CS61B Lecture 33

Wednesday, April 16, 2020

Graphs

Graphs are data structures that can be used to express non-hierarchically related items. For example,

- Networks: pipelines, roads, assignment problems
- Processes: flow charts, Markov models
- Partial orderings: PERT charts, makefiles
- Connected structures as used in Git

A graph consists of:

- A set of **nodes**, otherwise known as vertices.
- A set of **edges**, pairs of nodes. Nodes with an edge between are **adjacent**.
- Depending on the problem, nodes or edges may have **labels** or **weights**.

Typically we call the node set

$$V = \{v_0, v_1 \dots\}$$

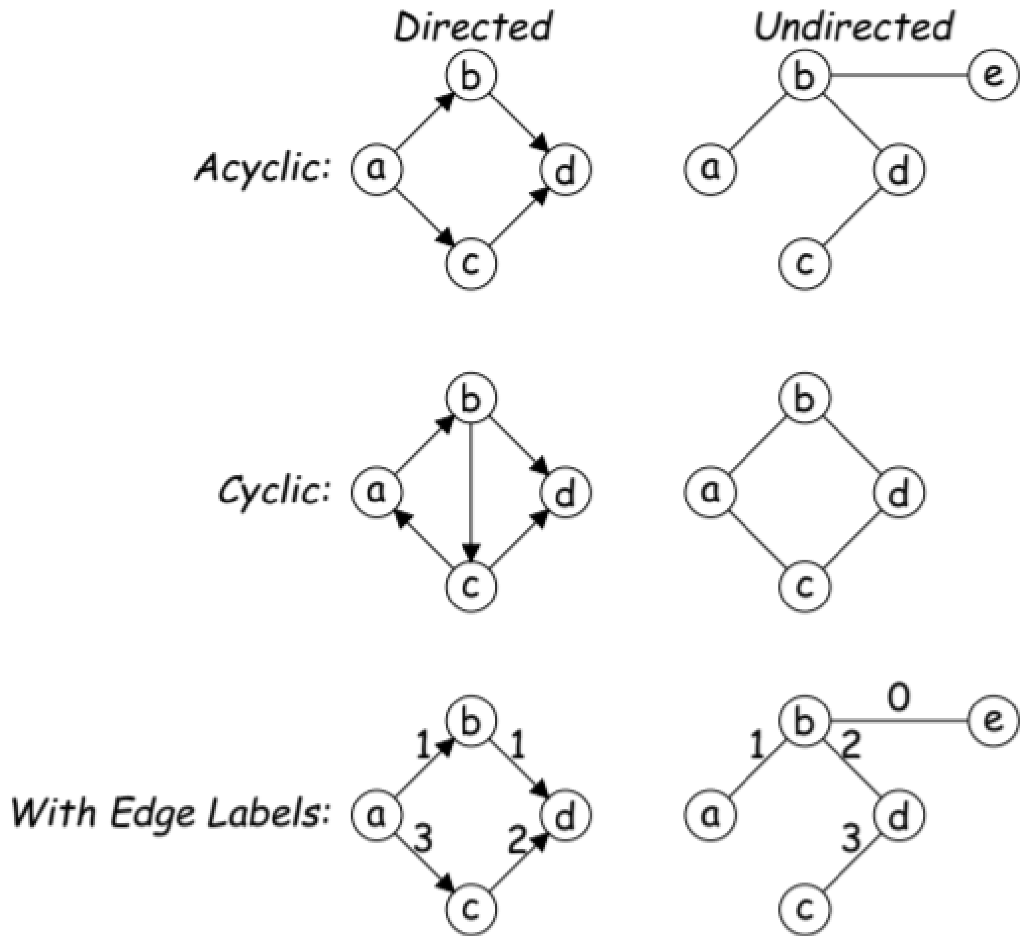
and the edge set

$$E$$

If the edges have an order (first, second), they are called **directed edges**, and the graph is a **directed graph** (digraph). Otherwise, it is an undirected graph.

Edges are **incident** to their nodes. Directed edges exit one node and enter the next.

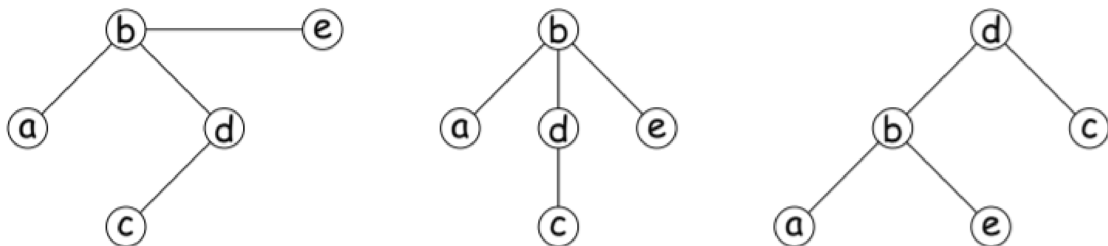
A cycle is a path without repeated edges leading from a node back to itself. A graph is **cyclic** if it has a cycle, otherwise it is **acyclic**. We can combine these terms, e.g. Directed Acyclic Graph - DAG.



Trees are graphs. A graph is connected if there is a (possibly directed) path between every pair of nodes. That is, if one node pair is reachable from the other.

A DAG is a (rooted) tree iff connected, and every node but the root has exactly one parent.

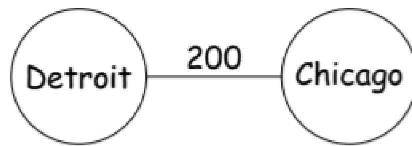
A connected, acyclic, undirected graph is also called a **free tree**, free as in we are free to pick the root. For example, all of the following are the same graph:



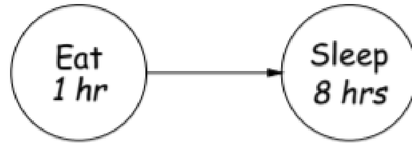
Uses

Here are a few ways one might use graphs:

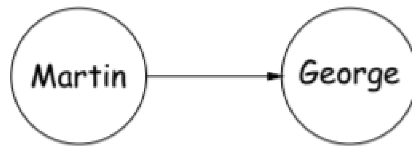
Edge = Connecting road, with length.



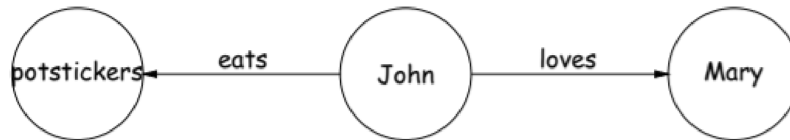
Edge = Must be completed before; Node label = time to complete.



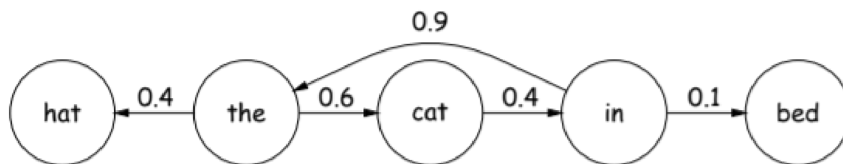
Edge = Begat



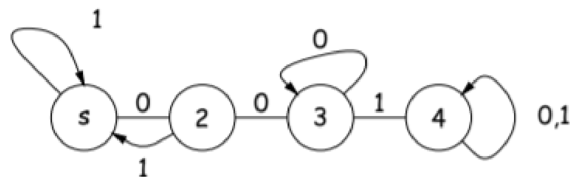
Edge = some relationship



Edge = next state might be (with probability)



Edge = next state in state machine, label is triggering input. (Start at s. Being in state 4 means "there is a substring '001' somewhere in the input".)

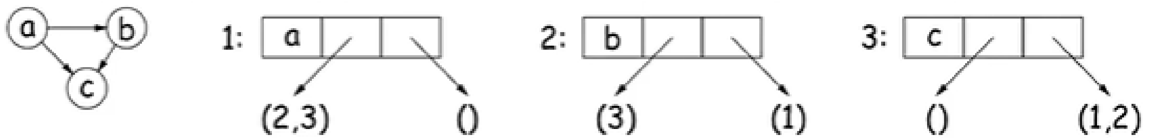


Representation

It is often useful to number the nodes, and use the numbers in edges.

Edge List Representation

This is where we each node contains some kind of list, of its successors (and possibly predecessors):



Edge Set Representation

Collection of all edges. For above:

$$\{(1,2),(1,3),(2,3)\}$$

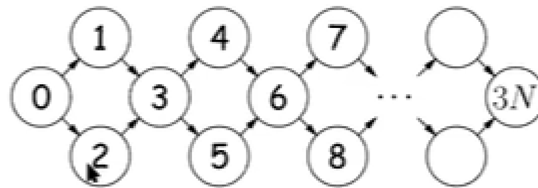
Adjacency matrix

We can also represent connections with matrix entry. Shown above:

	1	2	3
1	0	1	1
2	0	0	1
3	0	0	1

Traversing Graphs

Many algorithms on graphs depend on traversing all or some nodes. We can't quite use recursion because of cycles, and even in acyclic graphs, we can still get combinatorial explosions:



In the above example, treating 0 as the root and doing recursive traversal down the two edges at each node gives us:

$$\Theta(2^N)$$

operations!

Typically, we try to visit each node a constant number of times.

Recursive Depth-First Traversal

We can fix looping and combinatorial problems using the "breadcrumb" method used in the maze example in an earlier lectures. We mark nodes as we traverse them, and don't traverse previously marked nodes.

It makes sense to talk about preorder and postorder traversal as we did for trees.

```
void preorderTraverse(Graph G, Node v) {
    if (!v.marked) {
        mark(v);
        // visit v
        for (/* Edge(v, w) in G */) {
            preorderTraverse(G, w);
        }
```

```
    }
}

void postorderTraverse(Graph G, Node v) {
    if (!v.marked) {
        mark(v);
        for (/* Edge(v, w) in G */) {
            postorderTraverse(G, w);
        }
        // visit v
    }
}
```

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

```
void preorderTraverse(Graph G) {
    clearAllMarks();
    for (Node v : G) {
        preorderTraverse(G, v);
    }
}

void postorderTraverse(Graph G) {
    clearAllMarks();
    for (Node v : G) {
        postorderTraverse(G, v);
    }
}
```

Topological Sorting

With topological sorting, the problem we are trying to solve is finding a linear order of nodes consistent with the edges. That is, order the nodes

$$v_0, v_1, \ldots, v_k$$

such that

$$v_k$$

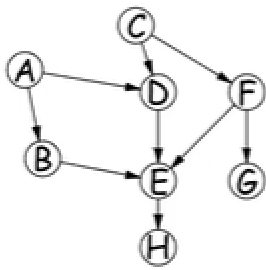
is never reachable from

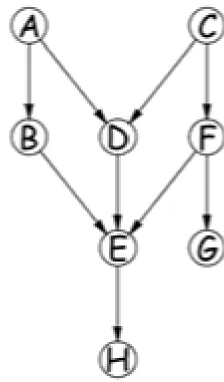
$$v_{k'}$$

if

$$k' > k$$

Graph (two views)





Possible Orderings

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

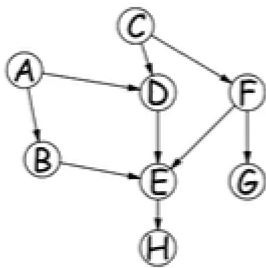
Gmake (a build control system) and PERT charts both solve this.

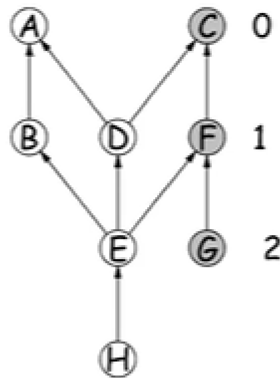
Sorting and Depth-First Search

Suppose we reverse the links on the graph above. If we do a recursive DFS on the reverse graph, 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 it is safe to put that node first.

In general, a postorder traversal of a reversed graph visits nodes only after all predecessors have been visited.





Numbers show post-order traversal order starting from G: every-thing that must come before G.

General Graph Traversal Algorithm

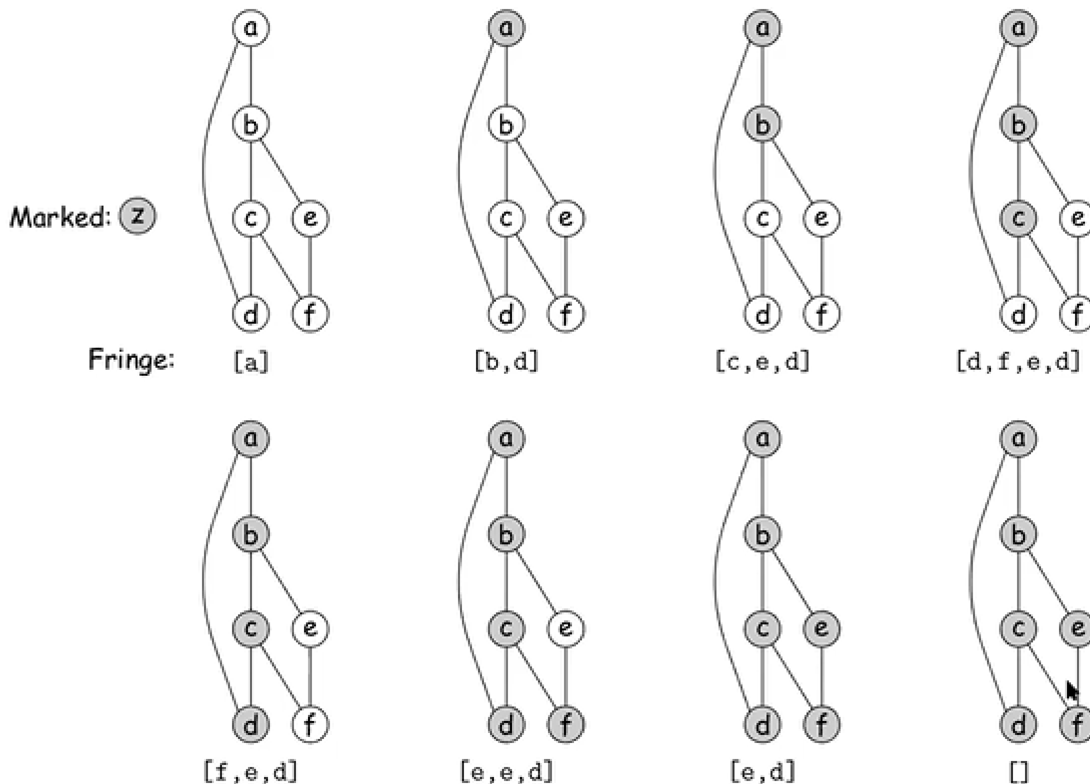
```
COLLECTION OF VERTICES fringe;
fringe = INITIAL COLLECTION;
while (!fringe.isEmpty()) {
    Vertex v = fringe.removeHighestPriorityItem();
    if (!v.marked) {
        v.mark();
        v.visit();
        for (Edge w : v) {
            if (w.needsProcessing()) {
                fringe.add(w);
            }
        }
    }
}
```

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

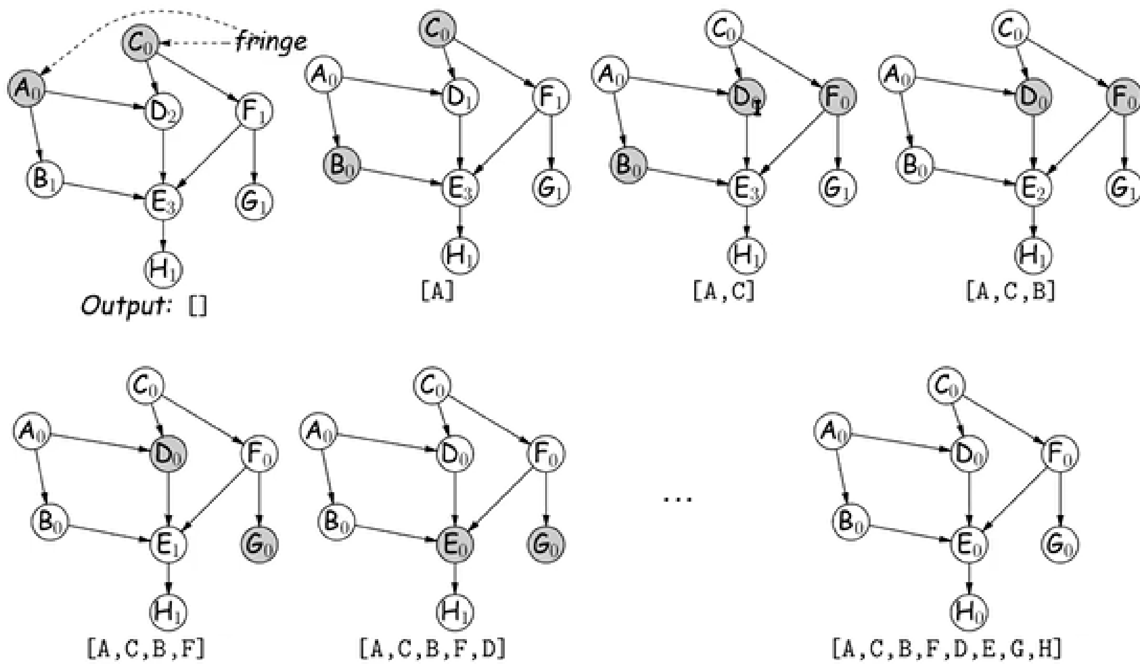
DFS Algorithm

If we wanted to visit every node reachable from `v` once, we visit nodes further from starting point first:

```
Stack<Vertex> fringe;
fringe = stack containing v;
while (!fringe.isEmpty()) {
    Vertex v = fringe.pop();
    if (!v.marked) {
        v.mark();
        v.visit();
        for (Edge w : v) {
            if (!w.marked) {
                fringe.push(w);
            }
        }
    }
}
```



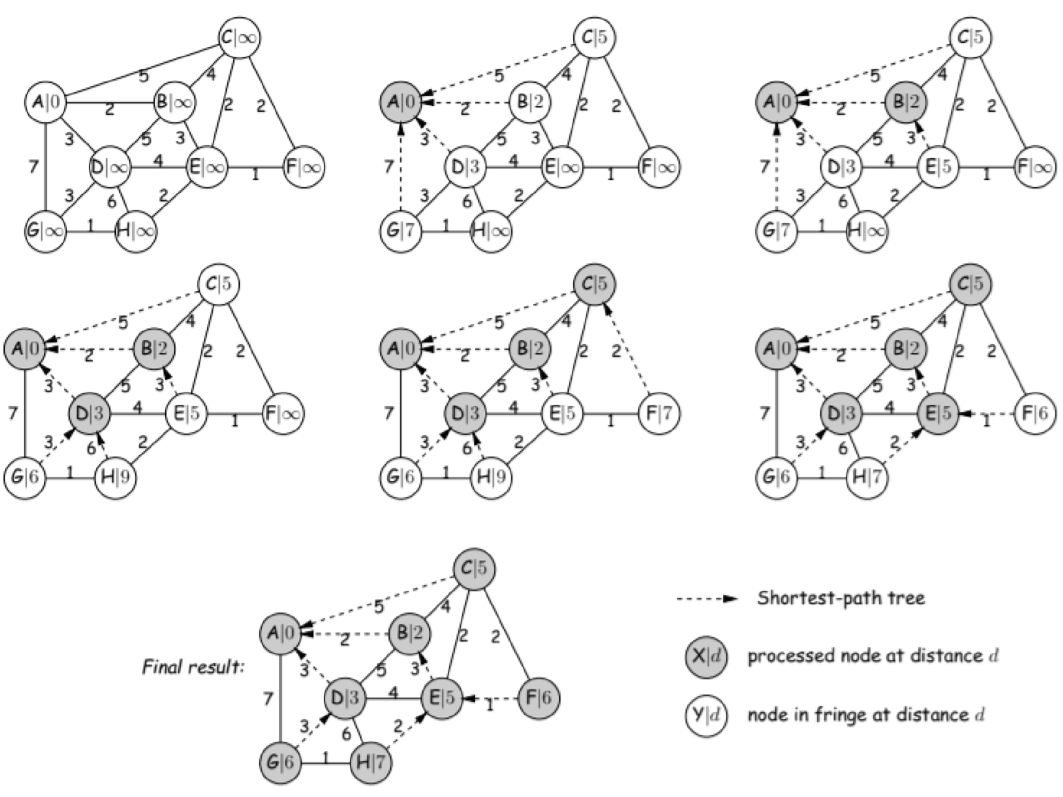
Topological Sort in Action



Shortest Paths

Djikstra's Algorithm is a series of steps on how to compute the shotrest paths from a given source node, `s`, to all nodes, given a graph with non-negative edge weights, where shortest means sum of weights along the path is the smallest.

```
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 (Edge w : v) {
    if (v.dist() + weight(v,w) < w.dist())
        { w.dist() = v.dist() + weight(v,w); w.back() = v; }
    }
}
```


Final result:
Shortest-path tree
X|d processed node at distance d
Y|d node in fringe at distance d