Dynamic Microsets for RAMs

J.A. La Poutré

RUU-CS-93-22
July 1993

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Dynamic Microsets for RAMs

J.A. La Poutré

Technical Report RUU-CS-93-22
July 1993

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Dynamic Microsets for RAMs*

J.A. La Poutré

Department of Computer Science, Utrecht University,

P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract

We generalize the concept of Gabow and Tarjan's microsets [3] to dynamic microsets. The dynamic microsets allow the use of direct addressing in a RAM for incremental problems that can be partitioned into subproblems which are merged from time to time. A sequence of operations on dynamic microsets can be performed in $O(n+m.\alpha(m,n))$ time, where n is the size of the problem and m is the number of queries.

1 Introduction

In [3], Gabow and Tarjan introduce the notion of microsets for obtaining a fast (linear) algorithm for the Union-Find problem on a RAM (Random Access Machine) for the special case that the structure of the problem (viz., the structure of the sequence of Union operations) is known in advance. For subproblems on abstract subuniverses of sufficiently small size (e.g., of $\theta(\frac{\log n}{\log\log n})$ elements) and defined by the structure of the problem, the needed information is precomputed and stored in tables (where the RAM model is needed for fast access in the table). Then each "real" occurring small set has references to corresponding abstract sets in the corresponding subproblems, and operations can be performed by making use of this precomputed information: the precomputed solutions for the generic subproblems on abstract subuniverses are "translated" to the actual set. The latter means that a (fixed) correspondence is implemented between the real elements and the abstract elements. However, in case the (structure of the) sequence of Unions is not known in advance, the above technique does not apply, since this correspondence can change arbitrarily. This is sustained by the fact that the general Union-Find problem on a RAM has a non-linear lower bound of $\Omega(n+m.\alpha(m,n))$ [2]. For various other dynamic problems

*The research of the author has been made possible by a fellowship of the Royal Netherlands Academy of Sciences and Arts (KNAW). This research was partially supported by the ESPRIT Basic Research Actions of the EC under contract no. 7141 (project ALCOM II).

(especially graph problems), it seems that we really have to make use of direct addressing in a RAM to obtain an $\alpha(m,n)$ bound [4, 9]. (For the nearest common ancestor problem with linking, there exists a lower bound of $\Omega(\log\log n)$ for the pointer machine model [5], but there is a solution that runs in $O(n+m.\alpha(m,n))$ time on a RAM [4].) This makes the problem of maintaining the above correspondence information important, as we will further illustrate later on.

We therefore study the problem of maintaining disjoint sets of nodes that is given as follows. Let U be a universe of n elements. Suppose U is partitioned into a collection of (named) singleton sets, suppose that every node x has a so-called *local name*, initially being the number 1, and suppose we want to be able to perform the following operations:

- Merge(A,B): for two sets named A and B, rename the nodes in set B by adding $|A|$ (i.e., the size of set A) to their local names, and join the two sets A and B into one new set, say S, and return the name of the result S.
- Find(x): return the name of the set in which element x is contained, and return the local name of x.
- Search(A,k): return the element x that is contained in the set named A, and that has local name k.

The occurring set names must satisfy the condition that, at every moment, the names of the existing sets are distinct. Moreover, no occurring set ever has more than b elements for some number b. We call this problem the *dynamic microset problem*. Note that we have a query time of $O(\log b)$ for the operation *Search*, if we solve the problem by means of balanced search trees.

We present a data structure for this problem that has an optimal time complexity of $O(n + m.\alpha(m,n))$ for n elements and m operations for any b with $b \leq min\{\frac{1}{14}\log n, \frac{1}{6}wordsize\}$, where *wordsize* is the size of a machine word in the RAM model (which is usually taken as $\log n$). Moreover, the f^{th} operation Find or Search can be performed in $O(\alpha(f,n))$ worst-case time. ($\alpha(m,n)$ is the inverse Ackermann function [6, 12].)

The problem has applications in cases in which we want to use the specific property of address calculation of a RAM. Applications occur in maintaining the 3-vertex-connectivity relation for dynamic graphs [8] or the problem of incremental planarity testing [9]. The use is as follows. We precompute answers for types of (abstract) subproblems of small size (e.g. $O(\log\log n)$) which are sufficiently few (each type is represented by a number). The occurring ("real") collection of small subproblems usually corresponds to a partition of the elements into sets. Then, sufficiently small sets (and, hence, subproblems) are treated as microsets, where the specific type information related to these can just be stored with their names. Since the correspondence between the local names inside the abstract microsets (the types) and the

actual sets (consisting of the "real" elements) must be maintained, this corresponds to the above problem: Viz., for actual elements, the name of a set must be obtained, and when by this name, corresponding additional precomputed information (stored in tables) is obtained in terms of local names of the abstract elements, the corresponding "real" elements must be retrieved again. Since these microsets can be arbitrarily joined from time to time, this dynamic relation between local names and real elements must be maintained.

An example of the ideas occurring in an application of this data structure is e.g. of the following type, which corresponds to [8]. (In [9], another kind of partition is used too, which does not correspond to connected components themselves.) Initially, the problems start from a situation where the nodes (elements) are completely separated and thus can be seen as singleton sets. From time to time, such sets of nodes are joined because of dynamic operations. The biggest occurring sets, say, of size $\Omega(\log n)$ are partitioned into clusters of size $\theta(\log n)$ (generally speaking), and then clusters are contracted into new nodes that are subject to the further computations. Then, joinings and other processing can be done by a simple algorithm that can even spend time linear to the smallest set to be joined: this just gives an overall time complexity of $O(\frac{n}{\log n}.\log \frac{n}{\log n}) = O(n)$. Depending on the complexity of the considered problem, this idea can recursively be applied for sets with size between $\Omega(\log \log n)$ and $O(\log n)$, and so on ($O(1)$ times), until sets of sufficiently small size are obtained for which all the query and operation information can be encoded in machine words and can be stored into tables of size $O(n)$. For these sets, the dynamic microset structure is used, and the problem can be solved for these small sets by means of table lookup and the dynamic microset data structure like described above. For further description of the ideas in this case, we refer to [8].

2 Preliminaries

The Ackermann function A as we use it is defined as follows. For $i, x \geq 0$ function A is given by

$$\begin{array}{lcll} A(0,x) & = & 2x & \text{for } x \geq 0 \\ A(i,0) & = & 1 & \text{for } i \geq 1 \\ A(i,x) & = & A(i-1, A(i,x-1)) & \text{for } i \geq 1,\ x \geq 1. \end{array} \tag{1}$$

The row inverse a of the Ackermann function is defined by

$$a(i,n) = min\{j \geq 0 | A(i,j) \geq n\} \tag{2}$$

for $i, n \geq 0$.

The functional inverse α of the Ackermann function is defined by

$$\alpha(m,n) = min\{i \geq 1 | A(i, 4\lceil m/n \rceil) \geq n\} \tag{3}$$

for $m \geq 0$, $n \geq 1$. Here we take $\lceil 0 \rceil = 1$ in contrast to its usual definition.

Note that $\alpha(0,n) = \alpha(n,n)$. The last two definitions differ slightly from those appearing in [10, 11, 12]. However, it is easily shown that the differences are bounded by some additive constants (except for the functions $a(0,n)$ and $a(1,n)$).

By means of the row inverse of the Ackermann function we can express the functional inverse α as follows.

Lemma 2.1 $\alpha(m,n) = min\{i \geq 1 | a(i,n) \leq 4.\lceil m/n \rceil\}$.

For further treatment of the Ackermann function, we refer to [6, 7].

3 Dynamic Microsets

We describe the implementation of the Dynamic Microset structure. We have a Union-Find structure UF to maintain the microsets.

We use the UF(i) structures as given in [6]. The reason for this is that this structure allows us to perform Finds in $O(i)$ time, where i can be chosen to be $O(\alpha(m,n))$. This is convenient for many applications of microsets, since these typically have a big worst-case time for joining structures (within a fast amortized time), whereas the queries, usually using the Finds, can be performed much faster. The reason for the former is that together with microsets, a natural way to handle sets that are bigger than the microset size is the "recompute the smallest one" strategy when sets are merged in the structure. We refer to [8, 9] for examples, where queries take $O(\alpha(m,n))$ worst-case time.

For readers not familiar with the UF-structure given in [6], we give the outlines of it that are relevant for our data structure. As usual, sets are represented by rooted trees, where now the elements are stored in the leaves only. The trees are layered, where a layer is a collection of all the nodes in a tree that have the same distance to the root. All leaves are in one layer, and the layers (except for the root) in a UF(i) structure are numbered consecutively from i for the layer containing the leaves down to some positive integer. The root of a tree is the set name. To each layer j in each tree, a number $lowindex(j)$ is associated (which is less than $\log n$, where n is the total number of elements), together with a pointer $clus(j)$ to a node in layer $j-1$, called cluster node for layer j (if it exists). This information occurs in a list related to the corresponding set name. The Find operation follows root path (hence, in $O(i)$ time), whereas the Union operation joins two trees by putting all

the nodes of some layer j of one of the trees below node $clus(j)$ in layer $j-1$ of the other tree, or by putting all the nodes of layer j of both trees below the root (if $j = 1$) or below a newly created node (if $j > 1$) in a new layer $j-1$ which is put directly below the root. This is determined by the Ackermann values for parameters j and $lowindex(j)$ (which need not to be computed for any practical situation). The structure is recursive in the way that a UF(i) structure of which layer i is deleted satisfies the constraints of a UF($i-1$) structure (except for trees consisting of the root only). For further details we refer to [6].

The UF(i) structures of [6] are used with the following modifications. We make UF(i) trees ordered. I.e., for each node x in a UF(i) tree, we relate a number to each son of x, such that the sons of x (forming the list called $sons(x)$) are consecutively numbered starting from 1 (and, hence, up to $|sons(x)|$). As usual, this gives an order on all the nodes in a certain layer of a tree (viz., given by the lexicographical order of the local names of the path from the root to a considered node x). The order of the leaves of a tree, which are the elements of a set, is the order induced by their local names in the set. Each set name s has a field $tt(s)$ containing a number, called the *tree type*. The tree type is a number that is uniquely related to a class of ordered UF trees and contains all necessary information that is used in the UF algorithms, on which we shall elaborate later. (It reflects the equivalence class of all ordered UF-trees that are isomorphic with some UF-tree.) We further augment the UF(i) trees as follows. Every internal node x in a UF(i) tree T has an array a_x of size $size(a_x) = 2^{\lceil \log |sons(x)| \rceil}$. (Hence, $\frac{1}{2}.size(a_x) < |sons(x)| \leq size(a_x)$.) Then $a_x(k)$ contains a pointer to the son of node x with number k. (The array a_x replaces the list $sons(x)$.) We shall further consider this representation with arrays later on. Finally, we only deal with trees of at most g nodes, for some integer g that will be chosen later.

When two microsets s and t are joined, this is done by performing $UNION(s,t)$ in the UF structure, followed by a call $tt(s) := mergetrees(tt(s), tt(t))$ to set the tree type of the resulting tree $tt(s)$. This is done as follows with respect to the UF structure. For each layer j in a UF(i) tree with root s, instead of using one pointer $clus(s,j)$ to a cluster node for this layer (that, hence, is a node in layer $j-1$), we have two pointers $clus_l(s,j)$ and $clus_r(s,j)$ that point to the leftmost and the rightmost node in layer $j-1$. Then the Union algorithms are adapted as follows. If (all the) nodes in a layer are generated or handled in some way, then this is done in the order in which they occur(red) in the layer. When additional nodes are put in the list $sons(x)$ of node x, this is done for the array a_x as follows. Nodes are added in array a_x until it is filled up. If a_x is full and a node must be added, then it is replaced by a new array of double size, with the same contents on the "old" positions as the old one. For each node y of an old UF-tree that is removed, the corresponding array is given free. (In the full version of [6], this refers to Figure 7, lines 9 and 16 of Figure 6, and Figure 4.)

Note that the Union operation is deterministic, so that the *mergetrees* operation is determined by the tree types alone. We will give procedure *mergetrees* in the sequel.

We have the following "local" operation to deal with the Union operation:

- $mergetrees(tt_1, tt_2)$: return the tree type of the resulting tree if two trees of type tt_1 and tt_2 are concatenated (in this order).

For operation Search we introduce the following local operation:

- $treepath(tt, k)$: given the tree type tt, output the sequence of the numbers related to the nodes on the root path of the node with local name k in a tree of type tt.

Then Search(A, k) is computed by obtaining the tree type $tt(A)$, performing $treepath(tt(A), k)$, which returns the number sequence *seq*, and then traversing the UF tree downwards from the root by repeatedly taking the son with number specified by *seq*, until we reach a leaf. This leaf is the k^{th} element in set A.

Finally, for the Find(x) operation, which also outputs the local name of x in its set, we have the following local operation on tree types. We take that $i \leq \log g$ for the UF(i) structure used (which is a straightforward property for the meaningful use of UF(i)).

- $local(tt, seq)$: given the tree type tt and the sequence *seq* of the (at most $\log g$) numbers related to the nodes on the root path of a leaf x, return the local name of x.

We implement the local operations. To do this, we have the tables $MERGETREES$, $TREEPATH$, and $LOCAL$, by which the operations can easily be performed, and that contain the answers for the corresponding operations given above (with the same parameters). The above operations can then be performed by simple table lookup.

The tables can be computed as follows. For tables, only trees with at most g nodes are considered (g is a number satisfying some constraints to be given later). Generate all possible ordered rooted trees with leaves at the same depth being at most $\log g$ and with at most g nodes, where the sons of each node are numbered consecutively starting from 1 (related to each son as its order number), and where the layers (except for the root) are numbered consecutively with numbers between 1 and $\log g$, and where a value "*lowindex*" is assigned to each layer ($-1 \leq lowindex < \log g$). To each such tree, relate a distinct number tt as tree type (consecutively, starting from 0), and put tt together with the tree in a table, the type table. Also,

compute the appropriate Ackermann values $A(i,x)$ [6] and store them in a table too. Then do the following, as far as the trees and computations are meaningful with respect to the Union-Find data structure (e.g., the types tt_1 and tt_2 for procedure $MERGETREES$ should correspond to the same UF(i)) and yield trees of at most g nodes.

- $MERGETREES$: for all existing tree types tt_1 and tt_2, compute the corresponding resulting tree by simulating the UNION(i) algorithm for the appropriate i (given by tt_1 and tt_2), if possible, and find the tree type tt' of this tree in the type table (if any).
- $TREEPATH$: for all existing values of tt and k, obtain the sequence of the numbers related to the nodes on the root path of the node with local name k (if any).
- $LOCAL$: for all existing values of tt and seq, obtain the local name k of the leaf for which the numbers of the nodes on the root path correspond to the sequence seq (if any).

Note that the number of the ordered trees (without the layer numbers and the *lowindex* numbers) is at most $O(2^{2g})$, which can be seen by using the well-known up-down coding of trees (a traversal on the tree starting at the root gives a binary sequence of length $2g-2$, where a 0 is generated if an edge is traversed downwards and a 1 is generated if an edge is traversed upwards). Thus, the total number of tree types is $O(2^{2g}.\log g.(\log g+1)^{\log g})$ (taking into account the layer numbers and the *lowindex* numbers), which is $O(2^{3g})$. Hence, writing $F(g) = 2^{2g}.\log g.(\log g+1)^{\log g}$, all these computations take $O(F^2(g).g.F(g).g + F(g).g.g + F(g).g^{\log g}.g) = O(2^{7g})$ time, using that the Union operation can be simulated in $O(g)$ time, that the tree type list can be checked in $O(F(g).g)$ time, and that local computations inside a tree can be done in $O(g)$ time. If $g \leq \frac{1}{7}\log n$, then $O(2^{7g}) = O(n)$, and the tables also take $O(n)$ space (the implementation of a tree in the type list takes $O(g)$ space.) Finally, the numbers representing the tree types (and the other numbers occurring above) can be stored in one machine word if $g \leq \frac{1}{3}.wordsize$.

By means of these operations our original data structure can be implemented. Now, the Merge and Find operations work in the same order of time as needed to perform these operations in the UF data structure. By means of operation $treepath(tt,k)$, the operation $Search(A,k)$ works in time $O(depth)$ as well, where $depth$ is the depth of the UF-tree for A.

For the "dynamic arrays" a_x, we do the following. We know that for n elements, the space needed for all the occupied array positions is at most $2n$, since there are at most $2n$ nodes in the UF structure. Therefore, we have a total size of at most $4n$ for all the needed storage together, and we use an array $A[1..8n]$ to simulate these arrays. To do this, for a node x, $ind(x)$ is an index in $[1..8n]$, and $s(x)$ is the size

of a_x. The values $a_x(k)$ for $1 \leq k \leq s(x)$ are stored as $A[ind(x) + k - 1]$, thus a_x occupies the part $A[ind(x)...ind(x) + s(x) - 1]$, and these parts of array A do not overlap for different x. Conversely, a separate field of $A[ind(x)]$ contains a pointer to node x (where these fields for those positions that are not equal to some $ind(y)$ just contain nil). Finally, we have an index $free$ that indicates a position such that $A[free..8n]$ is not occupied by any a_x.

To claim a new a_x of size $s(x)$, we just make $ind(x) := free$ and $free := ind(x) + s(x)$, where the old values of the old a_x are copied to the appropriate positions. If the new memory block doesn't fit into A ($free > 8n + 1$), then the entire structure is rebuilt such that all blocks occupy a consecutive beginning part of A.

The time complexity of this strategy is $O(n)$ time plus $O(1)$ amortized time per newly occupied position in an array. This is seen as follows. Obviously, claiming a new array takes $O(s(x))$ time, which corresponds to $O(1)$ time per "added" position, since each time a new array is claimed for a node x, the size $s(x)$ is doubled. On the other hand, rebuilding the entire structure takes $O(n)$ time, which also can be charged to each newly claimed array for $O(s(x))$ time (this is because after each rebuilding at most the first half of array A is filled, and rebuilding is not started until the second half of the array has been claimed completely). Hence, if this $O(1)$ time is charged to setting the contents of an array position, this does not increase the order of time complexity of the algorithms (apart from $O(n)$ time in total).

By applying UF(i) for some i, we thus obtain a dynamic microset structure called DM(i). To be able to initialise structures DM(i) (UF(i)) ($1 \leq i \leq \log g$) into collections of given sets, the initial tree types for UF(i) trees ($1 \leq i \leq \log g$) are stored in a separate table, with as parameters the number of elements (i.e., the number of leaves) and the number i with $1 \leq i \leq \log g$. (This table has a size at most $g.\log g$.)

We have the following theorem. We take $g = 2b$ and we replace i by $\lfloor \log g \rfloor$ if $i \geq \log g$; this does not influence the analysis since $a(i,h) \leq 4$ for $i \geq \lfloor \log g \rfloor$, $h \leq g$. The time complexity follows from [6] and the above observations for the microset structure.

Theorem 3.1 *A DM(i) structure and the algorithms that solve the dynamic microset problem with $b \leq min\{\frac{1}{14}\log n, \frac{1}{6}wordsize\}$ can be implemented such that the following holds. The total time that is needed for all Merge operations for a universe with n elements is $O(n.a(i,n))$ and the time needed for a Find or Search operation is $O(i)$, whereas the initialisation can be performed in $O(n)$ time and the entire structure takes $O(n)$ space ($i \geq 1$, $n \geq 2$).*

By applying the optimal structure UF(α) as in [6] (Section 5), we obtain the structure called DM(α), and we get the following theorem.

Theorem 3.2 *A DM(α) structure and the algorithms that solve the dynamic microset problem can be implemented with $b \leq min\{\frac{1}{14}\log n, \frac{1}{6}wordsize\}$ such that the following holds. The total time needed for all Merges and m Finds and Searchs is $O(n + m.\alpha(m,n))$, while the f^{th} Find or Search takes $O(\alpha(f,n))$ time, and where n is the total number of elements ($n \geq 2$). The initialisation takes $O(n)$ time and the entire structure takes $O(n)$ space.*

In case we want to extend the collection of elements from time to time (insertion of a new element in the universe), we do this for as for the Union-Find structure, where each transformation (rebuilding) of the Union-Find structure yields a recomputation of the appropriate tables. We now take $g = 4.2.b$ (since the transformation occurs at the latest when the number of elements becomes 4 times as large as the previous transformation), and, hence, $b \leq min\{\frac{1}{56}\log n, \frac{1}{24}wordsize\}$.

4 Applications

The dynamic microsets as developed in this paper can be applied in various problems that deal with incremental computation. Examples of these are the following.

- Maintaining the triconnected components in a graph while new edges or nodes are inserted one by one.. We refer to [8].
- Incremental planarity testing for general graphs, where again edges and nodes may be inserted in the graph, as long as it stays planar, and where it can be tested at any moment whether an edge may be inserted. We refer to [9]. This improves the results in [1, 13].
- Maintaining the 4-edge-connected components in a graph while new edges or nodes are inserted one by one [14].

All these algorithms run in time $O(n + m.\alpha(m,n))$, where n, m define the problem sizes (n is the number of nodes, and m is at most the total number of operations). Also, an incremental structure for trees can be obtained [9], that supports generalized nearest common ancestor queries, where e.g. two trees may be linked by a new edge or by the identification of two times two nodes (with the deletion of an appropriate edge), or where trees are partitioned into subtrees that may be joined and that are treated as one for alternative nca-queries (returning appropriate "sons" of nca's), thus giving an extension of the operations supported in [4].

5 Conclusion

We remark that dynamic microsets can be applied as an abstract data structure, and that it therefore can be used as a building block. It enables algorithms to make use of the power of direct addressing in a RAM, especially incremental algorithms. Finally, since the Union-Find problem is part of the dynamic microsets problem, it is easily seen by [2] that the structure is optimal on a RAM.

References

[1] G. Di Battista and R. Tamassia, Incremental Planarity Testing, Proc. 30^{th} Ann. Symp. on Found. of Comp. Sci. (FOCS) 1989, 436-441.

[2] M.L. Fredman and M.E. Saks, The Cell-Probe Complexity of Dynamic Data Structures, Proc. 21^{th} Ann. ACM Symp. on Theory of Comput. (STOC) 1989, 345-354

[3] H.N. Gabow and R.E. Tarjan, A Linear Time Algorithm for a Special Case of Disjoint Set Union, J. Comput. Syst. Sci. 30 (1985), 209-221.

[4] H.N. Gabow, Data Structures for Weighted Matching and Nearest Common Ancestors with Linking, Proc. 1^{st} Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA) 1990, 434-443.

[5] D. Harel and R.E. Tarjan, Fast Algorithms for Finding Nearest Common Ancestors, SIAM J. Comput. 13 (1984) 338-355.

[6] J.A. La Poutré, New Techniques for the Union-Find Problem, Proc. 1^{st} Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA) 1990, 54-63.

[7] J.A. La Poutré, Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines, Proc. 22^{th} Ann. ACM Symp. on Theory of Comput. (STOC) 1990, 34-44.

[8] J.A. La Poutré, Maintenance of Triconnected Components of Graphs, Proc. 19^{th} Int. Colloq. on Automata, Languages, and Programming (ICALP), LNCS 623 (1992), 354-365.

[9] J.A. La Poutré, Alpha-Algorithms for Incremental Planarity Testing, in preparation.

[10] R.E. Tarjan, Efficiency of a Good but Not Linear Set Union Algorithm, J. ACM 22, No. 2, April 1975, pp 215-225.

[11] R.E. Tarjan, A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets, J. Comput. Syst. Sci. 18 (1979), 110-127.

[12] R.E. Tarjan and J. van Leeuwen, Worst-case Analysis of Set Union Algorithms, J. ACM, 31, (1984), pp. 245-281.

[13] J. Westbrook, Fast incremental planarity testing, Proc. 19^{th} Int. Colloq. on Automata, Languages, and Programming (ICALP), LNCS 623 (1992), 342-353.

[14] J. Westbrook, personal communication, 1993.