Lecture 12: Graphs II: Depth-First Search

Lecture Overview

- Depth-First Search
- Edge Classification
- Cycle Testing
- Topological Sort

Recall:

- graph search: explore a graph
  e.g., find a path from start vertex s to a desired vertex
- adjacency lists: array Adj of $|V|$ linked lists
  - for each vertex $u \in V$, Adj$[u]$ stores u's neighbors, i.e., $\{v \in V \mid (u, v) \in E\}$
    (just outgoing edges if directed)

For example:

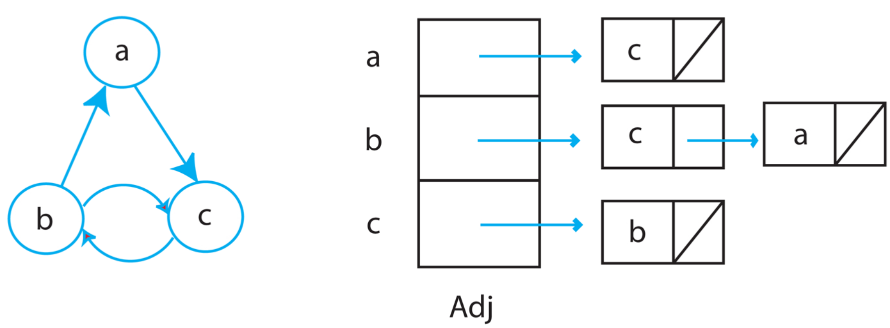


Figure 1: Adjacency Lists

Breadth-first Search (BFS):

Explore level-by-level from s — find shortest paths

Depth-First Search (DFS)

This is like exploring a maze (Charles Pierre Tremaux, Cretan labyrinth). You can use this for finding paths, detecting cycles, checking whether a graph is bipartite (color edges opposite of parent's, then make sure that edges always connect opposite colors).

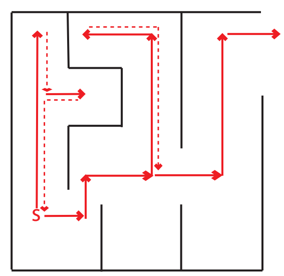


Figure 2: Depth-First Search Frontier

Depth First Search Algorithm

- follow path until you get stuck
- backtrack along breadcrumbs until reach unexplored neighbor
- recursively explore, being careful not to repeat a vertex
- another way to think about it: touch a wall and keep walking!

```
        DFS-visit (Adj, s):
start v →   for v in Adj [s]:
                if v not in parent:
                    parent [v] = s
finish v →          DFS-visit (Adj, v)

        DFS (V, Adj)
            parent = { }
            for s in V:
                if s not in parent:
                    parent [s] = None
                    DFS-visit (Adj, s)
```

search from start vertex s (only see stuff reachable from s)

explore entire graph

(could do same to extend BFS)

Figure 3: Depth-First Search Algorithm

Example

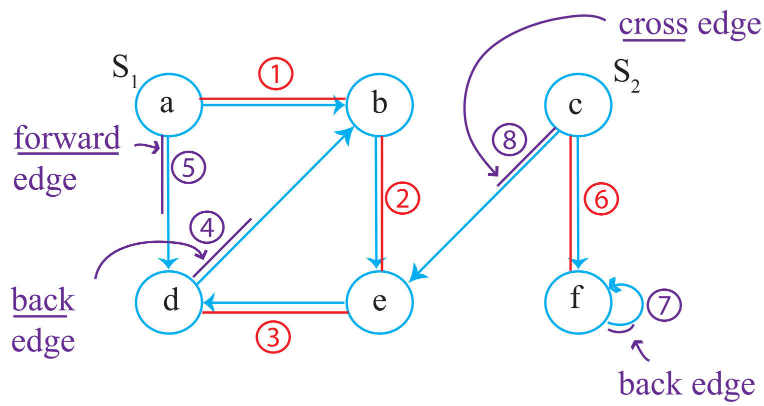


Figure 4: Depth-First Traversal

Edge Classification

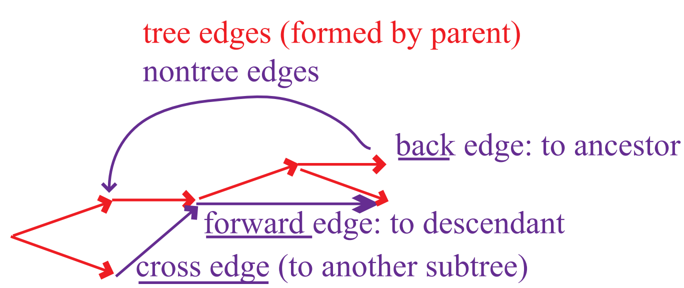


Figure 5: Edge Classification

- to compute this classification (back or not), mark nodes for duration they are "on the stack"
- only tree and back edges in undirected graph

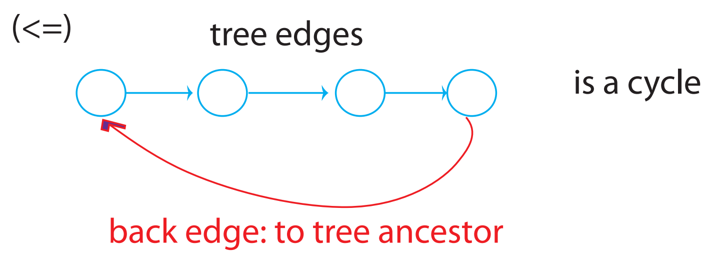
Analysis

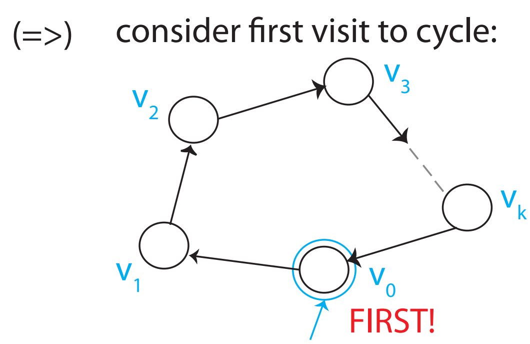
- DFS-visit gets called with a vertex s only once (because then parent$[s]$ has been set)
  $\Longrightarrow$ time in DFS-visit $= \displaystyle\sum_{s \in V} |\text{Adj}[s]| = O(E)$
- DFS outer loop adds just $O(V)$
  $\Longrightarrow O(V + E)$ time (linear time)

Cycle Detection

Graph G has a cycle $\Leftrightarrow$ DFS has a back edge

Proof





- before visit to v_i finishes,
  will visit v_{i+1} (& finish):
  will consider edge (v_i, v_{i+1})
  $\implies$ visit v_{i+1} now or already did
- $\implies$ before visit to v_0 finishes,
  will visit v_k (& didn't before)
- $\implies$ before visit to v_k (or v_0) finishes,
  will see (v_k, v_0) as back edge

Job scheduling

Given Directed Acylic Graph (DAG), where vertices represent tasks & edges represent dependencies, order tasks without violating dependencies

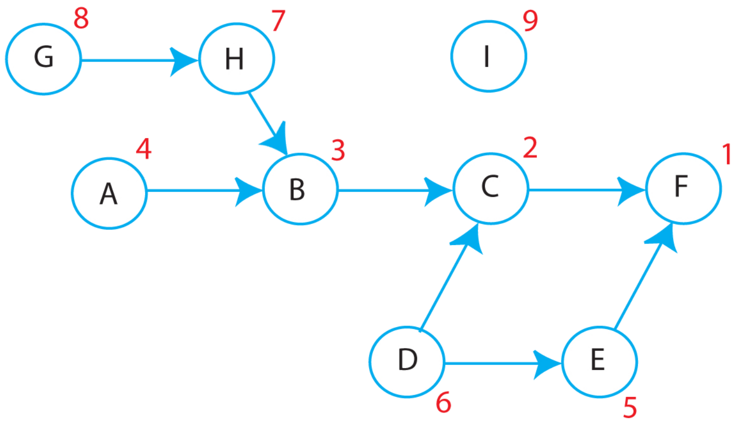


Figure 6: Dependence Graph: DFS Finishing Times

Source:

$$\text{Source} = \text{vertex with no incoming edges}$$
$$= \text{schedulable at beginning (A,G,D,I)}$$

Attempt:

BFS from each source:

- from A finds A, B, C, F (you're already doing C before you do D, if you choose to just take this ordering)
- from D finds D, CE, F
- from G finds G, H (then B, C, F, but depending upon what order this source is BFS-ed vs vertex A, this could be wrong)
- from I finds I

Topological Sort

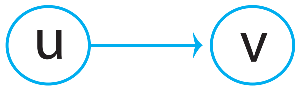
Reverse of DFS finishing times (time at which DFS-Visit(v) finishes)

DFS-Visit(v)
...
order.append(v)

order.reverse()

Correctness

For any edge (u, v) — u ordered before v, i.e., v finished before u



- if u visited before v:
  - before visit to u finishes, will visit v (via (u, v) or otherwise)
  - $\implies$ v finishes before u
- if v visited before u:
  - graph is acyclic
  - $\implies$ u cannot be reached from v
  - $\implies$ visit to v finishes before visiting u