Simple DFS on the Complement of a Graph and on Partially Complemented Digraphs

Benson Joeris

Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON N2L 3G1, Canada

Nathan Lindzey

Department of Mathematics Colorado State University, Fort Collins, CO 80521

Ross M. McConnell

Department of Computer Science, Colorado State University, Fort Collins, CO 80521

Nissa Osheim

Department of Computer Science, Colorado State University, Fort Collins, CO 80521

Abstract

A *complementation operation* on a vertex of a digraph changes all outgoing arcs into non-arcs, and outgoing non-arcs into arcs. Given a digraph G, a *partially complemented digraph* $\widetilde{G}$ is a digraph obtained from G by performing a sequence of vertex complement operations on G. Dahlhaus et al. showed that, given an adjacency-list representation of $\widetilde{G}$, depth-first search (DFS) on G can be performed in $O(n+\widetilde{m})$ time, where n is the number of vertices and $\widetilde{m}$ is the number of edges in $\widetilde{G}$. This can be used for finding a depth-first spanning forest and the strongly connected components of the complement of G in time that is linear in the size of G, and Dahlhaus et al. gives applications to finding the modular decomposition of an undirected graph that require that some adjacency lists be complemented and others not. To achieve this bound, their algorithm makes use of a somewhat complicated stack-like data structure to simulate the recursion stack, instead of implementing it directly as a recursive algorithm. We give a recursive $O(n+\widetilde{m})$ algorithm that requires no such data-structures.

1. Introduction

A *complementation operation* on a vertex of a digraph changes all outgoing arcs into non-arcs, and outgoing non-arcs into arcs. A *partially complemented representation* $\widetilde{G}$ of G is a digraph obtained from a sequence of vertex complement operations, starting with a digraph G, and marking the vertices that have been complemented. Let n denote the number of vertices and m denote the number of edges of G, and let $\widetilde{m}$ denote the number of edges of $\widetilde{G}$. In [2], it is shown how to run several algorithms on G, given $\widetilde{G}$, in $O(n + \widetilde{m})$ time. This can be sublinear in the size of G.

Their algorithm for DFS on partially complemented digraphs was notably more complicated than their algorithm for BFS despite the comparable simplicity of DFS and BFS in the usual context.

Their algorithm for DFS is not recursive and is complicated by the use of so-called *complement stacks*, a stack-like data structure used to simultaneously simulate the recursion stack and keep track of which undiscovered vertices will not be called from which vertices on the recursion stack. This raised the question of whether there exists a more natural recursive DFS algorithm for partially complemented digraphs. To this end, we give an elementary recursive $O(n+\widetilde{m})$ algorithm for performing depth-first search on G, given $\widetilde{G}$.

A notable special case is when every vertex is complemented, that is, $\widetilde{G} = \overline{G}$ where $\overline{G}$ denotes the complement of G. Algorithms for performing DFS on G, given $\overline{G}$, have also been developed [6, 7], the most efficient of which runs in $O(n + \overline{m})$ time where n and $\overline{m}$ is the number of vertices and number edges of $\overline{G}$ respectively. To achieve this bound, the algorithm in [6] makes use of the Gabow-Tarjan disjoint set data structure [4]. Our algorithm gives a more direct approach.

2. An Application

A *module* of an undirected graph is a set Y of vertices such that every vertex not in Y is either a neighbor of all vertices in Y or a neighbor of none of them. Examples are $\{a, x\}$ and $\{a, b, c, d, x\}$ in the graph in part 1 of Figure 1. There is a canonical recursive decomposition of every undirected graph, called the *modular decomposition*, which implicitly represents all modules of a graph. This decomposition was first discovered by Gallai [5], and has many applications to combinatorial algorithms on graphs. An $O(n^2)$ algorithm for computing it is given in [3]. A much more complicated $O(n + m)$ algorithm was later given in [8].

The algorithm of [3] chooses a vertex x, finds the set $\mathcal{M}$ of maximal modules that do not contain x, selects x and one vertex from each member of $\mathcal{M}$, forming an induced subgraph $G' = (V', E')$. It is not hard to show that all modules of G' of size greater than one contain x; they correspond to the ancestors of x in the modular decomposition tree. It then recursively finds the modular decomposition of the subgraph induced by M, for each $M \in \mathcal{M}$, to find the subtrees rooted at siblings of ancestors in the modular decomposition, completing the modular decomposition of G.

The only bottleneck to an $O(m \log n)$ bound for this simple approach is the problem of finding the modules of of G' that properly contain x. Since they correspond to ancestors of x, we know that they are ordered by containment. Let $(Y_1, Y_2, \ldots, Y_k)$ be this ordering, that is, $\{Y_1, Y_2, \ldots, Y_k\}$ are the modules of G' that contain x, and for each $i \in \{1, 2, \ldots, k-1\}$, $Y_i \subset Y_{i+1}$. An example of such a G' is the one depicted in part 1 of Figure 1, and $Y_1 = \{x, a\}$, $Y_2 = \{x, a, b, c, d\}$, $Y_3 = \{x, a, b, c, d, e, f, g, h\}$ and Y_4 is the set of vertices of G'. These are depicted in Part 5 of the figure.

To find these, the algorithm of [3] constructs a directed a graph $D(G') = (V' \setminus \{x\}, A)$, whose vertices are the vertices of G' other than x, and where there is a directed edge from vertex y to vertex z if and only if y is adjacent to exactly one of x and z in G'. Part 2 of Figure 1 depicts $D(G')$ for the example graph G'. The following observation follows from the definition of $D(G')$: *A set Y of vertices of G' is a module if and only if $Y \setminus \{x\}$ has no incoming directed edge from $V' \setminus Y$ in $D(G')$.* A set Y is a module if and only if no vertex outside of Y fails to have a uniform relationship to members of Y, and this happens if and only if it has the same relationship to x as it does to all other members of Y, hence if and only if the condition of the observation applies.

From this observation, it is immediate that for each $i \in \{1, 2, \ldots, k-1\}$, $Y_{i+1} \setminus Y_i$ is a strongly connected component of $D(G')$. This is depicted in part 3 of the figure. Also, because these modules are ordered by containment, it is immediate from the observation that there must be a unique topological sort of the strong component graph of $D(G')$, giving the sets $Y_{i+1} \setminus Y_i$ in ascending order of i. This is depicted in part 4 of the figure. Since finding a topological sort of the strongly connected component graph of a digraph can be accomplished by calls to DFS, the problem of finding $(Y_1, Y_2, \ldots, Y_k)$ reduces to DFS on $D(G')$.

The difficulty for the time bound is that $D(G')$ can be much larger than G'. Because of this, the best time bound that could be stated for the algorithm at the time of publication of [3] was $O(n^2)$. However, using the results of this paper, we may obtain the strongly-connected components of $D(G')$ in $O(n' + m')$ time, where n' and m' are the number of vertices and edges of $G' - x$, respectively, giving an $O(n + m)$ bound on time spent on this step over all recursive calls, as follows. *The key insight is that $D(G')$ is in the the complement equivalence class of $G' - x$.* Those vertices that are non-neighbors of x retain their adjacency list from $G' - x$ in $D(G')$, while those that are neighbors of x receive the complements of their adjacency lists from $G' - x$ in $D(G')$. By the the DFS algorithm we describe below, we may therefore perform calls to DFS on $D(G')$ until all vertices have been visited, in $O(n' + m')$ time.

An algorithm for finding strongly-connected components, due to Tarjan [9], is described as a textbook algorithm in [1]. The algorithm is obtained by adding extra steps to DFS. Each vertex is labeled with a discovery time, indicating when a call is first made on it and pushed on a stack. An invariant is thus that the elements on the stack appear in ascending order of discovery time, from bottom to top. Another invariant is that a vertex remains on the stack at least until the time the call to DFS on it has finished, and that it is never pushed more than once. When a call on a vertex v finishes, it is labeled with a `lowlink` label, which is equal to the minimum of its own discovery time, the `lowlink` labels of its children and the discovery times of neighbors on the stack. If its `lowlink` label is equal to its discovery time, the stack is popped down through v, giving a strongly-connected component.

If a partially complemented representation $\widetilde{G}$ of a digraph G is used, then

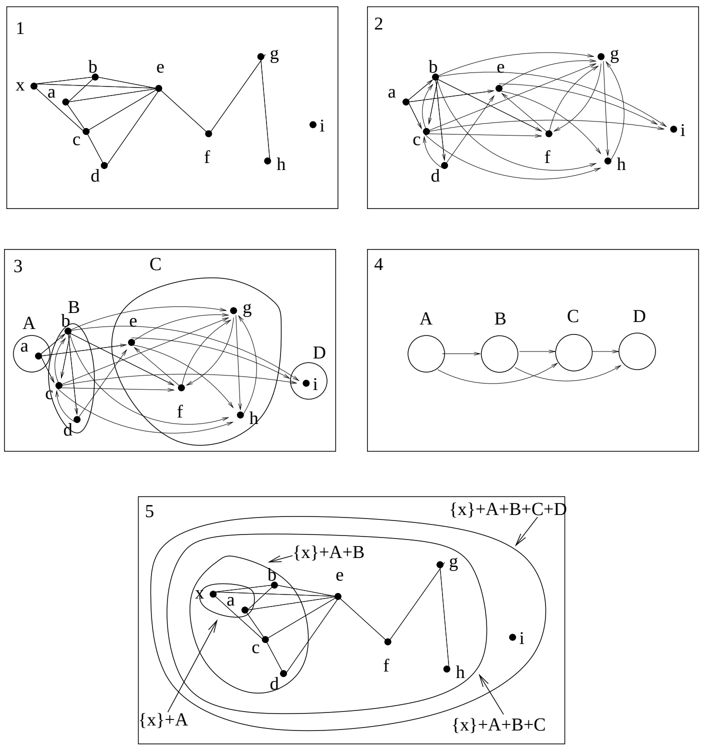

Figure 1: Finding the modules that contain x in a graph where these modules are ordered by containment.

the time spent pushing to and popping from the stack is $O(n)$, since each vertex is pushed only once. The time spent examining the `lowlink` labels of children is $O(n)$, since the DFS forest has $O(n)$ edges.

It remains to bound the cost of finding the minimum discovery time of neighbors of v in G on the stack. If v is not complemented in $\widetilde{G}$, it takes $O(|\widetilde{N}(v)|)$ time to find the minimum discovery time of vertices on the stack, by examining the set $\widetilde{N}(v)$ of neighbors of v and determining whether they are on the stack. If v is complemented, this step can be simulated by marking the set $\widetilde{N}(v)$ of non-neighbors of v in G and traversing the stack from bottom to top until an unmarked vertex w is found, then unmarking $\widetilde{N}(v)$. Because the discovery times are increasing from the bottom to the top of the stack, w has the minimum discovery time of any neighbor on the stack, and it takes $O(|\widetilde{N}(v)|)$ time to find in this case also. This gives an $O(n + \widetilde{m})$ time bound for strongly connected components of a digraph, where $\widetilde{m}$ is the number of edges in $\widetilde{G}$.

Therefore, for the problem of [3], it takes time linear in the size of G' to find the modules of G', since G' serves in the role of $\widetilde{G}$ for $D(G)$. Since the step of finding the strongly-connected components is no longer the bottleneck operation over all recursive calls, this gives an $O(n+m\log n)$ bound for the algorithm of [3]. Using a variant of this idea, [2] obtains a linear-time algorithm for modular decomposition that is much simpler than that of [8], and the algorithm we describe below can be used in place of the implementation of DFS that it uses.

3. Preliminaries

We will assume that the vertices are numbered 1 through n. For vertices u and v, let $u < v$ denote that the vertex number of u is smaller than that of v.

Let $\widetilde{N}(v)$ denote the neighbors of v in $\widetilde{G}$. That is, if v is uncomplemented, $\widetilde{N}(v)$ is a list of neighbors of v in G, and if v is complemented, it is a list of non-neighbors of v in G. We are given $\widetilde{N}(v)$ for each vertex v. We assume that $\widetilde{N}(v)$ is given in a doubly-linked list, sorted by vertex number. This ordering can be achieved in $O(n + \widetilde{m})$ time by assigning an order to the vertices by numbering them arbitrarily from 1 to n and then lexicographically sorting the set $\{(v, w)|v \in V \text{ and } w \in \widetilde{N}(v)\}$, by radix sorting [1], since the set has $O(m)$ elements and the members of each pair are integers from 1 to n. Each vertex is labeled with a bit to indicate whether it is complemented, specifying whether $\widetilde{N}(v)$ should be interpreted as neighbors or non-neighbors of v in G. We will find it convenient to assume that $\widetilde{N}(v)$ is terminated by a fictitious vertex whose vertex number, $n + 1$, is larger than those of any vertex in G.

4. The DFS Algorithm

A vertex is *discovered* when a recursive call to DFS is made on it. At all times, we maintain a doubly-linked list U of undiscovered vertices, which is sorted by vertex number. Initially, U contains all vertices of G.

Algorithm 1: DFS(v)

```
Data: A current undiscovered vertex $v$, and a global ordered
      doubly-linked list $U$ of undiscovered vertices
remove($U$,$v$);
if $v$ is uncomplemented then
    for $u \in \widetilde{N}(v)$ do
        if $u$ is undiscovered then
            DFS($u$);
else
    $u_v \longleftarrow$ head($U$);
    $n_v \longleftarrow$ head($\widetilde{N}(v)$);
    while $u_v \neq null$ do
        if $u_v = n_v$ then
            $u_v \longleftarrow$ next($U, u_v$);
            $n_v \longleftarrow$ next($\widetilde{N}(v), n_v$);
        else if $u_v > n_v$ then
            $s \longleftarrow n_v$;
            $n_v \longleftarrow$ next($\widetilde{N}(v), n_v$);
            remove($\widetilde{N}(v), s$);
        else
            DFS($u_v$);
            // restarting step ...
            $w = \mathrm{prev}(\widetilde{N}(v), n_v)$;
            while $w \neq null$ and $w \notin U$ do
                $t = w$;
                $w \longleftarrow$ prev($\widetilde{N}(v), t$);
                remove($\widetilde{N}(v), t$));
            if $w = null$ then
                $u_v \longleftarrow$ head($U$);
            else
                $u_v \longleftarrow$ next($U, w$);
```

When a recursive call is made on an undiscovered vertex v, it is removed from U and marked as discovered. If v is uncomplemented, the algorithm for generating recursive calls from it is exactly what it is in standard DFS: for each $w \in \widetilde{N}(v)$, if w is undiscovered, a recursive call is made on it.

If v is complemented, then the presence of each $w \in \widetilde{N}(v)$ is used to block any recursive call on w from v, since this means that w is not a neighbor of v in G. If w has been discovered, however, its absence from U suffices to block a recursive call on it from v. This allows us to remove w from $\widetilde{N}(v)$ while maintaining the following invariant:

Invariant 1. *All undiscovered non-neighbors in G of each complemented vertex v remain in $\widetilde{N}(v)$.*

The invariant suffices to prevent recursive calls from v on non-neighbors of v. Removal of elements from $\widetilde{N}(v)$ while maintaining this invariant is the key to our time bound, since it may be necessary to traverse an element $w \in \widetilde{N}(v)$ more than once, and we can charge the extra cost of multiple traversals to deletions of vertices from $\widetilde{N}(v)$.

We traverse the lists for $\widetilde{N}(v)$ and U in parallel, in a manner similar to the `merge` operation in `mergesort`, advancing the pointer to the lower-numbered vertex at each step, or advancing both pointers in their lists if they point to the same vertex.

Let n_v be the current vertex in $\widetilde{N}(v)$ and let u_v be the current vertex in U. If $n_v = u_v$, it is a non-neighbor of v, hence we cannot make a recursive call on it from v. We set n_v to its successor in $\widetilde{N}(v)$ and u_v to its successor in U. If $n_v \notin U$, which is detected if $n_v < u_v$, then n_v has already been discovered, and we can advance n_v to the next vertex in $\widetilde{N}(v)$ and remove n_v from $\widetilde{N}(v)$, which respects Invariant 1. If $u_v \notin \widetilde{N}(v)$, which is detected if $u_v < n_v$, we make a recursive call on u_v, and when this recursive call returns, we perform the following *restarting step*:

- Advance u_v to be the first vertex u'_v in U with a higher vertex number than the current u_v.

The difficulty in implementing the restarting step efficiently is that when the recursive call on u_v returns, u_v and possibly many other vertices were discovered and removed from U during the recursive call on it. It is thus not a simple matter of finding the successor of u_v in U. We discuss an efficient implementation below.

By induction on the number of times u_v is advanced, we see that the following invariant is maintained:

Invariant 2. *Whenever u_v advances in U, its predecessors in U are members of $\widetilde{N}(v)$, hence non-neighbors of v.*

The call on v returns when u_v moves past the end of U, which happens before n_v moves past the end of $\widetilde{N}(v)$, due to the presence of the fictitious vertex numbered $n + 1$ at the end of $\widetilde{N}(v)$.

For the correctness, let k be the number of undiscovered vertices when v is discovered. Since the number of undiscovered vertices is less than k when each recursive call is generated from v, we may assume by induction on the number of undiscovered vertices that each recursive call generated from v faithfully executes a DFS, given the marking of vertices as undiscovered or discovered when the call is made. If v is not complemented, the correctness of the call on it is immediate. If v is complemented, the correctness of the call on it follows from Invariant 2 and the fact that a recursive call is made on u_v whenever it is found to be a neighbor of v.

To implement the restarting step, we traverse $\widetilde{N}(v)$ backward, starting at $\texttt{prev}(\widetilde{N}(v), n_v)$, removing vertices from $\widetilde{N}(v)$ that are no longer in U. Their removal does not violate Invariant 1 or Invariant 2.

Eventually, we have either encountered a vertex w that is in both $\widetilde{N}(v)$ and U, or we have deleted all predecessors of u_v in $\widetilde{N}(v)$. Suppose we encounter w. All predecessors of u_v in U were non-neighbors of v when n_v was last advanced. Also, $u_v < n_v$, since we wouldn't have reached u_v to make a recursive call on it if n_v were less than u_v, and we wouldn't have made a recursive call on it if they were equal. Finally, w is now the predecessor of n_v in $\widetilde{N}(v)$ since we have deleted all vertices from n_v back to w. It follows that the successor of w is greater than u_v, hence the successor of w now gives u'_v. By a similar argument, if all predecessors of n_v are deleted from $\widetilde{N}(v)$, the first element of U gives u'_v.

Though the restarting step is not an $O(1)$ operation, the total time required by restarting steps over the entire DFS is $O(n + \widetilde{m})$, since all but $O(1)$ of a restarting operation can be charged to elements that it deletes from some list $\widetilde{N}(v)$, and the initial sum of sizes of these lists is $\widetilde{m}$. Pseudocode of the algorithm is given as Algorithm 1, and makes use of the following $O(1)$ operations:

- `head`(L): returns the head node of a doubly-linked list L.
- `next`(L,n): returns the next node of the node n that exists in L, or null if no such node exists.
- `prev`(L,n): returns the previous node of the node n that exists in L, or null if no such node exists.
- `remove`(L,n): removes a node n from doubly-linked list L.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[2] E. Dahlhaus, J. Gustedt, and R. M. McConnell. Partially complemented representations of digraphs. *Discrete Mathematics & Theoretical Computer Science*, 5(1):147–168, 2002.

[3] A. Ehrenfeucht, H. N. Gabow, R. M. McConnell, and S. J. Sullivan. An $O(n^2)$ divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs. *Journal of Algorithms*, 16:283–294, 1994.

[4] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *STOC*, pages 246–251, 1983.

[5] T. Gallai. Transitiv orientierbare Graphen. *Acta Math. Acad. Sci. Hungar.*, 18:25–66, 1967.

[6] H. Ito and M. Yokoyama. Linear time algorithms for graph search and connectivity determination on complement graphs. *Inf. Process. Lett.*, 66(4):209–213, 1998.

[7] M.-Y. Kao, N. Occhiogrosso, and S.-H. Teng. Simple and efficient graph compression schemes for dense and complement graphs. *J. Comb. Optim.*, 2(4):351–359, 1998.

[8] R. M. McConnell and J. P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.

[9] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.