Graphs and modeling

Formulating a simple, precise specification of a computational problem is often a prerequisite to writing a computer program for solving the problem. Many computational problems are best stated in terms of graphs. A directed graph $G(V,E)$ consists of a finite set of vertices V and a set of (directed) edges or arcs E. An arc is an ordered pair of vertices (v,w) and is usually indicated by drawing a line between v and w, with an arrow pointing towards w. Stated in mathematical terms, a directed graph $G(V,E)$ is just a binary relation $E \subseteq V \times V$ on a finite set V. Undirected graphs may be regarded as special kinds of directed graphs, such that $(u,v) \in E \leftrightarrow (v,u) \in E$. Thus, since the directions of the edges are unimportant, an undirected graph $G(V,E)$ consists of a finite set of vertices V, and a set of edges E, each of which is an unordered pair of vertices $\{u,v\}$.

Graphs model many situations. For example, the vertices of a graph can represent cities, with edges representing highways that connect them. In this case, each edge might also have an associated length. Alternatively, an edge might represent a flight from one city to another, and each edge might have a weight which represents the cost of the flight. A typical problem in this context is to compute shortest paths: given that you wish to travel from city X to city Y, what is the shortest path (or the cheapest flight schedule). We will find very efficient algorithms for solving these problems.

A seemingly similar problem is the traveling salesman problem. Supposing that a traveling salesman wishes to visit each city exactly once and return to his starting point, in what order should he visit the cities to minimize the total distance traveled? Unlike the shortest paths problem, however, this problem has no known efficient algorithm. This is an example of an NP-complete problem, and one we will study towards the end of this course.

A different context in which graphs play a critical modeling role is in networks of pipes or communication links. These can, in general, be modeled by directed graphs with capacities on the edges. A directed edge from u to v with capacity c might represent a cable that can carry a flow of at most c calls per unit time from u to v. A typical problem in this context is the max-flow problem: given a communications network modeled by a directed graph with capacities on the edges, and two special vertices — a source s and a sink t — what is the maximum rate at which calls from s to t can be made? There are ingenious techniques for solving these types of flow problems.

In all the cases mentioned above, the vertices and edges of the graph represented something quite concrete such as cities and highways. Often, graphs will be used to represent more abstract relationships. For example, the vertices of a graph might represent tasks, and the edges might represent precedence constraints: a directed edge from u to v

says that task u must be completed before v can be started. An important problem in this context is scheduling: in what order should the tasks be scheduled so that all the precedence constraints are satisfied. There are extremely fast algorithms for this problem that we will see shortly.

Representing graphs on the computer

One common representation for a graph $G(V,E)$ is the *adjacency matrix*. Suppose $V = \{1,\cdots,n\}$. The adjacency matrix for $G(V,E)$ is an $n \times n$ matrix A, where $a_{i,j} = 1$ if $(i,j) \in E$ and $a_{i,j} = 0$ otherwise.[1] The advantage of the adjacency matrix representation is that it takes constant time (just one memory access) to determine whether or not there is an edge between any two given vertices. In the case that each edge has an associated length or a weight, the adjacency matrix representation can be appropriately modified so entry $a_{i,j}$ contains that length or weight instead of just a 1. The disadvantage of the adjacency matrix representation is that it requires $\Omega(n^2)$ storage, even if the graph has as few as $O(n)$ edges. Moreover, just examining all the entries of the matrix would require $\Omega(n^2)$ steps, thus precluding the possibility of linear time algorithms for graphs with $o(n^2)$ edges (at least in cases where all the matrix entries must be examined).

An alternative representation for a graph $G(V,E)$ is the *adjacency list* representation. We say that a vertex j is adjacent to a vertex i if $(i,j) \in E$. The adjacency list for a vertex i is a list of all the vertices adjacent to i (in any order). To represent the graph, we use an array of size n to represent the vertices of the graph, and the i^{th} element of the array points to the adjacency list of the i^{th} vertex. The total storage used by an adjacency list representation of a graph with n vertices and m edges is $O(n+m)$. The adjacency list representation hence avoids the disadvantage of using more space than necessary. We will use this representation for all our graph algorithms that take linear or near linear time. A disadvantage of adjacency lists, however, is that determining whether there is an edge from vertex i to vertex j may take as many as n steps, since there is no systematic shortcut to scanning the adjacency list of vertex i. For applications where determining if there is an edge between two vertices is the bottleneck, the adjacency matrix is thus preferable.

Depth first search

There are two fundamental algorithms for searching a graph: depth first search and breadth first search. To better understand the need for these procedures, let us imagine the computer's view of a graph that has been input

[1] Generally, we use either n or $|V|$ for the number of nodes in a graph, and m or $|E|$ for the number of edges.

into it, in the adjacency list representation. The computer's view is fundamentally *local* to a specific vertex: it can examine each of the edges adjacent to a vertex in turn, by traversing its adjacency list; it can also mark vertices as visited. One way to think of these operations is to imagine exploring a dark maze with a flashlight and a piece of chalk. You are allowed to illuminate any corridor of the maze emanating from your current position, and you are also allowed to use the chalk to mark your current location in the maze as having been visited. The question is how to find your way around the maze.

We now show how the depth first search allows the computer to find its way around the input graph using just these primitives. (We will examine breadth first search shortly.)

Depth first search is technique for exploring a graph using a stack as the basic data structure. We start by defining a recursive procedure search (the stack is implicit in the recursive calls of search): search is invoked on a vertex v, and explores all previously unexplored vertices reachable from v.

```
Procedure search(v)
    vertex v
    explored(v) := 1
    previsit(v)
    for (v,w) ∈ E
        if explored(w) = 0 then search(w)
    rof
    postvisit(v)
end search


Procedure DFS (G(V,E))
    graph G(V,E)
    for each v ∈ V do
        explored(v) := 0
    rof
    for each v ∈ V do
        if explored(v) = 0 then search(v)
    rof
end DFS
```

By modifying the procedures previsit and postvisit, we can use DFS to solve a number of important problems, as we shall see. It is easy to see that depth first search takes $O(|V|+|E|)$ steps (assuming previsit and postvisit take $O(1)$ time), since it explores from each vertex once, and the exploration involves a constant number of steps per outgoing edge.

The procedure search defines a tree in a natural way: each time that search discovers a new vertex, say w, we can incorporate w into the tree by connecting w to the vertex v it was discovered from via the edge (v,w). The remaining edges of the graph can be classified into three types:

- Forward edges - these go from a vertex to a descendant (other than child) in the DFS tree.
- Back edges - these go from a vertex to an ancestor in the DFS tree.
- Cross edges - these go from "right to left"– there is no ancestral relation.

Question: Explain why if the graph is undirected, there can be no cross edges.

One natural use of previsit and postvisit could each keep a counter that is increased each time one of these routines is accessed; this corresponds naturally to a notion of time. Each routine could assign to each vertex a preorder number (time) and a postorder number (time) based on the counter. If we think of depth first search as using an explicit stack, then the previsit number is assigned when the vertex is first placed on the stack, and the postvisit number is assigned when the vertex is removed from the stack. Note that this implies that the intervals $[preorder(u), postorder(u)]$ and $[preorder(v), postorder(v)]$ are either disjoint, or one contains the other.

An important property of depth-first search is that the contents of the stack at any time yield a path from the root to some vertex in the depth first search tree. (Why?) This allows us to prove the following property of the postorder numbering:

Claim 3.1 *If $(u,v) \in E$ then $postorder(u) < postorder(v) \iff (u,v)$ is a back edge.*

Proof: If $postorder(u) < postorder(v)$ then v must be pushed on the stack before u. Otherwise, the existence of edge (u,v) ensures that v must be pushed onto the stack before u can be popped, resulting in $postorder(v) < postorder(u)$ — contradiction. Furthermore, since v cannot be popped before u, it must still be on the stack when u is pushed on to it. It follows that v is on the path from the root to u in the depth first search tree, and therefore (u,v) is a back edge.

The other direction is trivial. ■

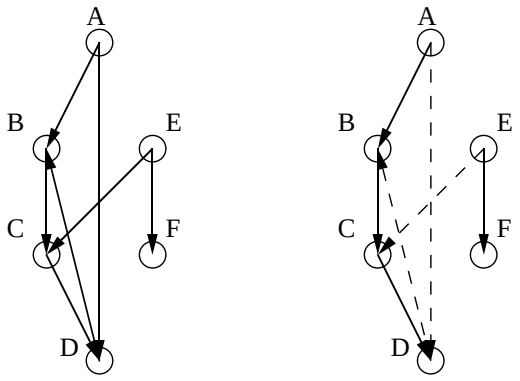
Exercise: What conditions to the preorder and postorder numbers have to satisfy if (u,v) is a forward edge? A cross edge?

Claim 3.2 *$G(V,E)$ has a cycle iff the DFS of $G(V,E)$ yields a back edge.*

Proof: If (u,v) is a back edge, then (u,v) together with the path from v to u in the depth first tree form a cycle.

Conversely, for any cycle in $G(V,E)$, consider the vertex assigned the smallest postorder number. Then the edge leaving this vertex in the cycle must be a back edge by Claim 3.1, since it goes from a lower postorder number to a higher postorder number. ■

Application of DFS: Topological sort



Graph is explored in preorder ABCDEF.
Postorder is DCBAFE.
DB is a back edge.
AD is a forward edge.
EC is a cross edge.

Figure 3.1: A sample depth-first search.

We now suggest an algorithm for the scheduling problem described previously. Given a directed graph $G(V,E)$, whose vertices $V = \{v_1, \ldots v_n\}$ represent tasks, and whose edges represent precedence constraints: a directed edge from u to v says that task u must be completed before v can be started. The problem of topological sorting asks: in what order should the tasks be scheduled so that all the precedence constraints are satisfied.

Note: The graph must be acyclic for this to be possible. (Why?) Directed acyclic graphs appear so frequently they are commonly referred to as DAGs.

Claim 3.3 *If the tasks are scheduled by decreasing postorder number, then all precedence constraints are satisfied.*

Proof: If G is acyclic then the DFS of G produces no back edges by Claim 3.2. Therefore by Claim 3.1, $(u,v) \in G$ implies $postorder(u) > postorder(v)$. So, if we process the tasks in decreasing order by postorder number, when task v is processed, all tasks with precedence constraints into v (and therefore higher postorder numbers) must already have been processed. ■

There's another way to think about topologically sorting a DAG. Each DAG has a *source*, which is a vertex with no incoming edges. Similarly, each DAG has a *sink*, which is a vertex with no outgoing edges. (Proving this is an exercise.) Another way to topologically order the vertices of a DAG is to repeatedly output a source, remove it from the graph, and repeat until the graph is empty. Why does this work? Similarly, once could repeatedly output sinks, and this gives the reverse of a valid topological order. Again, why?

Strongly Connected Components

Connectivity in undirected graphs is rather straightforward. A graph that is not connected can naturally be decomposed into several connected components (Figure 3.2). DFS does this handily: each restart of DFS marks a new connected component.

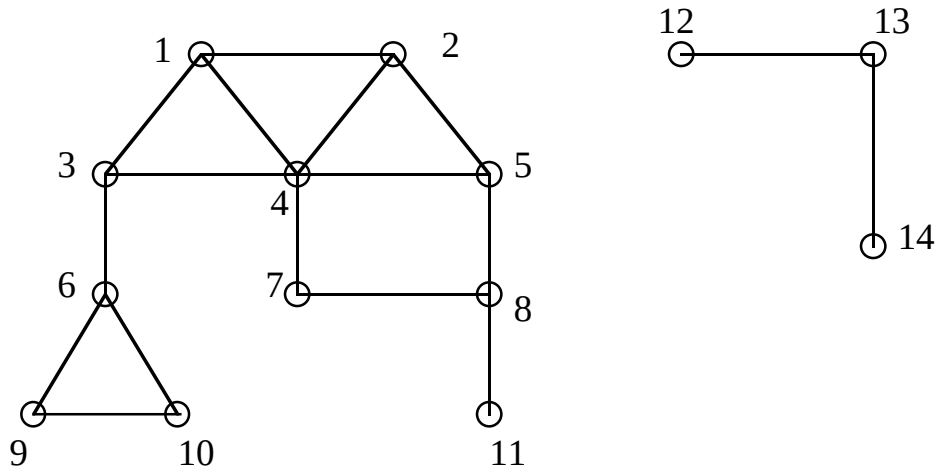


Figure 3.2: An undirected graph

In directed graphs, what connectivity means is more subtle. In some primitive sense, the directed graph in Figure 3.3 appears connected, since if it were an undirected graph, it would be connected. But there is no path from vertex 12 to 6, or from 6 to 1, so saying the graph is connected would be misleading.

We must begin with a meaningful definition of connectivity in directed graphs. Call two vertices u and v of a directed graph $G = (V,E)$ *connected* if there is a path from u to v, and one from v to u. This relation between vertices is reflexive, symmetric, and transitive (check!), so it is an *equivalence relation* on the vertices. As such, it partitions V into disjoint sets, called the *strongly connected components (SCC's)* of the graph (in Figure 3.3 there are four SCC's). Within a strongly connected component, every pair of vertices are connected.

We now imagine shrinking each SCC into a vertex (a supervertex), and draw an edge (a superedge) from SCC X to SCC Y if there is at least one edge from a vertex in X to a vertex in Y. The resulting directed graph has to be a directed acyclic graph (DAG) – that is to say, it can have no cycles (see Figure 3.3). The reason is simple: a cycle containing several SCC's would merge to a single SCC, since there would be a path between every pair of vertices in the SCC's of the cycle. Hence, every directed graph is a DAG of its SCC's.

This important decomposition theorem allows one to think of connectivity information of a directed graph in two levels. At the top level we have a DAG, which has a useful, simple structure. For example, as we have mentioned before, a DAG is guaranteed to have at least one *source* (a vertex without incoming edges) and a *sink*

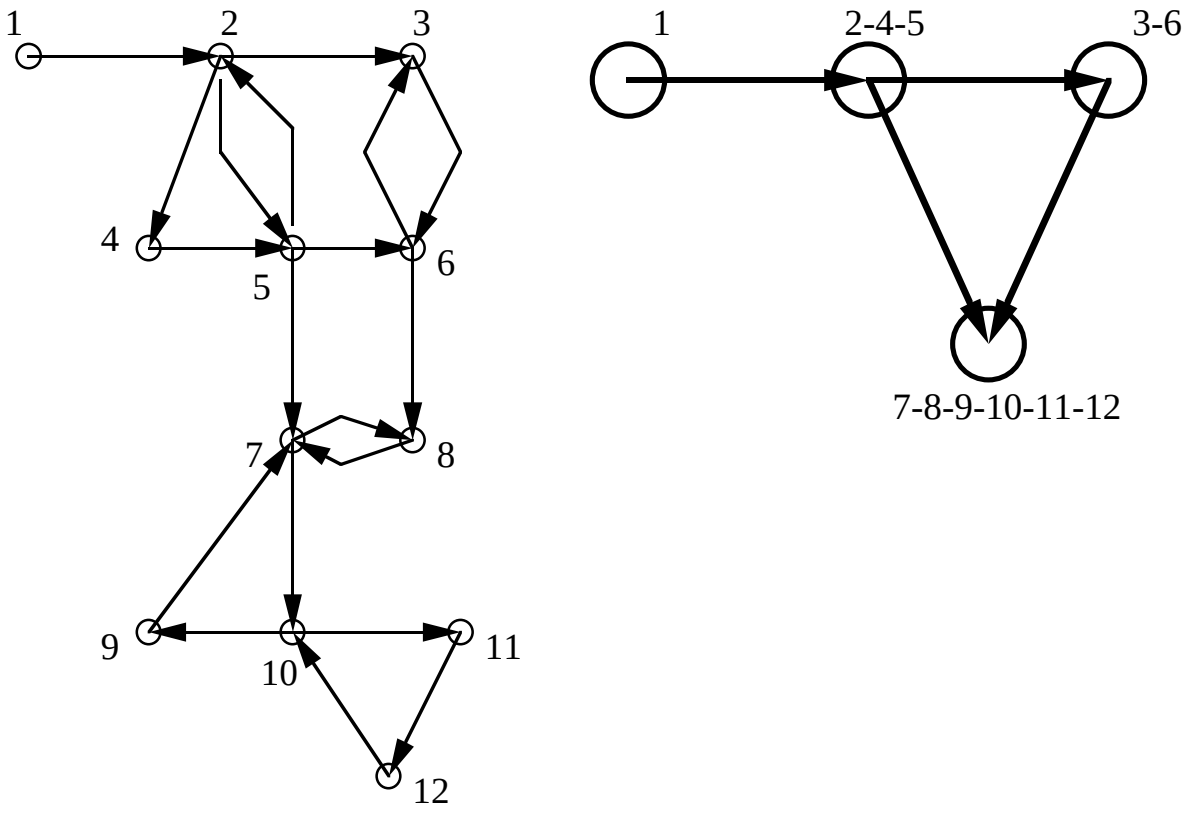


Figure 3.3: A directed graph and its SCC's

(a vertex without outgoing edges). If we want more details, we could look inside a vertex of the DAG to see the full-fledged SCC —a completely connected graph— that lies there.

This decomposition is extremely useful and informative; it is thus very fortunate that we have a very efficient algorithm, based on DFS, that finds the strongly connected components *in linear time!* We motivate this algorithm next. It is based on several interesting and slightly subtle properties of DFS:

Property 1: If DFS is started at a vertex *v*, then it will get stuck and restarted precisely when all vertices in the SCC of *v*, and in all the SCC's that are reachable from the SCC of *v*, are visited. Consequently, if DFS is started at a vertex of a sink SCC (a SCC that has no edges leaving it in the DAG of SCC's), then it will get stuck after it visits precisely the vertices of this SCC.

For example, if DFS is started at vertex 11 in Figure 3.3 (a vertex in the only sink SCC in this graph), then it will visit the six vertices in the sink SCC before getting stuck: vertices 12, 10, 9, 7, 8. Property 1 suggests a way of starting a decomposition algorithm, by finding the first SCC: start DFS from a vertex in a sink SCC, and, when stuck, output the vertices that have been visited. They form an SCC!

Of course, this leaves us with two problems: (A) How to guess a vertex in a sink SCC, and (B) how to continue our algorithm by outputting the next SCC, and so on.

Let us first face Problem (A). It turns out that it will be easier not to look for vertices in a sink SCC, but instead look for vertices in a *source* SCC. In particular:

Property 2: The vertex with the highest postorder number in DFS (that is, the vertex where the DFS ends) belongs to a source SCC.

The proof is by contradiction. If Property 2 were not not true, and v is the vertex with the highest post-order number, then there would be an incoming edge (u,w) with u not in the SCC of v and w in the SCC of v. If u were searched before v, then u clearly has a higher postorder number. If u were searched after v, then since u does not lie in v's SCC, it must not be searched until v is popped from the search stack, so again u must have a higher postorder number than v.

The reason behind Property 2 is thus not hard to see: if there is an SCC "above" the SCC of the vertex where the DFS ends, then the DFS should have ended in that SCC (reaching it either by restarting or by backtracking).

Property 2 provides an indirect solution to Problem (A). Consider a graph G and the *reverse* graph G^R —G with the directions of all edges reversed. G^R has precisely the same SCC's as G (why?). So, if we make a DFS in G^R, then the vertex where we end (the one with the highest post-order) belongs to a source SCC of G^R —that is to say, a sink SCC of G. We have solved Problem (A).

Onwards to Problem (B). How does the algorithm continue after the first sink component is output? The solution is clear: delete the SCC just output from G^R, and make another DFS in the remaining graph. The only problem is, this would be a quadratic, not linear, algorithm, since we would run an $O(m)$ DFS algorithm for up to each or $O(n)$ vertices. How can we avoid this extra work? The key observation here is that we do not have to make a new DFS in the remaining graph:

Property 3: If we make a DFS in a directed graph, and then delete a source SCC of this graph, what remains is a DFS in the remaining graph (the pre-order and post-order numbers may now not be consecutive, but they will be of the right relative magnitude).

This is also easy to justify. We just imagine two runs of the DFS algorithm, one with and one without the source SCC. Consider a transcript recording the steps of the DFS algorithm. It is easy to see that the transcript of both runs would be the same (assuming they both made the same choices of what edges to follow at what points), except where the the first went through the source SCC.

Property 3 allows us to use induction to continue our SCC algorithm. After we output the first SCC, we can use the same DFS information from G^R to output the second SCC, the third SCC, and so on. The full algorithm can thus

be described as follows:

Step 1: Perform DFS on G^R.

Step 2: Perform DFS on G, processing unsearched vertices in the order of decreasing postorder numbers from the DFS of Step 1. At the beginning and every restart print "*New SCC:*" When visiting vertex v, print v.

This algorithm is linear-time, since the total work is really just two depth-first searches, each of which is linear time. **Question:** (How does one construct G^R from G?) If we run this algorithm on Figure 3.3, Step 1 yields the following order on the vertices (decreasing postorder in G^R's DFS): 7, 9, 10, 12, 11, 8, 3, 6, 2, 5, 4, 1. Step 2 now produces the following output: New SCC: 7, 8, 10, 9, 11, 12, New SCC: 3, 6, New SCC: 2, 4, 5, New SCC: 1.

Incidentally, there *is* more sophisticated connectivity information that one can derive from undirected graphs. An *articulation point* is a vertex whose deletion increases the number of connected components in the undirected graph. In Figure 3.2 there are 4 articulation points: 3, 6, 8, and 13. Articulation points divide the graph into *biconnected components* (the pieces of the graph between articulation points) and *bridge edges*. Biconnected components are maximal edge sets (of at least 2 edges) such that any two edges on the set lie on a common cycle. For example, the large connected component of the graph in Figure 3.2 contains the biconnected components on edges between vertices 1-2-3-4-5-7-8 and 6-9-10. The remaining edges are 3-6 and 8-11 are bridge edges; they disconnect the graph. Not coincidentally, this more sophisticated and subtle connectivity information can also be captured by DFS.

Putting in Into Practice

Suppose you are debugging your latest huge software program for a major industrial client. The program has hundreds of procedures, each of which must be carefully tested for bugs.

You realize that, to save yourself some work, it would be best to analyze the procedures in a particular order. For instance, if procedure Write_Check() calls Get_Check_Number(), you would probably want to test Get_Check_Number() first. That way, when you look for the bugs in Write_Check(), you do not have to worry about checking (or re-checking) Get_Check_Number(). (Let's ignore the specious argument that if there are no bugs, you might avoid testing and debugging Get_Check_Number() altogether by starting with Write_Check().)

You can easily generate a list of what procedures each procedure calls with a single pass through the code. So here's the problem: given your program, determine what schedule you should give your testing and debugging team, so that a procedure will be debugged only after anything it calls will be debugged.

Go through the program, creating one vertex for each procedure. Introduce a directed edge from vertex A to vertex B if the procedure A calls B. This directed edge represents the fact that A must be debugged before B. We call this graph the *procedure graph*. If this graph is acyclic, then the topological sort will give you a valid ordering for

the debugging.

What if the graph is not acyclic? Then your program uses *mutual recursion*; that is, there is some chain of procedures through which a procedure might end up calling itself. For example, this would be the case if procedure A calls procedure B, procedure B calls procedure C, and procedure C calls procedure A. A topological sort will detect these cycles, but what we really want is a list of them, since instances of mutual recursion are harder to test and debug.

In this case, we should use the strongly connected components algorithm on the procedure graph. The SCC algorithm will find all the cycles, showing all instances of mutual recursion. Moreover, if we collapse the cycles in the graph, so that instances of mutual recursion are treated as one large super-procedure, then the SCC algorithm will provide a valid debugging ordering for all the procedures in this modified graph. That is, the SCC algorithm will topologically sort the underlying SCC DAG.