On-line reevaluation of functions

Peter Bro Miltersen*
Aarhus University, Computer Science Department
Ny Munkegade, DK 8000 Aarhus C.
pbmiltersen@daimi.aau.dk

January 1992

Abstract

Given a finite set S and a function $f : S^n \to S^m$, we consider the problem of making a data structure which maintains a value of $x \in S^n$ and allows us to efficiently change an arbitrary coordinate of x and efficiently evaluate $f_i(x)$ for arbitrary i. We both examine the problem for specific choices of f and relate the possibility of an efficient solution to general properties of f: expressibility as a formula, space complexity and time complexity.

1 Introduction

Let S be a finite set, and let $f : S^n \to S^m$ be a function, $f(x) = (f_1(x), f_2(x), \ldots, f_m(x))$. In this paper we consider the problem of *reevaluating f on-line* which is the implementation of a data structure with the following properties: The data structure is supposed to maintain the value of a vector $x \in S^n$. We are given a vector $x = x_1x_2 \ldots x_n \in S^n$ and initialize the data structure to

*Work partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

reevaluation of f:

`memory`

$x = (x_1, x_2, \ldots, x_n) \in S^n$

`operations`

$\underline{update(i, s), i \in 1, \ldots, n, s \in S}$

$x_i \leftarrow s$

$\underline{query(j), j \in \{1, \ldots, m\}}$

`return` y_j, where $(y_1, \ldots, y_m) = f(x_1, x_2, \ldots, x_n)$

Figure 1: On-line function reevaluation

this x. After that we are given a sequence of *update*- and *query*-instructions, which we must serve on-line. An $update(i, s)$-instruction with $1 \leq i \leq n$ and $s \in S$ changes the value of x_i to s. A $query(i)$-operation with $1 \leq i \leq m$ returns the value of $f_i(x)$ for the current value of x. Our goal is to attain a speed-up over the naive method of simply storing the value of x and evaluating f from scratch. Figure 1 summarizes the specification of the data structure. If $S = \{0, 1\}$ and $m = 1$ we get the important special case of *on-line rerecognition* of a boolean language. Several natural problems in the literature can be described as on-line reevaluation.

- Maintenance of the connected components of an undirected graph, considered by Frederickson [5], who also lists earlier papers. We are given an undirected graph and are allowed to
  - Insert edges.
  - Delete edges.
  - Ask whether two vertices are in the same connected component.

  Clearly, this problem is equivalent to online reevaluation of f, where f is the function which takes the adjacency matrix of an undirected graph and returns the adjacency matrix of its transitive closure.

- Other dynamic graph or hypergraph problems, e.g. existence of paths, transitive reduction and hypergraph connectivity [2, 11, 12, 15]. In these cases, the solutions often only apply to restricted problems where only either insertions of edges or deletions of edges are allowed. This

corresponds to restricting the update-operation in the reevaluation problem so that only updating to certain values is allowed.

- Maintenance of partial sums of an array. In this case the x_i's are member of some semigroup S and $f_j(x) = \prod_{i=1}^{j} x_i$. Lower bounds for this problem in various models have been derived by Fredman [6], Yao [22] and Fredman & Saks [7].

The concept of on-line reevaluation seems to be a natural way of looking at these problems and a natural source for new data structure problems. In this paper, we will examine the problem for both specific functions and for classes of f defined by restricting the complexity of f in some conventional complexity measure.

2 Models of computation

Since our interest is speed up in a sequential computation, the choice of a model of computation seems significant. Several models have been considered for the complexity of data structures.

- The *RAC*, defined by Angluin & Valiant [1]. This is a random access machine with word size bounded by a function of the input size, usually $O(\log n)$. The operations on each word can be performed in constant time. The motivation of the model is to prevent unnatural operations within a word, while preserving unit cost manipulation of small data.

- The *Cell probe model*, considered by Yao [21] and Fredman & Saks [7]. In this model the data structure is a collection of words. Reading a word or changing a word is charged one unit, but computation is free. The combinatorial flavor of this model and the fact that it focuses on access of data makes it very appealing. Furthermore, a cell probe machine with word size $\log n$ can simulate a RAC, so lower bounds in the cell probe model carries over to the realistic RAC-model. The cell probe model with word size 1 is obviously a canonical model for the access complexity of data structures as it is independent of any architectural considerations. We refer to this model as the *bit probe model*.

- The *pointer machine*, by Tarjan [19] and others. In this model, the data structure is a potentially unbounded collection of records with a bounded number of fields which are pointers to other records. A pointer can be traced in constant time. It is easy to see that without restrictions on space or initialization time, this property makes it possible to reevaluate any computable function on n variables on-line in time $O(\log n)$ by precomputing every value and organizing them in an exponential size data structure. This approach is not possible in any of the two other models, since a linear number of bits is required to store a pointer in the data structure. In the RAC-model, exponential initialization time only enables us to reevaluate in time $O(\frac{n}{\log\ n})$, as is shown below.

For the rest of the paper, we adopt the RAC-model with word size $O(\log n)$, except where otherwise is explicitly stated, and we will say that a function f can be reevaluated on-line in time t if t is a worst case time bound for an update or query operation in a data structure for the reevaluation problem in this model. Furthermore we will usually require that the initialization time is polynomial. However, some observations on the bit probe model are useful. Let $B(f)$ be the worst case operation complexity of an optimal implementation of a bit probe structure for reevaluating $f : S^n \to S^m$. For concreteness, assume $S = \{0, 1\}$. Since we are only charged for accessing the data structure, the naive implementation, which simply keeps the value of x stored and reads it to compute a value $f_i(x)$ when required, has bit probe complexity n. Thus every f has $B(f) \leq n$. We can actually do slightly better than this.

Theorem 1 *For all functions* $f : \{0, 1\}^n \to \{0, 1\}^m$,

$$B(f) \leq n - \lfloor \log_2 \log_2 n \rfloor + 2$$

Proof Let $r = \lfloor \log_2 \log_2(n - 2\log_2 \log_2 n) \rfloor$. We consider a data structure for maintaining $x \in \{0, 1\}^n$ consisting of two parts.

- The first $n - r$ bits of x.
- For each predicate $p : \{0, 1\}^r \to \{0, 1\}$, the value of p on the final r bits of x.

In order to change one of the first $n - r$ bits of x, we need only touch 1 bit in the structure. In order to change one of the final r bits in x we read all r bits, and recompute the value of every predicate on them. We thus have to touch $r + 2^{2^r} \leq n - \log_2 \log_2 n$ bits. In order to evaluate $f_j(x)$, we read the first $n - r$ bits of x, let these be the string x_1. Let p be the predicate defined by $p(x_2) = f_j(x_1 x_2)$. The value of p on the find r bits of x is the value of $f_j(x)$. This value can be read directly in the data structure. Thus, evaluation requires $n - r + 1$ probes.

□

For most functions the upper bound can not be improved much, as the following theorem shows.

Theorem 2 *Let* $f : \{0,1\}^n \rightarrow \{0,1\}$ *be a* random *function. With probability at least* $1 - 2^{-2^{n-1}}$, *it holds that* $B(f) > n - 2 \log_2 n - 5$.

Proof We can without loss of generality assume that 0^n is the only legal initial value for x. Let us call a bit probe algorithm which clears its binary memory to zero when initialized on input 0^n for *normal*. Given a bit probe algorithm A for f, we can convert it into an equivalent normal algorithm in the following way. Suppose A initializes its memory to the binary vector $\mathbf{y}$. We then define the algorithm A' which clears its memory when initialized to 0^n. When A reads the memory location z_i, A' reads z_i and computes $z_i \oplus y_i$, where $\oplus$ denotes exclusive or. When A stores the value b in z_i, A' stores $b \oplus y_i$. Since the values y_i are constant, we are not charged for using them, and A' clearly has the same external behavior as A. So we can restrict our attention to normal algorithms, which are completely specified by the implementation of the $2n + 1$ possible operations, which in turn can be specified by binary trees with three types of nodes:

- Binary read-nodes labeled with an address in the memory. When such a node is encountered the address is read, and dependent on whether the content is 0 or 1, the algorithm proceeds to the left or the right son.
- Unary write-nodes labeled with an address and a binary value. When such a node is encountered, the binary value is written at the address, and the algorithm proceeds to the son of the node.

- Leaves, labeled with a binary answer in case the operation is query, and not labeled otherwise.

A bit probe algorithm running in time at most t is described by $2n+1$ trees of height at most t, one tree for each possible update operation and one for the query operation. The total number of nodes in these trees is at most $(2n+1)(2^{t+1}-1) \leq 6n2^t$. A node can be specified by an address and an integer from 1 to 5, indicating whether it is a zero leaf, a one leaf, a read node, a write zero node or a write one node. If we without loss of generality assume the addresses are numbered $1, \ldots, 6n2^t$, there are at most $(30n2^t)^{6n2^t}$ bit probe algorithms running in time at most t. The number of bit probe algorithms running in time at most $n - 2\log_2 n - 5$ is thus at most $2^{2^{n-1}}$. There are, however, 2^{2^n} functions of n variables, so the probability that a random function has bit probe complexity at most $n-2\log_2 n-5$ is at most $\frac{2^{2^{n-1}}}{2^{2^n}} = 2^{-2^{n-1}}$.

□

Corollary 1 *Most functions f on n variables requite time $\Omega(\frac{n}{\log n})$ to be reevaluated on-line on a RAC, even when an arbitrary amount of initialization based on a table for f is allowed.*

Proof A RAC-algorithm running in time $t(n)$ can be simulated by a bit probe algorithm running in time $t(n)\log n$.

□

By similar counting arguments we can also prove lower bounds for concrete functions. Let f be a boolean function on a variable set X. A subfunction of f on $Y \subseteq X$ is a function obtained from f by setting the variables of $X - Y$ to constants [3].

Theorem 3 *Let $f : \{0,1\}^n \to \{0,1\}$ be a function so that f supports s different subfunctions on a set of variables Y of size m. Then*

$$B(f) \geq \log_2 \log_2 s - \log_2 \log_2 \log_2 s - \log_2 m - \log_2 \log_2 m - 5$$

Proof If $s \leq 2^{2^5}$, there is nothing to prove, so we will assume $s > 2^{2^5}$ and thus $m \geq 6$. As in the previous proof, there are at most $(30m2^t)^{6m2^t}$ different bit probe algorithms running in time at most t for functions on m variables. Since we can obtain bit probe algorithms for s different functions on m variables by restricting the algorithm for f, we must have $(30m2^t)^{6m2^t} \geq s$, where $t = B(f)$. When $m \geq 6$ this implies

$$t \geq \log_2 \log_2 s - \log_2 \log_2 \log_2 s - \log_2 m - \log_2 \log_2 m - 5$$

□

Let $access_n : \{0,1\}^{n+\lceil \log_2 n \rceil} \rightarrow \{0,1\}$ be the function which takes as input a bit vector $x = x_0 x_2 \ldots x_{n-1} \in \{0,1\}^n$ and a binary number $y \in \{0,1\}^{\lceil \log_2 n \rceil}$ denoting an integer i and outputs x_i. This function describes reading a random access memory. An obvious upper bound on $B(access_n)$ is $\log_2 n$. Let $disitinct_n : \{0,1\}^{2n \log_2 n} \rightarrow \{0,1\}$ be the function which takes as input n boolean strings of length $2 \log_2 n$ and outputs 1 if and only if all strings are distinct. By using a nonstandard representation of the integers to represent the number of occurences of each possible string, it is possible to obtain $B(access_n) = 2 \log_2 n + o(\log_2 n)$. Theorem 3 provides good lower bounds for these functions.

Corollary 2

$$B(access_n) \geq \log_2 n - o(\log n)$$

$$B(distinct_n) \geq \log_2 n - o(\log n)$$

It is, however, easy to see that theorem 3 is unable to provide larger lower bound on boolean functions on n variables than $\log_2 n - o(\log n)$. Fredman & Saks gives a lower bound of $\Omega(\log n / \log \log n)$ for the bit probe complexity of another specific boolean function (described in section 3). We do not know of a better lower bound than $\Omega(\log n)$ for any easily computed function. Theorem 2 gives us a function $f \in SPACE(2^{n^{O(1)}})$ with $B(f_n) = n - o(n)$ for every n (where f_n is f restricted to $\{0,1\}^n$), because we can within these space bounds store the decision trees of a linear time bit probe algorithm. We can then find the lexicographically first optimal algorithm whose complexity is at least $n - 2 \log_2 n - 5$ by existentially guessing such an algorithm, universally guessing a faster algorithm and existentially guessing a sequence of

operations which we simulate on the two algorithms and verify that they give a different answer at some point. This eastablishes the optimality of the first algorithm. We then universally guess a lexicographically smaller algorithm of the same complexity and verify that this is not optimal. Finally we use the first algorithm to evaluate f. Since alternating space with a bounded number of alternations can be simulated with a polynomial overhead in deterministic space, thy entire algorithm is in $SPACE(2^{n^{O(1)}})$. It seems hard to arrive on anything better, even if we move to the more restrictive RAC-model. Thus, a situation similar to the one in boolean complexity theory arises: It is known that most functions has maximal complexity, but it is hard to obtain large lower bounds for any particular, simple to define function. Despite this difficulty, for many of the problems mentioned in the intruduction no sublinear time data structure is known. It is therefore natural to conjecture that there exists functions which are easy to evaluate but do not possess very efficient data structures for on-line reevaluation. This conjecture is easily formalized.

Definition 1 *Let L be a predicate on $\{0,1\}^*$. Let L_n be the restriction of L to $\{0,1\}^n$. If there exists a RAC-algorithm which initializes on input n and $x \in \{0,1\}^n$ in time polynomial in n and thereafter rerecognizes L_n with a worst case operation time which is polylogarithmic in n, L is said to be* dynamic. *The class of dynamic languages is denoted by D.*

Conjecture 1 *P is not included in D.*

The restriction to polynomial time initialization is made to ensure that $D \subseteq P$. Polylogarithmic worst case operation time is invariant under reasonable changes of the model, and has been regarded as the criterion of fastness in domains such as set maintenance and computational geometry. Observe that if $D = P$, any problem in P can be solved in quasi-linear time on a RAC if polynomial time preprocessing on the input size is allowed.

Since the conjecture seems difficult to solve we will, in analogy with complexity theory, find a notion of reduction which makes it possible to state completeness result. Let us consider the following concept of an *oracle RAC*. Let $f = f_1, f_2, \ldots$ be a family of function such that the domain of f_n is $\{0,1\}^n$. A RAC using oracles for f is a RAC equipped with an unbounded number of hardware (oracle) implementations of data structures for the on-line reevaluation problem for f_m, for all $m \leq p(n)$, where $p(n)$ is a polynomial

in the input size n. The data structures for f_m are all initialized to 0^m when the RAC starts initializing on some input $x \in \{0,1\}^n$. The operations on the oracle data structures are, similarly to the other basic hardware operations on the RAC, assumed to be carried out in constant time.

Hardware implementation of certain general useful data structures is a quite natural model, given the current state of VLSI, Implementations of dictionaries in special purpose architectures has been the object of much research, see e.g. Ottman et al. [14]. The class of data stucture problems which can be solved efficiently given a supposed fast hardware structure gives us a measure of the usefulness of this structure.

Definition 2 *Let $g = g_1, g_2, \ldots$ and $f = f_1, f_2, \ldots$ be families of functions. If there exists a RAC-algorithm using oracles for g which initializes on input n and $x \in \{0,1\}^n$ in time polynomial in n and thereafter reevaluates f_n with a worst case operation time which is polylogarithmic in n, we say that $g \leq_D f$. If $g \leq_D f$ and $f \leq_D g$, we say that $g \equiv_D f$.*

It is easily seen f and g are boolean predicates such that $g \in D$ and $f \leq_D g$, then $f \in D$. As we shall see in section 6, this concept of reduction makes it possible to state natural problems, equivalent to $D \neq P$. For now, we will show that the restriction to predicates in the definition of D is insignificant.

Proposition 1 *Each polynomial time computable function $f : \{0,1\}^n \to \{0,1\}^m$ is $\leq_D$-reducible to a polynomial time computable predicate p.*

Proof Let y be a binary string of length $\lceil \log_2 m \rceil$ and define $p(xy) = 1$ if and only if $f_i(x) = 1$, where y is the binary notation for i. Assume we have a polylog-implementation of the rerecognition problem for p. We can ask for the value of $f_i(x)$ for any i by changing y, which is only a logarithmic number of changes, each taking polylogarithmic time.

□

3 On-line reevaluation of formulae

In boolean complexity theory, an important complexity measure of a function f is its formula size $L(f)$ [16]. In this section, we prove that if a function is definable by a formula where each variable does not appear too many times, f can be reevaluated efficiently on-line.

Let S be a discrete set. A *formula* F of order d over S is a tree where each node has a number of sons $\leq d$, this number is called the *arity* of the node. Each node of arity i contains a function $S^i \to S$, each leaf contains an element in S. The value of a leaf is its content and the value of a node is defined inductively in the usual way. Assume that the tree contains n leaves and m nodes and let the contents of the nodes be fixed. The *generalized evaluation function* $\tilde{F}$ is the function $S^n \to S^m$ which takes as input the value of each leaf and returns the value of each node. We consider the problem of reevaluating $\tilde{F}$ on-line.

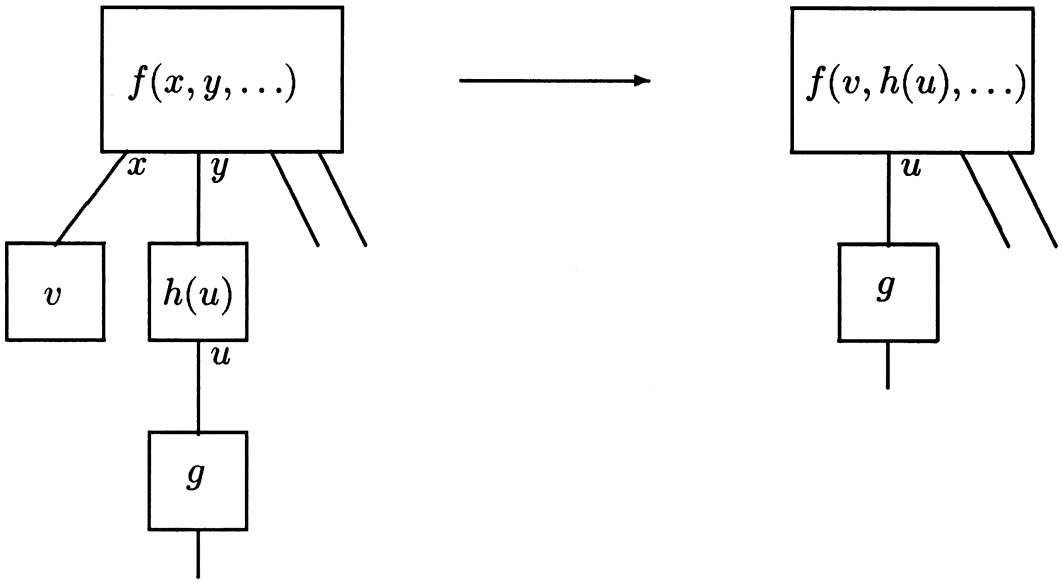

Figure 2: Tree contraction

Theorem 4 *$\tilde{F}$ can be reevaluated in time $c \log n$ where c is a constant, dependent on $|S|$ and d.*

Proof The data structure is based upon the well known *parallel* algorithm for expression evaluation [8]. First we define the *contraction* F' of a formula F. A node in F is called *removable* if it has arity 1 and its father is not removable. F' is obtained from F by removing all leaves and all removable nodes and modifying the content of the fathers of the removed leaves and nodes as shown in figure 2 (Since S is finite, functions of bounded arity has a bounded representation).

It is easy to see that if a node is not removed, its value in F is equal to its value in F'. To solve the on-line reevaluation problem, we maintain the trees $F, F', F'', \ldots, F^{(i)}$, where the size of $F^{(i)}$ is 1. It is easy to see that the size of F' is at most about half the size of F, so $i = O(\log n)$. The structure of . the trees does not change when the input is changed, so we only has to maintain the content of each node. To do this, consider the more general situation where we are also allowed to change the content of a node in F. If the node is not removed in F', we recursively change the value of the same node in F'. If it is removed, we calculate the new content of its father in F' and recursively change the value of this node in F'. Thus, the complexity of changing the value of any node or leaf in F is $O(\log n)$. Next, consider the problem of evaluating any node or leaf v in F. If v is a leaf, its value is equal to its content. If v is a node which is not removed in F', we recursively evaluate the corresponding node in F'. If v is removed in F', its son is not, so we recursively evaluate the son in F' and apply the content of v to the computed value. Thus, the complexity of evaluating any node is $O(\log n)$.

□

We might note that similarly to the parallel algorithm the technique extends to on-line reevaluation of formulae over $\{+, -, \cdot, /\}$ on an arithmetic RAM, since the functions which arise have a finite representation. For details, see Gibbons & Rytter [8]. Also, it has been brought to the author's attention that a more complex data structure for a much more general problem has recently been constructed by Cohen & Tamassia [4].

Corollary 3 *Regular languages are dynamic.*

Proof If $L \subseteq \Sigma^*$ is recognized by a finite automaton with state set Q,

initial state s and transition function $\circ : Q \times \Sigma \to Q$, the membership problem $x_1x_2x_3 \ldots x_n \in L$ is reduced to evaluating $(\cdots((s \circ x_1) \circ x_2) \circ \cdots \circ x_n)$ and checking if the result is a final state.

□

Corollary 4 *The sum of two binary numbers can be reevaluated on-line in time* $O(\log n)$.

Proof The digits in the sum of two binary numbers can be computed by a finite state machine which scans the input from right to left. The value of each digit can thus be computed from a subresult of a formula, similarly to the one in corollary 3. The algorithm is similar to the *carry look ahead* addition algorithm in parallel computation.

□

The problem of maintaining the partial sums of an array can be regarded as on-line reevaluation of the partial results of the formula $(\cdots((x_1 \circ x_2) \circ x_3) \circ \cdots \circ x_n)$, where $\circ$ is the semigroup operation. If the semigroup is exclusive or over the booleans, Fredman & Saks [7] show a lower bound of $\Omega(\log n \log\log n)$ in the $O(\log n)$-word size cell probe model. Interestingly, if exclusive or is replaced by inclusive or, an $O(\log\log n)$-implementation in the RAC-model with word size $O(\log n)$ and hence also in the cell-probe model becomes possible, by the priority queue of Van Emde Boas, Kaas & Zijlstra [20]. If the bit probe model is used, the author know of no implementation of either problem with better bounds than $O(\log n)$.

4 On-line reevaluation of space bounded computations

Let T be a Turing machine with a read-only input tape, and a number of work tapes. T may also be equipped with a write-only output tape. The input $x_1x_2 \ldots x_n$ are given on the input tape, surrounded by blanks. We adopt the following absolute measure of the space s used by a computation: s is the

number of bits needed to describe the state of the finite control, the content of each work tape with the input tape excluded and the position of each work tape head with the input tape head excluded (if we furthermore allow the Turing machine to be nonuniform, this is the nonuniform space measure of Ibarra & Ravikumar [9]). Standard techniques give that an $s(n)$-space bounded computations can be simulated from scratch on a RAC in time $O(n2^{s(n)})$, if the Turing machine is deterministic and in time $O(n(2^{s(n)})^2)$ if it is not. We consider in this section the problem of reevaluating a space bounded computation on-line. Corollary 3 extends in the obvious way to on-line space-bounded computations (where the input tape head is allowed to move right only). In this section we shall prove results about off-line space bounded computations (where the input tape head is allowed to move in both directions).

Theorem 5 *Let L be a language, accepted in space $s(n)$ on a Turing machine M. If M is deterministic, L can be rerecognized on-line in time $O((\log\ n)2^{s(n)})$ with initialization time $O(n2^{s(n)})$. If M is nondeterministic, L can be rerecognized on-line in time $0((\log\ n)2^{\alpha s(n)})$ with initialization time $O(n2^{\alpha s(n)})$, where α is a constant so that transitive closure of an $n \times n$ boolean matrix can be found in time $O(n^\alpha)$.*

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

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

enters semi-configuration v.

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

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

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

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

$$V = S \times \{l, r_1, l_2, r\}$$

and edges given by the following rules

$$\begin{aligned}
(u,l)R_1(v,l) &\Rightarrow <(u,l),(v,l)> \in E \\
(u,l)R_1(v,r) &\Rightarrow <(u,l),(v,l_2)> \in E \\
(u,r)R_1(v,l) &\Rightarrow <(u,r_1),(v,l)> \in E \\
(u,r)R_1(v,r) &\Rightarrow <(u,r_1),(v,l_2)> \in E \\
(u,l)R_2(v,l) &\Rightarrow <(u,l_2),(v,r_1)> \in E \\
(u,l)R_2(v,r) &\Rightarrow <(u,l_2),(v,r)> \in E \\
(u,r)R_2(v,l) &\Rightarrow <(u,r),(v,r_1)> \in E \\
(u,r)R_2(v,r) &\Rightarrow <(u,r),(v,r)> \in E
\end{aligned}$$

Let R be the binary relation on $S \times \{l, r\}$ defined by taking the transitive closure of G and restricting it to $S \times \{l, r\}$. We refer to R as the concatenation $R_1 * R_2$ of R_1 and R_2. It is easily seen that if $i < k < j$ then $R_{i,j} = R_{i,k} * R_{(k+1),j}$. Suppose $|S| = n$. Given adjacency matrices of R_1 and R_2, an

adjacency matrix for $R_1 * R_2$ can be computed in time $O(n^\alpha)$. If R_1 and R_2 are functions, $R_1 * R_2$ is a function too, and a table for R can be computed from tables for R_1 and R_2 in time $O(n)$. This suggests the following data structure: Given a tape segment $x_i \dots x_j$, we maintain a representation of the relation $R_{i,j}$ by recursively maintaining $R_{i,k}$ and $R_{k+1,j}$ where $k = \lfloor \frac{i+j}{2} \rfloor$. Since we keep a representation of $R_{0,n+1}$ we can decide membership of L in constant time. When a letter in the input x_i is changed, we only have to recompute each of the $R_{j,k}$ in the data structure for which $j \leq i \leq k$, i.e. $\log_2 n$ $*$-computations on relations of size $2^{s(n)}$. This can be done in time $O((\log n)2^{s(n)})$ if the relations are functional, i.e. if the Turing Machine is deterministic, and in time $O((\log n)2^{\alpha s(n)})$ in the general case.

□

In the deterministic case, we thus achieve a speedup of $O(\frac{n}{\log n})$ over the direct simulation when reevaluating. In tne nondeterministic case we only get a speedup when $s(n) < \frac{1}{\alpha-2} \log n$. Unfortunately, this does not imply speedup results for the languages themselves, since the direct simulation of the space bounded machine may be (and usual is) a non-optimal way of evaluating the function with respect to time. If we get sublinear time, we are however usually certain of having achieved a speedup, and from a bit probe perspective, the theorem is only interesting in this case. For languages computable in space $c \log n$ for small constants c, we get on-line rerecognition in time $O(n^\beta)$ with $\beta < 1$. For languages accepted in sublogarithmic space we get on-line rerecognition in time $n^{o(1)}$. Unfortunately, interesting examples of such languages are hard to come by. For general discussion on sublog languages, see Ibarra & Ravikumar [9].

Corollary 5 *If L is accepted by an off-line Turing machine with a one dimensional input tape running within space $O(\log \log n)$, L is dynamic.*

The technique extends to a d-dimensional input tape. In the case $d = 2$, if the input is given in a square, we may divide the square into 4 subsquares and consider traffic between these subsquares. However, since the boundaries of any segments we might divide the tape into are large, we get worse time bounds.

Theorem 6 *Let L be a language, accepted in space $s(n)$ by a deterministic*

Turing machine M with an input tape of dimension d on which the input is given in a d-cube. L can be rerecognized on-line in time $O((\log n)n^{\frac{d-1}{d}}2^{s(n)})$.

By using a different kind of a relations and a different notion of concatenation, we can extend the theorem to hold with similar bounds for alternating Turing machines with a bounded number of alternations. The details are tedious, but not difficult.

Theorem 7 *Let L be a language, accepted in space $s(n)$ on an alternating Turing machine M with a bounded number of alternations. L can be rerecognized on-line in time $O((\log n)2^{O(s(n))})$.*

This is a nontrivial extension only in the case where $s(n) = o(\log n)$ since space bounded computations with larger bounds are known to be closed under complementation by the result of Immerman and Szelepcsényi [10, 18]. As a final extension, we consider evaluating fictions, rather than recognizing languages.

Theorem 8 *Let f be a function, computed in space $s(n)$ on a deterministic Turing machine transducer M. f can be reevaluated on-line in time $O((\log n)2^{s(n)})$.*

Proof With definitions as in the proof of the previous theorem, define the function $f_{i,j} : S \times \{L, R\} \to S \times \{L, R\} \times \mathbf{N}$ by $f_{i,j}(u, D) = (v, E, t)$ if and only if $(u, D)R_{i,j}(v, E)$ and the output tape head is moved t cells to the right during the computation. It is easy to extend the definition of concatenation to such functions in such a way that $f_{i,j} = f_{i,k} * f_{(k+1),j}$. We maintain a finite representation of functions $f_{i,j}$ as in the previous proof. When we are asked the value of a specific output cell, we use the stored functions to track the place in the computation where the cell was written and what was written there. □

5 On-line rerecognition of Dyck Languages

Let D_1 be the language of matching parentheses over $\{(,)\}$, and let D_2 be the language of matching parentheses over $\{(, [,),]\}$. We shall consider rere-

cognizing these two languages on-line, and show that while D_1 is dynamic, D_2 appears to be harder. This is interesting, since the two language are of comparable difficulty from almost any perspective, sequential (both are recognizable in linear time by deterministic pushdown automata) or parallel (both are complete for TC^0).

Theorem 9 *D_1 can be rerecognized on-line in time $O(\log n)$.*

Proof Let x be a string over (and). Let $l(x)$ denote the number of unmatched left parentheses in x and let $r(x)$ denote the number of unmatched right parentheses in x. Suppose $x = yz$. We then have that

$$\begin{aligned} r(x) &= r(y) + \max(0, r(z) - l(y)) \\ l(x) &= l(z) + \max(0, l(y) - r(z)) \end{aligned}$$

Since $x \in D_1$ if and only if $l(x) = r(x) = 0$, the recursion immediately suggests an $O(\log n)$ RAC-implementation of D_1. The bit probe complexity is $O(\log^2 n)$.

□

No such simple approach works for D_2. We can, however, use the same approach to reduce the problem to an apparently simpler one. Let x be a string on $\{0, 1, \#\}$. Define the string $x/\#$ to be x with #'s removed and the rest of the characters kept in the origins order. Let L be the language defined by

$$L = \{xy || x| = |y| \wedge x/\# = y/\#\}$$

The best algorithm the author is aware of for rerecognizing L is due to Sven Skyum [17] and has worst case operation complexity $O(n^{0.67})$. We conjecture that L is not dynamic.

Theorem 10

$$L \equiv_D D_2$$

Proof

- $L \leq_D D_2$: Suppose we want to decide membership of xy in L, $|x| = |y|$. We transform this to an instance of D_2 in the following way: Let $\tilde{x}$ be

defined by replacing all occurrences of 0 in x by ((, all occurrences of 1 by [[and all occurrences of # by (). Let $\tilde{y}$ be defined by replacing all occurrences of 0 in y by)), all occurrences of 1 by]] and all occurrences of # by (). Let $\tilde{y}^R$ be the string $\tilde{y}$ reversed. Now $xy \in L$ if and only if $\tilde{x}(\tilde{y}^R) \in D_2$. Furthermore, if we change a letter in x or y, we change only 2 letters in $\tilde{x}(\tilde{y}^R)$. Therefore, given an oracle implementation of the on-line rerecognition problem for D_2, we get a polylog time implementation for L.

- $D_2 \leq_D L$: By using the same data structure as for D_1, we can decide if the string would match correctly if any left parenthesis could match any right parenthesis. We then only need to recognize the language D' which consists of strings where parentheses which are matched by some other parenthesis is matched by the correct kind. Let x be a string with the two kinds of parentheses, and let $x = yz$. We may divide the parentheses in y into 3 classes,

  1. The ones which are matched by a parenthesis in y, correctly or incorrectly.
  2. The ones which are matched by a parenthesis in z, correctly or incorrectly.
  3. The ones which are not matched by any parenthesis in x.

  Similarly for the parentheses in z. Let $\tilde{y}$ be the string where each (of type 2 in y has been replaced by a 0, each [of type 2 has been replaced by a 1 and each parenthesis in y which is not of type 2 has been replaced by a #. Let $\tilde{z}$ be the string where each) of type 2 in z has been replaced by a 0, each] of type 2 has been replaced by a 1 and each parenthesis in y which is not of type 2 has been replaced by a #. We now see that $x \in D'$ if and only if $y \in D'$, $z \in D'$ and $\tilde{y}(\tilde{z}^R) \in L$. By using oracle implementations of L the problem is then solved, if we are able to maintain $\tilde{x}$ for each string x in the tree. But this is easily seen to be possible if we maintain the positions of each of the 3 types of parentheses of x in three balanced binary search trees.

□

6 On-line rerecognition of languages in P

As mentioned in the introduction, the question of whether all polynomial time computable problem admits fast reevaluation seems important. Here we discuss a complete problem for this class which has been considered in the literature as a data structure problem in its own right. Consider the problem of maintaining the closure of a directed hypergraph. The following definitions are taken from [2], where data structures and algorithms for this problems are given.

Definition 3 *A* directed hypergraph *H is a pair (V, E) where V is a set of* nodes *and E is a set of* hyperedges. *Each hyperedge is an ordered pair (X, i) from an arbitrary nonempty set $X \in P(V)$ (source set) to a single node $i \in V$ (target node).*

Definition 4 *Given a hypergraph $H = (V, E)$, a nonempty subset of nodes $X \subseteq V$ and a node $i \in V$, there is a (directed) hypeypath from X to i, if one of the following conditions holds:*

1. *$i \in X$ (extended reflexivity), or*
2. *there is a hyperedge $(Y, i) \in E$ and for each node $j \in Y$ there exkts a hyperpath from X to j (extended transitivity).*

Let us restrict ourselves to hypergraphs with bounded edge size where $|x| \leq d$ for each $(X, i) \in E$. A hypergraph with bounded edge size can be represented by a polynomial size adjacency matrix, since less than $|V|^{d+1}$ hyperedges are possible.

Definition 5 *Given a hypergraph $H = (V, E)$ with an edge size bound d, we define the hypergraph $H^* = (V, E^*)$ where $(X, i) \in E^*$ iff $|X| \leq d$ and there is a hyperpath from X to i in H. H^* is called the* restricted transitive closure *of H.*

Note that the case $d = 1$ corresponds to transitive closure of an ordinary, directed graph. Let $hclose_n^d$ be the function which given the adjacency matrix of a hypergraph H with $|V| = n$ returns the adjacency matrix of H^*, and let us consider the problem of reevaluating $hclose_n^d$ on-line. Algorithms and

data structures for restrictions of this problem are given by Ausiello et al. [2].

Theorem 11 *$hclose_n^d$ is $\leq_D$-hard for P for $d \geq 2$.*

Proof The proof follow the proof of completeness for P of Generability with respect to logspace-reduction by Jones & Laaser [13]. Let L be a boolean language in P, and let C_n be a polynomial size straight line program for deciding instances of L of size n with input variables $x = x_1 \ldots x_n$ and internal variables $v_1, \ldots, v_m$, where v_m is the output variable. Given an input $y_i \in \{0,1\}^n$, consider the following hypergraph:

$$V = \bigcup_{i=1}^{m} \{(v_i, 0), (v_i, 1)\} \cup \bigcup_{i=1}^{n} \{(x_i, 0), (x_i, 1)\} \cup \{s\}$$

$$E = \bigcup_{v_i \leftarrow v_j * v_k, y, z} \{(\{(v_j, y), (v_k, z)\}, (v_i, y * z))\} \cup \bigcup_i \{(s, (x_i, v(x_i))\}$$

where $v(x_i)$ is the value of x_i in the current assignment of x. Clearly, there is a hyperpath from $\{s\}$ to $(v_m, 1)$ if and only if $x \in L$. Furthermore, if the input changes on one bit, only one edge has to be removed and one has to be inserted in the hypergraph. Thus, L_n can be reevaluated on-line in polylogarithmic time, given oracle implementations of $hclose_{2m+n+1}$.

□

Corollary 6 *$hclose^j \equiv_D hclose^i$ for any $i, j \geq 2$.*

Proof $hclose^d$ is a polynomial time computable function.

□

Efficient data structures for the on-line reevaluation of the transitive closure of dynamic hypergraphs are thus of universal interest, but the completeness result makes it unlikely that an algorithm with an operation complexity of $n^{o(1)}$ exists. Several other problems which are P-complete with respect to some usual reduction, e.g. logspace-reduction, remains so with respect to $\leq_D$-reductions. This is so because in most P-completeness proofs the reduction function f does not expand the input significantly and the Hamming distance

between $f(x)$ and $f(y)$ is small when the Hamming distance between x and y is small. Unfortunately, this is usually not the case for problems complete for smaller classes, e.g. *NLOG*, so we do not know if on-line reevaluation of the transitive closure of an ordinary directed graph is $\leq_D$ complete for any interesting class of problems. This weakness of the reductions is probably a major obstacle for the development of a structural theory of dynamic problems. However, we still think that this notion reducibility deserves father study.

Acknowledgment

I sincerely thank Gudmund S. Frandsen, Erik Meineche Schmidt and Sven Skyum for many helpful discussions, comments and suggestions.

References

[1] D. Angluin, L. G. Valiant: *Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings*, J. Comp. Sys. Sci. 18 (1979) pp. 155–193.

[2] G. Ausiello, U. Nanni, G. F. Italiano: *Dynamic Maintenance of Directed Hypergraphs*, Theoretical Computer Science 72 (1990) pp. 97–117.

[3] R. B. Boppana, M. Sipser: *The Complexity of Finite Functions*, Handbook of Theoretical Computer Science, Elsevier Science Publishers B.V. (1990), pp. 757–804.

[4] R. F. Cohen, R. Tamassia: *Dynamic Expression Trees and their Applications*, Proc. 2nd Annual ACM-siam Symposium on Discrete Algorithms, San Francisco (1991), pp. 52–61.

[5] G. N. Frederickson, *Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications*, Siam J. Comp. 14 (1985), pp. 781–798.

[6] M. L. Fredman: *The Complexity of Maintaining an Array and Computing its Partial Sums*, JACM 29 (1982), 250–260.

[7] M. L. Fredman, M. E. Saks: *The Cell Probe Complexity of Dynamic Data Structures*, Proc. 21st Annual ACM Symposium on Theory of Computing, Seattle (1989), pp. 345–354.

[8] A. Gibbons, W. Rytter: *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge (1988).

[9] O. H. Ibarra, B. Ravikumar: *Sublogarithmic-space Turing Machines, Nonuniform Space Complexity, and Closure Properties*, Math. Systems Theory 21 (1988), pp. 1–17.

[10] N. Immerman: *Nondeterministic Space is Closed Under Complementation*, Siam J. Comput. 17 (1988) pp. 935–938.

[11] G. F. Italiano: *Amortized Eficiency of a Path Retrieval Data Structure*, Theoretical Computer Science 48 (1986), pp. 273–281.

[12] G. F. Italiano: *Finding Paths and Deleting Edges in Directed Acyclic Graphs*, Inf. Proc. Lett. 28 (1988) pp. 5–11.

[13] N. D. Jones, W, T. Laaser: *Complete Problems for Deterministic Polynomial Time*, Theoretical Computer Science 3 (1976) pp. 105–117

[14] T. A. Ottman, A. L. Rosenberg, L. J. Stockmeyer: *A dictionary machine (for VLSI), IEEE Transactions on Computers*, Vol. C-31 (1982), pp. 892–897.

[15] J. A. La Poutré, J. van Leeuwen: *Maintenance of Transitive Closures and Transitive Reductions of Graphs*, in Graph-Theoretic Concepts in Computer Science International Workshop WG'87, Lecture Notes in Computer Science vol. 314 (1988), pp. 106–120.

[16] J. E. Savage: *The Complexity of Computing*, J. Wiley, New York (1976).

[17] S. Skyum, Personal Communication.

[18] R. Szelepcsényi: *The Method of Forcing for Nondeterministic Automata*, Bull. European Association Theor. Comp. Sci. (Oct. 1987), pp. 96–100.

[19] R. E. Tarjan: *A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets*, J. Comp. Sys. Sci. 18 (1979), pp. 110–127.

[20] P. Van Emde Boas, R. Kaas, E. Ziljstra: *Design and implementation of an efficient priority queue*, Math. Systems Theory 10 (1977), pp. 99–127.

[21] A. C. Yao: *Should tables be sorted?*, JACM 28 (1981), pp. 615–628.

[22] A. C. Yao: *On the complexity of maintaining partial sums*, SIAM J. Comput. 14 (1985), pp. 277–289.