Strongly Connected Components

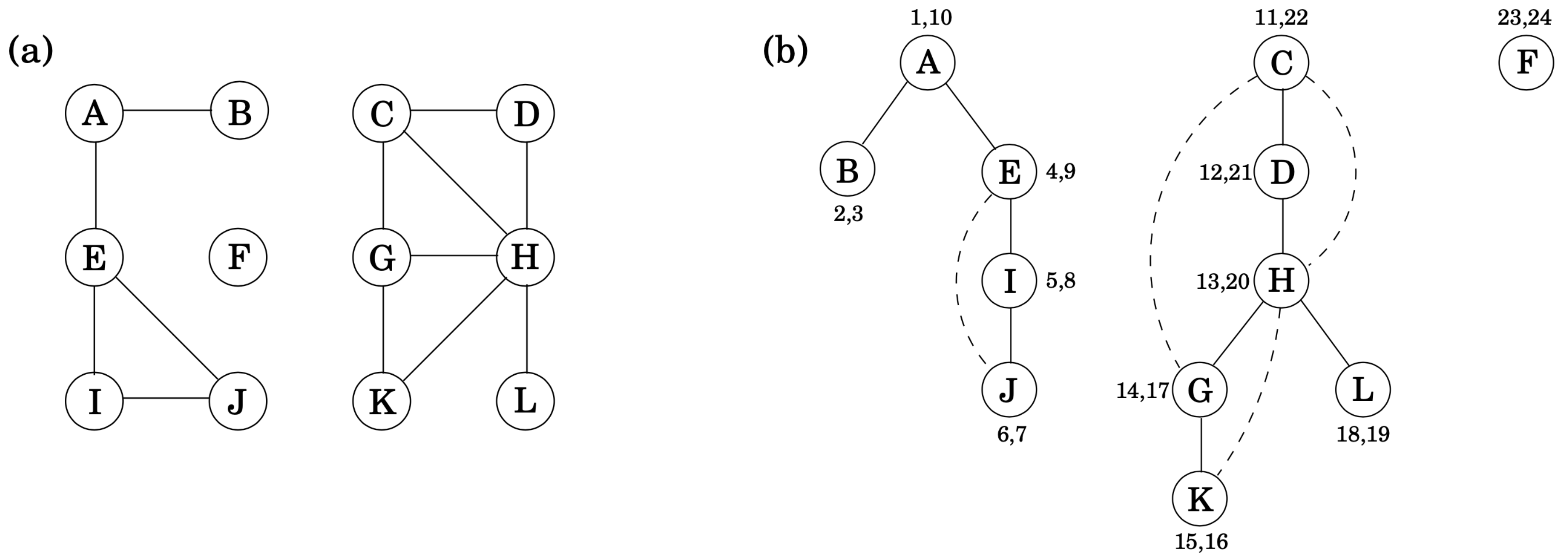
Yixiong Gao

July 8th, 2022

Connectivity in undirected graphs

- The definition of **connected** is straightforward.
- We can get each **connected component** by DFS in O(n) time.

Figure 3.6 (a) A 12-node graph. (b) DFS search forest.

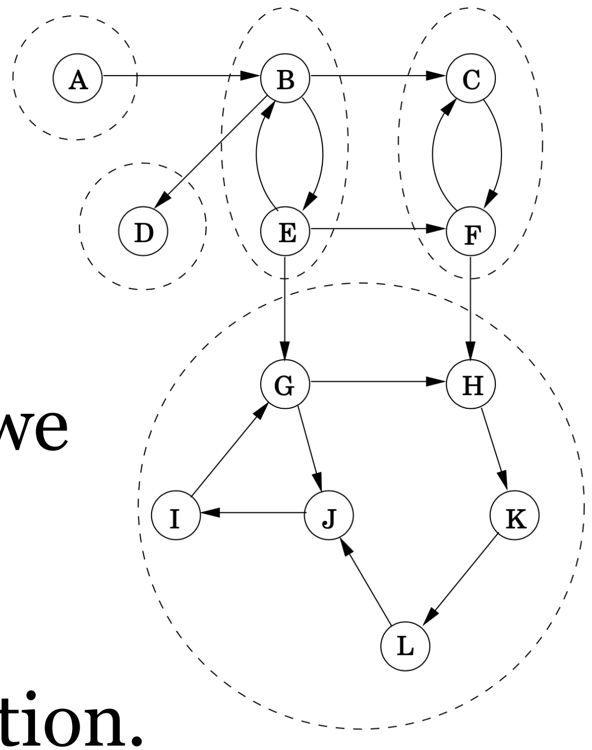


Connectivity in directed graphs

- Two nodes u and v are *connected* if:

  There is a path from u to v *and* a path from v to u.

- This relation partitions V into **disjoint sets** that we call ***strongly connected components***.

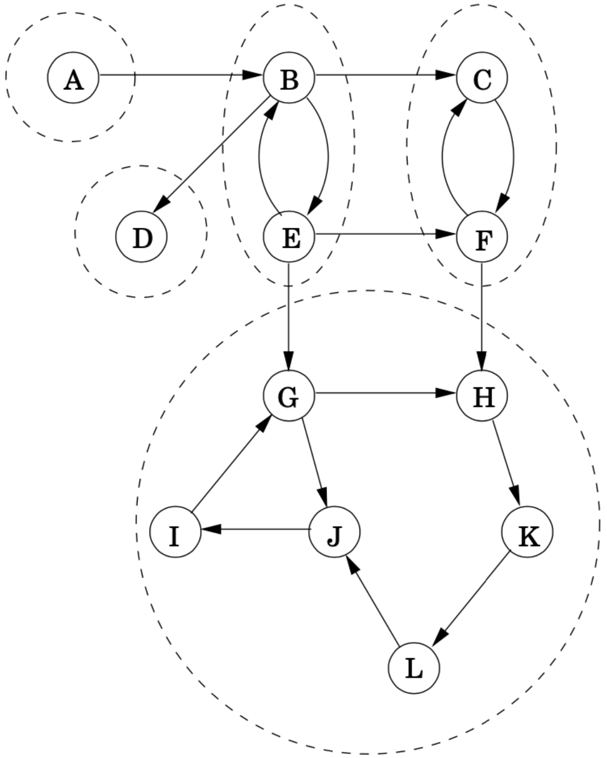


(Exercise 3.30) "connected" is an equivalence relation.

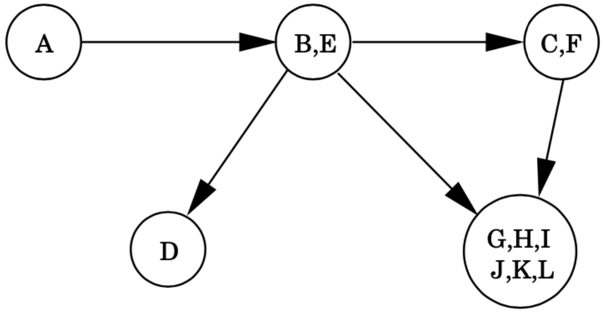
Build *meta-graph* by SCC

- Shrink each SCC down to a single *meta-node*.
- Draw edges between *meta-nodes* by original edges.
- The resulting *meta-graph* must be a **DAG**. (why?)

(a)



(b)



Meaning

- **Property** *Every directed graph is a DAG of its SCCs*.

- The connectivity structure of a directed graph is two-tiered.

- Top : a DAG which is a rather simple structure (linearized).

- Detail : look inside one of the nodes of this DAG and examine the strongly connected component within.

Kosaraju's Algorithm

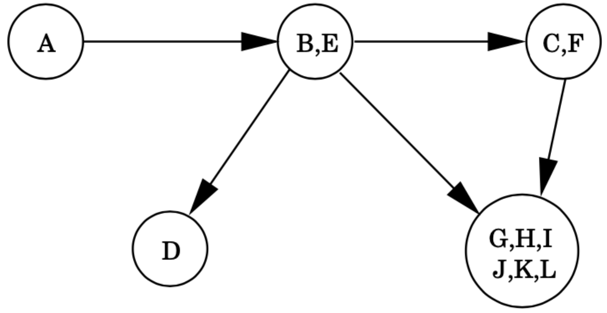
Getting all the SCC in linear time by DFS twice.

Overview

- Motivation: (1) Every directed graph is a DAG of its SCCs;
  (2) DAG can be linearized.

<u>**Algorithm** Kosaraju's algorithm (abstract version)</u>

while the graph is not empty **do**
  1: Find a **sink** SCC;
  2: Delete it from the graph;
end while



Find a sink SCC

- Solution: (1) Find a **node** that lies somewhere in the sink SCC;
  (2) Explore at this node, we will traverse the whole SCC.

Property 1

If the explore subroutine is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.

- All nodes in the same SCC are pairwise reachable (connected);
- Otherwise, node u can’t reach other nodes in different SCC.
  - Cause the SCC which node u lies is a sink component.

Find a node lies in the sink SCC

Property 2

If C and C′ are SCCs, and there is an edge from a node in C to a node in C′,

then the highest **post number** in C is bigger than the highest post number in C′.

- If dfs visits C before C′, the first node visited in C will have a higher post number than any node of C′.
- Otherwise, the dfs will get stuck after seeing all of C′ but before seeing any of C, in which case the property follows immediately.

Property 3

The node that receives the **highest post number** in a dfs must lie in a **source** strongly connected component.

Cont’d

Property 3

The node that receives the **highest post number** in a dfs must lie in a **source** strongly connected component.

- However, what we need is a node in the **sink** component.

- Consider the **reverse** graph G^R (G with all edge reversed) !
- G^R has the same SCCs as G (the relation “connected” is symmetry).
  - The meta-graph of G^R is also the reverse graph of the meta-graph of G.

- The node with the highest post number comes from a **source** SCC in G^R, which is to say a **sink** SCC in G.

Kosaraju’s Algorithm

Algorithm Kosaraju’s algorithm (abstract version)

while the graph is not empty **do**

1: Find a **sink** SCC;

2: Delete it from the graph;

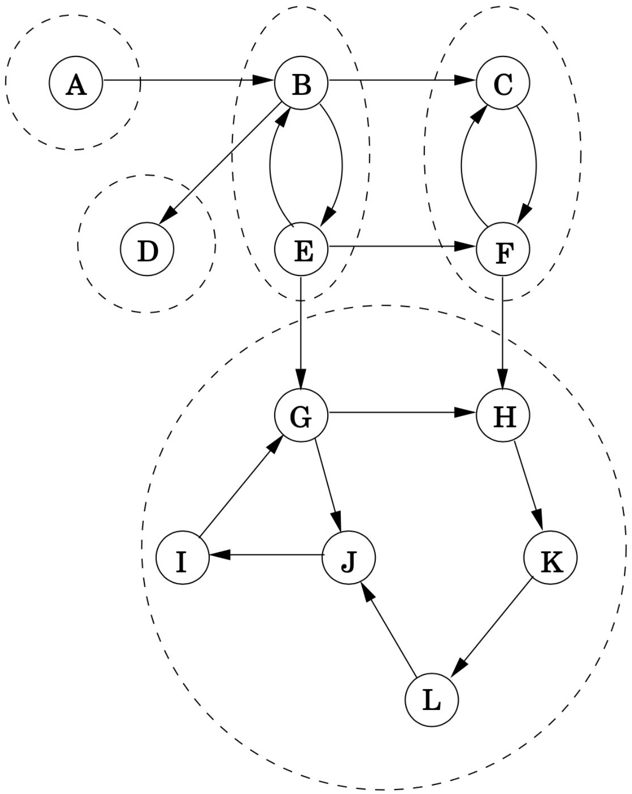
end while

Algorithm Kosaraju’s algorithm

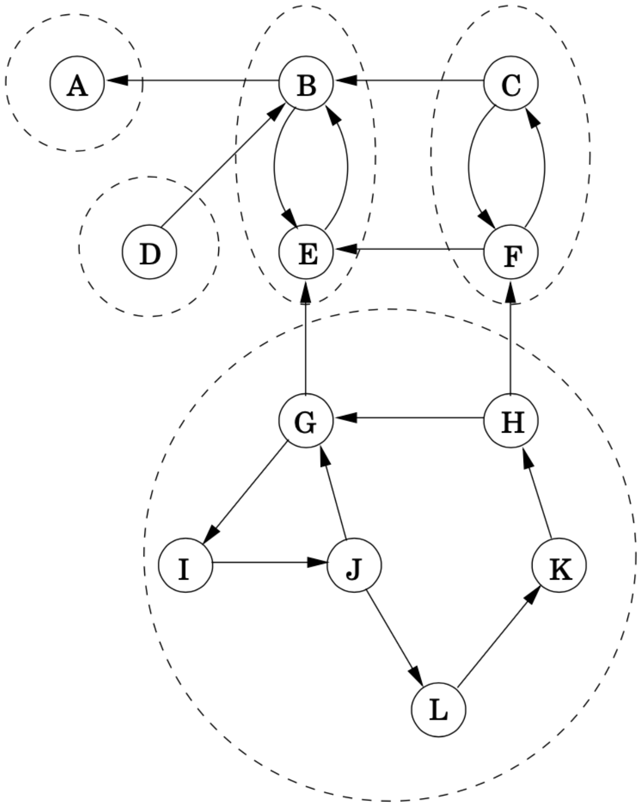
1. Run dfs on G^R, record the post number.
2. Run dfs on G , following decreasing order of their post numbers from step 1. (“Delete” means ignoring nodes has been visited.)

Example

the directed graph



the reverse graph



Time Complexity

<u>Algorithm</u> <u>Kosaraju's algorithm</u>

1. Run dfs on G^R, record the post number.
2. Run dfs on G , following decreasing order of their post numbers from step 1. ("Delete" means ignoring nodes has been visited.)

- This algorithm is linear-time, only the constant in the linear term is about twice that of straight depth-first search.
- *Question: How can we order the vertices of G by decreasing post numbers in linear time?

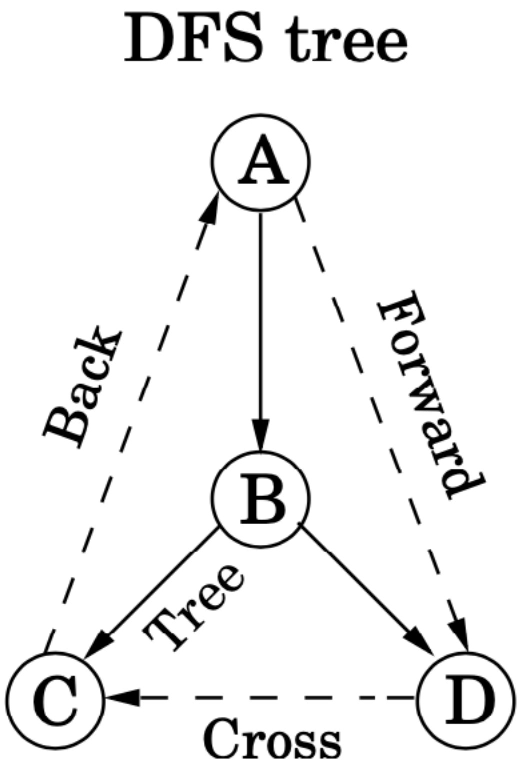
Tarjan’s SCC Algorithm

Getting all the SCC in linear time by DFS only once.

Overview

- The collection of dfs trees is a spanning forest of the graph.
- The strongly connected components will be recovered as certain subtrees of this forest.
- The roots of these subtrees are called the "roots" of the strongly connected components.
  - Any node of a strongly connected component might serve as a root, if it happens to be the first node of a component that is discovered by search.
- How to find the correspond subtree of a SCC ?

Review



Tree edges are actually part of the DFS forest.

Forward edges lead from a node to a *nonchild* descendant in the DFS tree.

Back edges lead to an ancestor in the DFS tree.

Cross edges lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already postvisited).

- What kinds of edges are useful?

Motivation

- The relation “connected” is based on circles.
- Consider finding circles on the DFS tree by adding edges.

- Back edge (u, v) : nodes on the path from v to u are all connected, since they forms a circle.
- Cross edge (u, v) : if v could reach some ancestors of u , it’s useful. {(u, v), path from v to ancestor of u, path from ancestor of u to u}
  - *Here we only need to consider the ancestor with minimum depth.
- Actually we can consider Back edge and Cross edge in the same way.
- What we concerned is the node v could reach with minimum depth.

Stack invariant

- Nodes are placed on a stack in the order they are visited.
- A node remains on the stack after it has been visited if and only if **there exists a path in the input graph from it to some node earlier on the stack**.
- In other words, it means that in the DFS a node would be only removed from the stack after all its **connected paths** have been traversed. When the DFS will backtrack it would remove the nodes on a single path and return to the root in order to start a new path.

Pop principle

- At the end of the call that visits v and its descendants, we know whether v itself has a path to any node earlier on the stack.
  - If so, the call returns, leaving v on the stack to preserve the invariant.
  - If not, then v must be the root of SCC, which consists of v together with any nodes later on the stack than v (such nodes all have paths back to v but not to any earlier node, because if they had paths to earlier nodes then v would also have paths to earlier nodes which is false).

- The connected component rooted at v is then popped from the stack and returned, again preserving the invariant.

Lowlink Mark

- index[u] : the pre number in pre-visit DFS;
- lowlink[u] : the smallest index of any node **on the stack** known to be reachable from v through v's DFS **subtree**, including v itself.
- We can easily find that lowlink[u] ≤ index[u] all the time.

- At the end of the call that visits v and its descendants,

  If lowlink[u] = index [u] , then the node u is a “root” of a SCC!

Tarjan's SCC Algorithm

Algorithm tarjan **is**
  input: graph $G = (V, E)$
  output: set of strongly connected components (sets of vertices)

  index := 0
  S := empty stack
  for each v **in** V **do**
    if v.index is undefined **then**
      strongconnect(v)
    end if
  end for

```
function strongconnect(v)
    // Set the depth index for v to the smallest unused index
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)
    v.onStack := true

    // Consider successors of v
    for each (v, w) in E do
        if w.index is undefined then
            // Successor w has not yet been visited; recurse on it
            strongconnect(w)
            v.lowlink := min(v.lowlink, w.lowlink)
        else if w.onStack then
            // Successor w is in stack S and hence in the current SCC
            // If w is not on stack, then (v, w) is an edge pointing to an SCC already found and must be ignored
            // Note: The next line may look odd - but is correct.
            // It says w.index not w.lowlink; that is deliberate and from the original paper
            v.lowlink := min(v.lowlink, w.index)
        end if
    end for

    // If v is a root node, pop the stack and generate an SCC
    if v.lowlink = v.index then
        start a new strongly connected component
        repeat
            w := S.pop()
            w.onStack := false
            add w to current strongly connected component
        while w ≠ v
        output the current strongly connected component
    end if
end function
```

Time Complexity

- The Tarjan procedure is called once for each node;
- The forall statement considers each edge at most once.
- The algorithm's running time is therefore linear in the number of edges and nodes in G, which is, $O(|V|+|E|)$.

- In order to achieve this complexity, the test for whether w is on the stack should be done in constant time. This may be done, for example, by storing a flag on each node that indicates whether it is on the stack, and performing this test by examining the flag.

Thanks for listening!