Galil, S. Winograd

tension fields, *Theoret. Comput. Sci.* **8** (1979) 359-377.
plicative complexity depends on the field of constants,

itations, *SIAM J. Comput.* (1980).
ons necessary to compute certain functions, *Comm.*

y of the Discrete Fourier Transform, *Adv. in Math.*

rier Transform, *Math. Comput.* **32** (1978) 175-199.
tional elements, *Soviet Math. Dokl.* **4** (1963) 714-716.
ving linear multiplicative complexities, *J. ACM* **24**

minimal division-free algorithms are all bilinear, *J.*

se minimal algorithms are all quadratic, *J Algorithms*

ithms for the computation of bilinear mappings: I.
ret. Comput. Sci. **7** (1978) 1-24.
ithms for the computation of bilinear mappings: II.
tion, *Theoret. Comput. Sci.* **7** (1978) 124-148.
thms for the computation of bilinear mappings: III.
and yx where $x, y \in M_2(K)$, *Theoret. Comput. Sci.* **7**

sification of all the minimal bilinear algorithms for
wo polynomials, Part II: The algebra $G[u]/\langle u^n \rangle$, to

plications in simply generated local algebra, Tech.

Theoretical Computer Science 58 (1988) 57-68
North-Holland

A TRADEOFF BETWEEN SEARCH AND UPDATE TIME FOR THE IMPLICIT DICTIONARY PROBLEM *

Allan BORODIN
Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4

Faith E. FICH
Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4

Friedhelm MEYER AUF DER HEIDE
Fachbereich Informatik, Universität Dortmund, 46 Dortmund-Eichlinghofen, Fed. Rep. Germany

Eli UPFAL
Weizmann Institute, Rehovot 76100, Israel

Avi WIGDERSON
Institute of Mathematics and Computer Science, Hebrew University, Givat-Ram, Jerusalem 91904, Israel

Abstract. This paper proves a tradeoff between the time it takes to search for elements in an implicit dictionary and the time it takes to update the value of elements in specified locations of the dictionary. It essentially shows that if the update time is constant, then the search time is $\Omega(n^{\epsilon})$ for some constant $\epsilon > 0$.

1. Introduction

A dictionary is a data type that supports two operations: search and update. The former determines whether a given element y is one of the elements in the dictionary. This is useful, for example, to determine whether a word you have just written is spelled correctly. If the element is in the dictionary, its location may also have to be found.

Updates modify the contents of the dictionary. We may wish to add an element (e.g., a new piece of jargon), delete an element (e.g., an obsolete word), or replace one element by another (e.g., a correction of a spelling mistake). By using dummy values, we may view additions and deletions as instances of replace. This is convenient for purposes of analysis since the number of elements in the dictionary remains fixed.

* This work began while the authors were at IBM Research Lab, San Jose, CA, and was also supported in part by an IBM Faculty Development Award and National Science Foundation Grant MCS-8402676.

0304-3975/88/$3.50 © 1988, Elsevier Science Publishers B.V. (North-Holland)

Balanced tree schemes, such as 2-3 trees, can be used to implement dictionaries efficiently. Both search and update can be done in $O(\log n)$ steps, where n is the number of elements in the dictionary. However, explicit pointers are used and extra space is needed to store them (i.e., in addition to the space used to store the elements). For large dictionaries, this may be significant.

A natural question to ask is whether this extra space is needed to efficiently implement a dictionary. For example, although heaps are naturally implemented as binary trees using explicit pointers, they can also be represented without extra space. The pointers are implicit—the elements in locations $2i$ and $2i+1$ are the left and right children respectively of the element in location i.

Implicit data structures [11] are those in which only the number of elements and the elements themselves are stored explicitly. The n elements are stored in an array of length n, each cell of which is capable of holding exactly one element.

There are a number of ways to implement dictionaries implicitly. If we store the elements in an unordered list, then update can be done in constant time, but search requires linear time. On the other hand, if the dictionary is maintained as an array sorted in increasing order, then search can be done in logarithmic time, but updates may require that all the elements in the array be moved. Rotated lists are arrays that can be sorted into increasing order by performing a cyclic shift (rotation) of the elements. They are not much more difficult to search, but only half the elements have to be moved in the worst case. Notice that, by the information theory lower bound, any comparison-based search algorithm must perform at least a logarithmic number of comparisons in the worst case.

Munro and Suwanda [11] were the first to consider explicitly the implicit dictionary problem. They showed that if the elements are stored partially sorted into a triangular grid, then search and update can be performed in $O(\sqrt{n})$ steps. Using blocks of rotated lists (sorted relative to one another), they were able to improve the search time to $O(\log n)$, keep the number of moves per update at $O(\sqrt{n})$, and only increase the number of comparisons per update to $O(\sqrt{n}\log n)$. Combining these ideas, they also produced an implicit dictionary that can be searched or updated in $O(n^{1/3}\log n)$ time. By using rotated lists in a recursive manner, Frederickson [6] was able to achieve $O(\log n)$ search time and $O(n^{\sqrt{2/\log n}}(\log n)^{3/2})$ update time. Recently, Munro [9, 10] created implicit dictionaries that use $O((\log n)^2)$ time for both search and update. His basic approach is to have the order of elements within blocks of the array implicitly represent pointers.

Throughout this paper, we employ a comparison-based model with elements drawn from a totally ordered universe. Other models, allowing more general tests or in which elements are drawn from a very small universe [12], will not be considered here.

Munro and Suwanda [11] proved that if the contents of the array locations are constrained to satisfy some fixed partial order, then the product of search and update time is $\Omega(n)$. Thus, their triangular grid scheme, which achieves an $O(n)$ product of search and update time, is optimal among this class of algorithms.

Partial orders naturally arise as the
Lynch, and Yao [3] considered the c
tradeoff between preprocessing and sear
of comparisons performed to respectivel
array of length n then, in the worst cas
proved that this result also holds in the

In general, the information available
the array locations in a dictionary *canno*
for example, the rotated list. Since ev
minimum element, the only partial orde
is the trivial partial order (containing n
relative order of any allowable sequenc
by a permutation. An implicit diction
permutations together with an associate
algorithm.

Suppose that the n elements of an a
of p different permutations. If all com
involve the element being sought, then
[4] prove that $\Omega(p^{1/n})$ comparisons ar
between two elements in the array, whic
not allowed.) This lower bound is even
ministic algorithms [1]. Notice, howeve
rotated list is, in the worst case, as har

In this paper, we provide a general tra
time for the implicit dictionary problem
model and then consider a number of si
we show that if update is easy, such a
difficult. In particular, constant update
$\Omega(n^{\varepsilon})$ for some constant $\varepsilon > 0$.

2. The tradeoff

Associated with an implicit dictionary
the possible relative orders of the conte
algorithm for the dictionary is a compa
the array x, provided that the permut
contents of the locations of x belongs
internal nodes of this tree are labelled b
$y : x[j]$. For convenience, we assume tha
thus the equality branch will not be take
the element being sought is not in the a
at any comparison.

es, can be used to implement dictionaries
be done in O(log n) steps, where n is the
wever, explicit pointers are used and extra
ion to the space used to store the elements).
cant.
this extra space is needed to efficiently
though heaps are naturally implemented
ey can also be represented without extra
nents in locations $2i$ and $2i+1$ are the left
ment in location i.
n which only the number of elements and
itly. The n elements are stored in an array
of holding exactly one element.
ent dictionaries implicitly. If we store the
e can be done in constant time, but search
f the dictionary is maintained as an array
n be done in logarithmic time, but updates
array be moved. Rotated lists are arrays
by performing a cyclic shift (rotation) of
fficult to search, but only half the elements
ice that, by the information theory lower
orithm must perform at least a logarithmic
.
o consider explicitly the implicit dictionary
are stored partially sorted into a triangular
rformed in $O(\sqrt{n})$ steps. Using blocks of
er), they were able to improve the search
es per update at $O(\sqrt{n})$, and only increase
$O(\sqrt{n}\log n)$. Combining these ideas, they
can be searched or updated in $O(n^{1/3}\log n)$
ve manner, Frederickson [6] was able to
$^{n}(\log n)^{3/2})$ update time. Recently, Munro
use $O((\log n)^2)$ time for both search and
ne order of elements within blocks of the

comparison-based model with elements
Other models, allowing more general tests
small universe [12], will not be considered

if the contents of the array locations are
l order, then the product of search and
lar grid scheme, which achieves an $O(n)$
timal among this class of algorithms.

Partial orders naturally arise as the result of preprocessing. Borodin, Guibas, Lynch, and Yao [3] considered the closely related problem of determining the tradeoff between preprocessing and search time. If $P(n)$ and $S(n)$ denote the number of comparisons performed to respectively preprocess and search an initially unsorted array of length n then, in the worst case, $P(n)+n\log S(n)$ is $\Omega(n\log n)$. Mairson proved that this result also holds in the average case [8].

In general, the information available about the relative order of the contents of the array locations in a dictionary *cannot* be described by a partial order. Consider, for example, the rotated list. Since every location in the array can contain the minimum element, the only partial order guaranteed to be satisfied by a rotated list is the trivial partial order (containing no relations among elements). However, the relative order of any allowable sequence of elements in the array *can* be described by a permutation. An implicit dictionary can be viewed as a set of allowable permutations together with an associated search algorithm and an associated update algorithm.

Suppose that the n elements of an array can be arranged according to any one of p different permutations. If all comparisons performed during a search must involve the element being sought, then Alt, Mehlhorn, and Munro [2, 1], and Cook [4] prove that $\Omega(p^{1/n})$ comparisons are needed in the worst case. (Comparisons between two elements in the array, which may help to identify the permutation, are not allowed.) This lower bound is even true in the average case and for nondeterministic algorithms [1]. Notice, however, that under these assumptions searching a rotated list is, in the worst case, as hard as searching a completely unordered list.

In this paper, we provide a general tradeoff between worst-case search and update time for the implicit dictionary problem. We begin by describing our computational model and then consider a number of situations in which search is difficult. Finally, we show that if update is easy, such a situation must occur and, hence, search is difficult. In particular, constant update time implies a worst case search time of $\Omega(n^{\varepsilon})$ for some constant $\varepsilon>0$.

2. The tradeoff

Associated with an implicit dictionary, there is a set of permutations that describe the possible relative orders of the contents of the locations in the array. A search algorithm for the dictionary is a comparison tree that determines whether y is in the array x, provided that the permutation describing the relative order of the contents of the locations of x belongs to the set of allowable permutations. The internal nodes of this tree are labelled by comparisons of the forms $x[i]:x[j]$ and $y:x[j]$. For convenience, we assume that the elements of the array are distinct and thus the equality branch will not be taken at any comparison of the first type. When the element being sought is not in the array, the equality branch will not be taken at any comparison.

The update algorithm for the dictionary is given an array location l and a new value y as input. Depending on the value of l, it performs a sequence of comparisons involving y and the elements of the array. It then replaces the element in location l of the array by the value y. Based on the outcome of the comparisons, it finally rearranges the elements in at most m array locations so that the permutation associated with the resulting array is in the set of allowable permutations. We say that such a dictionary performs at most m moves per update.

Another way to specify an update is to provide two values y and y'. Algorithms would be required to replace the value y by the value y' and then rearrange the contents of the array as above. Such an approach requires that a search for the value y be performed as part of the update and, although it is perhaps more natural, it would make the tradeoff result of this paper meaningless.

In certain instances, it is easy to prove lower bounds on the search time. For example, if the set of arrays depicted in Fig. 1 were all allowed by a dictionary, then the search algorithm would have worst case complexity n. In particular, no information about the relative values of the contents of the other locations in the array can help us to determine if there is a location containing the value 1.

2	4	6	8	···	$2n-2$	$2n$
1	4	6	8	···	$2n-2$	$2n$
2	1	6	8	···	$2n-2$	$2n$
2	4	1	8	···	$2n-2$	$2n$
⋮						
2	4	6	8	···	$2n-2$	1

Fig. 1. A set of arrays that hides the value 1.

In Lemmas 2.1 and 2.2, we shall look at generalizations of the above situation. We obtain our tradeoff between search and update by showing, in Theorem 2.3, that some such situation must occur.

Let $d \geq 0$ and $n > 0$ be integers and let $I \subseteq \{1, \ldots, n\}$. For each $i \in I \cup \{0\}$, suppose x_i is an array of length n containing distinct numbers. For $i \in I$, let

$$D_i = \{j \in \{1, \ldots, n\} - \{i\} \mid x_i[j] \neq x_0[j]\}$$

denote the set of locations, excluding i, where x_i differs from x_0. Furthermore, suppose that, for all $i \in I$, $\# D_i \leq d$ and $x_i[i] = v$ where v is a number not appearing in the array x_0. The sequence of arrays in Fig. 2 satisfies these conditions with $v = 77$ and $d = 2$.

x_0	63	12	82
x_1	77	12	13
x_2	63	77	82
x_4	63	12	82
x_5	63	12	82
x_6	63	59	82

Fig. 2. A set of arrays tha

Lemma 2.1. *Any implicit dictionary* c *performs at most* $c \geq 1$ *comparisons per comparisons per search, in the worst*

Proof. By induction on d. If $d = 0$, some input, examine all possible loc that there is some location l such th x_0 in that location. We then consider l with a new value. For a *large* fract performed by the update algorithm. $d - 1$.

Let T be a comparison tree for Without loss of generality, we may some problem instance (x, y) where is a value whose membership in the depth of T. Thus S is the worst-case

Choose u to be a number differe all $i \in I \cup \{0\}$ and all $j \in \{1, \ldots, n\}$ the same outcome. Notice that x_i i u. Because T gives different answer them to different leaves. However, two inputs differ. Therefore, their $y : x[i]$.

In fact, any path from the root to the form $y : x[j]$. Otherwise, conside this leaf for some value of y. Since path, (x, y) reaches the same leaf that T determines whether x conta

y is given an array location l and a new
l, it performs a sequence of comparisons
It then replaces the element in location
e outcome of the comparisons, it finally
rray locations so that the permutation
e set of allowable permutations. We say
moves per update.
provide two values y and y'. Algorithms
by the value y' and then rearrange the
approach requires that a search for the
and, although it is perhaps more natural,
aper meaningless.
lower bounds on the search time. For
Fig. 1 were all allowed by a dictionary,
rst case complexity n. In particular, no
ne contents of the other locations in the
a location containing the value 1.

···	$2n-2$	$2n$
···	$2n-2$	$2n$
···	$2n-2$	$2n$
···	$2n-2$	$2n$
···	$2n-2$	1

nat hides the value 1.

t generalizations of the above situation.
nd update by showing, in Theorem 2.3,

$\in \{1, \ldots, n\}$. For each $i \in I \cup \{0\}$, suppose
ct numbers. For $i \in I$, let

$[j]\}$

where x_i differs from x_0. Furthermore,
$] = v$ where v is a number not appearing
g. 2 satisfies these conditions with $v = 77$

x_0	63	12	82	18	80	13	23	30
x_1	77	12	13	18	80	82	23	30
x_2	63	77	82	54	80	13	23	30
x_4	63	12	82	77	80	13	54	30
x_5	63	12	82	18	77	13	45	30
x_6	63	59	82	18	80	77	94	30

Fig. 2. A set of arrays that satisfies the conditions of Lemma 2.1.

Lemma 2.1. *Any implicit dictionary of size n that allows the arrays x_i for all $i \in I$ and performs at most $c \geq 1$ comparisons per update must perform at least $(\#I/2^{(c+1)d})^{1/(d+1)}$ comparisons per search, in the worst case.*

Proof. By induction on d. If $d = 0$, we show that any search algorithm must, for some input, examine all possible locations in which v may occur. If $d > 0$, we show that there is some location l such that a *large* number of the arrays x_i differ from x_0 in that location. We then consider the effect, on these arrays, of updating location l with a new value. For a *large* fraction of these arrays, exactly the same moves are performed by the update algorithm. This results in an instance of the problem for $d-1$.

Let T be a comparison tree for searching any array allowed by the dictionary. Without loss of generality, we may assume that every leaf of T can be reached by some problem instance (x, y) where x is an array allowed by the dictionary and y is a value whose membership in the array x is to be determined. Let S denote the depth of T. Thus S is the worst-case number of comparisons performed in any search.

Choose u to be a number different from, but sufficiently close to v so that, for all $i \in I \cup \{0\}$ and all $j \in \{1, \ldots, n\} - \{i\}$, the comparisons $v : x_i[j]$ and $u : x_i[j]$ have the same outcome. Notice that x_i is an allowable array which contains v, but not u. Because T gives different answers for the inputs (x_i, u) and (x_i, v), it must send them to different leaves. However, $y : x[i]$ is the only comparison on which these two inputs differ. Therefore, their root-to-leaf paths must split at the comparison $y : x[i]$.

In fact, any path from the root to a leaf must contain at least one comparison of the form $y : x[j]$. Otherwise, consider any allowable array x such that (x, y) reaches this leaf for some value of y. Since y is not involved in any comparisons along the path, (x, y) reaches the same leaf for all values of y. But this contradicts the fact that T determines whether x contains the value y.

Consider the root-to-leaf path P taken by the input (x_0, u). Note that x_0 is not necessarily an array allowed by the dictionary. Let a be the number of comparisons of the form $y:x[j]$ and let b be the number of comparisons of the form $x[j]:x[k]$ on the path P. Then $S \geq a+b$ and $a \geq 1$.

First, suppose $d=0$. Let $J=\{i \in I \mid$ the comparison $y:x[i]$ does not occur on the path $P\}$. Then $\#J \geq \#I - a$. Since $i \in J$, the comparison $y:x[i]$ does not occur on the path P. For all $i \in J$, the root-to-leaf path taken by the input (x_i, u) contains the comparison $y:x[i]$ and thus deviates from the path P at some point. Now x_i and x_0 agree everywhere except at location i. Therefore, the deviation must occur at a comparison involving $x[i]$ and some other element of x. For each $i \in J$, the point of deviation from the path P is different. To see this, suppose that $i, j \in J$ and that the paths taken by inputs (x_i, u) and (x_j, u) both deviate from path P at the comparison $x[i]:x[j]$. If P follows the $<$ branch at this comparison (i.e., $x_0[i] < x_0[j]$), then the inputs (x_j, u) and (x_i, u) take the other branch (i.e., $x_j[i] > x_j[j]$ and $x_i[i] > x_i[j]$). But this is impossible since $x_0[i] = x_j[i]$, $x_j[j] = v = x_i[i]$, and $x_i[j] = x_0[j]$. Similarly, $x_0[i] > x_0[j]$ leads to a contradiction. Thus $b \geq \#J$, $S \geq a+b \geq \#I$, and the lemma is true for $d=0$.

Now suppose $d \geq 1$ and assume the lemma is true for $d-1$. Let $J=\{i \in I \mid i \neq j, k$ for all comparisons $y:x[j]$ and $x[j]:x[k]$ on the path $P\}$. Then $\#J \geq \#I - a - 2b$. As above, all of the inputs (x_i, u), with $i \in J$, deviate from the path P at some point. If this occurs at a comparison $y:x[j]$, then $j \in D_i$. If this occurs at a comparison $x[j]:x[k]$, then either j or k is an element of D_i. Therefore, there is a subset $K \subseteq J$ containing at least $1/(a+2b)$ of the elements in J such that the sets D_i, for all $i \in K$, have an element in common, i.e., $\bigcap_{i \in K} D_i \neq \emptyset$. Since $a+b \leq S$ and $a \geq 1$, it follows that

$$\#K \geq \frac{\#J}{a+2b} \geq \frac{\#I}{2S-1} - 1.$$

Let $l \in \bigcap_{i \in K} D_i$. Consider the effect on the arrays x_i, $i \in K$, of performing an update that replaces the element in location l with a new value w. Here w is chosen so that it is not an element of array x_i for any $i \in K \cup \{0\}$. The moves that are performed in response to this update depend only on the sequence of outcomes of the c comparisons performed. Because all of the elements being compared are distinct, the outcome of any comparison cannot be equality. There are 2^c different sequences of outcomes possible and, hence, there is a subset $L \subseteq K$, with $\#L \geq \#K/2^c$, such that the same moves are performed on all arrays x_i with $i \in L$. Let $\tau:\{1, \ldots, n\} \to \{1, \ldots, n\}$ be the permutation describing this set of moves.

Viewed properly, we now have an instance of the original problem for $d-1$. Specifically, let $I' = \tau(L)$ and, for $j \in \{1, \ldots, n\}$ and $i \in I'$, let

$$x_0'[j] = \begin{cases} w & \text{if } j = \tau(l), \\ x_0[\tau^{-1}(j)] & \text{if } j \neq \tau(l) \end{cases}$$

and

$$x_i'[j] = \begin{cases} w & \text{if} \\ x_{\tau^{-1}(i)}[\tau^{-1}(j)] & \text{if} \end{cases}$$

Because v does not appear in the ar
If $i \in I'$, then $\tau^{-1}(i) \in L \subseteq K$. Note t
$l \notin D_l$. Hence, $l \neq \tau^{-1}(i)$ and $x_i'[i]$
does not contain duplicate entries
$i \in I' \cup \{0\}$, the elements of the arra

$$\begin{aligned} D_i' &= \{j \in \{1, \ldots, n\} - \{i\} \mid x \\ &= \{j \in \{1, \ldots, n\} - \{i, \tau(\\ &= \{k \in \{1, \ldots, n\} - \{\tau^{-1} \\ &= D_{\tau^{-1}(i)} - \{l\} \end{aligned}$$

and, because $l \in D_{\tau^{-1}(i)}$ for all $\tau^{-1}(i$
By the induction hypothesis
$(\#I/(2S-1)-1)/2^c$; therefore,

$$\begin{aligned} \#I &\leq (2^c \#I' + 1)(2S-1) \\ &\leq (2^c 2^{(c+1)(d-1)} S^d + 1)(\\ &= 2^{(c+1)d} S^{d+1} - 2^c 2^{(c+1} \\ &\leq 2^{(c+1)d} S^{d+1} - 1 \\ &\leq 2^{(c+1)d} S^{d+1}. \end{aligned}$$

Hence, the lemma is true for d. I
$d \geq 0$. □

We now consider another situat
array of length n containing the s
be a different value. For $i \in I \subseteq \{1,$
all n values in the set $\{v_0, v_1, \ldots,$

$$D_i = \{j \in \{1, \ldots, n\} - \{i\} \mid$$

Furthermore, suppose $\#D_i \leq d$.

These conditions are similar to
require v_0 to be in different locati
elements in each array z_i is com
sequence of arrays in Fig. 3, with

and

$$x_i'[j] = \begin{cases} w & \text{if } j = \tau(l), \\ x_{\tau^{-1}(i)}[\tau^{-1}(j)] & \text{if } j \neq \tau(l). \end{cases}$$

Because v does not appear in the array x_0, it does not appear in the array x_0' either. If $i \in I'$, then $\tau^{-1}(i) \in L \subseteq K$. Note that $l \notin K$ since the definition of D_l implies that $l \notin D_l$. Hence, $l \neq \tau^{-1}(i)$ and $x_i'[i] = x_{\tau^{-1}(i)}[\tau^{-1}(i)] = v$. For $i \in I \cup \{0\}$, the array x_i does not contain duplicate entries and w is not contained in x_i. Therefore, for $i \in I' \cup \{0\}$, the elements of the array x_i' are distinct. Finally,

$$\begin{aligned} D_i' &= \{j \in \{1, \ldots, n\} - \{i\} \mid x_i'[j] \neq x_0'[j]\} \\ &= \{j \in \{1, \ldots, n\} - \{i, \tau(l)\} \mid x_{\tau^{-1}(i)}[\tau^{-1}(j)] \neq x_0[\tau^{-1}(j)]\} \\ &= \{k \in \{1, \ldots, n\} - \{\tau^{-1}(i), l\} \mid x_{\tau^{-1}(i)}[k] \neq x_0[k]\} \\ &= D_{\tau^{-1}(i)} - \{l\} \end{aligned}$$

and, because $l \in D_{\tau^{-1}(i)}$ for all $\tau^{-1}(i) \in L = \tau^{-1}(I')$, it follows that $\#D_i' \leq d-1$.

By the induction hypothesis, $S \geq (\#I'/2^{(c+1)(d-1)})^{1/d}$. Now $\#I' = \#L \geq (\#I/(2S-1)-1)/2^c$; therefore,

$$\begin{aligned} \#I &\leq (2^c \#I' + 1)(2S-1) \\ &\leq (2^c 2^{(c+1)(d-1)} S^d + 1)(2S-1) \\ &= 2^{(c+1)d} S^{d+1} - 2^c 2^{(c+1)(d-1)} S^d + 2S - 1 \\ &\leq 2^{(c+1)d} S^{d+1} - 1 \qquad \text{(since } c, d \geq 1\text{)} \\ &\leq 2^{(c+1)d} S^{d+1}. \end{aligned}$$

Hence, the lemma is true for d. By induction, the lemma is true for all integers $d \geq 0$. □

We now consider another situation in which searching is difficult. Let z_0 be an array of length n containing the sequence $v_1, \ldots, v_n$ of distinct values and let v_0 be a different value. For $i \in I \subseteq \{1, \ldots, n\}$, let z_i be an array of length n containing all n values in the set $\{v_0, v_1, \ldots, v_n\} - \{v_i\}$ and let

$$D_i = \{j \in \{1, \ldots, n\} - \{i\} \mid z_0[j] \neq z_i[j]\}.$$

Furthermore, suppose $\#D_i \leq d$.

These conditions are similar to the conditions for Lemma 2.1. Here we do not require v_0 to be in different locations for each of the z_i arrays. However, the set of elements in each array z_i is completely determined. For example, consider the sequence of arrays in Fig. 3, with $v_0 = 77$ and $d = 2$.

z_0	63	12	82	18	80	13	23	30
z_1	77	12	13	18	80	82	23	30
z_2	63	80	82	18	77	13	23	30
z_4	63	12	82	80	77	13	23	30
z_5	63	12	82	18	23	13	77	30
z_6	63	80	82	18	77	12	23	30

Fig. 3. A set of arrays that satisfies the conditions of Lemma 2.2.

Lemma 2.2. *Any implicit dictionary that allows the arrays z_i for all $i \in I$ and performs at most $c \geq 1$ comparisons per update must perform at least*

$$\left\lfloor \left(\frac{\#I^{2/(d+1)}}{2^{(c+1)d}} \right)^{1/(d+2)} \right\rfloor$$

comparisons per search, in the worst case.

Proof. By induction on d. Let T be a comparison tree for searching any array allowed by the dictionary and let S denote the depth of T. Essentially, we shall show that there is some value which occurs at many different locations in the z_i arrays, and this will enable us to apply Lemma 2.1.

If $d = 0$, then $z_i[i] = v_0$ for all $i \in I$. By Lemma 2.1, $S \geq \#I$. Now let $d \geq 1$ and assume the lemma is true for $d - 1$.

Let $L = \{l \in \{1, \ldots, n\} \mid z_i[l] = v_0 \text{ for some } i \in I\}$ denote the set of locations where v_0 occurs and, for each $l \in L$, let $I_l = \{i \in I \mid z_i[l] = v_0\}$ be the indices of those arrays in which v_0 occurs in location l.

We first consider the case when

$$\#I_l \leq \left\lfloor \left(\frac{\#I}{2^{(c+1)}} \right)^{d/(d+2)} \right\rfloor \quad \text{for all } l \in L.$$

Then

$$\#L \geq \#I \Big/ \left\lfloor \left(\frac{\#I}{2^{(c+1)}} \right)^{d/(d+2)} \right\rfloor \geq (\#I^2 2^{(c+1)d})^{1/(d+2)}.$$

In this case, the value v_0 occurs in a large number of different locations and Lemma 2.1 can be applied directly.

More formally, let $z_0' = z_0$ and, for ea
$D_l' = \{j \in \{1, \ldots, n\} - \{l\} \mid z_l'[j] \neq z_0'[j]\} =$
we know that $l \in D_i \cup \{i\}$. Thus $\#D_l' \leq$

$$S \geq \left(\frac{\#L}{2^{(c+1)d}} \right)^{1/(d+1)} \geq \left(\frac{(\#I^2 2^{(}}{2} \right.$$

$$= \left(\frac{\#I^{2/(d+1)}}{2^{(c+1)d}} \right)^{1/(d+2)} \geq \left\lfloor \left(\frac{\#I}{2^{(} \right. \right.$$

Now, we suppose there is a location

$$\#I_l \geq \left\lfloor \left(\frac{\#I}{2^{(c+1)}} \right)^{d/(d+2)} \right\rfloor + 1.$$

Thus location l contains the value v_0 fo

Viewed properly, we have an inst
Specifically, define

$$z_0'[j] = \begin{cases} z_0[j] & \text{for } j \neq l, \\ v_0 & \text{for } j = l \end{cases}$$

and

$$v_i' = \begin{cases} v_l & \text{for } i = 0, \\ v_0 & \text{for } i = l, \\ v_i & \text{for } i \in \{1, \ldots, n\} - \{ \end{cases}$$

Let $I' = I_l - \{l\}$ and let $z_i' =$
$\{j \in \{1, \ldots, n\} - \{i\} \mid z_0'[j] \neq z_i'[j]\} = D_i -$

By the induction hypothesis,

$$S \geq \left\lfloor \left(\frac{(\#I')^{2/d}}{2^{(c+1)(d-1)}} \right)^{1/(d+1)} \right\rfloor.$$

Since

$$\#I' \geq \left\lfloor \left(\frac{\#I}{2^{(c+1)}} \right)^{d/(d+2)} \right\rfloor,$$

we have

$$S \geq \left\lfloor \left(\frac{\left\lfloor \left(\frac{\#I}{2^{(c+1)}} \right)^{d/(d+2)} \right\rfloor^{2/d}}{2^{(c+1)(d-1)}} \right)^{1/} \right.$$

We use the facts that, for all positive in
and $\lfloor r \rfloor^{1/b} \geq \lfloor r^{1/b} \rfloor$. Since $2^{(c+1)(d-1)d/2}$

$$S \geq \left\lfloor \left\lfloor \frac{\left(\frac{\#I}{2^{(c+1)}} \right)^{d/(d+2)}}{2^{(c+1)(d-1)d/2}} \right\rfloor^{2/d(d+1)} \right.$$

More formally, let $z'_0 = z_0$ and, for each $l \in L$, choose $z'_l = z_i$ for some $i \in I_l$. Then $D'_l = \{j \in \{1, \ldots, n\} - \{l\} \mid z'_l[j] \neq z'_0[j]\} = D_i \cup \{i\} - \{l\}$. Since $z_i[l] = v_0 \neq z_0[l] = z'_0[l]$, we know that $l \in D_i \cup \{i\}$. Thus $\# D'_l \leq \# D_i \leq d$. By Lemma 2.1,

$$S \geq \left(\frac{\# L}{2^{(c+1)d}}\right)^{1/(d+1)} \geq \left(\frac{(\# I 2^{(c+1)d})^{1/(d+2)}}{2^{(c+1)d}}\right)^{1/(d+1)}$$

$$= \left(\frac{\# I^{2/(d+1)}}{2^{(c+1)d}}\right)^{1/(d+2)} \geq \left\lfloor \left(\frac{\# I^{2/(d+1)}}{2^{(c+1)d}}\right)^{1/(d+2)} \right\rfloor.$$

Now, we suppose there is a location $l \in L$ such that

$$\# I_l \geq \left\lfloor \left(\frac{\# I}{2^{(c+1)}}\right)^{d/(d+2)} \right\rfloor + 1.$$

Thus location l contains the value v_0 for a large number of the z_i arrays.

Viewed properly, we have an instance of the original problem for $d-1$. Specifically, define

$$z'_0[j] = \begin{cases} z_0[j] & \text{for } j \neq l, \\ v_0 & \text{for } j = l \end{cases}$$

and

$$v'_i = \begin{cases} v_l & \text{for } i = 0, \\ v_0 & \text{for } i = l, \\ v_i & \text{for } i \in \{1, \ldots, n\} - \{l\}. \end{cases}$$

Let $I' = I_l - \{l\}$ and let $z'_i = z_i$ for all $i \in I'$. Then $D'_i = \{j \in \{1, \ldots, n\} - \{i\} \mid z'_0[j] \neq z'_i[j]\} = D_i - \{l\}$ for all $i \in I'$; hence, $\# D'_i \leq d-1$.

By the induction hypothesis,

$$S \geq \left\lfloor \left(\frac{(\# I')^{2/d}}{2^{(c+1)(d-1)}}\right)^{1/(d+1)} \right\rfloor.$$

Since

$$\# I' \geq \left\lfloor \left(\frac{\# I}{2^{(c+1)}}\right)^{d/(d+2)} \right\rfloor,$$

we have

$$S \geq \left\lfloor \left(\frac{\left\lfloor \left(\frac{\# I}{2^{(c+1)}}\right)^{d/(d+2)} \right\rfloor^{2/d}}{2^{(c+1)(d-1)}} \right)^{1/(d+1)} \right\rfloor = \left\lfloor \left(\frac{\left\lfloor \left(\frac{\# I}{2^{(c+1)}}\right)^{d/(d+2)} \right\rfloor}{2^{(c+1)(d-1)d/2}} \right)^{2/d(d+1)} \right\rfloor.$$

We use the facts that, for all positive integers b and real numbers r, $\lfloor r \rfloor / b \geq \lfloor r/b \rfloor$ and $\lfloor r \rfloor^{1/b} \geq \lfloor r^{1/b} \rfloor$. Since $2^{(c+1)(d-1)d/2}$ and $d(d+1)/2$ are integers,

$$S \geq \left\lfloor \left\lfloor \frac{\left(\frac{\# I}{2^{(c+1)}}\right)^{d/(d+2)}}{2^{(c+1)(d-1)d/2}} \right\rfloor^{2/d(d+1)} \right\rfloor \geq \left\lfloor \left(\frac{\left(\frac{\# I}{2^{(c+1)}}\right)^{d/(d+2)}}{2^{(c+1)(d-1)d/2}} \right)^{2/d(d+1)} \right\rfloor$$

$$= \left\lfloor \left(\frac{\# I^{2/(d+1)}}{2^{(c+1)d}} \right)^{1/(d+2)} \right\rfloor .$$

Thus the claim is true for d and, hence, by induction, for all integers $d \geq 0$. □

Finally, we prove the desired tradeoff between update and search time.

Theorem 2.3. *Any implicit dictionary for length-n arrays that performs at most m moves and $c \geq 1$ comparisons per update must perform at least $\lfloor (n^{2/(m+1)}/2^{(c+1)m})^{1/(m+2)} \rfloor$ comparisons per search, in the worst case.*

Proof. Let $I = \{1, \ldots, n\}$, let z_0 be any allowable array in the dictionary, and let v be a value not in z_0. For $i \in I$, let z_i be the array that is obtained from z_0 using the update algorithm to replace the element in location i with the value v.

For $i \in I$, define $D_i = \{j \in \{1, \ldots, n\} - \{i\} \mid z_i[j] \neq z_0[j]\}$. Since at most m moves per update are performed, $\# D_i \leq m$. The result follows from the application of Lemma 2.2. □

Corollary 2.4. *Any implicit dictionary for length n arrays that has constant update time has $\Omega(n^{\varepsilon})$ search time for some constant $\varepsilon > 0$.*

3. Conclusions

The technique used to obtain the tradeoff between search and update time may consider only a small fraction of the allowable permutations. Furthermore, it allows the search and update algorithms to change after any update has been performed. Improvements to this tradeoff could therefore be possible. It remains unclear whether logarithmic update time must imply nonpolylogarithmic search time for implicit dictionaries.

A tradeoff between the number of moves per update and search time would also be interesting. For this result, it would not matter whether the element to be replaced during an update is specified by its value or its location. We believe that any dictionary in which only a constant number of moves per update are performed should very quickly get disorganized and, hence, be difficult to search.

One of the motivations for studying the implicit dictionary problem is to understand the relationship between the amount of time it takes to perform a search within an array and easily described properties of the set of allowable permutations. Specifically, we would like to characterize those sets of permutations for which searching is easy (i.e., $O(\log n)$ or $(\log n)^{O(1)}$ comparisons, in the worst case) and those for which searching is hard.

This is analogous to work done by Li
They showed that the logarithm of the
bound for the worst-case number of c
only information known about the c
satisfy the partial order.

We conjecture that if the set of all
worst-case search time must be large. N
not logarithmic or perhaps not even po
is $\Omega(n!/c^n)$ for some constant $c > 0$.

An interesting example of a large set
was devised by Feldman [5]. He consi
$\{1, \ldots, n\}$ that are involutions (i.e., π^2
fix all the even-numbered locations (i.
more than $(\frac{1}{4}n)!$ such permutations. Su
of elements in the array x can be describ
Searching for an element y in the arra
by the following procedure. First, binar
locations of x. Suppose y is not found.
location such that all even-numbered
than y and all even-numbered location
y. (If no such i exists, which happens
in the array x.) Next, the rank j of elen
search on the even-numbered location
with $x[j]$. Since the permutation asso
change locations i and j. Thus, if y is

Studying tradeoffs for the implicit d
rather a stepping stone towards a genera
problems. In particular, for dictiona
constant update time imply slow searc

References

[1] H. Alt and K. Mehlhorn, Searching semis
[2] H. Alt, K. Mehlhorn and J.I. Munro, Par
Process. Lett. **19** (1984) 61–65.
[3] A. Borodin, L. Guibas, N. Lynch and A.
Process. Lett. **12** (1981) 71–75.
[4] S. Cook, Personal communication, 1982.
[5] P. Feldman, Personal communication, 198
[6] G.N. Frederickson, Implicit data structure
[7] N. Linial and M. Saks, Information bounds
in: *Proc. 24th Ann. Symp. on Foundations*
[8] H. Mairson, Average case lower bounds
Proc. 26th Ann. Symp. on Foundations of C

This is analogous to work done by Linial and Saks [7] for searching partial orders. They showed that the logarithm of the number of ideals in a partial order is a lower bound for the worst-case number of comparisons needed to search an array if the only information known about the contents of the array locations are that they satisfy the partial order.

We conjecture that if the set of allowable permutations is very large, then the worst-case search time must be large. More specifically, the search time is probably not logarithmic or perhaps not even polylogarithmic if the number of permutations is $\Omega(n!/c^n)$ for some constant $c>0$.

An interesting example of a large set of permutations that can be searched quickly was devised by Feldman [5]. He considered the set of permutations $\pi:\{1,\ldots,n\}\to\{1,\ldots,n\}$ that are involutions (i.e., π^2 is the identity permutation) and, moreover, fix all the even-numbered locations (i.e., $\pi(2i)=2i$ for $i=1,\ldots,\lfloor\frac{1}{2}n\rfloor$). There are more than $(\frac{1}{4}n)!$ such permutations. Suppose that the relative order of the sequence of elements in the array x can be described by an unknown one of these permutations. Searching for an element y in the array x can be accomplished in logarithmic time by the following procedure. First, binary search is performed on the even-numbered locations of x. Suppose y is not found. Let i denote the unique odd-numbered array location such that all even-numbered locations less than i contain elements less than y and all even-numbered locations greater than i contain elements greater than y. (If no such i exists, which happens when n is even and $x[n]<y$, then y is not in the array x.) Next, the rank j of element $x[i]$ is determined by performing binary search on the even-numbered locations of x a second time. Finally, y is compared with $x[j]$. Since the permutation associated with x is an involution, it must interchange locations i and j. Thus, if y is in the array x, it is in location j.

Studying tradeoffs for the implicit dictionary problem is not an end in itself, but rather a stepping stone towards a general understanding of tradeoffs for data structure problems. In particular, for dictionaries that may have explicit pointers, does constant update time imply slow search?

References

[1] H. Alt and K. Mehlhorn, Searching semisorted tables, *SICOMP* **14** (1985) 840–848.

[2] H. Alt, K. Mehlhorn and J.I. Munro, Partial match retrieval in implicit data structures, *Inform. Process. Lett.* **19** (1984) 61–65.

[3] A. Borodin, L. Guibas, N. Lynch and A. Yao, Efficient searching using partial ordering, *Inform. Process. Lett.* **12** (1981) 71–75.

[4] S. Cook, Personal communication, 1982.

[5] P. Feldman, Personal communication, 1985.

[6] G.N. Frederickson, Implicit data structures for the dictionary problem, *J. ACM* **30** (1983) 80–94.

[7] N. Linial and M. Saks, Information bounds are good for search problems on ordered data structures in: *Proc. 24th Ann. Symp. on Foundations of Computer Science* (1983) 473–475.

[8] H. Mairson, Average case lower bounds on the construction and searching of partial orders, in: *Proc. 26th Ann. Symp. on Foundations of Computer Science* (1985) 303–311.

[9] J.I. Munro, An implicit data structure for the dictionary problem that runs in polylog time, in: *Proc. 25th Ann. Symp. on Foundations of Computer Science* (1984) 369–374.
[10] J.I. Munro, An implicit data structure supporting insertion, deletion and search in $O(\log^2 n)$ time, Manuscript, University of Waterloo, 1985.
[11] J.I. Munro and H. Suwanda, Implicit data structures, *J. Comput. System Sci.* **21** (1980) 236–250.
[12] A.C. Yao, Should tables be sorted?, *J. ACM* **3** (1981) 615–628.

Theoretical Computer Science 58 (1988) 69–
North-Holland

ON THE INTERSECTIO

Franz J. BRANDENBURG
Fakultät für Mathematik und Informa

Abstract. What do a pushdown stack
Is it a counter? If we add a retrieval re
and a queue, or, by symmetry, of a
pushdown and a one-reset queue? I
conjectured by Autebert et al. (1979),

We approach these problems in te
real-time machines whose storage tap
We disprove all conjectures from ab
intersection of pushdowns and queu
one-reset. In fact, there is a complet
languages and the context-free langua
that are defined by machines with a
restrictions respectively.

1. Introduction

The cover of Jean Berstel's bo
Languages" illustrates the relationsh
languages. A "?" at the intersectio
the one-counter languages, Rocl, p
is whether this intersection is ex
languages. In terms of machines th
are at the intersection of one-reve
the restrictions on a pushdown to o
alphabet independent and do they

Rephrased in terms of least trios o
holds or not? Autebert et al. [
[3, p. 208, 266] indicates the opposit
et al. conjecture that $\mathcal{M}(\mathrm{PAL}) \cap \mathcal{M}(C$
languages in $\mathcal{M}(\mathrm{COPY}) - \mathcal{M}(S)$. C
which in turn is a special case of a

We disprove all these conjecture
families of languages over PAL, C
we do not have an answer on the '

0304-3975/88/$3.50 © 1988, Elsevier Scien