Algorithmica (2002) 32: 364–395
DOI: 10.1007/s00453-001-0079-6

Algorithmica
© 2002 Springer-Verlag New York Inc.

# Lower Bounds for Dynamic Data Structures on Algebraic RAMs[1]

A. M. Ben-Amram[2] and Z. Galil[3]

**Abstract.** In a seminal paper of 1989, Fredman and Saks proved lower bounds for some important data-structure problems in the cell probe model. This model assumes that data structures are stored in memory with a known word length. In this paper we consider random access machines (RAMs) that can add, subtract, compare, multiply and divide integer or real numbers, with no size limitation. These are referred to as *algebraic RAMs*. We prove new lower bounds for two important data-structure problems, *union-find* and *dynamic prefix sums*. To this end we apply the *generalized Fredman–Saks technique* introduced by the authors in a previous paper. The generalized technique relies on a certain well-defined function, *Output Variability*, that characterizes in some sense the power of the computational model. Fredman and Saks' work implied bounds on output variability for the cell probe model; in this paper we provide the first bounds for algebraic RAMs, and show that they suffice for proving tight lower bounds for useful problems.

An important feature of the problems we consider is that in a data structure of size $n$, the data stored are members of $\{0, \ldots, n\}$. This makes the derivation of lower bounds for such problems on a RAM without word-size limitations a particular challenge; previous RAM lower bounds we are aware of depend on the fact that the data for the computation can vary over a large domain.

**Key Words.** Random access machine, Cell-probe lower bounds, Union-find, Dynamic prefix sum.

## 1. Introduction.

In a seminal paper, Fredman and Saks [17] proved lower bounds for some important data-structure problems in the cell probe model: a model where the complexity of a computation is measured by the number of memory cells accessed, and memory cells have a specific word size (number of bits). A typical word size is logarithmic in the size of the problem instance. In particular, tight lower bounds were given on worst-case and amortized operation cost for the *union-find* problem and the *prefix sum* problem.

In a previous paper [5] we showed how their results can be derived from a main theorem which involves two quantities: *Problem Variability* (PV), a characteristic of the difficulty of a dynamic problem; and *Output Variability* (OV), a characteristic of the power of an abstract machine, or computational model. This separation means that by establishing OV for different computational models we can rather easily make use of

[1] A preliminary exposition of this work was presented at the 32nd FOCS conference [3] and most of it originates from the first author's Ph.D. thesis [6]. The work by the first author was partially supported by a Charles Clore fellowship. The second author's work was partially supported by NSF Grant CCR-90-14605 and CISE Institutional Infrastructure Grant CDA-90-24735.

[2] Tel-Aviv Academic College, 4 Antokolski Street, 64044 Tel-Aviv, Israel. amirben@server.mta.ac.il.

[3] Computer Science, Room 450, Columbia University, New York, NY 10027, USA. galil@cs.columbia.edu.

Received October 17, 1999; revised April 26, 2000. Communicated by M.-Y. Kao.
Online publication October 12, 2001.

problem analysis done once. In fact, what we actually use is an upper bound on OV. To clarify this idea, consider the following examples. The number of leaves of a decision tree of height $h$ is bounded by $2^h$; this simple upper bound on the "power" of the tree model is the key to standard lower bound results such as $\Omega(n \log n)$ for sorting. A more involved upper bound is that given by Ben-Or [7] on the number of connected components of a set recognized by an *algebraic computation tree*. This too is an upper bound that implies lower bounds in much the same way as the sorting result. The definition of OV (given in Section 2) is a little more complicated as it is geared towards the handling of dynamic problems.

Attempts to extend classical lower bounds from decision or computation trees to Random Access Machines (RAMs) had to deal with the RAM's capability of *indirect addressing,* the capability that makes it a "random access" machine. This powerful feature enables fast solutions to certain problems, for example, sorting integers of a limited range with bucket sort [1]. A seminal paper that showed how to cope with indirect addressing and its combination with computational instructions is [27]. The method relies on the fact that in a large set of possible inputs, there will be some that do not satisfy a given non-trivial equation. Thus a worst-case input can be chosen that defies non-trivial use of indirect addressing. This approach has been later used in several works, e.g., [9], [12], [24], [26], and [6]. The analysis in this paper requires a further refinement of the technique because the inputs and outputs in our type of problems belong to a small set (say $\{1, \ldots, n\}$) and the algorithm might exploit it.

The main result of this paper is an upper bound on OV for the algebraic RAM, an idealized random access machine with capabilities for computation with real numbers. This model is strictly stronger than the classical RAM model with unbounded integers [1], [10], but is incomparable with the cell probe model, since the latter has a bounded word size but an unrestricted "instruction set."

Interestingly, we obtain a bound on the RAM's OV that is quite close to the bound known for cell probe. Plugging this result into the previous lower-bound arguments, we obtain the first tight lower bounds for the complexity of the above data-structure problems on the algebraic RAM. More specifically, we prove an $\Omega(\alpha(m, n))$ lower bound on the amortized cost of $n - 1$ unions and $m$ finds; an $\Omega(\log n / \log\log n)$ worst-case lower bound for a single union or find; and an $\Omega(\log n / \log\log m)$ amortized lower bound for $m$ prefix sum operations on an array of length $n$. For more details and additional results, including tradeoff relations between the costs of *union* and *find*, see the following sections.

This paper begins with definitions of PV and OV and the presentation of the Main Theorem from [5]. We then define our RAM models and state the results on their OV. The following sections include the lower-bound proofs for the above problems. We include all the definitions and attempt to make this paper self-contained, except for not repeating the proof of any statement that we can simply cite from [5]. In fact it is the very merit of our method that we can re-use much of the earlier work (a large part of that even goes back to [17], but Fredman and Saks' presentation was not fitted to our framework, so it was remolded thus in [5]).

The derivation of bounds on RAMs' OV, which is the main result in a technical sense, is given last. Thus, a reader can find out how these bounds are used before delving into the details of their proof.

**2. Output Variability and Problem Variability.** Consider a *query program* that is given an input number in the range $1 \cdots n$ and, using a data structure kept in memory, produces an answer in the range $1 \cdots m$. The significance of these numbers is problem dependent; in fact, any range of $m$ consecutive integers may replace the one above, and any set of size $n$ can be used as query inputs. For notational convenience, we use the above simple sets.

The query program can be said to have two sources of input: external (here a single number) and internal (the data structure stored in memory). We consider the memory to consist of a (possibly infinite) set of *cells*. A *memory image* $M$ is a sequence of integers $M(0), M(1), \ldots$ which specifies the contents of the cells. Let $Q(i, M)$ denote the result of a query with input $i$ on the memory image $M$. The vector of results $(Q(1, M), \ldots, Q(n, M))$ will be denoted by $\bar{Q}(M)$.

OV is a characteristic of the efficiency in which a computational model supports dynamic data structures. The definition involves two different cost measures associated with programs for data-structure problems on the given model. The first is the *data-structure cost* $w$, which we leave unspecified for the moment. A typical choice would be the amount of memory it uses. The second is the *query cost* $q$. Query cost is required to be at least the number of memory cells read during the processing of the query, but may account for additional processing costs as well.

The OV of a model is the function $OV(w, x, q)$ defined as follows. Let $M$ be a memory image that contains a data structure whose cost is $w$. Let $M^x$ be the set of memory images obtained from $M$ by modifying the contents of at most $x$ cells. Let $Q$ be a query program such that for all inputs $i$ and memory images $N \in M^x$ the computation of $Q(i, N)$ costs at most $q$; the definition of query cost is also model dependent, but is required to be at least the number of memory cells read while processing it. More generally, we consider a $q$-truncated program: this is a program $Q$ obtained from a real query program $Q'$ by forcing it to halt and output a "dummy" answer (say, 1) whenever its run would originally cost more than $q$. $OV(w, x, q)$ is the supremum, over all such memory images $M$ and programs $Q$, of the cardinality of the set of vectors $\{\bar{Q}(N) \mid N \in M^x\}$.

For a simple example of how OV can be bounded for a useful model we refer the reader to [5], where the cell probe model is considered. A slightly more involved, and more precise, bound on OV for the cell probe model can be found in [2].

We next define PV. This definition involves both queries and updates; we denote by $U$ the set of possible inputs to an update operation, assumed finite.

An *update scheme* $\mathcal{U}$ is defined as a set of sequences of update operations which are subdivided into $z$ *rounds* of lengths $r_1, r_2, \ldots, r_z$. Formally, $\mathcal{U} \subseteq U^{r_1+\cdots+r_z}$. We denote by $\mathcal{U}_j$ the set obtained by taking the first $j$ rounds of each sequence in $\mathcal{U}$:

$$\mathcal{U}_j = \{\sigma \in U^{r_1+\cdots+r_j} \mid \exists\mu\colon\ \sigma\mu \in \mathcal{U}\}.$$

Thus $\mathcal{U}$ is the same as $\mathcal{U}_z$. For obtaining our results, we require $\mathcal{U}$ to obey the *equipartition property*: for each member of $\mathcal{U}_j$ the number of its continuations ($\mu$ above) is the same. This holds, in particular, if the operations of each round are chosen independently. Here is an example: let $U = \{a, b, c\}$, $r_1 = r_2 = r_3 = 2$ and $r_4 = r_5 = r_6 = 1$; thus $z \geq 6$.

A possible scheme is represented by the following expression:

$$\mathcal{U}_6 = \begin{Bmatrix} ab \\ ac \end{Bmatrix} \begin{Bmatrix} aa \\ bb \end{Bmatrix} \{cb\} \begin{Bmatrix} a \\ b \\ c \end{Bmatrix} \{a\} \begin{Bmatrix} b \\ c \end{Bmatrix}.$$

We have $|\mathcal{U}_3| = 4$, $|\mathcal{U}_6| = 24$.

Let $0 < i \leq j$ be round numbers. For each sequence of operations $\sigma \in \mathcal{U}_j$ define $A(\sigma)$ to be the vector of correct answers to queries on $i = 1, \ldots, n$ following the execution of $\sigma$. For each $\tau \in \mathcal{U}_{i-1}$ we define $C_{i,j}(\tau)$ to be the set of continuations of $\tau$ to a sequence in $\mathcal{U}_j$:

$$C_{i,j}(\tau) = \{\mu \colon \tau\mu \in \mathcal{U}_j\}.$$

Referring to the above example,

$$C_{4,6}(abbbcb) = \begin{Bmatrix} a \\ b \\ c \end{Bmatrix} \{a\} \begin{Bmatrix} b \\ c \end{Bmatrix}.$$

Note that the equipartition property implies that $|C_{i,j}(\tau)|$ is the same for all $\tau$. For defining PV we need just one more notation. For an arbitrary vector $v \in \mathbb{Z}^n$, let $B_d(v)$ denote the ball of radius $d$ (with respect to Hamming distance) centered at the vector $v$. We define

$$g_\delta(i, j, \tau) = \max_{v \in \mathbb{Z}^n} \frac{|\{\mu \in C_{i,j}(\tau) \colon A(\tau\mu) \in B_{\delta n}(v)\}|}{|C_{i,j}(\tau)|},$$

thus $g_\delta(i, j, \tau)$ indicates the fraction of answer vectors that may be "close" to an arbitrary vector $v$. We define $PV_{\mathcal{U},\delta}(i, j)$ by

$$(PV_{\mathcal{U},\delta}(i, j))^{-1} = \frac{1}{|\mathcal{U}_{i-1}|} \sum_{\tau \in \mathcal{U}_{i-1}} g_\delta(i, j, \tau).$$

A large value of PV indicates that $g_\delta$ is often small; hence the set of answer vectors following a sequence from $\mathcal{U}_j$ is sparse.

**3. The Main Theorem and Some Corollaries.** The Main Theorem of [5] states a connection between PV, OV and the complexity of solving the given problem in the given computational model.

The theorem refers to a set of operation sequences which include both updates and queries. These sequences are obtained by enriching an update scheme (as described in the last section) with query operations. We further associate with these sequences an *epoch scheme*, that defines a subdivision of each sequence into time-intervals called epochs.

Let $\mathcal{U}$ be a given update scheme, and let $k \leq z$ be a round number. A *pattern* for $\mathcal{U}_k$ is a string $\pi_k$ of $k + f$ letters, $k$ $\mathsf{u}$'s and $f$ $\mathsf{q}$'s, for some $f \geq 0$. According to the pattern, $|\mathcal{U}_k| \cdot n^f$ operation sequences are formed by assigning the update rounds specified by $\mathcal{U}_k$ to the positions marked by $\mathsf{u}$, and a single query operation, ranging over the $n$ possible queries, to each position marked by q. We denote the set of operation sequences obtained by $\Sigma_k = \Sigma(\mathcal{U}, \pi_k)$.

Let $\pi_k$ be as above and let $j \leq k$. The subpattern $\pi_j$ is defined as the prefix of $\pi_k$ that extends up to, but not including, the $(j+1)$st u (if $j = k$, then $\pi_j = \pi_k$). An *epoch scheme* $\mathcal{E}$ for $\mathcal{U}_j$ is defined by subdividing the pattern string into *epochs*. The subdivision is defined by a set of indices $1 = j_1 < j_2 < \cdots \leq j$. Epoch $e$ consists of the operations ranging from the $j_e$th u up to (but not including) the $j_{e+1}$st (or the end of the string). Thus each epoch may contain both u's (update rounds) and q's (queries), but all start with an update round. This subdivision naturally induces a subdivision on every sequence in $\Sigma_j$. Here is an example (the vertical lines delimit epochs):

$$\pi_6 = \mathsf{u|uuq|uuquq|}, \qquad j_1 = 1, \quad j_2 = 2, \quad j_3 = 4, \quad j = 6.$$

This pattern can be associated with the example for $\mathcal{U}_6$ presented in the previous section. Taking the set of query inputs to be $\{1, 2\}$ each q can be assigned two values. We obtain $|\Sigma_6| = 24 \cdot 2^3 = 192$. One particular sequence in $\Sigma_6$ (in fact, the first in lexicographic order) is $ab|aacb1|aa1b1|$. The vertical lines delimit epochs, induced by epochs of the pattern.

MAIN THEOREM. *Fix a data-structure problem and a model of computation. Consider an update scheme $\mathcal{U}$, an epoch scheme $\mathcal{E}$ for $\mathcal{U}_j$ that includes $q$ epochs, constants $\delta > 0$ and $c < 1$ and parameters $x_e$ and $w$ such that for all epochs $e$,*

$$OV(w, x_e, q) \leq cPV_{\mathcal{U},\delta}(j_e, j).$$

*Assume that there is a constant $d$, $c < d \leq 1$, such that at least a fraction $d$ of the pairs $(\sigma, e) \in \Sigma_j \times \{1, \ldots, q\}$ satisfy the following conditions*: (i) *Throughout $\sigma$, the cost of the data structure is bounded by $w$.* (ii) *At most $x_e$ memory cells are written subsequent to epoch $e$. Then $\delta(d - c)q$ is a lower bound on the average cost of a query that follows an operation sequence chosen randomly from $\Sigma_j$.*

The theorem establishes a tradeoff between the cost of maintaining the data structure ($w$ and $x_e$) and the cost of querying it ($q$). Such a tradeoff leads in particular to a lower bound on operation cost, which is obtained as follows. The quantities $w$ and $x_e$ can be bounded if the total cost of a sequence of operations is known. Hence, every assumption of the form "all operations cost at most $q$" leads to a bound on $OV(w, x_e, q)$. To derive the lower bound, we compute a cost $q$ such that if all operations cost less than $q$, $OV(w, x_e, q)$ will be small enough to satisfy the theorem and it will follow that the query must cost at least $q$, a contradiction.

While the cost measures referred to in the theorem are worst-case costs, it can also be used to prove lower bounds on average and amortized complexity as well as for randomized algorithms. A general discussion of how such results are obtained appears in [5]. Since the application of the Main Theorem to the specific problems we consider required some sophisticated arguments (all due to [17]), we formulated in [5] some lemmas, or rather corollaries of the Main Theorem, tailored towards the specific applications. We now restate these corollaries without repeating their proofs.

The first corollary is used in proving inverse-Ackermann lower bounds, and is used in Section 5 in conjunction with the Union-Find problem. For $i \geq 1$ and $j \geq 0$, we define the Ackermann function $A(i, j)$ by

$$\begin{aligned} A(i,0) &= 2 && \text{for} \quad i > 1; \\ A(1,j) &= 2^j && \text{for} \quad j \geq 0; \\ A(i,j) &= A(i-1, A(i, j-1)) && \text{for} \quad i > 1, \quad j \geq 1. \end{aligned}$$

Let

$$\alpha_k(n) = \min\{j \mid A(k,j) > n\},$$

$$\alpha(m,n) = \min\left\{k \mid A\left(k, \left\lceil \frac{m}{n} \right\rceil\right) > \log n\right\}.$$

COROLLARY 1. *Consider models where the cost of a data structure is bounded by the cost of the operation sequence that created it. Assume that there are an update scheme $\mathcal{U}$ which includes at most $n$ operations spanning $g(n) = \frac{1}{2}\log n$ rounds, constants $K, \delta > 0$ and $c < 1$ such that for all $i < j$,*

$$cPV_{\mathcal{U},\delta}(i,j) \geq OV\left(ng(n), \frac{n}{Kq2^i}, q\right),$$

*where $q = \alpha(m,n)$. Then there are operation sequences made of an update sequence from $\mathcal{U}$ interspersed with $m$ queries whose total cost is $\Omega(m\alpha(m,n))$.*

*If the inequality holds for a fixed $q$, then there are sequences with $n$ queries such that either the queries cost at least $qn$ or the updates cost $\Omega(n\alpha_{q+1}(n))$*

REMARK. The assumption made in the lemma regarding data-structure cost essentially expresses the idea that before the operation sequence is begun, there are no data in memory. This holds for our RAM models by virtue of the assumption that the memory is initially zero.

The second corollary is used in Section 6 in conjunction with the Prefix Sum problem.

Here we restrict our update scheme to having the same number of update operations in each round, say $r$, and the same number of possible choices for each operation, say $h$. Thus $|\mathcal{U}_z| = h^{rz}$.

We consider "easy" cases concerning PV computation, in which there is a uniform bound on the fractions $g_\delta(i, j, \tau)$, in the following sense:

DEFINITION. Update scheme $\mathcal{U}$ is $(V, \delta)$-*bounded* if $V$ is a function that satisfies, for all $1 \leq i \leq j \leq z$ and $\tau \in \mathcal{U}_{i-1}$,

$$g_\delta(i, j, \tau) \leq V(j - i + 1). \tag{1}$$

Recall that an epoch scheme spans a given number $j$ of rounds, for some $j \leq z$. We define $L_e = j - j_e + 1$, the number of rounds since the beginning of epoch $e$. Consider a pattern uquq . . . uq, including $z$ u's and $z$ q's. We denote by $\Sigma_z$ the set of update/query sequences corresponding to this pattern in the usual manner.

COROLLARY 2. *Let $\mathcal{U}$ be $(V, \delta)$-bounded and consist of z rounds. Let $\mathcal{E}$ be epoch scheme for $\mathcal{U}_j$, $j \leq z/2$, that includes q epochs. Consider a constant* $0 < c < 1/2$ *and parameters $w$, $x$ such that*

$$OV(w, 4xL_{e+1}/z, q) \cdot V(L_e) \leq c$$

*for all $e < q$. Assume that for a random choice of $\sigma \in \Sigma_z$, the following conditions hold together with probability at least $p > 2c$*: (i) *throughout $\sigma$, the cost of the data structure is bounded by $w$*; (ii) *the number of memory writes throughout $\sigma$ is bounded by $x$. Then $\delta(p/4 - c/2)zq$ is a lower bound on the average total cost of queries in $\sigma$. In particular, we obtain a lower bound of $\Omega(q)$ on the amortized cost of a query.*

**4. Random Access Machines.** The RAM is widely adopted as a model for studying the complexity of algorithms in a quite realistic way. It has been popularized by textbooks such as [1] and [28]. The former defined the elementary data type to be the integers; the latter, the real numbers. In addition to these changes of *data type*, we also find several variants of *instruction sets*. In this section we present a general notation for RAM models, following [6], and use it to specify the variants under consideration precisely.

All the RAMs we define share the following structure. The machine consists of a processing unit and a memory unit. The processing unit runs the program; to this end it contains a "program counter" that indicates the next instruction to be executed (the different instructions are described below). It also makes use of a finite set of *operating registers* $r_1, \ldots, r_k$, whose number is fixed for any given program, as they can only be accessed by being named in an instruction. These registers are used for all arithmetics and tests, leaving the *memory* with the sole role of data storage.

This description is made specific by the choice of three parameters:

The *domain* $\mathcal{D}$ is the set of values that may be manipulated by the machine as "units of data." Every memory cell or operating register holds one element of $\mathcal{D}$.

The *address space* $\mathcal{A}$ is the set of values that may be used as memory addresses. Thus the size of memory is $|\mathcal{A}|$, and the standard idealized model uses $\mathcal{A} = \mathbb{N}$. In the case that $\mathcal{A}$ is strictly contained in $\mathcal{D}$, a program may *fault* by attempting to use a value in $\mathcal{D} \backslash \mathcal{A}$ as an address. In this case the program may be considered invalid, or we may consider its result to be $\perp$ (undefined).

The *set of primitive functions* $\mathcal{F}$ defines the basic operations on data values. We call the RAM *algebraic* if $\mathcal{F}$ consists of the field operations $\{+, -, \times, /\}$.

The instruction set of the RAM contains the following groups. In the notation for instructions, $r_i, r_j, r_k$ are register names; $x$ is a constant from $\mathcal{D}$. The notation $\langle r_i \rangle$ refers to the memory cell whose address is given by the contents of $r_i$.

*Direct Assignments*:

$$r_i \leftarrow x,$$

$$r_i \leftarrow r_j.$$

*Memory Access*:

$$\begin{aligned} \langle r_i \rangle &\leftarrow r_j, \\ r_j &\leftarrow \langle r_i \rangle. \end{aligned}$$

*Flow control instructions*:

goto *label*
if $r_i \bowtie r_j$ goto *label*
$\bowtie \in \{=, <, \leq, \ldots\}$
halt

*Arithmetic Instructions*:

$$\begin{aligned} r_i &\leftarrow r_j + r_k, \\ r_i &\leftarrow r_j - r_k, \end{aligned}$$

etc., as provided by $\mathcal{F}$.

The initial contents of memory cells, before written into by a program, is assumed in this paper to be zero.

Our results in this paper are given for a *real-number RAM*, namely a RAM where $\mathcal{D} = \Re$. The main application for this model is to study algorithms on real numbers, e.g., in computational geometry [28]; but it is conceivable that a problem on integer inputs will be solved faster using the power of the real-number RAM. Thus, we consider the real-number RAM also in conjunction with integer-constrained problems. At any rate, it is obviously valid to use a strong model for a lower-bound proof. In the same spirit, we also adopt the rather non-standard choice $\mathcal{A} = \Re$ (memory cells are addressed with real numbers).

We consider three RAM variants that differ on the primitive set $\mathcal{F}$. These models are: RAM($\pm$), with $\mathcal{F} = \{+, -\}$; RAM($\times$), with $\mathcal{F} = \{+, -, \times\}$; and RAM($/$), with $\mathcal{F} = \{+, -, \times, /\}$.

The following result is proved in Section 8.

THEOREM 1. *The models RAM($\pm$), RAM($\times$) and RAM($/$) satisfy*

$$OV(w, x, q) \leq (2^q dnw)^{3x} (dn)^x$$

*for $w, x > 1$, where $d$ is $1$ for RAM($\pm$) and $2^q$ for RAM($\times$) and RAM($/$).*

## 5. The Union-Find Problem.

We consider the union-find problem [30], [19] in the following framework: we start with $n$ singleton sets $\{1\}, \ldots, \{n\}$, which are named $1, \ldots, n$, respectively. $n - 1$ *union* operations and $m$ *find*s are to be performed. $n$ is assumed, without loss of generality, to be an even power of two. Each union operation specifies the names of the sets to unite and a name for the resulting set, all integers are in the range $1, 2, \ldots, 2n$. This range allows us to give a new name to each created set: we always give the name $n + i$ to the result of the $i$th union. A query (find) specifies an element and returns the name of the set currently containing it.

Our *update scheme* $\mathcal{U}$ for this problem contains $1/2 \log n$ rounds. Round $k$ comprises $n/2^k$ operations which pair sets of size $2^{k-1}$ to a set of size $2^k$. Thus, the number of sets at the beginning of round $k$ is $n[k] = n/2^{k-1}$. By our naming convention, the names of these $n[k]$ sets will span a pre-determined range of $n[k]$ consecutive integers. Therefore the union operations of each round can be specified independently of former as well as of later rounds. The update scheme contains all possible sequences of such operations; this guarantees the equipartition property.

LEMMA 1 [5]. $PV_{\mathcal{U},1/4}(i, j) \geq 8^{-n[i]}(n[j+1])^{n[i]/2} \geq 8^{-n[i]}n^{n[i]/4}$.

THEOREM 2. *If a RAM(/) algorithm for union-find executes each union in cost bounded by $k$, $7 \leq k \leq n$, then there are sequences of union operations such that the cost of a subsequent find is $\Omega(\log_k n)$. Hence, for every algorithm, at least one of the operations has a worst-case cost of $\Omega(\log n/\log\log n)$.*

PROOF. We use the above update scheme of $\frac{1}{2}\log n$ rounds, dividing it into $q$ epochs where

$$q = \frac{\log n}{6\log k}.$$

Each epoch contains $\beta = \lceil(\frac{1}{2}\log n)/q\rceil = \lceil 3\log k\rceil$ rounds (except the last which may be shorter). We add no queries in between since we are only interested in the cost of a single query following the updates, as given by the Main Theorem. Define

$$\begin{aligned} q &= \frac{\log n}{6\log k}, \\ x_e &= \frac{n[j_e]}{k^2}, \\ w &= nk, \end{aligned}$$

$$\delta = \tfrac{1}{4}, \qquad c = \tfrac{1}{2}, \qquad d = 1.$$

We will prove that the above epoch scheme and parameter definitions satisfy the conditions of the Main Theorem. Note that $d = 1$ means that all the sequences considered must respect the bounds $w$ and $x_e$ we give. Since we look for a worst-case lower bound, we *assume* that no update operation in the sequences considered writes more than $k$ cells (otherwise the conclusion of the theorem holds). This implies that the number of cells written throughout a sequence of updates is bounded by $w = nk$. To estimate the number of cells written subsequent to epoch $e$, recall that round $i$ includes $n[i]/2 = n/2^i$ operations, and that epoch $e$ extends from round $j_e = (e-1)\beta + 1$ up to round $e\beta$. Furthermore, each operation costs at most $q$ and therefore writes at most $q$ cells. It follows that an upper bound on the number of cells written is

$$\sum_{i=e\beta+1}^{(1/2)\log n} \frac{n}{2^i}k < \frac{n}{2^{e\beta}}k = \frac{n[j_e]}{2^\beta}k = \frac{n[j_e]}{2^{\lceil 3\log k\rceil}}k \leq \frac{n[j_e]}{k^2} = x_e.$$

Theorem 1 yields

$$OV(w, x_e, q) \le (2^{2q} n^2 k)^{3x_e} (2^q n)^{x_e} \le (2^q n)^{7n[j_e]/k^2} k^{n[j_e]/k^2}$$

and, using the facts $q \le \log n/6$ and $k^{1/k^2} < 2$,

$$OV(w, x_e, q) < n^{9n[j_e]/k^2} 2^{n[j_e]}.$$

Combining with Lemma 1,

$$\frac{OV(w, x_e, q)}{PV_{\mathcal{U},\delta}(j_e, j)} < n^{9n[j_e]/k^2} 2^{n[j_e]} 8^{n[j_e]} n^{-n[j_e]/4} = (16 n^{(9/k^2)-(1/4)})^{n[j_e]}.$$

For $k \ge 7$, this expression tends to zero as $n \to \infty$, so that for $n$ large enough it becomes less than $c$. We have established all the conditions of the Main Theorem, which now shows that the average cost of a subsequent query is at least $\delta(d - c)q = q/8 = \Omega(\log_k n)$. □

We remark that the $\Omega(\log n/\log\log n)$ lower bound has been proved, and shown tight, by Blum [8] for a certain class of pointer algorithms. Smid [29] modified Blum's pointer algorithm to match the above tradeoff for any value of $k$. Since these pointer algorithms can be efficiently implemented on an ordinary integer RAM, we conclude that Theorem 2 is optimal.

THEOREM 3. *Any RAM(/) algorithm for solving the union-find problem requires $\Omega(m\alpha(m, n))$ time, in the worst case, to execute a sequence of $n - 1$ unions and $m$ finds. Moreover, for any fixed $q$, there are sequences of $n - 1$ unions and $n$ finds such that either the unions cost at least $qn$ or the finds cost $\Omega(n\alpha_{q+1}(n))$.*

PROOF. We apply Corollary 1 (Section 3) as follows: $\mathcal{U}$ is the update scheme already described. We choose $\delta = \frac{1}{4}$ so Lemma 1 applies. Further define $c = \frac{1}{2}$ and $K = 25$. The results follow from Corollary 1 provided

$$c\, PV_{\mathcal{U},\delta}(i, j) \ge OV\left(ng(n), \frac{n}{Kq2^i}, q\right).$$

Substituting the parameters we chose and using Theorem 1,

$$\begin{aligned} OV\left(ng(n), \frac{n}{Kq2^i}, q\right) &= OV\left(\tfrac{1}{2} n \log n, \frac{n}{50q2^{i-1}}, q\right) \\ &\le 2^{0.14n[i]} n^{0.14n[i]/q} (\log n)^{0.06n[i]/q} \end{aligned}$$

(recall $n[k] = n/2^{k-1}$). By Lemma 1,

$$\begin{aligned} \frac{OV(ng(n), n/Kq2^i, q)}{PV_{\mathcal{U},\delta}(i, j)} &\le 2^{0.14n[i]} n^{0.14n[i]/q} (\log n)^{0.06n[i]/q} 8^{n[i]} n^{-0.25n[i]} \\ &= \left((\log n)^{0.06/q} 2^{3.14} n^{(0.14/q - 0.25)}\right)^{n[i]}, \end{aligned}$$

this tends to zero as $n \to \infty$ and in particular becomes less than $c$ when $n$ is large enough. Lemma 1 thus applies and yields the lower bounds. □

Optimality of Theorem 3 is shown by matching upper bounds by Tarjan [30] and La Poutré [25] (the latter also gives a tradeoff solution). Both papers give pointer algorithms that can be readily implemented on an integer RAM. Recently, Alstrup et al. [2] have given algorithms that are at the same time worst-case optimal (matching Theorem 2) and amortized-time optimal (matching Theorem 3).

We finally remark that both lower bounds still hold if we average on the set of inputs for each find, and apply to the expected time if the algorithm is randomized. These observations are common to results proved by the Fredman–Saks technique and are justified in [5].

## 6. The Prefix Sum Problem.

The *prefix sum* problem is a basic and simple data-structure problem that has been described by Fredman [15] as "a toy problem which is both tractable and surprisingly interesting." We define the problem as follows:

PREFIX SUMS *mod* $k$. We represent an array $T[1], \ldots, T[n]$ of integers. Initially all $T[i]$ are zero. The update operation is add$(i, \Delta)$ which implements $T[i] \leftarrow T[i] + \Delta$, and the query is sum$(j)$ which returns $\sum_{i \le j} T[i] (\text{mod}\, k)$.

The above problem is denoted by PS$(n, k, M)$ if $\Delta$ in update operations is guaranteed to be bounded by $M$. The unrestricted prefix sum problem is PS$(n, M, M)$ where $M$ is greater than any number that is ever to be represented. The simplest variant is PS$(n, 2, 2)$, also called *prefix parity* since in essence we ask for the parity of a prefix of the array.

Not only does the prefix sum problem occur in many applications, a lower bound for PS$(n, 2, 2)$ is useful for deducing lower bounds for other, seemingly different, problems [16], [17], [22]. Naturally, the lower bound also applies to PS$(n, k, M)$ with larger $k$ and $M$.

Fredman and Saks [17] gave a lower bound of $\Omega(\log n / \log b)$ for the amortized cost of PS$(n, 2, 2)$ in the cell probe model with word size $b \ge \log n$. This bound is re-proved using our framework in [5]. By a simple extension of an algorithm by Dietz [11], a matching upper bound (in fact, a worst-case bound) can be given for all $b \ge \log n$. This algorithm uses certain functions which on a RAM are implemented via tables; preparing the tables takes $O(2^{b^\varepsilon})$ time for a certain $0 < \varepsilon < 1$. Besides this function, ordinary arithmetics on words of $b$ bits are used. Since our RAM can use unbounded integers, we can choose the value of $b$ that fits us best. This will depend on the number of operations to be performed, since we would like the cost of building the tables to amortize over these operations. Specifically, suppose that the number of operations to be performed is given as $m \ge n$. The program chooses $b \approx \log m$. Thus the tables can be built in $O(2^{b^\varepsilon}) = O(m)$ time. The RAM program builds the tables and then proceeds with Dietz's algorithm; we obtain an upper bound of $O(\log n / \log\log m)$ amortized time per operation. The space used is $\Theta(2^{b^\varepsilon} \cdot b^\varepsilon + n)$. If $m$ is not known in advance, we start with $b = \log n$ and increase $b$ once in a while. Since this is a well-known technique we omit the details.

What happens if we do not want the space to keep growing as the number of operations grows? We can impose a bound on the amount of memory that we are going to use and select the value of $b$ accordingly. If we bound the memory by $2^{(\log n)^\kappa}$, for a constant $\kappa$, the value of $b$ that we may use becomes polylogarithmic in $n$. Dietz's algorithm will then run in $\Omega(\log n/\log\log n)$ time per operation. The next theorem provides matching lower bounds:

THEOREM 4. *Any RAM(/) algorithm for* $\mathrm{PS}(n, 2, 2)$ *requires, in the worst case,* $\Omega(m \log n/\log\log m)$ *time for executing a sequence of* $m \geq n$ *update and query operations. If the space used by the program is bounded by* $2^{(\log n)^\kappa}$*, for a constant* $\kappa$*, then* $\Omega(m \log n/\log\log n)$ *time is required for all* $m \geq n$.

As mentioned in the last section, the lower bounds hold even if we average on the set of inputs for each query and if the algorithm is randomized. Furthermore, the update scheme used in our proof consists of operations add$(i, \Delta)$ where the sequence of indices $i$ is fixed in advance. Thus knowledge of this sequence does not make the problem easier (on the models we consider).

We remark that for $m < n$, our proof method yields a lower bound of $\Omega(\log m/\log\log n)$ amortized time. This may be matched on the cell probe model by an algorithm that makes use of the fact that at most $m$ distinct indices appear in update operations. Whether this can be matched on the algebraic RAM is an open problem.

We prove Theorem 4 in the unbounded-space setting. The proof of the claim on bounded space is almost identical, using the imposed bound for the parameter $w$.

We use a $(V, \delta)$-bounded update scheme (Section 3, Definition 3). The update scheme $\mathcal{U}$ consists of $z = m/2$ rounds, where each round is a single update operation (so that when adding a single query to each round, we consider sequences of $m$ operations). For $\varphi = (1 + \sqrt{5})/2$ (Fibonacci number), let $I_k = (\lfloor nk\varphi^{-1} \rfloor \bmod n)$. We define

$$\mathcal{U} = \{u_1 u_2 \cdots u_z \mid u_k \text{ is } \operatorname{add}(I_k, \Delta_k), \Delta_k \in \{0, 1\}\}.$$

Thus there are two alternatives for each update operation in $\mathcal{U}$, which are chosen with equal probability.

LEMMA 2 [5]. *Update scheme* $\mathcal{U}$ *is* $(V, \delta)$*-bounded with* $\delta = \frac{1}{30}$ *and* $V(\ell) = 2^{-0.25\ell}$.

This lemma allows us to apply Corollary 2. The corollary requires us to consider the set of update/query sequences $\Sigma_z$ formed according to the pattern uquq ... uq, including $2z = m$ operations. For a given prefix sum algorithm, let $x$ be twice the average number of memory writes throughout an entire operation sequence chosen uniformly from $\Sigma_z$. As a bound on the data-structure cost we choose $w = x$ since all memory cells are initially zero. The probabilistic condition of the lemma clearly holds with $p = 1/2$.

Let $\chi = x/z$. Note that $\chi$ is twice the average number of writes per round; necessarily $\chi \geq 1$ since a given round has to modify the data structure at least in every other sequence. We can also assume $\chi \leq \log n$ (i.e., $x \leq z \log n$); otherwise the desired lower bound holds anyway.

We describe an epoch scheme $\mathcal{E}$ spanning $j = \sqrt{n}$ rounds, so $j < z/2$, as required by Corollary 2. The division of these rounds into epochs has the form of a geometric sequence, growing from the last round backwards. Specifically, the $i$ latest epochs contain $L_{q-i+1} = (\alpha\chi)^i$ update operations where $\alpha$ is a parameter to be chosen later.

Accordingly, the number of epochs is

$$q = \left\lfloor \frac{\log\sqrt{n}}{\log(\alpha\chi)} \right\rfloor.$$

Finally, we choose $\delta = \frac{1}{30}$ and $c = \frac{1}{8}$.

It remains to verify the inequality involving OV and $V(L_e)$ and determine the value of $\alpha$. We get a bound on OV from Theorem 1 and the inequalities $q < \log\sqrt{n}$ and $w < m\sqrt{n}$:

$$\begin{aligned} OV(w, 4xL_{e+1}/z, q) &= OV(w, 4\chi L_{e+1}, q) \\ &= OV(w, 4L_e/\alpha, q) \\ &\le (2^{2q}nw)^{12L_e/\alpha}(2^q n)^{4L_e/\alpha} \\ &\le (n^{5/2}m)^{12L_e/\alpha}(n^{3/2})^{4L_e/\alpha} \\ &= (n^{36}m^{12})^{L_e/\alpha} \\ &\le m^{48L_e/\alpha}. \end{aligned}$$

Combining the last inequality with Lemma 1 and taking the (base 2) logarithm, we get

$$\begin{aligned} \log(OV(w, 4xL_{e+1}/z, q)\cdot V(L_e)) &\le (48L_e/\alpha)(\log m) - 0.25L_e \\ &= L_e(48\log m/\alpha - 0.25). \end{aligned}$$

To apply Corollary 2, this expression has to be bounded by $\log c = -3$, so we choose $\alpha$ large enough to make the above expression negative. Since $L_e$ is a decreasing sequence, and the inequality is required for $e < q$, it suffices to ensure that

$$\begin{aligned} -3 \ge L_{q-1}(48\log m/\alpha - 0.25) &= \alpha\chi(48\log m/\alpha - 0.25) \\ &= \chi(48\log m - 0.25\alpha) \end{aligned}$$

This inequality holds for all $\alpha \ge 192\log m + 12$ (given $\chi \ge 1$). Hence we let $\alpha = \lceil(192\log m + 12)\chi\rceil/\chi$, the smallest number greater than $192\log m + 12$ such that $\alpha\chi$ is an integer.

We have thus established the conditions for Corollary 2. To evaluate the lower bound that results we substitute the value of $\alpha\chi$ in the definition of $q$, obtaining

$$q = \left\lfloor \frac{\log\sqrt{n}}{\log(\alpha\chi)} \right\rfloor \ge \left\lfloor \frac{1/2\log n}{\log(192\log m + 12) + \log\chi} \right\rfloor = \Omega\left(\frac{\log n}{\log m}\right).$$

## 7. An Algebraic Toolbox.

This section contains algebraic background for the following proof and includes the Counting Theorem, a combinatorial geometry result which may be interesting in its own right.

### 7.1. *Preliminaries*.

We recall some definitions and results from algebraic geometry (for details and proofs see [23]). An *affine algebraic variety* (henceforth variety) in $\mathbb{C}^k$ is defined by a set of algebraic equations in the $k$ coordinate variables: i.e., for polynomials $p_1, p_2, \ldots, p_r$ the set $\{\mathbf{x} \in \mathbb{C}^k \mid p_1(\mathbf{x}) = p_2(\mathbf{x}) = \cdots = p_r(\mathbf{x}) = 0\}$.

THEOREM 5. *Unions and intersections of a finite number of varieties are varieties.*

Given a collection of sets, a single set of the collection is called *redundant* if it is contained in the union of the other ones, and *irredundant* otherwise. A representation of a given set as a union of subsets is called *irredundant* if none of the sets in the representation is redundant.

THEOREM 6. *For every variety $V$, there is a unique decomposition into an irredundant set of varieties $\{V_i\}$ such that $\bigcup V_i = V$ and no $V_i$ can be further decomposed this way.*

The varieties in the above decomposition are called the *components* of $V$. We denote the set of components by $\mathtt{C}(V)$.

REMARK 1. Since each component of $V$ is a variety, the union of any proper subset of the components is also a variety, properly contained in $V$.

REMARK 2. It is easy to show that for $\{V_i\}$ to be an irredundant decomposition of $V$ (not necessarily into components), each $V_i$ must contain at least one *component* which is not contained in the union of the other ones.

DEFINITION. A variety of one component is called *irreducible*.

LEMMA 3. *Let $\{V_i\}$ be the components of $V$, and let $U$ be a variety. If $K$ is a component of $U \cap V$, then it is also a component of at least one of the varieties $U \cap V_i$.*

DEFINITION. The *dimension* of an irreducible variety $V$ in $\mathbb{C}^k$, denoted $\dim V$, is the largest length of a chain $V = V_0 \supset V_1 \supset \cdots \supset V_d \neq \emptyset$ of irreducible varieties. $0 \leq \dim V \leq k$. The *codimension* of $V$, $\operatorname{cod} V$, is $k - \dim V$.

The dimension of an arbitrary variety is defined as the largest of the dimensions of its components.

DEFINITION. For any variety $V$, $\mathtt{C}_j(V)$ is the union of components of $V$ possessing a given codimension $j$.

$\mathtt{C}_j(V)$ is a *pure-dimensional* variety—all its components have the same dimension.

THEOREM 7. *For varieties $U$, $V$ such that $U \subseteq V$,* $\operatorname{cod} U \geq \operatorname{cod} V$. *If $V$ is irreducible, and $U \subset V$, then* $\operatorname{cod} U > \operatorname{cod} V$.

THEOREM 8. *Let $V_1, V_2$ be pure-dimensional varieties in $\mathbb{C}^k$, and suppose $V_1 \cap V_2 \neq \emptyset$. Then every component of $V_1 \cap V_2$ has codimension at most* $\operatorname{cod} V_1 + \operatorname{cod} V_2$.

DEFINITION. For an irreducible variety $V$, its *degree* is the maximal cardinality of a finite set obtained by intersecting $V$ with a linear affine subspace.

The degree of a non-empty variety is positive.

Following Heinz and Schnorr [21], we define the degree of an arbitrary variety to be the sum of the degrees of its components. By this definition we have:

FACT 1. *The degree of a union of varieties is bounded by the sum of their own degrees. The degree of a variety is at least the number of its components.*

Heinz and Schnorr give the following variant of Bézout's theorem:

THEOREM 9 (Bézout Inequality). *Let $V_1, V_2$ be affine algebraic varieties in $\mathbb{C}^k$. Then* $\deg(V_1 \cap V_2) \leq \deg V_1 \cdot \deg V_2$.

$\mathbb{C}^k$ itself constitutes an irreducible variety of codimension 0 and degree 1. A set defined by a single non-degenerate equation (that is, not an identity) is called a *hypersurface*, and has codimension 1. The degree of a hypersurface is bounded by the degree of the polynomials in the defining equation.

7.2. *The Counting Theorem.* Let $x$ be a positive integer. Suppose we are given a vector of $n$ rational functions on $\mathbb{C}^x$, that is, functions of the form $R/T$ where $R$ and $T$ are polynomials. Evaluating them in some point of $\mathbb{C}^x$ yields a vector of results, belonging to $\mathbb{C}^n$. Let $S$ be a finite set of values and suppose we are only interested in result vectors within $S^n$. The Counting Theorem gives a bound for the number of such vectors that can be achieved. It is interesting that this bound is independent of $n$.

COUNTING THEOREM. *For $i = 1, 2, \ldots, n$ let $f_i = R_i/T_i$, where both $R_i$ and $T_i$ are polynomials on $\mathbb{C}^x$ of degree at most $d$. Let $S \subseteq \mathbb{C}$ be finite and let $m = |S|$. Let $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_n(\mathbf{x}))$. Then the number of distinct values from $S^n$ achieved by $\mathbf{f}(\mathbf{x})$ for $\mathbf{x} \in \mathbb{C}^x$ is bounded by $(dm)^x$.*

PROOF. For $0 \leq i \leq n$, and $v = (v_1, v_2, \ldots, v_i) \in S^i$, let

$$P_i(v_1, \ldots, v_i) = \{\mathbf{x} \in \mathbb{C}^x \mid R_j(\mathbf{x}) = v_j T_j(\mathbf{x}),\ j = 1, \ldots, i\}$$

and let

$$(*) \qquad \mathcal{P}_i = \bigcup_{v \in S^i} P_i(v).$$

$P_i(v)$ and $\mathcal{P}_i$ are clearly varieties in $\mathbb{C}^x$. Finally, let

$$\varphi(i) = \sum_{K \in \mathrm{C}(\mathcal{P}_i)} (dm)^{-\operatorname{cod} K} \deg K.$$

Let $N$ be the number of distinct values from $S^n$ achieved by $\mathbf{f}(\mathbf{x})$. Let $v \in S^n$. Obviously, $P_n(v)$ includes all points $\mathbf{x}$ where $\mathbf{f}(\mathbf{x}) = v$. These points must satisfy $T_1(\mathbf{x}), T_2(\mathbf{x}),$

$\dots, T_n(\mathbf{x}) \neq 0$ (we call such points *non-singular*). Therefore, $N$ is bounded by the number of sets $P_n(v)$ that contain non-singular points. We call these non-singular sets. Consider $\mathcal{P}_n$ : we show that in the union $(*)$ defining it, each non-singular $P_n(v)$ is non-redundant. To see this, let $\mathbf{x}$ be a non-singular point such that $R_j(\mathbf{x}) = v_j T_j(\mathbf{x})$ for all $j$. Changing any element of $v$ will violate the corresponding equation, so no $P_n(u)$, with $u \neq v$, can contain $\mathbf{x}$. It follows, by Remark 2, that the number of non-singular sets is bounded by the number of components of $\mathcal{P}_n$, and therefore (using both parts of Fact 1)

$$N \leq \sum_{K \in \mathtt{C}(\mathcal{P}_n)} \deg K \leq (dm)^x \varphi(n).$$

The number we seek is thus bounded by $(dm)^x \varphi(n)$.

Consider the following claims:

(1) $\varphi(0) = 1$.
(2) $\varphi(i)$ is a non-increasing function of $i$.

It is easy to see that together they complete the proof of the theorem.

*Proof of* (1). The set $S^0$ contains only one vector, of zero length; thus there is only one trivial set $P_0$, the whole space. It contains a single component (itself) of codimension 0 and degree 1, so $\varphi(0) = 1$.

*Proof of* (2). To show that $\varphi(i)$ is non-increasing, we relate the components of $\mathcal{P}_{i+1}$ to those of $\mathcal{P}_i$. Each set $P_{i+1}(v)$ is obtained from a set $P_i(u)$, where $u$ contains the first $i$ elements of $v$, by adding the condition $R_{i+1}(\mathbf{x}) = v_{i+1} T_{i+1}(\mathbf{x})$. If this equation is an identity, we have $f_{i+1} = R_{i+1}(\mathbf{x})/T_{i+1}(\mathbf{x})$ identically equal to $v_{i+1}$; thus we can ignore this component of the vectors. If this is not the case, the equation defines a hypersurface of codimension 1 and degree at most $d$. Thus

$$P_{i+1}(v) = P_i(u) \cap H,$$

where $H$ is the hypersurface. Note that a different hypersurface corresponds to each element of $S$.

Let $Q$ be a component of $P_{i+1}(v)$. By Lemma 3 it is also a component of $K \cap H$ for some $K \in \mathtt{C}(P_i(u))$. By Theorem 8,

$$\operatorname{cod} Q \leq \operatorname{cod} K + \operatorname{cod} H = \operatorname{cod} K + 1.$$

Moreover, by Theorem 7 we have $\operatorname{cod} Q = \operatorname{cod} K + 1$ unless $Q = K$, that is, unless $K \subseteq H$. We thus divide the components of $\mathcal{P}_{i+1}$ into two groups. The first includes components of $\mathcal{P}_i$ that are completely contained in one of the hypersurfaces $H$. Each component of this kind will contribute the same term to $\varphi(i+1)$ as it did to $\varphi(i)$. The second group includes components $Q = K \cap H$ that are proper subsets of the original component $K$. Such sets can count as components only if $K$ itself is not in $\mathcal{P}_{i+1}$, that is, does not belong to the first group. Suppose it does not; then we "gain" a host of new components, resulting from $K$'s intersection with $m$ hypersurfaces. Let $U$ be the union of all these components; $U$ is the result of intersecting $K$ with the union of $m$ hypersurfaces, each of degree $\leq d$. By the Bézout inequality,

$$\deg U = \sum_{Q \in \mathtt{C}(U)} \deg Q \leq md \deg K.$$

The contribution of all these components to $\varphi(i+1)$ is

$$\sum_{Q\in\mathbf{C}(U)} (dm)^{-\operatorname{cod} Q} \deg Q = (dm)^{-\operatorname{cod} K-1} \sum_{Q\in\mathbf{C}(U)} \deg Q \le (dm)^{-\operatorname{cod} K} \deg K,$$

which is the term contributed to $\varphi(i)$ by $K$. We conclude that $\varphi(i+1) \le \varphi(i)$. □

REMARK. It is easy to verify that the theorem holds as well with respect to functions $f_i$ which are polynomials of degree at most $d$.

## 8. Output Variability of Real-Number RAMs.

In this section we prove Theorem 1, giving an upper bound for the OV of three RAM variants, namely, RAM($\pm$), RAM($\times$) and RAM($/$). The functions that can be computed by a RAM($\pm$) or RAM($\times$) program are mathematically simpler than those that involve division as well. However, the proof of our result is almost the same for all models; so we complete first the proof for the first two models, and then briefly describe the modifications needed to handle RAM($/$).

A program segment which only refers to registers (no memory access instructions) and does not branch is *oblivious* in the sense that both program flow and data flow do not depend on values in memory. The *non-oblivious* instructions are *conditional branch* and the memory access instructions. For RAM($\pm$), we define the cost of a computation as the number of non-oblivious instructions executed; addition and subtraction are free. In RAM($\times$), multiplications are also counted. Note that we do not restrict the number of registers, and yet we count only accesses to memory cells. This resembles the situation in typical computers, where register access and instructions of additive type are cheap: our cost criterion counts instructions that are typically slow.

Recall that the definition of OV (Section 2) involves parameters $n$, $m$, $w$, $x$ and $q$. For the rest of this section, these parameters are fixed. We further fix $M$ and $Q$ as in the definition of OV. Our goal is to bound the number of vectors $\{\bar{Q}(X)\}$ that can be obtained by sequences of $x$ memory writes that update $M$ to produce $X$.

Comparing the current case to that of the cell probe model can indicate the source for difficulty in this proof. In the cell probe model there is a finite number of values to each word and this induces a finite number of possible computations. On the contrary, the bounds of the above theorem allow an infinite number of possible computations: each cell written may contain an arbitrary integer, and since it can be used for indirect addressing, the set of accessible addresses is also infinite. Thus, in order to bound the number of achievable result vectors, we partition the memory images in $M^x$ into classes such that in every class, a unique answer vector is obtained. Our aim is to show an upper bound on the number of these classes. We proceed to build the proof in stages, starting with an easy case and generalizing as we go.

First, we have to make some definitions for describing the contents of the memory image $X$. We denote by $W$ the set of addresses of non-zero cells in $M$. Let $A$ be the set of $x$ addresses updated to produce $X$ (there is no need to consider cases where $|A| < x$ since one can always rewrite a cell with its old value). The union, $W \cup A$, is the set of addresses that may be non-zero in $X$. The intersection, $W \cap A$, is the set of addresses re-written. Throughout most of the proof, we assume this set to be fixed. This means that we concentrate on one of the equivalence classes defined on $M^x$ by

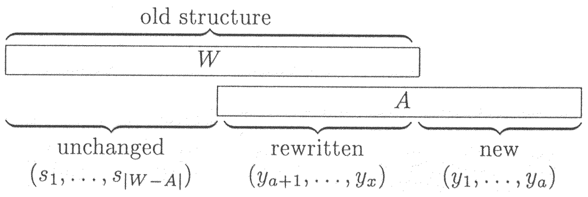


**Fig. 8.1.** Naming of addresses in $X$.

the identity of cells re-written. Call this class $\mathcal{M}$. The other addresses in $A$ are called *new*. We do not fix these addresses, which have infinitely many possible values. The variables $X_1, \ldots, X_x$ denote the $x$ *values* written. For definiteness, we relate $X_1, \ldots, X_a$ to new cells ($a = |A-W|$) and $X_{a+1}, \ldots, X_x$ to the re-written cells, each group given in increasing order of address. We denote their respective addresses by $y_1, \ldots, y_x$. Note that $y_1, \ldots, y_a$ are *variables* while the other addresses are *constants*. We denote the elements of $W - A$ by $s_1, \ldots, s_{|W-A|}$ (see Figure 8.1).

Through the proof we make some formal transformations and annotations of the query program. We refer the reader to examples accompanying the proof for illustration of the transformations described.

For a start, we replace the query program by $n$ programs, where program $i$ carries out the computation of $Q(i, X)$; thus the query input $i$ is changed into a program constant. This means that tests for the value of $i$, or computation of functions of $i$, can be removed from the program; as a result, such instructions need not be accounted for in our cost criterion. Anyway, we now have $n$ different programs which depend solely on the contents of memory. We start our process by writing these programs in a "low-level language," namely, an assembly-like language where memory access is made explicit, making use of registers $R_1, R_2, \ldots$ and where program statements are numbered (we refer to such statements as "instructions," although as a matter of convenience such a statement may include an arithmetic expression and thus represent a group of arithmetic instructions). See Example 1.

8.1. *Resolute Straight-Line Programs*. It is easy to see that a value computed by a straight-line sequence of instructions can be written down as an expression in the values

```
while $\langle\langle 0\rangle\rangle \le i$ do
    $\langle 0\rangle \leftarrow 2 \times \langle\langle 0\rangle\rangle + \langle i\rangle$
endwhile
return $\langle 0\rangle + 5$
```

The translation to low-level language for $i = 3$:

```
1:  $R_1 \leftarrow \langle 0\rangle$
2:  $R_2 \leftarrow \langle R_1\rangle$
3:  if $R_2 > 3$ goto 7
4:  $R_3 \leftarrow \langle 3\rangle$
5:  $\langle 0\rangle \leftarrow 2 \times R_2 + R_3$
6:  goto 1
7:  return $R_1 + 5$
```

**Example 1.** A query program.

```
1:  $R_1 \leftarrow \langle 3\rangle$                  $Y_1$
2:  $R_2 \leftarrow \langle 0\rangle$                  $Y_2$
3:  $\langle 0\rangle \leftarrow R_1 - 1$              $g_3(\bar{Y}) = Y_1 - 1$
4:  $R_3 \leftarrow \langle 0\rangle$                  $Y_4$
5:  **return** $R_1 \times R_2 + R_3$    $f_5(\bar{Y}) = Y_1Y_2 + Y_4$
```

**Example 2.** A straight-line program with annotations.

read from memory throughout the process. This expression is built from the arithmetic operations available to the machine, and is therefore a polynomial. The values of $d$ indicated in the theorem are used as bounds on the largest degree that such a polynomial may take. In RAM($\pm$), having no multiplication, no nonlinear term may be produced, hence $d = 1$. For RAM($\times$), we note that the term of largest degree computable with $q$ operations is $x^{2^q}$, hence $d = 2^q$. One of the difficulties of the proof is that the values read from memory may depend in intricate ways on the values of program variables through use of indirect addressing. The approach we take is to perform as much static analysis as possible to determine both program flow and data flow. To this end, we associate with each LOAD instruction $I$ a formal variable $Y_I$ that represents the value read. Wherever a value is obtained as the result of a straight-line computation, we can express it as a polynomial in these variables. Such polynomials will be denoted by $f_I$, $g_I$, etc. For instance, a STORE instruction uses two values: $f_I$ will denote the address and $g_I$ the value stored. At a later stage we will define a consistent naming for these functions. Such a function may depend, in general, on all the $Y$-variables of preceding instructions. We thus denote by $\bar{Y}_I$ the list of all $Y$-variables appearing before instruction $I$ in the straight-line program ($\bar{Y}$ is used when $I$ is understood from the context). See Example 2.

It is important to distinguish between these *formal variables* and the variables of the program. The former obtain values not by running the program but by analyzing it. We say they *evaluate* to a certain value. This value is not necessarily a number, since it will describe the result of a certain computation which may involve the inputs $X_1, \ldots, X_x$. So, these values will be polynomials in $\mathbf{x} = (X_1, \ldots, X_x)$.

In the current stage of the proof, we simplify the problem by restricting ourselves to direct-addressing straight-line programs, i.e., we assume that each access to memory uses a direct address which belongs to $W$ (the last restriction can be easily removed). For such programs, we can pick up instruction after instruction and evaluate the associated formal variable or polynomial. If $I$ is a LOAD instruction with direct address $\alpha$, the formal variable $Y_I$ can be evaluated as follows, assuming for the moment that there are no preceding STORE instructions. If $\alpha \in W - A$, it is one of the unchanged cells in $X$ so $Y_I = M(\alpha)$ (the contents of cell $\alpha$ in $M$). If $\alpha \in A$, $Y_I$ evaluates to the variable $X_j$ such that $y_j = \alpha$. Once the variables preceding an instruction that uses a function $f(\bar{Y})$ have been evaluated, $f$ can also be evaluated (symbolically) by simple substitution. This will determine, for instance, the value written by a STORE instruction. If the LOAD instruction $I$ is preceded by some STORE instructions, we look backwards from $I$ for the last STORE into $\langle\alpha\rangle$. $Y_I$ evaluates to the function stored by that instruction. Finally, we arrive at a polynomial (in $\mathbf{x}$) which represents the output of the program (Example 3).

Now let $f_i(\mathbf{x})$ be the polynomial computed by the query $Q(i, X)$. Thus the answer vector is given by $(f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_n(\mathbf{x}))$. Since the query answers must be in the

Assume the program of Example 2 is run on the following memory image:

| Address | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Value | $X_1$ | 3 | $X_2$ | 2 |

(hence $y_1 = 0$ and $y_2 = 2$). Analysis of the program yields the following results (a right arrow denotes "evaluates to"):

$$\begin{aligned} Y_1 &\to 2 \\ Y_2 &\to X_1 \\ g_3 &\to 1 \\ Y_4 &\to g_3 \to 1 \\ f_5 &\to 2X_1 + 1 \end{aligned}$$

**Example 3.** Symbolic evaluation in a program without indirect addressing.

range $1, \ldots, m$, the Counting Theorem shows that the number of different answer vectors obtainable is bounded by $(dm)^x$. We denote this number by $n_v$.

The last result can be extended to programs that use indirect addressing. Indirect addressing is introduced by allowing the address argument of a memory access instruction to be an arithmetic expression in $\bar{Y}$. With an instruction $I$ of that sort we will associate, in addition to the variable $Y_I$ which represents the value fetched (if it is a LOAD instruction), a polynomial $f_I(\bar{Y})$ which represents the address. When $f_I(\bar{Y})$ is a constant, it is possible to determine by static analysis (i.e., by looking at the program) what is stored in the cell accessed. To this end, we *resolve* the reference (as described above) to determine either the identity of the STORE operation that gave it its current value, or identify it with one of the cells in $W$, or decide it is an unused cell and zero will be read. When $f_I(\bar{Y})$ is not a constant, this decision may still be possible. Here for identifying the referenced cell as one that has been modified in a certain STORE instruction, it is required that the function $f_I(\bar{Y})$ coincide with the function that represents the address in the STORE instruction. A *resolute* program is a query program together with a set $\mathcal{A}$ of allowable images such that, for $X \in \mathcal{A}$, it is possible to resolve all memory references in the program. Direct-access programs treated above are resolute with no limitation on $X$ (except for the identity of rewritten cells, which we fixed for this discussion). For other programs, a finer analysis is required.

Consider the program in Example 4. There are three LOAD instructions in the program with associated $Y$ variables $Y_1$, $Y_2$ and $Y_4$. The condition $y_1 = 1$ implies that the memory cell of address 1 is identified with the variable $X_1$. Therefore, line 1 sets $Y_1$ to $X_1$. Line 2 uses indirect addressing where the function used for the address is exactly $Y_1$, hence we know it as the variable $X_1$. While we do not know its value, the condition $y_2 = X_1$ implies that $X_1$ gives the address of $X_2$: thus $Y_2$ evaluates to $X_2$. In line 3 the expression $Y_1 - Y_2$ (which evaluates to $X_1 - X_2$) is stored in a cell whose address is given by the

```
1:  $R_1 \leftarrow \langle 1 \rangle$
2:  $R_2 \leftarrow \langle R_1 \rangle$
3:  $\langle R_2 \times R_2 \times R_2 - 7 \times R_2 \rangle \leftarrow R_1 - R_2$
4:  $R_3 \leftarrow \langle 6 \rangle$
5:  **return**$R_3$
```

**Example 4.** A resolute program. $\mathcal{A}$ is defined by the conditions: $y_1 = 1$, $y_2 = X_1$, $X_2 \in \{-1, -2, 3\}$. The reader may resolve the references and verify that the return value evaluates to $X_1 - X_2$.

function $(Y_2)^3 - 7Y_2$. The next instruction accesses address 6. Therefore, we need to know whether the cell just written happens to be $\langle 6 \rangle$. By substituting $X_2$ for $Y_2$ we obtain the equation $(X_2)^3 - 7X_2 = 6$ whose solutions are $\{-1, -2, 3\}$. Therefore, on this set, $Y_3$ obtains the value $X_1 - X_2$, and line 5 makes it the result of the query. Note that $y_2$ may happen to equal 6, and in this case $X_2$ will be re-written; but this does not affect what comes next. Note also that if $X_2$ does not satisfy our equation, cell 6 will retain its prior value; in this case it will be important to know whether $y_2 = 6$. We see that resolving a program with indirect addressing involves algebraic equations that induce a partition into cases; those may have to be broken into subcases (by means of other equations) and so forth. We will later formalize this process, in order to estimate the cardinality of the partition obtained at its completion.

So, $n_v = (dm)^x$ *actually bounds the number of different answers that can be given by a straight-line resolute program*. Given a set of straight-line programs, e.g., the paths of a computation tree, we will be interested in sets $\mathcal{A}$ such that all these programs can be simultaneously resolved for memory images in $\mathcal{A}$. In such a case we say that the computation tree is resolute.

8.2. *Branching Programs*. We now turn our attention to programs that include branching. The program computing $Q(i, X)$ can be represented in a standard way as a tree $T_i$. In this tree, internal nodes represent the non-oblivious instructions, while the leaves correspond to **return** statements and specify the query output. The height of this tree will be bounded by $q$.

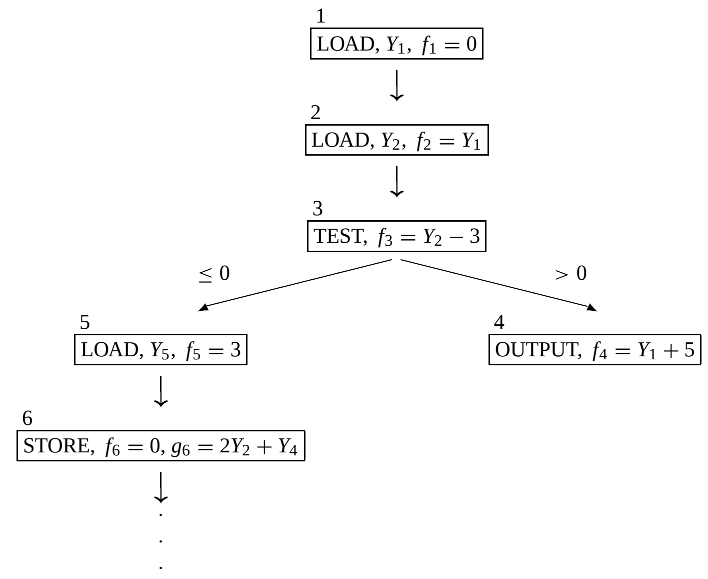
The path from the root to a given node describes a straight-line sequence of instructions, and can be treated as a straight-line program. Thus the operands of the instruction in node $\nu$ can be written down as polynomials in $Y$-variables associated with ancestors of $\nu$ representing LOAD instructions.

More precisely, each node $\nu$ has one of these types:

(i) LOAD (memory read). A variable $Y_\nu$ is associated with this node to represent the value read. The address accessed is determined by a polynomial in the results of preceding LOAD instructions, $f_\nu(\bar{Y})$. A LOAD node has one child representing the next instruction.
(ii) STORE (memory write). Two functions are associated with such a node: $f_\nu(\bar{Y})$ determines the address and $g_\nu(\bar{Y})$ the value written.
(iii) TEST (comparison), which has two children, and the computation proceeds to one of them depending on the conditional $f_\nu(\bar{Y}) > 0$.
(iv) OUTPUT, which is a leaf, where $f_\nu(\bar{Y})$ describes the query result.

We call these trees *the computation trees* (Example 5).

We assume that we have restricted the inputs to a set $\mathcal{A}$ so that *every path in the program trees* is resolute. Then all the functions $f_\nu$ in the trees evaluate to polynomials in $\mathbf{x} = (X_1, \ldots, X_x)$; it is convenient to regard them as polynomials on $\Re^x$. The paths taken by the programs for a certain input is determined by the outcome of the comparisons in the test nodes of the trees. Therefore, we now partition $\Re^x$ according to comparison results to obtain subsets which determine unique computation paths. For estimating the size of this partition, we make use of Warren's lemma on sign sequences of polynomials.



**Example 5.** The tree $T_3$ for the program of Example 1. The tree is obtained by unwinding the while-loop, and is truncated at height $q$.

For any real number $x$, we define

$$\operatorname{sgn}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ +1, & x > 0. \end{cases}$$

LEMMA 4. *Let $f_1, \ldots, f_r$ be real polynomials in $x$ variables, each of degree at most $d \geq 1$. If $r \geq x$, the number of sign sequences $\operatorname{sgn} \mathbf{f}(\mathbf{x}) = (\operatorname{sgn} f_1(\mathbf{x}), \ldots, \operatorname{sgn} f_r(\mathbf{x}))$ that consist of terms $+1$, $-1$ only does not exceed $(4edr/x)^x$, where $e$ is the base of the natural logarithm.*

PROOF. See [31]. $\square$

LEMMA 5. *Let $\mathcal{A} \subseteq \mathcal{M}$ be a set of inputs (memory images) on which all computation trees resolve. The size of the partition induced on $\mathcal{A}$ by the results of all comparisons performed in all the computation trees is bounded by $(d2^{q+2}n)^x$.*

PROOF. Consider a list $f_1, \ldots, f_r$ of all the functions associated with comparison nodes in the trees $T_1, \ldots, T_n$; since the height of each computation tree is bounded by $q$, and each path of interest must contain one output node and one memory read node, we have $r < 2^{q-2}n$. Since the trees are resolute, each $f_i$ evaluates to a polynomial (which we consider to be defined on $\Re^x$), and, as argued before, the degree of each polynomial is at most $d$. We can assume that none of these polynomials is a constant, since such comparison would be redundant and can be ignored. We denote by sn $f_i(\mathbf{x})$ the result of the comparison $f_i(\mathbf{x}) > 0$?, that is,

$$\operatorname{sn} f_i(\mathbf{x}) = \begin{cases} 1 & \operatorname{sgn} f_i(\mathbf{x}) = 1, \\ -1 & \operatorname{sgn} f_i(\mathbf{x}) < 1. \end{cases}$$

The size of the partition in question is bounded by the number of different values taken by the vector $\operatorname{sn} \mathbf{f}(\mathbf{x}) = (\operatorname{sn} f_1(\mathbf{x}), \ldots, \operatorname{sn} f_r(\mathbf{x}))$ as $\mathbf{x}$ ranges over $\Re^x$. Assume that there are $\gamma$ such values; let $\Gamma \subseteq \Re^x$ be a set such that $|\Gamma| = \gamma$ and each possible value of $\operatorname{sn} \mathbf{f}(\mathbf{x})$ is achieved on $\Gamma$. Let

$$\varepsilon = \min\{ f_i(\mathbf{x}) \mid 0 < i \leq r,\ \mathbf{x} \in \Gamma \text{ and } f_i(\mathbf{x}) > 0\}$$

and define $g_i(\mathbf{x}) = f_i(\mathbf{x}) - \varepsilon/2$ for all $i$. Thus at all the points of $\Gamma$, $g_i(\mathbf{x})$ is different from zero, and $\operatorname{sn} \mathbf{f}$ coincides with $\operatorname{sgn} \mathbf{g}$. Moreover, at these points $\operatorname{sgn} \mathbf{g}$ consists of terms $+1$, $-1$ only (as required for Lemma 4 to give a correct bound). The degree of the polynomials $g_i$ is the same as of $f_i$. Thus if $r \geq x$, the result follows by Lemma 4. Otherwise, $(d2^{q+2}n)^x > (d2^{q+2}n)^r > 2^r$, which is a trivial bound on the number of such sequences. □

We conclude that $\mathcal{A}$ can be broken into at most $n_t = (d2^{q+2}n)^x$ classes, such that for each class one output node can be singled out for every computation tree, and these nodes will be reached for all the inputs in the class. Thus, in each class the trees degenerate into $n$ straight-line programs; the number of answer vectors that can be obtained in this case is thus bounded by $n_v$. We obtain

Corollary 3. *Let $\mathcal{A} \subseteq \mathcal{M}$ be a set of inputs on which all computation trees are resolute. The size of the partition induced on $\mathcal{A}$ by the identity of answer vectors is bounded by $n_t n_v = (d2^{q+2}n)^x m^x$.*

8.3. *The Meta Tree.* Our next goal is to partition the set of memory images $\mathcal{M}$ into classes such that on each class, all computation trees are (simultaneously) resolute. To this end, we pick the memory access nodes in the computation trees one at at time. For each node, we find conditions on the variables $X_1, \ldots, X_x$; $y_1, \ldots, y_a$ that yield a resolution of the memory reference, namely force the value used for an address to coincide either with a "known" address or be distinct from all of them (in this case it hits a zero cell). For the sake of analysis we embed the possible vectors $(\mathbf{x}; \mathbf{y})$ in the vector space $\mathbb{C}^{x+a}$. We obtain a partition of $\mathbb{C}^{x+a}$ into subsets such that for all values of $(\mathbf{x}; \mathbf{y})$ in a given subset the resolution of the reference is fixed. Note that it is possible that such a subset will include no valid input, for example because $\mathbf{y}$ must consist of natural numbers to be valid. However, we are only looking for an upper bound on the size of the partition obtained, and the size of the partition on $\mathbb{C}^{x+a}$ will do.

To deal with a particular memory access node, it is necessary to resolve its ancestors (in the computation tree) first so that we know how the memory looks when the node is reached. Therefore, we arrange all the memory access nodes in the $n$ trees in a list $\nu_0, \nu_1, \ldots, \nu_{L-1}$ which follows pre-order for each tree. Note that the height of the trees is bounded by $q$ and the nodes of interest are internal nodes; therefore $L < 2^{q-1}n$. Starting with a trivial partition in which $\mathbb{C}^{x+a}$ is a single class, we proceed through

$L$ steps in which the partition is repeatedly refined, i.e., each class is subdivided into smaller classes. This process naturally defines a tree which we call the *meta tree*: a level of this tree corresponds to a memory access node (level $i$ for node $\nu_i$), so there are $L$ levels. Each node of the meta tree corresponds to one class of its level, and has children in the next level for all the subclasses of that class. We remark that this tree is an abstract structure which exists only in the analysis.

To proceed with the analysis, we introduce some additional formal variables which help in tracing the usage of memory addresses. First, we represent the contents of memory by a set of formal variables which bear names of the form $c(u)$ (for "contents of address $u$"). $u$ may be either a number or the name of another formal variable. Initially, we have

$$(2) \quad c(u) = \begin{cases} M(u) & \text{for} \quad u \in W - A \quad (\text{here } u \text{ accepts integer values}), \\ X_l & \text{for} \quad u = y_l, \quad l = 1, \dots, x \quad (\text{here } u \text{ takes formal values}). \end{cases}$$

Next, with each node $\nu$ that represents a memory write, we associate a formal variable $Z_\nu$ to represent the address written to. The expression $c(Z_\nu)$ naturally denotes the value written.

So, a node $\lambda$ of level $i$ in the meta tree defines:

(i) A subset $P_\lambda$ of $\mathbb{C}^{x+a}$ which limits the values of $X_1, \dots, X_x; y_1, \dots, y_a$.
(ii) An assignment $A_\lambda$ which assigns to each $Y$, $Z$ and $c$-variable defined before this node a value from the polynomial ring $\mathbb{C}[X_1, \dots, X_x]$.

The construction of the tree is designed to satisfy the following claim; recall that $\bar{Y}(\nu)$ is the list of $Y$-variables associated with ancestors of node $\nu$.

CLAIM 1. *The subsets $\{P_\lambda\}$ associated with the nodes of level $i$ in the meta tree form a partition of $\mathbb{C}^{x+a}$. For each vector $(X_1, \dots, X_x; y_1, \dots, y_a) \in P_\lambda$, if a computation of a query using these values reaches $\nu_i$, the values of formal variables associated with ancestors of $\nu_i$ and the contents of memory when $\nu_i$ is reached are given by $A_\lambda$.*

This invariant leads immediately to the following corollary, regarding the finest partition obtained.

COROLLARY 4. *Each subset associated with a leaf of the meta tree determines the values of all formal variables associated with the computation trees.*

The structure of the tree will be now defined inductively in order to establish Claim 1. Let $E$ be an expression, and let $A$ be an *assignment*. The notation $E[A]$ is used for the result of replacing each variable in the expression $E$ by the value assigned to it in $A$.

(i) The root $r$ of the meta tree has $P_r = \mathbb{C}^{x+a}$ and $A_r$ only includes the values for $c(u)$ given in (2). These assignments correctly represent the situation when node $\nu_0$ is reached because there are no memory access nodes preceding it.
(ii) Let $\lambda$ be a node of level $i$, and assume the construction has been correctly carried out up to this level. Recall that $f_{\nu_i}$ is defined in terms of $\bar{Y}_{\nu_i}$. Since $A_\lambda$ has to resolve all the variables in $\bar{Y}_{\nu_i}$, $f_{\nu_i}[A_\lambda]$ is a function of $X_1, \dots, X_x$. Let $Z_1, \dots, Z_t$ be the $Z$-variables along the path from $\nu_i$ to the root of its computation tree. Consider the

list of “address expressions”

$$Z_1, \ldots, Z_t; y_1, \ldots, y_x; s_1, \ldots, s_{|W-A|}.$$

This list goes backwards in time: the $Z$-variables are listed from our node back to the root which designates the start of our program, $y_i$ denotes addresses modified in the epochs preceding this query, and $s_j$ are addresses in use prior to these updates. Each element of this list is either a constant, or must be included in $A_\lambda$. An element $u$ in the list will be called *a duplicate* if there exists an element $v$ preceding $u$ in the list (i.e., newer), such that the functions $u[A_\lambda]$ and $v[A_\lambda]$ coincide over $P_\lambda$. Pick elements from this list from left to right, skipping duplicates. Call the elements so chosen $u_1, u_2, \ldots$ (their number will be considered later). This process finds out the memory cells that are in use (not zero), for the following reason. Addresses eligible to be non-zero are those in the initial data structure ($s_i$), those modified during updates ($y_j$) and those modified in the execution of this query ($Z_k$). If a memory cell is written twice, the last writer leaves its mark, so we remove the older appearance of a duplicate address. Now for $j = 1, 2, \ldots$ let

$$P_j = \{(\mathbf{x}; \mathbf{y}) \in P_\lambda \mid f_{\nu_i}[A_\lambda](\mathbf{x}) = u_j[A_\lambda](\mathbf{x}; \mathbf{y})\}, \tag{3}$$

this is the set on which the address accessed by our node coincides with $u_j$. Let

$$P_0 = P_\lambda - \bigcup_{j=1}^{k} P_j;$$

this is the set on which the address selects an unused cell.

The node $\lambda$ has a child $\lambda_j$ for each $j$ such that $P_j$ is not empty. With this node we associate the set $P_j$ and the assignment $A_\lambda$ augmented according to the type of node $\nu_i$:

(i) $\nu_i$ is a read node. Then we add an assignment for $Y_\lambda$, which should reflect the result of the memory read. Thus in $\lambda_0$ the value zero is assigned. The value of $c(u_j)$ is assigned in $\lambda_j$.

(ii) $\nu_i$ is a write node. Then we add assignments for $Z_\lambda$ and $c(Z_\lambda)$. Regardless of $j$, these are $Z_\lambda \leftarrow f_{\nu_i}[A_\lambda]$ and $c(Z_\lambda) \leftarrow g_{\nu_i}[A_\lambda]$. We still have a different child for each $P_j$; this will ensure that in subsequent nodes, *duplicate removal* works properly, as address expressions will either coincide or differ on the whole of each class.

Recall that $w$ bounds the number of cells in use; hence there are at most $w$ children $\lambda_j$ with $j > 0$. We proceed to estimate the cardinality of the finest partition (equivalently, the number of leaves).

LEMMA 6. *The partition of $\mathbb{C}^{x+a}$ induced by leaves of the meta tree is of cardinality $\leq (2^{q-1}dwn)^{x+a}$.*

The rest of this subsection is devoted to the proof of Lemma 6. The central idea is arguing about the dimension of the classes associated with $P_\lambda$. We will show that if the number of these sets increases, their dimension must decrease, whereby the desired bound will follow.

Recall that $d$ is a bound on the degree of any polynomial computed by our query program.

LEMMA 7. *Let $0 \leq i < L$. Let $\mathcal{P}_i$ be the partition of $\mathbb{C}^{x+a}$ defined by level $i$ of the meta tree. We can define, for each $P \in \mathcal{P}_i$, a variety $\overline{P}$, such that $\overline{P} \supseteq P$ and the following property holds. For all $0 \leq j \leq x + a$, let $N_{i,j} = \sum_{P \in \mathcal{P}_i} \deg \mathtt{C}_j(\overline{P})$. Then $N_{i,j} \leq \binom{i}{j}(wd)^j$.*

PROOF. We proceed by induction on $i$. For $i = 0$, the level contains a single node, the root $r$. $P_r = \mathbb{C}^{x+a}$ so we define $\overline{P}_r$ to be the same. Note that $\mathtt{C}_j(\mathbb{C}^{x+a})$ is $\mathbb{C}^{x+a}$ itself for $j = 0$ and is empty for $j > 0$. $\deg \mathbb{C}^{x+a} = 1$, so $N_{0,0} = 1 = \binom{0}{0}(wd)^0$, while for $j > 0$ we have $N_{0,j} = 0$.

We next assume the sets have been defined and the lemma holds for level $i$ and consider level $i + 1$.

Let $\lambda$ be any node in level $i$. Consider the sets $P_0, \ldots, P_k$ associated with its children (we renumber the children consecutively so $k \leq w$). For $l \geq 1$, $P_l$ is defined (by (3)) as the set of points in $P_\lambda$ that satisfy a certain equation in $x + a$ variables; hence $P_l = P_\lambda \cap S_l$, where $S_l$ is either a hypersurface in $\mathbb{C}^{x+a}$ or the whole space (if the equation is an identity). Recall that $P_j$ must be non-empty for $\lambda_j$ to exist; hence $\overline{P}_\lambda \cap S_l$ is non-empty as well. We tentatively define $\overline{P}_l$ to be $\overline{P}_\lambda \cap S_l$. Obviously this definition satisfies the requirement $\overline{P}_l \supseteq P_l$, and $\overline{P}_l$ is clearly a variety. Our goal now is to relate the components of the sets $\overline{P}_l$ to those of $\overline{P}_\lambda$.

If there is an $l$ such that $\overline{P}_\lambda \subseteq S_l$, then $\overline{P}_l = \overline{P}_\lambda$ and $\lambda_l$ must be the only child of $\lambda$: this is due to the fact that $\mathcal{P}_{i+1}$ is a partition. In this case the components of $\overline{P}_l$ are simply those of $\overline{P}_\lambda$.

Assume now that no $S_l$ contains the whole of $P_\lambda$; then no $S_l$ can be the whole space. Each $S_l$ must be a hypersurface, of codimension 1 and degree at most $d$. Obviously

$$\overline{P}_l = \bigcup_k \bigcup_{K \in \mathtt{C}_k(\overline{P}_\lambda)} K \cap S_l.$$

Lemma 3 shows that in order to study the components of $\overline{P}_l$ it suffices to look at the components of each intersection $K \cap S_l$. Let $K \in \mathtt{C}_k(\overline{P}_\lambda)$. By Theorem 8, $K \cap S_l$ is a variety of codimension bounded by $k+1$. Let $Q$ be one of its components. Since $Q \subseteq K$, by Theorem 7 cod $Q \geq k$; and cod $Q = k$ if and only if $Q = K$, that is, if and only if $K \cap S_l \supseteq K$, i.e, $K \subseteq S_l$.

The last argument shows that all components of $\mathtt{C}_j(\overline{P}_l)$ belong to one of the following types: (i) components of $\mathtt{C}_j(\overline{P}_\lambda)$ which pass on to $\mathtt{C}_j(\overline{P}_l)$ unchanged; (ii) components created by the intersection of $C_{j-1}(\overline{P}_\lambda)$ with $S_l$.

We next consider $\lambda_0$. Provided that $P_0$ is not empty, we define $\overline{P}_0 = \overline{P}_\lambda$. It thus inherits all the components unchanged.

For our counting arguments we want each component of $\overline{P}_\lambda$ which is repeated in the next level to appear only once, that is, only in a single $\overline{P}_l$. We achieve this by deleting redundant appearances of components; however, we must preserve the property that $\overline{P} \supseteq P$ for all sets $P$ involved. Suppose that a component of $\overline{P}_\lambda$ is contained in two different hypersurfaces $S_l$ and $S_m$. Then it is contained in their intersection. However, by looking at (3) we see that a point of $S_l \cap S_m$ is a point where $u_l[A_\lambda] = u_m[A_\lambda]$; such equality cannot hold within $P_\lambda$ by virtue of the *duplicate removal* process. It follows that we can safely *delete* this component from both $\overline{P}_l$ and $\overline{P}_m$. Next, if a component of $\overline{P}_\lambda$ is contained in a single $S_l$, it may be deleted from $\overline{P}_0$. This is safe because $P_0$ excludes $S_l$. Recall that deleting whole components of a variety produces a variety (Remark 1).

To sum up, we ensure that each component of $\mathtt{C}_j(\overline{P}_\lambda)$ is repeated at most once in the next level, while additional components of same codimension may emerge from the intersection of $C_{j-1}(\overline{P}_\lambda)$ with the ($\leq w$) hypersurfaces. The degree of components of the first kind remains what it was in $\overline{P}_\lambda$; for the second kind, we use Bézout's inequality:

$$\deg(C_{j-1}(\overline{P}_l) \cap S_l) \leq d \cdot \deg C_{j-1}(\overline{P}_l).$$

Summing the bounds we have on the degrees of these two kinds of components, we obtain

$$\begin{aligned} N_{i+1,j} &\leq N_{i,j} + wdN_{i,j-1} \\ &\leq \binom{i}{j}(wd)^j + wd\binom{i}{j-1}(wd)^{j-1} \\ &= \left(\binom{i}{j} + \binom{i}{j-1}\right)(wd)^j \\ &= \binom{i+1}{j}(wd)^j. \qquad \square \end{aligned}$$

We now estimate the cardinality of the finest partition, $\mathcal{P}_L$. Each $P \in \mathcal{P}_L$ is a non-empty set and is therefore contained in a non-empty $\overline{P}$. Let $j = \dim \overline{P}$. Then $\overline{P}$ contributes at least 1 to $N_{L,j}$. Therefore the cardinality of the finest partition is bounded by

$$\sum_{j=0}^{x+a} N_{L,j} \leq \sum_{j=0}^{x+a} \binom{L}{j}(wd)^j \leq (wd)^{x+a} \sum_{j=0}^{x+a} \binom{L}{j} \leq (wdL)^{x+a} \leq (2^{q-1}wdn)^{x+a}.$$

The last inequality follows from the following claim, completing the proof of Lemma 6.

CLAIM 2. *For all* $1 < m \leq n$, $\sum_{k=0}^{m} \binom{n}{k} \leq n^m$.

PROOF. For $m = 2$,

$$\sum_{k=0}^{m} \binom{n}{k} = 1 + n + \binom{n}{2} = \frac{n^2 + n + 2}{2} \leq n^2.$$

For $m > 2$, we assume the lemma holds for $m - 1$ and use induction:

$$\sum_{k=0}^{m} \binom{n}{k} = \sum_{k=0}^{m-1} \binom{n}{k} + \binom{n}{m} \leq n^{m-1} + \binom{n}{m} < n^{m-1} + n^{m-1}(n-1) = n^m. \qquad \square$$

8.4. *Wrapping up the Proof.* Let $n_{ad}$ be the number of choices for $W \cap A$. For a single choice of these addresses we have computed the size $n_p$ of the partition induced by leaves of the meta tree. Concentrating on one of these parts, we obtain resolute programs. By Corollary 3, the number of answer vectors that can be obtained for memory images in this part is bounded by $n_t n_v$. Therefore the number of different query answer vectors that can be obtained when both the addresses and values of modified cells vary freely is bounded by

$$n_{ad} n_p n_t n_v.$$

As $|W| \leq w$, we have $n_{ad} \leq \sum_{j \leq x} \binom{w}{j}$; and Claim 2 gives $n_{ad} \leq w^x$. By Lemmas 6 and 5 we deduce

$$OV(w, x, q) \leq w^x (2^{q-1} dwn)^{2x} (2^{q+2} dn)^x (dm)^x \leq (2^q dwn)^{3x} (dm)^x.$$

8.5. *Output Variability of RAM(/).* Regarding RAM(/), Theorem 1 claims that

$$OV(w, x, q) \leq (2^{2q} nw)^{3x} (2^q n)^x.$$

The proof follows the same lines as for the previous models. $n_{ad}$, $n_p$, $n_t$ and $n_v$ are defined in the same way. In the last section we used the fact that a function built with the operators $+$, $-$ and $\times$ is a polynomial in the variables used. When division is allowed, the class of functions obtained is the class of *rational functions*. This is a result of the identities:

$$\begin{aligned} \frac{P}{Q} \pm \frac{R}{T} &= \frac{PT \pm RQ}{QT}, \\ \frac{P}{Q} \cdot \frac{R}{T} &= \frac{PR}{QT}, \\ \frac{P}{Q} \bigg/ \frac{R}{T} &= \frac{PT}{QR}, \end{aligned}$$

whereby it also follows that if $P/Q$ is obtained with less than $q$ algebraic operations, then both deg $P$ and deg $Q$ are bounded by $2^{q-1}$. This bound will hold for all the functions computed throughout a query program whose total length is bounded by $q$.

Consider first $n_p$. We note that the equation in rational functions $P/Q = R/T$ is equivalent to the polynomial equation $PT = QR$. In the equations (2) that determine the meta-tree partition, the degree of each of $P$, $Q$, $R$ and $T$ is bounded by $2^{q-1}$; hence the degree of the equation is bounded by $2^q$. We can apply Lemma 6 to obtain $n_p \leq (2^{q-1} 2^q wn)^{x+a} \leq (2^{2q-1} nw)^{2x}$.

We now turn to $n_t$. This partition is defined by inequalities which now have the form $P/Q > 0$. This inequality can be rewritten as $PQ > 0$ with the same result. $PQ$ is a polynomial of degree bounded by $2^q$. This is the case considered in Lemma 5, so again $n_t = (2^{2q+2} n)^x$.

Next we consider the number $n_v$ of answer vectors that can be obtained once the identity of cells accessed and branches taken has been fixed. The results are now produced by a vector of $n$ real-valued rational functions $f_1, f_2, \ldots, f_n$ ($f_i$ represents the output of $Q(i)$), and $n_v$ is at most the number of vectors from $\{1, 2, \ldots, m\}^n$ that can be obtained by $(f_1(\mathbf{x}), \ldots, f_n(\mathbf{x}))$ for $\mathbf{x} \in \mathbb{C}^x$. Using the Counting Theorem with $d = 2^q$ yields $n_v \leq (2^q m)^x$.

$n_{ad}$ is the same as in the last subsection; so

$$OV(w, x, q) \leq n_{ad} n_p n_r n_v \leq w^x (2^{2q-1} nw)^{2x} (2^{2q+2} n)^x (2^q m)^x = (2^{2q} nw)^{3x} (2^q m)^x,$$

completing the proof of Theorem 1.

## 9. Other RAM Models.

The general definition of the RAM family in Section 4 developed as a direct result of the fact that so many variants have been used in the literature. One group of variants consists of those that manipulate integer numbers with instructions that are not algebraic. Examples include *integer division* and *bit operations*. Such additions invalidate the considerations on which our bounds on OV hinge. An indication of the possible consequences of such enhancement has been given by Paul and Simon [27]. They prove a lower bound of $\Omega(n \log n)$ for sorting $n$ integers in the arithmetic model; on the other hand, they show that the instructions of *integer division* and *bitwise* AND can be used to break the lower bound and in fact to sort in linear time.

Let RAM($\wedge$) (resp. RAM($\div$)) be obtained from RAM($\times$) by adding a primitive of bitwise AND (resp. integer division). We next show that both machines solve *union-find* in constant time per operation. Thus our lower bounds are broken too. Note that these instructions do not contribute to the creation of larger integers, but they do allow a program to make use of information which is encoded as part of a large number.

We describe the union-find algorithm for RAM($\wedge$). For convenience, we adopt the following version of the problem: the elements are named $0, \ldots, n-1$, which are also the names of the singleton sets initially containing them. Each *union* operation specifies the names of two sets and one of these becomes the name of the union. We also assume that $n$ is a power of two.

The pivot of the algorithm is a single integer $U$ that describes the current sets. At any moment, let $F_i$ be the result of a *find* on $i$, i.e., the name of the set containing the element $i$. Then

$$U = \sum_{i=0}^{n-1} F_i \cdot n^i.$$

In addition, for each set $S_j$ we have an integer $U_j$ which describes the current contents of these sets:

$$U_j = \sum_{i \in S_j} n^i.$$

The initial contents of these variables should therefore be

$$\begin{aligned} U &= \sum_{i=0}^{n-1} i \cdot n^i, \\ U_j &= n^j. \end{aligned}$$

These values can be set by a linear-time initialization phase; if desired, this work can be divided among union operations to avoid the setup phase.

It is easy to see that the following operations implement correctly the operation union$(i, j)$, which adjoins $S_i$ to $S_j$:

$$\begin{aligned} U_j &\leftarrow U_j + U_i, \\ U &\leftarrow U - iU_i + jU_i. \end{aligned}$$

We now explain the implementation of *find*, which uses three more tables that can be built beforehand or on-line as discussed above. For each $i = 0, \dots, n-1$ we need

$$\begin{aligned} M_i &= (n-1)n^i, \\ N_i &= n^{n-i} \end{aligned}$$

and an array $T$ where $T[j \cdot n^n] = j$ (only these array items must be set). The reader may verify that the following expression yields the name of the set containing $i$:

$$T[N_i \cdot (U \wedge M_i)].$$

The solution for RAM($\div$) is similar.

**10. Conclusion.** The generalized Fredman–Saks technique presented in [5] was designed to be applicable to other models or computation in addition to the cell probe model originally used by Fredman and Saks. Lower-bound proofs using this framework can be easily transferred to a different model if a good bound on its OV is found. The main contribution of this paper was the derivation of such a bound on the OV of real-number algebraic RAMs. This means, in essence, that every lower bound obtained in the cell probe model using that technique can now be transferred to a RAM lower bound.

We have applied the result to obtain tight lower bounds for two central data-structure problems in algebraic real-number RAMs. So far, work on the complexity of these problems has concentrated on different computational models and the reader is invited to find more references and comparison with previous work in [5].

It is interesting to notice that the complexity of the computational problems considered in this work is the same for an integer RAM with additive instructions only and for a real-number RAM with multiplication and division. We suggest that these problems have some essential simplicity, related to the fact that their character is more one of managing data structures than one of calculating with data, and that for problems of this kind there is no advantage to using the stronger algebraic operations. Making this informal notion more precise is an issue for further research. Note that multiplication *can* be of advantage when combined with non-algebraic operations such as integer division or bit fiddling, a fact well known in data-structure research.

Since the theory of lower bounds in the cell probe model has developed faster than the theory for RAMs, it seems natural to try to extend lower-bound techniques from the former model to the latter. In addition to our work, such extensions can be found in [13] and [14]. The latter is the only other work, known to us, that proves RAM lower bounds for dynamic problems.

Another natural direction for further research is to consider RAM models with strong, non-algebraic instructions, but limitations on the word length. These “word RAMs” stand halfway between the cell probe model and the algebraic RAM. Of course, the model is only significant for lower bounds if the cell probe model does not yield a satisfactory result for the given word length. Hagerup [20] surveys the interesting recent developments regarding this model.

## References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] S. Alstrup, A. M. Ben-Amram and T. Rauhe, Worst-case and amortised optimality in union-find, *Proc*. 31*st ACM Symp*. *on Theory of Computing* (STOC), 1999, pp. 499–506.

[3] A. M. Ben-Amram, On the Power of Random Access Machines, Thesis, Tel-Aviv University, 1995.

[4] A. M. Ben-Amram and Z. Galil, Lower bounds for data structure problems in RAMs, *Proc*. 32*nd Annual IEEE Symp*. *on Foundations of Computer Science* (FOCS), San Juan, PR, 1991, pp. 622–631.

[5] A. M. Ben-Amram and Z. Galil, A generalization of a lower bound technique due to Fredman and Saks, *Algorithmica* 30:1 (2001), 34–66.

[6] A. M. Ben-Amram and Z. Galil, Topological lower bounds for algebraic random access machines, *SIAM J. Comput*., to appear.

[7] M. Ben-Or, Lower bounds on algebraic computation trees, *Proc*. 15*th Annual ACM Symp*. *on Theory of Computing* (STOC), 1983, pp. 80–86.

[8] N. Blum, On the single-operation worst-case time complexity of the disjoint set union problem, *SIAM J. Comput*. 15:4 (1986), 1021–1024.

[9] N. H. Bshouty, Lower bounds for the complexity of functions in random access machines, *J. Assoc*. *Comput*. *Mach*. 40:2 (1993), 211–223.

[10] S. A. Cook and R. A. Reckhow, Time bounded random access machines, *J. Comput*. *System Sci*. 7:4 (1973), 354–375.

[11] P. F. Dietz, Optimal algorithms for list indexing and subset rank, *Proc*. *Workshop on Algorithms and Data Structures* (WADS), 1989, pp. 39–46.

[12] E. Dittert and M. J. O'Donnell, Lower bounds for sorting with realistic instruction sets, *IEEE Trans*. *Comput*. C-34:4 (1985), 311–317.

[13] F. E. Fich and P. B. Miltersen, Tables should be sorted (on Random Access Machines), *Proc*. 4*th International Workshop on Algorithms and Data Structures* (WADS), 1995, Lecture Notes in Computer Science 955, Springer-Verlag, Berlin, pp. 482–494.

[14] G. S. Frandsen, J. P. Hansen and P. B. Miltersen, Lower bounds for dynamic algebraic problems, *Proc*. 16*th Symp*. *on Theoretical Aspects of Computer Science* (STACS), 1999, Lecture Notes in Computer Science 1563, Springer-Verlag, Berlin, pp. 362–372.

[15] M. L. Fredman, The complexity of maintaining an array and computing its partial sums, *J. Assoc*. *Comput*. *Mach*. 29:1 (1982), 250–260.

[16] M. L. Fredman and M. Rauch Henzinger, Lower bounds for fully dynamic connectivity problems in graphs, *Algorithmica* 22:3 (1998), 351–362.

[17] M. L. Fredman and M. E. Saks, On the cell probe complexity of dynamic data structures, *Proc*. 21*st Annual ACM Symp*. *on Theory of Computing*, Seattle, WA, 1989, pp. 345–354.

[18] H. N. Gabow, Data structures for weighted matching and nearest common ancestors with linking, *Proc*. 1*st Symp*. *on Discrete Algorithms* (SODA), 1990, pp. 434–443.

[19] Z. Galil and G. F. Italiano, Data structures and algorithms for disjoint set union problems, *ACM Comput*. *Surveys* 23:3 (1991), pp. 319–344.

[20] T. Hagerup, Sorting and searching on the word RAM, *Proc*. 15*th Symp*. *on Theoretical Aspects of Computer Science* (STACS), 1998, Lecture Notes in Computer Science 1373, Springer-Verlag, Berlin, pp. 366–398.

[21] J. Heinz and C. P. Schnorr, Testing polynomials which are easy to compute, *Proc*. 12*th Annual ACM Symp*. *on Theory of Computing* (STOC), Los Angeles, CA, 1980, pp. 262–272.

[22] T. Husfeldt, T. Rauhe and S. Skyum, Lower bounds for dynamic transitive closure, planar point location, and parentheses matching, *Nordic J. Comput*. 3 (1996), 323–336.

[23] K. Kendig, *Elementary Algebraic Geometry*, Springer-Verlag, New York, 1977.

[24] P. Klein and F. Meyer auf der Heide, A lower time bound for the knapsack problem on random access machines, *Acta Inform*. 19 (1983), 385–395.

[25] J. A. La Poutré, New techniques for the union-find problem, *Proc*. *First Annual ACM–SIAM Symp*. *on Discrete Algorithms* (1990), pp. 54–63. Full version: Technical Report RUU-CS-89-19, Department of Computer Science, Utrecht University, 1989.

[26] F. Meyer auf der Heide, Lower bounds for solving linear diophantine equations on random access machines, *J. Assoc*. *Comput*. *Mach*. 32:4 (1985), 929–937.

[27] W. Paul and J. Simon, Decision trees and random access machines, in *Logic and Algorithmic*, monographie n$^{o}$ 30 de l’enseignement mathématique, Université de Genève, 1982; see also K. Mehlhorn, *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*, Springer-Verlag, New York, 1984, pp. 85–94.

[28] F. P. Preparata and M. I. Shamos, *Computational Geometry*: *An Introduction*, 2nd printing, Springer-Verlag, 1985/1988.

[29] M. H. M. Smid, A data structure for the Union-Find problem having good single-operation complexity, in *ALCOM*: *Algorithms Review*, Newsletter of the ESPRIT II Basic Research Actions Program Project no. 3075 (ALCOM), 1990.

[30] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[31] H. E. Warren, Lower bounds for approximation by nonlinear manifolds, *Trans*. *Amer. Math*. *Soc*. 133:1 (1968), 167–178.