Electronic Colloquium on Computational Complexity - Reports Series 1995 - available via:
ECCC
TR95-062
FTP: ftp.eccc.uni-trier.de:/pub/eccc/
WWW: http://www.eccc.uni-trier.de/eccc/
Email: ftpmail@ftp.eccc.uni-trier.de with subject 'help eccc'

Tel-Aviv University
The Raymond and Beverly Sackler
Faculty of Exact Sciences
School of Mathematics

On Data Structure Tradeoffs and an Application to Union-Find

Amir M. Ben-Amram*
and
Zvi Galil**

ABSTRACT

Consider a problem involving updates and queries of a data structure. Assume that there exists a family of algorithms which exhibit a tradeoff between query and update time. We demonstrate a general technique of constructing from such a family a single algorithm with best amortized time. We indicate some applications of this technique. On the other hand, when a given algorithm achieves similar amortized performance, we may be able to obtain from it a family of algorithms that exhibit the underlying tradeoff. We exemplify this by a family of union-find algorithms trading union time for find time. The algorithms are obtained in a simple way from the well known path compression algorithm.

* Work supported in part by a Charles Clore fellowship.
* Partially supported by NSF grant CCR-93-16209 and CISE Institutional Infrastructure Grant CDA-90-24735.

1. Introduction

A well known feature of data structure problems is the existence of tradeoffs between the update and query time. Actually, a common form of solutions to data structure problems is that of a family of algorithms $(\mathcal{A}_k)$, such that $\mathcal{A}_k$ performs a query in $O(k)$ time, while requiring $O(t_k(n))$ time per update. Here n is the number of elements inserted or updates performed, and $t_k(n)$ is a function which increases slower as k gets larger. Some examples appear in a later section. On the other hand, for certain problems we have solutions which optimize the overall time of processing a sequence of requests; such algorithms avoid investing too much in an update, but when queries recur, the data structure is gradually conditioned for faster processing of forthcoming ones. A well known example is the *union-find* algorithm [12].

In this paper we try to relate the two types of solutions. We show that given a solution of the first kind (a *tradeoff family*), we can (under plausible restrictions) derive a solution of the second kind. As for the other direction, we cannot give a general transformation, but it seems that in the "typical" case this should be possible. We give such derivation for the union-find algorithm. For a somewhat broader support for this thesis, we point out that several lower bounds on the performance of on-line algorithms suggest a tradeoff in the possible solutions [6,7,3].

2. Terms of the Game

A *data structure problem* involves processing two types of requests: *update*, which adds information to the structure, and *query*, which retrieves information. We assume that the data structure starts its life in a specified initial state; its life further spans n update operations and m queries. In an *on-line* solution, updates and queries are intermixed; the sequence in which they appear is not known in advance. If the total time spent in query operations throughout a sequence is mt we say that the *amortized query time* for this sequence is t. We say that an algorithm has an amortized query time of $q(m,n)$ if this is the worst-case time for all sequences of n updates and m queries. *Amortized update time* is similarly defined with respect to update operations. *Amortized operation time* refers to both types of operations, thus it will be $t(m,n)$ if the time spent in processing all the $m+n$ operations is bounded by $(m+n)t(m,n)$.

A *preprocessing* algorithm is one that receives a list of all the updates, and after processing them starts receiving queries on line. Our analysis applies to preprocessing algorithms as a special case of on-line ones, however following the general treatment we indicate how the preprocessing case can be handled more directly.

We are interested in families of algorithms $(\mathcal{A}_k)$ such that the query time for $\mathcal{A}_k$ is $O(k)$ and the update time is $O(t_k(n))$ for a suitable sequence of functions t_k. Such a family will be called a tradeoff family. A sequence of functions $t_k : \mathbb{N} \to \mathbb{N}$ is *suitable* if it has the following properties (for all $k, n \geq 1$):

(1) $t_k(1) = 1$ and $t_k(n) \leq t_k(n+1) \leq t_k(n) + 1$.

(2) $t_k(n) > 1 \;\rightarrow\; t_{k+1}(n) \leq {}^1\!/_2 t_k(n)$.

(3) $t_k(2n) \leq 2t_k(n)$.

Conditions (1) and (3) indicate that $t_k(n) \leq n$ (note this is the time for a single update) and that t_k does not grow in sudden leaps, but with a certain regularity. Condition (2) indicates that $t_k(n)$ decreases fast as k increases.

By a *functional hierarchy* we mean a matrix $T(k,n)$ such that $T(k,\cdot)$ (the row function) increases faster as k grows. A suitable sequence of functions might be obtained by inverting the row functions of such a hierarchy; formally, we define

$$T(k,j) = \min\{\, n \mid t_k(n) > j \,\}. \tag{4}$$

Claim 1. *For every suitable family (t_k), the function T defined by (4) satisfies*

$$t_k(n) = \min\{\, j \mid T(k,j) > n \,\}.$$

It also satisfies (for all $k \geq 1$):

$$T(k+1,1) \geq 2T(k,1).$$

Proof. We define To see that it satisfies the first part of the claim, let

$$j_0 = \min\{\, j \mid T(k,j) > n \,\}.$$

Then $T(k,j_0) > n$. By definition of T this implies

$$t_k(n) \leq j_0\,.$$

The definition of j_0 also implies that $T(k,j_0-1) \leq n$; hence by (4)

$$t_k(n) > j_0 - 1 \implies t_k(n) \geq j_0\,.$$

Combining the two inequalities we get $t_k(n) = j_0$, proving the first part of the claim.

For the second part, let

$$n_0 = T(k,1) = \min\{\, n \mid t_k(n) > 1 \,\}.$$

Thus

$$\begin{aligned}
& t_k(n_0 - 1) = 1 \\
\implies & t_k(2n_0 - 2) \leq 2 && \text{by (3)} \\
\implies & t_k(2n_0 - 1) \leq 3 && \text{by (1)} \\
\implies & t_{k+1}(2n_0 - 1) = 1 && \text{by (2)} \\
\implies & T(k+1,1) \geq 2n_0 && \text{by (4).}
\end{aligned}$$

□

We next define the "diagonal" function

$$t(m,n) = \min\{\, k \mid T(k,[m/n]) > n \,\}$$

where $[m/n] \stackrel{\text{def}}{=} \max(1, \lfloor \frac{m}{n} \rfloor)$.

We next display the connection between this function and the running time of algorithms. Consider a tradeoff family $(\mathcal{A}_k)$. The running time of $\mathcal{A}_k$ on a sequence of n updates and m queries is (ignoring constant factors) $nt_k(n) + mk$. The best total running time will be obtained by choosing k as to minimize this expression.

Claim 2. *Let* $f_k(m,n) = nt_k(n)+mk$. *Then* $f_{t(m,n)}(m,n) = O(n+mt(m,n))$. *Moreover, for all* k, $f_k(m,n) = \Omega(f_{t(m,n)}(m,n))$.

The claim shows that the value $k = t(m,n)$ approximately minimizes $f_k(n,m)$.

Proof. Because the proof is quite technical, we only give the main ideas. On one hand, note that for $k > t(m,n)$,

$$f_k(m,n) > nt_k(n) + mt(m,n) \geq n + mt(m,n)\,.$$

On the other hand, for $k < t(m,n)$, let $\ell = t(m,n) - k$; then $t_k(n) \geq 2^\ell t_{t(m,n)}(n)$ because of (2). Thus decreasing k makes the $nt_k(n)$ term of $g_k(m,n)$ grow exponentially while the term mk decreases linearly. Therefore $f_k(m,n)$ becomes larger than $f_{t(m,n)}(m,n)$. □

It is important for the sequel to note the function $t(n,n)$, which by definition equals $\min\{\, k \mid T(k,1) > n \,\}$. Thus

$$k \geq t(n,n) \;\rightarrow\; t_k(n) = 1.$$

This function grows at most by one when n is increased by one; we assume that the maintenance of $t(n,n)$ under the operation of increasing n by one (starting at $n = 0$) can be carried out in constant (possibly amortized) time. This holds for the functions we meet in the applications given later.

A well known example is the Ackermann hierarchy [12,2,8], where the role of $T(k,n)$ is played by the following function A:

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

The row inverse functions of the Ackermann hierarchy are known as $\alpha_k(n)$. The "diagonal" function $\alpha(m,n)$ has been used by Tarjan to describe the running time of a union-find algorithm [12] (in fact his definition of the function is a bit different, but the difference in value is bounded by one).

We consider only families of algorithms which are *uniform*. This means that a single, finite program can run any algorithm of the family, given k as input. In the proof below, we will actually run some members of the family concurrently; this is the *multitasking* technique, which we proceed to describe in detail. The experienced programmer will find it easy to implement on any standard model of computation.

A task is an execution sequence of a program. It is standard procedure in programming a sequential machine to interrupt the execution of a program so that it can be continued later by saving a record of its *state*— usually the program counter and a small number of registers. In this way the execution of several tasks can be interleaved. We assume that the operations of saving a *state record* (SR) and loading one for use take constant time, and that the size of the state record is a fixed constant (possibly depending on the program). All tasks will be broken into *steps* of bounded duration, such that it will be convenient to

suspend a task between consecutive steps. In principle, we can suspend a task anytime, so this is no restriction on the program.

Each of the members of the family that we want to run concurrently will constitute a task. For managing the tasks, we make use of a *list* of state records (a linked list is the natural implementation of this structure). A "rover" pointer into the list indicates the next task to be elected for execution. As each of the algorithms is designed to process requests on-line, we will hold a queue of the incoming requests and each task will have its pointer into the queue. We will only put update requests on this queue, so its size is $O(n)$. In a *scheduler round*, certain suspended tasks are invoked and allowed to proceed for a pre-determined number of steps. More specifically, the scheduler begins by loading the SR indicated by the rover pointer and executing the task until enough steps have been performed or the task finishes processing all its input (reaching the end of the queue). In the latter case it moves the rover to the next task on the list. The state of each task which loses control is stored back in its record on the list.

3. The Main Result

Theorem 1. *Let (t_k) be a suitable sequence of functions and $(\mathcal{A}_k)$ a tradeoff family of algorithms for (t_k). Then we can construct an algorithm for the same problem running in $O(1)$ amortized time per update and $O(t(m,n))$ amortized time per query.*

Proof. We begin with discussing the update procedure, in which we attempt to keep the (amortized) operation time constant. Since the bound guaranteed is $O(t_k(n))$, this requires increasing the value of k we use as the number of updates n grows larger. More specifically, we will use $\mathcal{A}_k$ for $k = t(n,n)$. This choice prevails up to $n = T(k+1,1)$ where $t(n,n)$ becomes $k+1$. We conclude that for each k, we will start using $\mathcal{A}_k$ once n reaches $T(k,1)$. We will maintain the current value of $t(n,n)$ in a variable named K. We denote the data structure of $\mathcal{A}_k$ by $\mathcal{D}_k$.

Running $\mathcal{A}_K$ alone will present a problem when it is time to increase K, because we will have no data in $\mathcal{D}_{K+1}$. Therefore, for each value of K, $\mathcal{A}_{K+1}$ will be also run as a task concurrent to $\mathcal{A}_K$ (which remains *the main task*). When K increases, we want $\mathcal{A}_{K+1}$ to have processed all past update requests, so that $\mathcal{D}_{K+1}$ is up-to-date. To this end, we feed up to three update requests to $\mathcal{A}_{K+1}$ each time that $\mathcal{A}_K$ handles a single update. The resulting procedure is shown in Figure 3.1.

Note that there always is at least one update for $\mathcal{A}_{K+1}$ to process, this is the request just input. But if there is more in the queue, it will process up to three requests. Started at $n = T(K,1)$, it initially has $T(K,1)$ requests to go. At the rate of three requests at a time, it will catch up with $\mathcal{A}_K$ at $n = {}^3\!/_2 T(K,1)$, which by Claim 1 is less than $T(K+1,1)$. This shows that once it becomes the main task, it is ready to answer new queries. Note also that we do not kill the former task, $\mathcal{A}_{K-1}$. We simply don't invoke it at the moment, but it may read its input queue at a later time.

When we start a new task, we add its state record in front of the SR list. Thus the two recent tasks, $\mathcal{A}_K$ and $\mathcal{A}_{K+1}$, are the first tasks in the list. The *rover pointer* is reset,

```
procedure update(request)
put request on the input queue
n ← n + 1
if t(n, n) > K then
K ← K + 1
invoke A_{K+1} as an additional task
end if
run A_K on the request
advance the input pointer of A_K
repeat 3 times
if there is input for A_{K+1} then
    run A_{K+1} on the next request in its input queue
    advance the input pointer of A_{K+1}
end if
end repeat
```

Figure 3.1

following each update operation, to the SR of $\mathcal{A}_{K-1}$, the third in the list and the first to have awaiting input.

Queries can always be answered by $\mathcal{A}_K$, but when the number of queries becomes large relative to n, we wish to decrease their processing time. Faster queries are possible using lower values of k, so to this end, we will allot time to older tasks (with smaller k) in order to allow them to process queued updates. Recall that tasks are called to execution in LIFO order, following the list. We can thus maintain a variable k_0 to identify the smallest k such that $\mathcal{D}_k$ is up-to-date. Following each update, it will be reset to K.

When a query is made, a scheduler round is started where the variable k_0 indicates the number of steps to be performed. It is possible that, during this round, a task completes processing all its input; then k_0 is changed to the index of that task. As a result of the LIFO order, this change can only decrease k_0 by one. Therefore the count of steps performed must eventually meet k_0 at which point this round will end. Next, $\mathcal{A}_{k_0}$ is used to process the query (we use the updated value of k_0); this takes $O(k_0)$ time units, as did the scheduler round, for a total of $O(k_0)$ time.

Update time: The running time of the *update* procedure can be divided into time invested in managing the structure, and time required by $\mathcal{A}_K$ and $\mathcal{A}_{K+1}$ for processing. The "management" includes maintaining $t(n, n)$ and invoking at most one new task, which we assume to require constant time per operation. We now consider the time spent in the active tasks. Note that the task $\mathcal{A}_k$ is called up by the update procedure exactly for processing the first $T(k+1, 1) - 1$ update requests. The amortized time per update of $\mathcal{A}_k$ is $O(t_k(n))$ which for $n < T(k+1, 1)$ is $O(1)$. This means the total time it took is bounded by a constant (say c) times the number of updates performed. This constant is the same for all tasks. Divide the numbers $1, \ldots, n$ (counting the updates) into $K = t(n, n)$ epochs using the dividers $T(1, 1), \ldots, T(K-1, 1)$. That is, the first epoch extends from 1

to $T(1,1)-1$, the second from $T(2,1)$ to $T(3,1)-1$ and so forth. Let ℓ_k be the length of the kth epoch. The update procedure activates $\mathcal{A}_k$ exactly during epochs $k-1$ and k, during which it performs at most $3\ell_{k-1}+\ell_k$ updates (where $\ell_0=\ell_{K+1}=0$). Summing up, we obtain that the total running time is bounded by

$$\sum_{k=1}^{K} 3c\,\ell_{k-1}+c\,\ell_k < 4c\sum_{k=1}^{K}\ell_k = 4c\,n.$$

This proves that the *amortized* time of our update procedure is constant. Actually, the time for each update is bounded by about four times the bound of the single update by a single algorithm of the family, plus the time for maintaining $t(n,n)$ and task handling; if all of these take constant time (not just in amortized sense), so does our update procedure.

Query time: Let $s=t(m,n)$. We show the amortized query time is $O(s)$. We have shown that the query time is dominated by the value of k_0 at its end. Thus, each query which ended with $k_0 \le s$ clearly used $O(s)$ time. Assume that at the end of processing some query, k_0 is greater than s. It follows that $\mathcal{D}_s$ is not up-to-date at this moment. Therefore, at the activation of the query, there were update requests queued for processing by $\mathcal{A}_s, \mathcal{A}_{s+1}, \ldots$ and the k_0 steps allotted by the scheduler round were used up by this processing. Regarding the whole history of the structure, we obtain that the total running time of queries of the latter kind is bounded by the processing time of all the update requests by $\mathcal{A}_s, \mathcal{A}_{s+1}, \ldots, \mathcal{A}_{K-1}$. By the properties of the algorithm family this is $O(\sum_{i=s}^{K-1} nt_i(n))$. Using property (2) of the functions (t_k), this is $O(nt_s(n))$. By definition of $s=t(m,n)$, it is $O(n\lceil m/n\rceil)=O(n+m)$. Joining the two kinds of queries, their total processing time is $O(n+m+ms)$. If $m \ge n$, this is clearly $O(ms)$ so the amortized operation time is $O(s)$.

If $m<n$, $t(m,n)=t(n,n)$ by definition. Therefore $s=t(n,n)=K$; and $O(K)$ is an upper bound on the processing time of each single query (being the time of the slowest task). □

3.1 Preprocessing Problems

For preprocessing problems, the use of multitasking may be reduced. In the preprocessing phase, we process all the update operations in the fastest way ($O(1)$ time per update). To this end we use $\mathcal{A}_K$ for $K=t(n,n)$, providing a data structure $\mathcal{D}_K$.

In the second phase, we process queries using the data structure built, but at the same time the updates are processed again using $\mathcal{A}_{K-1}$. When $\mathcal{A}_{K-1}$ completes processing all the input, the data structure it produced can replace the one of $\mathcal{A}_K$. Following queries will be directed to $\mathcal{D}_{K-1}$ while $\mathcal{D}_{K-2}$ will start being built. To sum up, at any moment we have a complete data structure for querying and a single additional task that is building a new structure.

Since $\mathcal{A}_K$ performs n updates in $O(n)$ time, this is the cost of our preprocessing phase. For the query phase, let $s=t(m,n)$. As above, we want to bound the time invested in queries where the last structure built, $\mathcal{D}_k$, has $k>s$ (all other queries take $O(s)$ time). Since each such query spends an amount of time equal to the length of the query proper in building the next structure, the time we seek is bounded (up to a

constant factor) by the time it takes to build $\mathcal{D}_{K-1}, \ldots, \mathcal{D}_s$. As above, this is bounded by $O(\sum_{i=s}^{K-1} nt_i(n)) = O(n+m)$, and by the same considerations we obtain amortized time of $O(s)$ per query.

3.2 Cost in Space

All the algorithms we constructed require $O(n)$ space for keeping the list of inputs which is fed in turn to each of the algorithms invoked, plus some space for book-keeping functions, and finally the data structures themselves.

In the preprocessing setting, the space requirements for data structures will be at most twice those of a single member of the family, as only two members are operating at the same time.

The algorithm constructed for the on-line case may require the sum of the amounts of space required by $\mathcal{A}_1, \ldots, \mathcal{A}_{K+1}$. This potentially large requirement may be cut down for specific problems. In particular, there are algorithm families where the differences between members are such that most of the data structure does not have to be replicated; an example is the algorithm family for *union-find* given in the next section. This is due to the fact that a member with longer update time spends it on rearranging the structure in a way which is more efficient for queries but leaves it amenable to further updates by its faster fellows without decreasing their efficiency.

3.3 Applications

In [2], Ben-Amram and Galil display a family of algorithms for solving the on-line *retrieval by position* problem on a shifting machine. This is a machine which has a finite number of registers but can store unbounded integers in them and use these in computations including a *shift* instruction. The problem calls for storing arbitrary data and retrieving the ith datum stored on a query for i. The algorithms described form a tradeoff family for the inverse Ackermann functions (α_k). The main theorem applies and shows how to construct a solution of best running time for a sequence of updates and queries, which also features updates in bounded time. The value of K at any point of the algorithm is easily determined from the data structure.

In [5], preprocessing algorithms are given which display the same tradeoff between update and query time. For the LCA (lowest common ancestor in a static tree, where n is the number of nodes), preprocessing can be accomplished in $O(\alpha_k(n))$ parallel time to support a query in $O(k)$ time: the functional hierarchy is again Ackermann. The interpretation of our theorem for such a case yields a solution whose preprocessing time is $O(1)$, and if the n processors receive queries simultaneously, they will each process a query in $O(\alpha(m,n))$ time.

In the same paper, an algorithm family is given for the *level ancestor* problem (given a node and a number i, determine the ith ancestor of the node). In this family, the update time of $\mathcal{A}_k$ is given by $\beta_k(n) = \log^{(k)}(n)$. This demonstrates a non-Ackermann hierarchy. The query time resulting from the application of our method is

$$\beta(m,n) = \min\{\, i \mid \log^{(i)}(n) < [m/n] \,\}.$$

The Ackermann hierarchy also appears in algorithms from Alon and Schieber [1]. They present a family of algorithms for answering *on-line product queries* after preprocessing. They pair preprocessing in $O(n\alpha_k(n))$ time with a query time of $O(k)$. Thus our theorem applies and yields a preprocessing time of $O(n)$ with query time $O(\alpha(m,n))$.

4. Recovering the Tradeoff

The union-find algorithm analyzed by Tarjan [12,13] is well known for being a simple algorithm having remarkable efficiency. In fact, this algorithm achieves the time of $O(m\alpha(m,n))$ for m operations, where n is the number of sets created and α is an inverse Ackermann function. We have found such functions to describe the "balance point" in a sequence of solutions having a tradeoff structure. Therefore in this case too it is natural to expect a sequence of solutions which displays the underlying tradeoff. Union-find algorithms that trade union time for find time have already been presented by La-Poutré [9]. However, his algorithms are not related at all to the path compression algorithm. In contrast, we attempt to give the most natural modification of the path compression algorithm that yields a tradeoff family. The complexity analysis of our algorithms is also very close to Tarjan's original analysis. In fact, from our proofs an alternative presentation of Tarjan's result can be derived, which differs only slightly from his presentation but may have some classroom value.

We observe that there is a common feature to the algorithm families that we have studied. It is a leveled data structure, where in a query, one must move from level to level at a constant cost. Thus each additional level increases the cost of a query, but contributes to a faster update. The function t_k describes the average cost of maintaining a k-level structure.

We suggest that whenever similar time bounds can be achieved, the algorithm can be put in the above framework. This was explicit in the previous examples; but in union-find, the simplicity of the algorithm does not reveal it. However, it does appear in Tarjan's analysis of the running time. For this analysis, a *level number* was assigned to each node, and the query time was related to the number of levels encountered on a find path. This picture resembles the one described above, and we deduce that by keeping the number of levels bounded by k, we should get the member $\mathcal{U}_k$ of a family of union-find algorithms.

4.1 Algorithm $\mathcal{U}_k$

The algorithm is a variant of the solution in [12]. Sets are represented as rooted trees, where each set element is a tree node and the root node is used to identify the set. Thus to execute a *find* we follow pointers from the element specified to the root of its tree. This is augmented with *path compression*, meaning that all the pointers in this path are redirected to the root, thus cutting down the time of future finds. A *union* is implemented by linking the root of one tree to that of the other; as in [12], we use the strategy of *union by rank*. The rank is a number assigned to each node, which can be used as a bound on tree height; its exact computation is specified in the sequel. *Union by rank* means that in each union operation, the tree root who gets linked to the other is the one having smaller rank (either one in case of equality). We augment the data structure by maintaining

$\alpha_k(r)$ for every value of r which appears as a rank of some root. We omit elaborating on techniques for maintaining these values, stating only that this does not lengthen the union time (such techniques are elaborated in [2]). We further point out that this algorithm can be implemented as a *pure pointer algorithms* [11,4] by representing those values as lists of pointers. This is feasible thanks to the restricted way in which these values are used in our algorithm, as explained in [11].

We modify the *union* operation by adding a rule of *subtree compression*. Consider a *union* in which node x becomes a child of root y. There are two cases:

(i) $\text{rank}(x) = \text{rank}(y)$. Then the rank of y is incremented. Afterwards, we check whether

$$\alpha_k(\text{rank}(y)) > \alpha_k(\text{rank}(x)). \tag{1}$$

If this is the case, we compress the whole tree rooted at y, i.e. connect all the proper descendants of y directly to y.

(ii) the ranks of x and y are different to start with. Then we check (1) as well, but if it holds we move to y just the descendants of x.

Theorem 2. *For a sequence of operations on n elements, including $m \geq n$ finds, Algorithm $\mathcal{U}_k$ uses $O(k)$ amortized time per find and $O(\alpha_k(n))$ amortized time per union.*

The formulation of this theorem differs from the notation of the previous sections in the use of n. However, the number of union operations is at most $n-1$ and can be assumed to exceed $n/2$, for otherwise some sets stay "out of the game", and do not affect the time bounds of our algorithm. Thus the result would not be different if the number of *union*s were used.

The proof follows the outline of Tarjan's analysis in [12], employing his "method of multiple partitions."

For every node x, $p(x)$ denotes the parent of x in the current structure ($p(x) = x$ signifies a root). We further define $f(x)$ to be the root of the current tree where x belongs (the *find* result).

We define the auxiliary function $B(i,j)$, similar to $A(i,j)$, by

$$\begin{aligned} B(0,j) &= j && \text{for } j \geq 0; \\ B(i,0) &= 0 && \text{for } i \geq 1; \\ B(i,j) &= A(i,j) && \text{for } i \geq 1,\ j \geq 1. \end{aligned}$$

For each level $i \geq 1$ we define a partition of the natural numbers into *blocks* by

$$\text{Block}(i,j) = [B(i,j) \ldots B(i,j+1) - 1]$$

(see Figure 4.1). Note that for $i > 0$, each $\text{Block}(i,j)$ is a union of some blocks of level $i-1$. Let b_{ij} be the number of level $i-1$ blocks contained in $\text{Block}(i,j)$.

The following lemma summarizes some simple observations on the algorithm.

Lemma 1. *If x is any node,* rank(x) $\leq$ rank($p(x)$), *with the inequality strict if $p(x) \neq x$. The value of* rank(x) *increases as time passes until x is linked as a child of some other node; subsequently* rank(x) *does not change, but p(x) and f(x) do. The value of $rank(f(x))$ is a nondecreasing function of time.*

We define the *layer* of a node x to be the smallest i for which the ranks of x and $f(x)$ lie both in the same i-level block. From the last lemma and the form of the block structure we see that layer(x) is zero if and only if x is a root and becomes positive once $p(x)$ is assigned a value other than x; subsequently, it can only increase, as rank($f(x)$) gets farther from rank(x).

Lemma 2. *Following any operation in $\mathcal{U}_k$, each node whose layer is greater than k is a leaf.*

Proof. The statement is trivially true when the algorithm starts. A *find* operation does not affect layers as it changes neither ranks of nodes nor the identity of roots. Further, a node that was a leaf before the operation remains so afterwards. Hence the truth of the statement before the *find* implies its fulfillment afterwards. A *union* operation may affect the layer of a node by changing the identity of its root or the node's rank. In case (i) of the union operation, the rank of y is changed. This may cause the layer of descendants of y to surpass k only if α_k(rank(y)) increases. But in this case, we compress the tree, making each affected node a leaf. Case (ii) is similar. □

4.2 Proof of Theorem 2

Update time: apart from subtree compression, each *union* takes constant time. Subtree compression affects each node x exactly at those *union*s where $\alpha_k(rank(f(x)))$ increases. For each x, $rank(f(x))$ starts at zero and increases up to a value which is less than n. Thus $\alpha_k(rank(f(x)))$ increases from zero up to a value bounded by $\alpha_k(n)$. Thus, the total cost of these operations over n *union*s is bounded by $\alpha_k(n)$ per node, or $n\alpha_k(n)$ altogether.

Query time: The running time of a sequence of operations is evaluated using a credit-debit accounting scheme. In this method, each *find* is given c credits to spend. One credit will pay for a constant amount of computing. If we run out of credits before completing an operation, we can get more credits by creating debits. The total running time is majorized by mc plus the number of debits in existence at the end of the run.

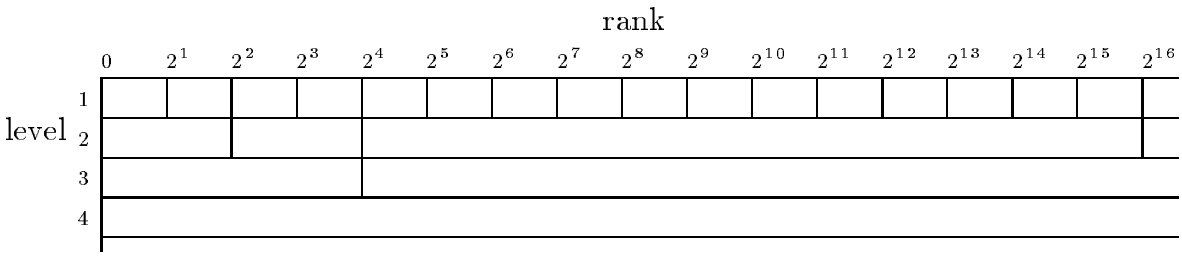

Figure 4.1

A graphical representation of the partitions defined by Block(i, j). Level zero is omitted and a logarithmic scale is used. (from [12]).

In a *find* operation, we need one credit for each node on the find path (this is the path from x up to $f(x)$). We assign one of the credits allocated to this operation to the *last* node on the path belonging to each layer. Thus, the number of credits we need to allocate is bounded by the maximum number of different layers which may be encountered on the path. The rest of the nodes are covered with debits. Since the root is the only node of layer zero, and only ranks of roots ever change, it follows that every node x starts receiving debits only after its rank has ceased to change. We now relate the number of debits received to this rank.

Lemma 3. *For each i, the number of debits received by x while in layer i is bounded by $b_{i,j} - 1$ where j is the unique integer satisfying* $\mathrm{rank}(x) \in \mathrm{Block}(i, j)$.

Proof. Consider a *find* operation in which x receives a debit. This means that x is not the last node on the path of layer i. Since the layer function is monotone along the path, it follows that $p(x)$ has also layer i. Also $p(x) \neq f(x)$, because the root and only the root has layer zero. Thus $\mathrm{rank}(p(x)) < \mathrm{rank}(f(x))$. Moreover, the root is not in the same block with $p(x)$ in level $i-1$, for otherwise $\mathrm{layer}(p(x))$ would be less than i. We deduce that following the path compression, the new value of $\mathrm{rank}(p(x))$ lies at least one level $i-1$ block above the old one.

This can happen at most $b_{ij} - 1$ times, because once $\mathrm{rank}(f(x))$ transcends $\mathrm{Block}(i, j)$ the layer of x is no longer i. Thus the number of *finds* in question is bounded by $b_{ij} - 1$. □

From Lemma 2 it follows that on any find path, at most one node (the leaf) may be of layer larger than k. Consequently the number of different layers on the path is bounded by $k+2$, and the credit c needed for each *find* is $O(k)$. Moreover, only layers 1 through k contribute to the collection of debits. Thus using the last lemma, the number of debits is bounded by

$$\sum_{i=1}^{k} \sum_{j \geq 0} n_{ij}(b_{ij} - 1), \tag{2}$$

where n_{ij} is the number of nodes which, by the end of the run, have a rank in $\mathrm{Block}(i, j)$.

Obviously $n_{i0} \leq n$. By definitions, $b_{ij} \leq A(i, j)$ and in particular, $b_{i0} = 2$, hence $n_{i0}(b_{i0} - 1) \leq n$. Tarjan shows [12, p.28] that for each $j \geq 1$, $n_{ij} \leq n/2^{A(i,j)-1}$. Thus the above sum is bounded by

$$\begin{aligned} \sum_{i=1}^{k} \left(n + \sum_{j \geq 1} \frac{n}{2^{A(i,j)-1}} (A(i,j) - 1) \right) &< kn + n \sum_{i=1}^{k} \sum_{l \geq A(i,1)} \frac{l-1}{2^{l-1}} \\ &< kn + n \sum_{i=1}^{k} \frac{A(i,1)}{2^{A(i,1)-2}} < (k+6)n. \end{aligned}$$

Hence the amortized time is $O(k + \frac{(k+6)n}{m})$. Having assumed $m \geq n$, we get $O(k)$. □

We remark that, in this algorithm, we must have an assumption such as $m \geq n$ for obtaining $O(k)$-time find, since a sequence of unions can be easily built to make a small

number of *find*s take a lot of time. In fact, this is unavoidable in general, provided that the update time is required to be $O(\alpha_k(n))$; for a proof see the worst-case lower bounds [6,7,3].

Our analysis differs from Tarjan's in using layer(x) where he used level(x), the smallest i for which rank(x) and $rank(p(x))$ lie in the same i-level block. However, Tarjan's proof could be rephrased in terms of *layer*. We suggest that it may be advantageous to do it this way, since the layer function is monotone along the path, while *level* is not; this makes the choice of *layer* somewhat more intuitive.

Acknowledgment

Helpful remarks from Noga Alon are gratefully acknowledged.

REFERENCES

[1] N. Alon and B. Schieber, "Optimal preprocessing for answering on-line product queries," preprint, 1987.

[2] A. M. Ben-Amram and Z. Galil, "On the power of the shift instruction," to appear in *Infromation and Computation.*

[3] A. M. Ben-Amram and Z. Galil, "Lower bounds for data structure problems on RAMs," *Proc. Thirty-Second Annual IEEE Symp. on Foundations of Computer Science*, San-Juan, PR 1991. See also Ben-Amram's PhD thesis (Tel-Aviv University, 1994).

[4] A. M. Ben-Amram and Z. Galil, "On pointers versus addresses," *J. of the ACM* 39:3 (1992) 617–649.

[5] O. Berkman and U. Vishkin, "Recursive *-tree parallel data-structure," *Proc. Thirtieth Annual IEEE Symp. on Foundations of Computer Science*, Singer Island, FL 1989.

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

[7] M. L. Fredman and M. E. Saks, "On the cell probe complexity of dynamic data structures," *Proc. Twenty-First Annual ACM Symp. on Theory of Computing*, Seattle, WA 1989.

[8] S. Hart and M. Sharir, "Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes," *Combinatorica* 6 (1986), 151–177.

[9] J. A. La Poutré, "New techniques for the union-find problem," *Proc. First Annual ACM-SIAM Symp. on Discrete Algorithms*, 54–63.

[10] R. E. Tarjan and J. Van Leeuwen, "Worst case analysis of set union algorithms," *J. of the ACM* 31 (1984), 245–281.

[11] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. Comput. System Sci.* 18 (1979), 110–127.

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

[13] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. of the ACM* 22 (1975), 215–225.