Graphs

Subhash Suri

February 28, 2019

1 Basic Definitions

- Graphs are useful models for reasoning about relations among objects and networked systems.
- Mathematically, a graph has a set vertices (also called nodes) V, often labeled $v_1, v_2, \ldots, v_n$, and a set of edges E, labeled $e_1, e_2, \ldots, e_m$.
- Each edge (u, v) "joins" two nodes u and v.
- We write $G = (V, E)$ for the graph with vertex set V and edge set E.
- **Some Examples of Graphs.**
  - Transportation Networks. The map of routes served by an airline carrier forms a graph, whose nodes are the airports, and we have an edge (u, v) whenever airline has a non-stop flight from u to v. Typically, airline edges are undirected—flight (u, v) also means a flight (v, u).

    Other transportation networks: rail networks, road networks.
  - Communication Networks. Internet is essentially a collection of computers connected by communication links. Nodes are computers, and edges are physical links.

    Wireless networks: devices, and wireless connections.
  - Information networks. WWW has web pages as nodes, and hyperlinks as edges.
  - Social networks.
  - Dependency graphs: nodes = courses, and edges = prereqs;
- Graphs also serve as a powerful modeling tool for many real-life problems even when they are not overtly network-related. Proper application of graph theory ideas can drastically reduce the solution time for some important problems.

- In applications, where pair (u, v) is distinct from pair (v, u), the graph is *directed*. Otherwise, the graph is undirected. We can convert an undirected graph to a directed one by duplicating edges, and orienting them both ways.
- When (u, v) is an edge, we say v is *adjacent to (or, neighbor of)* u. A loop is an edge with both endpoints being the same.
- In undirected graphs, the *degree* of a node equals its number of neighbors. In directed graphs, we have The *out-degree* and the *in-degree*.
- In some applications, the edges can be associated with weights or costs.

2 Representations of Graphs

- **Adjacency Matrix:** a 2-dim array $V \times V$. For each edge (u, v), set $A[u, v] = 1$, or equal to cost, etc. Use infinity or 0 for non-edges.
- This representation takes $|V|^2$ space even if graph has very few edges; e.g. street map, which typically has $O(V)$ edges. CAn be infeasible when V is large. (One small advantage of this representation is that it is easy to check if any pair u, v forms an edge of the graph.)
- **Adjacency List:**. An array of (header cells for) adjacency lists. The ith cell points to a linked list of all vertices adjacent to vertex v_i.
- Example:

1 :	2	4	3
2 :	4	5	
3 :	6		
4 :	6	7	3
5 :	4	7	
6 :			
7 :	6		

- This representation takes $O(|E|)$ space, since each directed edge stored just once. If G is undirected (u, v) appears in lists of both u and v.

3 Paths and Connectivity

- One of the fundamental operations in graphs is that of traversing a sequence of nodes (and edges).

- Such a traversal could correspond to user browsing web pages by following hyper linkes, rumor passing by word of mouth, or travel route of an airline passenger, email passing through a chain of routers, etc.

- To formalize graph exploration problems, we need the concept of a path.

- A *path* is sequence of vertices $w_1, w_2, \ldots, w_k$ such that each pair (w_i, w_{i+1}) is an edge of G. The *length* of a path is the number of edges in it, or total weight if each edge has a weight associated with it.

- A *simple path* has no repeated vertex, except first and last can be the same; in that case, the path is a cycle.

- **Connectivity.** An undirected graph is *connected* if there is a path between any two vertices. A directed graph with this property is *strongly connected.* A weakly connected graph—underlying graph connected but the directed graph may not have directed path between all pairs.

- A *minimally connected* graph is a *tree*: it is a connected graph without any cycles.

  1. Any tree on n nodes has $n-1$ edges, and therefore the deletion of a single node or edge disconnects it.
  2. Often it is useful to *root* the tree at a particular node r, and then *orient* all edges away from r. EXAMPLE.
  3. In a rooted tree, each node (except root) has a parent, and if u is the parent of w, then w is called a *child* of u.
  4. More generally, w is called a *descenedant* of u, and u an *ancestor* of w, if u lies on the path from w to the root.

4 Graph Connectivity and Graph Traversals

- We start with one of the most basic questions regarding graphs. Given a graph $G = (V, E)$, and two nodes s and t, is there a path joining s and t?

- This is called the *st*-connectivity problem (same as the classical Maze problem). In small graphs, one can decide this by visual inspection, but quickly becomes challenging in large graphs.

- More generally, given a start node s, what are all the nodes reachable from s? This set is called the *connected component* of G containing s.

- In program control-flow analysis, e.g., unreachable nodes are dead-code, which can be eliminated. Similarly, if there is a node from which **exit** is unreachable, then program contains an infinite loop.

- In garbage collection, reachability finds memory objects accessible by the program.

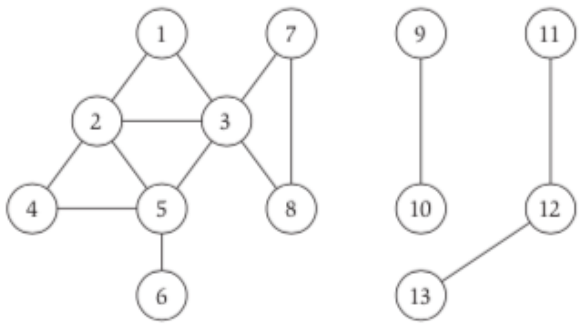


Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

- There are two simple algorithms for *st*-connectivity

4.1 Breadth First Search

- The simplest algorithm for *st*-connectivity is the following. We start at s, and explore outward in all possible directions.

- We just have to make sure we don't get stuck in a loop, so we use *markers* to keep track of nodes we have already visited.

- Each node will get a *level number* (also called level). Initially, we have only s, which is level 0. The next iteration adds previously unreached nodes that have an edge to an already reached node. More specifically,

- We initialize Level $L_0 = \{s\}$; i.e. level 0 containing just s. Level L_1 consists of all neighbors of s.

- Assuming we have levels $L_0, L_1, \cdots, L_i$, define

$$L_{i+1} = \text{ nodes not yet encountered who have an edge to some node in level } L_i$$

- Example with 3 connected components.

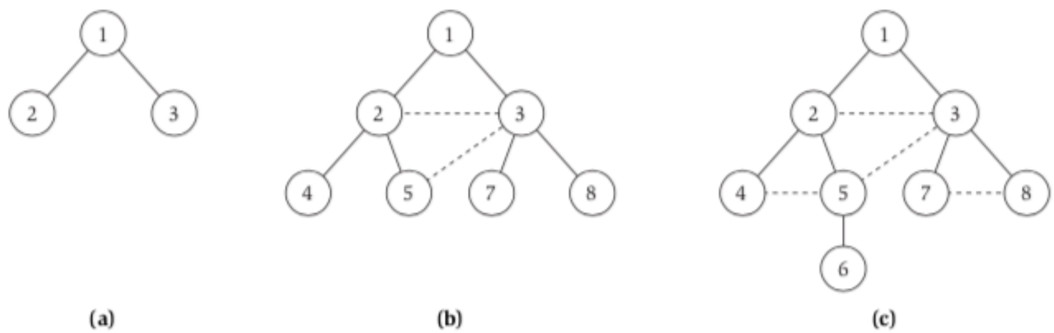


Figure 3.3 The construction of a breadth-first search tree T for the graph in Figure 3.2, with (a), (b), and (c) depicting the successive layers that are added. The solid edges are the edges of T; the dotted edges are in the connected component of G containing node 1, but do not belong to T.

- The level by level exploration of G produces a tree-like structure, which is called the BFS tree of G.

- For each $j \geq 1$, level L_j consists of all nodes at distance exactly j from s. There is a path from s to t if and only if s appears in some level of BSF from s.

- Let T be a BFS tree, and let x and y be nodes in T belonging to different levels L_i and L_j. If (x, y) is an edge of G, then $|i - j| \leq 1$.

- Proof. For contradiction, assume $i < j - 1$. When x is scanned in level i, the edge (x, y) will add y to level L_{i+1}, ensuring $j \leq i + 1$.

- BFS can be constructed in $O(m + n)$ time, using Adj List representation of G.

- The set of nodes discovered by the BFS is precisely those reachable from s. We refer to this set R as the *connected component* of G containing s.

- Once we have R, we can simply check if t belongs to R, and if so we have st connectivity.

- BFS however is only one way to discover R. Another, and a very different, method is depth first search.

5 Depth First Search

- Depth-First-Search (DFS) uses a method similar to the exploration of mazes:
- Starting at s, we take the first edge out of s, and continue recursively untill we reach a *dead end*—a node for which all neighbors have already been explored.
- We then backtrack until we get to a node with at least one explored neighbor.
- This is called DFS search, because it explores G by going as deeply as possible and only retreating when necessary.

```
DFS(u)
    Mark u as Explored and add u to R
    For each edge (u, v) incident to u
        if v is not marked Explored, then
             recursively call DFS(v)
        endif
    endfor.
```

- We can also implement DFS non-recursively.

```
Stack Implementation of DFS:

DFS(s)

   Init S to be a stack with one item s
   While S not empty
        Take a node u from S
        If Explored[u] = False then
           Set Explored[u] = True
           For each edge (u,v) incident to u
               Add v to stack S
           endfor
        endif
   endwhile
```

- Example from Kleinberg-Tardos.

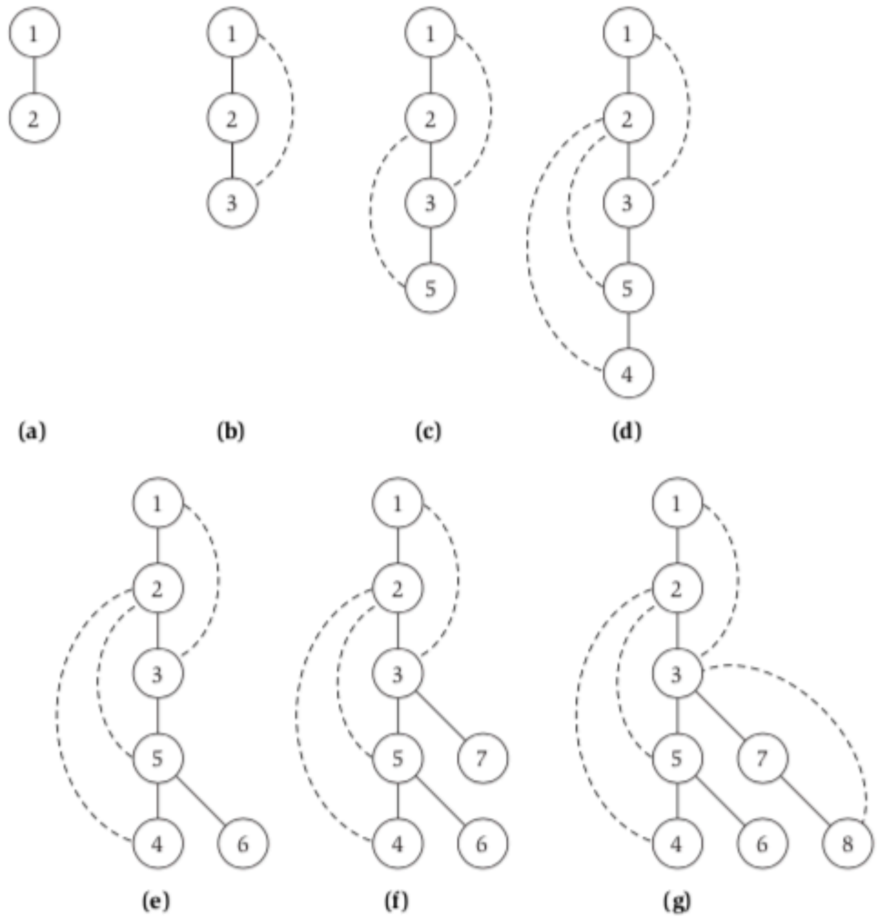


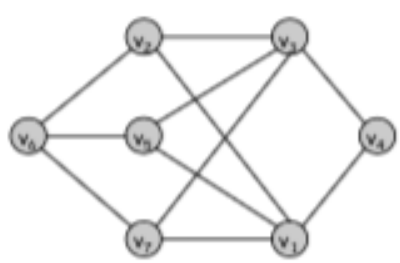
Figure 3.5 The construction of a depth-first search tree T for the graph in Figure 3.2, with (a) through (g) depicting the nodes as they are discovered in sequence. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T.

- DFS also runs in $O(m + n)$ time, where $n = |V|$ and $m = |E|$.
- Although the DFS tree looks very different from the BFS tree of G, we can make strong claims about how non-tree edges connect the nodes of DFS.

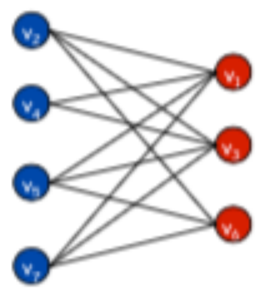
- **Fact 1.** For a recursive call $DFS(u)$, all nodes that are marked *explored* between the invocation and the end of the recursive call are descendants of u in T.
- **Fact 2.** Let T be a DFS tree, let x, y be two nodes in T that have an edge between them in G, but (x, y) is not an edge of T. Then, one of x or y is an ancestor of the other.
- Suppose not, and assume that x is reached first in DFS. When the edge (x, y) is examined during the execution of $DFS(x)$, it is not added to T because y is marked Explored. Since y was not marked Explored when $DFS(x)$ was first invoked, it must have been discovered during the recursive call. Thus, by Fact 1, y must be a descendant of x.
- **Connected Components Fact.** For any two nodes s and t in G, their conenected components are either identical or disjoint.

6 Applications of BFS and DFS

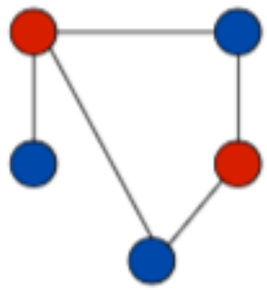
- **Testing Bipartititeness.** A graph G is bipartite if its vertex set V can be partitioned into sets X and Y in such a way that every edge of G has one end in X and the other in Y.
- Often we use colors red and blue (or 0 and 1) to represent X and Y.



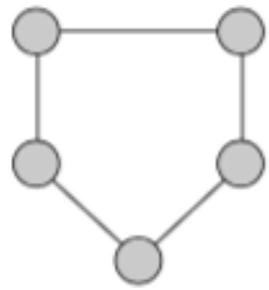
a bipartite graph G



another drawing of G



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

- A triangle is not bipartite: any partition will contain two nodes on the same side with an edge between them. The same argument also holds if G is an odd-length cycle.
- Turns out however that odd cycles are the only obstacle for G to be bipartite: that is, G is bipartite if and only if it does not contain an odd cycle.
- In fact, one can use BFS to decide whether G is bipartite, and in the end either discover the sets X and Y, or detect an odd-cycle, thereby showing that G is not bipartite.
- We can easily assume that G is connected. Otherwise, we can apply the algorithm to each connected component separately.
- The algorithm begins by picking any arbitrary vertex s, and color it 0.
- Now all neighbors of s must be colored 1, and these are precisely the nodes of level 1.
- We alternate between colors: the nodes at level i are colored 0 if i is even, and colored 1 if i is odd.
- At the end of the algorithm, we simply go back and check if the endpoints of each edge of G are colored differently. If not, that edge (x, y) together with the path in the BFS from x to y is an odd cycle.
- Therefore, bipartiteness of a grapg G can be decided in $O(m + n)$ time.

7 Bi-Connectivity

- An undirected graph G is *bi-connected* if we must delete at least two nodes (and their incident edges) to disconnect G. In other words, there is no single node whose removal disconnects G.

- **Articulation point** is a node v whose removal disconnects G. Thus, G is bi-connected if and only if there is no articulation point.

- A naive algorithm to check for bi-connectivity: remove each vertex v in turn, and check if $G \setminus \{v\}$ is connected. Using DFS (or BFS) for connectivity, the overall algorithm will run in $O((n+m)^2)$ time.

- A classical application of DFS is the Hopcroft-Tarjan algorithm that checks for bi-connectivity (and finds all bi-connected components of G) in $O(n+m)$ time.

- The main idea is to run a DFS while maintaining the following information for each vertex v of the DFS tree T:

  1. $Num(v)$: position of v in the DFS visit order, starting with $Num(root) = 1$,
  2. $low(v)$: the lowest-numbered vertex reached from v by taking zero or more tree edges followed by (at most) one back edge.

- We can think of $Num(v)$ as the time stamp of the first visit to v. Then the function low can be defined as:

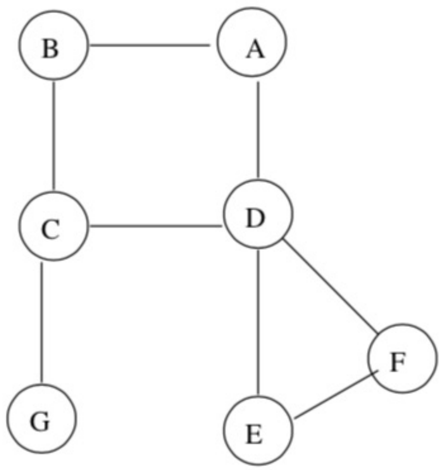
$$low(v) = \min\left(Num(v), \{Num(w) : (u,w) \text{ is a back edge for some descendant } u \text{ of } v\}\right)$$

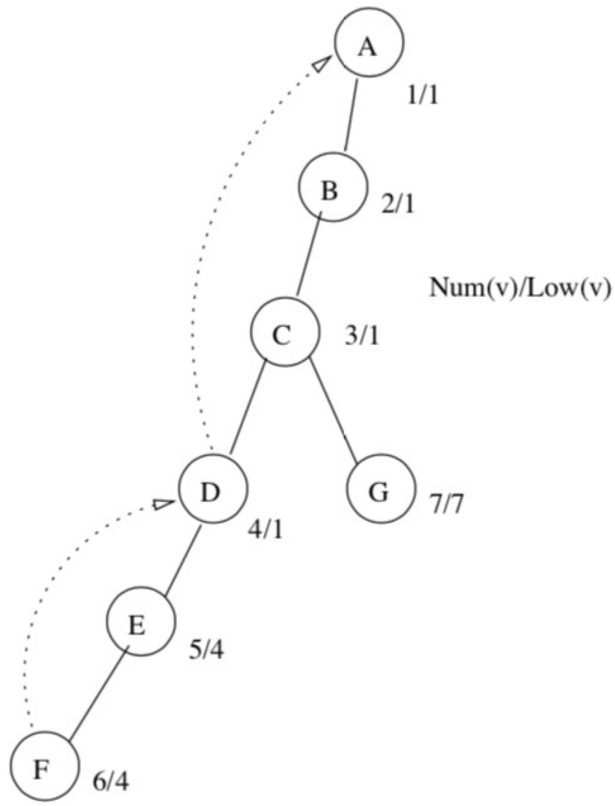
- In other words, $low(v)$ is the minimum of

  1. $Num(v)$
  2. lowest $Num(w)$ among all back edges (v,w)
  3. the smallest $low(w)$ among all children w of v.

- The $low()$ values of all the nodes can be computed in linear time, by performaing a *post-order* traversal of T.

- Example.



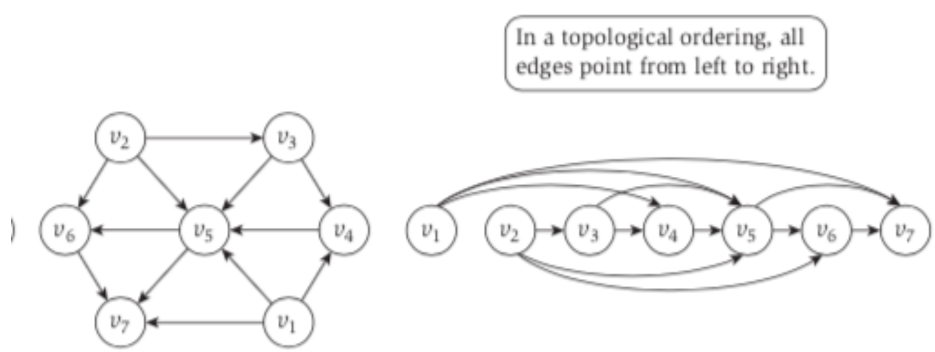


- Once we have these computed, detecting articulation points is easy:
  1. the root is an articulation point, if it has more than one child;
  2. any non-root node v is an articulation point if it has a child w with $low(w) \geq Num(v)$.
- For proof, notice that if v is an articulation point then none of the nodes explored during the recursive call at v have an edge that goes to the other component, and thus the $low()$ value for all these points is $\geq Num(v)$.

8 Topological Sort

- An *undirected* graph *without cycles* has a very simple structure: each connected component is a tree.

- A *directed* graph, on the other hand, can have a complex structure even if it contains no directed cycles. For instance, it can have lots of edges, in fact $\binom{n}{2}$ edges.
- A directed graph without cycles is naturally called a *Directed Acyclic Graph* (DAG), and such graphs are very common in computer science.
- Suppose you have a set of tasks, which are subject to a set of precedence constraints: some jobs cannot be done before others. How shall you schedule the jobs without violating any precedence constraint?
- Model as a *directed* graph where jobs are nodes and precedence relations are directed edges.
- Given a set of tasks with dependencies, we would like to find an ordering in which tasks can be performed, so that all dependencies are respected.
- Such an ordering is called *topological ordering*. Specifically, a topological ordering of a graph G is an ordering of its nodes as $v_1, v_2, \ldots, v_n$, so that for every edge (v_i, v_j), we have $i < j$.
- In other words, all edges point in the "forward" direction in the ordering.
- An example.



- In fact, the *existence* of a topological ordering is an immediate "proof" that G has no cycles.
- Claim: *If G has a topological ordering, then G is a DAG.*

- Proof. By contradiction. Suppose G has a topo ordering $v_1, v_2, \ldots, v_n$, but also a cycle C. Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i—thus, (v_j, v_i) is an edge of G. But by our choice of i, we have that $j > i$, which contradicts the assumption that $v_1, v_2, \ldots, v_n$ is a topological ordering.

- We now discuss several algorithms for computing a topological ordering.

- The first algorithms uses the observation that in every DAG, there is a node with no incoming edges.

- We can prove this by contradiction. Suppose G is a DAG but every vertex has at least one incoming edge. We pick one such node v, and start following edges "backwards" from v: it has an incoming edge (u, v), we walk back to u, which itself has an incoming edge, say, (x, u), and so on. Since each node has an incoming edge, we can continue this process indefinitely, but after $n + 1$ steps we will have encountered at least one node twice, because there are only n nodes in G. The sequence of edges in our walk is a cycle, completing the proof.

- Computing a topological ordering.

```
Algorithm:
Find a vertex v with zero in-degree (must exist!)
Print v, delete v, and its outgoing edges;
Repeat.
```

```
Improved Topological Sort

    Compute all vertices' indegs
    Enqueue all those with zero indeg
    Pick a vertex w from the queue;
         put w next in schedule
         for each vertex v adj to w
             decrement v's indeg
             add v to queue if its indeg = 0
```

- This code only looks at each edge once, so runs in $O(n + m)$ time.

- The second algorithms uses DFS, and is based on the following claim: *G is a DAG if and only if any DFS forest of it contains no back edges.*

- Proof of the Claim:

  1. Suppose G has no cycles. Then, there cannot be a back edge in a DFS of G because if (u, v) is a back edge, then it creates a cycle with the tree path connecting v to u.
  2. Assign each vertex v its DFS *finish time*. Then, all edges of G are directed from higher finish time to smaller finish time. Ordering the vertices in the *reverse finish time* gives the topological ordering.

9 Strong Bi-Connectivity

- DFS and BFS algorithms work on directed graphs, without any significant change: while visiting a vertex v, we just scan v's out neighbors.
- In directed graphs, however, we need a stronger definition of a connected components. We put two vertices u and v in the same component only if we have a directed path from u to v **and** a path from v to u.
- Example.
- We can find strong connected components of G also in $O(|V| + |E|)$ time, by using DFS, but in a more careful way.
- Historically, the first linear time algorithm dates back to 70s by Hopcroft and Tarjan.
- A simpler algorithm is by Koraraju-Sharir. It performs two DFS once on G, and once on G^R, which is G with all edges reversed.
- The algorithm has the following structure.

  1. Perform a DFS in G.
  2. Number the vertices in the **post-order**, namely, the order in which their recursive calls finish.
  3. Construct the reverse graph G^R, in which each edge has the opposite orientation than G.
  4. Perform a second DFS on G^R, always starting at the highest numbered vertex in our ordering, and output each subtree found by this DFS as a SCC, and remove it from the graph.

- We illustrate the algorithm on an example. Figure 1 shows a directed graph, and its DFS.

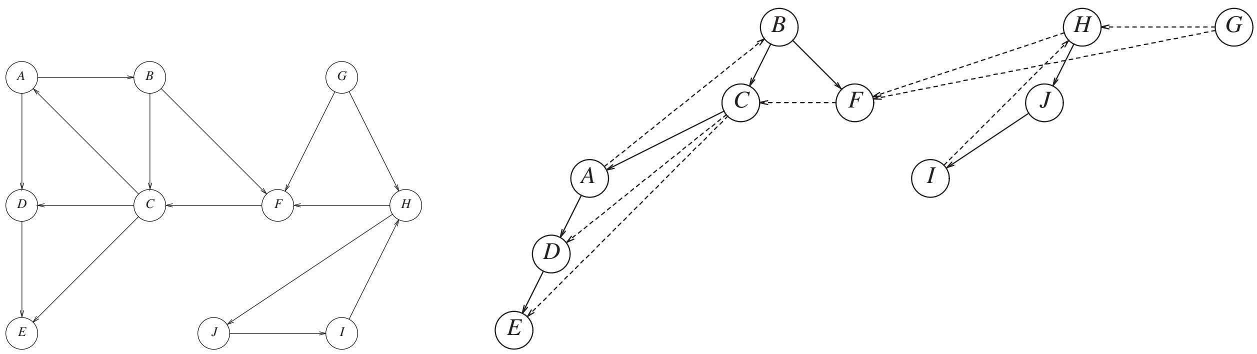


Figure 1: A directed graph and its DFS.

- The numbering of nodes (in the largest to smallest) order is: $G, H, J, I, B, F, C, A, D, E$.
- Thus, in our example (see Figure 2), the first DFS starts at node G, numbered 10. This leads nowhere, so G is a singleton node component. We remove it, and continue.
- Next DFS starts at H, and this call adds I and J to the component of H.
- Next starts at B, and adds $\{A, C, F\}$ before finishing.
- DFS at D ends with singleton, as does for E.

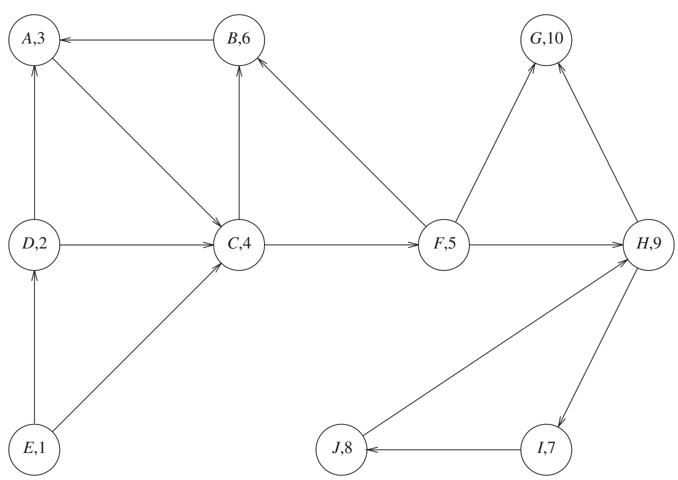


Figure 2: G^R, with post-order numbering from the first DFS.

- Proof of Correctness. To show that each tree found during the second DFS (on G^R) is a strongly connected component, we will demonstrate the following: *Suppose T is a tree, with root node x, found by the DFS in G^R, and v is a node in T. Then, the* *original graph G contains both a directed path from x to v, and a directed path from v to x.*

- This suffices since if v and w are two nodes in T, then we can reach w from v by following the path v to the root x plus path from x to w, and vice versa.

- To prove the first path that G contains a path from v to x, we note the following. The node v is a descendant of x in T, which means there is a directed path from x to v in G^R. Since each edge of G^R is the reverse of its copy in G, this path corresponds to a *v to x path in G.*

- We now prove the converse that G also contains a path from x to v.
  - Since x is the root of T, it means that x has the larger finish time than v in the first DFS.
  - Therefore, during that DFS in G, the recursive call at v finished before the recursive call at x finished.
  - Since we have already shown a v to x path in G, it must be that v is a descendant of x in the DFS of G—otherwise, v would finish *after* x.

- Therefore, there is a path from x to v, and the proof is complete.