AN ON-LINE EDGE-DELETION PROBLEM

by

S. Even and Y. Shiloach

Technical Report #154
(Revised: June 1979)
AN ON-LINE EDGE-DELETION PROBLEM

by

S. Even and Y. Shiloach

November 1977
(Revised June 1979)

ABSTRACT

We are given an undirected graph \( G = (V,E) \) from which edges are deleted one at a time and questions of the type: "Are the vertices \( u \) and \( v \) in the same connected component?" have to be answered "on line".

We present an algorithm which maintains a data-structure in which each question is answered in constant time and the total time which is involved in answering \( q \) questions and maintaining the data-structure is \( O(q + |V| \cdot |E|) \).
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 \([1,2]\) 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 \([2]\)) , 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 of checking the connectivity for each question separately takes time \( O(q \cdot |E|) \).

This on-line problem was tackled by Cheston \([3]; \text{Chapter 5}\). 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 edge deletions. All four algorithms maintain data-structures which enable 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 complexity time bound his algorithms achieve is \( O(q + |E|^2) \).

In Section 2 we shall 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 shall show 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 [4] solved the corresponding problem for random graphs by an \( O(q + |V|^2 \log |V|) \) 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 shall use a table for the vertices in which the name of the component to which a vertex belongs is specified. Thus, the question of "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 shall scan \( T \) from both endpoints of \( e \), in parallel\(^*\), 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. Thus, if one part contains \( n_1 \) vertices and the other \( n_2 \geq n_1 \),

\(^*\) The meaning of parallel is not that of a parallel processing. We simply mean that if algorithm \( A \) 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, executing the sequence \( (\alpha_1,\beta_1,\alpha_2,\beta_2,\ldots) \).
then both the scan and the reassignment take at most \( O(n^1) \) time. Let \( f(n) \) be the maximum time spent by the algorithm, on a tree with \( n \) vertices if all its edges are removed one by one. We have

\[
f(n) \leq f(n_1) + f(n_2) + c \cdot n_1,
\]

for some constant \( c \) and some \( n_1 \) and \( n_2 \) where \( n_1 < n/2 \) and \( n_1 + n_2 = n \). Clearly, \( f(1) = 0 \).

**Lemma 1.** \( f(n) = O(n \log n) \).

**Proof.** By induction on \( n \). Assume that for \( n_0 < n \) \( f(n_0) \leq K \cdot n_0 \log n_0 \). For definiteness, let us assume that the base of all the logs is \( 2 \). By the definition of \( f \) and the inductive hypothesis, we have

\[
f(n) \leq K \cdot n_1 \log n_1 + K \cdot n_2 \log n_2 + c \cdot n_1,
\]

where \( n_1 \) and \( n_2 \) are as in the discussion preceding the lemma. It is sufficient to prove that

\[
K \cdot n_1 \log n_1 + K \cdot n_2 \log n_2 + c \cdot n_1 \leq K \cdot n \log n,
\]

which is equivalent to

\[
K \cdot n_1 (\log n_1 - \log n) + K \cdot n_2 (\log n_2 - \log n) + c \cdot n_1 \leq 0.
\]

Let \( x = n_1/n \). Then \( 0 < x \leq 1/2 \) and the above is equivalent to showing that

\[
g(x) = x \log x + (1-x) \log (1-x) + \frac{c}{K} \cdot x \leq 0.
\]

It is easy to verify that \( \lim_{x \to 0} g(x) = 0 \) and if \( K \geq c/2 \) then \( g(1/2) \leq 0 \). Moreover, \( g''(x) = (\log e)(\frac{1}{x} + \frac{1}{1-x}) > 0 \) for all \( 0 < x < 1/2 \). Thus \( g(x) \) is convex and therefore \( g(x) \leq 0 \) for all \( 0 < x < 1/2 \). \( \Box \)
3. AN ALGORITHM FOR GENERAL GRAPHS

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 the 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 \( O(|V| \cdot |E|) \).

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 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. If the original component has \( m \) edges and the two new components have \( m_1 \) and \( m_2 \) edges, respectively, then \( m_1 + m_2 = m - 1 \) and without loss of generality we can assume that \( m_1 < m/2 \). Therefore, \( m_1 < m/2 \).
The scan of the smaller component, and therefore the whole scan is bounded by \( O(m) \) time. Let \( g(m) \) be the maximum time spent by process A on a component of \( m \) edges upon deletions of edges which break it or its derived components. Thus,

\[
g(m) \leq g(m_1) + g(m_2) + c \cdot m_1
\]

for some constant \( c \) and some \( m_1 \) and \( m_2 \) such that \( m_1 < m/2 \) and \( m_1 + m_2 = m-1 \).

**Lemma 2.** \( g(m) = O(m \log m) \).

The proof is similar to that of Lemma 1.

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 in 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. 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

* Artificial edges are introduced in order to keep all the connected components "hanged" 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.
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 $u \rightarrow v$ 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. It 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 $v$ be the first element of $Q$. Remove $v$ from $Q$.
3. Remove $v$ from the level it is in say $(L_j)$, and put it in the next level $(L_{j+1})$.
4. For each $v \rightarrow v'$ in $\beta(v)$, remove $e'$ from $\beta(v')$ and put it in $\gamma(v')$.
5. $\alpha(v) + \beta(v)$.
For each \( v \xrightarrow{e} v' \) in \( \gamma(v) \), remove \( e \) from \( \alpha(v') \) and put it in \( \beta(v') \); if the new \( \alpha(v') \) is empty, put \( v' \) on \( Q \).

(7) \( \beta(v) + \gamma(v) \), \( \gamma(v) \) \( \cup \emptyset \).

(8) If \( \alpha(v) \) is empty, put \( v \) 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 will halt, recognizing the break, 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 \( u \xrightarrow{-} v \). 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 DFS 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 \( O(|V|E|E|) \). For each \( v \), taken

off \( Q \), the amount of time spent in the procedure is proportional to \( d(v) \),

the degree of \( v \). However, we can "charge" the edges instead.
Namely, charge the cost of handling an edge $e'$ to the edge, each time it is looked at. Now observe what whenever $e'$ is looked at 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 $O(|V|\cdot|E|)$.

REFERENCES


ACKNOWLEDGMENT

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