Finding strongly-connected components

Tyler Moore

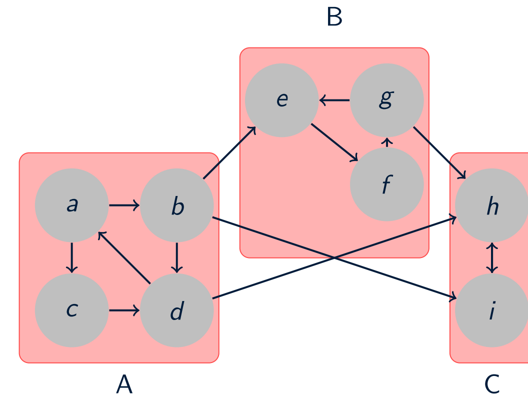
CSE 3353, SMU, Dallas, TX

Lecture 9

Some slides created by or adapted from Dr. Kevin Wayne. For more information see `http://www.cs.princeton.edu/~wayne/kleinberg-tardos`. Some code reused or adapted from Python Algorithms by Magnus Lie Hetland.

Strongly connected components

A strongly connected component is the maximal subset of a graph with a directed path between any two vertices



Strong connectivity

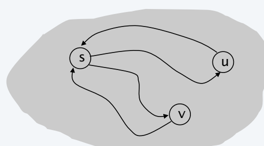
Def. Nodes u and v are mutually reachable if there is a both path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf. $\Rightarrow$ Follows from definition.
Pf. $\Leftarrow$ Path from u to v: concatenate $u \leadsto s$ path with $s \leadsto v$ path.
Path from v to u: concatenate $v \leadsto s$ path with $s \leadsto u$ path. ▪

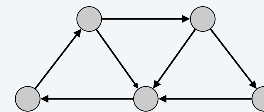
ok if paths overlap



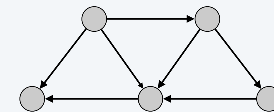
Strong connectivity: algorithm

Theorem. Can determine if G is strongly connected in $O(m + n)$ time.
Pf.

- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in $G^{reverse}$. (reverse orientation of every edge in G)
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. ▪



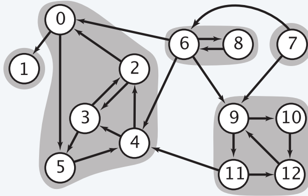
strongly connected



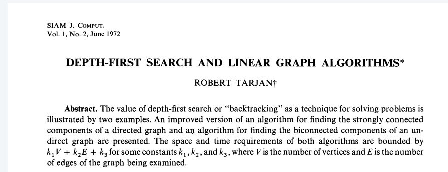
not strongly connected

Strong components

Def. A strong component is a maximal subset of mutually reachable nodes.



Theorem. [Tarjan 1972] Can find all strong components in $O(m + n)$ time.



SIAM J. Comput.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

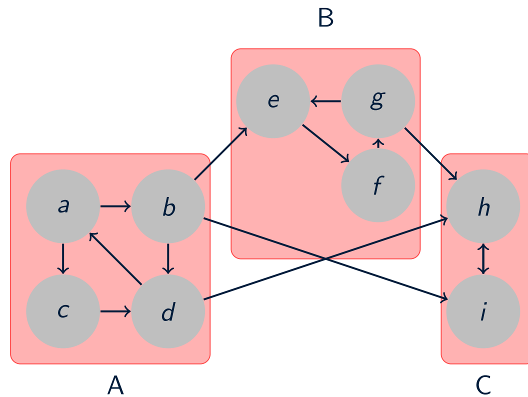
Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.

43

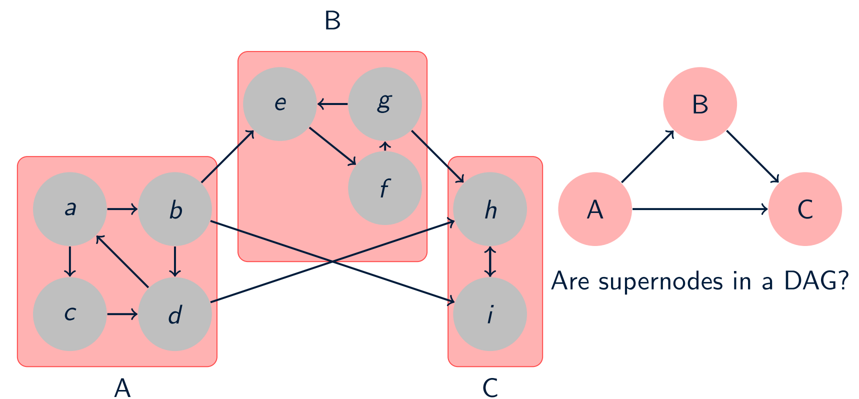
Is Wikipedia a strongly connected graph?

Strongly connected components

A strongly connected component is the maximal subset of a graph with a directed path between any two vertices

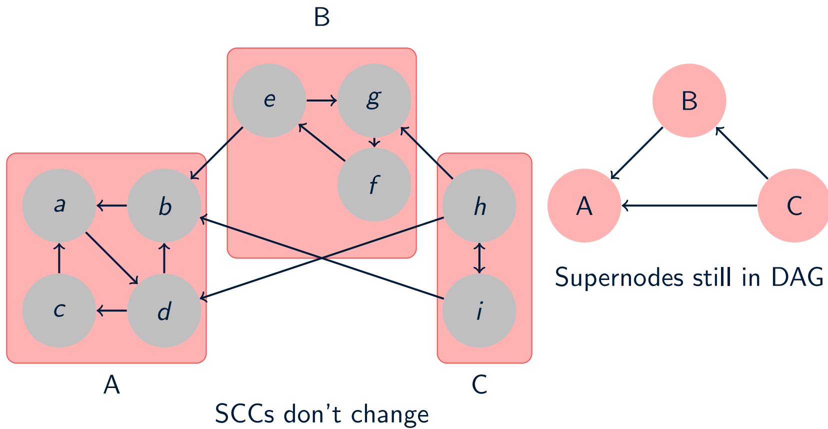


Strongly connected components



Strongly connected components

What if we transpose all edges?



Supernodes still in DAG

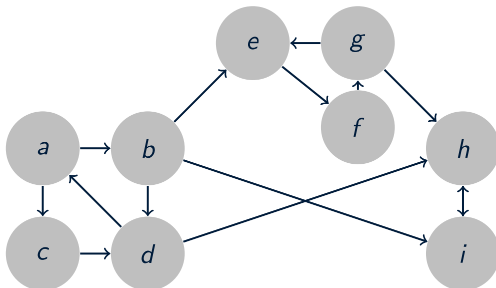
SCCs don't change

Kosaraju's algorithm for finding SCCs

1. Get a topological sort of all vertices
2. Transpose the graph (reverse all edges)
3. Traverse the graph in topologically sorted order, adding an SCC each time a dead end is reached.

Kosaraju's Algorithm for Finding Strongly Connected Components

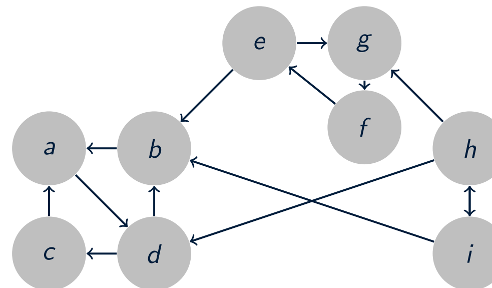
1. Get a topological sort of all vertices



- topsort: [a, b, e, f, g, c, d, h, i]
- seen: {}
- sccs: []

Kosaraju's Algorithm for Finding Strongly Connected Components

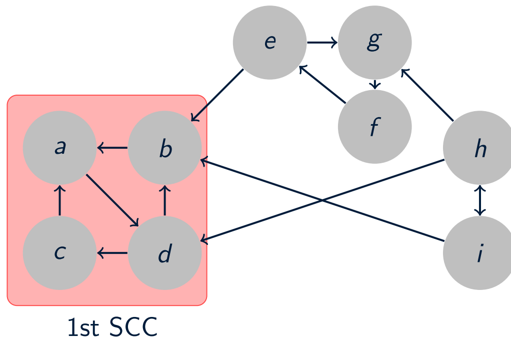
2. Reverse all edges



- topsort: [a, b, e, f, g, c, d, h, i]
- seen: {}
- sccs: []

Kosaraju's Algorithm for Finding Strongly Connected Components

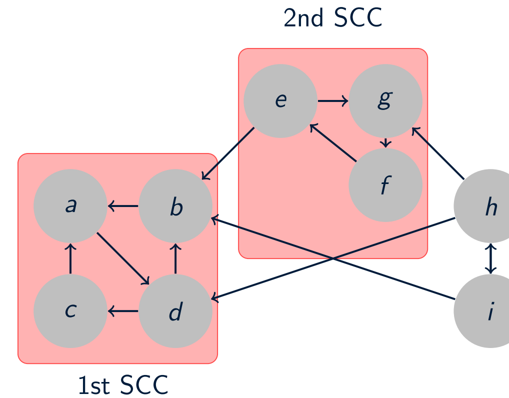
3. Traverse the graph in topologically sorted order, adding an SCC each time a dead end is reached.



- topsort: [a, b, e, f, g, c, d, h, i]
- seen: {a,b,c,d}
- sccs: [{a,b,c,d}]

Kosaraju's Algorithm for Finding Strongly Connected Components

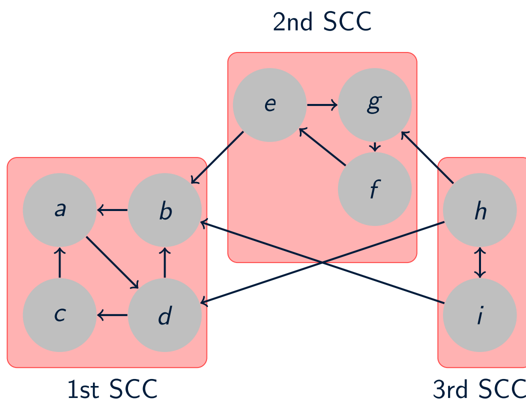
3. Traverse the graph in topologically sorted order, adding an SCC each time a dead end is reached.



- topsort: [a, b, e, f, g, c, d, h, i]
- seen: {a,b,c,d,e,g,f}
- sccs: [{a,b,c,d},{e,g,f}]

Kosaraju's Algorithm for Finding Strongly Connected Components

3. Traverse the graph in topologically sorted order, adding an SCC each time a dead end is reached.



- topsort: [a, b, e, f, g, c, d, h, i]
- seen: {a,b,c,d,e,g,f,h,i}
- sccs: [{a,b,c,d},{e,g,f},{h,i}]

Code for Kosaraju's SCC Algorithm

```
def tr(G):                          # Transpose (rev. edges of) G
    GT = {}
    for u in G: GT[u] = set()       # Get all the nodes in there
    for u in G:
        for v in G[u]:
            GT[v].add(u)            # Add all reverse edges
    return GT

def scc(G):
    GT = tr(G)                      # Get the transposed graph
    sccs, seen = [], set()
    for u in iter_dfs_topsort(G): # DFS starting points
        if u in seen: continue    # Ignore covered nodes
        C = walk(GT, u, seen)     # Don't go "backward" (seen)
        seen.update(C)            # We've now seen C
        sccs.append(C)            # Another SCC found
        print sccs
```

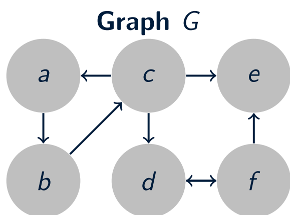
Code for Kosaraju's SCC Algorithm (topological sort)

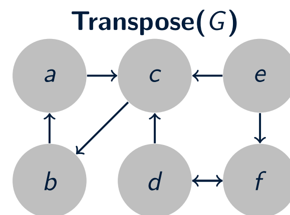
```
def iter_dfs_topsort(G):
    S, Q, res = set(), [], []  # Visited-set and queue
    for u in G:                #Cover entire graph
        Q.append(u)
        while Q:               # Planned nodes left?
            v = Q.pop()        # Get one
            if u in S: continue
            S.add(v)           # We've visited it now
            Q.extend(G[v])
        res.append(u)     # Postorder process: add node when finished
    res.reverse()
    return res
```


Code for Kosaraju's SCC Algorithm (graph traversal)

```
def walk(G, s, S=set()):#Walk the graph from node s
    P, Q = dict(), set()#Predecessors + "to do" queue
    P[s] = None         #s has no predecessor
    Q.add(s)            #We plan on starting with s
    while Q:            #Still nodes to visit
        u = Q.pop()     #Pick one, arbitrarily
        for v in G[u].difference(P, S): # New nodes?
            Q.add(v)    #We plan to visit them!
            P[v] = u    #Remember where we came from
    return P            #The traversal tree
```

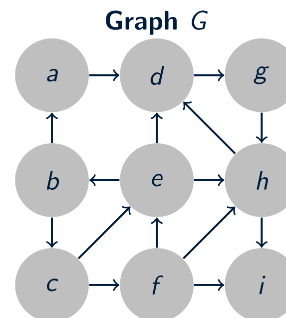

Exercise 1: Apply Kosaraju's SCC Algorithm

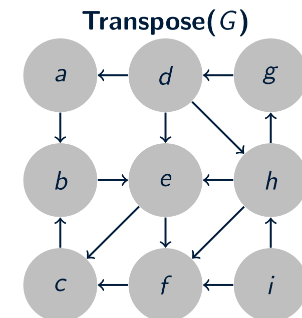




- What is the topological sort of G? Let's make the DFS tree starting from a
- What are the strongly connected components?

Exercise 2: Apply Kosaraju's SCC Algorithm





- What is the topological sort of G? Let's make the DFS tree starting from a
- What are the strongly connected components?