A Tradeoff Between Search and Update Time for the Implicit Dictionary Problem

Allan Borodin, University of Toronto, Toronto, Canada
Faith E. Fich, University of Washington, Seattle, USA
Friedhelm Meyer auf der Heide, Johann Wolfgang Goethe Universität, Frankfurt, West Germany
Eli Upfal, IBM Almaden Research Center, San Jose, USA
Avi Wigderson, Mathematical Science Research Institute, Berkeley, USA

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 $\Omega(n^{\varepsilon})$ for some constant $\varepsilon > 0$.

1. Introduction

A dictionary is a data structure 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.

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 [MS] 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 searching requires linear time. On the other hand, if the dictionary is maintained as an array sorted in increasing order, then searching 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 difficu
case. Notice that, by the inform
perform at least a logarithmic nu

Munro and Suwanda [M
They showed that if the element
can be performed in $O(\sqrt{n})$ ste
were able to improve the search
only increase the number of con
produced an implicit dictionary
lists in a recursive manner,
$O(n^{\sqrt{2/\log n}}(\log n)^{3/2})$ update tin
$O((\log n)^2)$ time for both search
blocks of the array implicitly rep

Throughout this paper, w
ly ordered universe. Other mod
very small universe [Y] will not

Munro and Suwanda [M
satisfy some fixed partial order,
lar grid scheme, which achieves
of algorithms.

Partial orders naturally
[BGLY] considered the closely
search time. If $P(n)$ and $S(n)$
cess and search an initially un
$\Omega(n \log n)$. Mairson proved tha

In general, the informati
tions in a dictionary *cannot* be
Since every location in the arra
be satisfied by a rotated list is t
ever, the relative order of any al
tation. An implicit dictionary c
ated search algorithm and an ass

Suppose that the n elem
mutations. If all comparisons
Alt, Mehlhorn, and Munro [AM
the worst case. (Comparisons b
tation, are not allowed.) This lo
gorithms [AM]. Notice, howev
as hard as searching a completel

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 [MS] were the first to explicitly consider 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 [Fr] was able to achieve $O(\log n)$ search time and $O(n^{\sqrt{2/\log n}}(\log n)^{3/2})$ update time. Recently, Munro [M1], [M2] 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 [Y] will not be considered here.

Munro and Suwanda [MS] 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 result of preprocessing. Borodin, Guibas, Lynch, and Yao [BGLY] 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 [Ma].

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 [AMM], [AM] and Cook [C] 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 [AM]. Notice, however, 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.

2	4	6	8	$\cdots$	$2n-2$	$2n$
1	4	6	8	$\cdots$	$2n-2$	$2n$
2	1	6	8	$\cdots$	$2n-2$	$2n$
2	4	1	8	$\cdots$	$2n-2$	$2n$
			$\vdots$			
2	4	6	8	$\cdots$	$2n-2$	1

Figure 1. A Set of Arrays that Hides the Value 1

In certain instances, it is easy to prove lower bounds on the search time. For example, if the set of arrays depicted in Figure 1 were all allowed by a dictionary, then the search algorithm would have worst case complexity n. In pa
other locations in the array can h

In Lemmas 1 and 2, we
between search and update by sh

Let $d \geq 0$ and $n > 0$ be in
an array of length n containing
$D_i =$
denote the set of locations, exc
$i \in I$, $\#D_i \leq d$ and $x_i[i] = v$ w
quence of arrays satisfies these c

x_0

x_1

x_2

x_4

x_5

x_6

Figure 2. A S

Lemma 1. Any implicit dicti
most $c \geq 1$ comparisons per upd
worst case.

Proof: by induction on d. If d
all possible locations in which
large number of the arrays x_i d
rays, of updating location l wi
moves are performed by the upd

Let T be a comparison
generality, we may assume that
x is an array allowed by the dic
mined. Let S denote the depth
search.

Choose u to be a numbe
and all $j \in \{1, \ldots, n\} - \{i$

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.

In Lemmas 1 and 2, we look at generalizations of the above situation. We obtain our tradeoff between search and update by showing, in Theorem 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 following sequence of arrays satisfies these conditions with $v = 77$ and $d = 2$.

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	30	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

Figure 2. A Set of Arrays that Satisfies the Conditions of Lemma 1

Lemma 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 $\left(\frac{\#I}{2^{(c+1)d}} \right)^{\frac{1}{d+1}}$ 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. No-

tice 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 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 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 $\tau : \{ 1, \ldots, n \} \rightarrow \{ 1, \ldots, n \}$ 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

$$x_0'[j] = \begin{cases} w & \text{if } j = \tau(l) \\ x_0[\tau^{-1}(j)] & \text{if } j \neq \tau(l) \end{cases} \quad \text{and} \quad x_i'[j] = \begin{cases} w & \text{if } j = \tau(l) \\ x_{\tau^{-1}(i)}[\tau^{-1}(j)] & \text{if } j \neq \tau(l) \end{cases}.$$

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, $S \geq \left\lceil \frac{\#I'}{2^{(c+1)(d-1)}} \right\rceil^{\frac{1}{d}}$. Now $\#I' = \#L \geq \left\lceil \frac{\#I}{2S-1} - 1 \right\rceil / 2^c$; therefore

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

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

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

Figure 3. A Set of Arrays that Satisfies the Conditions of Lemma 2

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 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 Figure 3, with $v_0 = 77$ and $d = 2$.

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

If $d = 0$, then $z_i[i] = v_0$ for all $i \in I$. By Lemma 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 $\#I_l \leq \left\lfloor \left(\frac{\#I}{2^{(c+1)}} \right)^{\frac{d}{d+2}} \right\rfloor$ for all $l \in L$. Then

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

In this case, the value v_0 occurs in a large number of different locations and Lemma 1 can be applied directly.

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

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

Now, we suppose there is a location $l \in L$ such that $\#I_l \geq \left\lfloor \left(\frac{\#I}{2^{(c+1)}} \right)^{\frac{d}{d+2}} \right\rfloor + 1$. 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

$$z_0'[j] = \begin{cases} z_0[j] \text{ for } j \neq l \\ v_0 \text{ for } j = l \end{cases} \quad \text{and} \quad v_i' = \begin{cases} v_l \text{ for } i = 0 \\ v_0 \text{ for } i = l \\ v_i \text{ for } i \in \{1, \ldots, n\} - \{l\}. \end{cases}$$

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

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

We use the facts that, for all positive integers b and real numbers r, $\frac{\lfloor r \rfloor}{b} \geq \left\lfloor \frac{r}{b} \right\rfloor$ 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,

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

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 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 $\left\lfloor \left(\frac{n^{2/(m+1)}}{2^{(c+1)m}} \right)^{\frac{1}{m+2}} \right\rfloor$ 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. □

Corollary 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 Linial and Saks [LS] 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 [Fe]. He considered the set of permutations $\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ 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 $i = 1, \ldots, \lfloor n/2 \rfloor$). There are more than $(n/4)!$ 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?

Acknowledgements

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

References

[AM] H. Alt and K. Mehlhorn, *Searching Semisorted Tables*, SICOMP, vol. 14 no. 4, 1985, pages 840-843.

[AMM] H. Alt, K. Mehlhorn, and J.I. Munro, *Partial Match Retrieval in Implicit Data Structures*, Information Processing Letters, vol. 19 no. 2, 1984, pages 61-65.

[BGLY] A. Borodin, L. Guibas, N. Lynch, and A. Yao, *Efficient Searching Using Partial Ordering*, Information Processing Letters, vol. 12 no. 2, 1981, pages 71-75.

[C] S. Cook, personal communication.

[Fe] P. Feldman, personal communication.

[Fr] G.N. Frederickson, *Implicit Data Structures for the Dictionary Problem*, J. ACM, vol. 30 no. 1, 1983, pages 80-94.

[LS] N. Linial and M. Saks, *Information Bounds are Good for Search Problems on Ordered Data Structures*, 24th Annual Symposium on Foundations of Computer Science, 1983, pages 473-475.

[Ma] H. Mairson, *Average Case Lower Bounds on the Construction and Searching of Partial Orders*, 26th Annual Symposium on Foundations of Computer Science, 1985, pages 303-311.

[M1] Munro, J.I., *An Implicit Data Structure for the Dictionary Problem that Runs in Polylog Time*, 25th Annual Symposium on Foundations of Computer Science, 1984, pages 369-374.

[M2] Munro, J.I., *An Implicit Data Structure Supporting Insertion, Deletion and Search in $O(\log^2 n)$ Time*, manuscript, University of Waterloo, 1985.

[MS] J.I. Munro and H. Suwanda, *Implicit Data Structures*, JCSS, vol. 21, 1980, pages 236-250.

[Y] A.C. Yao, *Should Tables be Sorted?*, J.ACM vol. 3, 1981, pages 615-628.