CS61B Lecture #35

Recursive Depth-First Traversal of a Graph

- Can fix looping and combinatorial problems using the "bread-crumb" method used in earlier lectures for a maze.
- That is, *mark* nodes as we traverse them and don't traverse previously marked nodes.
- Makes sense to talk about *preorder* and *postorder*, as for trees.

```
void preorderTraverse(Graph G, Node v) {
   if (v is unmarked) {
     mark (v);
     visit v;
     for (Edge (v, w) ∈ G)
       traverse(G, w);
   }
}
```

```
void postorderTraverse(Graph G, Node v) {
   if (v is unmarked) {
     mark (v);
     for (Edge (v, w) ∈ G)
       traverse(G, w);
     visit v;
   }
}
```

Recursive Depth-First Traversal of a Graph (II)

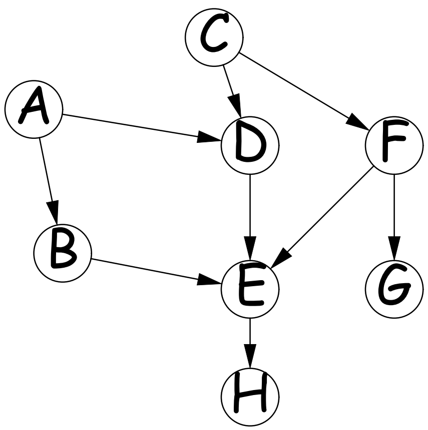
- We are often interested in traversing *all* nodes of a graph, not just those reachable from one node.
- So we can repeat the procedure as long as there are unmarked nodes.

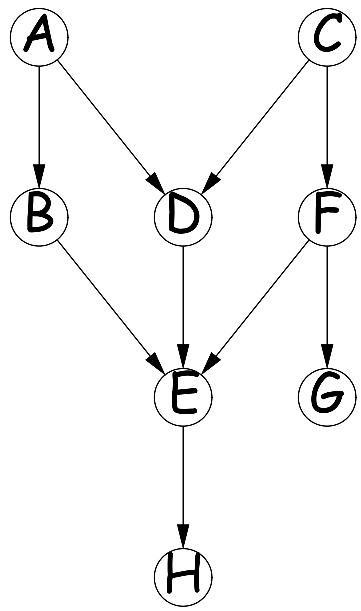
```
void preorderTraverse(Graph G) {
   for (v ∈ nodes of G) {
      preorderTraverse(G, v);
}

void postorderTraverse(Graph G) {
   for (v ∈ nodes of G) {
      postorderTraverse(G, v);
}
```

Topological Sorting

Problem: Given a DAG, find a linear order of nodes consistent with the edges.

- That is, order the nodes $v_0,\ v_1,\ \ldots$ such that v_k is never reachable from $v_{k'}$ if $k' > k$.
- Gmake does this. Also PERT charts.

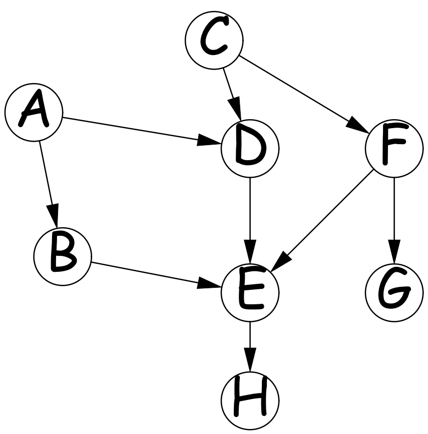


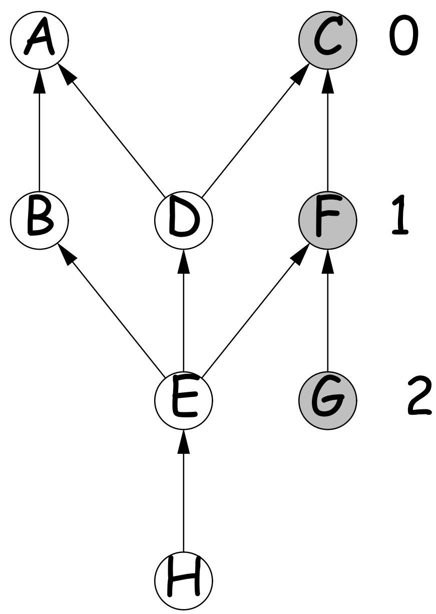


A	C	C
C	A	F
B	F	G
D	D	A
F	B	B
E	G	D
G	E	E
H	H	H

Sorting and Depth First Search

- **Observation**: Suppose we *reverse the links* on our graph.
- If we do a recursive DFS on the reverse graph, starting from node H, for example, we will find all nodes that must come *before* H.
- When the search reaches a node in the reversed graph and there are no successors, we know that it is safe to put that node first.
- In general, a *postorder* traversal of the reversed graph visits nodes only after all predecessors have been visited.





Numbers show postorder traversal order starting from G: everything that must come before G.

General Graph Traversal Algorithm

```
COLLECTION_OF_VERTICES fringe;

fringe = INITIAL_COLLECTION;
while (! fringe.isEmpty()) {
  Vertex v = fringe.REMOVE_HIGHEST_PRIORITY_ITEM();

  if (! MARKED(v)) {
    MARK(v);
    VISIT(v);
    For each edge (v,w) {
      if (NEEDS_PROCESSING(w))
        Add w to fringe;
    }
  }
}
```

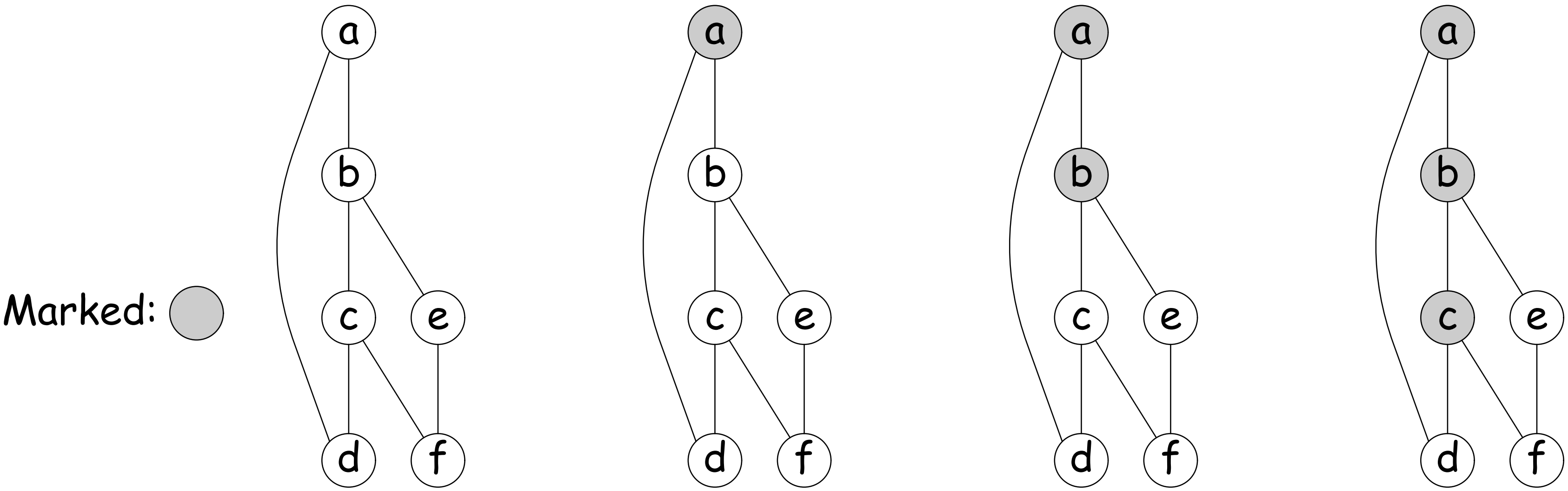
Replace *COLLECTION_OF_VERTICES*, *INITIAL_COLLECTION*, etc. with various types, expressions, or methods to different graph algorithms.

Example: Depth-First Traversal

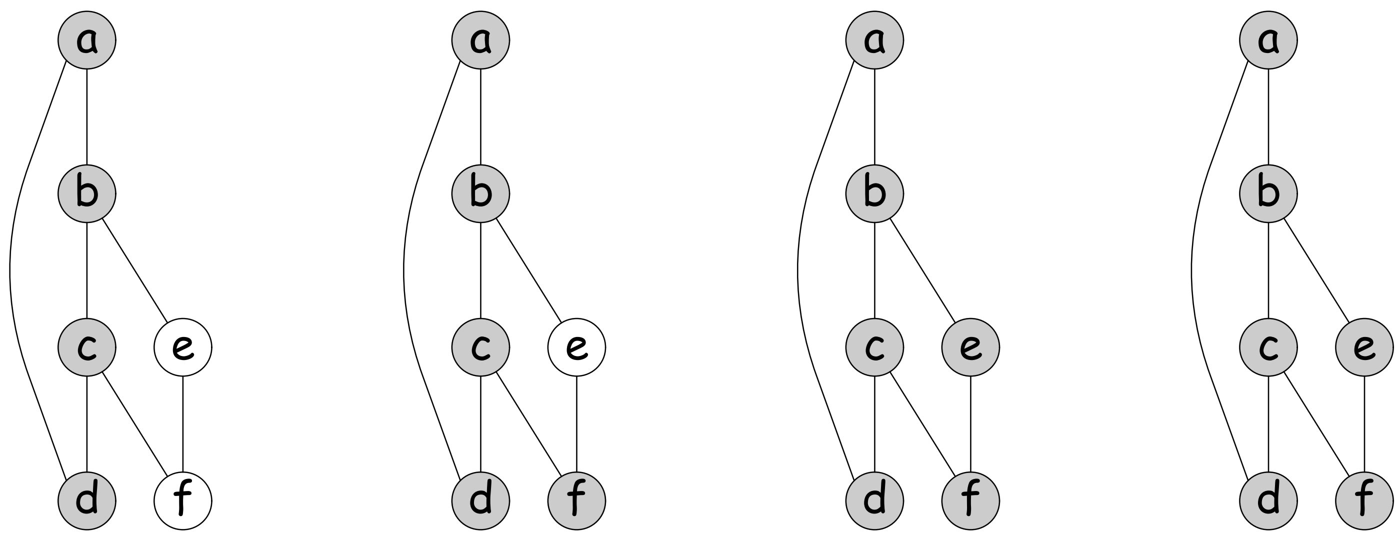
Problem: Visit every node reachable from v once, visiting nodes further from start first.

```
Stack<Vertex> fringe;

fringe = stack containing {v};
while (! fringe.isEmpty()) {
  Vertex v = fringe.pop ();

  if (! marked(v)) {
    mark(v);
    VISIT(v);
    For each edge (v,w) {
      if (! marked (w))
        fringe.push (w);
    }
  }
}
```

Depth-First Traversal Illustrated

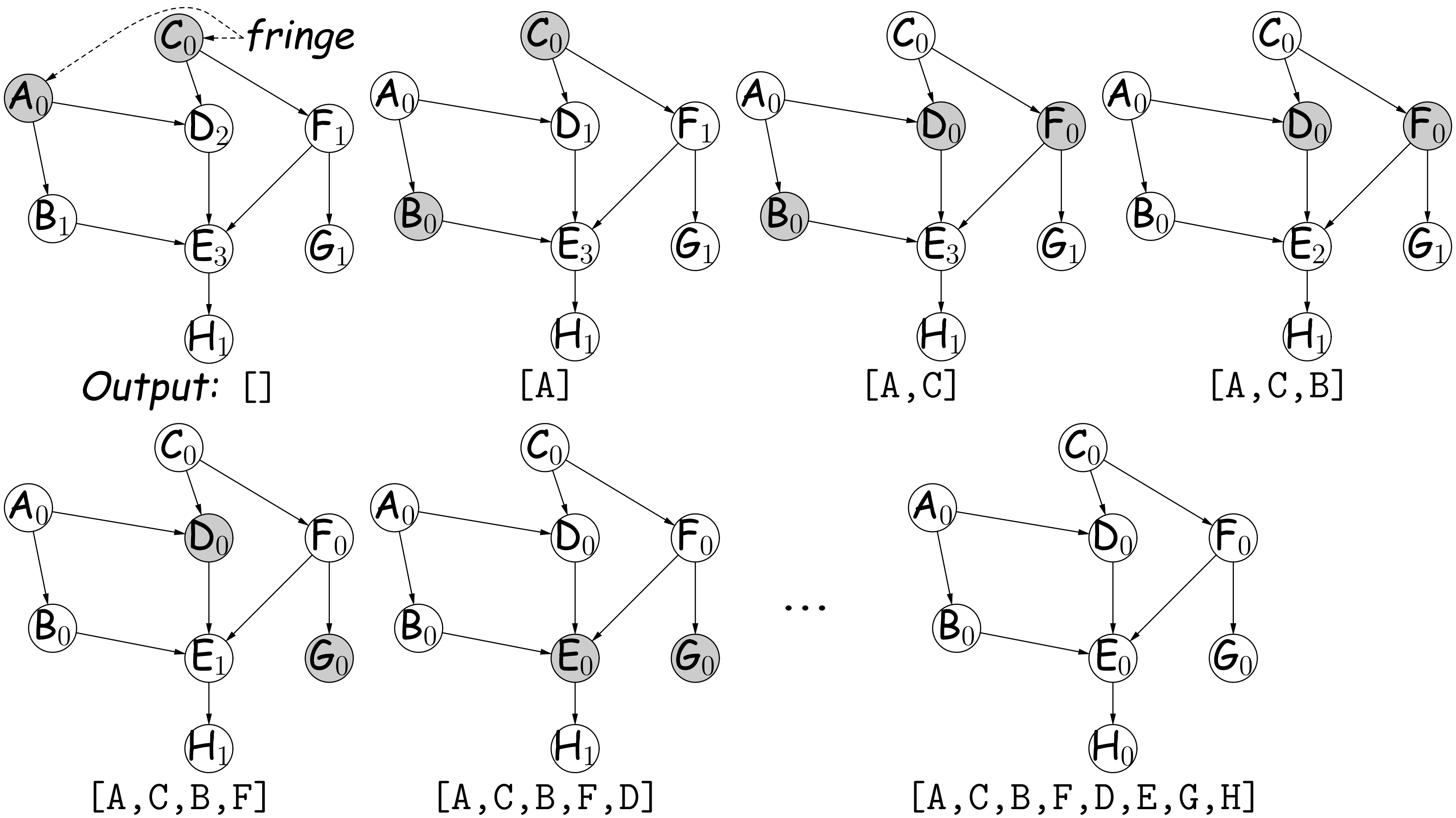


Fringe: [a] [b,d] [c,e,d] [d,f,e,d]



[f,e,d] [e,e,d] [e,d] []

Topological Sort in Action



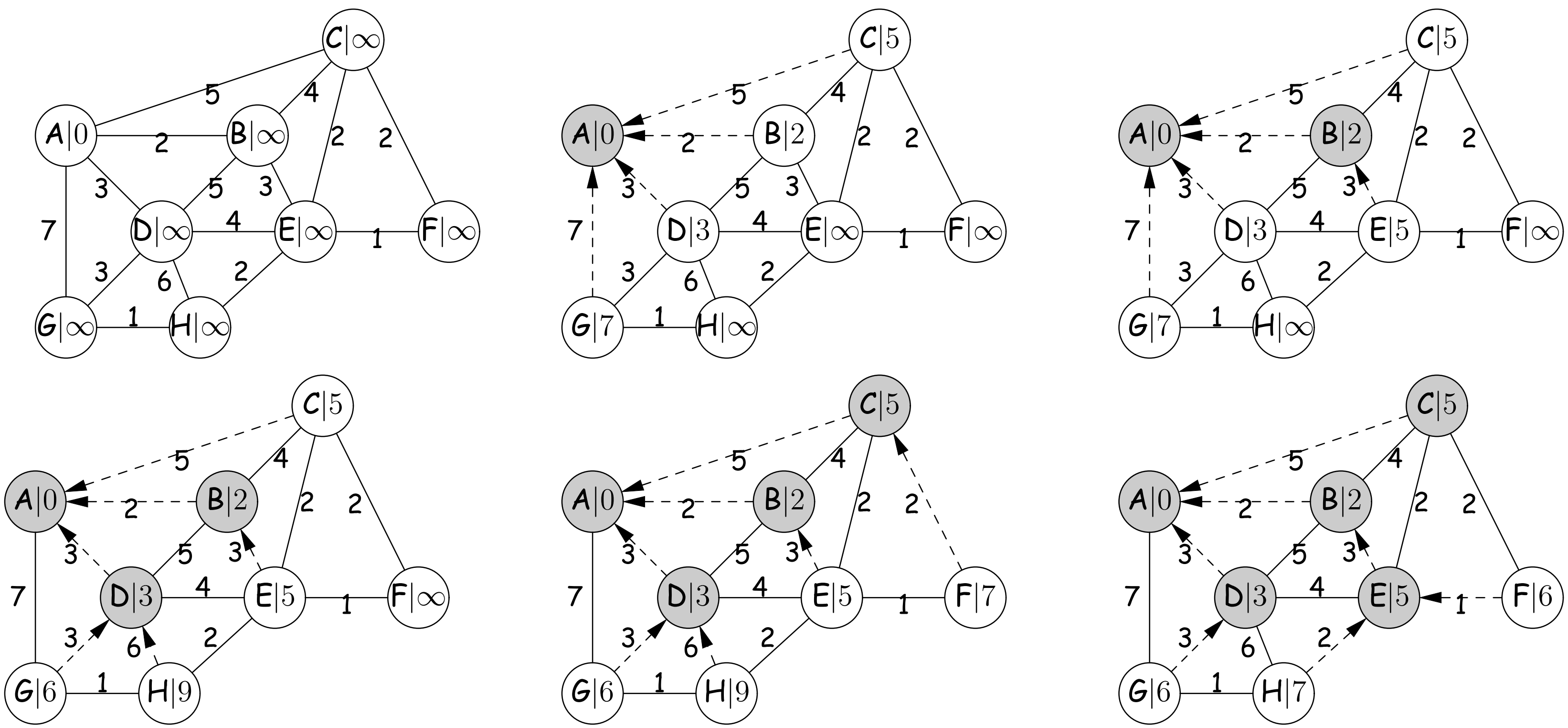
Shortest Paths: Dijkstra's Algorithm

Problem: Given a graph (directed or undirected) with non-negative edge weights, compute shortest paths from given source node, s, to all nodes.

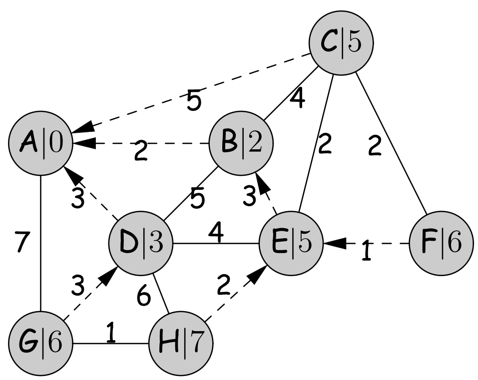
- "Shortest" = sum of weights along path is smallest.
- For each node, keep estimated distance from s, ...
- ...and of preceding node in shortest path from s.

```
PriorityQueue<Vertex> fringe;
For each node v { v.dist() = ∞; v.back() = null; }
s.dist() = 0;
fringe = priority queue ordered by smallest .dist();
add all vertices to fringe;
while (! fringe.isEmpty()) {
  Vertex v = fringe.removeFirst ();

  For each edge (v,w) {
    if (v.dist() + weight(v,w) < w.dist())
      { w.dist() = v.dist() + weight(v,w); w.back() = v; }
  }
}
```

Example



Final result:



- - -► Shortest-path tree

X|d processed node at distance d

Y|d node in fringe at distance d