JOURNAL OF COMPUTER AND SYSTEM SCIENCES 18, 110–127 (1979)

# A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets*

ROBERT ENDRE TARJAN†

*Computer Science Department, Stanford University, Stanford, California 94305*

Received December 16, 1977; revised January 9, 1978

This paper describes a machine model intended to be useful in deriving realistic complexity bounds for tasks requiring list processing. As an example of the use of the model, the paper defines a class of algorithms which compute unions of disjoint sets on-line, and proves that any such algorithm requires nonlinear time in the worst case. All set union algorithms known to the author are instances of the model and are thus subject to the derived bound. One of the known algorithms achieves the bound to within a constant factor.

## 1. INTRODUCTION

Computer scientists have attempted for many years to derive lower bounds on the complexity of computational problems. This effort has met with some success, providing, for example, exponential lower bounds on the complexity of equivalence for regular expressions [13], validity in Presburger arithmetic [14], and circularity in attribute grammars [7]. In addition to these bounds for hard problems, several results for simpler problems exist, including bounds on the number of comparisons required for ordering problems [9], on the number of data accesses required for testing properties of graphs [15], and on the number of arithmetic operations required for evaluating various polynomials [2].

In spite of this progress, one domain, that of list-processing problems, is almost enitrely devoid of lower-bound results. Though the subject of data structures is now part of the standard computer science curriculum, and every computer science library contains many books on the subject, with the exception of a few results on the relative power of various data structures, nothing is known about the inherent power of pointer manipulation.

One reason for this state of affairs is the lack of a thoroughly understood machine model which is both realistic and theoretically accessible. One candidate, the random-access

* This research was supported in part by National Science Foundation grant MCS75-22870 and by the Office of Naval Research contract N00014-76-C-0688. The United States Government's right to retain a nonexclusive royalty-free license in and to copyright covering this paper is acknowledged.

† Some of this work was done while the author was visiting the Faculty of Mathematics at the University of Bielefeld, Bielefeld, West Germany.

0022-0000/79/020110-18$02.00/0

machine [1], which has been used by several authors to provide realistic measures of the complexity of various algorithms, seems too powerful to analyze easily. It also has certain defects, such as allowing unbounded parallelism if a "uniform cost" measure [1] is used.

However, another possible model exists. In 1953 Kolmogorov [11, 12] proposed a machine which operates by manipulating pointers connecting nodes. Fifteen years later Knuth [8] proposed a similar machine, which he called a *linking automaton.* Later and independently Schönhage [16] defined such a machine, which he called a *storage modification machine*, and he showed that such machines can simulate Turing machines with multidimensional tapes in real time. Although these machines provide a useful tool for describing pointer manipulation algorithms, no bounds on their computational power except Schönhage's seem to exist.

This paper describes an extension of Knuth's machine, called a *pointer machine*. The paper defines a class of algorithms which use such a machine to solve the disjoint set union problem, and proves that any such algorithm requires nonlinear time (in the worst case). The class of algorithms is general enough to encompass all set union algorithms known to the author. This result shows that it is possible (in at least one case) to derive a nonlinear lower bound on the complexity of a list-processing problem using a realistic computer model. The result also provides a partial solution to [8, Exercise 2.6.1], which asks for an exploration of the properties of linking automata.

## 2. Pointer Machines

A *pointer machine* consists of a *memory* and a finite number of *registers*. The registers are of two types: *data registers* and *pointer registers*. The memory consists of a finite but expandable pool of *records*. Each record consists of a finite number of *fields*, each of which is either a *data field* or a *pointer field*. Each field has an identifying *name*. All records are identical in structure; that is, they contain the same fields.

A pointer machine manipulates *data* and *pointers*. A pointer either specifies a particular record or is null ($\varnothing$). Each pointer register and pointer field can store one pointer. Data can be of any kind whatsoever (integers, logical values, strings, real numbers, vectors, etc.). Each data register and data field can store one datum.

A *program* for a pointer machine consists of a sequence of *instructions*, numbered consecutively from one. Each instruction is of one of the following eight types. The last instruction of every program is a **halt**. Execution and running time of pointer machines are defined in the obvious way; we charge one unit of time per machine instruction executed.

Each $r$ below denotes a pointer register, each $s$ denotes a data register, each $t$ denotes a register of any type, and each $n$ denotes a field name.

| | |
|---|---|
| $r \leftarrow \varnothing$ | Place a null pointer in register $r$. |
| $t_1 \leftarrow t_2$ | ($t_1$ and $t_2$ must be of the same type). Place the contents of register $t_2$ in register $t_1$, erasing what was there previously. |

| | |
|---|---|
| $t \leftarrow n(r)$ | ($n$ and $t$ must be of the same type). Place the contents of the $n$ field of the record specified by the contents of $r$ into register $t$, erasing what was there previously. (If $r$ contains $\varnothing$, this instruction does nothing.) |
| $n(r) \leftarrow t$ | ($n$ and $t$ must be of the same type). Place the contents of $t$ into the $n$ field of the record specified by the contents of $r$, erasing what was there previously. (If $r$ contains $\varnothing$, this instruction does nothing.) |
| $s_1 \leftarrow s_2 \theta s_3$ | Combine the data in registers $s_2$ and $s_3$ by applying the operation $\theta$. Store the result in $s_1$, erasing what was there previously. |
| **create** $r$ | Create a new record (not specified by any existing pointer) and place a pointer to it in $r$. All fields of the new record initially contain a special value called *undefined* ($\Lambda$). |
| **halt** | Cease execution. |

**if** *condition* **then go to** $i$

If the condition is true, then transfer control to instruction $i$. If the condition is false, do nothing.

Each condition in an **if** instruction is of one of the following types.

| | |
|---|---|
| **true** | Always true. |
| $t_1 = t_2$ | ($t_1$ and $t_2$ must be of the same type). True if the contents of $t_1$ and $t_2$ are the same. |
| $p(s_1, s_2)$ | True if the contents of $s_1$ and $s_2$ satisfy the predicate $p$, where $p$ is any predicate on data. |

To completely specify a pointer machine, we must describe the data and the types of operations allowed on the data. Henceforth we use the term *symbol* in a technical sense to refer to data on which no operations are permitted except testing for equality. A *pure pointer machine* is a pointer machine with *no* data. Knuth's linking automaton is a pointer machine with only symbols as data.

In a pointer machine, access to memory is by explicit reference only; no computation on pointers is possible. The pointer-machine model is thus apparently less powerful than the random-access model with uniform cost measure [1]; pointer machines lack the ability to use address arithmetic for such purposes as manipulating a hash table [9], performing a radix sort [9], or accessing a dense matrix [8]. These machines are, however, powerful enough to simulate such list-processing languages as LISP and to model the list-processing features of Algol-W, PL/1, and other general purpose languages.

## 3. The Disjoint Set Union Problem

Let $S_1, S_2, \ldots, S_n$ be $n$ disjoint sets, each containing a single element. The *disjoint set union problem* is to carry out a sequence of operations of the following two types on the sets.

*find*$(x)$: determine the name of the set containing element $x$.

*union*$(A, B)$: add all elements of set $B$ to set $A$ (destroying set $B$).

The operations are to be carried out *on-line*; that is, each instruction must be completed before the next one is known. We assume that the sequence of operations contains exactly $n - 1$ union operations (so that after the last union all elements are in one set) and $m \geqslant n$ intermixed find operations (if $m < n$, some elements are never found).

The disjoint set-union problem is an abstraction of the operations necessary to implement FORTRAN EQUIVALENCE and COMMON statements [5]. Algorithms for this problem and for a generalization of it have applications in graph theory [18], global code optimization [18, 19], and linear algebra [19]. A number of algorithms exist [1, 4, 5, 6].

A *pointer-machine solution* to the set-union problem consists of a pointer machine, a representation of the input sets as collections of records, a program for carrying out a find, and a program for carrying out a union. The pointer machine solves the set-union problem in the following way. Initially the machine memory represents the input sets. Each find is carried out by executing the find program, which halts having identified the set containing the desired element. Each union is carried out by executing the union program, which halts having modified the contents of memory to reflect the union. We make the following assumptions concerning the details of this process.

(3.1) Each set and each element has a distinct associated symbol.

(3.2) No record in the collection for an input set contains the symbol of any other set or of any element outside the set.

(3.3) No record in the collection for an input set contains a pointer to any record outside the collection.

(3.4) Before the find program is executed to locate the set containing an element $x$, a pointer to some record containing the symbol for $x$ is placed in the designated input register $r_1$ and $\Lambda$ is placed in all other registers. The find program halts with the symbol for the set containing $x$ in the designated output register $s_0$.

(3.5) Before the union program is excuted to add elements in set $B$ to set $A$, pointers to records containing the symbols for $A$ and $B$ are placed in the designated input registers $r_1$ and $r_2$ respectively, and $\Lambda$ is placed in all other registers. The union program halts with no output.

The *sequence of steps* associated with a set-union problem and a pointer-machine solution is the sequence of steps executed by the machine when it carries out the finds and unions. The length of this sequence measures the total running time of the machine. The main result of this paper is a nonlinear lower bound (as a function of $n$ and $m$) on the length of any sequence of steps which solves a worst-case instance of the set-union problem.

The formulation described above is intended to be realistic and to facilitate derivation of a lower bound. Assumption (3.1) above, requiring that sets and elements be represented by symbols, makes it impossible to encode all elements of a set into a single datum and to move this datum at a cost of one step per move; without this restriction there is a pointer machine which can solve any set-union problem in linear time. Assumptions (3.2), (3.3), and (3.4) imply that the machine, when performing a find on some element $x$, has access only to records representing the set containing $x$. Assumptions (3.2), (3.3), and (3.5) imply that the machine, when performing a union on sets $A$ and $B$, has access only to records representing the sets $A$ and $B$. It follows by induction on the number of finds and unions that (3.2) and (3.3) hold for the sets existing at any time during the computation, not just for the input sets. In other words, the contents of memory after any particular find or union can be partitioned into collections of records such that each collection corresponds to a currently existing set, all symbols for the set and its elements occur only in the corresponding collection of records, and no record in one collection contains a pointer to a record in another collection. Without assumptions (3.2)–(3.5) any particular instance of the set-union problem can be solved in linear time by initially moving symbols for all sets and elements into a single record and solving all finds by accessing only this record, though the author conjectures that even without assumptions (3.2)–(3.5) no *single* pointer machine can solve *all* instances of the set-union problem in linear time.

If an algorithm for the set-union problem is to be useful in practice, the symbol of each set and of each element should be stored in exactly *one* record, so that the initialization for finds (3.4) and unions (3.5) is uniquely defined. All the algorithms in the literature have this property, but the lower-bound proof does not require it.

A number of set-union algorithms have been proposed and analyzed (see [1, 3, 4, 5, 6, 8, 10, 17, 20]). It is easy to implement each of these algorithms on a pointer machine. We consider only the fastest (in the worst-case, asymptotic sense) such algorithm, *path compression with weighted union.* The algorithm represents each element by a single record with four fields: *element*, *set*, *parent*, and *pointer*. Symbol field *element* contains the symbol of the element corresponding to the record. During the computation, a currently existing set is represented by a rooted tree,[1] each vertex of which is a record corresponding to an element in the set. The pointer field *parent* of each record in such a tree points to the parent of the record in the tree; the *parent* field of the root is $\varnothing$. The root contains the symbol of the set in symbol field *set* and the size (number of elements) of the set in integer field *size*. Figure 3.1 illustrates this data structure.

A union of sets $A$ and $B$ is performed by comparing the sizes of $A$ and $B$. If $A$ is larger, the parent of the root of $B$ is set equal to the root of $A$ and the *size* field of the root of $A$ is updated. If $B$ is larger, the parent of the root of $A$ is set equal to the root of $B$, and the

[1] A *rooted tree* $T$ is a connected, acyclic, undirected graph with a unique distinguished vertex $r$, called the *root* of $T$. If $v$ and $w$ are vertices of $T$ such that $v$ is on the (unique) simple path from $r$ to $w$, then $v$ is an *ancestor* of $w$ and $w$ is a *descendant* of $v$. This relationship is denoted by $v \overset{*}{\rightarrow} w$. The relationship $v \overset{*}{\rightarrow} w$ and $v \neq w$ is denoted by $v \overset{+}{\rightarrow} w$. If $v \overset{*}{\rightarrow} w$ and $(v, w)$ is an edge of $T$, then $v$ is the *parent* of $w$ and $w$ is a *child* of $v$. This relationship is denoted by $v \rightarrow w$. Two vertices $v$ and $w$ are *unrelated* if $v$ is neither an ancestor nor a descendant of $w$. A *leaf* is a vertex with no children. The *depth* $d(v)$ of a vertex $v$ is the length (number of edges) of the simple path from the root to $v$. The *subtree* of $T$ *rooted at* vertex $v$ is the subgraph of $T$ induced by the descendants of $v$, with $v$ as root.

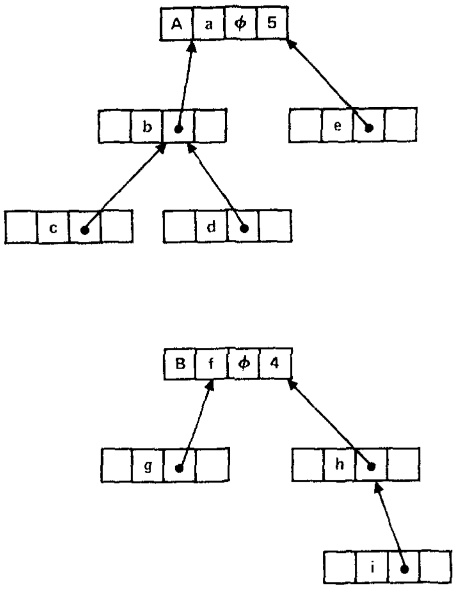

FIG. 3.1. Data structure for set union algorithm. Sets are $A = \{a, b, c, d, e\}$, $B = \{f, g, h, i\}$.

*size* and *set* fields of the root of $B$ are updated. Table I contains an Algol-like program for union. It is easy to translate this into a pointer-machine program.

*A* find on element $x$ is performed by following parent pointers from the record representing $x$ until reaching a record with a null parent. This record is a tree root and contains the symbol for the set containing $x$. In a second pass, the parent of each vertex on the path from $x$ to the root is set equal to the root. This heuristic, called ***path compression***, saves time in later finds. Table II contains a program for the find operation.

TABLE I

Program for Weighted Union

```
procedure union;
    if $size(r_1) < size(r_2)$ then
        begin
        $set(r_2) \leftrightarrow set(r_1)$;
        $parent(r_1) \leftarrow r_2$ ;
        $size(r_2) \leftarrow size(r_1) + size(r_2)$
        end
    else begin
        $parent(r_2) \leftarrow r_1$ ;
        $size(r_1) \leftarrow size(r_1) + size(r_2)$
        end;
```

TABLE II

Program for Find with Path Compression

```
procedure find;
    begin
    root ← current ← $r_1$ ;
    while parent(root) $\neq \varnothing$ do root ← parent(root);
    while parent(current) $\neq \varnothing$ do
        begin
        save ← parent(current);
        parent(current) ← root;
        current ← save
    end end;
```

This set union algorithm is very difficult to analyze; see [4, 6, 17]. Its worst-case running time is $O(m\alpha(m, n))$ [17], where $\alpha(m, n)$ is a functional inverse of Ackermann's function defined as follows.

For $i, j \geqslant 0$ let the function $A(i, j)$ be defined by

$$\begin{aligned} A(i, 0) &= 0; && \\ A(0, j) &= 2; && \text{for } j \geqslant 1; \\ A(i, 1) &= A(i-1, 2) && \text{for } i \geqslant 1; \\ A(i, j) &= A(i-1, A(i, j-1)) && \text{for } i \geqslant 1, \ j \geqslant 2. \end{aligned} \tag{3.1}$$

Let

$$a(i, n) = \min\{j \mid A(i, j) > \log_2 n\} \tag{3.2}$$

and

$$\alpha(m, n) = \min\{i \geqslant 1 \mid A(i, \lfloor 2m/n \rfloor) > \log_2 n\}.^2 \tag{3.3}$$

The functions $A(i, j)$ and $\alpha(m, n)$ as defined here differ slightly from those appearing in [17], but it is routine to show that the difference in $\alpha(m, n)$ is bounded by an additive constent.

This algorithm requires that records contain integer data fields and that pointer machines add and compare. It is natural to ask whether weighted union can be implemented on a pure pointer machine in such a way that the total time for all unions is $O(n)$. The answer is yes.

Each nonnegative integer is represented by a list which encodes the binary digits of the integer. A zero is encoded by a null pointer; a one is encoded by a nonnull pointer. The digit list is singly linked from the low-order digit to the high-order digit. Figure 3.2 illustrates this representation.

[2] For any real number $x$, $\lfloor x \rfloor$ denotes the greatest integer not larger than $x$.

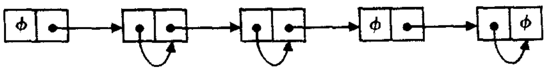

FIG. 3.2. Representation of $26 = 10110_2$ as a list.

Two integers are added by scanning the digit lists and adding digit-by-digit, propagating carries in the usual fashion. The scan stops after the end of the *shorter* list is reached *and* the last carry stops propagating. Two integers are compared by scanning both simultaneously and noting the highest-order digit on which they differ. The scan need only extend to the end of the shorter digit list; the integer with the longer digit list must be larger. We leave as an exercise the implementation of these algorithms as pointer-machine programs.

The $n - 1$ union operations carried out by the algorithm perform the following arithmetic. Initially there are $n$ integers, each equal to one. During a union, two of the integers are compared and then added. After $n - 1$ unions, a single integer equal to $n$ remains. Since comparing two integers requires no more time than adding them, it suffices to bound the time required by all the additions.

LEMMA 3.1. *Let $a$, $b$, $c$ be integers such that $a + b = c$ and let $(a_i)$, $(b_i)$, $(c_i)$, respectively, be their binary digit lists ($a = \sum_{i=0}^{\infty} a_i 2^i$, $b = \sum_{i=0}^{\infty} b_i 2^i$, $c = \sum_{i=0}^{\infty} c_i 2^i$; $a_i, b_i, c_i \in \{0, 1\}$). Let $d_i$ be the carry from the* i*th position when $a$ and $b$ are added. Then $\sum_{i=0}^{k}(a_i + b_i) = d_k + \sum_{i=0}^{k}(c_i + d_i)$ for all $k$. In particular, $\sum_{i=0}^{\infty}(a_i + b_i) = \sum_{i=0}^{\infty}(c_i + d_i)$.*

*Proof.* For $i \geq 0$, $a_i + b_i + d_{i-1} = c_i + 2d_i$ (assuming $d_{-1} = 0$). Thus $a_i + b_i = c_i + d_i + (d_i - d_{i-1})$. Summing from $i = 0$ to $i = k$ gives the lemma. ∎

The time needed to add two binary integers by pointer machine is proportional to the length of the shorter integer plus the number of carries. By Lemma 3.1 the total number of ones in the binary representations of both integers is equal to the number of ones in the binary representation of the sum plus the number of carries. Consider the arithmetic performed during the union operations. Initially, the total number of ones in the binary representations of all the set sizes is $n$. Each carry performed during an addition causes the total number of ones to decrease by one. Thus the total number of carries cannot exceed $n - 1$, and the time required for all carries is $O(n)$.

It remains to bound the sum of the lengths of the shorter of each pair of integers added during union operations. Let $f(n)$ be a worst-case bound on this total length as a function of $n$. Then $f(1) = 0$, and

$$f(n) = \max\{\lfloor \log_2 k \rfloor + 1 + f(k) + f(n-k) \mid 1 \leq k \leq n/2\} \qquad \text{for} \quad n > 1,$$

since the length of the binary representation of $k$ is $\lfloor \log_2 k \rfloor + 1$.

LEMMA 3.2. $f(n) \leq 2n - \log_2 n - 2$.

*Proof.* By induction on $n$.

$$f(1) = 0 \leqslant 2 - \log_2 1 - 2.$$

Let $n \geq 2$ and suppose the lemma is true for all values less than $n$. Let $k$ be such that $1 \leq k \leq n/2$ and

$$f(n) = \lfloor \log_2 k \rfloor + 1 + f(k) + f(n-k).$$

By the induction hypothesis

$$\begin{aligned} f(n) &\leq \log_2 k + 1 + 2k - \log_2 k - 2 + 2(n-k) - \log_2(n-k) - 2 \\ &\leq 2n - (\log_2(n-k) + 1) - 2 \\ &\leq 2n - \log_2 n - 2 \qquad \text{since } k \leq n/2. \quad \square \end{aligned}$$

It follows that the total time to perform all arithmetic associated with the union operations is $O(n)$, and the following theorem holds.

THEOREM 3.1. *There exists a pure pointer machine which solves any disjoint set-union problem in $O(m\alpha(m, n))$ time.*

## 4. A NONLINEAR LOWER BOUND

This section shows that for all $m$ and $n$ there is a set-union problem which requires at least $cm\alpha(m, n)$ steps to solve by pointer machine, where $c$ is a positive constant independent of $m$ and $n$. Rather than consider pointer machines, we consider sequences of pointer-machine steps. Given a set-union problem, a sequence of pointer-machine steps is said to solve it if there is some pointer machine, some set of union programs, one for each union, and some set of find programs, one for each find, such that when the sequence of programs corresponding to the sequence of union and find operations is executed according to the conventions of Section 3, the given sequence of pointer-machine steps results and the find programs produce correct answers. Note that any sequence of pointer-machine steps can be carried out by a nonbranching pointer-machine program. We thus assume without loss of generality that no **if** instructions occur in any of the union or find programs.

The lower-bound proof consists of two parts. First, we convert any solution to a set-union problem into a simplified normal form, while increasing the running time by at most a constant factor. This conversion proceeds in two steps, described in Theorems 4.1 and 4.2. Next, we apply a variant of the lower-bound proof in [17] to show that any normal form solution contains a nonlinear number of steps.

THEOREM 4.1. *Let $S_1$ be any sequence of pointer-machine steps which solves a set-union problem. Then there is a sequence of pointer-machine steps $S_2$ which also solves the set-union problem and has the following properties:*

(4.1) $|S_2| \leqslant 2(m + n + |S_1|)$.

(4.2) *$S_2$ manipulates no data except set and element symbols.*

(4.3) *$S_2$ represents each input set by a single record and contains no* **create** *instruction.*

(4.4) *$S_2$ fetches a symbol from memory only as the last instruction of a find and not at all during a union.*

*Proof.* Let $S_1$ be a sequence of pointer-machine steps which solves some set-union problem. Delete from $S_1$ all steps which manipulate data other than set and element symbols. The sequence $S_1$ now has property (4.2) and still solves the set-union problem.

The sequence $S_2$ to be constructed manipulates records corresponding to the sets, the elements, and the records manipulated by $S_1$. Initially the memory of $S_2$ consists of one record for each input set $A = \{a\}$. This record is the *representative* of the set $A$, of the element $a$, and of each record in the initial collection of records by which $S_1$ represents $A$. Each record created by $S_1$ also has a representative in the memory of $S_2$, defined as follows. The representative of a record created during execution of *find*$(a)$ is the representative of $a$. The representative of a record created during execution of *union*$(A, B)$ is the representative of $A$. For any object $x$ (set, element, or record), let $x^*$ denote the representative of $x$.

$S_2$ simulates $S_1$ step-by-step. If $S_1$ and $S_2$ are executed in parallel, the memory and registers of $S_2$ correspond to the memory and registers of $S_1$ in the following way.

(4.5) If $R_1$ and $R_2$ are records in the memory of $S_1$ such that $R_1$ contains a pointer to $R_2$, then $R_1^*$ contains a pointer to $R_2^*$ (unless $R_1^* = R_2^*$).

(4.6) If $R$ is a record containing a set or element symbol $x$, then $R^*$ contains a pointer to $x^*$ and $x^*$ contains a pointer to $R^*$ (unless $R^* = x^*$).

(4.7) If some register of $S_1$ contains a pointer to a record $R$, then some register of $S_2$ contains a pointer to $R^*$.

(4.8) If some register of $S_1$ contains a set or element symbol $x$, then some register of $S_2$ contains a pointer to $x^*$.

(4.9) During execution of *find*$(a)$, $S_2$ maintains a pointer to $a^*$ in a register. During execution of *union*$(A, B)$, $S_2$ maintains a pointer to $A^*$ in a register.

Initially the memory of $S_2$ consists of all the representatives, each containing the symbol of the corresponding set, the symbol of the corresponding element, and no pointers. Properties (4.5)–(4.9) hold initially.

Let *find*$(a)$ be a typical find. $S_1$ begins *find*$(a)$ with a pointer in $r_1$ to a record $R$ containing the symbol for $a$. If (4.6) holds before the find, either $R^* = a^*$ or $a^*$ contains a pointer to $R^*$. $S_2$ begins the find with a pointer to $a^*$ in $r_1$. $S_2$'s first step is to fetch a pointer to $R^*$ into a register. This preserves (4.5)–(4.9).

Let *union*$(A, B)$ be a typical union. $S_1$ begins *union*$(A, B)$ with pointers in $r_1$, $r_2$ to records $R_1$, $R_2$ containing the symbols for $A$, $B$, respectively. If (4.6) holds before the find, either $R^* = A^*$ or $A^*$ contains a pointer to $R_1^*$; similarly either $R_2^* = B^*$ or $B^*$ contains a pointer to $R_2^*$. $S_2$ begins the union with pointers to $A^*$, $B^*$ in $r_1$, $r_2$, respectively. $S_2$'s first two steps are to fetch pointers to $R_2^*$ and $R_1^*$ into registers. This preserves (4.5)–(4.9).

$S_2$ simulates each step of $S_1$ in the following way. Each time $S_1$ fetches a pointer to a record $R_2$ from a record $R_1$, $S_2$ fetches a pointer to $R_2^*$ from $R_1^*$ (possible by (4.5)). Each time $S_2$ stores a pointer to a record $R_2$ in a record $R_1$, $S_2$ stores a pointer to $R_2^*$ in $R_1^*$ (possible by (4.7)). Each time $S_1$ fetches a set or element symbol $x$ from a record $R$, $S_2$ fetches a pointer to $x^*$ from $R^*$ (possible by (4.6)). Each time $S_1$ stores a set or element symbol $x$ into a record $R$, $S_2$ stores a pointer to $x^*$ in $R^*$ and a pointer to $R^*$ in $x^*$ (possible by (4.7) and (4.8)). Each time $S_1$ creates a record, $S_2$ does nothing. At the end of each find, $S_2$ fetches the appropriate set symbol. Each of these steps preserves (4.5)–(4.9). The sequence $S_2$ constructed in this way carries out the finds and has properties (4.1)–(4.4). ∎

We can represent the memory manipulated by a pointer machine as an undirected graph, with one vertex $R^*$ for each record $R$ and one edge for each pointer. If a record $R_1$ contains a pointer to a record $R_2$, then $(R_1^*, R_2^*)$ is an edge in the graph. This representation motivates the following definition, which reformulates the set-union problem as a graph-construction problem.

A *link solution* to a set-union problem consists of a set of vertices $V$, one for each initial set and element, and a sequence of instructions of the form *link*$(v, w)$, where $v, w \in V$. The sequence of link instructions constructs a graph edge-by-edge, starting from the graph with vertex set $V$ and no edges; *link*$(x, y)$ constructs edge $(x, y)$. For any initial set or element $x$, let $x^*$ denote the corresponding vertex. The sequence of link instructions must have the following properties.

(4.10) The sequence of links can be partitioned into contiguous subsequences, each subsequence corresponding to a union or find operation.

(4.11) Let *find*$(a)$ with answer $A$ be a typical find. Each *link*$(x, y)$ in the subsequence for *find*$(a)$ is such that $x = A^*$ and the distance between $x$ and $y$ in the graph existing before the link is two. If $A^* \neq a^*$, then the instruction *link*$(A^*, a^*)$ occurs either in the subsequence for *find*$(a)$ or earlier in the sequence.

(4.12) Let *union*$(A, B)$ be a typical union. Each *link*$(x, y)$ in the subsequence for *union*$(A, B)$ is such that $x = A^*$ and either $y = B^*$ or the distance between $x$ and $y$ in the graph existing before the link is two.

THEOREM 4.2. *Any set-union problem solvable in $k$ pointer-machine steps has a link solution of length not exceeding* $4m + 5n + 4k$.

*Proof.* Let $S_1$ be a sequence of $k$ pointer-machine steps which solves a set-union problem. Let $S_2$ be a sequence of pointer-machine steps satisfying Theorem 4.1. Then $|S_2| \leqslant 2(m + n + k)$. From $S_2$ we construct a link solution $S_3$ satisfying the theorem. The vertex set for $S_3$ consists of one vertex $R^*$ for each record $R$ manipulated by $S_2$. If $S_2$ and $S_3$ are executed in parallel, the following properties hold.

(4.13) If a record $R_1$ contains a pointer to a record $R_2$, then the distance between $R_1^*$ and $R_2^*$ is at most two.

(4.14) Let *find*($a$) with answer $A$ be a typical find. If during this find some register of $S_2$ contains a pointer to $R$, then either $A^* = R^*$ or $(A^*, R^*)$ is a previously constructed edge.

(4.15) Let *union*($A, B$) be a typical union. If during this union some register of $S_2$ contains a pointer to $R$, then either $A^* = R^*$ or $(A^*, R^*)$ is a previously constructed edge.

$S_3$ simulates $S_2$ instruction-by-instruction. Certainly (4.13)–(4.15) hold initially. Let *union*($A, B$) be a typical union. To begin the union, $S_3$ links $A^*$ and $B^*$. This preserves (4.13)–(4.15). Let *find*($a$) with answer $A$ be a typical find. Suppose $S_2$ fetches $l$ pointers from memory while carrying out the find. If (4.13) holds before the find, there must be a path of length $2l$ or less between $A^*$ and $a^*$ in the graph existing before the find. To begin the find $S_3$ links each vertex on this path to $A^*$. This preserves (4.13)–(4.15).

Consider a subsequence of $S_2$ corresponding either to a *find*($a$) with answer $A$ or a *union*($A, B$). Suppose $S_2$ fetches a pointer (say to $R_2$) from a record (say $R_1$). If (4.13)–(4.15) hold before the fetch, then there is a path between $A^*$ and $R_2^*$ of length at most three. $S_3$ links each vertex on this path to $A^*$. This preserves (4.13)–(4.15). Suppose $S_2$ stores a pointer (say $R_2$) in a record (say $R_1$). Then $S_2$ must first have pointers to $R_1$ and $R_2$ in registers. By (4.14) and (4.15) this means that the distance between $R_1^*$ and $R_2^*$ in the graph existing before the store is at most two, and no links need to be carried out to preserve (4.13)–(4.15). All other instructions in $S_2$ do not affect (4.13)–(4.15).

The total length of the sequence $S_3$ constructed in this way is at most $4m + 5n + 4k$, and the sequence clearly solves the set-union problem. ▌

In the following discussion we do not distinguish between an initial set, its single element, and the vertex representing the set and the element. We define the ***union tree*** of a sequence of unions as follows. The vertices of the tree are the initial sets. The edges are the pairs $(A, B)$ such that *union*($A, B$) occurs in the sequence. The root of the tree is the set remaining after all unions are carried out. With this definition, every *link*($v, w$) in a link solution to a set-union problem has the property that $v \xrightarrow{+} w$ in the union tree. In the worst-case set-union problems to be constructed below, the union tree is a complete binary tree.

The lower-bound proof makes use of a rapidly growing function $B(i, j)$ defined for $i, j \geqslant 1$ as follows.

$$\begin{aligned} B(1,j) &= 1 && \text{for } j \geqslant 1;\\ B(i,1) &= B(i-1,2)+1 && \text{for } i \geqslant 2;\\ B(i,j) &= B(i,j-1)+B(i-1,2^{B(i,j-1)}) && \text{for } i,j \geqslant 2. \end{aligned}$$

LEMMA 4.1. *$B(i,j) + 1 \leqslant A(i, 2j)$ for $i, j \geqslant 1$.*

*Proof.* Straightforward by double induction (see [17]). ▌

THEOREM 4.3. *For any $k, s \geqslant 1$, let $T$ be a complete binary tree of depth $d > B(k, s)$. Let $\{v_i \mid 1 \leqslant i \leqslant s2^{B(k,s)}\}$ be a set of pairwise unrelated vertices in $T$, each of depth strictly* *greater than $B(k, s)$, such that exactly $s$ vertices in $\{v_i\}$ occur in each subtree of $T$ rooted at a vertex of depth $B(k, s)$. Then for $n = 2^{k+1} - 1$ and $m = s2^{B(k,s)}$ there is a set-union problem for which*

(4.16) *the union tree is $T$;*

(4.17) *the set of finds is* $\{\text{find}(v_i) \mid 1 \leqslant i \leqslant m\}$;

(4.18) *the answer to each find is a vertex of depth strictly less than $B(k, s)$; and*

(4.19) *any link solution has length at least $km$, even if every edge $(v, w)$ such that $v \xrightarrow{+} w$ and $d(v) \geqslant B(k, s)$ in $T$ is allowed for free, and after each* link$(v, w)$ *every edge $(x, y)$ such that $v \xrightarrow{*} x \xrightarrow{+} y \xrightarrow{*} w$ is added for free.*

*Proof.* The proof is by double induction on $k$ and $s$ and is similar to the lower-bound proof in [17]. Suppose $k = 1$. Consider any set-union problem consisting of $n - 1$ unions which form $T$ followed by a find on each vertex in $\{v_i\}$. The answer to each find is the root of $T$; (4.18) holds since $B(k, s) > 0$. None of the originally free edges solves a find. Since the vertices in $\{v_i\}$ are pairwise unrelated, any *link*$(x, y)$ can solve only one find, even including the appropriate free edges. Thus (4.19) holds (Fig. 4.1).

Suppose the theorem holds for $k - 1$, $s = 2$. The following argument proves the theorem for $k$ with $s = 1$. Suppose the hypotheses of the theorem hold. Let $\{u_i \mid 1 \leqslant i \leqslant m\}$ be the set of vertices of depth $B(k, 1)$ in $T$, numbered so that $u_i \xrightarrow{+} v_i$. The vertices in $\{u_i\}$ are pairwise unrelated and exactly two occur in each subtree of $T$ rooted at a vertex of depth $B(k, 1) - 1 = B(k - 1, 2)$. By the induction hypothesis there is a set-union problem satisfying the theorem for $k' = k - 1$, $s' = 2$, $T$, $\{u_i\}$. Let the

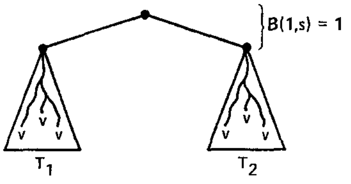

FIG. 4.1. Tree for the case $k = 1$. $T_1$ and $T_2$ are complete binary trees. Each $v$ denotes a vertex $v_i$; all vertices $v_i$ are at a distance of at least two from the root of $T$.

sequence of finds and unions in this set-union problem be $P_1$. Form $P_2$ from $P_1$ by replacing each *find*$(u_i)$ by *find*$(v_i)$. We claim the resulting sequence satisfies the theorem for $k$, $s = 1$, $T$, $\{v_i\}$ (Fig. 4.2).

Certainly (4.16)–(4.18) hold. Consider any sequence $S_2$ of of links which carries out $P_2$, allowing for free the edges described in (4.19). Form a sequence $S_1$ from $S_2$ by replacing each *link*$(x, y)$ such that $v_i \xrightarrow{*} y$ for some (uniquely determined) $i$ by *link*$(x, u_i)$. Delete from $S_1$ all links which do not create new edges. We claim $S_1$ carries out $P_1$ (allowing appropriate edges for free) and that $|S_1| \leqslant |S_2| - m$.

The following property is true initially and is preserved if $S_1$ and $S_2$ are executed in parallel (on separate graphs).

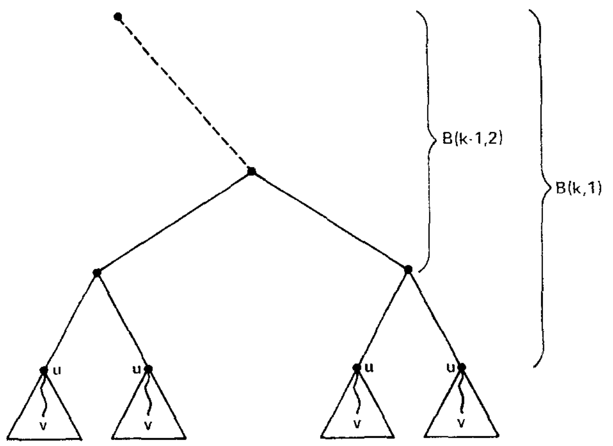

FIG. 4.2. Branch of tree for the case $s = 1$. Each $v$ denotes a vertex $v_i$; each $u$ denotes a vertex $u_i$.

(4.20) For $1 \leqslant i \leqslant m$, $u_i$ is adjacent in the graph manipulated by $S_1$ to all vertices adjacent to at least one descendant of $v_i$ in the graph manipulated by $S_2$.

It follows that $S_1$ carries out $P_1$.

For any $v_i$, consider the first *link*$(x, y)$ in $S_2$ such that $x \xrightarrow{+} u_i \xrightarrow{+} v_i \xrightarrow{*} y$. There must be such a link since none of the initially free edges solves *find*$(v_i)$ by (4.18). There must be a path of length two, say $(x, z)(z, y)$, between $x$ and $y$ in the $S_2$ graph existing before the link. Furthermore $z$ must satisfy $u_i \xrightarrow{*} z \xrightarrow{*} v_i$. It follows that $(x, u_i)$ is an edge of the $S_1$ graph existing before the link. Thus $S_1$ need not contain an instruction *link*$(x, u_i)$ corresponding to *link*$(x, y)$. This is true for any value of $i$. Hence $| S_1 | \leqslant | S_2 | - m$.

Since $(k - 1)m \leqslant | S_1 |$ by the induction hypothesis, $| S_2 | \geqslant km$, and (4.19) holds.

Suppose the theorem holds for $k$, $s - 1$ and also for $k - 1$, $B(k, s - 1)$. The following argument proves the theorem for $k$, $s$. Suppose the hypotheses of the theorem hold. Let $\{w_i \mid 1 \leqslant i \leqslant 2^{B(k,s)}\}$ be a subset of $\{v_i\}$ such that exactly one vertex $w_i$ occurs in each subtree of $T$ rooted at a vertex of depth $B(k, s)$. Let $\{u_i \mid 1 \leqslant i \leqslant 2^{B(k,s)}\}$ be the set of vertices of depth $B(k, s)$, numbered so that $u_i \xrightarrow{+} w_i$ (Fig. 4.3).

Consider the subtrees $T_j$, $1 \leqslant j \leqslant 2^{B(k,s)-B(k,s-1)}$, rooted at vertices of depth $B(k, s) - B(k, s - 1) = B(k - 1, 2^{B(k,s-1)})$ in $T$. Each subtree $T_j$ contains $(s - 1)2^{B(k,s-1)}$ vertices in $\{v_i\} - \{w_i\}$, exactly $s - 1$ in each subtree rooted at a vertex of depth $B(k, s)$. By the induction hypothesis there is a set-union problem satisfying the theorem for $k' = k$, $s' = s - 1$, $T_j$, $\{v \mid v \text{ is a vertex in } T_j \text{ and } v \in \{v_i\} - \{w_i\}\}$. Let $P_j$ be the sequence of unions and finds in this set-union problem.

The vertices in the set $\{u_i\}$ are pairwise unrelated and exactly $2^{B(k,s-1)}$ occur in each subtree $T_j$ of $T$. By the induction hypothesis there is a set-union problem satisfying the theorem for $k' = k - 1$, $s' = 2^{B(k,s-1)}$, $T$, $\{u_i\}$. Let $Q$ be the sequence of unions and finds in this set-union problem. Because appropriate edges are allowed for free, the sequence $Q$ can be permuted, without increasing the number of links required to carry out $Q$, so that all unions forming the subtrees $T_j$ occur before all other operations.

Let $Q'$ be formed from the permuted version of $Q$ by deleting all unions forming the subtrees $T_j$, let $Q''$ be formed from $Q'$ by replacing each *find*$(u_i)$ by *find*$(w_i)$, and let $P'' = P_1, P_2, \ldots, P_{2^{B(k,s)-B(k,s-1)}}, Q''$. We claim $P''$ defines a set-union problem which satisfies the theorem for $k$, $s$, $T$, $\{v_i\}$.

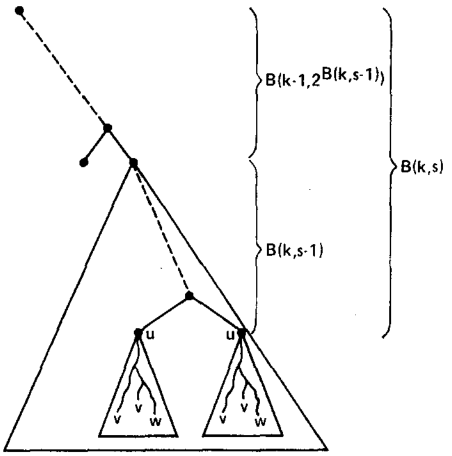

FIG. 4.3. Branch of tree for the general case. Each $v$ denotes a vertex $v_i$, each $w$ denotes a vertex $w_i$, and each $u$ denotes a vertex $u_i$. Finds on all vertices in $\{v_i\} - \{w_i\}$ occur within trees $T_j$, leaving finds on vertices $w_i$ to be performed in larger tree $T$.

Certainly (4.16)–(4.18) hold. Consider any sequence $S''$ of links which carries out $P''$, allowing for free the edges described in (4.19). Form a new sequence $S$ from $S''$ by replacing each *link*$(x, y)$ such that $w_i \xrightarrow{*} y$ for some (uniquely determined) $i$ by *link*$(x, u_i)$. Delete from $S$ all links which do not create new edges. The following property is true initially and is preserved if $S$ and $S''$ are executed in parallel (on separate graphs).

(4.21) For $1 \leqslant i \leqslant 2^{B(k,s)}$, $u_i$ is adjacent in the graph manipulated by $S$ to all vertices adjacent to at least one descendant of $w_i$ in the graph manipulated by $S''$.

It follows by an argument like that in the previous case that $S$ carries out

$$P' = P_1, P_2, \ldots, P_{2^{B(k,s)-B(k,s-1)}}, Q'$$

and that $|S| \leqslant |S''| - 2^{B(k,s)}$. $S$ can be written as $S = S_1, S_2, \ldots, S_{2^{B(k,s)-B(k,s-1)}}, U$, where $S_i$ carries out $P_i$ for $1 \leqslant i \leqslant 2^{B(k,s)-B(k,s-1)}$, allowing for free the edges described in (4.19), and $U$ carries out $Q'$, allowing for free the edges $(v, w)$ such that $v \xrightarrow{+} w$ and $d(v) \geqslant B(k-1, 2^{B(k,s-1)})$ and after each *link*$(v, w)$ allowing for free the edges $(x, y)$ such that $v \xrightarrow{*} x \xrightarrow{+} y \xrightarrow{*} w$. This means that $U$ carries out $Q$, allowing the appropriate edges for free. By (4.19), $|S_i| \geqslant k(s-1)\,2^{B(k,s-1)}$ for $1 \leqslant i \leqslant 2^{B(k,s)-B(k,s-1)}$, and $|U| \geqslant (k-1)\,2^{B(k,s)}$. It follows that

$$|S''| \geqslant |S| + 2^{B(k,s)} \geqslant k(s-1)2^{B(k,s)} + (k-1)2^{B(k,s)} + 2^{B(k,s)} = ks2^{B(k,s)} = km.$$

Thus (4.19) holds. By double induction, the theorem is true in general. ∎

COROLLARY 4.1. *Let $k, s \geqslant 1$. Let $T$ be a complete binary tree of depth $B(k, s)$. Then there is a set-union problem whose union tree is $T$, which contains $m = s2^{B(k,s)}$ finds, and which requires at least $(k - 1)m$ links for its solution.*

*Proof.* Choose $l \geqslant 1$ such that $2^l \geqslant s$. Let $T'$ be a complete binary tree formed by replacing each leaf of $T$ by a complete binary tree of height $l$. Let $\{v_i \mid 1 \leqslant i \leqslant m\}$ be any set of vertices satisfying the hypotheses of Theorem 4.3 for $k$, $s$, $T'$. For $1 \leqslant i \leqslant m$, let $u_i$ be the vertex of height $l$ in $T'$ such that $u_i \xrightarrow{+} v_i$. Let $P'$ be a sequence of unions and finds defining a set-union problem satisfying the conclusions of Theorem 4.3 for $k$, $s$, $T'$, $\{v_i\}$. Without loss of generality we can assume that the unions which form the subtrees of $T'$ rooted at height $l$ occur at the front of $P'$.

Form $P$ from $P'$ by deleting the unions which form the subtrees of $T'$ rooted at height $l$ and replacing each *find*$(v_i)$ by *find*$(u_i)$. We claim $P$ defines a set-union problem satisfying the conclusions of the corollary. Certainly $P$ contains $m$ finds and the union tree of $P$ is $T$. Suppose $S$ is a sequence of links which carries out $P$. Form $S'$ from $S$ by following each *link*$(x, u_i)$ which solves a *find*$(u_i)$ by link$(x, v_i)$. Then $S'$ carries out $P'$ if all edges $(v, w)$ with $d(v) \geqslant l$ are allowed for free. Thus $|S'| \geqslant km$, and $|S''| \geqslant (k - 1)m$. ∎

Theorem 4.2, Lemma 4.1, and Corollary 4.1 combine to establish the main result of this paper.

THEOREM 4.4. *There is a positive constant $c$ such that, for all $m \geqslant n \geqslant 1$, there is a set-union problem consisting of $m$ finds and $n - 1$ intermixed unions whose solution by pointer machine requires at least $cm\alpha(m, n)$ steps.*

*Proof.* Let $s = \lfloor m/n \rfloor$. Choose $k$ as large as possible such that $2^{B(k,s)+1} - 1 \leqslant n$. Partition the $n$ elements into as many sets as possible of size $2^{B(k,s)+1} - 1$, plus leftover elements. At most $n/2$ elements are leftover. On each set of $2^{B(k,s)+1} - 1$ elements, define a set-union problem satisfying Corollary 4.1. Concatenate these problems, add enough additional unions to combine all elements, including the leftovers, into a single set, and add enough additional finds to bring to total to $m$.

The resulting set-union problem contains $m$ finds, $n - 1$ intermixed unions, and requires at least $(k - 1)\, s2^{B(k,s)}n/2^{B(k,s)+2} = (k - 1)\, sn/4 \geqslant (k - 1)m/8$ links for its solution. By Theorem 4.2, this set-union problem requires at least $(k - 1)m/32 - m - 4n/4 \geqslant (k - 73)m/32$ pointer-machine steps for its solution.

If $\alpha(m, n) \geqslant 2$, $k \geqslant \alpha(m, n) - 1$ in this construction since

$$\begin{aligned} B(\alpha(m, n) - 1, s) + 1 &\leqslant A(\alpha(m, n) - 1, 2s) && \text{by Lemma 4.1} \\ &\leqslant \log_2 n && \text{by the definition of } \alpha. \end{aligned}$$

Thus the selected set-union problem requires at least $(\alpha(m, n) - 74)m/32 \geqslant \alpha(m, n)m/64$ pointer-machine steps, if $\alpha(m, n) \geqslant 148$. But if $\alpha(m, n) \leqslant 148$, *any* set-union problem requires at least $m \geqslant m\alpha(m, n)/148$ pointer-machine steps. Choosing $c = 1/148$ gives the theorem. ∎

## 5. Conclusions

This paper has described a machine model, called a *pointer machine*, suitable for analyzing list-processing problems. The model is similar to several previously proposed [8, 11, 12, 16]. Pointer machines are quite powerful; Schönhage [16] has shown that they can simulate Turing machines with multidimensional tapes in real time, and one can show that they can simulate random-access machines with logarithmic cost in real time.

The paper has analyzed the ability of pointer machines to compute disjoint set unions. Under certain natural restrictions, all pointer machines require nonlinear time to solve this problem. This lower bound characterizes the efficiency with which one can represent dynamic information of a certain kind in a list structure. The bound does not require that the machine be deterministic, or that the program of the machine be fixed while the problem size grows, or that the complexity of memory (number of fields per record) be fixed while the problem size grows.

This generality is achieved by making the assumption that the description of each set is stored separately and that moving the description of a set requires constant time per element. Without these assumptions the lower bound is not valid. The author conjectures, however, that the lower bound holds if the separate storage assumption is replaced by an assumption about the complexity of memory; namely, that every record contains only a fixed number of fields independent of the problem size.

The lower-bound proof would be simplified if one could show how to convert any pointer-machine solution for the set-union problem into a form to which the lower bound of [17] would apply directly. The author was unsuccessful in accomplishing this and believes it to be very hard. The lower bound in Theorem 4.4 is significantly more general than that in [17]; it covers arbitrary manipulation of pointers, whereas [17] allows only pointers between records on a find path.

## Acknowledgment

I would like to thank Professor Wolfgang Paul for his thoughtful criticism and valuable insights which contributed substantially to the lower bound proof.

## References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, Mass., 1974.
2. A. Borodin and I. Munro, "The Computational Complexity of Algebraic and Numeric Problems," Elsevier, New York, 1975.
3. J. Doyle and R. L. Rivest, Linear expected time of a simple union-find algorithm, *Inform. Processing Lett.* **5** (1976), 146–148.
4. M. J. Fischer, Efficiency of equivalence algorithms, *in* "Complexity of Computer Computations" (R. E. Miller and J. W. Thatcher, Eds.), pp. 153–168, Plenum, New York, 1972.
5. B. A. Galler and M. J. Fischer, An improved equivalence algorithm, *Comm. ACM* **7** (1964), 301–303.

6. J. E. Hopcroft and J. D. Ullman, Set merging algorithms, *SIAM J. Comput.* **2** (1973), 294–303.
7. M. Jazayeri, W. F. Ogden, and W. C. Rounds, The intrinsically exponential complexity of the circularity problem for attribute grammars, *Comm. ACM* **18** (1975), 697–706.
8. D. E. Knuth, "The Art of Computer Programming," Vol. 1, "Fundamental Algorithms," Addison–Wesley, Reading, Mass., 1968.
9. D. E. Knuth, "The Art of Computer Programming," Vol. 3, "Sorting and Searching," Addison–Wesley, Reading, Mass., 1975.
10. D. E. Knuth and A. Schönhage, The expected linearity of a simple equivalence algorithm, Technical Report STAN-CS-77-599, Computer Science Department, Stanford University, 1977.
11. A. N. Kolmogorov, On the notion of algorithm, *Uspehi Mat. Nauk.* **8** (1953), 175–176.
12. A. N. Kolmogorov and V. A. Uspenskii, On the definition of an algorithm, *Uspehi Mat. Nauk.* **13** (1958), 3–28; English translation *Amer. Math. Soc. Transl.* **29** (1963), 217–245.
13. A. R. Meyer and L. J. Stockmeyer, The equivalence problem for regular expressions with squaring requires exponential space, *in* "Proc. 13th Annual Symp. on Switching and Automata Theory, 1972," pp. 125–129.
14. M. J. Rabin and M. J. Fischer, Super-exponential complexity of Presburger arithmetic, *SIAM–Amer. Math. Soc. Proc.* **7** (1974), 27–41.
15. R. Rivest and J. Vuillemin, On recognizing graph properties from adjacency matrices, *Theoret. Comput. Sci.* **3** (1976), 371–384.
16. A. Schönhage, Real-time simulation of multidimensional Turing machines by storage modification machines, Project MAC Technical Memorandum 37, MIT, 1973.
17. R. E. Tarjan, Efficiency of a good but not linear disjoint set union algorithm, *J. Assoc. Comput. Mach.* **22** (1975), 215–225.
18. R. E. Tarjan, Applications of path compression on balanced trees, Technical Report STAN-CS-75-512, Computer Science Dept., Stanford University, 1975.
19. R. E. Tarjan, Solving path problems on directed graphs, Technical Report STAN-CS-75-528, Computer Science Dept., Stanford University, 1975.
20. A. C. Yao, On the average behavior of set merging algorithms, *in* "Proc. Eighth Annual ACM Symp. on Theory of Computing 1976," pp. 192–195.