SIAM J. COMPUT.
Vol. 32, No. 3, pp. 736–753

NEW LOWER BOUND TECHNIQUES FOR DYNAMIC PARTIAL SUMS AND RELATED PROBLEMS*

THORE HUSFELDT† AND THEIS RAUHE‡

Abstract. We study the complexity of the dynamic partial sum problem in the cell-probe model. We give the model access to nondeterministic queries and prove that the problem remains hard. We give the model access to the right answer ± 1 as an oracle and prove that the problem remains hard. This suggests which kind of information is hard to maintain.

From these results, we derive a number of lower bounds for dynamic algorithms and data structures: We prove lower bounds for dynamic algorithms for existential range queries, reachability in directed graphs, planarity testing, planar point location, incremental parsing, and fundamental data structure problems like maintaining the majority of the prefixes of a string of bits. We prove a lower bound for reachability in grid graphs in terms of the graph's width. We characterize the complexity of maintaining the value of any symmetric function on the prefixes of a bit string.

Key words. cell-probe model, partial sum, dynamic algorithm, data structure

AMS subject classifications. 68Q17, 68Q10, 68Q05, 68P05

PII. S0097539701391592

1. Introduction.

The partial sum problem is to maintain n bits $x_1, \ldots, x_n \in \{0, 1\}$ that are subject to updates

$$\textit{update}(i)\text{: change } x_i \text{ to } 1 - x_i$$

and compute queries about the partial sums $x_1 + \cdots + x_i$.

It is easy to construct data structures that provide either very fast updates (by computing the answer from scratch after each query) or very fast queries (by recomputing all partial sums after each update). However, in many partial sum problems—and in many dynamic problems in general—we cannot have both. This trade-off between *update time* and *query time* was established by Fredman and Saks [15], who showed that, with the parity query

$$\textit{parity}(i)\text{: return } x_1 + \cdots + x_i \mod 2,$$

the partial sum problem requires time $\Omega(\log n / \log\log n)$ per operation on the unit-cost RAM with logarithmic cell size. In other words, even the least significant bits of the partial sums are hard to maintain.

The motivation for the present paper is that the hardness of the problem depends on the following query: If the parity query is replaced by

$$\textit{or}(i)\text{: return "yes" iff } x_1 + \cdots + x_i \geq 1 \text{ (equivalently, return } x_1 \vee \cdots \vee x_i),$$

*Received by the editors June 29, 2001; accepted for publication (in revised form) October 22, 2002; published electronically April 23, 2003. A preliminary version of this paper appeared in *Proceedings of the 25th ICALP*, Lecture Notes in Comput. Sci. 1443, Springer-Verlag, Berlin, 1998, pp. 67–78.

http://www.siam.org/journals/sicomp/32-3/39159.html

†Department of Computer Science, Lund University, P.O. Box 118, SE–221 00 Lund, Sweden (thore@cs.lu.se). This author was partially supported by Swedish TFR, the ESPRIT Long Term Research Programme of the EU, project 20244 (ALCOM-IT), and BRICS (Basic Research in Computer Science), a center of the Danish National Research Foundation.

‡IT University of Copenhagen, Glentevej 67, DK–2400 Copenhagen NV, Denmark (theis@it-c.dk). This author was partially supported by the ESPRIT Long Term Research Programme of the EU, project 20244 (ALCOM-IT), and BRICS (Basic Research in Computer Science), a center of the Danish National Research Foundation.

then a Van Emde Boas tree provides an implementation in time $O(\log \log n)$ per operation, which is exponentially faster.

We show which queries are hard in this sense.

General partial sum queries. Consider two other natural partial sum queries:

$$\begin{aligned} &\textit{majority}(i)\text{: return 1 iff } x_1 + \cdots + x_i \geq \lceil \tfrac{1}{2} i \rceil, \\ &\textit{equality}(i)\text{: return 1 iff } x_1 + \cdots + x_i = \lceil \tfrac{1}{2} i \rceil. \end{aligned}$$

We can formulate 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.

Proposition 3 of the present paper shows that both problems require time $\Omega(\log n / \log \log n)$ per operation, just as parity. We then extend our analysis of the majority problem to the class of *threshold* functions and characterize the complexity of the resulting partial sum problem in terms of the size of the threshold in Proposition 4. This connects the majority problem, where the threshold is $\frac{1}{2}i$, and the *or* problem above, where the threshold is 1. Finally, we generalize this to the entire class of *symmetric* functions in Proposition 11.

Intriguingly, the resulting bounds closely resemble the corresponding results from Boolean circuit complexity, where these problems have been studied intensively, hinting at a connection between the dynamic and parallel realms.

Main contribution. Our main technical and conceptual contributions are lower bounds for partial sum problems in very strong models of computation. All our other results follow from these bounds.

The idea is to provide the query algorithm with well-defined parts of the answer for free without reducing the problem's complexity. We phrase the results for the *signed partial sum problem*. The problem is to maintain a string $x \in \{-1, 0, +1\}^n$ under the following operations:

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

We prove two theorems about this problem.

Theorem 1 shows that, even in models with *nondeterministic* queries (defined and discussed in section 2), the partial sum problem requires time $\Omega(\log n / \log \log n)$ per operation. It is known that this is also the deterministic complexity of the problem [9, 15], so nondeterminism does not help.

Theorem 3 studies the same problem in a *promise* setting, where the (deterministic) query algorithm receives an almost correct answer for free. The updates are as before, and the query is

$$\textit{parity}(i, s)\text{: return } x_1 + \cdots + x_i \mod 2 \text{ provided that } \left| s - \sum_{j=1}^{i} x_j \right| \leq 1$$

(otherwise, the behavior of the query algorithm is undefined).

We show that this problem still requires $\Omega(\log n / \log \log n)$ per operation.

Lower bounds for dynamic algorithms. We present some applications to dynamic algorithms and data structure problems other than partial sums. Because Theorems 1 and 3 hold in very strong models of computation, we can construct powerful reductions.

We can show that the existential problem for orthogonal range queries in the plane requires time $\Omega(\log^{1/2} n)$ per operation (Proposition 2). We also present bounds for planar point location in monotone subdivisions [5, 26], reachability in upward planar digraphs [28], and incremental parsing of balanced parentheses [11]. We show that these problems require time $\Omega(\log n/\log\log n)$ per operation (Propositions 5–8). It is known [10, 14, 17, 23] 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 [4], an exponential improvement. We prove a lower bound that is parameterized by the width w of the graph: Proposition 10 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.

Apart from the bound for the existential range query problem, for which the authors recently proved a stronger bound using a different technique [3], all these bounds are new and the best known.

Related work. Fredman introduced the partial sum problem as a "toy problem which is both tractable and surprisingly interesting" [13], and it has been the focal point of many investigations of dynamic complexity in a variety of models [15, 31]. We reason within the cell-probe model of Fredman [12] and Yao [30] with some extensions to cope with our stronger modes of computation. The model can be viewed as a nonuniform version of the random access computer with arbitrary register instructions. Lower bounds are especially valid on RAMs with unit-cost instructions and logarithmic cell size. The success of this model is partly due to the validity of these bounds 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 3 are proved by extending the chronogram method, which was introduced by Fredman and Saks [15] and got its name in [7].

The prefix parity problem was solved in [15], but no nontrivial lower bounds for the majority or equality problems follow from that. The results from [6, 21, 22, 29] 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 [2]; and [19] reports $\Omega((\log n/\log\log n)^{1/2})$ for equality and $\Omega(\log n/(\log\log n)^2)$ for the majority.

2. Nondeterminism in dynamic algorithms.

2.1. Example: Range queries. We can illustrate our concept of nondeterministic queries using the *existential range query* problem. The object is to maintain a set $S \subseteq \{1,\dots,n\}^2$ of points in the plane under the following operations:

update(x): add $x \in \{1,\dots,n\}^2$ to S, or remove it if it is already there,

exists(y): return "yes" iff S contains a point x in the rectangle defined by the origin and y, i.e., such that $x_1 \le y_1$ and $x_2 \le y_2$.

With *nondeterministic* queries, the problem is very easy: guess a point and verify that it is in $S \cap R$. This shows that positive instances of this problem have short,

maintainable witnesses—the points themselves. On the other hand, it is known that, for deterministic computation, this problem requires time $\Omega(\log n/\log\log n)$ [3]; we prove a somewhat weaker bound in Proposition 2. Thus 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; we shall show that queries about the *size* $|R \cap S|$ remain hard even with nondeterminism. Thus we see that the hardness of the two problems, both of which have the same deterministic complexity, hinges on information of a fundamentally different kind.

Another 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.

Thus we have identified a class of dynamic problems, namely, those with fast nondeterministic queries. Problems in this class have positive instances with short witnesses, and these witnesses can be maintained by an efficient data structure. This encompasses the class of problems where the outcome of each query depends on only a small number of updates. Contrast this with the problems identified in [15], where each update affects only a small number of queries, e.g., dictionary problems.

2.2. A model for nondeterministic query algorithms. We introduce a model for nondeterministic query algorithms for dynamic *decision* problems, where the query returns 0 or 1. We allow query algorithms to nondeterministically load a value into a memory cell. The semantics are as usual: The value returned by a nondeterministic query is 1 unless all nondeterministic choices return 0. For example, the following program solves in constant time the existential range query problem, storing all points from S in a two-dimensional array M:

update(x_1, x_2):
  $M[x_1, x_2] := 1 - M[x_1, x_2]$,

exists(y_1, y_2):
  guess $x_1 \le y_1$ and $x_2 \le y_2$
  return $M[x_1, x_2]$.

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." 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.

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

of x and queries about the parity of the prefix sums of x:

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

The data structure of Dietz [9] solves this problem deterministically in time $O(\log n/\log\log n)$ per operation with logarithmic cell size. The next theorem states that nondeterministic queries can do no better. We state this theorem as a trade-off between update and query time.

THEOREM 1. *Every nondeterministic algorithm for the signed partial sum problem with cell size b, update time t_u, and query time t_q must satisfy*

$$t_q = \Omega\left(\frac{\log n}{\log(bt_u \log n)}\right). \tag{1}$$

Also, the lower bound holds even if the algorithm requires

$$0 \le x_1 + \cdots + x_i \le \left\lceil \frac{\log n}{\log(bt_u \log n)} \right\rceil \tag{2}$$

for all i after each update.

The proof is given in the next section.

Note that the query cannot distinguish $+1$ from -1 (since $1 = -1 \mod 2$), so a data structure for the signed partial sum problem structure can treat -1 as $+1$. The reason for introducing -1 in the problem is the balancing condition (2), which continues previous work [19] on extending the chronogram method.

In section 5.2, we state a further generalization of Theorem 1, relating the terms in (1) and (2).

2.4. Lower bound for existential range queries. We give a lower bound of size $\Omega(\log^{1/2} n)$ for the existential range query problem; we consider cell size $b = \log n$ for concreteness. The value of this result lies in its simplicity; it provides a good illustration of how to apply Theorem 1. Using a different technique [3], the authors with Alstrup have since established $\Omega(\log n/\log(bt_u))$, which is optimal. However, before the present paper, no lower bound better than $\Omega(\log\log n/\log\log\log n)$ was known for this problem (which is rather central—see the discussion by Agarwal [1]), so the result provides an exponential yet, by now, outdated improvement.

Following [3], we start with the *existential marked ancestor problem*. Consider a full rooted tree with nodes V, number of leaves n, height h, and arity d, where

$$h = \log^{1/2} n, \quad d = 2^h. \tag{3}$$

Let $\pi(v)$ denote the nodes on the path from v to the root (including v). The problem is to maintain a subset of *marked* nodes $M \subseteq V$ under the following operations:

mark(v): insert $v \in V$ in M,

unmark(v): remove $v \in V$ from M,

exists(v): return "yes" iff any of v's ancestors are marked, i.e., if $\pi(v) \cap M \neq \emptyset$.

The *counting marked ancestors* problem supports the same updates, and the query is

parity(v): return $|M \cap \pi(v)| \mod 2$, the parity of the number of marked ancestors of v.

The parity prefix sum problem is a special case of this problem, where the tree is a path. We begin by showing that the problem is hard also for d-ary trees, where $d = \log^{1/2} n$.

LEMMA 1. *Every nondeterministic algorithm for counting marked ancestors in trees with update time t_u requires query time*

$$t_q = \Omega\left(\frac{\log n}{\log^{1/2} n + \log(t_u \log n)}\right).$$

Proof. Let x be a length n instance to the signed partial sum problem. We assume that $\log^{1/2} n$ is an integer. Consider a data structure for the counting marked ancestor problem for a tree T with parameters as in (3), and update and query time t_u and t_q. The ith leaf v_i of T corresponds to x_i. We will maintain that the parity of the number of marked ancestors to v_i is the parity of the ith prefix sum in x, i.e.,

$$|\pi(v_i) \cap M| = x_1 + \cdots + x_i \pmod 2.$$

Thus the time for a partial sum query is the same as the time for a marked ancestor query, t_q. To maintain the invariant whenever x_i is changed (and thus the parity of all prefix sums $\geq i$ are changed), we change the marking of the root of a number of disjoint subtrees in T, whose leaves correspond to $x_i, \ldots, x_n$. These roots are the right siblings of $\pi(v_{i-1})$, so there are at most dh updates. Thus the update time is at most $t_u dh = O(2^{\log^{1/2} n} t_u \log^{1/2} n)$. Now Theorem 1 implies the bound on the query time. □

The proof of the next proposition contains the crucial application of nondeterminism to transform a counting problem into an existential one.

PROPOSITION 1. *Existential Marked Ancestor requires time $\Omega(\log^{1/2} n)$ per operation.*

Proof. Consider an algorithm for the existential problem with update time t_u and query time t_q, and let T be an instance of the counting marked ancestor problem. Construct 2^h new instances T_w indexed by bit strings $w \in \{0,1\}^h$. We maintain that the markings in the first instance $T_{00\ldots0}$ are the same as in T. In general, the ith bit of w is cleared iff the markings in T_w on level i are the same as in T. More precisely, if v is a node on level i, we have

$$(v \text{ marked in } T_w) = w_i \oplus (v \text{ marked in } T),$$

where $\oplus$ denotes exclusive or. The crucial observation is the following: Let v be a leaf. Then w is the characteristic vector of $\pi(v) \cap M$ iff the path $\pi(v)$ in T_w is unmarked.

Whenever a node in T is marked or unmarked, we must update all 2^h instances, so the update time is $2^h t_u$. For a query, we guess the characteristic vector of $\pi(v) \cap M$ and verify that $\pi(v)$ is unmarked in T_w. This takes time $t_q + O(1)$. We finish the proof by applying the above lemma. □

Finally, we present the application to range queries.

PROPOSITION 2. *Existential Range Query requires time $\Omega(\log^{1/2} n)$ per operation.*

Proof. Embed the tree from the marked ancestor problem in the first quadrant of the plane, with the root in the origin and the nodes at depth i spread out evenly on the diagonal $y = -x + d^h - d^{h-i}$. The query rectangle has its upper-right corner in the queried node. □

2.5. Discussion. Analyzing the above proof, we see that the algorithm used in the reduction actually solves the complement of the problem; we use it to verify $\pi(v) \cap M = \emptyset$. Thus the proof also yields a bound on the nondeterministic complexity of the *emptiness* problem (to return "yes" iff the query rectangle is empty). In other words, there is no short, maintainable witness to the absence of points in the plane.

In contrast, the emptiness problem in one dimension does admit a fast nondeterministic algorithm, since we can maintain a doubly linked list of the inserted points, and the query can guess both the immediate predecessor and immediate successor of a query interval and verify that they are neighbors in S. Using a Van Emde Boas tree, this can be implemented in time $O(\log \log n)$ per update and constant query time.

3. Proof of Theorem 1. 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 section 3.5.

3.1. Model of computation. The computational model is an extension of the cell-probe model [12, 30]; since there is only a single query in the hard sequence of operations constructed in our proof, 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 [15]. There are two types of nodes: *Read* nodes are 2^b-ary and labeled by a memory address, and computation proceeds to the child identified at that address; *write* nodes are unary and labeled 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 labeled by a memory address or by "$\exists$." The leaves are labeled 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 section 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 backward, so epoch 1 corresponds to the end of u. In general, the update string is an element in $U = U_d U_{d-1} \cdots U_1$, where $U_t = \{0, 1\}^{e(t)}$ and where $e(t)$ is the length of epoch t such that $e(t) + \cdots + e(1) = \lfloor n^{t/d}/d \rfloor$. The length of the entire update sequence is $\lfloor n/d \rfloor$. The size of d and hence the growth rate of $e(t)$ are given by

$$d = \left\lceil \frac{\log n}{\log(b t_u \log n)} \right\rceil. \tag{4}$$

The goal is to establish that $t_q \in \Omega(d)$.

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 leads 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 $T(i,u) = \bigcap \{\, T(i,u,w) \mid w \in A(i,u) \,\}$ 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 2. *If M^u and M^v differ only on cells with time stamp t, then $q_i M^u \neq q_i M^v$ implies $t \in T(i,u) \cup T(i,v)$.*

Proof. Suppose, on the contrary, that $q_i M^u \neq q_i M^v$ 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 $t \notin T(i,u)$ 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 $q^u = (q_1 M^u, q_2 M^u, \ldots, q_n M^u) \in \{0,1\}^n$. Update sequences that differ only in epoch t are called *t-different*.

LEMMA 3. *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 section 3.5. The proof itself is as in [15] and is provided in section 3.5 for completeness.

Write $U_{>t}$ for $U_d \cdots U_{t+1}$, the set of update sequences prior to epoch t, and write $U_{<t}$ for $U_{t-1} \cdots U_1$, 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 at least the average of $|T(i,u)|$ over choices of $i \in \{1,\ldots,n\}$ and $u \in U$, so

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

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

LEMMA 4. *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 $u \in xU_t y$ satisfy $\big|\{\, 1 \leq i \leq n \mid t \in T(i,u) \,\}\big| \geq \frac{1}{16}n$ if $t_{\mathrm{q}} = \mathrm{O}(\log n)$.*

Proof. Consider the set $V \subseteq xU_t y$ 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

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

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 iff 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 need only to analyze 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 $r = t_{\mathrm{u}} \cdot (e(t-1) + \cdots + e(1))$. 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 $\{0, \ldots, 2^b - 1\}$ is bounded by

$$(n2^{t_{\mathrm{q}} b} \cdot 2^b)^r \leq |U_t|^{o(1)}, \tag{5}$$

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

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

because $\frac{15}{16}n$ entries of each vector depend only on cells with time stamps other than t. On these cells, the memories are indistinguishable and therefore yield the same result by Lemma 2. By (6), all vectors from the same class end up in a Hamming ball of diameter $\frac{1}{8}n$, so Lemma 3 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

$$|U| n t_{\mathrm{q}} \geq \sum_{t=1}^{d} |U_{>t}| \cdot |U_{<t}| \cdot \tfrac{1}{16}n \cdot \tfrac{1}{2}|U_t| = \tfrac{1}{32}nd|U|$$

and hence $t_{\mathrm{q}} \geq \frac{1}{32}d$ as desired.

3.5. Update scheme. The technical part that remains is to exhibit a set of update sequences U satisfying Lemma 3. There are a number of ways to do this; the following construction is one which simultaneously anticipates our needs in section 6 and satisfies the balancing condition (2).

To alleviate notation, we assume that n/d is an integer. Consider the updates in epoch t, and index them as $u_1 \cdots u_{e(t)} \in U_t$. If $u_i = 0$, then nothing happens in the ith update. Else it performs $\mathit{update}(j, a)$, where the update position j is given below. The new value is $a = (-1)^r$, where $r = 1 + u_1 + \cdots + u_i \mod 2$, so 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:

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

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

$$\left\lfloor \frac{n/d}{e(t)} \right\rfloor. \tag{7}$$

In summary, the ith update in epoch t affects the letter in row t and the column given by $(i-1)\cdot\lfloor (n/d)/e(t)\rfloor + 1$.

This update scheme satisfies the statement in Lemma 3.

Proof of Lemma 3. Let xU_ty 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 contains 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 iff 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 (7). 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 1's in w', the Hamming distance between two query vectors is at least

$$|w'|_1 \cdot d \cdot \left\lfloor \frac{n/d}{e(t)} \right\rfloor \ge \tfrac{1}{2}|w'|_1 \cdot \frac{n}{e(t)}, \tag{8}$$

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

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 1's in w' to $\frac{1}{4}e(t)$ for large n. Hence the number of choices for w' is bounded by

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

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 from the initial instance 0^n. 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 sum of every x^t is 0 or 1 by construction. It can be checked that the balancing bound (2) holds at all times.

Another important feature of this update scheme, which we will use to prove Theorem 3, is that, if x and y result from t-different updates, then $x^r = y^r$ for $r \ne t$

and hence

$$\left| \sum_{j=1}^{i} x_j - \sum_{j=1}^{i} y_j \right| \le 1 \tag{10}$$

for all i.

4. Partial sum queries. The next result shows that the majority and equality problems defined in the introduction are just as hard as the parity query from [15]. The proof is a simple application of Theorem 1.

PROPOSITION 3. *The prefix equality and prefix majority problems satisfy*

$$t_{\mathrm{q}} = \Omega\left(\frac{\log n}{\log(t_{\mathrm{u}} b \log n)} \right).$$

Proof. We first give the proof for prefix equality. Let $d = \lceil \log n / \log(b t_{\mathrm{u}} \log n) \rceil$.

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 $y^{(0)}, \dots, y^{(d)}$ as

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

Let $t_{\mathrm{u}} = t_{\mathrm{u}}(n)$ denote the update time of our prefix equality algorithm. Whenever x is changed, we make at most $2d+2$ updates in the strings $y^{(t)}$; so the update time is $(2d+2) \cdot t_{\mathrm{u}}(n + 2d + 2)$.

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

$$\sum_{j=-2d}^{2i-1} y_j^{(t)} = d - t + i + \sum_{j=1}^{i} x_j, \qquad 0 \le t \le d, \quad 1 \le i \le n. \tag{11}$$

Hence, in order to find the ith prefix sum of x, our algorithm can nondeterministically guess the sum $s \in \{0, \dots, d\}$; we can assume from the balancing condition (2) in Theorem 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. The conclusion is by Theorem 1.

The same bound must hold for the majority problem since we can write

$$\begin{aligned} x_1 + \cdots + x_i = \left\lceil \tfrac{1}{2} i \right\rceil \text{ iff } & x_1 + \cdots + x_i \\ & \ge \left\lceil \tfrac{1}{2} i \right\rceil \wedge \bar{x}_1 + \cdots + \bar{x}_i \ge \left\lceil \tfrac{1}{2} i \right\rceil, \end{aligned}$$

where $\bar{x}_i = 1 - x_i$, and these negated values are easily maintained. □

To study this kind of problem in a general, let the *threshold* ϑ be an integer function such that $\vartheta(i) \in \{0, \dots, \lceil \frac{1}{2} i \rceil\}$. The query in the *prefix threshold problem for* ϑ is

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

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 4. *Let $t_u = t_u(n)$ and $t_q = t_q(n)$ denote the update and query time of any cell size b implementation of the prefix threshold problem for a nice threshold ϑ. Then $t_q = \Omega(\log \vartheta / \log(t_u b \log \vartheta))$.*

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

Let ϑ be a nice function, and let $p(1), \ldots, p(n)$ be such that $\vartheta(p(i)) = i$. Assume we have an algorithm for the prefix problem for ϑ with the parameters given in the statement of the theorem. We will construct an algorithm for the majority with instance $x \in \{0,1\}^n$. Construct a bit string y as

$$y = 0 \cdots 0 x_1 x_1 0 \cdots 0 x_2 x_2 0 \cdots 0 x_n x_n,$$

where the letters of x are at positions $p(1)-1, p(1), p(2)-1, p(2), \ldots, p(n)-1, p(n)$; denote the length of y by $m = p(n)$.

The string y can be maintained in time $2t_u(m)$ for each update of x. For the query, note that $2x_1 + \cdots + 2x_i = y_1 + \cdots + y_{p(i)}$, so

$$x_1 + \cdots + x_i \geq \left\lceil \tfrac{1}{2} i \right\rceil \quad \text{iff} \quad y_1 + \cdots + y_{p(i)} \geq i = \vartheta(p(i)),$$

so the majority function (left-hand side) can be expressed in terms of the threshold function ϑ (right-hand side). Hence the query time is $t_q(m)$. However, from the bound on the complexity of the majority function, we know that

$$t_q(m) = \Omega\left(\frac{\log n}{\log(t_u(m) b(m) \log n)} \right).$$

The stated bound follows by substituting $\vartheta(m)$ for n. □

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 $O((\log \vartheta / \log\log n) + \log\log n)$ per update (if $\vartheta(1), \ldots, \vartheta(n)$ can be computed in the preprocessing stage of the algorithm). The left term in the expression stems from a search tree, and the right term stems from a priority queue, which vanishes for cell size $b = \Omega(\log^2 n)$; details are omitted. A comparison with Proposition 4 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 $O(\log\log n)$.

We consider a more general class of query functions in section 6.2.

5. Applications to dynamic algorithms. Theorem 1 suggests a new approach for proving lower bounds for dynamic algorithms by employing nondeterminism in the reduction from signed partial sum. We demonstrate this with a number of examples in this section. 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 [19].

5.1. 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 a closing bracket or whitespace.

balance: return "yes" iff the brackets in x are properly nested.

This problem was studied in [11], where an algorithm with polylogarithmic update time is presented.

PROPOSITION 5. *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 consider an instance $x \in \{0,-1,+1\}^n$ 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 "$\sqcup$" stands for space. Let c be the string "$\sqcup($" consisting of a single space and an opening bracket. We maintain a balanced string of brackets uvw, where $u = c^{2n}$, $v = b_1 b_2 \dots b_n$, and $w =)^{n-s}\sqcup^s$, 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 $\sqcup$ of b_i and an opening square bracket on the $\sqcup$ 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 $x_1 + \cdots + x_i$. The conclusion is by Theorem 1. □

5.2. Dynamic graph algorithms. Our techniques improve the lower bounds of a number of well-studied graph problems considered in [19].

Tamassia and Preparata [28] present an algorithm for the class of *upward planar source-sink graphs* that runs in time $O(\log n)$ per operation. These digraphs 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 6. *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, Poutré, and Rauch [20] present an efficient algorithm for a version of this problem, and a strong lower bound is exhibited by Fredman and Henzinger [14]. Our lower bound also holds 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 as follows:

planar(u,v): return "yes" iff the graph remains upward planar after insertion of edge (u,v).

This problem was studied by Tamassia [27], who found an $O(\log n)$ upper bound.

PROPOSITION 7. *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 $q \in \mathbf{R}^2$.

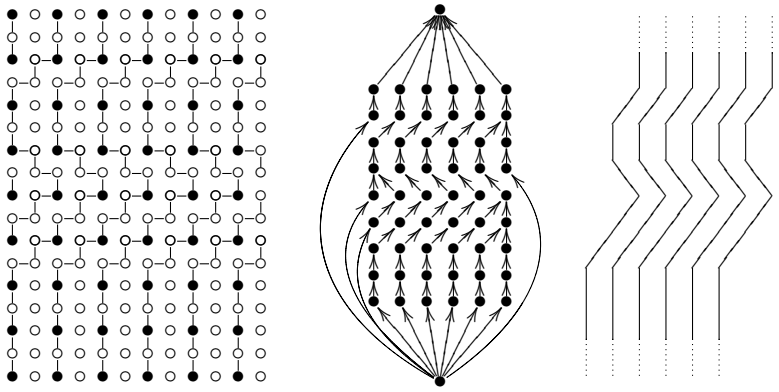

FIG. 1. *Planar graphs corresponding to* $x = (0, 0, +1, +1, -1, 0, +1, 0)$. *Left: Grid graph. Even grid points are marked* •*; odd grid points are marked* ○. *Middle: Upward planar source-sink graph. Right: Monotone planar subdivision.*

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 [26] 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 [5]. The lower bound for this problem in [19] 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" iff x is in the same polygon as the origin.

PROPOSITION 8. *Planar point location requires time* $\Omega(\log n / \log\log n)$ *per operation, even in monotone subdivisions.*

Traditionally, lower bounds in computational geometry are proved in an algebraic, comparison-based model (see [25] 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:

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" iff there is a path from x to y.

There are several well-known constructions that prove a lower bound for this problem [10, 14, 17, 23], but our proof translates to the other problems in Propositions 6–8. The details in these constructions are omitted; Figure 1 illustrates the structures arising in the reductions.

PROPOSITION 9. *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 Theorem 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 Figure 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 $(2s, 2j)$ iff $x_1 + \cdots + x_j$ 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 $O(w)$ 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 Theorem 1. □

The width of the hard graph above is logarithmic in the height, while the graphs constructed in [10, 14, 17, 23] 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 $O(\log\log n)$ by [4], 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 Theorem 1.

THEOREM 2. *Let* $d = O(\log n / \log(bt_u \log n))$ *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* $t_q = \Omega(d)$*. 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 parameterization can be done for all our problems.

PROPOSITION 10. *For every* $w = O(\log n / \log\log n)$*, dynamic reachability in grid graphs of width* w *requires time* $\Omega(w)$ *per operation.*

6. Refinement. We now take a somewhat subtler approach to our basic question than we take in section 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 $|s - \sum_{j=1}^{i} x_j| \le 1$ (for other values of s, the behavior 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 behavior of the query algorithm is undefined.)

The two problems are computationally equivalent.

THEOREM 3. *Let d be an integer function such that $d = \mathrm{O}(\log n/\log(t_{\mathrm{u}} b \log n))$. 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.*

6.1. Proof of Theorem 3. Most of the technical work for this result was already done in section 3.5, where we found that the instances resulting from two t-different updates have close prefix sums (10).

The query trees in our computational model are now deterministic decision trees as in [15]. However, 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 $(q_1^u M, \ldots, q_n^u M)$, 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 and compare this with the construction in section 3.3.

The next lemma corresponds to Lemma 2 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 5. *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,*

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

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 (10) 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., $q_i^{s+1} M^u = q_i^{s+1} M^v = q_i^s M^u = q_i^u M^u$. However, 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.

6.2. The partial sum problem for symmetric functions. Theorem 3 acts as an important ingredient in characterizing the dynamic complexity of all the *symmetric functions*, generalizing the results for the threshold functions of the last section. A Boolean function is *symmetric* if it depends only on the number of 1's 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 functions 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:

$$\begin{aligned} &\textit{update}(i)\text{: change } x_i \text{ to } \neg x_i,\\ &\textit{query}(i)\text{: return } f_i(x_1,\ldots,x_i). \end{aligned}$$

For example, taking f_i to be the parity function on i variables, we have the prefix parity problem of [15], and taking f_i to be the threshold function for $\vartheta(i)$, we have the problem from Proposition 4.

PROPOSITION 11. *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 $t_{\mathrm{q}} = \Omega(\log\vartheta/\log(t_{\mathrm{u}} b \log\vartheta))$.*

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 Proposition 3. 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 consider only 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 (11) in the proof of Proposition 3, if $g = x_1 + \cdots + x_i$, then $r_{-1}r_0r_{+1} = s_{d+i-1}s_{d+i}s_{d+i+1} = 001$. If instead $g-1$ is the correct sum, then $r_{-1}r_0 = 01$, and finally if $g+1$ is the correct sum, then $r_0r_{+1} = 00$. Hence these three cases for g can be distinguished by the above three queries, and they hence determine the correct answer for *refine*(i,g). The bound then follows from Theorem 3.

The rest of the proof is a padding argument that "stretches" the above to work for smaller ϑ similarly to the proof of Proposition 3. 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 [16, 8, 24].

Acknowledgments. The authors thank Arne Anderson, Gerth Stølting Brodal, Faith Fich, Peter Bro Miltersen, and Sven Skyum for valuable comments about various aspects of this work.

REFERENCES

[1] P. K. AGARWAL, *Range searching*, in Handbook of Discrete and Computational Geometry, J. E. Goodman and J. O'Rourke, eds., CRC Press, Boca Raton, FL, 1997, pp. 575–598.

[2] M. AJTAI, *A lower bound for finding predecessors in Yao's cell probe model*, Combinatorica, 8 (1988), pp. 235–247.

[3] S. ALSTRUP, T. HUSFELDT, AND T. RAUHE, *Marked ancestor problems*, in Proceedings of the 39th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1998, pp. 534–543.

[4] D. A. M. BARRINGTON, C.-J. LU, P. B. MILTERSEN, AND S. SKYUM, *Searching constant width mazes captures the AC^0-hierarchy*, in Proceedings of the Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 1373, Springer-Verlag, Berlin, 1998, pp. 73–83.

[5] H. BAUMGARTEN, H. JUNG, AND K. MEHLHORN, *Dynamic point location in general subdivisions*, in Proceedings of the 3rd Annual ACM–SIAM Symposium on Discrete Algorithms, ACM, New York, SIAM, Philadelphia, 1992, pp. 250–258.

[6] P. Beame and F. Fich, *Optimal bounds for the predecessor problem and related problems*, J. Comput. System Sci., 65 (2002), pp. 38–72.
[7] A. M. Ben-Amram and Z. Galil, *Lower bounds for data structure problems on RAMs*, in Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1991, pp. 622–631.
[8] B. Brustmann and I. Wegener, *The complexity of symmetric functions in bounded-depth circuits*, Inform. Process. Lett., 25 (1987), pp. 217–219.
[9] P. F. Dietz, *Optimal algorithms for list indexing and subset rank*, in Proceedings of the 1st Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 382, Springer-Verlag, Berlin, 1989, pp. 39–46.
[10] D. Eppstein, *Dynamic connectivity in digital images*, Inform. Process. Lett., 62 (1997), pp. 121–126.
[11] G. S. Frandsen, T. Husfeldt, P. B. Miltersen, T. Rauhe, and S. Skyum, *Dynamic algorithms for the Dyck languages*, in Proceedings of the 4th Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 955, Springer-Verlag, Berlin, 1995, pp. 98–108.
[12] M. L. Fredman, *Observations on the complexity of generating quasi-Gray codes*, SIAM J. Comput., 7 (1978), pp. 134–146.
[13] M. L. Fredman, *The complexity of maintaining an array and computing its partial sums*, J. ACM, 29 (1982), pp. 250–260.
[14] M. L. Fredman and M. R. Henzinger, *Lower bounds for fully dynamic connectivity problems in graphs*, Algorithmica, 22 (1998), pp. 351–362.
[15] M. L. Fredman and M. E. Saks, *The cell probe complexity of dynamic data structures*, in Proceedings of the 21st ACM Symposium on Theory of Computing, ACM, New York, 1989, pp. 345–354.
[16] J. T. Håstad, *Almost optimal lower bounds for small depth circuits*, in Proceedings of the 18th ACM Symposium on Theory of Computing, ACM, New York, 1986, pp. 6–20.
[17] T. Husfeldt, *Fully dynamic transitive closure in plane dags with one source and one sink*, in Proceedings of the 3rd European Symposium on Algorithms, Lecture Notes in Comput. Sci. 955, Springer-Verlag, Berlin, 1995, pp. 199–212.
[18] T. Husfeldt and T. Rauhe, *Hardness results for dynamic problems by extensions of Fredman and Saks' chronogram method*, in Proceedings of the 25th ICALP, Lecture Notes in Comput. Sci. 1443, Springer-Verlag, Berlin, 1998, pp. 67–78.
[19] T. Husfeldt, T. Rauhe, and S. Skyum, *Lower bounds for dynamic transitive closure, planar point location, and parentheses matching*, Nordic J. Comput., 3 (1996), pp. 323–336.
[20] G. F. Italiano, J. A. La Poutré, and M. H. Rauch, *Fully dynamic planarity testing in planar embedded graphs*, in Proceedings of the 1st Annual European Symposium on Algorithms, Lecture Notes in Comput. Sci. 726, Springer-Verlag, Berlin, 1993, pp. 212–223.
[21] P. B. Miltersen, *Lower bounds for union–split–find related problems on random access machines*, in Proceedings of the 26th ACM Symposium on Theory of Computing, ACM, New York, 1994, pp. 625–634.
[22] P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson, *On data structures and asymmetric communication complexity*, in Proceedings of the 27th ACM Symposium on Theory of Computing, ACM, New York, 1995, pp. 103–111.
[23] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia, *Complexity models for incremental computation*, Theoret. Comput. Sci., 130 (1994), pp. 203–236.
[24] S. Moran, *Generalized lower bounds derived from Hastad's main lemma*, Inform. Process. Lett., 25 (1987), pp. 383–388.
[25] F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer-Verlag, Berlin, 1985.
[26] F. P. Preparata and R. Tamassia, *Fully dynamic point location in a monotone subdivision*, SIAM J. Comput., 18 (1989), pp. 811–830.
[27] R. Tamassia, *On-line planar graph embedding*, J. Algorithms, 21 (1996), pp. 201–239.
[28] R. Tamassia and F. P. Preparata, *Dynamic maintenance of planar digraphs, with applications*, Algorithmica, 5 (1990), pp. 509–527.
[29] B. Xiao, *New Bounds in Cell Probe Model*, Doctoral dissertation, University of California San Diego, San Diego, CA, 1992.
[30] A. C. Yao, *Should tables be sorted?*, J. ACM, 28 (1981), pp. 615–628.
[31] A. C. Yao, *On the complexity of maintaining partial sums*, SIAM J. Comput., 14 (1985), pp. 277–288.