Applications of DFS and BFS

Slides by Carl Kingsford

Feb. 3, 2014

Based on/Reading: Chapter 3 of Kleinberg & Tardos

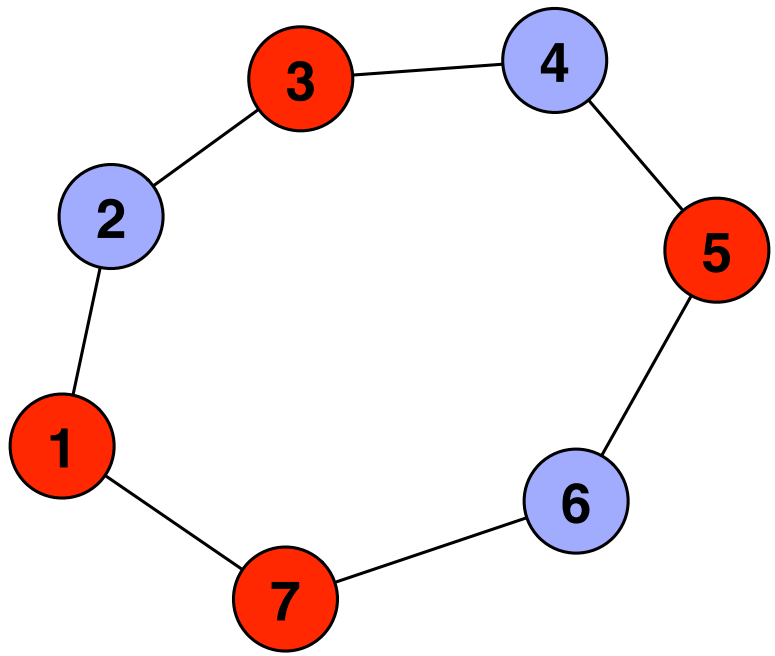
An Application of BFS

Testing Bipartiteness

A graph is bipartite if you can divide its nodes into 2 parts $(S, |V| - S)$ (a cut) so that every edge goes between the two parts S and $|V| - S$ and not within a single part.

Problem: Determine if a graph G is bipartite.

Bipartite graphs can't contain odd cycles:



Bipartite Testing

How can we test if G is bipartite?

Bipartite Testing

How can we test if G is bipartite?

- Do a BFS starting from some node s.
- Color even levels "blue" and odd levels "red."
- Check each edge to see if any edge has both endpoints the same level.

Bipartite Testing

How can we test if G is bipartite?

- Do a BFS starting from some node s.
- Color even levels "blue" and odd levels "red."
- Check each edge to see if any edge has both endpoints the same level.

Notice: Nodes in adjacent levels *must* get different colors because by construction there are edges between adjacent levels.

Therefore, the coloring above, is the only possible coloring (up to renaming the colors).

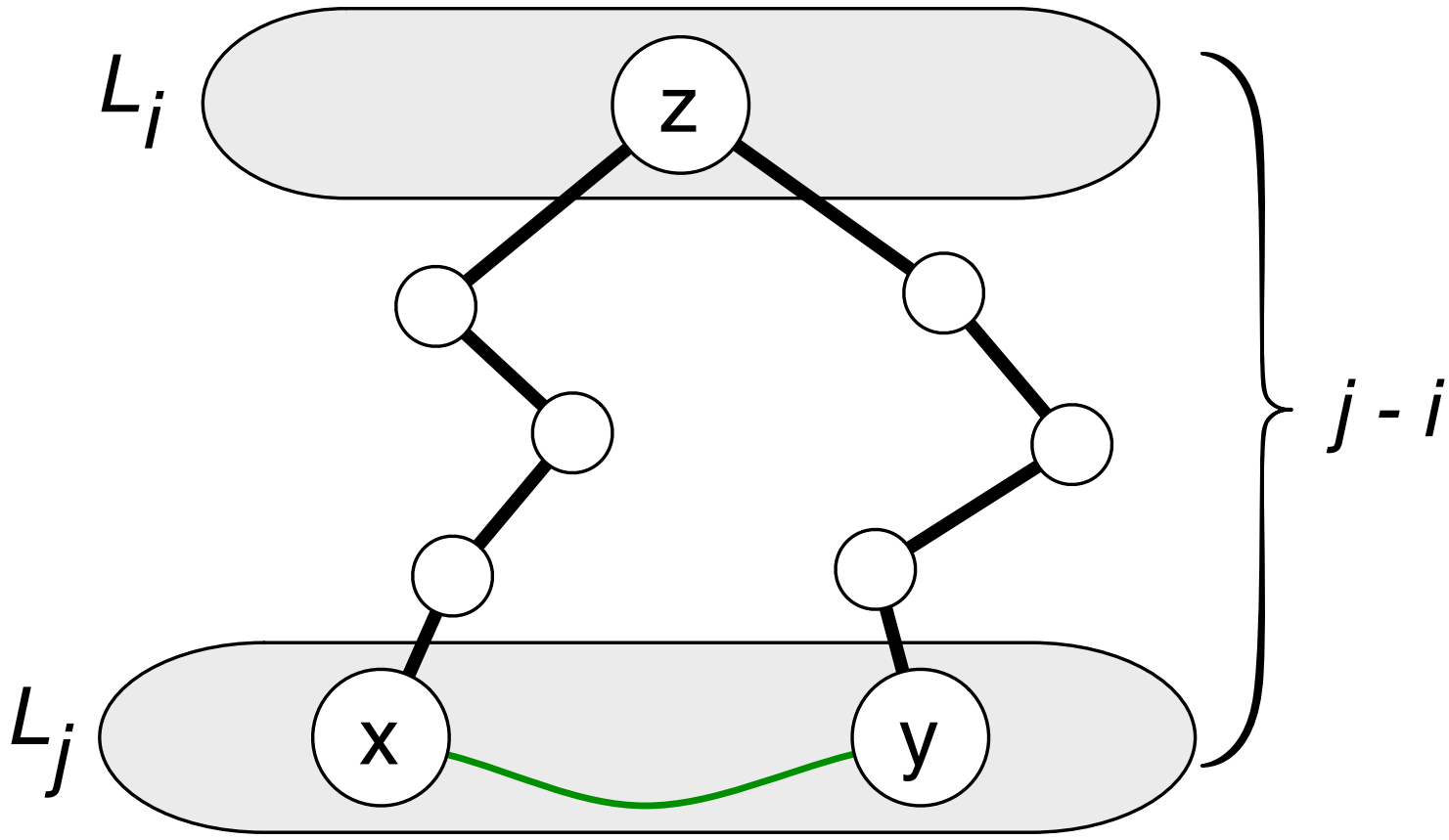
Proof of Correctness for Bipartite Testing

One of two cases happen:

1. There is no edge of G between two nodes of the same layer. In this case, *every* edge just connects two nodes in adjacent layers. But adjacent layers are oppositely colored, so G must be bipartite.

2. There is an edge of G joining two nodes x and y of the same layer L_j. Let $z \in L_i$ be the least common ancestor of x and y in the BFS tree T.
   $z - x - y - z$ is a cycle of length $2(j - i) + 1$, which is odd, so G is not bipartite.

Correctness, Picture

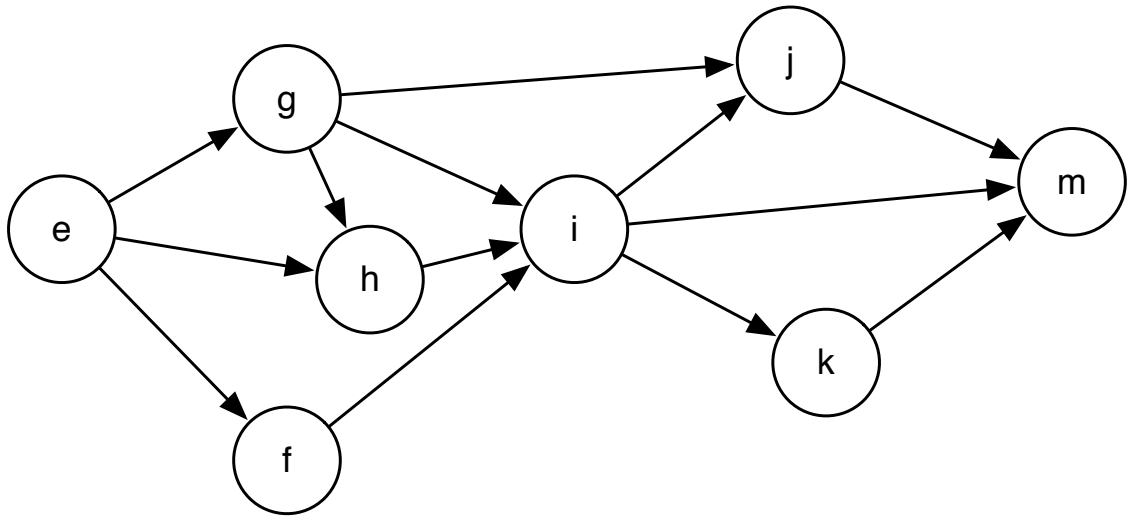


$x - z - y - x$ is a cycle of length $2(j - i) + 1$, which is an odd number.

An Application of DFS

DAGs

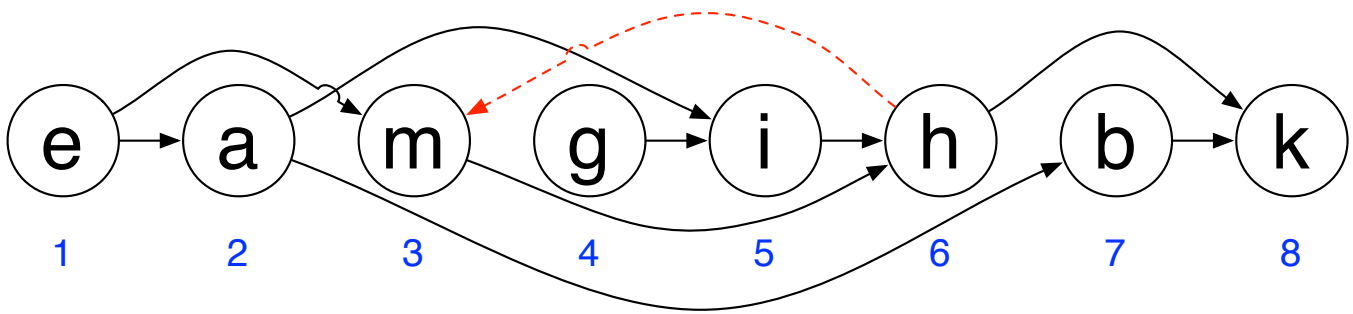
- A directed, acyclic graph (DAG) is a graph that contains no directed cycles. (After leaving any node u you can never get back to u by following edges along the arrows.)
- DAGs are very useful in modeling project dependencies: Task i has to be done before task j and k which have to be done before m.



Topological Sort

Given a DAG D representing dependencies, how do you order the jobs so that when a job is started, all its dependencies are done?

Topological Sort. Given a DAG $D = (V, E)$, find a mapping f from V to $\{1, \ldots, |V|\}$), so that for every edge $(u, v) \in E$, $f(u) < f(v)$, and each integer $1 \leq i \leq |V|$ is mapped to exactly once.



f(e) = 1
f(a) = 2
...

Topological Sort, II

Theorem. *Every DAG contains a vertex with no incoming edges.*

Topological Sort, II

Theorem. *Every DAG contains a vertex with no incoming edges.*

Proof. Suppose not.

Then keep following edges backward and in fewer than $n + 1$ steps you'll reach a node you've already visited.

This is a directed cycle, contradicting that the graph is a DAG. □

Topological Sort, II

Theorem. *Every DAG contains a vertex with no incoming edges.*

Proof. Suppose not.

Then keep following edges backward and in fewer than $n + 1$ steps you'll reach a node you've already visited.

This is a directed cycle, contradicting that the graph is a DAG. □

How can we turn this into an algorithm?

Topological Sort Algorithm

Topological sort:

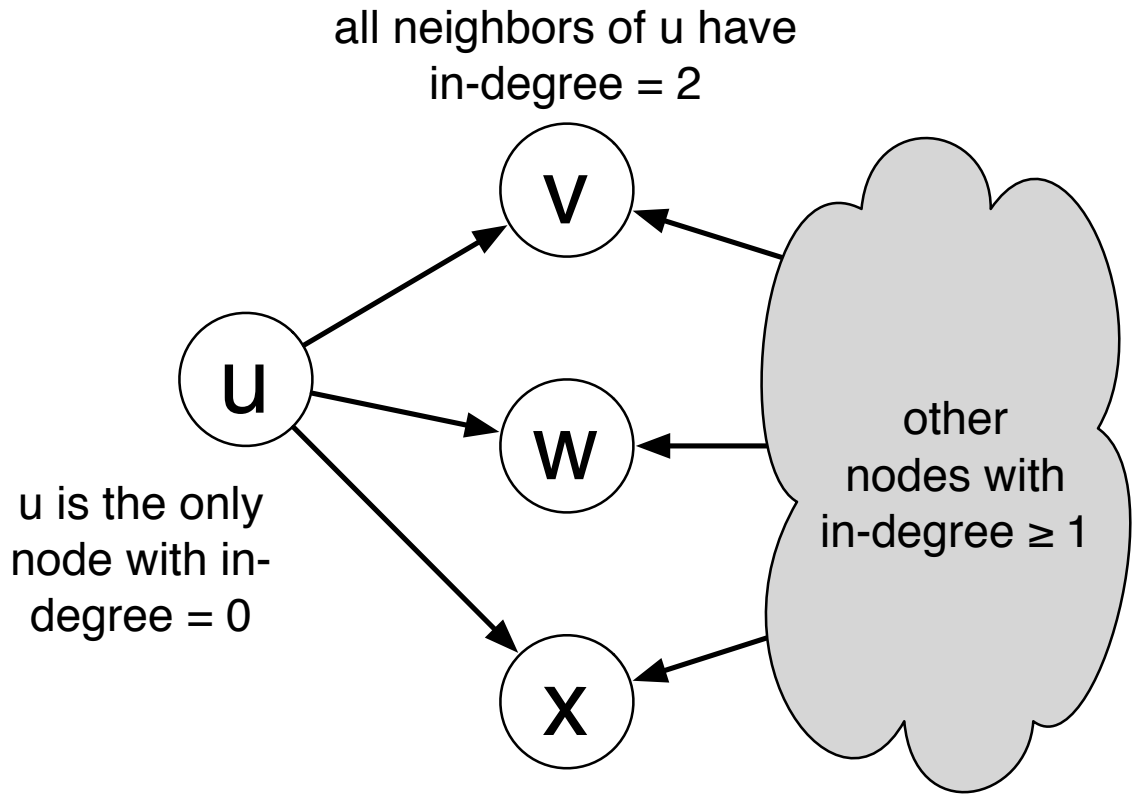
1. Let $i = 1$
2. Find a node u with no incoming edges, and let $f(u) = i$
3. Delete u from the graph
4. Increment i

Implementation: Maintain

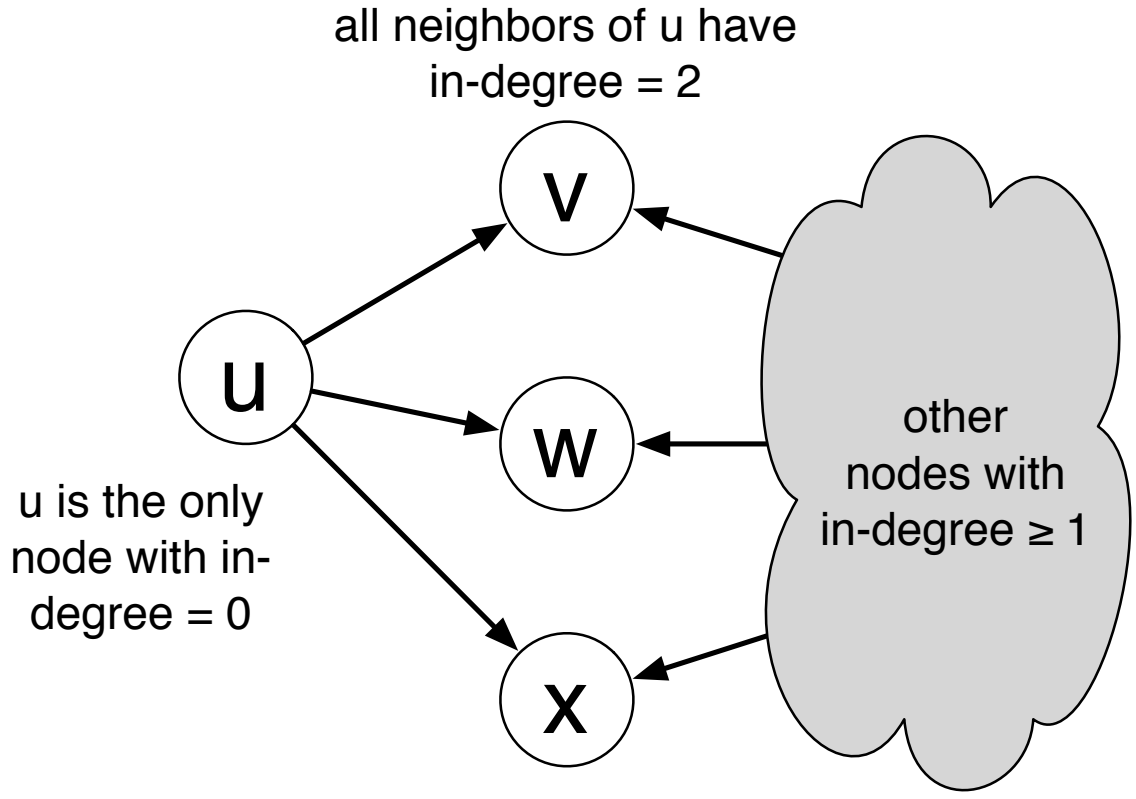
- `Income[w]` = number of incoming edges for node w
- a list S of nodes that currently have no incoming edges.

When we delete a node u, we decrement `Income[w]` for all neighbors w of u. If `Income[w]` becomes 0, we add w to S.

Can we get stuck in this case?



Can we get stuck in this case?



No, because the above can't happen: after removing u, the remaining graph is still a DAG.

Topological Sort Running Time

1. Initialize `Income[w]` array:
   - Count the number of edges adjacent to each node.
   - Implement via BFS or DFS.
   - Total running time $O(|V| + |E|)$.

2. For each node that currently has no incoming edges, decrement an entry in the income array, and possibly add something to S
   - This executes $O(|V|)$ times.
   - Handling each neighbor (decrement, add to S) takes $O(1)$ time.
   - So this entire step takes $O(|V| + |E|)$ time.

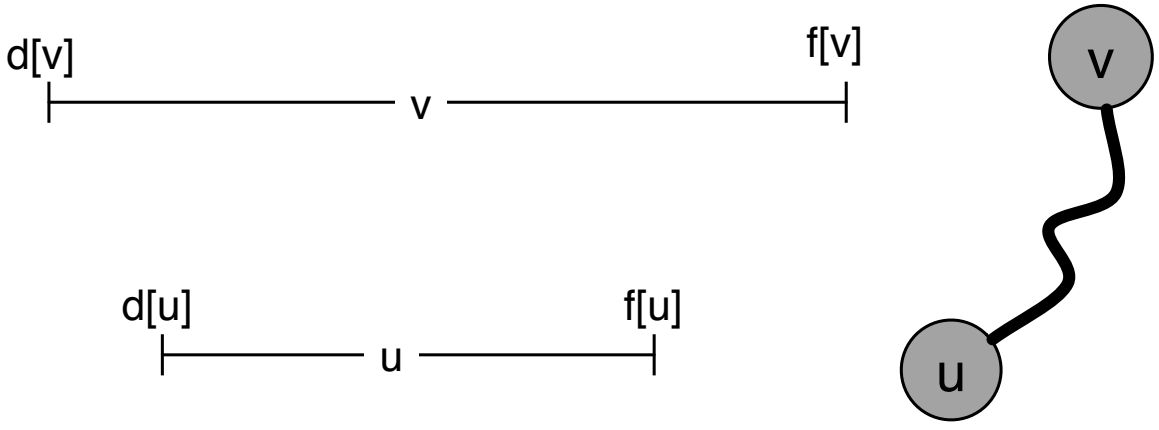
Total running time is: $O(|V| + |E|) + O(|V| + |E|) = O(|V| + |E|)$.
This is $O(|E|)$ for connected graphs.

Discovery and Finishing Times

DFS can be used to associate 2 numbers with each node of a graph G:

- discovery time: $d[u] =$ the time at which u is first visited
- finishing time: $f[u] =$ the time at which all u and all its neighbors have been visited.

Clearly $d[u] \leq f[u]$.

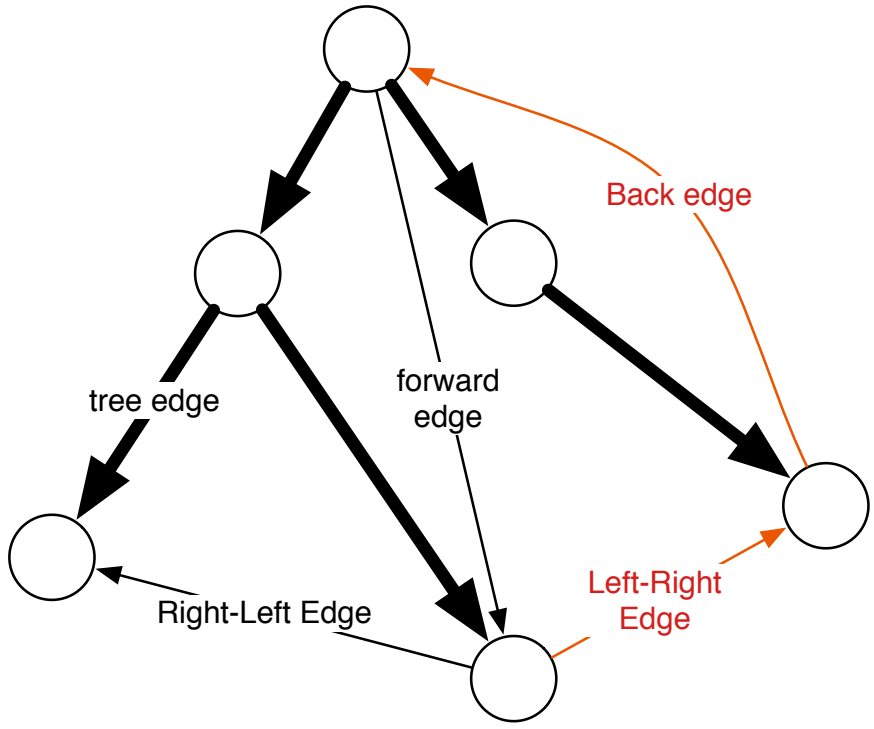


Non-DFS-Trees Edges of a DAG

Let (u, v) be an edge of a DAG D. What can we say about the relationship between $f[u]$ and $f[v]$?

Non-DFS-Trees Edges of a DAG

Let (u, v) be an edge of a DAG D. What can we say about the relationship between $f[u]$ and $f[v]$?



Back edges and Left-Right edges cannot occur

$\Longrightarrow$ $f[v] < f[u]$ if $(u, v) \in D$.

Topological Sort Via Finishing Times

Another Topological Sort Algorithm:

Every edge (u, v) in a DAG has $f[v] < f[u]$.

If we list nodes from largest $f[u]$ to smallest $f[u]$ then every edge goes from left to right.

Exactly a topological sort.

So: as each node is finished, add it to the front of a linked list.