CS483 Analysis of Algorithms
Lecture 03 – Graphs

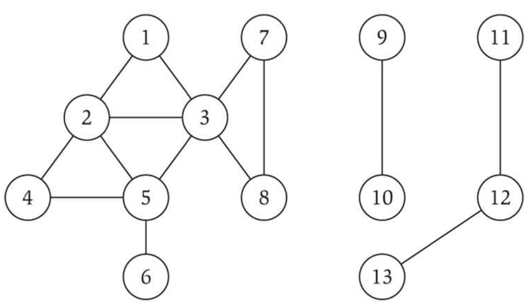
Jyh-Ming Lien

June 13, 2017

Introduction

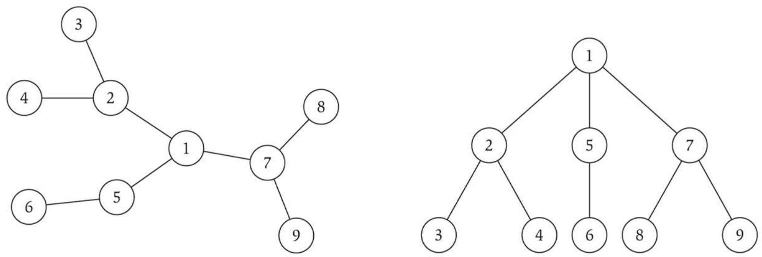
Graph Representation

- Terminology $G = (V, E)$
  - V = nodes or vertices $\{v\}$
  - E = edges between pairs of nodes, $\{e = (u, v)\}$, where u and v are called **ends** of e
  - For directed edge $e = (u, v)$ is an ordered list where u is the **tail** and v is the **head** and e **leaves** u and **enters** v.
  - A path is a sequence of vertices $v_1, v_2, \cdots, v_{k-1}, v_k$. A path is called **simple** if $v_i \neq v_j \forall i \neq j$
  - A cycle is a path $v_1, v_2, \cdots, v_{k-1}, v_k$ in which $v_1 = v_k, for k > 2$, and the first $k - 1$ nodes are all distinct
  - An undirected graph is **connected** if for every pair of nodes u and v, there is a path between u and v.



Graph and Tree

- An undirected graph G is a tree if
  - G is connected
  - G does not contain a cycle
  - G has $n - 1$ edges, where n is the number of nodes in G

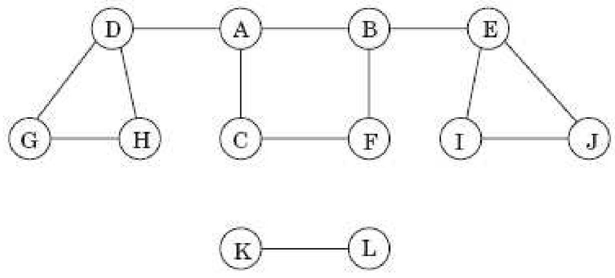


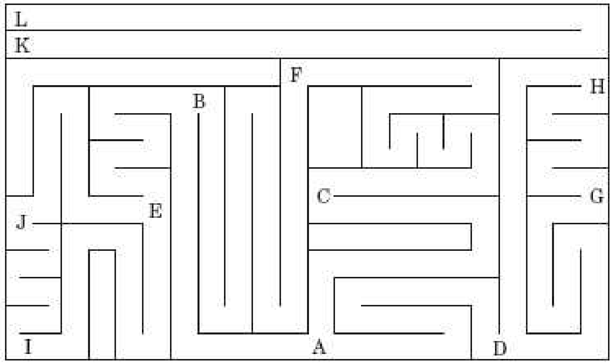
- Many algorithms work by converting a graph to a tree (the simplest representation of the graph)
  - shortest path tree
  - spanning tree
  - exploring tree (BFS, DFS, ...)
  - ...
  -

Explore graphs

Graph Search

- What parts of the graph are reachable from a given vertex? (i.e., connected components)
- Many problems require processing all graph vertices (and edges) in systematic fashion
- Basic tools to safely explore an unknown environment
  - Marker (mark places that you have visited) — a flag in each node/edge
  - Rope (to get you back to the start) — a stack





- Basic exploration algorithm

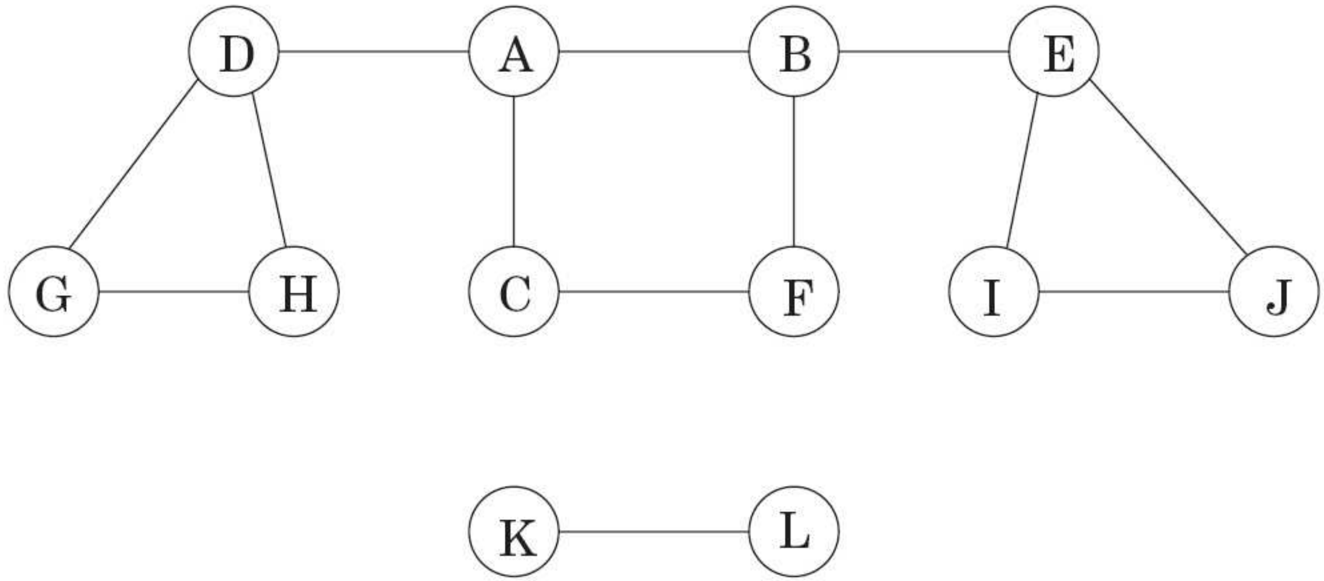
Algorithm 0.1: EXPLORE($G = \{V, E\}, v \in V$)

$v.visit \leftarrow$ **true**
previsit(v)
for each edge $(v, u) \in E$
  do $\begin{cases} \textbf{if } u.visit == \textbf{ false} \\ \quad \textbf{then } \text{EXPLORE}(G, u) \end{cases}$
postvisit(v)

- Can the algorithm always work?
  – *proof*

Graph Search

- Example: EXPLORE(B)



- DFS

Algorithm 0.2: DFS($G = \{V, E\}$)

for $v \in V$
  do $v.visit \leftarrow$ **false**
for $v \in V$
  do $\begin{cases} \textbf{if } !v.visit \\ \quad \textbf{then } \text{EXPLORE}(G, v) \end{cases}$

- BFS

Algorithm 0.3: $\text{BFS}(G, v)$

$v.visit \leftarrow$ **true**
$Q \leftarrow \emptyset$
Q.push(v)
while $Q \neq \emptyset$
  do {
    $v \leftarrow Q$.pop()
    (do something here)
    for each neighbor n of v
      do {
        if $n.visit ==$ **false**
          then {
            $n.visit \leftarrow$ **true**
            Q.push(n)

Algorithm 0.4: $\text{BFS}(G)$

for $v \in V$
  do $v.visit \leftarrow$ **false**
for $v \in V$
  do {
    if $!v.visit$
      then $\text{BFS}(G, v)$

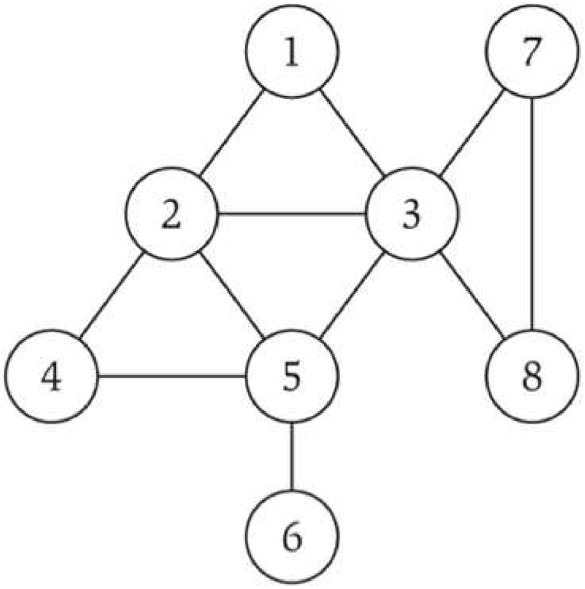
Breadth-first search

- BFS intuition: Explore outward from s in all possible directions, adding nodes one "layer" at a time.
- Another interpretation of BFS algorithm (a.k.a flooding):
  - $L_0 = \{v\}$.
  - L_1 = all neighbors of L_0.
  - L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
  - L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.
- Theorem: For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t iff t appears in some layer.
- Property: Let T be a BFS tree of $G = (V, E)$, and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.

  Proof:

Breadth-first search example

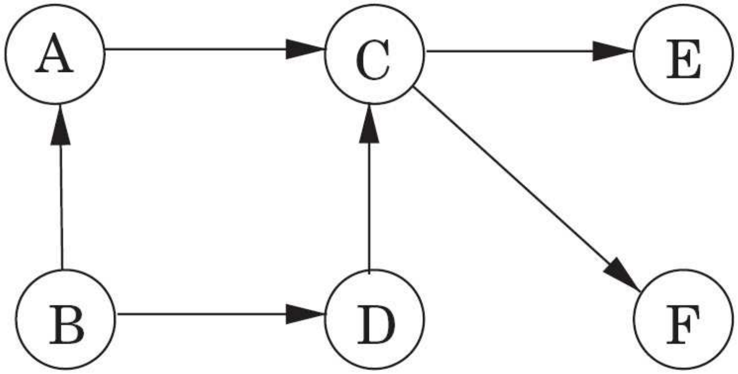
- Levels, BFS tree and cycles



Topological sort

Directed acyclic graphs

- A graph G without (directed) cycle is a *directed acyclic graphs* (DAG)
- DAG can be found in modeling many problems that involve prerequisite constraints (construction projects, document version control)

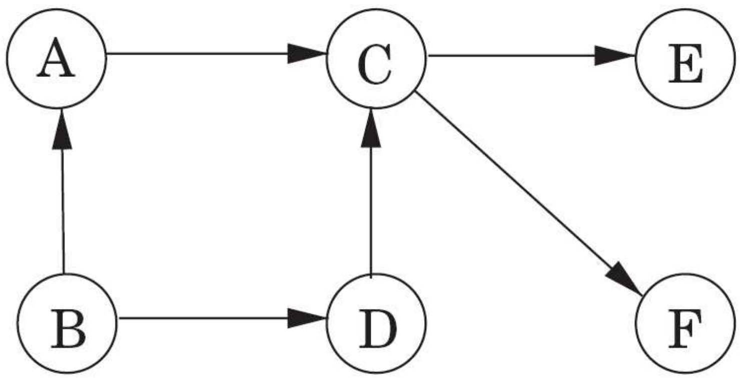


- Given a *directed* graph G, identify *cycles* in G

  – *proof*

DAG and Topological Sort

- **Topological sorting** or **Linearization**: Vertices of a DAG can be linearly ordered so that:
  - Every edge its starting vertex is listed before its ending vertex
  - Being a DAG is also a necessary condition for topological sorting be possible
- Example:



- Compute DSF and reverse the visit order

Algorithm 0.5: TS($G = \{V, E\}$)

$S \leftarrow \emptyset; L \leftarrow \emptyset$
S.push(v)
while $S \neq \emptyset$
do $\begin{cases} v \leftarrow S.\text{pop}() \\ L.\text{push\_front}(v) \\ \textbf{for each } \text{neighbor } n \text{ of } v \\ \quad \textbf{do } S.\text{push}(n) \end{cases}$
return (L)

- Why does it work?

- Time complexity?

Topological Sort: Using Source Removal

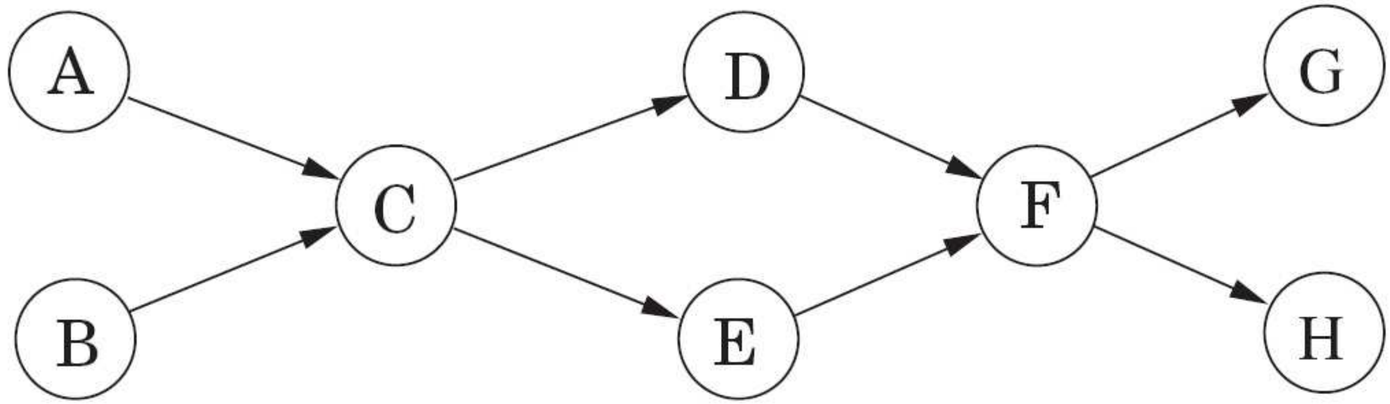
- Identify and remove sources iteratively.
  - A source is a vertex without incoming edges.

Algorithm 0.6: TS$(G = \{V, E\})$

$Q \leftarrow \emptyset; L \leftarrow \emptyset$
for each $v \in V$
  do $\begin{cases} \textbf{if } v \text{ is a source} \\ \quad \textbf{then } Q.\text{push}(v) \end{cases}$
while $S \neq \emptyset$
  do $\begin{cases} v \leftarrow Q.\text{pop}() \\ L.\text{push\_front}(v) \\ \textbf{for each } \text{neighbor } n \text{ of } v \\ \quad \textbf{do } \begin{cases} \textbf{if } n \text{ is a source} \\ \quad \textbf{then } Q.\text{push}(n) \end{cases} \end{cases}$
return (L)

- Why does it work?
- Time complexity?

- Example:

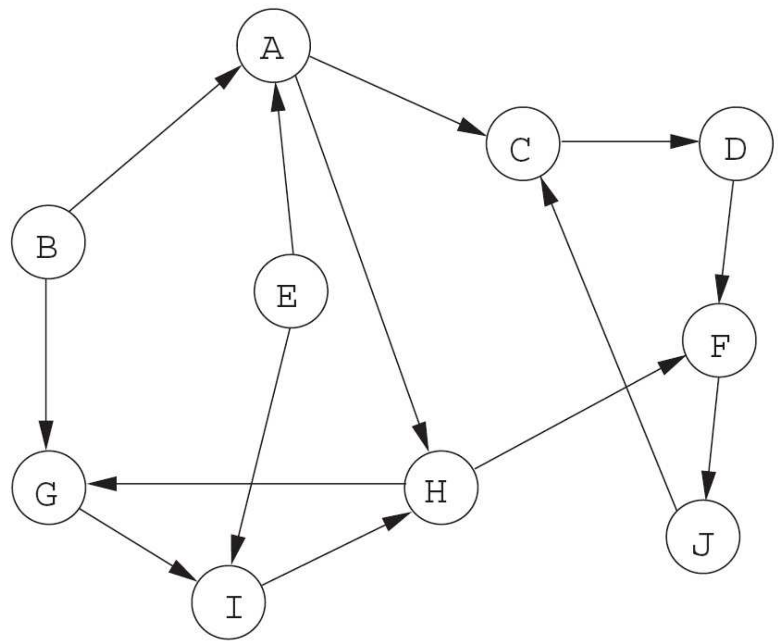


Strong connected components

Strongly connected components

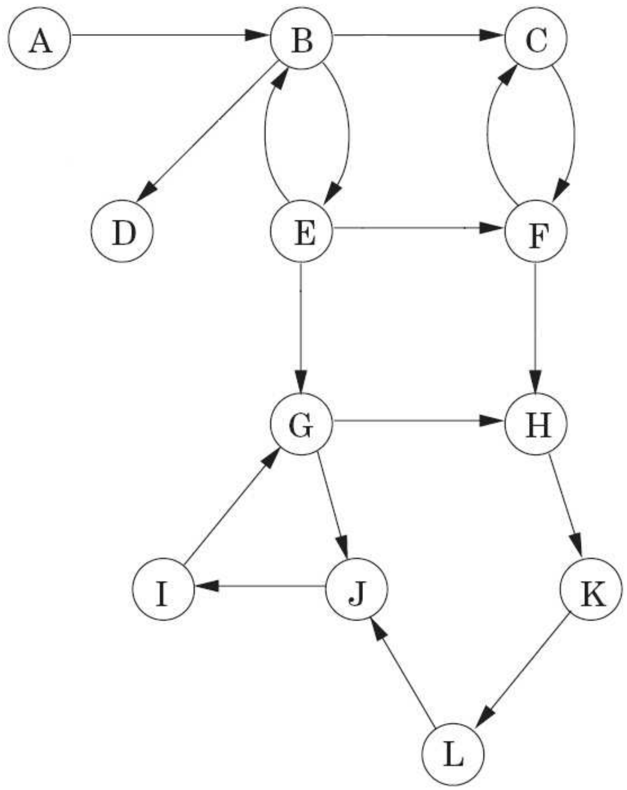
- **Definition**: Two nodes u and v are from the connected if and only if there is a path from u to v and a path from v to u.

- **Definition**: A set of vertices form a strongly connected component (SCC) iff any pairs of vertices are connected.



Strongly connected components

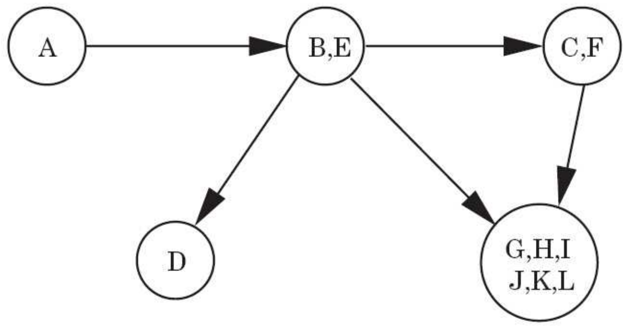
- Connected components in directed graph is less intuitive than that of undirected graph.
  - How many connected components are there in the graph below?



- How to compute SCCs from a directed graph?

Strongly connected components and DAG

- **Observation 1**: If we collapse each SCC to a node, the a obtain a



DAG!

- **Observation 2**: If we pick a node in the *sink node* (of the DAG) and run DFS from that node, then we will reveal all vertices in the *sink node*.
- Our strategy to find all SCC:
  - Find a vertex u in DAG sink node (?)
  - Final all vertices that is reachable from u and mark all from same SCC (easy)
  - Remove all nodes in the SCC and repeat until no nodes left (?)

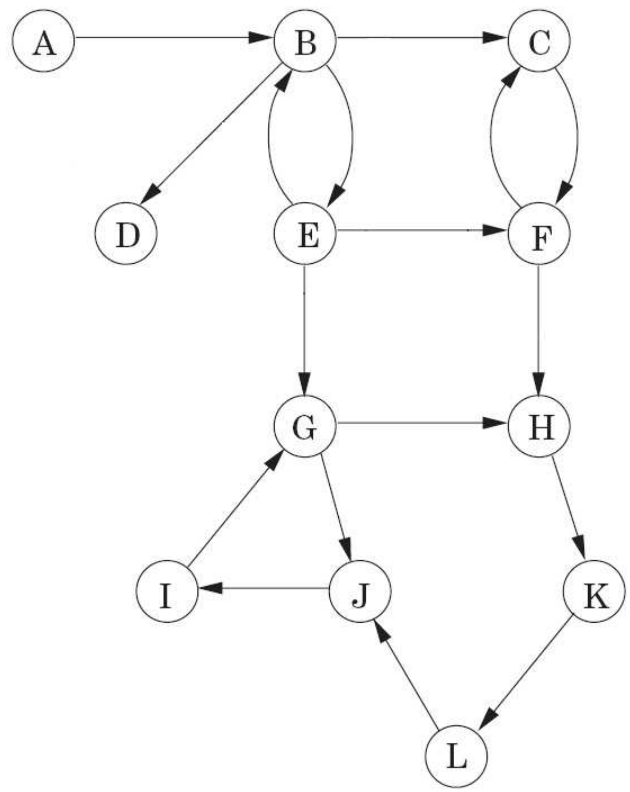
Strongly connected components and DAG

- **Task**: Find a vertex u in DAG sink node
- **Fact**: It is easier to find a **DAG** source node.
  - A node with largest *post* number is DFS must be in a DAG source node
  - If C and C' are SSCs and there is an edge from C to C', the largest *post* of C must be larger than that of C'.
    - *proof*:
    - Either start DFS from C or C'. If DFS starts from C' then all nodes in C' must finish before starting another DFS of C. If DFS starts in C, then the node that DSF starts from must have the largest post number.

- How to find a vertex u in DAG sink node?
  - Compute G^R whose vertices are the vertices of G and edges are the reverse of the edges of G
  - Perform DFS on G^R and the node with the largest *post* number of a node in the DAG sink node

Strongly connected components and DAG

- **Task**: Remove all nodes from the previous SCC and identify a new sink node
  - This can be done by marking *post* number in the previous SCC to -1 and find the node with the largest *post* number.
- Example:



Conclusion

Summary

- Graphs can be very useful for many problems.
- DFS can be used for
  - Explore the graph
  - Reveal relationship between the graph nodes and types of edges
  - Linearization for DAG
  - Identify cycles, connected components, strongly connected components