leqslant t\leqslant h_Z$ there is a tree in Z with exactly t leaves.

Let $T_1, T_2, \ldots, T_t$ be trees and let T be a tree with t leaves $x_1, x_2, \ldots, x_t$ from left to right. The composition $T[T_1, T_2, \ldots, T_t]$ denotes the tree obtained by grafting T_i onto x_i for every i. In other words for each i such that $1\leqslant i\leqslant t$ we replace the ith leaf x_i of T by the root of the ith tree T_i.

Definition 5.3. Let X be an α-proper class and let Z be a β-variety. We say that Z is a *regular* β-variety for X iff $\forall t \geqslant \alpha$, for any tree $T \in X_t$ and $T_1, T_2, \ldots, T_t$ in Z the composition $T[T_1, T_2, \ldots, T_t]$ is in X.

Let X be a α-proper class of trees and let Z be a regular β-variety of X. Essentially the class of Z-stratified trees in X are the trees in X that can be formed by taking *small* trees from X and building larger trees from them by layering trees from Z using the composition operator defined above. We now make these notions more precise.

Definition 5.4. Let X and Z be tree classes as defined above and let $k = \max\{\alpha l_Z, \lceil l_Z - 1/h_Z - l_Z \rceil\}$. Let γ be the smallest integer such that for each t with $\alpha \leqslant t \leqslant k$ there exists a tree $T \in X$ of height at most γ that has exactly t leaves (such a γ always exists). The class of Z-stratified trees (in X) is the smallest subclass of trees satisfying the following properties:

- *Each* $T \in X$ of height $\leqslant \gamma$ with $\alpha \leqslant t \leqslant k$ leaves is Z-stratified.
- If a tree T with t is Z-stratified, then so is the composition $T[T_1, T_2, \ldots, T_t]$ for any set of trees $T_1, T_2, \ldots, T_t \in Z$.

Overmars [19] showed that the class of Z-stratified trees generalizes many other search trees. He also showed that these trees can be used to efficiently store and search elements in a dynamic environment, using only $O(\log n)$ basic operations to perform inserts, deletes and queries. In the following theorem we show that dynamic algorithms which use Z-stratified trees can be transformed efficiently into parallel algorithms for the batch version of the same problem.

Theorem 5.5. *Let π be a problem in incr-*TIME$[\log^k n]$*, for some $k \geqslant 1$. Then π is in* NC^{k+2}*, if the following constraints hold:*

(1) *The data structure used by the dynamic algorithm is an augmented Z-stratified search tree. Each internal node may contain information relevant not only to enable search but may contain the result of some function f applied to its children. The only restriction is that f be computable in time $O(\log^k n)$.*

(2) *There is a special instance I_0 such that the tree corresponding to I_0 can be built in* LOGSPACE.

(3) *Any l-bit change to the current input I, where $1 \leqslant l \leqslant n$, involves modifying the tree by inserting or removing some number of elements. The elements to be inserted or deleted are functions of the current input, the current data structure and the change, and they can be computed in* LOGSPACE.

(4) *After all the insertions and deletions some sort of search is conducted in the tree in time $O(\log^k n)$ to get the answer to the updated instance.*

Proof. Given instance I we proceed as follows: First we create the tree corresponding to I_0 in LOGSPACE and calculate all the deletions and additions needed to get the

tree for I, also in LOGSPACE. Now we determine the order in which they are to be present in the final tree in NC^{k+2}. This can be done because the function is used in determining the order between two elements is computable in $O(\log^k n)$. Therefore, the set of elements in the tree for I can be sorted in $EREW^{k+1}$, and hence in NC^{k+2} [13]. We then construct the tree in time NC^{k+2} in a bottom-to-top sweep. This can be done because Z-stratified trees have logarithmic depth and the value of the function f at a node depends only on the contents of the nodes of its children. All we have to do now is to let one processor walk down the tree and get the answer to I in time $O(\log^k n)$. □

This deceptively simple theorem demonstrates why most known problems with efficient dynamic solutions have optimal parallel solutions. For example, we can use this theorem to parallelize dynamic algorithms using degree-balanced trees, height-balanced trees, path-balanced trees, etc. We can derive similar results for other classes of trees as well.

One important technique in getting dynamic solutions for problems is the divide and conquer approach. To exploit this technique Mehlhorn and Overmars [16] consider an important class of problems called order-decomposable problems.

Definition 5.6. A *set problem* is loosely defined as follows: Given a set V a *set problem* π asks some question about V. For example, the *maximum problem* asks for the largest element in a given set of numbers.

Note that in this definition π is not necessarily a decision problem. For many set problems it is possible to derive the answer over the total set V by combining the answers over two, in some ways separated, "halves" of the set. Set problems of this kind are called *order decomposable*. More formally, given a smooth nondecreasing integer function $C(n)$ we have the following definition for a *$C(n)$-order-decomposable* set problem.

Definition 5.7. A set problem π is called *$C(n)$-order-decomposable* if and only if there exists an ordering ORD (that can be used to order any input set V) and a binary function $\square$ such that the following holds: Given a set of $n \geqslant 1$ points $V = \{p_1, p_2, \ldots, p_n\}$, ordered according to ORD, for each $1 \leqslant i \leqslant n$, we have

$$\pi(\{p_1, p_2, \ldots, p_n\}) = \square(\pi(\{p_1, p_2, \ldots, p_i\}), \quad \pi(\{p_{i+1}, p_{i+2}, \ldots, p_n\})),$$

where $\square$ takes at most $C(n)$ time to compute when V contains n points. In other words a problem is order decomposable if after arranging it according to a specific ordering, the problem can be split at any point to give two smaller subproblems whose solutions can be glued together "efficiently" using the function $\square$ to get the solution for the original problem.

Mehlhorn and Overmars essentially prove that if π is an $O(\log^k n)$-order-decomposable set problem and it takes less than $O(\log^k n)$ time to compare two elements to determine their ordering with respect to ORD, then $\pi(V)$ for any set V can be maintained dynamically in time $O(\log^{k+1} n)$. We make the following rather simple extension.

Theorem 5.8. *If π is an $O(\log^k n)$-order-decomposable set problem and the ordering of two elements with respect to* ORD *takes time $O(\log^k n)$, then $\pi \in NC^{k+2}$ and a "suitable decision version of π" is in incr-*TIME$[\log^{k+1} n]$.

Proof. The parallel algorithm uses a divide and conquer approach to split the problem recursively into roughly equal parts. The solutions are later glued together using $\square$. We first sort the input according to ORD. This takes time $O(\log^{k+1} n)$ in a EREW PRAM and is therefore in NC^{k+2}, then we solve the problem for n singleton sets. We now use a bottom-up approach to glue the solutions back together. This takes $\log n$ phases and hence can be done in $O(\log^{k+1} n)$ by an EREW PRAM, which in turn implies that it can be done in NC^{k+2}. $\square$

This theorem gives automatically generated parallel algorithms for problems such as

- calculating the convex hull of a set of points,
- finding the maximal element in two dimension, and
- finding the union and intersections of a set of line segments.

For these three problems we get automatically generated parallel algorithms that place them in the class NC^3. Note that the automatically generated parallel algorithms are not as efficient as the special purpose algorithms for the same problems (in terms of processor utilization or parallel time) although they are within a logarithmic factor of the optimum. However, the aim of Theorem 5.8 is to show that such automatic transformations are possible and not to attempt the construction of optimal parallel algorithms for these problems.

6. Problems on comparator gates

In this section we consider the circuit-value and network-stability problems over comparator gates, introduced in [15]. We show that these two problems are in *incr*-LOGSPACE and in *rincr*-LOGSPACE(NLOGSPACE), respectively and thus their dynamic versions can be solved quickly in parallel, even though the batch versions have no known NC solutions.

The definitions of circuits and gates are taken from [15] and are given here for the sake of completeness.

Definition 6.1. A λ-input, μ-output *gate* is defined to a function g from the domain $\{0,1\}^\lambda$ to the range $\{0,1\}^\mu$. Thus, g maps a λ-bit word to a μ-bit word.

A *network* is a finite labeled directed graph. Source (in-degree zero) nodes of the directed graph have out-degree one and are called output nodes; sink (out-degree zero) nodes have in-degree one and are called output nodes. Each internal node is labeled with a gate and an ordering of its predecessors and successors. If an internal node has in-degree λ and out-degree μ, its gate has λ inputs and μ outputs. If the underlying directed graph of the network is acyclic, the network is a *circuit*.

In the preceding definitions of networks and circuits outputs of gates are not allowed to be explicitly duplicated. We are only allowed to use the μ outputs that a gate produces as inputs to the other gates.

A gate is said to preserve adjacency (is *adjacency preserving*) if it maps adjacent input words (binary words of the same length that differ in at most one bit) into adjacent output words. A gate is *monotone* if it cannot simulate the NOT gate. A circuit is adjacency preserving if it maps adjacent input words to adjacent output words.

A *comparator* gate is a 2-input, 2-output gate that takes as input a and b and gives as output $a \wedge b$ and $a \vee b$. The circuit-value problem over comparator gates (C-CV) is the circuit-value problem where all the gates are comparator gates.

Theorem 6.2. *The circuit-value problem over comparator gates* (C-CV) *is in incr-*LOGSPACE.

Proof. Compartor circuits are adjacency preserving; a one-bit change at any input bit only propagates one-bit changes to successive gates. Thus when an input bit of gate g changes exactly one of its outputs changes, which in turn changes an output of the gate g' which takes as input the changed output of g. Therefore, to propagate a single bit change we have to follow a path in the circuit changing the input and output values of the gates along the path.

- *Preprocessing:* The circuit is evaluated in polynomial time for the initial instance and the values at each edge are maintained.
- *Change in one input bit*: This change is propagated in LOGSPACE, by walking through the circuit. If one of the input values of a gate g changes then exactly one of its output values changes. Let us suppose output o_1 of g changes due to a change in one of its inputs. Since o_1 is fed as an input to exactly one gate g' this results in a change in the input of exactly one gate g' one of whose outputs therefore changes. Thus to propagate the input change we have to follow a path in the circuit changing one input and one output value at each gate on the path.
- *Introduction of a gate*: Let i_1 and i_2 be inputs to gates g_1 and g_2, such that the new gate g to be introduced takes as input i_1 and i_2 and feeds its outputs o_1 and o_2 to g_1 and g_2, respectively. To effect this change, we just walk down the circuit twice, starting at gates g_1 and g_2, taking into account the new inputs to these gates caused by the introduction of the new gate.
- *Deletion of a gate*: The process here is similar to insertion and we again walk down the circuit twice from the point where the deletion took place, updating edge values in the process.

- *Other changes*: Other changes like changing the connections between different gates in the circuit can be modeled as a constant number of add and delete operations.

All these steps can be done in logspace. Therefore, C-CV is in *incr*-LOGSPACE. Note that we have considered addition and deletion of gates in the circuit even though the input changes are restricted to changing zeros to ones and vice versa. Additions and deletion of gates can be achieved by changing the input-bits in a suitable coding of the input. For instance the initial input could be padded with zeros which could be changed at various points to introduce new gates. □

A similar argument holds for any circuit that is adjacency preserving. This result is interesting because C-CV is not known to be in NC. Therefore, C-CV has the distinction of being a problem that can be updated fast in parallel, though it is not known whether it can be solved fast from scratch in parallel. This also means that problems like the "lex-first maximal matching", problem which can be parsimoniously reduced to C-CV are in *incr*-LOGSPACE.

We now turn our attention to the network stability problem on comparator gates. A network is stable for a given input assignment if we can assign values to the edges which are consistent with the gate equations and the given input assignment. Given a network N consisting of only monotone gates and an input assignment S_{in}, it can be easily seen that there exists a stable configuration. Therefore, the answer to the network-stability problem on comparator gates is always a yes. The interesting question therefore, is the value of an edge in particular types of stable configurations. Given a network N and an input assignment S_{in} we are interested in knowing the value of an edge in the "most-zero" stable configuration S_{min}. This configuration has the property that if an edge e is 0 in any stable configuration S of the network, it is 0 in S_{min}.

Theorem 6.3. *The problem of finding whether an edge e is* 0 *in the "most-zero" configuration of a comparator network N is in rincer*-LOGSPACE(NLOGSPACE).

The basic idea behind the dynamic algorithm is the same as before. Since at each comparator gate if one input changes exactly one of the outputs changes, therefore when an input is changed we need to traverse a path in the network changing output values as we proceed. However, since we are now dealing with a network this "path of change" may not be a simple path. Furthermore, since we are only allowed to record changes to a write-only memory (we are only allowed logarithmic work space) we cannot re-read the changes that we have already made. We therefore have to resort to nondeterminism to recognize and traverse cycles so that we do not keep going around the same cycle over and over again. A complete proof is provided in the Appendix.

Subramanian [24] has shown that a number of problems concerning stable marriage are equivalent to the problem of network stability in comparator gates. Decision problems based on the "man-optimal stable marriage problem" and other problems

parsimoniously reducible to the comparator network stability problem are therefore in *rincr*-LOGSPACE(NLOGSPACE). Like the comparator circuit-value problem, these too have no known NC algorithms to solve them from scratch.

7. Conclusions and open problems

We have provided a firm theoretical base to conduct the study of incremental computation. We have demonstrated the existence of problems that are incrementally complete for various natural complexity classes and shown some important special cases wherein dynamic solutions imply parallel ones. We have also shown that the comparator circuit-value problem is in *incr*-LOGSPACE and the comparator network-stability problem is in *rincr*-LOGSPACE(NLOGSPACE). We would like canonical problems for the classes we have defined, and would like to answer some of the following questions.

(1) How is *incr*-POLYLOGTIME related to the class LOGSPACE?

- Is there some restriction to LOGSPACE that will automatically give us incremental algorithms for languages in this restricted class? The problem here seems to be in defining a meaningful restriction of LOGSPACE for which directed forest accessibility is *incr*-POLYLOGTIME-complete.
- Is there some restriction of *incr*-POLYLOGTIME which is a subset of LOGSPACE?

(2) What is the relation between *incr*-POLYLOGTIME and NC, the class of problems solvable by bounded fan-in circuits of polynomial size and polylog depth?

- Are restrictions of either class comparable to the other?

(3) What is the relation between *incr*-POLYLOGTIME and problems which have optimal parallel algorithms?

- Are there general reductions from one to the other? Do problems like *st*-numbering that have optimal parallel algorithms also have incremental algorithms.
- It seems hard to convert any parallel algorithm that uses reachability in some form into a corresponding dynamic algorithm for the same problem. We feel that restricting ourselves to parallel algorithms which avoid using transitive closure as a subroutine may help us in getting dynamic algorithms. This is the case with a number of recent parallel algorithms for planar graphs. It would be interesting to see if any generic reductions are possible in this restricted realm.

(4) A particularly interesting class of problems in computational geometry deals with the manipulation of generalized trees. In this restricted class there seems to be an even stronger relation between *incr*-POLYLOGTIME and NC, as is demonstrated by Theorems 5.5 and 5.8. We would like to explore this relation further.

(5) How are *incr*-POLYLOGTIME, *incr*-LOGSPACE and *rincr*-LOGSPACE(NLOGSPACE) related to the class *CC* of problems reducible to C-CV? Does the fact that comparator gates preserve adjacency make these problems incrementally tractable? We have partially solved this problem by showing that the circuit-value

problem on comparator gates is in *incr*-LOGSPACE and the network-stability problem is in *rincr*-LOGSPACE(NLOGSPACE). However we are unable to show that *CC* is in *rincr*-LOGSPACE(NLOGSPACE); this is because the many-one reduction used by Subramanian [24] may not be parsimonious. In particular we don't know whether the network-stability problem on nonbipartitionable X-gates is in *rincr*-LOGSPACE(NLOGSPACE).

We believe that the classes *incr*-TIME and *incr*-SPACE are an important means for getting a better understanding of the relationship between incremental and parallel computation.

Appendix

In this section we provide a detailed proof for Theorem 6.3. We reproduce the statement of the theorem for convenience.

Theorem 6.3. *The problem of finding whether an edge e is* 0 *in the "most-zero" configuration of a comparator network N is in rincr*-LOGSPACE(NLOGSPACE).

Proof.

- *Preprocessing*: The most-zero configuration of the network is evaluated in polynomial time for the initial instance and the values at each edge are maintained. The initial edge values are evaluated by *forcing* the inputs (determining the effect of the input on the network) one by one in some predetermined order until no more edges can be forced. For any input i the edges are forced form a path (not necessarily simple) from i to some output o. We will call this path the *snake* i and label every edge in the snake with i. The snake i is thus a sequence of edges which get the value of the ith input whose effect on the network is determined after the first $i-1$ inputs have been forced.

  Let π be the order in which the inputs are forced. We say that snake i is less than snake j, if input i was forced before input j. We call i a zero-snake (a one-snake) if the input i is zero (one). All the edges which are not forced by any inputs are given the value 0 and are given a special cycle label C. C is placed at the end of the ordering π. It can be shown that C is a collection of zero-cycles.
- *Update*: We now show how to maintain these values and the labels of snakes when an input is changed; the other operations of introducing or removing gates can be similarly handled. We consider just the change from 0 to 1; the other case is similar. The idea is to start at the changed input i and follow a path in the network changing the values of the edges along the path from 0 to 1. The path we follow may traverse the edges of many different snakes, we will therefore maintain a variable *Fl* which will tell us which snake we are currently following. During the course of our algorithm at times it would be convenient to change our perception of the input that has changed. We will therefore maintain a variable *Sn* which is our current

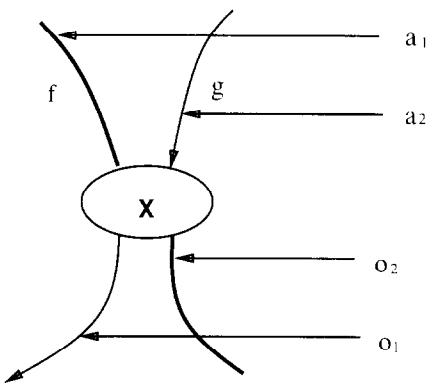

Fig. 1. When two zero snakes meet.

perception of the snake whose input has changed from zero to one. Initially both *Sn* and *Fl* have the value *i* (the changed input).

We walk down the circuit following label *Fl* changing edge values and replacing edge labels with *Sn*. All the changes are recorded in a write only copy of the network which in the beginning of the update corresponds to the current state of the network. Any changes made cannot be read again. This restriction is to constrain the algorithm to work in LOGSPACE. At each step we decide whether to continue along the same path, or to do certain intermediate computations. Let X be the current gate along the path, with input edges a_1 and a_2 and output edges o_1 and o_2. Let o_1 be the AND output and o_2 the OR output. Suppose that a_1 was changed from 0 to 1 in the last step then we make the decision on what to do next based on the following criteria.

(1) – a_1 is on the snake f and a_2 is on snake g.
  – Both f and g are zero-snakes.
  – f is less than g in the ordering.

We are now in a situation where o_1 was on f and o_2 was on g (Fig. 1). We therefore need to do some label changing since the OR output should be on snake f (this is so because f is less than g in the ordering and hence should force the OR output). Snake f should therefore get label g beyond X (or, put another way, the snakes are changing pieces). However we cannot just change the label of f to g, all the zero-snakes in between f and g will be affected as well. We do this by the subroutine ZERO-LABEL(f, g) (to be described later), which relabels edges in all zero-snakes $f \leqslant h < g$ with labels greater than h. It is clear that these will be the only snakes that will be affected. We now have to follow g instead of f. We therefore change the value of *Fl* to be g, assign the label *Sn* to the edge o_2, change its value from 0 to 1, and continue.

(2) – a_1 is on the snake f and a_2 is on snake g.
  – Both f and g are zero-snakes.
  – f is greater than g in the ordering.

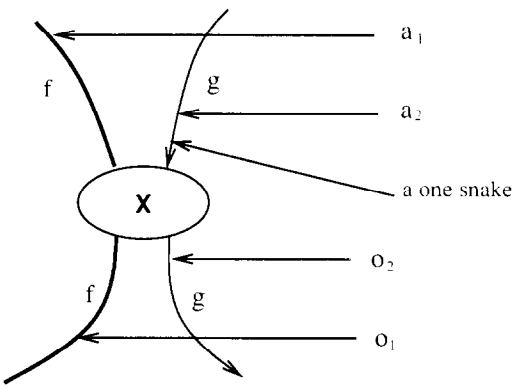

Fig. 2. When a zero-snake meets with a one-snake.

We are now in a situation where the AND output o_1 was on snake g and the OR output o_2 on snake f (Fig. 1), o_2 should now become 1, we therefore make no changes to either *Sn* or *Fl*. We change the edge o_2 to have label *Sn* and value 1 and continue forcing (changing labels and values as warranted) the input change in the network.

(3) – a_1 is on the snake f (or on C) and a_2 is on snake g.
– f is a zero-snake while g is a one-snake.
– *Sn* is less than g in the ordering.

We are in a situation similar to case 1 (Fig. 2). The OR output is on snake g (since g is a one-snake), but now that $f(C)$ has become part of the one-snake *Sn* the OR output should be on *Sn*. The edges on snake g therefore have to get the label *Sn* (and maybe others in between *Sn* and g). We now use a similar routine ONE-LABEL (*Sn*, g) to (this routine is symmetric to the ZERO-LABEL (f, g) subroutine) relabel the edges of all one-snakes $Sn < h \leqslant g$. A careful look at Fig. 2 shows that in the subsequent steps the network labellings have to change as if snake g had been changed from zero to one. For instance the o_1 output of the gate X must now get the value 1 and the label g. Therefore we assign the value g to *Sn*; change the edge o_1 to have value 1 and label *Sn* and continue (note: assigning *Sn* the value g is equivalent to changing our perception of the input that changed to g).

(4) – a_1 is on the snake f (or on C) and a_2 is on snake g.
– f is a zero-snake while g is a one-snake.
– *Sn* is greater than g in the ordering.

We are now in a situation similar to case 2 (Fig. 2); since *Sn* is greater than g the OR output o_2 should be on g. Now that a_1 changes to 1, so should o_1. *Sn* and *Fl* remain the same, and we continue forcing the input change in the network.

(5) – a_1 is on the snake f and a_2 is on C.
– f is a zero-snake.

We are now in a situation wherein one or more zero-cycles of C will become part of the new one-snake *Sn* (Fig. 3). We perform the following steps:

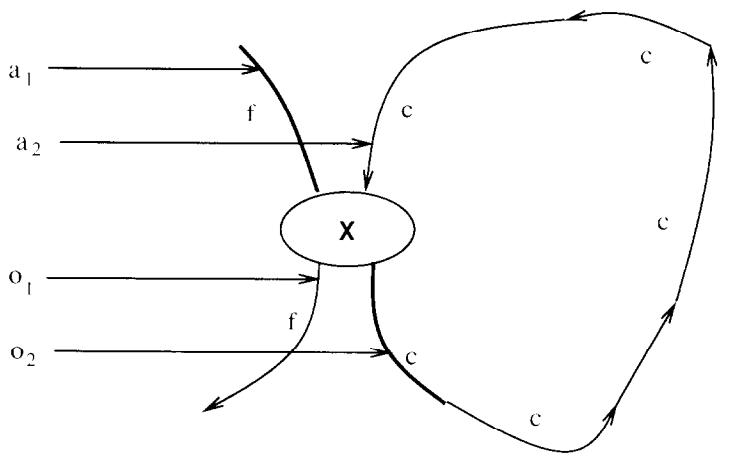

Fig. 3. When the forcing path meets a zero-cycle.

- Find out if the edges of this cycle C_1 of C have already had their values changed. This is done by a call to an oracle FORCED(a_1, a_2, X) which looks to see whether the edge a_2 is forced to 1 after a_1. The oracle FORCED works in NLOGSPACE and will be described later.
- If the cycle has already been forced we do not enter it but force the AND output o_1 to 1 instead and change its label to Sn before continuing.
- If C_1 has not been entered before, we mark X, and enter the cycle, by making use of a temporary follow TFl which is set to C.
- We continue with the forcing using the criteria for cases 2 and 4 until we get back to X. During this process we may enter into other zero-cycles, but we can argue inductively that we will always come back since the follow TFl is always C (this is so because cases 1 and 3 never apply since C is greater than all the snakes in the ordering).
- The only thing left to show is how to recognize that we have indeed come back. Suppose we arrive at a node Y that has input edges a_1 and a_2 and output edges o_1 and o_2 all of which have the label C. Further, suppose we arrive at the input edge a_2. We now need to determine whether we should take the AND output o_1 or to take the OR output o_2, we can determine that by calling the oracle FORCED(a_1, a_2, Y) and taking OR if it answers "no" and the AND output if it answers "yes."
- When we reach X we continue as before.

(6) – $a_1 a_2 o_1$ and o_2 belong to the same zero-snake f.

We have now encountered a loop in the snake f (Fig. 4). Since a_1 has changed in value to 1, we should now take the OR output o_2. The edges in the loop formed with the AND output should therefore get other labels. As before we need to see if the loop has been encountered; we therefore call FORCED(a_1, a_2, X) to determine if a_2 is forced after a_1. If the answer is "no" we change the label and value of o_2 and continue,

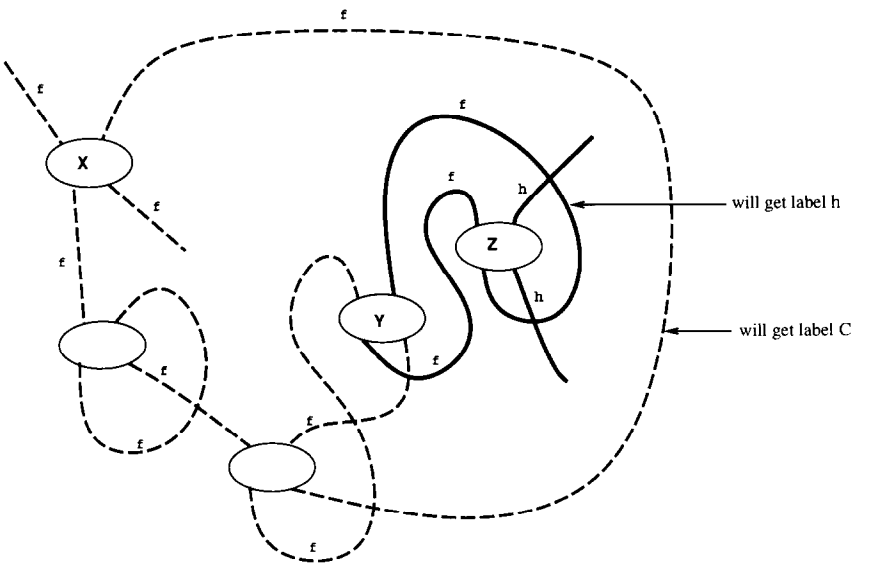

Fig. 4. A self-loop that breaks up.

otherwise we perform the following steps to change the labels of the loop L consisting of all the edges starting from o_1 to a_2 on the snake f.

- Change the labels of the edges in L to C (in the output tape).
- For all zero-snakes $h > f$ (in the reverse order) find the earliest node Y in L such that we can reach Y with a label h. Due to the change in snake f the default labeling for the edges in the loop L is C, however, if we can reach some node Y in L with some label h (where h is a zero-snake), then some of the edges in L beyond Y would get the label h (since C is greater than all the snakes in the ordering π). We therefore find the earliest such Y by calling the routine REACHED(Y, h) (to be described later) for all the Y's one by one. Some of the ways a label h can reach the node Y are shown in Figs. 4 and 5. In Fig. 4 we see the case wherein a subcycle of L gets the label h, and in Fig. 5 we see how to reach Y with a label h after a complicated interchange of labels. In this case all of L gets the label h.
- Label the edges in the smallest subcycle of L containing Y with label h.

The loop L has now been divided amongst the other snakes and C, note since we did all the writes in the reverse order as in π we are guaranteed that the various orderings will be obeyed at all nodes in the loop.

We now describe the subroutines FORCED, REACHED and ZERO-LABEL. The procedure ONE-LABEL is similar to the ZERO-LABEL procedure and is omitted. The procedure FORCED (a_1, a_2, Y) works as follows:

(1) Start at the input which has been changed and follow the path of forcing until a node with a zero-cycle is reached. At this point we know that if we take the OR

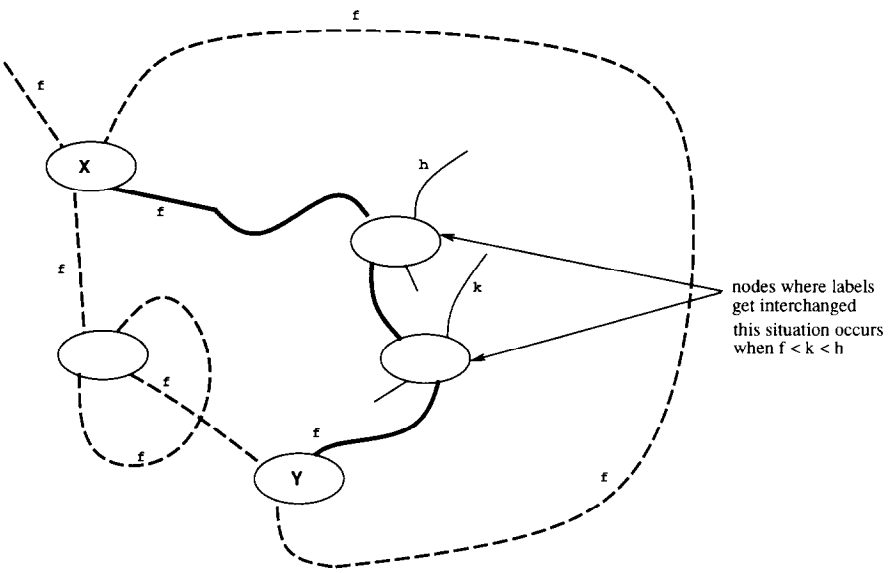

Fig. 5. How label interchanges affect the new labeling.

output we will come back; therefore we have a nondeterministic choice to make as to whether we should take the OR or the AND output to reach the edge a_2 of gate Y.

(2) If we reach a_1 first and then a_2 within n steps we answer "yes" otherwise "no."

The procedure REACHED(Y, h) works as follows:

(1) This procedure assumes that h is a zero-snake. We start at the input which has been changed and follow the path of forcing until we encounter a zero-cycle or a zero-snake $g \geqslant h$ that is higher than or equal to f (the current Fl) in the ordering. In the former case we make a nondeterministic choice and follow one of the two outputs, in the later case a nondeterministic choice is made to either continue with the forcing (which means following snake g) entering the labeling phase.

(2) At the beginning of the labeling phase the follow fl is set to f and the label l is set to g if $g \neq f$ and is set to C otherwise. We trace the snake fl till we meet a zero-snake g_1 such that $h \leqslant g_1 < l$. We now use nondeterminism again to decide which snake to follow, if we follow fl we change our current label l to g_1, otherwise we change fl to be equal to g_1.

(3) If we reach Y with label h in $2n$ steps we answer "yes," otherwise we answer "no".

The procedure ZERO-LABEL(f, g) works as follows:

(1) For all zero-snakes h, that $f \leqslant h < g$ consider all the zero-snakes $k \neq h$ and g in the reverse order, and do the following:

(2) Find the earliest node Y on h with inputs a_1, a_2 and outputs o_1, o_2 such that the AND output o_1 is on snake h, and Y can be reached with label k. This is done using the subroutine REACHED.

(3) If such a Y exists relabel all the edges in h beyond Y with the label k.

An inductive proof can be used to show that ZERO-LABEL(f, g) labels the snakes $f \leqslant h < g$ correctly. Note that all the changes are made on the write-only memory.

The proof of correctness of these procedures and the algorithm as a whole depends upon the following crucial facts.

- While following snake f if we reach a node X such that a_1 is on the snake f and a_2 is on snake g, both f and g are zero-snakes, and f is less than g in the ordering, then g has not been changed in any previous labeling.
- While following snake f if we reach a node X such that a_1 is on the snake f and a_2 is on snake g, f is a zero-snake while g is a one-snake, and the value of Sn is less than g in the ordering, then g has not been changed in any previous labeling.

The proof follows from the fact that after every zero-labeling the value of Fl becomes greater than all the snakes whose edges have changed and after every one-labeling the value of Sn becomes greater than all the snakes whose edges have changed. This implies that even though we are allowed to look only at the input tape, we have sufficient information to make correct decisions. In all our calls to NLOGSPACE oracles we have taken the liberty of assuming that the oracle gives both "yes" and "no" answers accurately, which is not entirely correct. Every call to any of the subroutines should be interpreted as a dual call to the routine and its complement (executed in an interleaved fashion). When one of them answers we stop the other routine, the whole computation is still in NLOGSPACE because NLOGSPACE and CO-NLOGSPACE are the same [12]. Since the algorithm above works in LOGSPACE and uses NLOGSPACE oracles, the network-stability problem is in *rincr*-LOGSPACE(NLOGSPACE).

Acknowledgments

We would like to thank the referees for pointing out the error in an earlier version of the proof of Theorem 3.5 and for a number of other useful suggestions.

References

[1] D. Barrington, Bounded width polynomial size programs recognize exactly those languages in NC^1, in: *Proc. 18th Symp. on Theory of Computing* (1986) 1–5.

[2] A.M. Berman, M.C. Paull and B.G. Ryder, Proving relative lower bounds for incremental algorithms, *Acta Inform.* **27** (1990) 665–683.

[3] R.B. Boppana and M. Sipser, The complexity of finite functions, in: *Handbook of Theoretical Science* A (1990) 757–804.

[4] A. Cook and P. Mckenzie, Problems complete for deterministic logarithmic space, *J. Algorithms* **8** (1987) 385–394.

[5] A. Delcher and S. Kasif, Complexity issues in parallel logic programming, Ph.D. Thesis, Johns Hopkins Univ., 1989.

[6] S. Even and H. Gazit, Updating distances in dynamic graphs, *Methods Oper. Res.* **49** (1985) 371–387.

[7] M.L. Fredman, Observations on the complexity of generating quasi-gray codes, *SIAM J. Comput.* **7** (1978) 134–146.
[8] M.L. Fredman, The complexity of maintaining an array and computing its partial sums, *J. Assoc. Comput. Mach.* **29** (1982) 250–260.
[9] M.L. Fredman and M.E. Saks, The cell probe complexity of dynamic data structures, in: *Proc. 21st ACM Symp. on Theory of Computing* (1989) 345–354.
[10] R. Greenlaw, H.J. Hoover and W.L. Ruzzo, A compendium of problems complete for P, Tech. Report TR-91-05-01, Dept. of Computer Science and Engineering, Univ. of Washington, 1991.
[11] L.J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, in: *Proc. 19th IEEE Symp. on Foundations of Computer Science* (1978) 8–21.
[12] N. Immerman, Nondeterministic Logspace is closed under complement, Tech. Report No YALEEU/DCS/TR 552, Yale Univ., 1987.
[13] R.M. Karp and V. Ramachandran, A survey of parallel algorithms for shared memory machines, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A* (Elsevier, Amsterdam, 1992) 871–941.
[14] R.E. Ladner, The circuit value problem is Log space complete for P, *SIGACT News* **7** (1975) 18–20.
[15] E. Mayr and A. Subramanian, The complexity of circuit value and network stability, in: *Proc. 4th Ann. Conf. on Structure in Complexity Theory* (1989) 114–123.
[16] K. Mehlhorn and M. Overmars, Optimal dynamization of decomposable searching problems, *Inform. Process. Lett.* **12** (1981) 93–98.
[17] P.B. Miltersen, On-line reevaluation of functions, Tech. Report ALCOM-91-63, Computer Science Dept., Aarhus Univ., Aarhus DK., May 1991.
[18] S. Miyano, S. Shiraishi and T. Shoudai, A list of P-complete problems, Research Report, Research Institute of Fundamental Information Science, Kyushu, Japan, 1989.
[19] M. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Vol. 156 (Springer, Berlin, 1983).
[20] M.C. Paull, C.C. Chang and A.M. Berman, Exploring the structure of incremental algorithms, Tech. Report DCS-TR-139, Dept. of Computer Science, Rutgers Univ., May 1984.
[21] J.H. Reif, A topological approach to dynamic graph connectivity, *Inform. Process. Lett.* **25** (1987) 65–70.
[22] S. Skyum and L.G. Valiant, A complexity theory based on boolean algebra, in: *Proc. 22nd IEEE Symp. on Foundations of Computer Science* (1981) 244–253.
[23] D.D. Sleator and R.E. Tarjan, A data structure for dynamic trees, *J. Comput. Systems Sci.* **24** (1983) 362–381.
[24] A. Subramanian, A new approach to stable matching problems, Tech. Report STAN-CS-89-1275, Dept. of Computer Science, Stanford Univ., 1989.