COMPSCI 311 Introduction to Algorithms

Lecture 5: Graphs: Bipartite, Directed, Topological Sorting

Marius Minea

University of Massachusetts Amherst

slides credit: Dan Sheldon, Akshay Krishnamurthy, Andrew McGregor

6 February 2019

Review and Outlook

- Graph traversal by BFS/DFS
  - Different versions of general exploration strategy
  - $O(m + n)$ time
  - Produce trees with useful properties (for other problems)
  - Basic algorithmic primitive — used in many other algorithms path from s to t, connected components
- Bipartite testing
- Directed graphs
  - Traversal
  - Strong connectivity
  - Topological sorting

Bipartite Graphs

Definition Graph $G = (V, E)$ is bipartite if V can be partitioned into sets X, Y such that every edge has one end in X and one in Y.

Can color nodes red/blue s.t. no edges between nodes of same color.

Examples

- Bipartite: student-college graph in stable matching
- Bipartite: client-server connections
- Not bipartite: "odd cycle" (cycle with odd # of nodes)
- Not bipartite: any graph containing odd cycle

Claim (easy): If G contains an odd cycle, it is not bipartite.

Bipartite Testing

Question Given $G = (V, E)$, is G bipartite?

Algorithm? **Idea**: run BFS from any node s

- L_0 = red
- L_1 = blue
- L_2 = red
- ...
- Even layers red, odd layers blue

What could go wrong? Edge between two nodes at same layer.

Bipartite Testing: Algorithm

Run BFS from any node s
if there is an edge between two nodes in same layer **then**
    Output "not bipartite"
else
    X = even layers
    Y = odd layers
end if

Correctness? Recall: all edges between same or adjacent layers.

1. If there are no edges between nodes in the same layer, then G is bipartite.
2. If there is an edge between two nodes in the same layer, G has an odd cycle and is not bipartite. Proof?

Bipartite Testing: Proof

- Let T be BFS tree of G and suppose (x, y) is an edge between two nodes in the layer j
- Let $z \in L_i$ be the least common ancestor of x and y (Useful in proofs: take least/greatest item with some property)
  - P_{zx} = path from z to x in T
  - P_{yz} = path from z to y in T
  - Path that follows P_{zx} then edge (x, y) then P_{yz} is a cycle of length $2(j - i) + 1$, which is odd
- Therefore G is not bipartite.

Directed Graphs

$G = (V, E)$

- $(u, v) \in E$ is a *directed* edge
- u points to v

Examples

- Facebook: undirected
- Twitter: directed
- Web: directed
- Road network: directed (if one-way roads)

Directed Graph Definitions

Most definitions extend naturally to directed graphs by mapping the word "edge" to "directed edge"

- **Directed path**: sequence $P = v_1, v_2, \ldots, v_{k-1}, v_k$ such that each consecutive pair v_i, v_{i+1} is joined by a *directed edge* in G. A $v_1 \to v_k$ path.
- **Directed cycle**: directed path with $v_1 = v_k$
- **Connected**? **Connected component**? More subtle, because now there can be a path from s to t but not vice versa.

Directed Graph Traversal

Reachability. Find all nodes reachable from some node s.

s-t shortest path.
What is the length of the shortest *directed* path from s to t?

Algorithm? BFS naturally extends to directed graphs.
Add v to L_{i+1} if there is a *directed* edge from L_i and v is not already discovered.

BFS in Directed Graph

BFS/DFS naturally extend to directed graphs.

```
BFS(s):
  mark s as "discovered"
  L[0] ← {s}, i ← 0
  while L[i] is not empty do
      L[i+1] ← empty list
      for all nodes v in L[i] do
          for all edges (v,w) leaving v do
              if w is not marked "discovered" then
                  mark w as "discovered"
                  put w in L[i+1]
              end if
          end for
      end for
      i ← i+1
  end while
```

Variations of Traversal

Traversal *from* s finds nodes t with path $s \rightsquigarrow t$
There may be no path $t \rightsquigarrow s$

Find all nodes v from which we can reach t? ($v \to t$ path)?
BFS following edges in reverse direction

Useful to keep adjacency lists for both outgoing and incoming edges.

Clicker Question 1

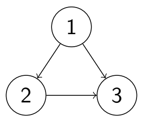
Suppose G is a directed path on n vertices and we call BFS repeatedly starting from any unexplored vertex until all nodes are explored. What is the maximum number of times BFS may be called?

A. 1
B. $n-1$
C. n
D. m

Differences in Traversing Directed Graphs

Recall: Tree = undirected, connected, acyclic graph
$\Rightarrow$ finding a non-tree edge (in BFS or DFS) = cycle
non-tree edge: reaching an already discovered node
(*except* for node's parent)

No longer true in directed graph:



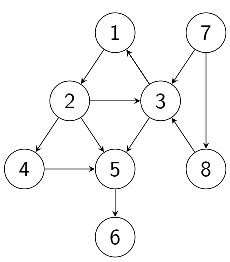
Edges in Directed Graph BFS

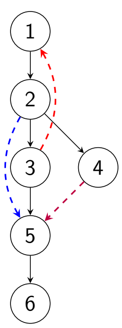
Paths are directed, no need to check for parent.

With respect to BFS tree, graph edges can go

- one level down (tree or non-tree edge)
  why not > 1 ? same reason, would add to next level
- same level (non-tree)
- any levels up (non-tree)

DFS in Directed Graphs





$3 \to 1$ is a back edge (to ancestor)
$2 \to 5$ is a forward edge (to descendant)
$4 \to 5$ is a cross edge (node in another subtree)

Clicker Question 2

Which of these types of edges must close a cycle ?

A: back edges

B: forward edges

C: cross edges

D: all of the above

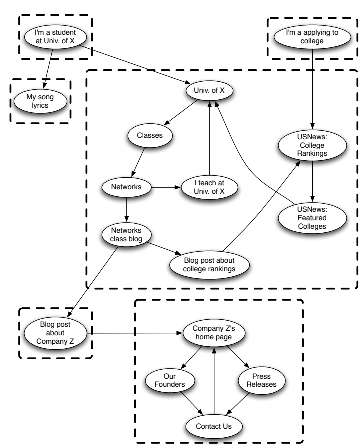
Identifying Edges in DFS

To detect the various edges, we track:

- start ("discovered") / end ("explored") of neighbor iteration
- order in which nodes are reached (running counter)

count = 0
DFS(u)
  $num[u]$ = ++count
  mark u as "discovered"
  for all edges (u, v) **do**
    if v is "unseen" **then**
      call DFS(v) recursively ▷ tree edge
    else if v is "discovered" **then** ▷ back edge
    else ▷ v is "explored"
      if $num[v] > num[u]$ **then** ▷ forward edge
      else ▷ $num[v] < num[u]$: cross edge
      end if
    end if
  end for
  mark u as "explored"

Directed Graph Connectivity



Strongly connected graph.
Directed path between any two nodes.

Strongly connected component (SCC).
Maximal subset of nodes with directed path between any two.

SCCs can be found in time $O(m + n)$. (Tarjan, 1972)

Clicker Question 3

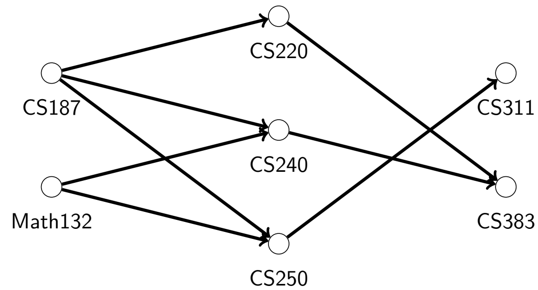
Consider the graph G' whose nodes are SCCs and there is an edge from C to D if any node in C has an edge to D.
Which of the following is always true?

A. G' is strongly connected
B. G' has a cycle
C. G' has at least $n/2$ nodes
D. G' is a DAG

Directed Acyclic Graphs

Definition
A directed acyclic graph (DAG) is a directed graph with no cycles.

Models *dependencies*, e.g. course prerequisites:



Math: (strict) partial order (irreflexive, antisymmetric, transitive)

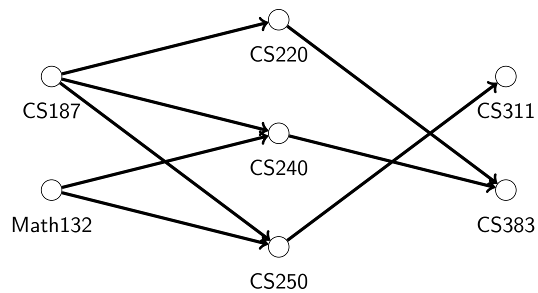
Topological Sorting

Definition A topological ordering of a directed graph is an ordering of the nodes such that all edges go "forward" in the ordering

- Label nodes $v_1, v_2, \ldots, v_n$ such that
- For all edges (v_i, v_j) we have $i < j$
- A way to order the classes so all prerequisites are satisfied

Q: Is a topological ordering possible for any graph?

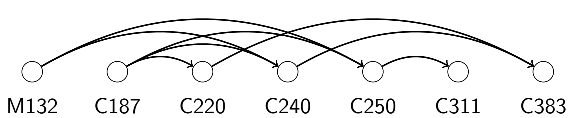
Topological Sorting



Exercise

1. Find a topological ordering.
2. Devise an algorithm to find a topological ordering.

Topological Ordering



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

Topological Sorting

Problem Given DAG G, compute a topological ordering for G.

topo-sort(G)
  while there are nodes remaining **do**
    Find a node v with no incoming edges
    Place v next in the order
    Delete v and all of its outgoing edges from G
  end while

Running time? $O(n^2 + m)$ easy. $O(m + n)$ more clever

Topological Sorting Analysis

- In a DAG, there is always a node v with no incoming edges. Try to prove. (contradiction, pigeonhole principle)
- Removing a node v from a DAG, produces a new DAG.
- Any node with no incoming edges can be first in topological ordering.

Theorem: G is a DAG if and only if G has a topological ordering.

Topological Sorting in $O(m + n)$

topo-sort(G)
  while there are nodes remaining **do**
    Find a node v with no incoming edges
    Place v next in the order
    Delete v and all of its outgoing edges from G
  end while

Optimization: don't search every time for nodes w/o incoming edges

- Keep and update incoming edge count for each node (setup in $O(m + n)$, each update constant-time)
- Keep set of nodes of nodes with incoming edges; add node when its count becomes zero
- Running time: $O(m + n)$