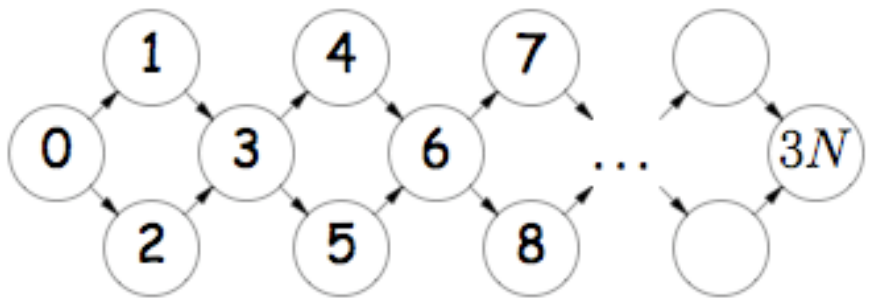
CS 61B Data Structures and Programming Methodology

Aug 5, 2008

David Sun

Graph Traversal

- Many algorithms on graphs depend on traversing all or some subset of the vertices.
- Unlike tree traversals, straight up *recursion* doesn't quite work because of the existence of *cycles* and *multiple paths* from the a vertex to another vertex.



- Treat 0 as the root and do recursive traversal down the two edges out of each vertex: $\Theta(2^N)$ operations!

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;
   }
  }
}
```

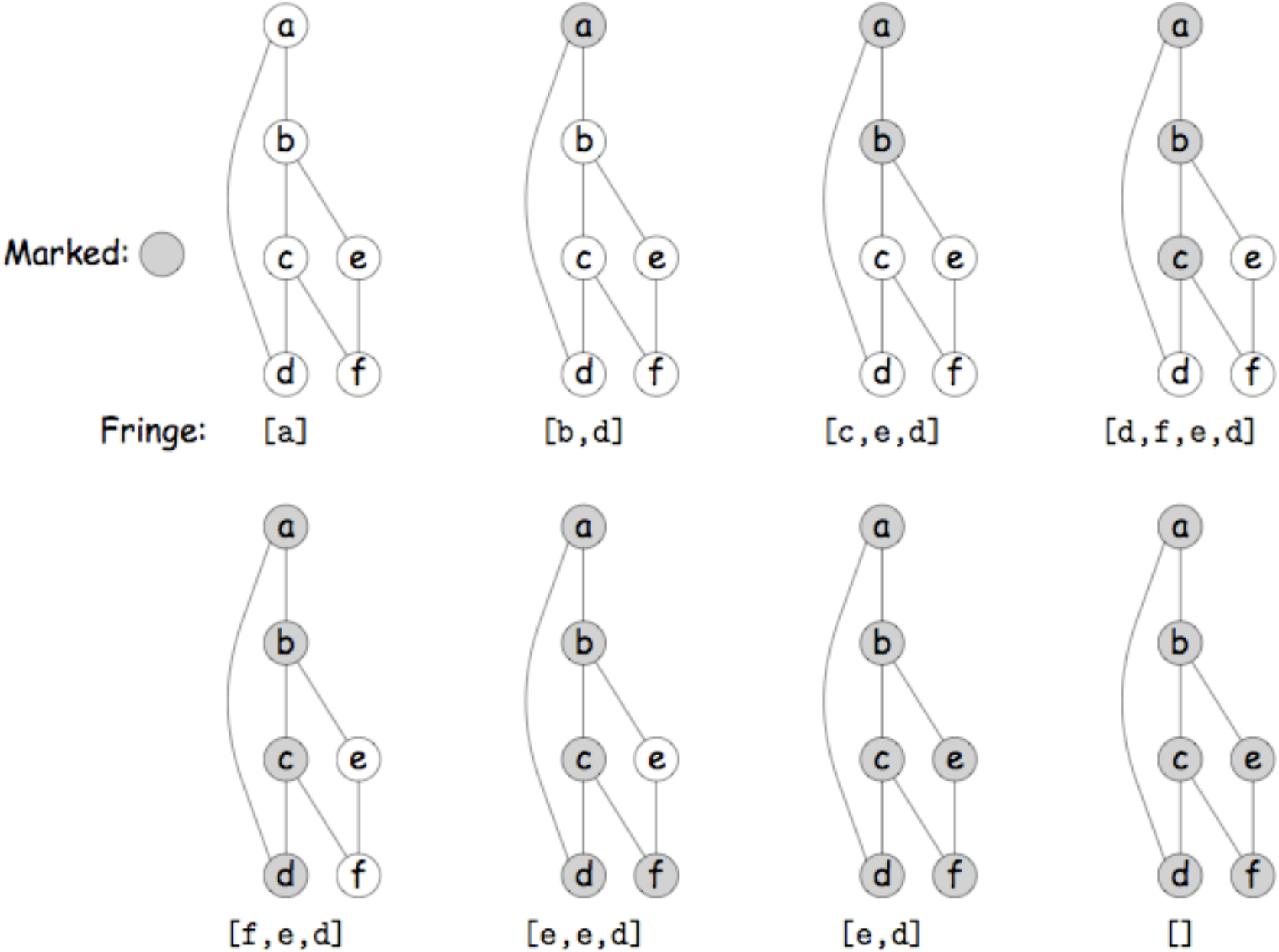
Replace COLLECITON OF VERTICES, INITIAL COLLECTION, etc. with various types, expressions, or methods to different graph algorithms.

Depth First Traversal

- General idea:
  - Starts at an arbitrary vertex and searches a graph as "deeply" as possible as early as possible.
  - When DFS visits a vertex *u*, it checks every vertex *v* that can be reached by some edge *(u, v)*. If *v* has not yet been visited, DFS visits it recursively.
- Avoid cycles:
  - Each vertex is marked when the vertex is first visited.

Iterative Depth First Traversal

```
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);
      }
   }
}
```


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

Recursive Depth First Traversal

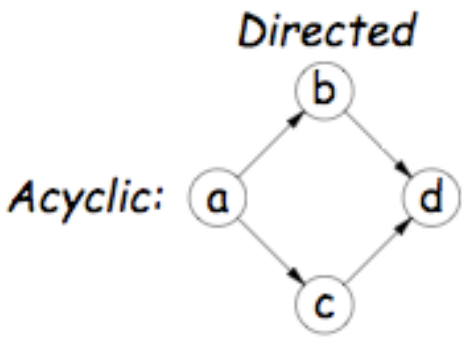
```
void traverse (Graph G) {
   For each vertex v in G {
       traverse (G, v);
   }
}

void traverse (Graph G, vertex v) {
   if (v is unmarked) {
       mark(v);
       VISIT(v);
       For each edge (v,w) in G {
          traverse(G,w)
       }
   }
}
```

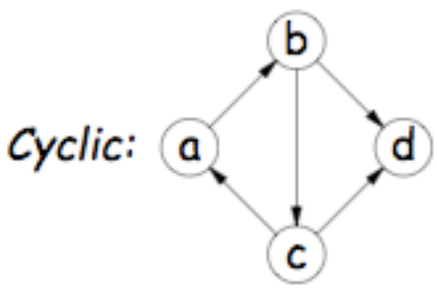
Running Time of DFS

- DFS must check *edge* once:
  - $O(|V| + |E|)$ time if you use an adjacency list
  - $O(|V|^2)$ time if you use an adjacency matrix.
  - Hence, an adjacency list is asymptotically faster if the graph is sparse.
- What's an application of DFS?
  - Suppose you want to determine whether there is a path from a vertex *u* to another vertex *v*.
  - Just do DFS from *u*, and see if *v* gets visited.
  - If not, you can't there from here.

Cycles

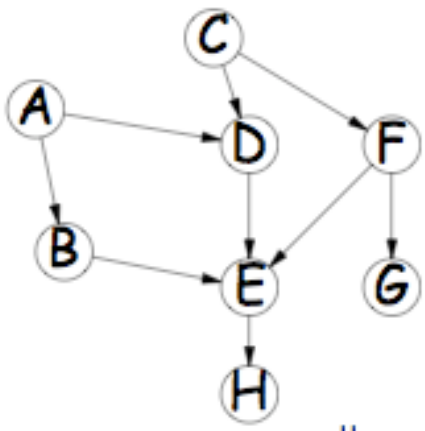
- A *cycle* is a *path* without repeated edges leading from a vertex back itself.
- A graph is called ***cyclic*** if it has a a cycle, else *acyclic.*
- DAG: *directed acyclic graph*.





Topological Sort

- Problem:
  - given a DAG, find a *linear ordering* of all the *vertices* that's consistent with the edges: if *(u,v)* is an edge, then *u* appears before *v* in the ordering.
- A topological sort of a DAG can be viewed as an *ordering* of its vertices along a horizontal line so that all directed edges go from *left to right.*

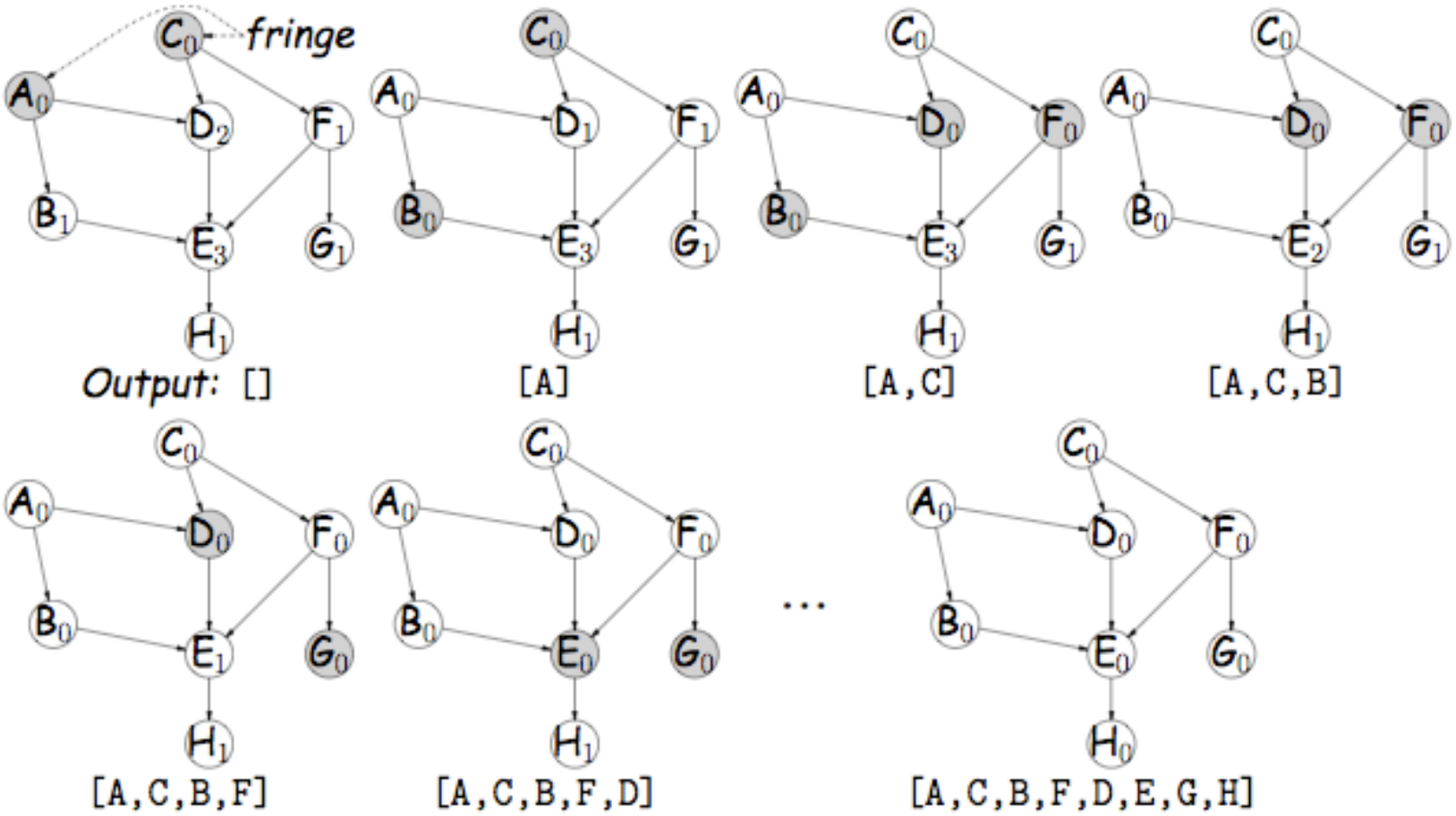


⇓

[A,B,C,F,D,G,E,H], or
[A,C,B,D,F,E,G,H], or
[A,B,C,F,D,E,H,G], or
⋮

Topological Sort

```
Set<Vertex> fringe;
fringe = set of all vertex with no predecessors;

while (!fringe.isEmpty()) {
   Vertex v = fringe.removeOne ();
   add v to end of result list;
   For each edge (v,w) {
      decrease predecessor count of w;
      if (predecessor count of w == 0)
         fringe.add (w);
   }
}
```


fringe
Output: []
[A]
[A,C]
[A,C,B]
[A,C,B,F]
[A,C,B,F,D]
...
[A,C,B,F,D,E,G,H]

Topological Sort Using Recursion

- Observation: if we do a depth-first traversal on a DAG whose edges are reversed, and execute the recursive traverse procedure, we finish executing traverse(G,v) in proper topologically sorted order.

```
void topologicalSort (Graph G) {
   For each vertex v in G {
        traverse (G, v);
   }
}
void traverse (Graph G, Vertex v) {
   if (v is unmarked) {
        mark(v);
        VISIT(v);
        For each edge (v,w) in G {
            traverse(G,w)
        }
        add v to the result list.
   }

 }
}
```

Weighted Graphs

- A weighted graph is a graph in which each edge is labeled with a numerical weight.
- A weight might express:
  - the distance between two vertices,
  - the cost of moving from one to the other,
  - the resistance between two points in an electrical circuit.
- Representations:
  - In an adjacency matrix, use an array of ints/doubles rather than booleans.
  - In an adjacency list, each node must be expanded to include a weight, in addition to the reference to the destination vertex.

Shortest Path Problem

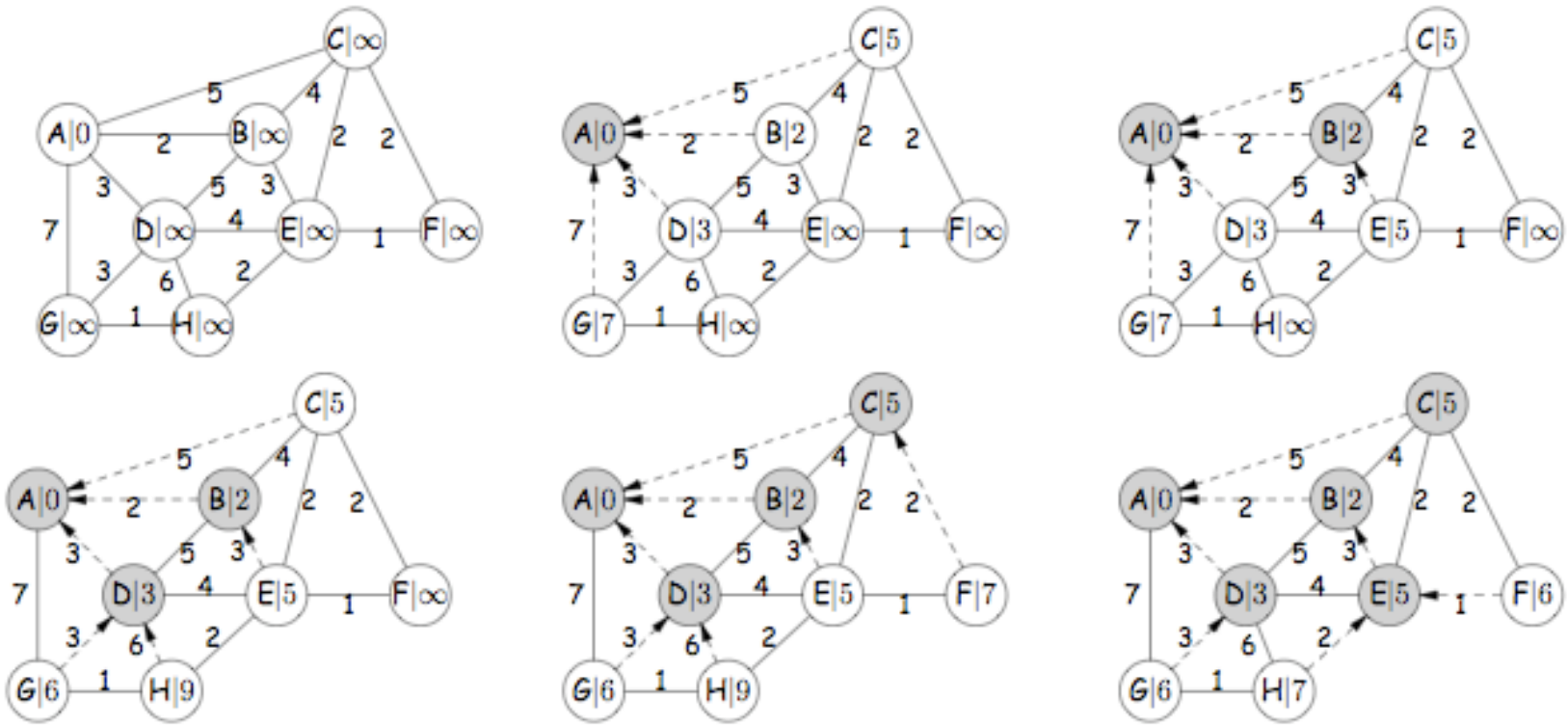
Suppose a graph represents a highway map, and each road is labeled with the amount of time it takes to drive from one interchange to the next. What's the fastest way to drive from Berkeley to Los Angeles?

Shortest Paths: Dijkstra's Algorithm

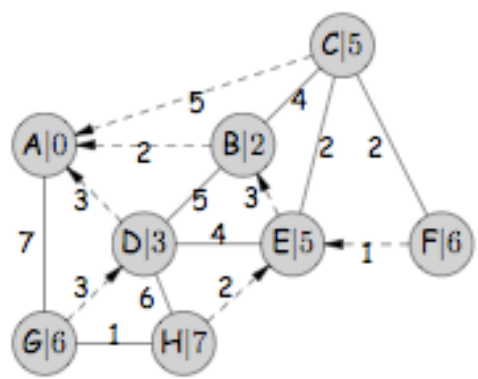
- Problem: Given a graph (directed or undirected) with *non-negative* edge weights, compute shortest paths from given source vertex, *s*, to all vertices.
  - "Shortest" = sum of weights along path is smallest.
  - For each vertex, keep estimated distance from *s*, . . .
  - . . . and of preceding vertex in shortest path from s.

```
PriorityQueue<Vertex> fringe;
For each vertex 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.removeMin();
   For each edge (v,w) {
    if (v.dist() + weight(v,w) < w.dist()){
       w.dist() = v.dist() + weight(v,w); w.back() = v;
    }
   }
}
```

What's a general property of *v*?



Final result:



Shortest-path tree

X|d processed node at distance d

Y|d node in fringe at distance d