COMPSCI H311: Honors Colloquium – Introduction to Algorithms

Lecture 1: Algorithms for Strongly Connected Components

Marius Minea
marius@cs.umass.edu

University of Massachusetts Amherst

February 2, 2021

Colloquium Topics (tentative)

1. Strongly Connected Components.
2. Amortized Analysis
3. Huffman Codes and Data Compression
4. Generating Functions for Recurrence Relations
5. Convolutions and the Fast Fourier Transform
6. RNA Secondary Structure Prediction
7. Advanced Network Flow Algorithms
8. Network Flow Applications
9. Co-NP and the Asymmetry of NP
10. Space complexity and PSPACE
11. Approximation Algorithms
12. Local Search
13. Randomization Algorithms

Logistics

Grading:

70% homeworks ($\sim$1 problem / week)

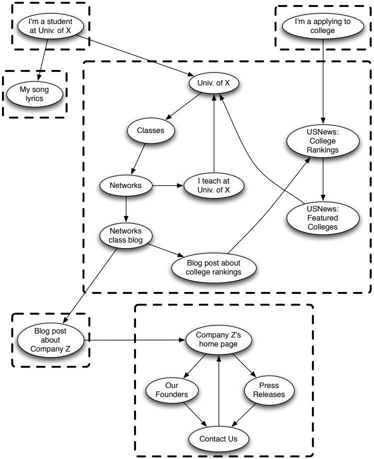
30% final project (discuss/agree topic))

Today

Strongly Connected Components – Three Algorithms

- Tarjan
- Kosaraju
- Path-based

Strongly Connected Components



In a SCC, there is a path from every node x to every node y.

A SCC is a *maximal* subgraph with this property.

Equivalent: $\forall x, y$, path $x \rightsquigarrow y$ and path $y \rightsquigarrow x$ (swap x and y)

Trivial SCC: single node

How to Compute a SCC?

Idea 1: path $x \rightsquigarrow y$ and $y \rightsquigarrow x$:
  compute set of nodes reachable *from* x
  compute set of nodes *backwards* reachable from x
  intersect
Problem: set intersection may be expensive

Idea 2: look for cycles (all nodes in a cycle are in same SCC).
DFS from some node x: any back edge closes a cycle.

Issue: what about back edges further up / down ?

All three algorithms: recursive DFS, plus extra bookkeeping

Tarjan's Algorithm (1972)

Recover SCC as subtrees of DFS spanning forest
Root of SCC = first SCC node reached by DFS

- number nodes as discovered (v.index)
- keep explicit stack of visited nodes
  (may not immediately pop on return from recursion);
- keep v.onStack flag for each node v

Invariant: node stays on stack iff it has path to ancestor
v.lowlink: highest known reachable ancestor (w/ lowest index)
– keep on stack if v.lowlink $<$ v.index
– remove / set as root if v.lowlink $=$ v.index

Tarjan's Algorithm (cont'd)

Updates when reaching successor $v \to w$:

if w unexplored — tree edge

  explore

  v.lowlink = min(v.lowlink, w.lowlink)

(if w reaches ancestor of v, so can v through w)

else if w is on stack — w is above v (back edge)

  v.lowlink = min(v.lowlink, w.index)

(have found path from v to ancestor w).

else no change — forward/cross edge

Examples

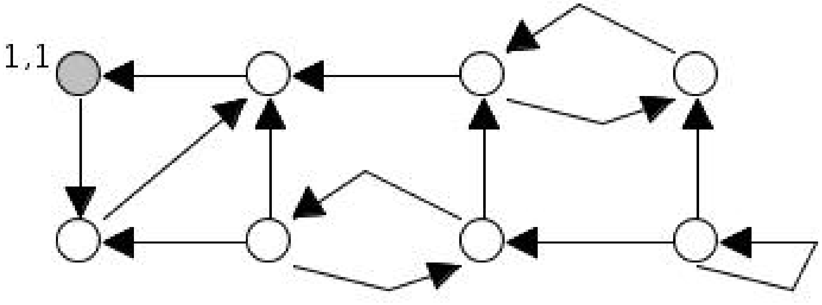


Figure: https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

Tarjan's Algorithm (cont'd)

```
index = 0; S = empty
for all v do
    if v not visited then
        SCC(v)
function SCC(v)
    v.lowlink = v.index = ++index
    S.push(v); v.onStack = true;
    for all neighbors w of v do
        if w not visited then
            SCC(w)
            v.lowlink = min(v.lowlink, w.lowlink)
        else if w.onStack then
            v.lowlink = min(v.lowlink, w.index)
    if v.lowlink = v.index then
        pop nodes from S until v into SCC(v)
```

Kosaraju SCC Algorithm (1978)

Two DFS calls, on graph and reversed graph.

In DFS, prepend nodes to list L in *postorder* (fully explored).
L will have nodes in reverse order of finishing times.

Then DFS the reverse graph, each tree is a SCC.

Insight:
If there is only a path $u \rightsquigarrow v$, then u will be ahead of v in list L
We can't reach v from u in $G^R \implies$ different SCCs

If there are paths $u \rightsquigarrow v$ and $v \rightsquigarrow u$, there could be any order.

If we reach v when searching from u in G^R, then there is also a $v \rightsquigarrow u$ path in G, so they are in the same SCC.

Example

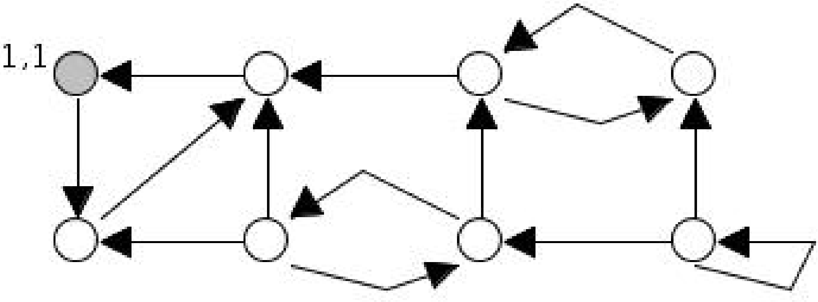


Figure: https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

Path-Based SCC Algorithm (Dijkstra 1976)

Maintains *two* stacks (in addition to recursion)
– stack S: all vertices unassigned to an SCC, in order reached
– stack P: vertices that are not known to belong to different SCCs (new path)
– and running number for each vertex (in order discovered)

function $\textsc{search}(v)$
  push v onto S and P
  for all neighbors w of v **do**
    if w not visited **then**
      $\textsc{search}(w)$
    else if w unassigned **then**
      pop from P until top(P) has number $\leq w$
  if $v = \text{top}(P)$ **then**
    pop from S until v into new SCC

Intuition: nodes on a path segment P contracted to single node

Example

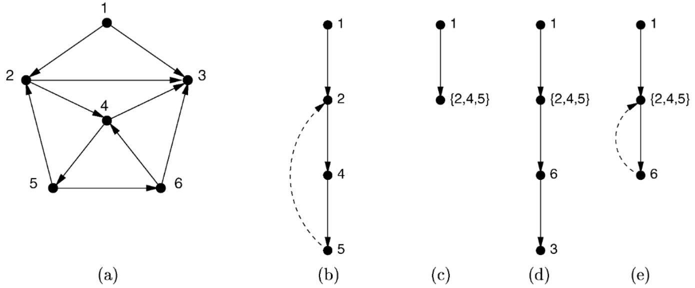


Fig. 1. (a) Digraph G. (b)–(e) Path P (solid edges) in the first several steps of the algorithm. Strong component {3} is output in (d).

Wrap-Up and Similar Algorithms

Complexity: all these algorithms have linear complexity,
$O(|V| + |E|)$
(based on DFS, plus constant-time work per node or edge)

In undirected graphs: *biconnected* components
$=$ subgraphs that stay connected when removing any one node

Biconnected components have *articulation points* in common
(node that when removed disconnects the graph)