Topics
1. BFS Claims. 2. Testing Bipartiteness: An application of BFS. 3. Directed Acyclic graphs and Topological ordering.

BFS Claims

1. For each $j \geq 1$, layer L_j produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if t appears in some layer.

2. Let T be a breadth-first search tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G. Then i and j differ by at most 1.

   Proof. Suppose by way of contradiction that i and j differed by more than 1; in particular, suppose $i < j - 1$. Now consider the point in the BFS algorithm when the edges incident to x were being examined. Since x belongs to layer Li, the only nodes discovered from x belong to layers L_i+1 and earlier; hence, if y is a neighbor of x, then it should have been discovered by this point at the latest and hence should belong to layer L_i+1 or earlier.

3. The implementation of the BFS algorithm runs in time O(m + n) (i.e., linear in the input size), if the graph is given by the adjacency list representation.

   Proof:

- We need to observe that the For loop processing a node u can take less than O(n) time if u has only a few neighbors.
- As before, let nu denote the degree of node u, the number of edges incident to u. Now, the time spent in the For loop considering edges incident to node u is $O(n_u)$, so the total over all nodes is $O(\Sigma_{u \in V} n_u)$. Recall that $\Sigma_{u \in V} n_u = 2m$, and so the total time spent considering edges over the whole algorithm is O(m).
- We need O(n) additional time to set up lists and manage the array Discovered.
- So the total time spent is O(m + n) as claimed.

Testing Bipartiteness: An application of BFS

The Problem

Clearly a triangle is not bipartite, since we can color one node red, another one blue, and then we can't do anything with the third node.
More generally, if a graph G simply contains an odd cycle, then we can apply an argument; thus we have established the following.

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

Designing the Algorithm

1. We can implement this on top of BFS, by simply taking the implementation of BFS and adding an extra array Color over the nodes.
2. Whenever we get to a step in BFS where we are adding a node v to a list L[i + 1], we assign Color[v]= red if i + 1 is an even number, and Color[v]= blue if i + 1 is an odd number.
3. At the end of this procedure, we simply scan all the edges and determine whether there is any edge for which both ends received the same color.
4. Thus, the total running time for the coloring algorithm is O(m + n), just as it is for BFS.

Analyzing the Algorithm

1. *Let G be a connected graph, and let L1, L2, . . . be the layers produced by BFS starting at node s. Then exactly one of the following two things must hold.*

   (i) There is no edge of G joining two nodes of the same layer. In this case G is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue.

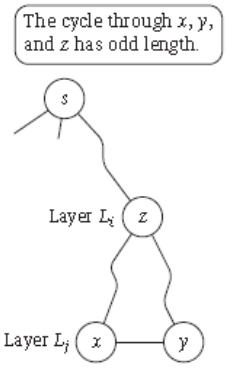
   (ii) There is an edge of G joining two nodes of the same layer. In this case, G contains an odd-length cycle, and so it cannot be bipartite.

Proof. First consider case (i), where we suppose that there is no edge joining two nodes of the same layer. We know that every edge of G joins nodes either in the same layer or in adjacent layers.

- Our assumption for case (i) is precisely that the first of these two alternatives never happens, so this means that every edge joins two nodes in adjacent layers.
- But our coloring procedure gives nodes in adjacent layers the opposite colors, and so every edge has ends with opposite colors. Thus this coloring establishes that G is bipartite.

Now suppose we are in case (ii); why must G contain an odd cycle?

- *We are told that G contains an edge joining two nodes of the same layer.*
- *Suppose this is the edge e = (x, y), with x, y ∈ Lj. Also, for notational reasons, recall that L0 ("layer 0") is the set consisting of just s.*
- *Now consider the BFS tree T produced by our algorithm, and let z be the node whose layer number is as large as possible, subject to the condition that z is an ancestor of both x and y in T;*
- *Suppose z ∈ Li, where i < j. We now have the situation pictured in Figure.*
- *We consider the cycle C defined by following the z-x path in T, then the edge e and then the y-z path in T. The length of this cycle is (j − i) + 1+ (j − i), adding the length of its three parts separately; this is equal to 2(j − i) + 1, which is an odd number.*



Directed Acyclic graphs and Topological ordering

If a directed graph has no cycles, we call it—naturally enough—a directed acyclic graph, or a DAG for short.

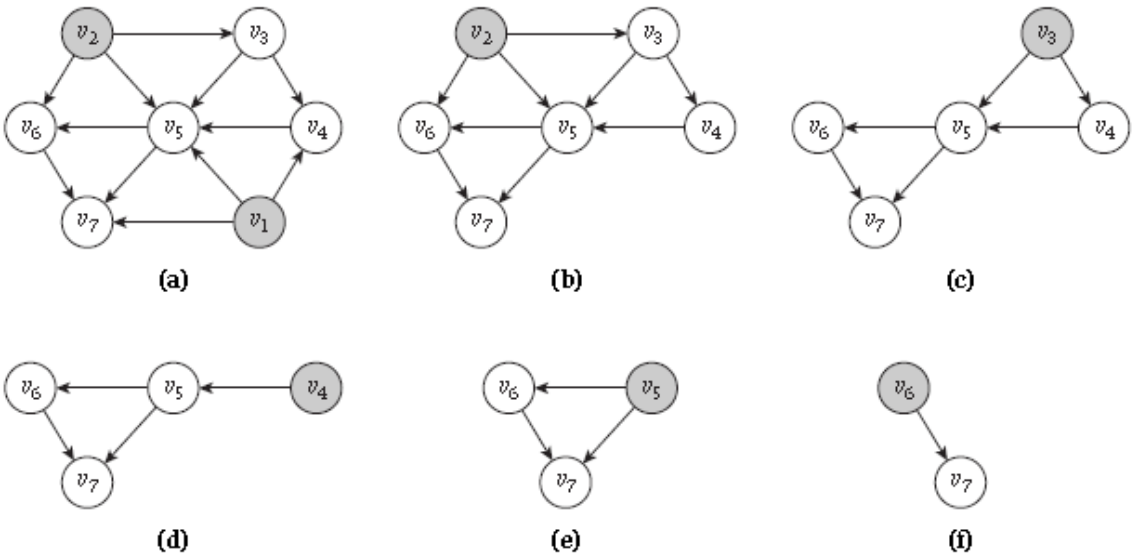
The Problem

- DAGs can be used to encode precedence relations or dependencies in a natural way. Suppose we have a set of tasks labeled {1, 2, . . . , n} that need to be performed, and there are dependencies among them stipulating, for certain pairs i and j, that i must be performed before j.
- For example, the tasks may be courses, with prerequisite requirements stating that certain courses must be taken before others.
- We can represent such an interdependent set of tasks by introducing a node for each task, and a directed edge (i, j) whenever i must be done before j. If the precedence relation is to be at all meaningful, the resulting graph G must be a DAG.

Designing and Analyzing the Algorithm

```
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
```

Example



Analysis

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

Proof. Suppose, by way of contradiction, that G has a topological ordering v1, v2, . . . , vn, and also has a cycle C. Let vi be the lowest-indexed node on C, and let vj be the node on C just before vi—thus (vj , vi) is an edge. But by our choice of i, we have j > i, which contradicts the assumption that v1, v2, . . . , vn was a topological ordering.

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

Proof:

- Since G is a DAG, there is a node v with no incoming edges.
- We place v first in the topological ordering; this is safe, since all edges out of v will point forward.
- Now G–{v} is a DAG, since deleting v cannot create any cycles that weren't there previously. Also, G–{v} has n nodes, so we can apply the induction hypothesis to obtain a topological ordering of G–{v}.
- We append the nodes of G–{v} in this order after v; this is an ordering of G in which all edges point forward, and hence it is a topological ordering.