BRICS RS-97-32 Husfeldt & Rauhe: Extensions of the Chronogram Method

BRICS

Basic Research in Computer Science

Hardness Results for Dynamic Problems by Extensions of Fredman and Saks' Chronogram Method

Thore Husfeldt
Theis Rauhe

BRICS Report Series **RS-97-32**
ISSN 0909-0878 **November 1997**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/97/32/`

HARDNESS RESULTS FOR DYNAMIC PROBLEMS BY EXTENSIONS OF FREDMAN AND SAKS' CHRONOGRAM METHOD

THORE HUSFELDT AND THEIS RAUHE

BRICS†, University of Aarhus

November 1997

Abstract We introduce new models for dynamic computation based on the cell probe model of Fredman and Yao. We give these models access to nondeterministic queries or the right answer ± 1 as an oracle. We prove that for the dynamic partial sum problem, these new powers do not help, the problem retains its lower bound of $\Omega(\log n/\log\log n)$.

From these results we easily derive a large number of lower bounds of order $\Omega(\log n/\log\log n)$ for conventional dynamic models like the random access machine. We prove lower bounds for dynamic algorithms for reachability in directed graphs, planarity testing, planar point location, incremental parsing, fundamental data structure problems like maintaining the majority of the prefixes of a string of bits and range queries. We characterise the complexity of maintaining the value of any symmetric function on the prefixes of a bit string.

---

† BRICS, Basic Research in Computer Science, Centre of the Danish National Research Foundation. Dept. of Computer Science, University of Aarhus, Ny Munkegade, 8000 Aarhus C, Denmark.

Supported by the ESPRIT Long Term Research Programme of the EU, project number 20244 (ALCOM-IT).

1. Introduction

Update versus query time. For dynamic problems, two trivial solutions are immediate: Either the algorithm spends time after each update reorganising the data structure to anticipate every future query, or the algorithm spends time after each query to read the entire history of updates. However, a crucial property of many hard problems is that these two cannot be optimised simultaneously. This tradeoff between *update time* and *query time* was studied using the *chronogram method* by Fredman and Saks [13], a result that has proved extremely useful for lower bounds for dynamic algorithm and data structures.

The method of [13] is an information-theoretic argument formalising the idea that not all relevant information about the updates can be passed on to a typical query. The present paper takes a closer look at this information, asking what kind of information is responsible for the hardness of the problem. Our approach is to provide the query algorithm with well-defined aspects of the information for free, e.g., we consider nondeterministic query algorithms.

Example: Range queries. We can illustrate our approach using range query problems. The object is to maintain a set $S \subseteq \{1, \ldots, n\}^2$ of points in the plane, the updates insert and remove points from S. An *existential* range query asks whether a given rectangle R contains a point from S. This problem requires time $\Omega(\log\log n/\log\log\log n)$ [4, 19].

With nondeterministic queries, this problem becomes trivial: guess a point and verify that it is in $S \cap R$. In other words, the sole reason for the hardness of this problem lies in maintaining precisely the kind of information that nondeterminism provides for free. However, this is not true for all problems; our main result implies that reporting the *parity* of $|R \cap S|$ remains just as hard as without nondeterminism, so the hardness of this problem hinges on information of a fundamentally different kind.

Main contribution. We state our two main results in terms of the *signed partial sum problem.* The problem is to maintain a string $x \in \{-1, 0, +1\}^n$ under updates that change the letters of x and queries of the form

query(i): return $x_1 + \cdots + x_i \mod 2$.

We prove two theorems about this problem. Theorem 1 shows that even in models with nondeterministic queries, the partial sum problem requires time $\Omega(\log n/\log\log)$ per operation with logarithmic cell size. It is known that this is also the deterministic complexity of the problem [7, 13], so nondeterminism does not help.

Our second main result studies the same problem in a *promise* setting, where the query algorithm receives almost the correct answer for free. The updates are as before, and the query is

parity(i, s): return $x_1 + \cdots + x_i \mod 2$ provided that $|s - \sum_{j=1}^{i} x_j| \leq 1$ (otherwise the behaviour of the query algorithm is undefined).

Theorem 2 shows that this problem still requires $\Omega(\log n/\log\log n)$ per operation.

We reason within the cell probe model of Fredman [10] and Yao [27], with some extensions to cope with our stronger modes of computation. This can be viewed as a nonuniform version of the random access computer with arbitrary register instructions. Especially, our lower bounds are valid on random access machines with unit-cost instructions on logarithmic cell size. The success of this model is partly due to the validity of these bound in light of schemes like hashing, indirect addressing, bucketing, pointer manipulation, or recent algorithms that exploit the parallelism inherent in unit-cost instructions. For these reasons the cell probe model has arguably become the model of choice for lower bounds for dynamic computation.

Theorems 1 and 2 are proved by extending the chronogram method, which was introduced by Fredman and Saks [13] and got its name in [5].

Lower bounds for dynamic algorithms. Our results suggest a new general technique for proving lower bounds for dynamic algorithm and data structure problems. Because Thms. 1 and 2 hold in very strong models of computation, we can exploit these strengths in our reductions—this yields simple proofs. We support our claims about the versatility of this technique by exhibiting a number of new lower bounds for well-studied problems.

For example, we prove optimality or near-optimality of a number of dynamic algorithms in the literature from various fields like dynamic string and graph algorithms and computational geometry. These include planar point location in monotone subdivisions [3, 23], reachability in upward planar digraphs [25], and incremental parsing of balanced parentheses [9]. We show that all these problems require time $\Omega(\log n/\log\log n)$ per operation.

It is known [8, 12, 15, 20] that this is also a lower bound for reachability in grid graphs. However, grid graphs of constant width allow a reachability algorithm in time $O(\log\log n)$ per operation [2], an exponential improvement. Our technique is sufficiently versatile to prove a lower bound that is parameterised by the width w of the graph: Prop. 6 states that dynamic reachability for grid graphs of width $w = O(\log n/\log\log n)$ requires time $\Omega(w)$ per operation, bridging the gap between the two results.

Partial sum problems. The partial sum problem was introduced by Fredman as a 'toy problem which is both tractable and surprisingly interesting' [11] and has been the focal point of many investigations of dynamic complexity in a variety of models [28, 13]. The problem is to maintain a string x of n bits under updates that change the bits of x and queries for the prefix sums of x. It was shown in [13] that the parity query

parity(i): return $x_1 + \cdots + x_i \mod 2$,

requires time $\Omega(\log n/\log\log n)$, so even the least significant bit is hard to maintain.We turn to two other natural and potentially easier variants, where the query operations are

majority(i): return 1 iff $x_1 + \cdots + x_i \geq \lceil \frac{1}{2}n \rceil$,
equality(i): return 1 iff $x_1 + \cdots + x_i = \lceil \frac{1}{2}n \rceil$.

These problems arise in many data structures, e.g. when following paths towards heavy subtrees in balanced search trees. We can also dress up these problems as database queries like 'did as many male as female guests arrive before noon?' or 'are more French than English talks scheduled between Tuesday and Friday?' Similarly, these problems can be viewed as natural *range query* problems in Computational Geometry.

No nontrivial lower bounds for these two problems follow from [13]. The results from [4, 18, 19, 26] can be seen to imply $\Omega(\log\log n/\log\log\log n)$ lower bounds using an entirely different technique based on Ajtai's result [1]; and [16] reports $\Omega((\log n/\log\log n)^{1/2})$ for equality and $\Omega(\log n/(\log\log n)^2)$ for the majority.

Proposition 7 of the present paper shows that both problems require time $\Omega(\log n/\log\log n)$ per update, just as parity. Again, the proof is a simple application of our main results. We then extend our analysis of the majority problem to the class of *threshold* functions, and characterise the complexity of the resulting partial sum problem in terms of the size of the threshold in Prop. 8. We can generalise this even further, to the entire class of symmetric functions in Prop. 9. Intriguingly, the resulting bounds closely resemble the corresponding results from Boolean circuit complexity, where these problems have been studied intensively, hinting a connection between the dynamic and parallel realms.

Limitations of the chronogram method. A large number of hardness results for dynamic problems employ the chronogram method, usually by constructing a reduction from a partial sum problem. Our results imply in some precise sense that this method is unable to distinguish deterministic from nondeterministic computation. In particular, this method cannot prove lower bounds for a problem that are better than the best nondeterministic algorithm. This is an important guide in the search for lower bounds for a large class of problems, including for example existential range searching and convex hull.

Outline of paper. Section 2 introduces dynamic algorithms with nondeterministic queries and contains the statement of Thm. 1; the proof of this result, which is the main technical contribution of this paper, takes up Sect. 3. Our lower bounds for dynamic algorithms and partial sum problems are presented in Sect. 4. Finally, Sect. 5 introduces the notion of refinement and presents Thm. 2 and Prop. 8.

2. Nondeterminism in Dynamic Algorithms

2.1. Nondeterministic query algorithms. We now introduce our notion of nondeterministic query algorithms for dynamic decision problems. We allow query algorithms to nondeterministically load a value into a memory cell. The semantics is as usual: The value returned by a nondeterministic query is 1 unless all nondeterministic choices return 0. For example, the following program solves the existential range query problem from the introduction, storing all points from S in a two-dimensional array M:

update(i,j):	*query*(R):
$M[i,j] := \neg M[i,j]$	**guess** $(i,j) \in R$
	return $M[i,j]$

We should mention that we have not defined the *side-effects* of a nondeterministic query algorithm, i.e., the effect of its assignments to memory. This can be done in a number of ways; for example we might say that if there are computations (i.e., sequences of nondeterministic choices) that result in '1', the algorithm will execute one of these computations; otherwise it will execute a computation leading to '0'. We mention that our lower bound is immune to precisely how these effects are defined, since the hard operation sequence constructed in the proof needs only a single query, which happens at the very end.

Nondeterministic queries are a powerful tool for a number of well-studied problems. A good example from Computational Geometry is *dynamic convex hull*, the problem of maintaining the convex hull of a set of points S, where points are inserted and removed. The query operation asks whether the query point q lies inside or outside the convex hull of S. Again, we can solve this problem with a trivial update algorithm that simply stores S in a large table (in the cell probe model we do not worry about memory space, otherwise we can use standard dictionaries). The nondeterministic query guesses three points from S and verifies that the query point lies in the triangle spanned by these points—a well known result in plane geometry asserts that this is necessary and sufficient.

In general, a problem is amenable to nondeterminism, if the outcome of each query depends on only a bounded number of updates. Contrast this with the problems identified in [13], where each update affects only a bounded number of queries, e.g., dictionary problems.

2.2. Signed partial sum. The *signed partial sum* problem is to maintain a string $x \in \{-1, 0, +1\}^n$, initially 0^n, under updates that change the letters of x and queries about the parity of the prefix sums of x

update(i, a): change x_i to $a \in \{-1, 0, +1\}$,
query(i): return $x_1 + \cdots + x_i \mod 2$.

The data structure of Dietz [7] solves this problem, deterministically, in time $\mathrm{O}(\log n / \log\log n)$ per operation with logarithmic cell size. The next theorem states that nondeterministic queries can do no better. We state theorem as a trade-off between update and query time.

Theorem 1. *Every nondeterministic algorithm for the signed partial sum problem with cell size* $\log n$*, update time* t_{u}*, and query time* t_{q} *must satisfy*

$$t_{\mathrm{q}} = \Omega\Big(\frac{\log n}{\log(bt_{\mathrm{u}}\log n)}\Big). \tag{1}$$

The lower bound holds even if the algorithm requires

$$0 \le x_1 + \cdots + x_i \le \Big\lceil \frac{\log n}{\log(bt_{\mathrm{u}}\log n)} \Big\rceil \tag{2}$$

for all i *after each update.*

The balancing condition (2) continues previous work [16] on extending the chronogram method, which is implicit in the constructions in the present paper. In Sect. 4.2 we state a further generalisation of Thm. 1, relating the terms in (1) and (2).

3. Proof of Theorem 1

The proof of Theorem 1 follows the same line as previous applications of the chronogram method [13, 5], but several concepts are changed. We give a self-contained presentation for completeness.

We consider a specific sequence of operations that consists of a number of updates followed by a single query. The update sequence is chosen at random from a set U defined in Sect. 3.5.

3.1. Model of computation. The computational model is an extension of the cell-probe model [10, 27]; since there is only a single query, which happens at the very end of the sequence, we can model query algorithms by nondeterministic decision trees.

More precisely, a *cell probe* algorithm consists of a family of trees, one for each operation, and a memory $M \in \{0, \dots, 2^b - 1\}^*$. We refer to the elements of M as *cells*, each of which can store a b-bit number. To each update we associate a decision–assignment tree as in [13]. There are two types of nodes: *Read* nodes are 2^b-ary and labelled by a memory address, computation proceeds to the child identified at that address; *write* nodes are unary and labelled by a memory address and a b-bit value, with the obvious semantics.

To each query we associate a nondeterministic decision tree of arity 2^b whose internal nodes are labelled by a memory address or by '$\exists$'. The leaves are labelled 0 or 1 to represent the possible answers to the query. We define the value $qM \in \{0, 1\}$ computed by a query tree q on memory M to be 1 if there exists a path from the root to a leaf with label 1. A witness of such an accepting computation is the description of the choices for the $\exists$ nodes. We let q_i denote the query tree corresponding to *query*(i). The query time t_{q} is the height of the largest query tree and the update time t_{u} is the height of the largest update tree; we account only for memory reads and writes and for nondeterministic choices, all other computation is for free.

3.2. Updates and epochs. Each update sequence in U is described by a binary string $u \in \{0, 1\}^*$. Each bit represents an update *update*(j, a). The parameters for these updates will be specified in Sect. 3.5. The update sequences $u \in U$ are split into d substrings each corresponding to an *epoch*. It turns out to be convenient that time flows backwards, so epoch 1 corresponds to the end of u. In general the update string is an element in

$$U = U_d U_{d-1} \cdots U_1, \qquad \text{where } U_t = \{0, 1\}^{e(t)},$$

where $e(t)$ is the length of epoch t given by

$$e(t) + \cdots + e(1) = \left\lfloor \frac{n^{t/d}}{d} \right\rfloor. \tag{3}$$

The length of the entire update sequence is $\lfloor n/d \rfloor$. The size of d and hence the growth rate of $e(t)$ is given by

$$d = \left\lceil \frac{\log n}{\log(b t_{\mathrm{u}} \log n)} \right\rceil \tag{4}$$

The goal is to establish that $t_{\mathrm{q}} \in \Omega(d)$.

3.3. Time stamps and nondeterminism. To each cell we associate a time stamp when it is written. A cell receives time stamp t if some update during epoch t writes to it, and none of the subsequent updates during epochs $t - 1$ to 1 write to it.

For an update sequence $u \in U$ let M^u denote the memory resulting from these updates (recall that updates are restricted to perform deterministically), starting with some arbitrary initial contents corresponding to the initial instance 0^n.

For index i and update string u let $T(i, u)$ denote the set of time stamps that are found on every accepting computation path of q_i on M^u. If there are no accepting computations, the set is empty. More formally, let w denote a witness for a computation path of q_i on M^u, and let $A(i, u)$ denote the set of witnesses that lead to accepting computations of q_i on M^u. Let for a moment $T(i, u, w)$

denote the set of time stamps encountered by the computation of q_i on M^u that is witnessed by w. Then

$$T(i,u) = \bigcap_{w\in A(i,u)} T(i,u,w), \quad \text{if } A(i,u) \neq \emptyset,$$

and $T(i,u)=\emptyset$ otherwise.

The simple lemma below is the tool to identify a read of a cell with time stamp t by nondeterministic queries.

Lemma 1. *If M^u and M^v differ only on cells with time stamp t then*

$$q_i M^u \neq q_i M^v \quad \textit{implies } t \in T(i,u) \cup T(i,v).$$

Proof. Suppose on the contrary $q_i M^u \neq q_i M^v$ and $t \notin T(i,u)\cup T(i,v)$. Assume without loss of generality that $q_i M^u = 1$ and $q_i M^v = 0$. Since $q \notin T(i,u)$ and $q_i M^u = 1$ there is an accepting computation path that avoids cells with time stamp t. However, this computation might as well be executed on M^v, by the premise. Hence q_i has an accepting computation on M^v as well, contradicting $q_i M^v = 0$. ∎

3.4. Lower bound on query time. The update sequences are chosen such that even if two sequences differ only in a single epoch, they still result in very different instances. To each update sequence $u \in U$ we associate the query vector $q^u = (q_1M^u, q_2M^u, \ldots, q_nM^u) \in \{0,1\}^n$. Update sequences that differ only in epoch t are called *t-different.*

Lemma 2. *No Hamming ball of diameter $\frac{1}{8}n$ can contain more than $|U_t|^{9/10}$ query vectors from t-different update sequences, for large n.*

The difficult part is constructing a set of update sequences for which the statement is true, which we present in Sect. 3.5. The proof itself is as in [13] and provided in Sect. 3.5 for completeness.

Write $U_{>t}$ for $U_d \cdots U_{t+1}$, the set of updates sequences prior to epoch t, and $U_{\leq t}$ for $U_t \cdots U_1$, the set of update sequences in epoch t to epoch 1. Assume for the rest of this section that $t_{\mathrm{q}} = \mathrm{O}(\log n)$, else there is nothing to prove. The worst-case query time t_{q} is larger than the average of $|T(i,u)|$ over choices of $i \in \{1,\ldots,m\}$ and $u \in U$, so

$$|U|nt_{\mathrm{q}} \geq \sum_{u\in U}\sum_{i=1}^{n}|T(i,u)| = \sum_{t=1}^{d}\sum_{u\in U_{>t}}\sum_{w\in U_{<t}}\sum_{v\in U_t}\sum_{i=1}^{n}\bigl(t\in T(i,uvw)\bigr)$$

The next lemma tells us how many $v\in U_t$ fail to make the last sum exceed $\frac{1}{16}n$.

Lemma 3. *Fix any epoch $1\leq t\leq d$ and past and future updates $x\in U_{<t}$, $y\in U_{>t}$. For large n, at least half of the update sequences $u \in xU_ty$ satisfy*

$$\bigl|\{\,1\leq i\leq n \mid t\in T(i,u)\,\}\bigr| \geq \tfrac{1}{16}n,$$

if $t_{\mathrm{q}} = \mathrm{O}(\log n)$.

Proof. Consider the set $V\subseteq xU_ty$ of updates after which fewer than $\frac{1}{16}n$ queries encounter time stamp t, i.e., xuy for $u\in U_t$ is in V if

$$\bigl|\{\,1\leq i\leq n \mid t\in T(i,xuy)\}\bigr| < \tfrac{1}{16}n.$$

We will bound the size of V below $\frac{1}{2}|U_t|$.

To this end partition V into equivalence classes such that u and v are in the same class if and only if M^u and M^v agree on all cells except maybe those with time stamp t. We first bound the number of such classes. Since all cells with time stamp greater than t have identical content (they depend only on the common prefix x), we only need to analyse the amount of information distributed among cells with time stamps $t-1$ to 1. The number of cells written during the last $t-1$ epochs is at most $r = t_{\mathrm{u}}\cdot\bigl(e(t-1)+\cdots+e(1)\bigr)$. Note that at most $n2^{t_{\mathrm{q}}b}$ different cells appear in the entire forest of query trees. The number of different ways we can choose such r cells and fix their content to some value in $\{0, \dots, 2^b - 1\}$ is bounded by:

$$(n 2^{t_q b} \cdot 2^b)^r \le |U_t|^{o(1)} \tag{5}$$

where the inequality uses (4). That is $|U_t|^{o(1)}$ bounds the number of equivalence classes of V.

It remains to bound the size of each class. Consider two query vectors q^u and q^v for u and v in the same equivalence class. Then

$$|q^u - q^v| \le \tfrac{1}{8} n, \tag{6}$$

because $\frac{15}{16} n$ entries of each vector depend only on cells with other time stamps than t. On these cells, the memories are indistinguishable and therefore yield the same result by Lem. 1. By (6), all vectors from the same class end up in a Hamming ball of diameter $\frac{1}{8}n$, so Lem. 2 tells us that there can be only $|U_t|^{\frac{9}{10}}$ of them. We conclude that the size of V is bounded by $|U_t|^{\frac{9}{10}} \cdot |U_t|^{o(1)}$, which is less than $\frac{1}{2}|U_t|$ for large n. ∎

By this lemma we obtain for large n:

$$|U| n t_q \ge \sum_{t=1}^{d} |U_{>t}| \cdot |U_{<t}| \cdot \tfrac{1}{16} n \cdot \tfrac{1}{2} |U_t| = \tfrac{1}{32} n d |U|,$$

and hence $t_q \ge \frac{1}{32} d$ as desired.

3.5. Update scheme. The technical part that remains is to exhibit a set of update sequences U satisfying Lem. 2. There are a number of ways to do this; the following construction is one which simultaneously anticipates our needs in Sect. 5 and satisfies the balancing condition (2).

Define the number of epochs d was given by (4). To alleviate notation we assume that n/d is an integer. Consider the updates in epoch t and index them as $u_1 \cdots u_{e(t)} \in U_t$. The ith update performs *update*(j, a), where the update position j is given below. The new value a is given by

$$(-1)^r, \quad \text{where } r = 1 + u_1 + \cdots + u_i \bmod 2, \tag{7}$$

i.e., such that the nonzero updates in u alternate between -1 and $+1$, starting with $+1$. The position of the affected letter is defined as follows. Write x as a table of dimension $d \times n/d$ like this:

$$\begin{bmatrix} x_1 & x_{d+1} & & x_{n-d+1} \\ x_2 & x_{d+2} & & x_{n-d+2} \\ \vdots & \vdots & \cdots & \vdots \\ x_d & x_{2d} & & x_n \end{bmatrix}.$$

All updates in epoch t will affect only the letters in row t. The updates of an epoch are spread out evenly from left to right across that row, so the distance between two of them is

$$\left\lfloor \frac{n/d}{e(t)} \right\rfloor. \tag{8}$$

In summary, the ith update in epoch t affects the letter in row t and the column given by

$$(i-1) \cdot \left\lfloor \frac{n/d}{e(t)} \right\rfloor + 1.$$

This update scheme satisfies the statement in Lem. 2.

Proof of Lem. 2. Let $xU_t y$ be any set of t-different update sequences. Pick any $u \in U_t$ and consider any Hamming ball of diameter $\frac{1}{8}n$ that contain query vector q^{xuy}. We will bound the number of $v \in U^t$ with query vector q^{xuy} ending up in that Hamming ball.

Let $w \in U_t$ record the difference between u and v, i.e., the ith letter of w is 1 if and only if u and v differ on the ith letter. Now let w' denote the string of prefix sum parities of w, i.e.

$$w'_i = w_1 + \cdots + w_i \mod 2, \quad 1 \le i \le e(t).$$

It is easy to see that w' records the difference between the query vectors resulting from u and v. Indeed, each 1 in w' yields an interval of indices where the vectors differ, and the length of this interval is d times the distance given by (8). In other words, each 1 in w' contributes as many points to the Hamming distance between the resulting query vectors. So if we let $|w'|_1$ denote the number of 1s in w', the Hamming distance between two query vectors is at least

$$|w'|_1 \cdot d \cdot \left\lfloor \frac{n/d}{e(t)} \right\rfloor \geq \tfrac{1}{2}|w'|_1 \cdot \frac{n}{e(t)}, \tag{9}$$

where we have used that $\lfloor a \rfloor \geq \frac{1}{2}a$ for $a \geq 1$.

By the triangle inequality, the maximum Hamming distance between two query vectors in the same ball is $\frac{1}{8}n$. This bounds the number of 1s in w' to $\frac{1}{4}e(t)$ for large n. Hence the number of choices for w' is bounded by

$$\sum_{i=0}^{\frac{1}{4}e(t)} \binom{e(t)}{i} < 2^{\frac{9}{10}e(t)}. \tag{10}$$

for large n. This also bounds the number choices of $v \in U_t$, since there is a one–to–one correspondence between v and w'. ∎

The prefix sums of instances resulting from our scheme are small: Let x denote an instance resulting from our scheme. Let x^t denote the string resulting from only the updates in epoch t and write x as $x^1 + \cdots + x^d$; this works because no two epochs write in the same positions. Then

$$\sum_{j=1}^{i} x_j = \sum_{j=1}^{i}\sum_{t=1}^{d} x_j^t = \sum_{t=1}^{d}\sum_{j=1}^{i} x_j^t \in \{0,\ldots,d\},$$

because the prefix sums of every x^t is 0 or 1 by construction. It can be checked that the balancing bound (2) holds at all times.

For later reference we also note that if x and y result from t-different updates then $x^r = y^r$ for $r \neq t$ and hence

$$\Big|\sum_{j=1}^{i} x_j - \sum_{j=1}^{i} y_j\Big| \leq 1, \tag{11}$$

for all i.

4. Lower bounds for dynamic algorithms and partial sum problems

Theorem 1 suggests a new approach for proving lower bounds by employing nondeterminism in the reduction from signed partial sum. We demonstrate this with a number of examples in this section; our bounds are better than previous results and the proofs are simpler. The results are presented for cell size $b = \log n$ for concreteness. Some of the reductions extend previous work of the authors with Søren Skyum [16].

4.1. Nested brackets. Consider the problem of maintaining a nested structure, i.e., a string x with round and square brackets under the following operations:

change(i,a): change x_i to a, where a is a round or square opening or closing bracket, or whitespace.

balance: return 'yes' if and only if the brackets in x are properly nested.

This problem was studied in [9], where an algorithm with polylogarithmic update time is presented.

Proposition 1. *Maintaining a string of nested brackets requires time $\Omega(\log n/\log\log n)$ per operation.*

Proof. Consider a deterministic algorithm for this problem and let $x \in \{0,-1,+1\}^n$ be an instance to signed partial sum. Let b_i be an encoding of x_i given by:

$$+1 \mapsto \,))\sqcup, \quad 0 \mapsto \,)\sqcup\sqcup, \quad -1 \mapsto \sqcup\sqcup\sqcup\,,$$

where '␣' stands for space. Let c be the string '␣('. We maintain a balanced string of brackets uvw, where $u = c^{2n}$, $v = b_1\,b_2\,\ldots\,b_n$ and $w =)^{n-s}␣^{s}$, where $s = x_1 + \cdots + x_n$. It is easy to see that uvw balances and can be maintained by a constant number of updates per update in x. For any prefix size i this construction enables efficient verification of a nondeterministic guess g of the prefix sum $x_1 + \cdots + x_i$: Place a closing square bracket on the last ␣ of b_i and an opening square bracket on the ␣ of the first c of suffix c^{i+g} of u. This modification keeps uvw balanced iff g is the right guess of prefix sum $x_1 + \cdots + x_i$. Conclusion by Thm. 1. ∎

4.2. Dynamic Graph Algorithms. Our techniques improve the lower bounds of a number of well-studied graph problems considered in [16].

Tamassia and Preparata [25] present an algorithm for the class of *upward planar source–sink graphs* that runs in time $O(\log n)$ per operation. These digraphs have have a planar embedding where all edges point upward (meaning that their projection on some fixed direction is positive) and where exactly one node has indegree 0 (the source) and exactly one node has outdegree 0 (the sink). The updates are:

insert(u, v): insert an edge from u to v
delete(u, v): delete the edge from u to v if it exists
reachable(u, v): return 'yes' iff there is a path from u to v.

The updates have to preserve the topology of the graph, including the embedding.

Proposition 2. *Dynamic reachability in upward planar source–sink graphs requires time* $\Omega(\log n/\log\log n)$ *per operation.*

Planarity testing is to maintain a planar graph where the query asks whether a new edge violates the planarity of the graph. Italiano *et al.* [17] present an efficient algorithm for a version of this problem, and a strong lower bound is exhibited by Henzinger and Fredman [12]. Our lower bound holds also for *upward* planarity testing, where the topology is further restricted to upward planar graphs. The updates insert and delete edges as above, and the query is

planar(u, v): return 'yes' if and only if the graph remains upward planar after insertion of edge (u, v).

This problem was studied by Tamassia [24], who found an $O(\log n)$ upper bound.

Proposition 3. *Upward planarity testing requires time* $\Omega(\log n/\log\log n)$ *per operation.*

A classical problem in Computational Geometry is *planar point location*: given a subdivision of the plane, i.e., a partition into polygonal regions induced by the straight-line embedding of a planar graph, determine the region of query point $q \in \mathbf{R}^2$. An important restriction of the problem considers only *monotone* subdivisions, where the subdivision consists of polygons that are monotone (so no horizontal line crosses any polygon more than twice). In the dynamic version of this problem updates manipulate the geometry of the subdivision. Preparata and Tamassia [23] give an algorithm that runs in time $O(\log^2 n)$ per operation, this was improved to query time $O(\log n)$ by Baumgarten, Jung, and Mehlhorn [3]. The lower bound for this problem in [16] applies only to algorithms returning the name of the region containing the queried point. The techniques of the present paper extend this bound to work for simpler decision queries like

query(x): return 'yes' if and only if x is in the same polygon as the origin.

Proposition 4. *Planar point location in monotone subdivisions requires time* $\Omega(\log n/\log\log n)$ *per operation.*

Traditionally, lower bounds in Computational Geometry are proved in an algebraic, comparison-based model (see [22] for a textbook account) that is broken by standard RAM operations like indirect addressing, bucketing, hashing, etc. Cell probe lower bounds for that field are lacking.

To explain our reduction we turn to the conceptually very simple class of *grid graphs.* The vertices of a grid graph of width w and height h are integer points (i, j) in the plane for $1 \le i \le w$ and $1 \le j \le h$. All edges have length 1 and are parallel to the axes. The dynamic reachability problem for these graphs is the following:

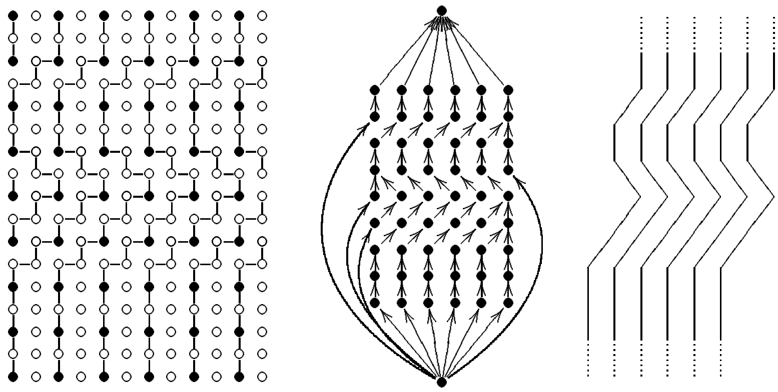

FIGURE 1. Planar graphs corresponding to $x = (0,\ 0,\ +1,\ +1,\ -1,\ 0,\ +1,\ 0)$. Left: grid graph. Even grid points are marked $\bullet$, odd grid points are marked $\circ$. Middle: upward planar source–sink graph. Right: monotone planar subdivision.

flip(x, y): add an edge between $x \in [w] \times [h]$ and $y \in [w] \times [h]$ or remove it if it exists,
reachable(x, y): return 'yes' if and only if there is a path from x to y.

There are several well-known constructions that prove a lower bound for this problem [8, 12, 15, 20], but our proof translates to the other problems in Props. 2 to 4. The details in these constructions are omitted, Fig. 1 illustrates the structures arising in the reductions.

Proposition 5. *Dynamic reachability in grid graphs requires time* $\Omega(\log n/\log\log n)$ *per operation.*

Proof. From an instance $x \in \{0, \pm 1\}^n$ to signed partial sum we build a grid graph on the points $\{0, \dots, 2w\} \times \{0, \dots, 2n\}$, where $w = \lceil \log n/\log\log n \rceil$. We will exploit the balancing constraint (2) of Thm. 1 to keep the instance within this width.

For every i and j, consider any point with even coordinates $(2i, 2j-2)$, drawn as $\bullet$ in Fig. 1, and connect it to one of the three even grid points above it using , , or , depending on whether $x_j = +1$, 0, or -1, respectively. The idea is that the path from $(0, 0)$ mimics the prefix sums of x in that it passes through $(2s, 2j)$ if and only if $x_1 + \cdots + x_j$ equals s. Hence a guess of the sum can be verified by a single reachability query in the graph.

It remains to note that the graph can be maintained efficiently. Any changed letter in x incurs $\mathrm{O}(w)$ edges to be inserted or deleted. So if the update time of the graph algorithm is polylogarithmic then the graph can be maintained in polylogarithmic time. The bound follows from Thm. 1. ∎

The width of the hard graph above is logarithmic in the height, while the graphs constructed in [8, 12, 15, 20] are square. Hence narrow grid graphs are as hard as square ones. However, this is not true for *very* narrow graphs: It is known that the reachability problem for grid graphs of *constant* width can be solved in time $\mathrm{O}(\log\log n)$ by [2], an exponential improvement. This leaves open the question of what happens for graphs of sublogarithmic width. To answer this, we introduce a subtler statement of Thm. 1.

Theorem 1 (Parameterised version). *Let* $d = \mathrm{O}\big(\log n/\log(bt_{\mathrm{u}}\log n)\big)$ *be an integer function. Every nondeterministic algorithm for signed partial sum with cell size* b*, update time* t_{u}*, and query time* t_{q} *must satisfy* $t_{\mathrm{q}} = \Omega(d)$*. The lower bound holds even if the algorithm requires* $0 \le x_1 + \cdots + x_i \le d$ *for all* i *after each update.*

This result implies a lower bound for grid graphs that smoothly connects the two extremes between linear and constant width. A similar parameterisation can be done for all our problems.

Proposition 6. *For every* $w = \mathrm{O}(\log n/\log\log n)$*, dynamic reachability in grid graphs of width* w *requires time* $\Omega(w)$ *per operation.*

4.3. Partial Sum Problems. The partial sum problem is to maintain a bit string $x \in \{0,1\}^n$ under the following operations

update(i): change x_i to $1 - x_i$,
sum(i): return $x_1 + \cdots + x_i$.

We turn to two other natural variants, *prefix majority* and *prefix equality* whose query operations are

majority(i): return 1 iff $x_1 + \cdots + x_i \geq \lceil \frac{1}{2} n \rceil$,
equality(i): return 1 iff $x_1 + \cdots + x_i = \lceil \frac{1}{2} n \rceil$.

The next result shows that these problems are just as hard as the parity query from [13]. The proof is again a simple application of Thm. 1.

Proposition 7. *The prefix equality and prefix majority problems both require time* $\Omega(\log n / \log\log n)$ *per operation.*

Proof. We give the proof for prefix equality. The proof for the prefix majority problem is almost the same. Let $d = \lceil \log n / \log\log n \rceil$.

An instance $x \in \{-1, 0, +1\}^n$ of signed partial sum is encoded as the binary string x' by

$$-1 \mapsto 00, \quad 0 \mapsto 01, \quad +1 \mapsto 11.$$

We maintain $d+1$ strings $y^{(0)}, \ldots, y^{(d)}$ as

$$y^{(t)} = (00)^t (01)^{d-t} x'.$$

Let t_{u} denote the update time of our prefix equality algorithm. Whenever x is changed, we make at most $2(d+1)$ updates in the strings $y^{(t)}$, which is within polylogarithmic time if t_{u} is polylogarithmic.

Index the strings $y^{(t)}$ from $-2d$ to $2n-1$. We then have

$$\sum_{j=-2d}^{2i-1} y_j^{(t)} = d - t + i + \sum_{j=1}^{i} x_j, \qquad 0 \leq t \leq d, \quad 1 \leq i \leq n. \tag{12}$$

Hence in order to find the ith prefix sum of x our algorithm can nondeterministically guess the sum $s \in \{0, \ldots, d\}$, we can assume from the balancing condition (2) in Thm. 1 that the sum is in that set, and verify $y_{-2d}^{(s)} + \cdots + y_{2i-1}^{(s)} = d + i$, which is the case iff *equality*$(2d + 2i)$ on $y^{(s)}$ returns 1. Conclusion by Thm. 1. ∎

There are other partial sum problems that are far easier. Consider the query

or(i): return 'yes' iff $x_1 + \cdots + x_i \geq 1$.

This problem, *prefix-or*, can be solved in time $\mathrm{O}(\log\log n)$ per operation by a van Emde Boas tree. To study this kind of problem in a general, let the *threshold* ϑ be an integer function such that $\vartheta(i) \in \{0, \ldots, \lceil \frac{1}{2} i \rceil\}$. The query in the *prefix threshold problem for* ϑ is

threshold(i): return 'yes' iff $x_1 + \cdots + x_i \geq \vartheta(i)$.

Prefix majority is the special case $\vartheta(i) = \lceil \frac{1}{2} i \rceil$, prefix-or is $\vartheta(i) = 1$. Now for our lower bound. Our assumption on ϑ is that there are integers $p(1) < p(2) < \cdots < p(i) < \cdots$ such that $\vartheta(p(i)) = i$. We call such functions *nice* for lack of a better word. It is reasonable to assume that ϑ is monotonically increasing, the niceness assumption also prevents it from skipping points.

Proposition 8. *Let* $t_{\mathrm{u}} = t_{\mathrm{u}}(n)$ *and* $t_{\mathrm{q}} = t_{\mathrm{q}}(n)$ *denote the update and query time of any cell size* b *implementation of the prefix threshold problem for a nice threshold* ϑ*. Then*

$$t_{\mathrm{q}} = \Omega\Big(\frac{\log \vartheta}{\log(t_{\mathrm{u}} b \log \vartheta)} \Big). \tag{13}$$

The proof is not difficult but tedious. The idea is to stretch an instance for a threshold problem, padding it with sufficiently many 0s or 1s to turn it into a majority problem.

To gauge the strength of this result we mention that the problem can be solved on the unit-cost RAM with logarithmic cell-size in time

$$\mathrm{O}\left(\frac{\log \vartheta}{\log\log n} + \log\log n\right), \tag{14}$$

per update (if $\vartheta(1), \dots, \vartheta(n)$ can be computed in the preprocessing stage of the algorithm). The left term in the expression stems from a search tree, the right term from a priority queue, which vanishes for cell size $b = \Omega(\log^2 n)$; details are omitted. Comparing (13) with (14) shows that the lower bound is tight for logarithmic cell size and $\vartheta = \Omega(\log^{\log\log n} n)$. For smaller thresholds, the bounds leave a gap of size $\mathrm{O}(\log\log n)$. We consider a more general problem in Sect. 5.2.

5. Refinement

We now take a somewhat subtler approach to our basic question than in Sect. 2. Instead of nondeterminism, we study the performance of query algorithms in a *promise* setting. We assume that the query algorithm for signed partial sum receives a value s that is promised to be *close to* (but not known to be equal to) the right sum and then decides between right and wrong values.

The *partial sum refinement* problem can be phrased as follows: Maintain a string $x \in \{0, \pm 1\}^n$, initially 0^n, under the following operations:

update(i, a): change x_i to $a \in \{-1, 0, +1\}$,

parity(i, s): return $x_1 + \cdots + x_i \mod 2$ provided that $|s - \sum_{j=1}^{i} x_j| \le 1$ (otherwise the behaviour of the query algorithm is undefined).

The problem gets its name from the following alternative definition, where the query operation is replaced by

refine(i, s): return 1 if $s = \sum_{j=1}^{i} x_j$ and 0 if $s \ne \sum_{j=1}^{i} x_j$, provided that $|s - \sum_{j=1}^{i} x_j| \le 1$. For other values of s, the answer is undefined.

The two problems reduce to each other.

Theorem 2. *Let d be an integer function such that*

$$d = \mathrm{O}\left(\frac{\log n}{\log(t_{\mathrm{u}} b \log n)}\right)$$

. Every algorithm for partial sum refinement with cell size b, update time t_{u} and query time t_{q} must satisfy $t_{\mathrm{q}} = \Omega(d)$. Moreover, this is true even for algorithms that require $0 \le x_1 + \cdots + x_i \le d$ for all i after each update.

5.1. Proof of Thm. 2. Most of the technical work for this result was already done in Sect. 3.5, where we found that the instances resulting from two t-different updates have close prefix sums (11).

The query trees in our computational model are now deterministic decision trees as in [13]. But there are more of them: we associate a tree q_i^s to each query *parity*(i, s), yielding $n(2n+1)$ trees. (We could reduce this number to $n(d+1)$ by the balancing constraint, but that does not improve the bounds.)

For update string u we write q_i^u for the query tree q_i^s corresponding to the 'right guess' $s = x_1 + \cdots + x_i$, where x is the instance resulting from updates u. The *query vector* is $(q_1^u M, \dots, q_n^u M)$, i.e., the responses yielded by guessing right every time. We let $T(i, u)$ denote the time stamps encountered by q_i^u on M^u, compare this with the construction in 3.3.

The next lemma corresponds to Lemma 1 and shows that our update scheme constructs different instances whose prefix sums are so close that the query trees cannot use the (almost correct) value given to them.

Lemma 4. *For t-different update sequence $u, v \in U_t$, if M^u and M^v differ only on cells with time stamp t then for all i:*

$$q_i^u M^u \ne q_i^v M^v \quad \textit{implies } t \in T(i, u) \cup T(i, v)$$

Proof. Assume to the contrary for some such t, u, v and i that $t \notin T(i,v)$ and $q_i^u M^u \neq q_i^v M^v$. Let x and y denote the input instances resulting from u and v, respectively. Let s denote $\sum_{j=1}^{i} x_j$. By (11) and without loss of generality, $\sum_{j=1}^{i} y_j = s+1$. By correctness, $q_i^s M^u = q_i^{s+1} M^u$. Since the computation path for $q_i^{s+1} M^v$ does not encounter time stamp t this computation might as well be executed on M^u with the same result, i.e., $q_i^{s+1} M^u = q_i^{s+1} M^v = q_i^s M^u = q_i^u M^u$. But this contradicts our assumption $q_i^u M^u \neq q_i^v M^v = q_i^{s+1} M^v$. ∎

The rest of the proof can be reused almost *ad verbatim*.

5.2. The Dynamic Prefix Problem for Symmetric Functions. Thm. 2 acts as an important ingredient in characterising the dynamic complexity of all the *symmetric functions*, generalising the results for the threshold functions of last section. A Boolean function is *symmetric* if it depends only on the number of 1s in the input $x = (x_1, \ldots, x_n)$. The symmetric functions include some of the most well-studied functions in complexity theory, like parity, majority, and the threshold functions.

In general, we can describe every symmetric function f in n variables by its *spectrum*, a string in $\{0,1\}^{n+1}$ whose ith letter is the value of f on inputs where exactly i variables are 1. The *boundary* of a spectrum s is the smallest value ϑ such that $s_{\lfloor \vartheta \rfloor} = s_{\lfloor \vartheta \rfloor + 1} = \cdots = s_{\lfloor n-\vartheta \rfloor}$. For instance the boundary of the parity or majority functions is $\frac{1}{2}n$, and for the threshold functions with threshold ϑ, the boundary is $\min(\vartheta, n-\vartheta)$.

Let $\langle f_n \rangle = (f_1, \ldots, f_n)$ be a sequence of symmetric Boolean function where the ith function f_i takes i variables. The *dynamic prefix problem for* $\langle f_n \rangle$ is to maintain a bit string $x \in \{0,1\}^n$ under the following operations:

update(i): change x_i to $\neg x_i$,

query(i): return $f_i(x_1, \ldots, x_i)$.

For example, taking f_i to be the parity function on i variables we have the prefix parity problem of [13], and taking f_i to be the threshold function for $\vartheta(i)$ we have the problem from Prop. 8.

Proposition 9. *Let ϑ be a nice function and let $\langle f_n \rangle$ be a sequence of symmetric functions where $f_i\colon \{0,1\}^i \to \{0,1\}$ has boundary $\vartheta(i)$. Let t_u and t_q denote the update and query time of any cell size b implementation of the dynamic prefix problem for $\langle f_n \rangle$. Then $t_\mathrm{q} = \Omega\big(\log\vartheta / \log(t_\mathrm{u} b \log\vartheta)\big)$.*

Proof. First assume that f_i's boundary is in the middle, i.e. $\vartheta(i) = \frac{1}{2}i$. Let $x \in \{+1, 0, -1\}^n$ denote an instance to prefix refinement and define d and maintain $d+1$ strings as in the proof for Prop. 7. Using the data structure for $\langle f_n \rangle$ we perform *refine*(i,g) as follows. Let s be the spectrum for f_{2i+2d}. Since its boundary is in the middle it is the case that

$$s_{d+i-1} s_{d+i} s_{d+i+1} \in \{001, 010, 011, 100, 101, 110\}.$$

We only consider the case 001 above—the other cases are treated similarly. Recall that we can assume $x_1 + \cdots + x_i \in \{g-1, g, g+1\}$. Let r_{-1}, r_0 and r_{+1} denote the answer of *query*$(2d+2i)$ on $y^{(g-1)}$, $y^{(g)}$ and $y^{(g+1)}$ respectively. By (12) in the proof of Prop. 7, if $g = x_1 + \cdots + x_i$ then $r_{-1} r_0 r_{+1} = s_{d+i-1} s_{d+i} s_{d+i+1} = 001$. If instead $g-1$ is the correct sum then $r_{-1} r_0 = 01$ and finally if $g+1$ is the correct sum then $r_0 r_{+1} = 00$. Hence these three cases for g can be distinguished by the above three queries and hence determine the correct answer for *refine*(i,g). The bound then follows from Thm. 2.

The rest of the proof is a padding argument that 'stretches' the above to work for smaller ϑ. We omit the details. ∎

Intriguingly, the bound in the proposition is precisely the same bound as for the size–depth trade-off for Boolean circuits for these functions [14, 6, 21].

Acknowledgements

The authors had valuable discussions with Arne Anderson, Gerth Stølting Brodal, Peter Bro Miltersen, and Sven Skyum about various aspects of this work.

References

[1] Miklós Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, 8(3):235–247, 1988.

[2] David A. Mix Barrington, Chi-Jen Lu, Peter Bro Miltersen, and Sven Skyum. Searching constant width mazes captures the AC^0-hierarchy. In *Ann. Symp. on Theoretical Aspects of Computer Science (STACS)*, 1998.

[3] Hanna Baumgarten, Hermann Jung, and Kurt Mehlhorn. Dynamic point location in general subdivisions. In *Proc. 3rd SODA*, pages 250–258, 1992.

[4] Paul Beame and Faith Fich. On searching sorted lists: A near-optimal lower bound. Manuscript, 1997.

[5] Amir M. Ben-Amram and Zvi Galil. Lower bounds for data structure problems on RAMs. In *Proc. 32nd FOCS*, pages 622–631. IEEE Computer Society, 1991.

[6] Bettina Brustmann and Ingo Wegener. The complexity of symmetric functions in bounded-depth circuits. *Information Processing Letters*, 25(4):217–219, 1987.

[7] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. 1st WADS*, volume 382 of *Lecture Notes in Computer Science*, pages 39–46. Springer Verlag, Berlin, 1989.

[8] David Eppstein. Dynamic connectivity in digital images. *Information Processing Letters*, 62(3):121–126, 1997.

[9] Gudmund Skovbjerg Frandsen, Thore Husfeldt, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum. Dynamic algorithms for the Dyck languages. In *Proc. 4th WADS*, volume 955 of *Lecture Notes in Computer Science*, pages 98–108. Springer, 1995.

[10] Michael L. Fredman. Observations on the complexity of generating quasi-Gray codes. *SIAM Journal of Computing*, 7(2):134–146, 1978.

[11] Michael L. Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM*, 29:250–260, 1982.

[12] Michael L. Fredman and Monika Rauch Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. To appear in Algorithmica. A preliminary version appears as "Improved data structures for fully dynamic biconnectivity" in Proc. 26th STOC, 1994.

[13] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st STOC*, pages 345–354, 1989.

[14] Johann T. Håstad. Almost optimal lower bounds for small depth circuits. In *Proc. 18th STOC*, pages 6–20, 1986.

[15] Thore Husfeldt. Fully dynamic transitive closure in plane dags with one source and one sink. In *Proc. 3rd ESA*, volume 955 of *Lecture Notes in Computer Science*, pages 199–212. Springer Verlag, Berlin, 1995.

[16] Thore Husfeldt, Theis Rauhe, and Søren Skyum. Lower bounds for dynamic transitive closure, planar point location, and parentheses matching. *Nordic Journal of Computing*, 3(4):323–336, 1996.

[17] Giuseppe F. Italiano, Johannes A. La Poutré, and Monika H. Rauch. Fully dynamic planarity testing in planar emdedded graphs. In *Proc. 1st Ann. European Symp. on Algorithms (ESA)*, volume 726 of *Lecture Notes in Computer Science*, pages 212–223. Springer Verlag, Berlin, 1993.

[18] Peter Bro Miltersen. Lower bounds for union–split–find related problems on random access machines. In *Proc. 26th STOC*, pages 625–634. ACM, 1994.

[19] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. In *Proc. 27th STOC*, pages 103–111. ACM, 1995.

[20] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia. Complexity models for incremental computation. *Theoretical Computer Science*, 130:203–236, 1994.

[21] Shlomo Moran. Generalized lower bounds derived from Hastad's main lemma. *Information Processing Letters*, 25:383–388, 1987.

[22] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry*. Springer, 1985.

[23] Franco P. Preparata and Roberto Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM Journal of Computing*, 18(4):811–830, 1989.

[24] Roberto Tamassia. On-line planar graph embedding. *Journal of Algorithms*, 21(2):201–239, 1996.

[25] Roberto Tamassia and Franco P. Preparata. Dynamic maintenance of planar digraphs, with applications. *Algorithmica*, 5:509–527, 1990.

[26] B. Xiao. *New bounds in cell probe model*. Doctoral dissertation, University of California, San Diego, 1992.

[27] Andrew C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, July 1981.

[28] Andrew C. Yao. On the complexity of maintaining partial sums. *SIAM Journal of Computing*, 14(2):277–288, 1985.

Recent BRICS Report Series Publications

RS-97-32 Thore Husfeldt and Theis Rauhe. *Hardness Results for Dynamic Problems by Extensions of Fredman and Saks' Chronogram Method*. November 1997. i+13 pp.

RS-97-31 Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. *Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using* UPPAAL. November 1997. 23 pp. To appear in *The 18th IEEE Real-Time Systems Symposium, RTSS '97 Proceedings*.

RS-97-30 Ulrich Kohlenbach. *Proof Theory and Computational Analysis*. November 1997. 38 pp.

RS-97-29 Luca Aceto, Augusto Burgueño, and Kim G. Larsen. *Model Checking via Reachability Testing for Timed Automata*. November 1997. 29 pp.

RS-97-28 Ronald Cramer, Ivan B. Damgård, and Ueli Maurer. *Span Programs and General Secure Multi-Party Computation*. November 1997. 27 pp.

RS-97-27 Ronald Cramer and Ivan B. Damgård. *Zero-Knowledge Proofs for Finite Field Arithmetic or: Can Zero-Knowledge be for Free?* November 1997. 33 pp.

RS-97-26 Luca Aceto and Anna Ingólfsdóttir. *A Characterization of Finitary Bisimulation*. October 1997. 9 pp. To appear in *Information Processing Letters*.

RS-97-25 David A. Mix Barrington, Chi-Jen Lu, Peter Bro Miltersen, and Sven Skyum. *Searching Constant Width Mazes Captures the AC^0 Hierarchy*. September 1997. 20 pp. To appear in *STACS '98: 15th Annual Symposium on Theoretical Aspects of Computer Science Proceedings*, LNCS, 1998.

RS-97-24 Søren B. Lassen. *Relational Reasoning about Contexts*. September 1997. 45 pp. To appear as a chapter in the book *Higher Order Operational Techniques in Semantics*, eds. Andrew D. Gordon and Andrew M. Pitts, Cambridge University Press.

RS-97-23 Ulrich Kohlenbach. *On the Arithmetical Content of Restricted Forms of Comprehension, Choice and General Uniform Boundedness*. August 1997. 35 pp.