Introduction to Graphs: Breadth-First, Depth-First Search, Topological Sort
Chapter 23

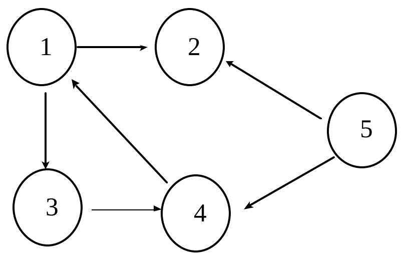
Graphs

So far we have examined trees in detail. Trees are a specific instance of a construct called a **graph**. In general, a graph is composed of edges *E* and vertices *V* that link the nodes together. A graph G is often denoted G=(V,E) where V is the set of vertices and E the set of edges.

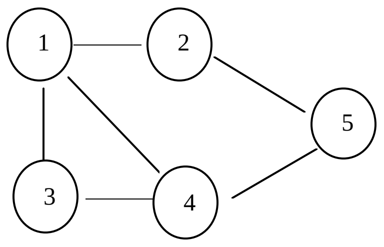
Two types of graphs:

1. Directed graphs: G=(V,E) where E is composed of ordered pairs of vertices; i.e. the edges have direction and point from one vertex to another.
2. Undirected graphs: G=(V,E) where E is composed of unordered pairs of vertices; i.e. the edges are bidirectional.

Directed Graph:



Undirected Graph:



The **degree** of a vertex in an undirected graph is the number of edges that leave/enter the vertex. The degree of a vertex in a directed graph is the same,but we distinguish between in-degree and out-degree. Degree = in-degree + out-degree.

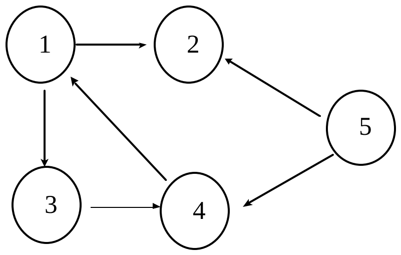
A path from u to v is <u, w1, …v> and (u,w1)(w1,w2)(w2,w3)…(w_n,v)

The running time of a graph algorithm expressed in terms of E and V, where E = |E| and V=|V|; e.g. G=O(EV) is |E| * |V|

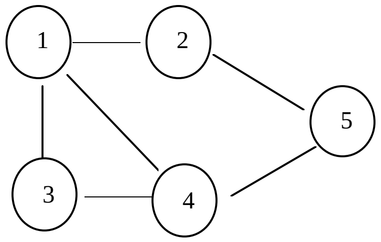
Implement a graph in three ways:

1. Pointers/memory for each node
2. Adjacency List
3. Adjacency-Matrix

Using an Adjacency List: List of pointers for each vertex



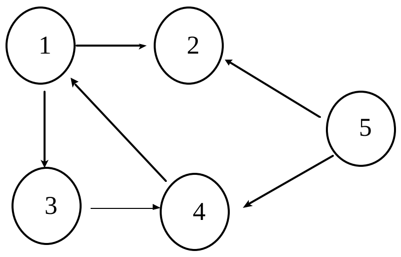
A				
1	→	2	→	3
2	→			
3	→	4		
4	→	1		
5	→	2	→	4



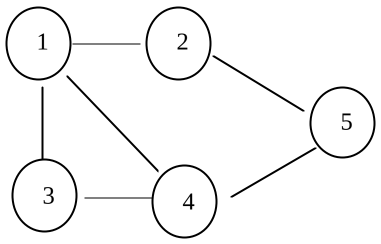
A						
1	→	2	→	3	→	4
2	→	1	→	5		
3	→	4	→	1		
4	→	1	→	3		
5	→	2	→	4		

The sum of the lengths of the adjacency lists is 2|E| in an undirected graph, and |E| in a directed graph.
The amount of memory to store the array for the adjacency list is O(max(V,E))=O(V+E).

Using an Adjacency Matrix:



	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	0
3	0	0	0	1	0
4	1	0	0	0	0
5	0	1	0	1	0



	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	1
3	1	0	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

When is using an adjacency matrix a good idea? A bad idea?
The matrix always uses $\Theta(v^2)$ memory. Usually easier to implement and perform lookup than an adjacency list.

Sparse graph: very few edges.
Dense graph: lots of edges. Up to v^2 edges if fully connected.

The adjacency matrix is a good way to represent a *weighted graph*. In a weighted graph, the edges have weights associated with them. Update matrix entry to contain the weight. Weights could indicate distance, cost, etc.

Search: The goal is to methodically explore every vertex and every edge; perhaps to do some processing on each. Will assume adjacency-list representation of the input graph.

Breadth-First-Search (BFS) :

Example 1: Binary Tree. This is a special case of a graph. The order of search is across levels. The root is examined first; then both children of the root; then the children of those nodes, etc.

(draw tree and show search path)

Sometimes we can stop the algorithm if we are looking for a particular element, but the general BFS algorithm runs through every node.

Example 2: directed graph:

1. Pick a source vertex S to start.
2. Find (or discover) the vertices that are adjacent to S.
3. Pick each child of S in turn and discover their vertices adjacent to that child.
4. Done when all children have been discovered and examined.

This results in a tree that is rooted at the source vertex S.

The idea is to find the distance from some Source vertex by expanding the “frontier” of what we have visited.

Pseudocode: Uses FIFO Queue Q

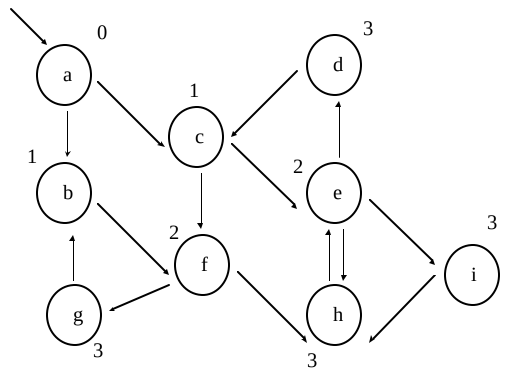
```
BFS(s)                                  ; s is our source vertex
    for each u∈ V - {s}                 ; Initialize unvisited vertices to ∞
        do d[u] ← ∞
    d[s] ← 0                            ; distance to source vertex is 0
    Q ← {s}                             ; Queue of vertices to visit
    while Q<>0 do
        remove u from Q
        for each v ∈ Adj[u] do          ; Get adjacent vertices
            if d[v]= ∞
                then d[v] ← d[u]+1      ; Increment depth
                put v onto Q            ; Add to nodes to explore
```

Differences from book:

Not tracking predecessor’s via *p*

Uses infinity instead of colors (white, gray)
Not using enqueue, dequeue; combine dequeue with head of Q

Example: (this is the final state, start with 0 and infinity as values)



Initially, d[a] is set to 0 and the rest to ∞.
Q $\leftarrow$ [a].
Remove head: Q$\leftarrow$[]
    children of a are c,b
    d[c]= ∞, d[b]= ∞ so d[c] $\leftarrow$d[a]+1=1, d[b] $\leftarrow$d[a]+1=1
    Q$\leftarrow$[c b]

Remove head: Q$\leftarrow$[b]
    children of c are e,f
    d[e]= ∞, d[f]= ∞ so d[e] $\leftarrow$d[c]+1=2, d[f] $\leftarrow$d[c]+1=2
    Q$\leftarrow$[b e f]

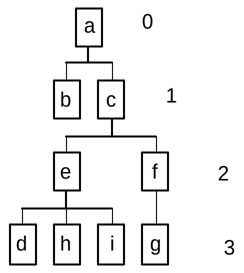
Remove head: Q$\leftarrow$[e f]
    children of b is f
    d[f] <> ∞, nothing done with it

Remove head: Q$\leftarrow$[f]
    children of e is d, i, h
    d[d]= ∞, d[i]= ∞, d[h]= ∞ so d[d] $\leftarrow$d[i] $\leftarrow$d[h] $\leftarrow$d[e]+1=3
    Q$\leftarrow$[f d i h]

Remove head: Q$\leftarrow$[d i h]
    children of d is g
    d[g]= ∞, so d[g] $\leftarrow$d[d]+1 = 3
    Q$\leftarrow$[d i h g]

Each of these has children that are already has a value less than ∞, so these will not set any further values and we are done with the BFS.

Can create a tree out of the order we visit the nodes:



Memory required: Need to maintain Q, which contains a list of all fringe vertices we need to explore.

Runtime: O(V+E) ; O(E) to scan through adjacency list and O(V) to visit each vertex. This is considered linear time in the size of G.

Claim: BFS always computes the shortest path distance in d[I] between S and vertex I. We will skip the proof for now.

What if some nodes are unreachable from the source? (reverse c-e,f-h edges). What values do these nodes get?

Depth First Search: Another method to search graphs.

Example 1: DFS on binary tree. Specialized case of more general graph. The order of the search is down paths and from left to right. The root is examined first; then the left child of the root; then the left child of this node, etc. until a leaf is found. At a leaf, backtrack to the lowest right child and repeat.

(Show example binary tree DFS)

Example 2: DFS on directed graph.

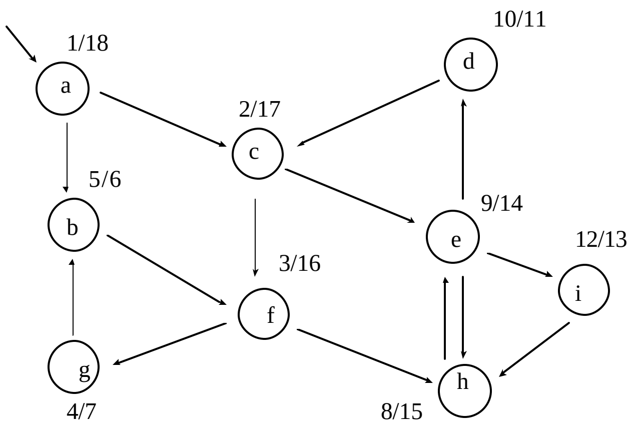
1. Start at some source vertex S.
2. Find (or explore) the first vertex that is adjacent to S.
3. Repeat with this vertex and explore the first vertex that is adjacent to it.
4. When a vertex is found that has no unexplored vertices adjacent to it then backtrack up one level
5. Done when all children have been discovered and examined.

Results in a forest of trees.

Pseudocode:

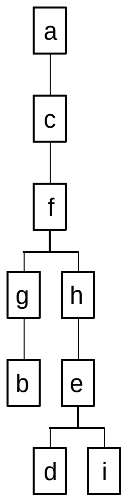
```
DFS(s)
    for each vertex $u \in V$
        do color[u] $\leftarrow$ White              ; not visited
    time $\leftarrow$ 1                             ; time stamp
    for each vertex $u \in V$
        do if color[u]=White
            then DFS-Visit(u,time)

DFS-Visit(u,time)
    color[u] $\leftarrow$ Gray                      ; in progress nodes
    d[u] $\leftarrow$ time                          ; d=discover time
    time $\leftarrow$ time+1
    for each v $\in$ Adj[u] do
        if color[u]=White
            then DFS-Visit(v,time)
    color[u] $\leftarrow$ Black
    f[u] $\leftarrow$ time $\leftarrow$ time+1      ; f=finish time
```

Example:

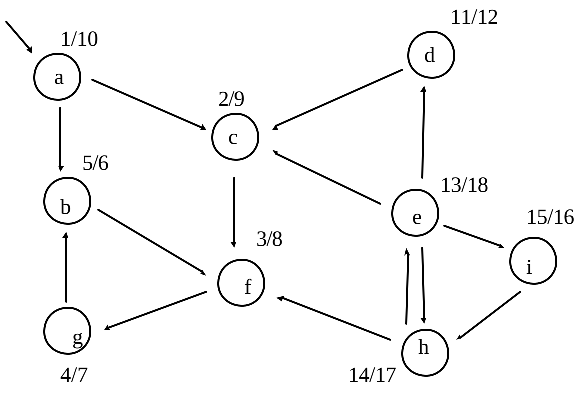


Numbers are Discover/Finish times. We could have different visit times depending on which edges we pick to traverse during the DFS.

The tree built by this search looks like:



What if some nodes are unreachable? We still visit those nodes in DFS. Consider if c-e, f-h links were reversed. Then we end up with two separate trees:



Still visit all vertices and get a forest: a set of unconnected graphs without cycles (a tree is a connected graph without cycles).

Time for DFS:

$O(V^2)$ - DFS loop goes O(V) times once for each vertex (can't be more than once, because a vertex does not stay white), and the loop over Adj runs up to V times.

But…

The for loop in DFS-Visit looks at every element in Adj once. It is charged once per edge for a directed graph, or twice if undirected. A small part of Adj is looked at during each recursive call but over the entire time the for loop is executed only the same number of times as the size of the adjacency list which is $\Theta(E)$.

Since the initial loop takes $\Theta(V)$ time, the total runtime is $\Theta(V+E)$. This is considered linear in terms of the size of the input adjacency-list representation. So if there are lots of edges then E dominates the runtime, otherwise V does.

Note: Don't have to track the backtracking/fringe as in BFS since this is done for us in the recursive calls and the stack. The stack makes the nodes ordered LIFO. The amount of storage needed is linear in terms of the depth of the tree.

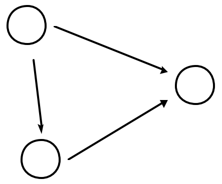
Types of Edges: There are 4 types. These will be useful later. DFS can be modified to classify edges as being of the correct type:

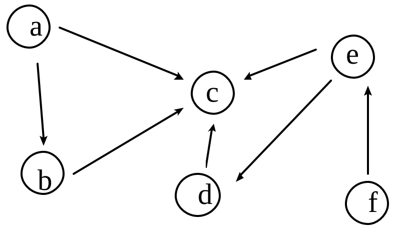
1. Tree Edge: An edge in a depth-first forest. Edge(u,v) is a tree edge if v was first discovered from u.
2. Back Edge: An edge that connects some vertex to an ancestor in a depth-first tree. Self-loops are back edges.
3. Forward Edge: An edge that connects some vertex to a descendant in a depth-first tree.
4. Cross Edge: Any other edge.

DAG's

Nothing to do with sheep! A DAG is a Directed Acyclic Graph. This is a directed graph that contains no cycles.

Examples:

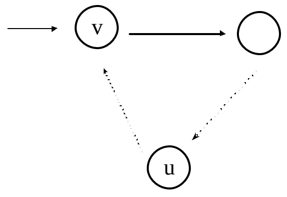




A directed graph D is acyclic iff a DFS of G yields no back edges.

Proof: Trivial. Acyclic means no back edge because a back edge makes a cycle. Suppose we have a back edge (u,v). Then v is an ancestor of u in the depth-first forest. But then there is already a path from v to u and the back edge makes a cycle.

Suppose G has a cycle c. But then DFS of G will have a back edge. Let v be the first vertex in c found by DFS and let u be a vertex in c that is connected back to v. Then when DFS expands the children of u, the vertex v is found. Since v is an ancestor of u the edge (u,v) is a back edge.



Topological Sort of a dag

A topological sort of a dag is an ordering of all the vertices of G so that if (u,v) is an edge then u is listed (sorted) before v. This is a different notion of sorting than we are used to. a,b,f,e,d,c and f,a,e,b,c,d are both topological sorts of the above dag. There may be multiple sorts; this is okay since a is not related to f, either vertex can come first.

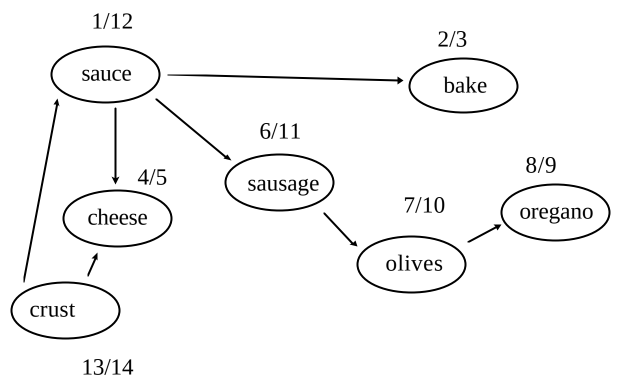
Main use: Indicate order of events, what should happen first

Algorithm for Topological-Sort:

1. Call DFS(G) to compute f(v), the finish time for each vertex.
2. As each vertex is finished insert it onto the front of the list.
3. Return the list.

Time is Θ (V+E), time for DFS.

Example: Pizza directed graph



DFS: Start with sauce.
The numbers indicate start/finish time. We insert into the list in reverse order of finish time.

Crust, Sauce, Sausage, Olives, Oregano, Cheese, Bake

Why does this work? Because we don't have any back edges in a dag, so we won't return to process a parent until after processing the children. We can order by finish times because a vertex that finishes earlier will be dependent on a vertex that finishes later.