Chapter 3 Decomposition of graphs

Joe Song

September 11, 2019

The notes are based on Chapter 3 of Dasgupta, Papadimitriou and Vazirani. Algorithms. 2008. McGraw-Hill. New York.

1 Introduction

Some graph problems can be very challenging.

The graph coloring problem:

> Color a map with the minimum number of colors so that adjacent countries do not share the same color.
>
> $\equiv$
>
> Scheduling exams so that the exams any student must take will not overlap in time.

A planar graph only needs 4 colors—it takes ~100 years to get a correct proof!

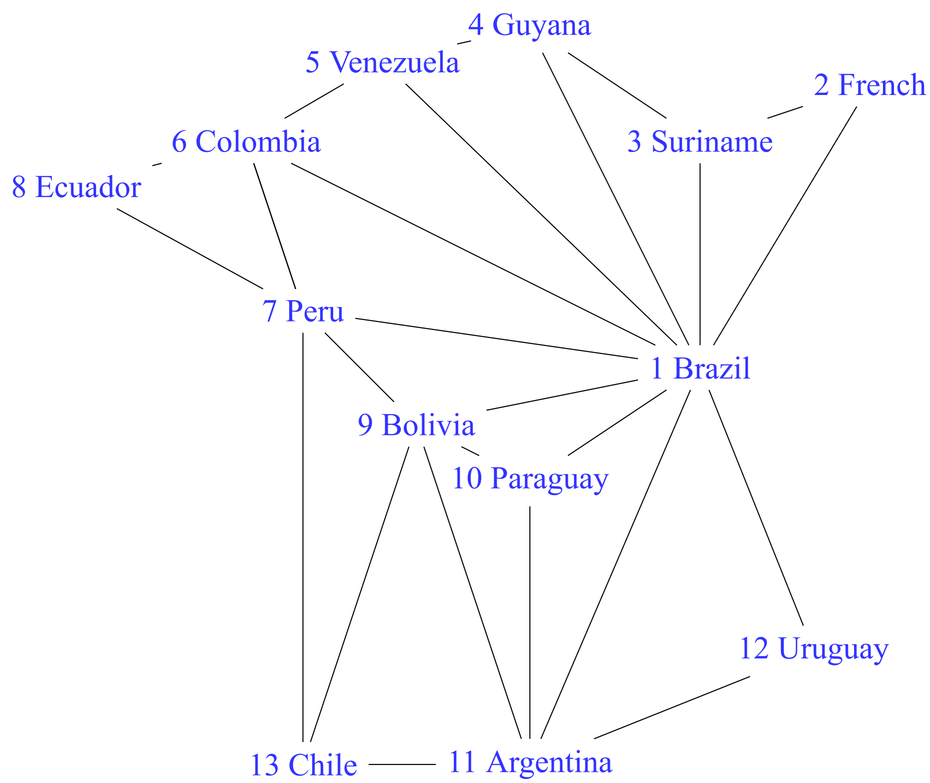
1879 Alfred Kempe published a paper saying 4 colors are maximum needed for planar graphs.

Fellow of the Royal Society

President of the London Mathematical Society

1890 Heawood said Kempe's proof is wrong! Instead he proved no more than 5 colors are needed!

1976 No more than 4 colors is finally proved by Kenneth Appel and Wolfgang Haken.



We will focus on graph problems that are efficiently solvable.

Definition of a *graph* G:

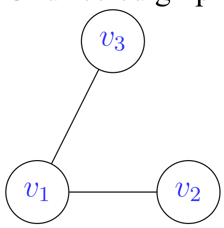
- a set of vertices (nodes) V
- a set of edges E. Each edge connects a pair of vertices.

Example of the above graph: $V = \{1, 2, \ldots, 13\}$, $E = \{(1, 2), (9, 11), \ldots\}$.

Directed edge (x, y) is directional from node x to y. (y, x) is from y to x.

1.1 Representation

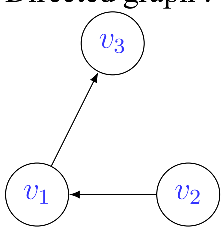
Undirected graph : → **Adjacency MATRIX** **Adjacency LIST**



	v_1	v_2	v_3
v_1	0	1	1
v_2	1	0	0
v_3	1	0	0

$v_1 : (v_3 \to v_2)$
$v_2 : (v_1)$
$v_3 : (v_1)$

Directed graph : → **Adjacency MATRIX** **Adjacency LIST**



	v_1	v_2	v_3
v_1	0	0	1
v_2	1	0	0
v_3	0	0	0

$v_1 : (v_3)$
$v_2 : (v_1)$
$v_3 : ()$

Adjacency matrix:

$A = \{a_{ij}\}$ for $V = \{v_1, \ldots, v_n\}$.

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Properties:

- Space complexity $O(|V|^2) = O(n^2)$—expensive!
- Checking the existence of edge (u, v) takes *constant* time $O(1)$—fast!

Adjacency list:

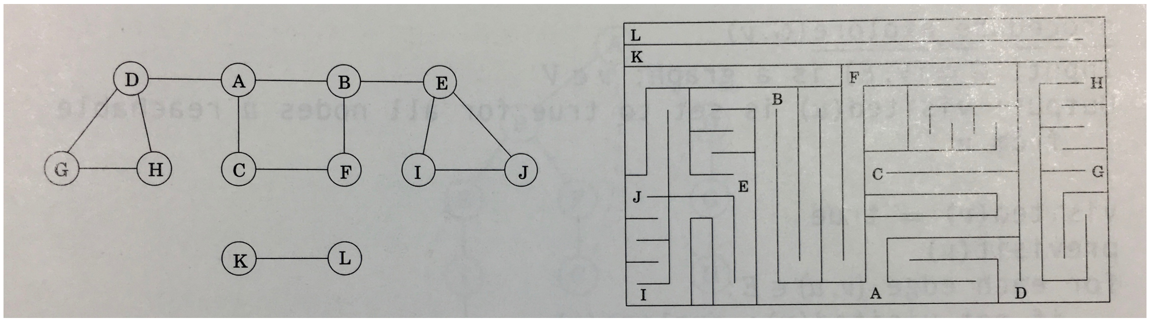
- Consists of $|V|$ linked lists, one for each node

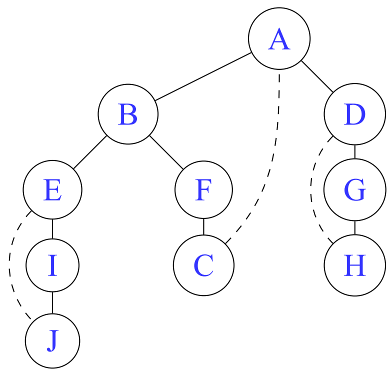
- The list for vertex u holds names of vertices that u has an outgoing edge.

Properties:

- undirected edges will be represented twice
- Space complexity $O(|V| + |E|)$
- Checking the existence of edge (u, v) can take $O(|E|)$ time

2 Depth-first search in undirected graphs





Exploring all reachable nodes from a single given node:

---

procedure explore(G, v)
Input: $G = (V, E)$ is a graph; $v \in V$
Output: visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)   (optional)
for each edge $(v, u) \in E$:
    if not visited(u): explore(G, u)
postvisit(v)   (optional)

Correctness:

Assume there is a path from v to u that is unexplored, shown as below.

v (source) ———————— z (visited) - - - w (unexplored) - - - - - - - - - - - u (unexplored)

We can find the last visited node z on the path (we can always find such one), anything after z is unvisited, where the first unvisited one is w with an edge with z. However, this leads to a contradiction as when z is explored, it would have noticed w and explored it. Therefore, we have a contradiction. Thus, we must all all nodes reachable from v visited after running the explore procedure.

---

procedure dfs(G)

for all $v \in V$:
    visited(v) = false

for all $v \in V$:
    if not visited(v): explore(G, v)

Running time:

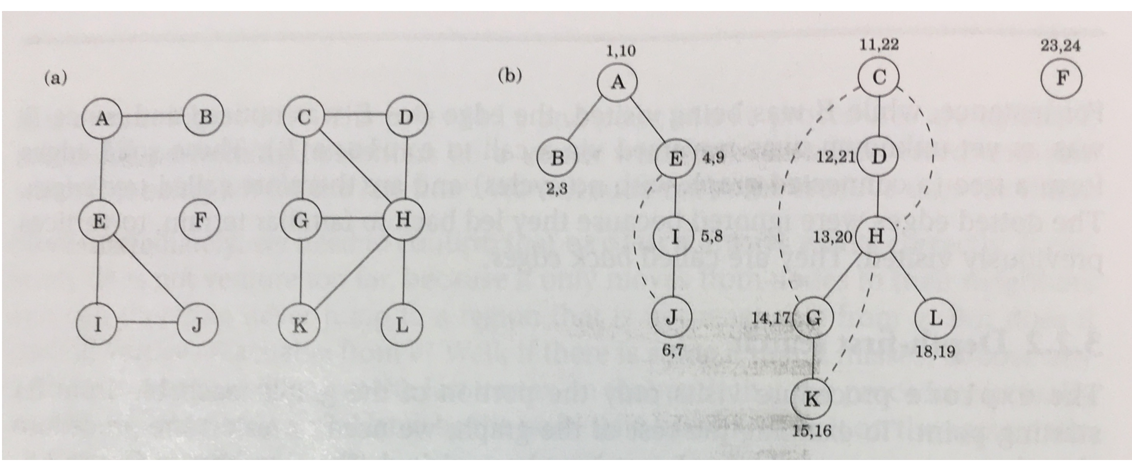
For each node, a constant amount of work is done

For each edge (x, y), it is visited twice: once in explore(x) and once in explore(y).

So the total running time is

$$O(|V| + |E|)$$

Connectivity:



An undirected graph is *connected* if there is a path between every pair of nodes.

A *connected component* is a subgraph that is internally connected but has no edges to the remaining nodes.

---

procedure previsit(v)
ccnum[v] = cc

- cc is the label of the current connected component, which is initialized to zero and increased every time the explore() procedure is called from dfs().
- ccnum[v] is the connected component number of node v.

Previsit/Postvisit ordering:

pre[v] is the time first discovery of node v.

post[v] is the time of final departure of node v.

---

procedure previsit(v)
pre[v] = clock
clock = clock + 1

---

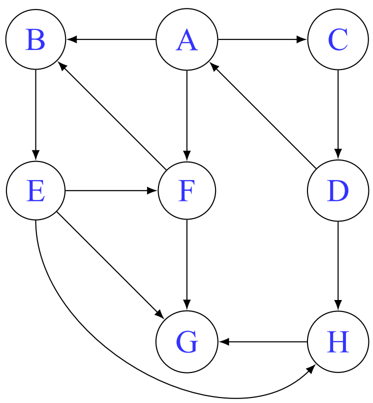
procedure postvisit(v)
post[v] = clock
clock = clock + 1

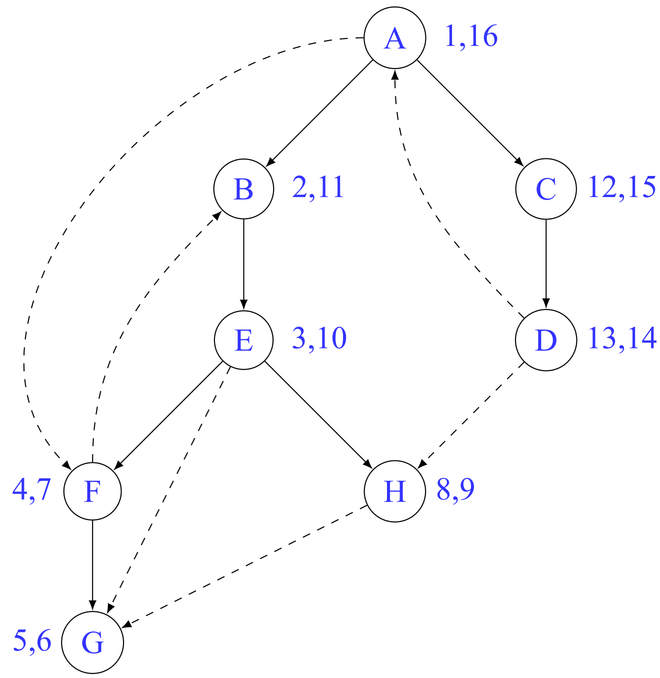
Property: For any nodes u and v, the two intervals [pre(u), post(u)] and [pre(v), post(v)] are either disjoint or one is contained within the other.

This is due to the last-in-first-out (stack) nature of the explore() procedure for node traversal.

3 Depth-first search in directed graphs

DFS is the same as in the undirected graph except that an edge can only be traversed along its direction.





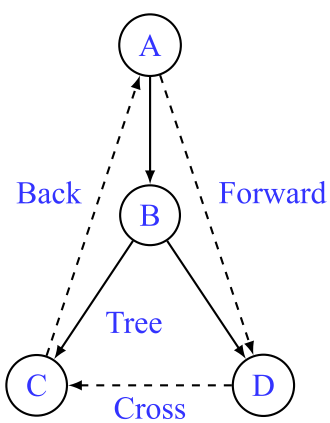
root: The starting node A

descendant: Everything else is descendant of A

ancestor: E is an ancestor of F, G, H

parent: C is the parent of D

child: D is the child of C



Tree edges: an edge of the DFS forest

Forward edges: a non-DFS-forest edge from a node to a non-child descendant in the DFS tree

Back edges: a non-DFS-forest edge from a node to an ancestor in the DFS tree

Cross edges: a non-DFS-forest edge leading to neither descendant nor ancestor – i.e., a node already been completely explored (i.e., postvisited)

The type of edge (u, v)	pre/post ordering for (u, v)
Tree/Forward	$pre[u] < pre[v] < post[v] < post[u]$
Back	$pre[v] < pre[u] < post[u] < post[v]$
Cross	$pre[v] < post[v] < pre[u] < post[u]$

Directed acyclic graphs (dags)

cycle: is a circular path $v_0 \to v_1 \to v_2 \to \ldots \to v_k \to v_0$.

Directed acyclic graph (dag): a directed graph without cycles.

Property of directed graph: A directed graph has a cycle if and only if its depth-first search reveals a back edge.

The "IF" part: If there is path from A to D and there is a back edge from D to A, then combining the path and the back edge would have formed a cycle.

"ONLY IF": (Proof by contradiction) If there is no back edge, no path can return to a node that is pre-visited earlier and it is impossible to have a cycle.

Property 1 of dag: A dag has no back edge on any DFS.

Property 2 of dag: In a dag, every edge leads to a vertex with a lower post number.

In a dag, there is no back edge, so the table above indicates that for every edge (u, v), post$[v] <$ post$[u]$.

This property suggests that one can *linearize* the nodes in a dag by decreasing post numbers. This is also called *topological sort*.

Property 3 of dag: Every dag has at least one source and at least one sink.

source node: a node with no incoming edges.

sink node: a node with no outgoing edges.

Proof by contradiction. If there is no source node, one can backtrack a path and eventually reach another node on the same path, which leads to a cycle.

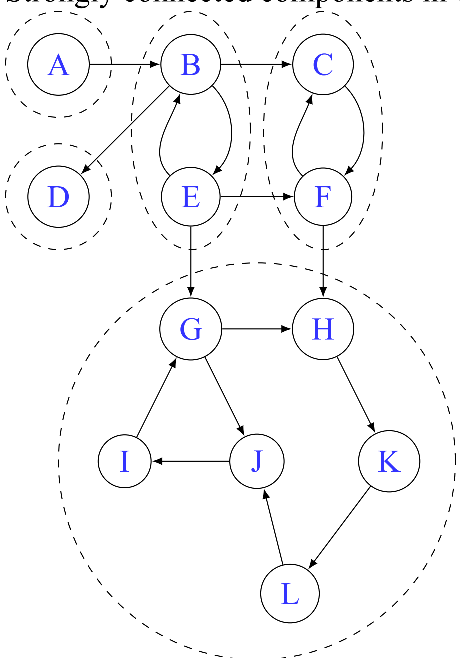
If there is no sink node, one can traverse a path and eventually reach another node already traversed on the same path, which leads to a cycle.

4 Strongly connected components

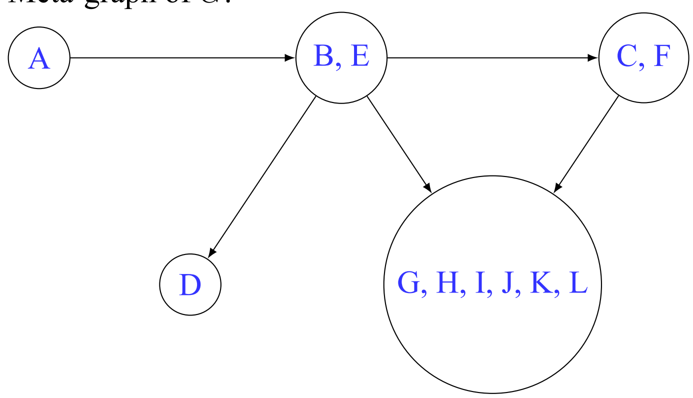
More subtle than the connectedness concept for undirected graphs.

Connectivity in a directed graph: Two nodes u and v are connected if there is a path from u to v and a path from v to u.

Strongly connected components in G:



Meta-graph of G:



Strongly connected components: Disjoint sets of connected nodes in a dag.

Meta-graph: Created by shrinking nodes in a strongly connected components to a single meta-node; Draw an edge from one meta-node to another if there is an edge between there respective components.

Property: The meta-graph of a directed graph is a dag.

Proof by contradiction. If there is a cycle in the meta-graph, then all strongly connected components on the cycle become one, because the cycle will link all nodes with the components, which is a contradiction that nodes in different components are disconnected.

4.1 Decomposition of a directed graph to strongly connected components

Ideas: Find a sink strongly connected component (on the meta-graph) first; Remove it from the graph; Find the sink strongly connected component of the remaining graph.

Considerations:

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.

This is evident and can be proved by contradiction.

Property 2 The node that receives the highest `post` number in a depth-first search must lie in a source strongly connected component.

This follows from Property 3.

Property 3 If C and C' are strongly connected components, 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 biggest `post` number in C'.

Proof:

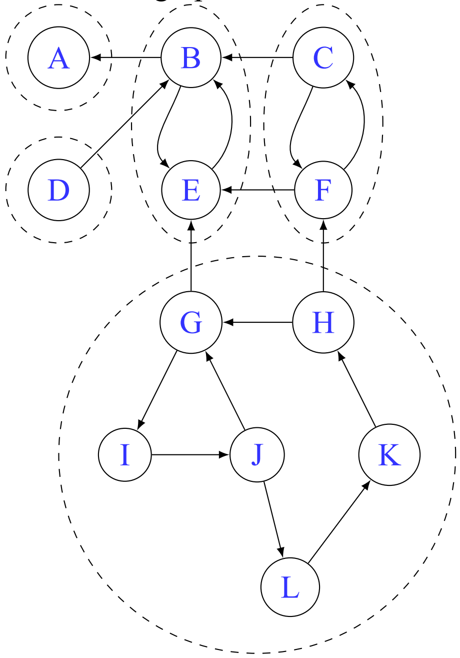
If the depth-first-search starts from a node in C first, it will traverse all nodes in C and C'. The start node obviously has the highest post number, which is in C.

If the depth-first-search starts from a node in C' first, it will traverse all nodes in C'. Then it will move on to a new node in C. The post number of any node in C will be greater than the post number in C'.

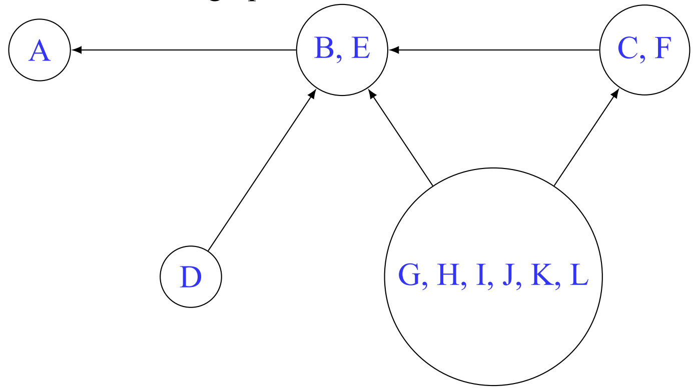
Implication:

If we create a reverse graph G^R by reversing all the edge directions in G, the node with the highest post number from a source strongly connected component in G^R is a node in a sink connected component in G.

Strongly connected components in reversed graph G^R:



Reversed meta-graph:



---

procedure Find-Strongly-Connected-Components(G)

Step 1. Run depth-first search on G^R

Step 2. Run the undirected connected components algorithm (but respect edge directions) on G, and during the depth-first search, process the vertices in decreasing order of their `post` numbers from Step 1.
