1 Strongly Connected Components

Connectivity in undirected graphs is rather straightforward: A graph that is not connected is naturally and obviously decomposed in several *connected components* (Figure 1). As we have seen, depth-first search does this handily: Each restart of the algorithm marks a new connected component.

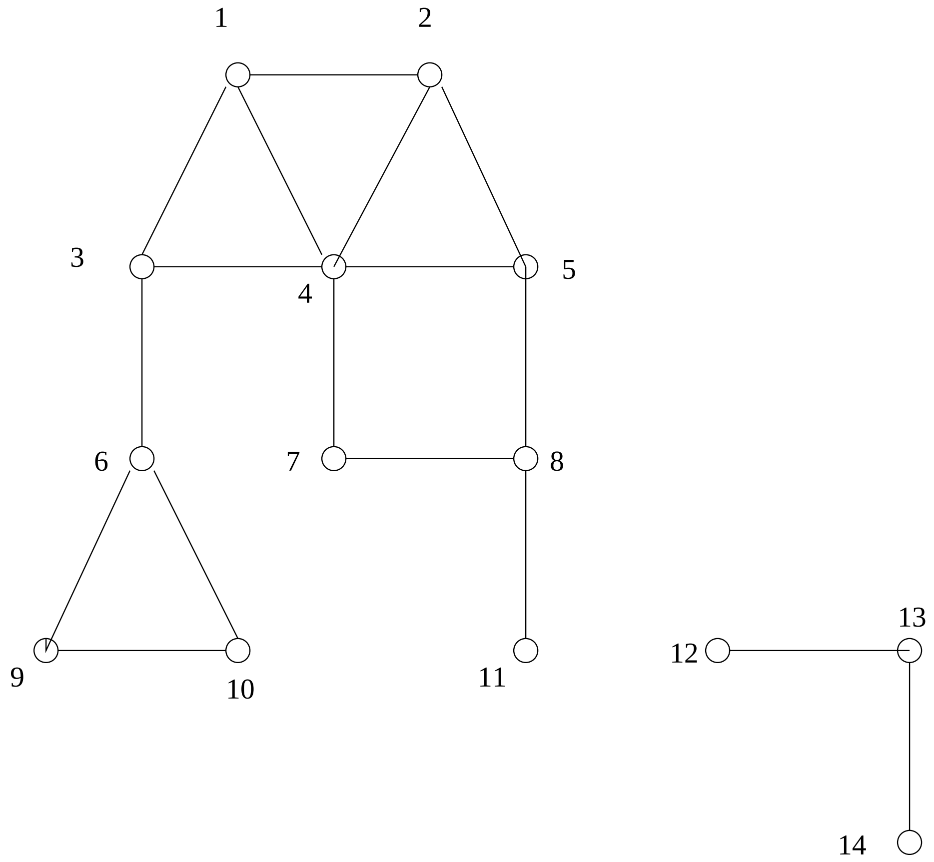

Figure 1: An undirected graph

In *directed graphs*, however, connectivity is more subtle. In some primitive sense, the directed graph in Figure 2 is "connected" (no part of it can be "pulled apart," so to speak, without "breaking" any edges). But this notion is hardly interesting or informative. This graph is certainly not connected, in that there is no path from node 12 to 6, or from 6 to 1. The only meaningful way to define connectivity in directed graphs is this:

Call two nodes 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 nodes is reflexive, symmetric, and transitive (check!), so it is an equivalence relation on the nodes. As such, it partitions V into disjoint sets, called the *strongly connected components* of the graph. In the directed graph of Figure 2 there are *four* strongly connected components.

If we shrink each of these strongly connected components down to a single node, and draw an edge between two of them if there is an edge from some node in the first to some node in the second, the resulting directed graph has to be a *directed acyclic graph (dag)* —that is to say, it can have no cycles (see Figure 2(b)). The reason is simple: A cycle containing several strongly connected components would merge them all to a single strongly connected component. We can restate this observation as follows:

Every directed graph is a dag of its strongly connected components.

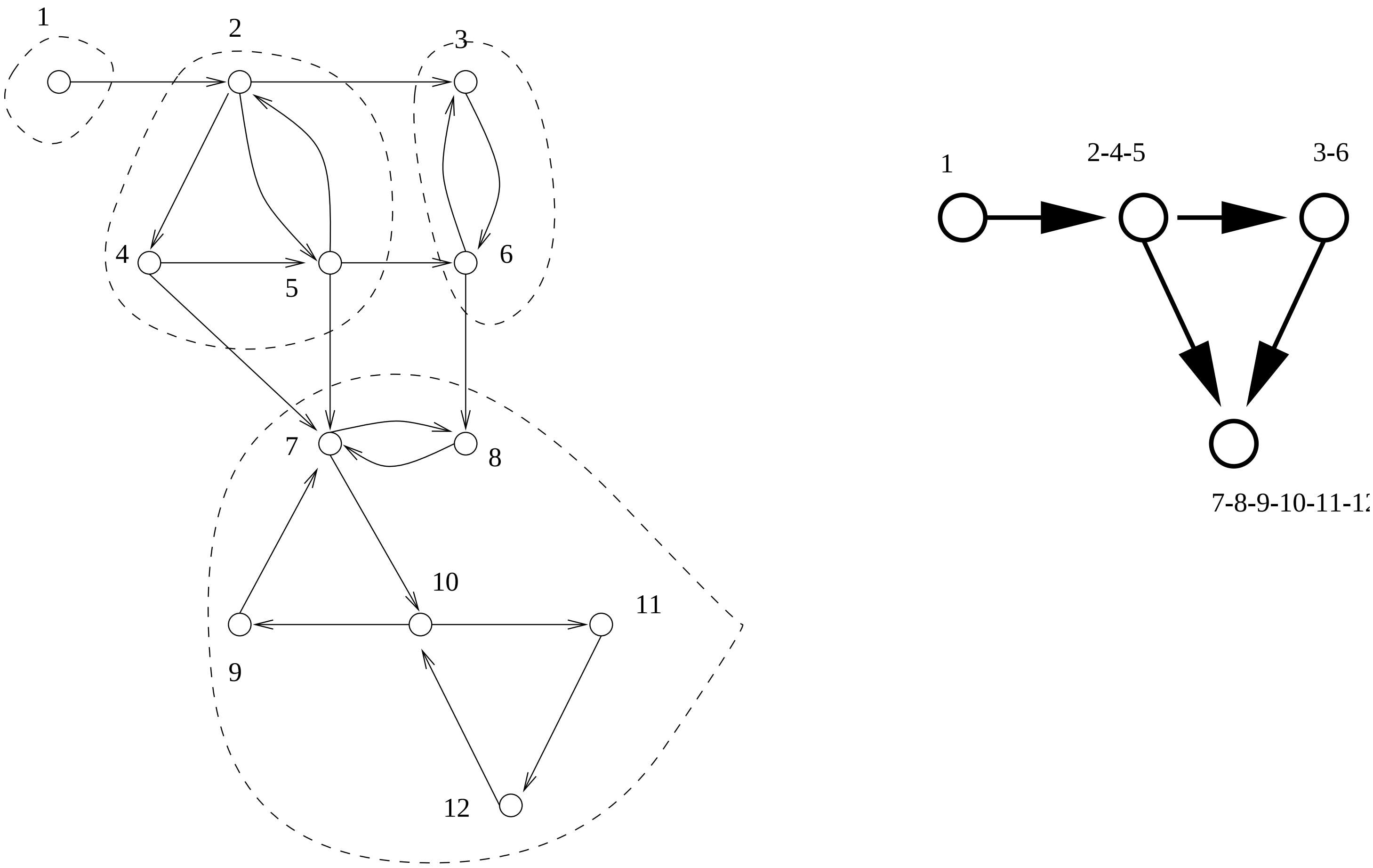

Figure 2: A directed graph and its strongly connected components

This important decomposition theorem allows one to fathom the subtle connectivity information of a directed graph in a two-level manner: At the top level we have a dag —a rather simple structure. For example, we know that a dag is guaranteed to have at least one *source* (a node without incoming edges) and at least one *sink* (a node without outgoing edges), and can be topologically sorted. If we want more details, we could look inside a node of the dag to see the full-fledged strongly connected component —a tightly 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 depth-first search, that finds it *in linear time!* We motivate this algorithm next. It is based on several interesting and slightly subtle properties of depth-first search:

Property 1: If depth-first search of a graph is started at a node u, then it will get stuck and restarted precisely when all nodes that are reachable from u have been visited. Therfore, if depth-first search is started at a node of a sink strongly connected component (a strongly connected component that has no edges leaving it in the dag of strongly connected components), then it will get stuck after it visits precisely the nodes of this strongly connected component.

For example, if depth-first search is started at node 11 in Figure 2 (a node in the only sink strongly connected component in this graph), then it will visit nodes 11, 12, 10, 9, 7, 8, —the six nodes form this sink strongly connected component— and then get stuck. Property 1 suggests a way of starting the sought decomposition algorithm, by finding our first strongly connected component: Start from any node in a sink strongly connected component, and, when stuck, output the nodes that have been visited: They form a strongly connected component!

Of course, this leaves us with two problems: (A) How to guess a node in a sink strongly connected component, and (B) how to continue our algorithm after outputting the first strongly connected component, by continuing with the second strongly connected component, and so on.

Let us first face Problem (A). It is hard to solve it directly. There is no easy, direct way to obtain a node in a sink strongly connected component. *But* there is a fairly easy way to obtain a node in a *source* strongly connected component. In particular:

Property 2: The node with the highest `post` number in depth-first search (that is, the node where the depth-first search ends) belongs to a source strongly connected component.

Property 2 follows from a more basic fact:

Property 3: Let C and C' be two strongly connected components of a graph, and suppose that there is an edge from a node of C to a node of C'. Then the node of C visited first by depth-first search has higher `post` than any node of C'.

In proof of Property 3, there are two cases: Either C is visited before C' by depth-first search, or the other way around. In the first case, depth-first search, started at C, visits all nodes of C and C' before getting stuck, and thus the node of C visited first was the last among the nodes of C and C' to finish. If C' was visited before C by depth-first search, then depth-first search from it was stuck before visiting any node of C (the nodes of C are not reachable from C'), and thus again the property is true.

Property 3 can be rephrased in the following suggestive way: *Arranging the strongly connected components of a directed graph in decreasing order of the highest* `post` *number in each topologically sorts the strongly connected components of the graph!* This is a generalization of our topological sorting algorithm for dags: After all, a dag is just a directed graph with singleton strongly connected components.

Property 2 provides an indirect solution to Problem (A): Consider the *reverse* graph of $G = (V, E)$, $G^R = (V, E^R)$ —G with the directions of all edges reversed. G_R has precisely the same strongly connected component's as G (why?). So, if we make a depth-first search in G^R, then the node where we end (the one with the highest `post` number) belongs to a source strongly connected component of G^R —that is to say, a sink strongly connected component of G. We have solved Problem (A)!

Onwards to Problem (B): How to continue after the first sink component is output? The solution is also provided by Property 3: After we output the first strongly connected componentand delete it from the graph, the node with the highest `post` from the depth-first search of G^R among the remaining nodes belongs to a sink strongly connected component of the remaining graph. Therefore, we can use the same depth-first search information from G^R to output the second strongly connected component, the third strongly connected component, and so on. The full algorithm is this:

Step 1: Perform depth-first search on G^R.

Step 2: Run the undirected connected components algorithm, processing in the depth-first search the nodes of G in order of decreasing `post` found in step 1.

Needless to say, this algorithm is as linear-time as depth-first search, only the constant of the linear term is about twice that of straight depth-first search. (*Question:* How does one construct G^R —that is, the adjacency lists of the reverse graph— from G in linear time? And how does one order the nodes of G in decreasing `post[v]` also in linear timne?) If we run this algorithm on the directed graph in Figure 2, Step 1 yields the following order on the nodes (decreasing post-order in G^R's depth-first search —watch the three "negatives " here): 7, 9, 10, 12, 11, 8, 3, 6, 2, 5, 4, 1. Step 2 now discoverse the following strongly connected components: component 1: 7, 8, 10, 9, 11, 12; component 2: 3, 6; component 3: 2, 4, 5; component 4: 1.

Incidentally, there *is* more sophisticated connectivity information that one can derive from undirected graphs. An *articulation point* is one whose deletion increases the number of connected components in the undirected graph (in Figure 1 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); this is not quite a partition, because neighboring biconnected components share an articulation point. For example, the large connected component of the graph in Figure 1 has four biconnected components: 1-2-3-4-5-7-8, 3-6, 6-9-10, and 8-11. Intuitively, biconnected components are the parts of the graph such that any two nodes have not just one path between them but two node-disjoint paths (unless the biconnected component happens to be a single edge). Just like any directed graph is a dag of its strongly connected components, any undirected graph can be considered *a tree of its biconnected components.* Not coincidentally, this more sophisticated and subtle connectivity information can also be captured by a slightly more sophisticatewd version of depth-first search.