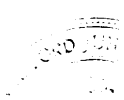
UNION-MEMBER ALGORITHMS FOR NON-DISJOINT SETS

by

Yossi Shiioach

STAN-CS-79-728
January 19 7 9

C O M P U T E R S C I E N C E D E P A R T M E N T
School of Humanities and Sciences
STANFORD UNIVERSITY



Union-Member Algorithms for Non-Disjoint Sets

Yossi Shiloach */

Computer Science Department
Stanford University
Stanford, California 94305

January 1979

Abstract.

In this paper we deal with the following -problem. We are given a finite set $U = (u_1, \ldots)$ and a set $\mathcal{S} = \{S_1, \ldots S_m\}$ of subsets of u. We are also given m-l UNION instructions that have the form UNION(S_i, S_j) and mean "add the set $S_i \cup S_j$ to the collection and delete S_1 and S_j." Interspaced among the UNIONs are MEMBER(i,j) questions that mean "does u_i belong to S_j?"

We present two algorithms that exhibit the trade-off among the three interesting parameters of this problem, which are:

1. Time required to answer one membership question.
2. Time required to perform the m-l UNIONs altogether.
3. Space .

We also give an application of these algorithms to the problem of 5-coloring of planar graphs.

Keywords. 5-coloring, height-balanced (AVL) tree, planar graphs, trie, UNION - MEMBER algorithms.

*/ This research was supported by a Chaim Weizmann Postdoctoral Fellowship and by National Science Foundation grants MCS 75-22870, MCS 77-23738.

1. Introduction.

Suppose we are given a finite set $U = \{u_1, \ldots, u_n\}$ and a set $\mathcal{S} = \{S_1, \ldots, S_m\}$ such that $S_j \subset U$ for all $1 \leq j \leq m$. The sets in $\mathcal{S}$ are not necessarily disjoint. We are also given m-1 UNION instructions that have the general form $\text{UNION}(S_i, S_j)$ which means: form a new set $S_{\min(i,j)}$ which is equal to $S_i \cup S_j$. Assigning the index min(i,j) to the new set guarantees that we don't give it an index of another set. Interspersed among the UNIONs are MEMBER(i,j) questions which mean: does u_i belong to S_j?

This problem has three interesting parameters.

1. The time required to answer a single MEMBER question.
2. The time required to perform the m-1 UNIONs altogether.
3. S-pace.

In Section 2 we shall present two algorithms that exhibit some trade-off among these three parameters.

The first algorithm answers a membership question in O(k) time, uses $O(I \cdot n^{1/k})$ space and requires $O(I\,k \log n)$ time to perform all the UNIONs. Here $I = |S_1| + |S_2| + \cdots + |S_m| + n$ represents the input size. A slight modification of this algorithm requires $O(I\,k \log m)$ time to perform all the UNIONs. Note that k need not be a constant. The interesting cases for this algorithm are when $1 \leq k \leq \log n$.

The second algorithm is designed for the case $k = \log n$. This algorithm uses recent results of Brown and Tarjan [BT] and improves the time bound for performing all the UNIONs to $O(I \log n)$. It requires $O(\log n)$ time to answer a membership question and O(1) space,

(These bounds are also obtained by the first algorithm upon substituting $k = \log n$.)

In Section 3 we give an implementation of these algorithms to the problem of coloring a planar graph with 5 colors. It yields a time bound which has the same asymptotic behavior as the recent algorithm of Lipton and Miller [LM] but is much simpler and faster practically.

2. Two UNION -MEMBER Algorithms.

2.1 The First Algorithm.

In the first algorithm we store each of the sets $S_1, \ldots, S_m$ in a trie structure with $n^{1/k}$ fields per node. This data-structure is described in detail in [K]. A slightly different presentation appears in [T].

The relevant facts about tries, as far as we are concerned, are:

1. A membership question can be answered in O(k) time.
2. Insertion takes O(k) time.
3. Each set S_j is stored in $O(|S_j| \cdot n^{1/k})$ space.

As we have mentioned before, k is an arbitrary positive integer which is not necessarily independent of n .

The last fact implies that the total space that we need to represent the initial configuration is $O(I \cdot n^{1/k})$. Basically, in order to perform $UNION(S_i, S_j)$ we take the set that has a smaller number of elements, say S_i , and insert its elements one by one into S_j . There are, however, three problems in this approach.

The first -problem is how to retrieve the elements of S_i efficiently. The trie structure does not support it very well and therefore each set will also be stored as a linked list. This requires O(1) space and therefore is negligible with respect to $O(I \cdot n^{1/k})$. The second problem is where do we store $S_i \cup S_j$. This problem contains, in fact, two sub-problems, namely where do we store the trie representation of $S_i \cup S_j$ and where do we store its list representation. In both cases we would like to store the representations of $S_i \cup S_j$ in the space that was occupied by the old representations of S_i and S_j . In this way we would

not exceed the space limit of the initial configuration. Practically, we first "clear" (see the third problem) the space that was occupied by S_i and then we insert S_i's elements one by one (reading them from the list) to the trie that represents S_j. This trie would probably have to expand and we let it use the vacant space of S_i. Obviously it won't need more space (assuming that in the beginning we have allocated $\lceil |S_r| n^{1/k} \rceil$ to every original set S_r, $1 \leq r \leq m$, even if its trie did not require that much space). If, when we insert an element of S_i we find that it has already been in S_j, then we delete it from the list of S_i. Finally this list will contain the set $S_i - S_j$ and then we just have to link it to S_j's list to yield a linked list without repetitions of the new set $S_i \cup S_j$. Note that the amount of time involved in these manipulations is still $O(k|S_i|)$ as if we have just inserted the elements of S_i, one by one, into S_j.

The third problem is the initialization of our data-structure and the clearance of spaces of sets that disappear such as S_i before. The solution to Exercise 2.12 in [AHU] allows us to avoid the initialization and therefore the clearance too. The implementation of this trick requires an extra $O(I)$ space and its time is also dominated by the overall bound of $O(I \cdot k \cdot \log n)$. It is quite straightforward and we shall leave the details to the reader.

So far we have shown that our data structure enables us to answer a membership question in $O(k)$ time and that we don't use more than $O(I\, n^{1/k})$ space in the whole algorithm. Let's show now that the time required to -perform all the UNIONs is $O(I\, k \log n)$,

Let $G = (V,E)$ be the bipartite graph defined by

$$V = V \cup \mathcal{S} \quad ; \quad E = \{(u_\ell, S_j) : u_\ell \in S_j, 1 \le \ell \le n, 1 \le j \le m\}.$$

Let's consider a UNION(S_i, S_j) in which S_i is inserted into S_j. Henceforth we shall assume that the new set will have the name of the accepting set, S_j in this case.

Let's consider all the edges of G that are incident with S_i. If (u_ℓ, S_i) is such an edge and $u_\ell \in S_j$ too, we say that the edge (u_ℓ, S_i) disappears when UNION(S_i, S_j) is performed. However, if $u_\ell \notin S_j$ then (u_ℓ, S_i) does not disappear but just changes its "name" to (u_ℓ, S_j). Thus, original edges of the graph can either disappear or change their names. One can easily see that the inherent complexity of the algorithm is in making edges disappear and in changing their names. These two operations take $O(k)$ time and therefore can be regarded as elementary operations. We then have to show that the number of elementary operations is $O(I \log n)$.

Let's consider UNION(S_i, S_j) again. The number of edges that disappear is $|S_i \cap S_j|$ and the number of edges that change their name is $|S_i - S_j|$. If $|S_i - S_j| \le |S_i \cap S_j|$ we will charge the disappearing edges also for the time involved in changing the name of the others; yet each edge that disappears will still be charged for at most one edge that changed its name. Since $|E| < I$ and each edge disappears at most once, the total number of elementary operations that will be charged on the accounts of disappearing edges will be $O(1)$. The accounts of edges that change their names are charged only when $|S_i - S_j| > |S_i \cap S_j|$. Since $|S_i| \le |S_j|$ this implies that $|S_i \cup S_j| > \frac{3}{2} S_i$. Thus, each edge can be charged for changing its name at most $\log_{3/2} n$ times, and this

yields the desired result. (Note that an edge can change its name more than $\log_{3/2} n$ times.)

A slight modification of the algorithm above yields a total time of $O(I\ k \log m)$ for performing all the UNIONs.

Let an original set denote a set that is an element of $\mathcal{S}$. If, when we perform UNION(S_i,S_j), we insert the one that contains a smaller number of original sets into the one that contains more original set, then we can easily get the bound above.

2.2 The Second Algorithm.

In this case we set $k = \log n$. Thus, we are interested in an algorithm that answers a membership question in $O(\log n)$ time, uses linear space and is as efficient as possible. Every data structure (including the previous one) that supports search and insertion in logarithmic time and linear space can meet these requirements with a total time of $O(I \log^2 n)$ for executing all the UNIONs. (One log term comes from the cost of a basic operation in the data structure and another one comes from the fact that an edge can be charged $O(\log n)$ times for changing its name.) The following algorithm uses a recent result of Brown and Tarjan [BT] that enables us to knock down one log term bringing the total time down to $O(I \log n)$.

This time we shall keep each set in a height-balanced (AVL) tree and not as a list. These trees will, however, represent sorted lists in the sense that if we traverse them in inorder, the indices of the u_i's will be strictly increasing (see [BT]). We also use an auxiliary space of size n. In order to perform UNION(S_i,S_j) in which S_i should be

inserted into S_j (i.e., $|S_i| \leq |S_j|$), we first read S_i from its tree in inorder and put it as a sorted list in the auxiliary space. Then we insert S_i's elements one by one in increasing order, into S_j's tree, allowing it to expand into the space occupied by S_i's tree (which we don't need any more). As we have mentioned before, there is a nifty trick that allows us to use this "dirty" space without cleaning it up first. From the same reason, we don't have to clear the auxiliary space and it can be used for all the UNIONs.

An AVL tree supports a search in log time and occupies linear space. Thus, we just have to show that the time bound for carrying out all the UNIONs is $O(I \log I)$.

At this point we have to turn to Brown and Tarjan's paper [BT]. This paper deals with fast merging algorithms. Using AVL trees to represent the sets in a sorted inorder manner, the authors were able to insert the elements of the smaller set, say S_i, one by one to the tree of the larger set, say S_j, in time of $O(|S_i|(1 + \log |S_j| - \log |S_i|))$, The resulting tree represents $S_i \cup S_j$ in a sorted inorder manner and therefore can be reused later.

In order to establish an $O(I \log I)$ time bound for our algorithm we shall use the same graph G as before and charge the operations to its edges. If $|S_i \cap S_j| \geq |S_i - S_j|$ we charge each of the disappearing edges by $2(1 + \log |S_j| - \log |S_i|) < 2 \log n + 1$. An edge can disappear at most once and therefore this account will not cause any trouble. When $|S_i - S_j| > |S_i \cap S_j|$ we charge each edge by $1 + \log |S_j| - \log |S_i|$. By the same reasoning as before, an edge will be charged by this amount when changing its name at most $\log_{3/2} n$ times. That takes care of

the 1 and we are left with $\log |S_j| - \log |S_i|$. Let (u_ℓ, S_i) be an edge which has just been charged by this amount and changed its name to (u_ℓ, S_j). Since sets keep growing all the time, the next time that our edge will be charged when changing its name from (u_ℓ, S_p) to (u_ℓ, S_q) we will have $|S_p| \geq |S_j|$. This time it will be charged by $\log |S_q| - \log |S_p|$, and together with the previous amount it will sum up to

$$\log |S_q| - \log |S_p| \quad \log |S_j| - \log |S_i| \leq \log |S_q|$$

This argument shows that all these amounts form kind of a telescoping series bounded by $\log n$. Summing everything up, an edge can be charged once by $2(1 + \log|S_j| - \log |S_i|)$ for some $S_i, S_j \subseteq U$ and can accumulate at most $2 \log n$ from charges that are made when it changes its name. Since $|E| < I$, the proof is complete,

3. An Application to 5-Coloring of Planar Graphs.

In a recent paper, R. J. Lipton and R. E. Miller [LM] present an $O(n \log n)$ algorithm for 5-coloring a planar graph with n vertices. However, the constant factor which they provide is derived from the recurrence relation $T(n) = T(\lambda n) + O(n \log n)$ in which λ can achieve values which are very close to $27/28$ and the multiplicative constant of $n \log n$ is not very small either. Even if it is just 2, it would yield $T(n) \approx 56\, n \log n$, while for all practical purposes $56 > \log n$.

Lipton and Miller's algorithm follows the lines of the constructive proof of the $-color theorem which is given in [H]. There is, however, a much simpler (and constructive) proof of the 5-color theorem which follows the lines of [O] and can be utilized by the algorithms above,

The proof proceeds by induction on n and the basis for the induction is trivial. Thus, let's assume that any planar graph with at most n-1 vertices is 5-colorable, and let G be a planar graph with n vertices, Obviously, if G contains a vertex of degree ≤ 4 we are done. If not, there exists a vertex, say v_0, of degree 5. Let $v_1,\ldots,v_5$ be v_0's neighbors. At least two of them, say v_1 and v_2, are not adjacent to each other. We now contract v_0, v_1, and v_2 into one vertex v_*, yielding a planar graph G' that has n-2 vertices. Let's consider a 5-coloring of G' in which v_* has color #1 and v_3, v_4, and v_5 has colors #3, 4, and 5, respectively. Now, we can color G by 5 colors assigning color #1 to v_1 and v_2 and color #2 to v_0. The proof is complete.

When one tries to extract an algorithm out of this proof, it seems that two operations have kind of a contradictory nature. One is the contraction of two vertices into one (contraction of three vertices can be regarded as two such steps), and the other is to determine whether two vertices are adjacent or not. Data structures that support fast adjacency tests, such as an adjacency matrix, usually require a lot of space and have poor performance in making contractions. Other data structures that support contractions in short time require too much time for adjacency tests. At this point, our algorithms get into the picture.

Let V be our universal set and let $\mathcal{S} = \{S_1, \ldots, S_n\}$ where S_i is the set of vertices adjacent with v_i, $1 \leq i \leq n$. In these terms, it is easy to see that contraction of v_i and v_j into one vertex transforms to UNION(S_i, S_j) and an adjacency test of v_i and v_j transforms to MEMBER(i,j). Obviously, the UNION- MEMBER routine is only a part of the 5-coloring algorithm. We have to store and update the degrees of the vertices, delete vertices of degree ≤ 4, and record some information that will enable us to expand the graph back from one "big" vertex (or 5 "big" vertices) to its original size and also trace the 5-coloring back from the smallest graph to the original one. Thus, a lot of details should be accomplished if one attempts to design a complete 5-coloring algorithm out of these ideas. However, the UNION - MEMBER routine is the core of such an algorithm and the most time-consuming part of it. All the other things can be done in linear time and space. Both UNION-MEMBER algorithms yield an O(n log n) time bound for the coloring algorithm, and the second one yields linear space too. We believe that the constant factor here is much lower than the one in [LM] and that the algorithm is conceptually simpler,

References

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass,, (1974).

[BT] M. R. Brown and R. E. Tarjan, "A Fast Merging Algorithm," Stanford Computer Science Department Report STAN-CS-77-625 (1977), (to appear in Journal ACM).

[H] F. Harary, Graph Theory, Addison-Wesley, Reading, Mass., (1969).

[K] D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., (1973).

[LM] R. J. Lipton and R. E. Miller, "A Batching Method for Coloring Planar Graphs," Information Processing Letters 7, 4 (1978), 185-188.

[O] 0. Ore, The Four Color Problem, Academic Press, (1967).

[T] R. E. Tarjan, "Storing a Sparse Table," Stanford Computer Science Department Report STAN-CS-78-683, (1978).