EE360C: Algorithms

Graphs
Part 3/3

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

In Class Exercise

Prove the following

We have a connected graph $G = (V, E)$ and a specific vertex $u \in V$. Suppose that we compute a depth first search tree rooted at u and obtain a tree T that includes all of the nodes of G. Suppose we then compute a breadth first search tree rooted at u and obtain the same tree T. Prove that $G = T$.

Proof

Suppose that G has an edge (a, b) that does not belong to T. Since T is a depth first search tree, one of the two ends must be an ancestor of the other. Let's say that a is an ancestor of b. Since T is also a breadth first search tree, the levels of the two nodes in T can differ by at most one. But if a is an ancestor of b and the distance from the root to b in T is at most one greater than the distance from the root to a, then a must be b's parent in T and $(a, b) \in T$. This is a contradiction.

Testing Bipartiteness

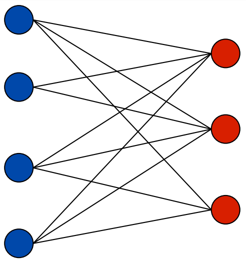
Bipartite Graphs

Definition

An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications

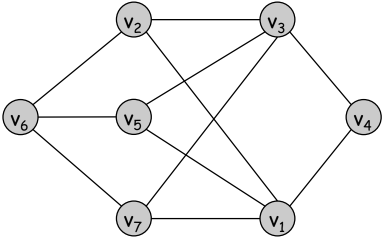
- Stable marriage: men = red, women = blue
- Scheduling: machines = red, jobs = blue

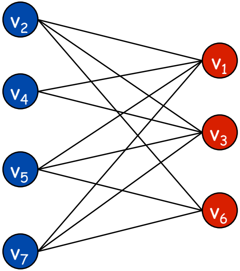


Testing Bipartiteness

Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand the structure of bipartite graphs





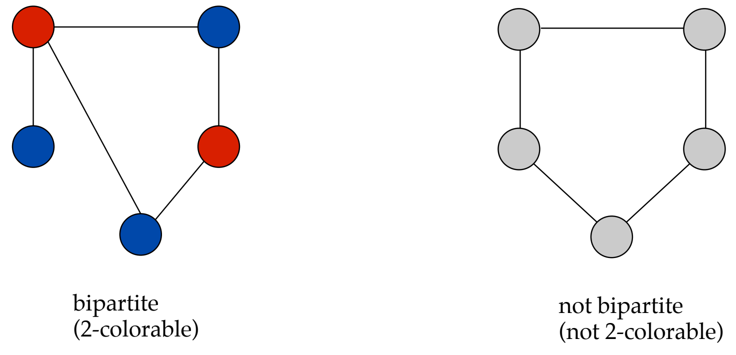
Proofs About Bipartiteness

Lemma

If a graph G is bipartite, it cannot contain an odd length cycle.

Proof Sketch

It is not possible to "2-color" the odd cycle (let alone the entire graph G)

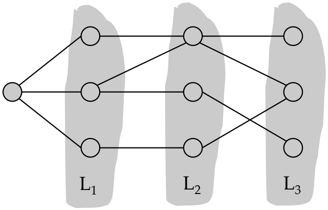


Bipartite Graphs

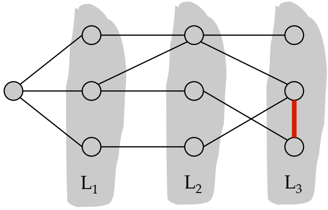
Lemma

Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

1. No edge of G joins two nodes of the same layer, and G is bipartite.
2. An edge of G joins two nodes in the same layer, and G contains an odd length cycle (and hence is not bipartite).



Case 1



Case 2

Bipartite Graphs

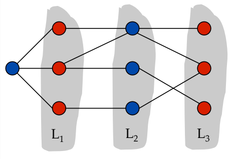
Lemma

Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

1. No edge of G joins two nodes of the same layer, and G is bipartite.

Proof (Case 1)

- Suppose no edge joins two nodes in the same layer.
- By the previous lemma (in BFS if (x, y) is an edge in G then the layer difference is at most 1.) this implies that all edges join nodes on adjacent levels.
- Then the bipartition is such that nodes on odd levels are red; nodes on even levels are blue.



Case 1

Bipartite Graphs

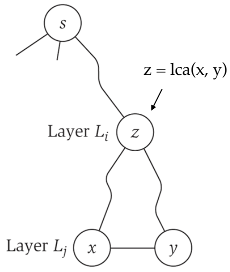
Lemma

Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

1. An edge of G joins two nodes in the same layer, and G contains an odd length cycle (and hence is not bipartite).

Proof (Case 2)

- Suppose (x, y) is an edge with x and y in the same level L_j.
- Let z be the lowest common ancestor of x and y. Let L_i be the level containing z.
- Consider the cycle that takes the edge from x to y, then the path from y to z, then the path from z to x.
- It's length is $1 + (j - i) + (j - i) = 2(j - i) + 1$, which is odd.

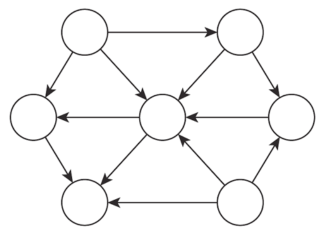


Connectivity in Directed Graphs

Directed Graphs

Directed Graph

In a directed graph, $G = (V, E)$, an edge (u, v) goes from node u to node v.



Example

In a web-graph, hyperlinks point *from* a web page *to* another.

- Directedness of the graph is crucial.
- Modern web search engines exploit the hyperlink structure to rank web pages by importance.

Directed Reachability

Given a node s, find all nodes reachable from s.

Directed $s - t$ Shortest Path Problem

Given two nodes s and t, what is the length of the shortest path between s and t?

Graph Search

- Breadth first search (and depth first search) extend naturally to directed graphs.
- Set of nodes t such that there is a path from s to t.
- How do I find a set of nodes that have a path to s?

Web Crawler

Start from web page s. Find all web pages linked from s, either directly or indirectly.

Strong Connectivity

Definition

Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Definition

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.

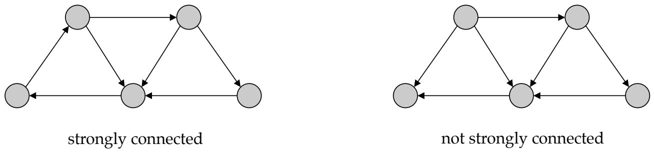
Determining Strong Connectivity

Theorem

We can determine if G is strongly connected in $O(m + n)$ time.

Algorithm

Idea: We have to show every node that is reachable from s, and every node is reachable to s.



Determining Strong Connectivity

Theorem

We can determine if G is strongly connected in $O(m+n)$ time.

Algorithm

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

Strong Components

Definition

The strong component containing a node s in a directed graph is the set of all v such that s and v are mutually reachable.

The previous algorithm is really computing the strong component containing s.

Theorem

For any two nodes s and t in a directed graph, their strong components are either identical or disjoint.

DAGs and Topological Ordering

Directed Acyclic Graphs

Definition

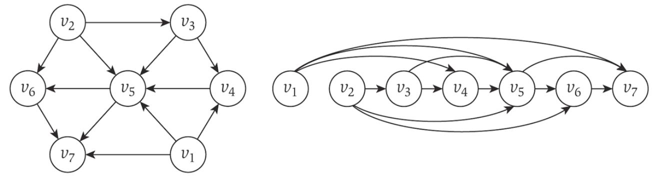
A DAG is a directed graph that contains no directed cycles.

Example

Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.

Definition

A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



Precedence Constraints

Precedence Constraints

Edge (v_i, v_j) means task v_i must occur before v_j.

Applications

- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j.
- Pipeline of computing jobs: output of job v_i needed to determine input of job v_j

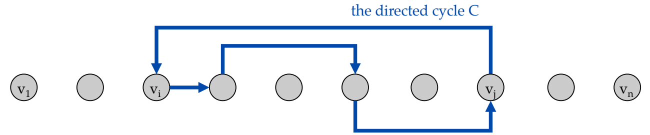
DAGs and Topological Sort

Lemma

If G has a topological order, then G is a DAG.

Proof (by contradiction)

- Suppose that G has a topological order $v_1, \dots v_n$ and that G also has a directed cycle.
- Let v_i be the lowest-indexed node in the cycle and let v_j be the node just before v_i. Thus (v_j, v_i) is an edge in E.
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, \dots v_n$ is a topological order, we must have $j < i$, a contradiction.



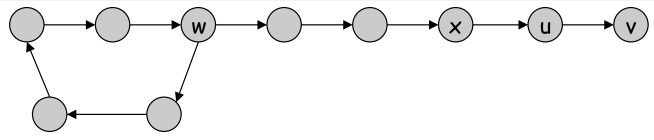
DAGs and Topological Sort (cont.)

Lemma

If G is a DAG, then G has a node with no incoming edges

Proof (by contradiction)

- Suppose G is a DAG and every node has at least one incoming edge.
- Pick any node v and begin following edges backward from v. Since v has at least one incoming edge (u, v), we can walk backward to u.
- Then since u has at least one incoming edge (x, u), we can walk backward to x.
- Repeat until we visit a node, say w, twice.
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle.



Computing a Topological Ordering

Lemma

If G is a DAG, then G has a topological ordering.

Proof (by induction)

- Base case: true if $n = 1$.
- Given a DAG on $n > 1$ nodes, find a node v with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place v first in the topological ordering, then append the nodes of $G - \{v\}$ in topological order. This is valid since v has no incoming edges.

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

Topological Sort Analysis

Theorem

The algorithm finds a topological order in $O(m+n)$ time.

Proof

- Maintain the following information:
  - `count[w]`: the remaining number of incoming edges
  - S: the set of remaining nodes with no incoming edges
- Initialization: $O(m+n)$ via a single scan through the graph
- Update: to delete v:
  - remove v from S
  - decrement `count[w]` for all edges v to w, and add w to S if `count[w]` hits 0
  - This is constant time per edge, and we go over each edge once

In Class Exercise

Provide an alternative algorithm to computing a topological sort that uses the DFS procedure directly.

```
DFS(u)
mark u as explored and add to R
for each (u, v) incident to u
    do if v is not explored
        DFS(v)
```

Questions