An On-Line Edge-Deletion Problem

SHIMON EVEN

University of California, Berkeley, California

AND

YOSSI SHILOACH

Stanford University, Stanford, California

ABSTRACT. There is given an undirected graph $G = (V, E)$ from which edges are deleted one at a time and about which questions of the type, "Are the vertices u and v in the same connected component?" have to be answered "on-line." There is presented an algorithm which maintains a data structure in which each question is answered in constant time and for which the total time involved in answering q questions and maintaining the data structure is $O(q + |V| \cdot |E|)$.

KEY WORDS AND PHRASES: algorithm, connectivity checks, edge deletion, on line

CR CATEGORIES: 5.25, 5.32

1. *Introduction*

Suppose we are given an undirected finite graph $G(V, E)$ from which edges may be deleted, one at a time, and about which questions of the type, "Are vertices u and v in the same connected component?" may have to be answered at any point in time. If the whole sequence of edge deletions and connectivity questions is known, then we can use the set union algorithm [1, 4] on the reversed sequence, by starting with the final graph $G'(V, E')$, finding its connected components in $O(E' + V)$ time, and adding the edges one by one until we reach $G(V, E)$. In this case q questions can be answered in $O(m\alpha(m, n))$ time (see [4]), where $m = |E - E'| + q$ and $n = |V| - 1$, namely, in time almost linear in the length of the sequence.

However, if we have to answer the questions in an "on-line" fashion, the problem seems to be much more time consuming. The naive algorithm which checks the connectivity for each question separately takes time $O(q \cdot |E|)$.

This on-line problem was tackled by Cheston [2, Ch. 5]. He introduced and compared the performance of four algorithms (excluding the naive one, which he called the "start over" algorithm) for updating the connectivity information after

The work of the first author was supported by the National Science Foundation under Grant 21492. The work of the second author was supported in part by a Chaim Weizmann Postdoctoral Fellowship and in part by ONR Contract N00014-76-C-0688.

This work was performed while the first author was on leave from the Technion, Haifa, Israel.

Authors' present addresses: S. Even, Computer Science Department, Technion, Haifa, Israel; Y. Shiloach, IBM Israel Scientific Center, Technion City, Haifa, Israel.

edge deletions. All four algorithms maintain data structures which enable one to answer a connectivity question in constant time. However, the time required for updating this connectivity information is $O(|E|)$ per edge deletion in the first two algorithms and less efficient in the latter two. Thus, the best time bound his algorithms achieve is $O(q + |E|^2)$.

In Section 2 we show how the problem can be solved in $O(q + |V| \log |V|)$ if G is a tree or a forest. The solution for trees is included primarily as a warm-up and because it is similar to a part of the solution for general graphs. In Section 3 we demonstrate a solution for general graphs in time $O(q + |E| \cdot |V|)$. Clearly, this is better than the naive algorithm mentioned above if $q \gg |V|$ and better than the algorithms of Cheston. By using a fast average-time algorithm for computing connected components, Karp [3] solved the corresponding problem for random graphs by an $O(q + |V|^2 \log^c |V|)$ average-time algorithm.

2. *An Algorithm for Circuit-Free Graphs*

If $G(V, E)$ is circuit-free, then it is either a tree or a forest, and the deletion of any edge breaks the graph into a forest with one more tree. We use a table for the vertices in which the name of the component to which a vertex belongs is specified. Thus the question, "Is a connected to b?" can always be answered in constant time. Therefore, answering q questions takes $O(q)$ time. It remains to be shown that updating the table takes at most $O(|V| \log |V|)$ time.

The number of edges $|E|$ in G is bounded by $|V| - 1$. Each time an edge e is deleted from a tree T, we scan T from both endpoints of e in *parallel*,[1] attempting to explore each component of T fully. When one of these scans terminates, we stop scanning, and a new name of a component is assigned to all the vertices on the part for which the scan terminated. Since the number of vertices (edges) in the renamed component does not exceed the number of vertices (edges) of its mate, each vertex can belong to a renamed component at most $\lfloor \log |V| \rfloor$ times. Let us "charge" a vertex, whose component name is changed, for the scanning of the edge through which it is reached and for the edge scanned, in parallel, in the mate component. Thus each vertex is "charged" at most $\lfloor \log |V| \rfloor$ times, yielding time complexity of $O(|V| \log |V|)$ for the whole process of updating the vertex tables.

3. *An Algorithm for General Graphs*

As in the algorithm of Section 2, we keep a vertex table which contains for each vertex the name of the component to which it belongs. Thus each question of whether two vertices belong to the same component can be answered by comparing the component names. The method for updating the component names is as follows.

Our scheme uses two processes which run in parallel. Process A checks whether the edge deletion breaks a component, and if it does, both processes halt. Process B checks whether the edge deletion does not break the component to which it belongs, and if it does not, again both processes halt. We bound the total time spent on runs which are halted by process A by $O(|E| \log |E|)$ and the total time spent on runs which are halted by process B by $O(|V| \cdot |E|)$, yielding an overall time complexity $O(|V| \cdot |E|)$.

[1] The meaning of parallel is not that of parallel processing. We simply mean that if algorithms A and B have to be executed and they are represented by two sequences of operations $(\alpha_1, \alpha_2, \ldots)$ and $(\beta_1, \beta_2, \ldots)$, respectively, then we carry them out alternatively by executing the sequence $(\alpha_1, \beta_1, \alpha_2, \beta_2, \ldots)$.

Process A, whose task is to detect early the cases in which the edge deletion breaks a component, may detect that the component does not break, but this is of no importance. In this case we ignore its conclusion and continue with process B until it reaches the already known fact. The reason for this is that the breadth-first search structure, used in process B and to be described shortly, must be maintained. Thus we need only discuss the complexity of process A in case the edge deletion breaks a component.

In process A we use some method of scanning, say depth-first search [1], and the process is similar to that of the previous section. We start scanning, in parallel, from both endpoints, a and b, of the deleted edge e. Once one of the scans terminates in failure, that is without reaching the other endpoint of e although all its edges have been examined, the other scan is terminated too. The original component is now broken into two components: The vertices of the smaller component (the one in which the scan terminated first) get a new component name. By an argument analogous to the one used in Section 2, in which the edges are "charged," instead of the vertices, the time complexity of process A is $O(|E|\log|E|)$.

Process B uses a breadth-first structure (BFS), and therefore an initialization is required to create the first BFS structure. This is done as follows. A vertex r is chosen and the BFS starts from it. The only vertex in level L_0 is r. All the vertices of distance i from r are in level L_i. If G is not connected, a new scan is started at some unscanned vertex v, v is put in L_1, and an *artificial edge* connects r with v; all vertices of distance i from v are now in level L_{i+1}, etc. Artificial edges are introduced in order to keep all the connected components in one BFS structure and are used only for this purpose. Maintaining a unified BFS structure will simplify the evaluation of the complexity later. Clearly, the artificial edges are used only in process B.

The structure has the following properties. A vertex v in level L_i, $i > 0$, has at least one edge connecting it to some vertex in L_{i-1}, and if there is only one such edge, it may be artificial, but if there are more, then none of them is artificial; v may have any number of edges connecting it with other vertices in L_i and with vertices in L_{i+1}, but no edges connect it with vertices of levels other than L_{i-1}, L_i, and L_{i+1}. Let $\alpha(v)$, $\beta(v)$, and $\gamma(v)$ be the sets of edges which connect it with L_{i-1}, L_i, and L_{i+1}, respectively.

Process B now proceeds as follows. When an edge $u \overset{e}{—} v$ is deleted, we check the levels of u and v. There are two cases:

Case 1. Both u and v are on the same level. In this case the edge deletion cannot change the components. The edge is simply deleted from $\beta(u)$ and $\beta(v)$, and process B halts (and therefore process A is halted too). We still have a BFS structure, as above.

Case 2. u and v are on different levels. Without loss of generality we can assume that $u \in L_{i-1}$ and $v \in L_i$. We remove e from $\gamma(u)$ and $\alpha(v)$.

Case 2.1 If the new $\alpha(v)$ is not empty, then the components have not changed, and both processes halt.

Case 2.2 If the new $\alpha(v)$ is empty, v has to drop at least one level, and its drop may cause a whole avalanche. We use a queue Q on which we put vertices whose level must be changed. Vertex v is put on Q and the following procedure is applied:

(1) If Q is empty, the procedure and both processes halt.
(2) Let w be the first element of Q. Remove w from Q.
(3) Remove w from its level (say, L_j), and put it in the next level (L_{j+1}).
(4) For each $w \overset{e'}{—} w'$ in $\beta(w)$, remove e' from $\beta(w')$ and put it in $\gamma(w')$.

(5) $\alpha(w) \leftarrow \beta(w)$.
(6) For each $w \overset{e'}{-} w'$ in $\gamma(w)$, remove e' from $\alpha(w')$ and put it in $\beta(w')$; if the new $\alpha(w')$ is empty, put w' on Q.
(7) $\beta(w) \leftarrow \gamma(w)$, $\gamma(w) \leftarrow \varnothing$.
(8) If $\alpha(w)$ is empty, put w on Q.
(9) Return to (1).

If the deletion of e does not break any component and we are in case 2.2, then eventually the procedure will halt. In this case it is easy to see that the BFS structure is maintained correctly. If its deletion does break a component, then the procedure will not halt by itself. However, process A, recognizing the break, will halt, and both processes will halt. In this case all the changes made in the BFS structure are ignored, and we go back to the BFS structure we had just before the deletion of e, except that e is now replaced by an artificial edge. Clearly, in this case v is now the root of a tree which includes the new component, and perhaps additional components, through some other artificial edges. Also, there are no edges connecting the descendants of v with any vertices which are not v's descendants, except the artificial edge u–v. One way to realize the return to the structure preceding the deletion of e without having to copy the whole structure is to keep on a stack all the changes that took place in the BFS structure since the deletion of e and undo them one by one. This way the processing time is only multiplied by a constant.

It remains to show that the total time spent on runs which are terminated by process B is bounded by $O(|V| \cdot |E|)$. For each w taken off Q the amount of time spent in the procedure is proportional to $d(w)$, the degree of w, since each "movement" of an edge takes some constant time. However, we can "charge" the edges instead, namely, "charge" the cost of handling an edge e' to the edge each time it is processed. Now observe what whenever e' is processed in the procedure, one of its endpoints drops by one level. Since the lowest level a vertex can reach in runs which are terminated by process B is $L_{|V|-1}$, an edge can be charged at most $2 \cdot |V|$. Thus the whole cost is bounded by $O(|V| \cdot |E|)$.

ACKNOWLEDGMENTS. The authors are indebted to Richard M. Karp and to the referees for their careful reading of the manuscript and for their valuable suggestions.

REFERENCES

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. CHESTON, G.A. Incremental algorithms in graph theory. Ph.D. Diss., Dep. of Computer Science, Univ. of Toronto, March 1976 (Tech. Rep. No. 91).
3. KARP, R.M. Private communication.
4. TARJAN, R.E. Efficiency of a good but not linear set union algorithm. *J. ACM 22,* 2 (April 1975), 215–225.

RECEIVED DECEMBER 1977; REVISED FEBRUARY 1980; ACCEPTED FEBRUARY 1980