Determining Single Connectivity in Directed Graphs

Adam L. Buchsbaum[1]
Martin C. Carlisle[2]

Research Report CS-TR-390-92
September 1992

Abstract

In this paper, we consider the problem of determining whether or not a directed graph contains a pair of vertices connected by two distinct simple paths; if it does not, we say it is *singly connected*. A straightforward implementation using n depth first searches requires $O(nm)$ time on an n-vertex, m-arc digraph; we obtain an $O(n^2)$ time algorithm by using contraction wherever possible.

[1]Supported by a Fannie and John Hertz Foundation fellowship, National Science Foundation Grant No. CCR-8920505, and the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) under NSF-STC88-09648.

[2]Supported by a National Science Foundation Graduate Fellowship and the Fannie and John Hertz Foundation.

1 Introduction

Graph connectivity has become a widely studied component of graph theory. To date, most of the work in this area has dealt with determining if an undirected graph contains *at least* two, three, or more vertex- or edge-disjoint paths between every pair of vertices (see, e.g., [5, 7, 8, 11, 12, 14, 15, 17, 20, 21]) or if a directed graph possesses analogous properties (see, e.g., [5, 16]). This paper investigates a different sort of connectivity problem: determining if a graph has *at most* some number of distinct paths between every pair of vertices. In particular, we say a directed graph is *singly connected* if for any pair of vertices, there exists at most one simple (no vertex repeated) path connecting them; i.e., if there are two distinct (but not necessarily vertex- or arc-disjoint) simple paths connecting any two vertices, the graph is not singly connected. Multiple paths via cycles are allowed (in particular, a graph which is merely one simple cycle is singly connected.) Note that the undirected version of this problem is trivial: An undirected graph is singly connected if and only if it is a tree. The directed problem in some sense is equivalent to checking the graph to see if it contains any redundant paths. This problem originally appears in [4].

A simple algorithm for solving this decision problem is to perform a depth first search (DFS) starting from each vertex, terminating the DFS after it visits all vertices reachable from the start vertex. (We use depth first search terminology from [1, 4] in this paper.) If the same vertex is encountered along two different simple paths from the initial vertex, we may conclude that the graph is not singly connected. Similarly, if we do not find such a case for any initial vertex, then the graph is singly connected. The question thus reduces to determining whether or not there are any forward or cross arcs in any of the DFS trees.

Given an n-vertex, m-arc directed graph, this naive algorithm uses n occurrences of DFS, each of which in the worst case requires $O(m)$ operations, yielding a total cost of $O(nm)$ time. The running time is therefore $O(n^2)$ for sparse graphs and $O(n^3)$ for dense graphs. (We assume the graphs are connected in the undirected sense.) An alternative approach performs one complete DFS starting from any vertex, yielding a DFS forest possibly with multiple tree roots. The problem is resolving the inter-tree cross arcs to determine if they produce forbidden multiple simple paths; again this can potentially require $O(nm)$ time.

In the following sections, we give an $O(n^2)$ algorithm for the single connectivity decision problem for directed graphs. This algorithm is optimal for dense graphs as in the worst case we must consider every arc. However, it yields no asymptotic improvement for sparse graphs.

1.1 Definitions

We use standard graph terminology; see, e.g., [22]. In particular, given a directed graph (digraph) $G = (V, A)$ with $n = |V|$ and $m = |A|$, a *path* connecting two vertices $u, v \in V$ is a sequence of nodes $(u = u_0, u_1, \ldots, u_l = v)$ such that $(u_i, u_{i+1}) \in A,\ 0 \leq i < l$. A path is a *simple path* if $u_i \neq u_j,\ \forall i \neq j$. Two paths $(u_0, \ldots, u_k)$ and $(v_0, \ldots, v_l)$ are *distinct* if either (1) $k \neq l$, or (2) $k = l$ and $u_i \neq v_i$ for some $0 \leq i \leq l$. A *cycle* is a path $(u_0, \ldots, u_l)$ such that $u_0 = u_l$; a *simple cycle* is a cycle $(u_0, \ldots, u_l)$ such that $u_i \neq u_j,\ 0 \leq i < j < l$. Given some cycle, a *chord* is a path of arcs, none of which is on the cycle, connecting two distinct nodes of the cycle. Note that this differs from the standard definition of a chord being an arc connecting two nonadjacent nodes of a cycle (see, e.g., [2, 13]); for a single arc to be a chord by our definition, however, it must in fact connect two nonadjacent nodes of the cycle.

Definition 1.1 *A digraph $G = (V, A)$ is* singly connected *if and only if for any two vertices $u, v \in V$, there exists at most one simple path connecting u and v.*

We note that cycles are permitted in the graph, and that there may in fact be more than one path between some pair of vertices in a singly connected digraph. However, there cannot be two or more distinct (but possibly not vertex- or arc-disjoint) simple paths between any pair of vertices.

For the remainder of this paper, all graphs are assumed to be directed and connected in the undirected sense; therefore $n - 1 \leq m \leq n(n-1)$.

2 Algorithm for Determining Single Connectivity

In this section, we first prove certain aspects of the naive algorithm. Then, building on these, we show how we can use contraction to improve the performance of the algorithm. First, we note that the existence of forward or cross arcs within a DFS tree implies that the graph is not singly connected. This clearly follows since if there is a forward arc (w, x) then there are two paths from w to x, namely the tree path and the forward arc. Also, if there is a cross arc (w, x), there are two paths from the root of the DFS tree to x, namely the tree path to x, and the tree path to w catenated with the arc (w, x). In the following lemma, we show this condition is not only sufficient but also necessary:

Lemma 2.1 *If G is not singly connected, then for some $v \in V$, the DFS tree rooted at v has a forward or cross arc.*

PROOF: Since G is not singly connected there exist $v, x \in V(G)$ such that two distinct simple paths connect v to x. Consider the DFS tree T rooted at v. Let $P = (v = t_0, \ldots, t_k = x)$ be the path in T from v to x, and let $P' = (v = u_0, \ldots, u_l = x)$ be some path in G distinct from P. Let $i = \max\left\{j < l \mid (u_j, u_{j+1}) \neq (t_{k-(l-j)}, t_{k-(l-j)+1})\right\}$. I.e., $(u_{i+1}, \ldots, u_l) = (t_{k-(l-i)+1}, \ldots, t_k)$, but $(u_i, \ldots, u_l) \neq (t_{k-(l-i)}, \ldots, t_k)$. Then (u_i, u_{i+1}) is a forward arc if $u_i = t_j$ for some j and a cross arc otherwise. $\square$

We now consider the effect of cycles on single connectivity. Intuitively, a cycle would appear to be equivalent to a single vertex, since each vertex in the cycle is reachable from every other vertex by a simple path. Thus, ideally we would like to contract each cycle (i.e., all the arcs forming each cycle) to a single vertex and run the naive algorithm on the reduced graph, checking only for forward or cross arcs. However, whereas contracting a cycle in a digraph preserves connectivity, it does not always preserve single connectivity. For example, consider the digraph consisting only of a cycle with a chordal arc; the graph itself is not singly connected, but contracting the cycle leaves only a single vertex which by definition is singly connected.

In fact, chords play a decisive role in determining when cycle contraction is useful. Again, note that if a graph has a chord, it is not singly connected (the chord from x to y and the cycle path from x to y are two distinct simple paths from x to y.) Now consider the strongly connected components of G. Let G^* be the graph formed by contracting each strongly connected component of G into a single vertex and removing *loop arcs*, i.e., arcs from a vertex to itself in G^*, formed by contracting arcs joining two nodes in the same strongly connected component of G. A *multiple arc* of G^* is a repeated arc between two distinct vertices in G^*, formed if more than one arc connected two strongly connected components in G. The following two lemmas relate the single connectivity of G to that of G^*.

Lemma 2.2 *If G^* has a multiple arc, then G is not singly connected.*

Proof: Let x and y be two vertices in G^* connected by multiple arcs (from x to y). Let X and Y be, respectively, the strongly connected components of G that contract to x and y in G^*. The multiple arcs in G^* arise from two arcs (x', y') and (x'', y'') in G such that $x', x'' \in V(X)$ and $y', y'' \in V(Y)$. It may be the case that either $x' = x''$ or $y' = y''$, but both pairs cannot be equal. Let P_x be the path in G from x' to x'', and let P_y be the path in G from y' to y''; at most one of P_x and P_y can be the null path. Now the path formed by catenating P_x with the arc (x'', y'') and the path formed by catenating the arc (x', y') with P_y demonstrate two distinct simple paths joining x' to y'' in G. $\square$

Lemma 2.3 *If G has no chords and G^* has no multiple arcs, then G is singly connected if and only if G^* is singly connected.*

Proof: Suppose G is not singly connected, and let $P_1 = (x = u_0, \ldots, u_{l_1} = y)$ and $P_2 = (x = v_0, \ldots, v_{l_2} = y)$ be two distinct simple paths from x to y in G. We can assume without loss of generality that P_1 and P_2 are vertex-disjoint. It must be that x and y are in different strongly connected components of G, or P_2 would yield a chord of the cycle formed by catenating a path from y to x with P_1. Let X (rsp. Y) be the strongly connected component of G (and its respective vertex in G^*) containing x (rsp. y). Let P_1^* and P_2^* be the paths in G^* from X to Y that result from the contraction applied to P_1 and P_2. Since G^* contains no multiple arcs, the first arc of P_1 must differ from that of P_2, and thus P_1 and P_2 are distinct simple paths in G^*.

Suppose G^* is not singly connected, and let P_1^* and P_2^* be two simple paths connecting two vertices X and Y in G. By replacing each vertex in the two paths by an appropriate path in the respective strongly connected component in G, these give rise to two simple paths connecting some x and y in G. $\square$

We can use Lemmas 2.2 and 2.3 to reduce the graph G to an acyclic graph G^* and then run a depth first search from each vertex of G^* to determine single connectivity. As we shall see, this reduces the complexity of the multiple DFS approach by a factor of $O(m/n)$. Postponing for now discussion of how to determine if G has a chord, our algorithm for testing whether G is singly connected is simply stated:

```
If G has a chord, then G is not singly connected.
Otherwise, compute the strongly connected components (SCC's) of G.
    Contract the SCC's and throw away loop arcs; let G* be the resulting graph.
    If G* has a multiple arc, then G is not strongly connected.
    Otherwise
        For every v ∈ V(G*) do
            Compute a depth first search tree T rooted at v.
            If T has a forward or cross arc, G is not singly connected.
        If no T had a forward or cross arc, G is singly connected.
```

Theorem 2.4 *The algorithm above correctly determines whether or not a digraph is singly connected.*

PROOF: As noted above, terminating negatively if G has a chord is correct. Similarly, Lemma 2.2 demonstrates that terminating negatively if G^* has a multiple arc is correct. If G has no chord and G^* no multiple arc, then Lemma 2.3 proves that G^* is singly connected if and only if G is. By the discussion at the beginning of Section 2 we know that terminating negatively upon discovery of a forward or cross arc in one of the T's in the algorithm is correct. Finally, if no T had a forward or cross arc, Lemma 2.1 states that it is correct to terminate positively. □

3 Chord Testing and Analysis of Running Time

We now show how to test whether or not G contains a chord via a simple depth first search. We believe this fact is already known but can find no mention of it in the literature; rather we see works on related topics such as testing graphs to see if *every* cycle contains a chord (see, e.g., [13]). We therefore include it for completeness.

Let F be a DFS forest rooted at any vertex of G, and for any $v \in V$, let $DFS(v)$ be $v's$ DFS number and $LOW(v)$ its low value in F. The computation of these values is described in [1, 4]. We assume there are no forward or intra-tree cross arcs in F (or we immediately determine that G is not singly connected).

Lemma 3.1 *If F has no forward or intra-tree cross arcs, then G has a chord if and only if there is some vertex v in F with*

- *two back arcs in F,*
- *a back arc and a child x with $LOW(x) < DFS(v)$ in F,*
- *or two distinct children x and y with $LOW(x) < DFS(v)$ and $LOW(y) < DFS(v)$ in F.*

PROOF: ($\Leftarrow$) Given any of the conditions listed in the lemma, there exist two distinct simple paths P and P' from v to its parent $p(v)$ in F. P yields a chord of the cycle formed by catenating P' with the arc $(p(v), v)$.

($\Rightarrow$) Let $C = (c_0, \ldots, c_l)$ be a cycle and $D = (c_i, d_0, \ldots, d_k, c_j)$ for some $0 \le i < j < l$ be a chord of C. We can assume that C is a simple cycle, otherwise we can derive from C and D a different simple cycle and a chord. By definition either (1) $k \ge 0$ and $d_s \ne c_t$ for all $0 \le s \le k$ and $0 \le t < l$, or (2) $k = -1$ and $j > i + 1$. Therefore, D catenated with the path $(c_j, \ldots, c_l, c_1, \ldots, c_i)$ forms another simple cycle C'. Let T be the tree in F containing the vertices of C and D. Since C is simple and T has no forward or cross arcs, C appears in T as only tree arcs and one back arc. The same can be said for C'. Now note that $DFS(c_i) > DFS(c_j)$; i.e., c_i is deeper than c_j in T. This follows from the proof of Lemma 2.1. Either c_i is the deepest vertex of C in T, in which case there is a back arc from c_i to the highest vertex of C in T, or c_i has a child $x \in V(C)$ in T with $LOW(x) < DFS(c_i)$. By the same reasoning, c_i has a back arc associated with C' or a child in T associated with C' with low value less than $DFS(c_i)$. In all cases, since C and C' are distinct, the back arc(s) and child(ren) rendered satisfy the lemma. □

We now turn to the analysis of the running time of the algorithm. The key to the $O(m/n)$ factor improvement over the naive algorithm is that contracting the strongly connected components in G reduces the cost of performing each of the depth first searches in the last stage of the algorithm from $O(m)$ to $O(n)$.

Theorem 3.2 *The algorithm of Section 2 takes $O(n^2)$ time and $O(m)$ space.*

PROOF: Lemma 3.1 describes how to test to see if G has a chord in $O(m)$ time via one complete DFS of G. Note that if we discover a forward or intra-tree cross arc during the DFS, we immediately terminate negatively. Computing the strongly connected components of G also takes $O(m)$ time [1, 4]. Contracting the strongly connected components can be accomplished in $O(m)$ time as follows. Let $S(v)$ denote the SCC containing vertex v in G. $V(G^*) = \bigcup_{v \in V(G)} \{S(v)\}$. For each arc (x, y) in $A(G)$ add an arc $(S(x), S(y))$ to $A(G^*)$ if $S(x) \neq S(y)$ (if $S(x) = S(y)$, this is loop arc and can be ignored.) The test for multiple arcs is easily done simultaneously with the construction. Since G^* is acyclic, there can be no back arcs in any of the DFS trees rooted at vertices in G^*. Therefore, each of the $O(n)$ DFS's requires $O(n)$ time.

Each step requires linear space. The $O(n)$ DFS's in the last stage of the algorithm can reuse the constant space for DFS marks in each vertex. □

4 Conclusion and Open Problems

The problem of determining whether or not a digraph is singly connected appears in [4]. To our knowledge, the algorithm presented here is the first to better the naive bound. Still, the $O(n^2)$ time bound achieved leaves much room for improvement.

We note one aspect of this problem that distinguishes it from more traditional connectivity problems. Problems such as k-vertex- and k-edge-connectivity are monotonic in that once a graph is so connected, adding additional edges does not destroy the property. For singly connected digraphs, this characteristic is reversed: Adding arcs can destroy the single connectivity of a digraph, but removing arcs can create single connectivity, both actions monotonically so.

One can consider a digraph as a network satisfying a set of communication requirements, i.e. pairs of vertices that must be connected by some path. Viewing the transitive closure of a digraph as a similar set of pairs, a digraph satisfies the communications requirements given as a set C if and only if its transitive closure T is such that $C \subseteq T$. Given a digraph with transitive closure T, it is always possible to create another digraph with transitive closure $T' \supseteq T$ but with the property that it is singly connected (it is not always possible to do this and have $T' = T$.) An open problem, then, is to construct efficiently a singly connected digraph satisfying some communications requirements. One might also wish to minimize (or minimalize) $T \backslash C$. This type of problem is akin to traditional connectivity *augmentation* problems; see, e.g., [3, 6, 9, 10, 18, 19, 23].

Finally, we note that the problems of k-connectivity in undirected graphs have their limited connectivity analogues: Given a digraph, are there at most k distinct simple paths connecting any pair of vertices? Further, these higher degree problems are no longer trivial in the undirected case.

5 Acknowledgement

We thank Bob Tarjan for insightful comments on an earlier draft of this paper which led to an improved presentation of the algorithm.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Co., Reading, MA, 1976.

[2] C. Berge. *Graphs and Hypergraphs.* North-Holland Publishing Co., Amsterdam, 1973.

[3] G.-R. Cai and Y.-G. Sur. The minimum augmentation of any graph to a k-edge-connected graph. *Networks*, 19(1):151–72, 1989.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1991.

[5] A. H. Esfahanian and S. L. Hakimi. On computing the connectivities of graphs and digraphs. *Networks*, 14(2):355–66, 1984.

[6] K. P. Eswaran and R. E. Tarjan. Augmentation problems. *SIAM Journal on Computing*, 5(4):653–65, 1976.

[7] S. Even. An algorithm for determining whether the connectivity of a graph is at least k. *SIAM Journal on Computing*, 4(3):393–6, 1975.

[8] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4(4):507–18, 1975.

[9] A. Frank. Augmenting graphs to meet edge-connectivity requirements. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 708–18, 1990.

[10] H. N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 812–21, 1991.

[11] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 112–22, 1991.

[12] Z. Galil. Finding the vertex connectivity of graphs. *SIAM Journal on Computing*, 9(1):197–9, 1980.

[13] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[14] J. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–58, 1973.

[15] A. Kanevsky and V. Ramachandran. Improved algorithms for graph four-connectivity. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 252–9, 1987.

[16] Y. Mansour and B. Schieber. Finding the edge connectivity of directed graphs. *Journal of Algorithms*, 10(1):76–85, 1989.

[17] D. W. Matula. Determining edge connectivity in $O(nm)$. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 249–51, 1987.

[18] D. Naor, D. Gusfield, and C. Martel. A fast algorithm for optimally increasing the edge-connectivity. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 698–707, 1990.

[19] A. Rosenthal and A. Goldner. Smallest augmentations to biconnect a graph. *SIAM Journal on Computing*, 6(1):55–66, 1977.

[20] C. P. Schnorr. Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM Journal on Computing*, 8(2):265–74, 1979.

[21] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–59, 1972.

[22] R. E. Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[23] T. Watanbe and A. Nakamura. Edge-connectivity augmentation problems. *Journal of Computer and System Sciences*, 35(1):96–144, 1987.