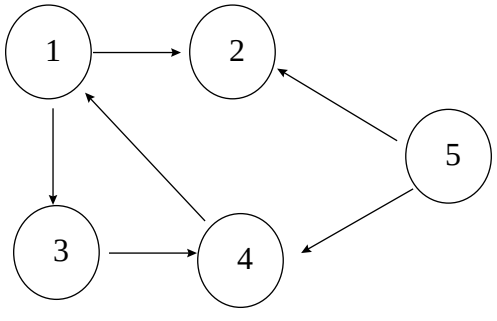
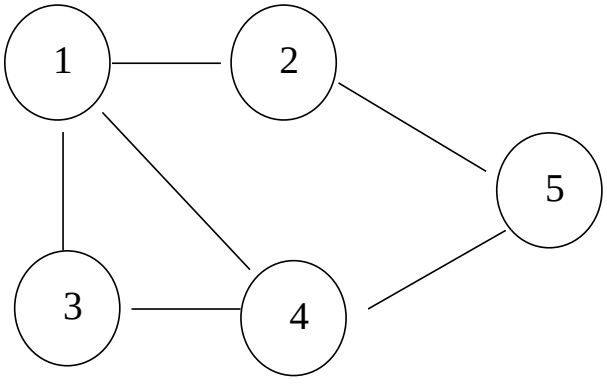
Review of Graphs

- 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:
  - 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.
  - Undirected graphs: G=(V,E) where E is composed of unordered pairs of vertices; i.e. the edges are bidirectional.

Directed Graph



Undirected Graph



Graph Terminology

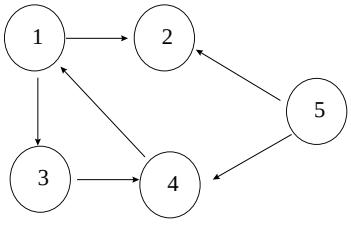
- The **degree** of a vertex in an undirected graph is the number of edges that leave/enter the vertex.
- The degree of a vertex in a directed graph is the same, but we distinguish between in-degree and out-degree. Degree = in-degree + out-degree.
- The running time of a graph algorithm expressed in terms of E and V, where E = |E| and V=|V|; e.g. G=O(EV) is |E| * |V|

Implementing a Graph

- Implement a graph in three ways:
  - Adjacency List
  - Adjacency-Matrix
  - Pointers/memory for each node (actually a form of adjacency list)

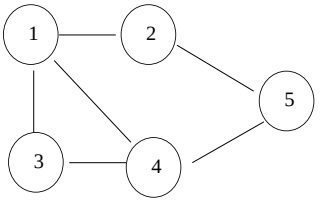
Adjacency List

- List of pointers for each vertex



1 → 2 → 3
2 →
3 → 4
4 → 1
5 → 2 → 4

Undirected Adjacency List

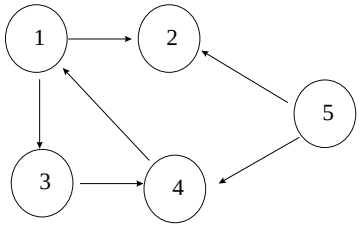


1 → 2 → 3 → 4
2 → 1 → 5
3 → 4 → 1
4 → 1 → 3
5 → 2 → 4

Adjacency List

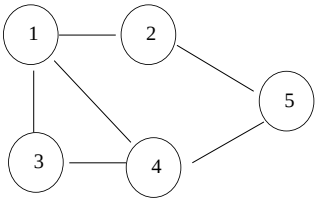
- The sum of the lengths of the adjacency lists is 2|E| in an undirected graph, and |E| in a directed graph.
- The amount of memory to store the array for the adjacency list is O(max(V,E))=O(V+E).

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

Undirected Adjacency Matrix



	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

Adjacency Matrix vs. List?

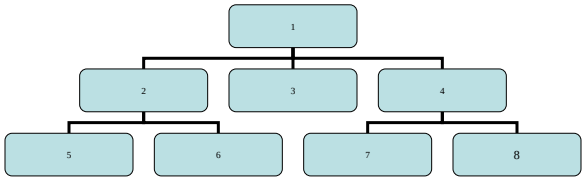
- The matrix always uses $\Theta(v^2)$ memory. Usually easier to implement and perform lookup than an adjacency list.
- Sparse graph: very few edges.
- Dense graph: lots of edges. Up to $O(v^2)$ edges if fully connected.
- The adjacency matrix is a good way to represent a *weighted graph*. In a weighted graph, the edges have weights associated with them. Update matrix entry to contain the weight. Weights could indicate distance, cost, etc.

Searching a Graph

- Search: The goal is to methodically explore every vertex and every edge; perhaps to do some processing on each.
- For the most part in our algorithms we will assume an adjacency-list representation of the input graph.

Breadth First Search

- 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.



Breadth First Search

- Example 2: Directed Graph

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

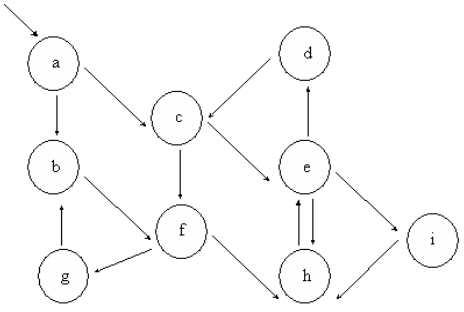
- This results in a tree that is rooted at the source vertex S.
- The idea is to find the distance from some Source vertex by expanding the “frontier” of what we have visited.

Breadth First Search Algorithm

- Pseudocode: Uses FIFO Queue Q

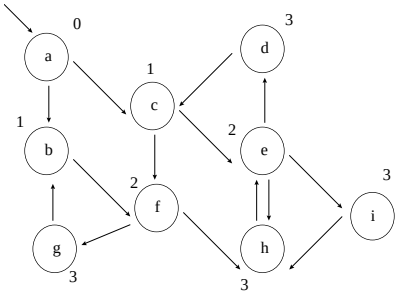
```
BFS(s)                              ; s is our source vertex
    for each u ∈V - {s}             ; Initialize unvisited vertices to ∞
        do d[u]← ∞
    d[s] ←0                         ; distance to source vertex is 0
    Q← {s}                          ; Queue of vertices to visit
    while 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
```


```
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
```

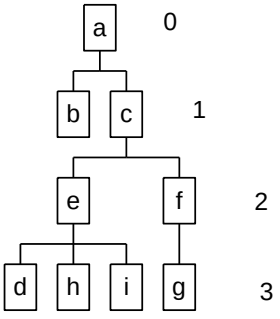



BFS Example

- Final State shown



Can create tree out of order we visit nodes

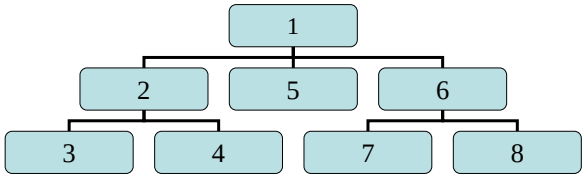


BFS Properties

- Memory required: Need to maintain Q, which contains a list of all fringe vertices we need to explore, O(V)
- 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.
- What if some nodes are unreachable from the source? (reverse c-e,f-h edges). What values do these nodes get?

Depth First Search

- 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.



Depth First Search

- Example 2: DFS on directed graph.

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

- Results in a forest of trees.

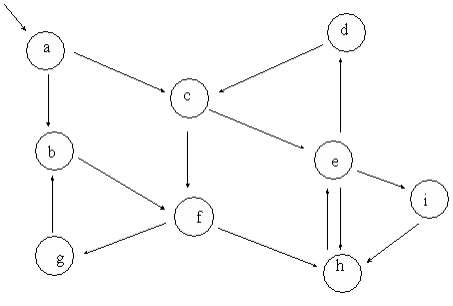
DFS Algorithm

- Pseudocode

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

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

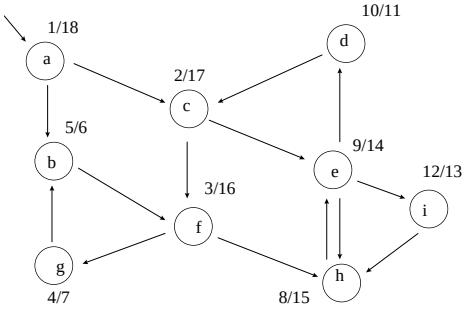
```
DFS(s)
        for each vertex u ∈V
                do color[u]←White                   ; not visited
        time←1                                      ; time stamp
        for each vertex u ∈V
                do if color[u]=White
                   then DFS-Visit(u,time)
```



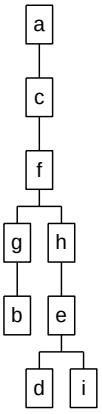

```
DFS-Visit(u,time)
        color[u] ←Gray
        d[u] ←time
        time←time+1
        for each v ∈Adj[u] do
                if color[u]=White
                   then DFS-Visit(v,time)
        color[u] ←Black
        f[u] ←time←time+1
```

DFS Example

- Result (start/finish time):

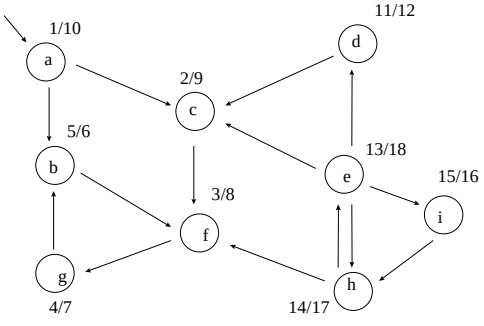


Tree:



DFS Example

- 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).

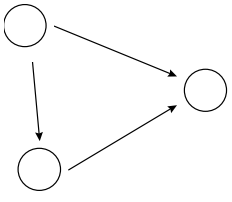


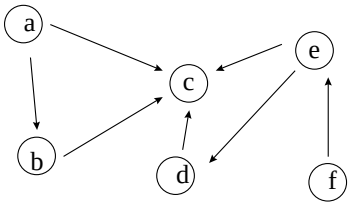
DFS Runtime

- $O(V^2)$ - 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 (E).
  - Since the initial loop takes (V) time, the total runtime is (V+E).
- 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 amount of storage needed is linear in terms of the depth of the tree.

DAG

- Directed Acyclic Graph
  - Nothing to do with sheep
  - This is a directed graph that contains no cycles
- A directed graph D is acyclic iff a DFS of G yields no back edges (an edge to a previously visited node).
  - Proof: Trivial. Acyclic means no back edge because a back edge makes a cycle.

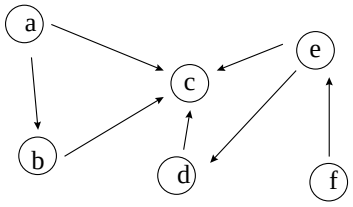




DAG

- DAG's are useful in various situations, e.g.:
  - Detection of loops for reference counting / garbage collection
  - Topological sort

- Topological sort
  - 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,d,c are both topological sorts of the dag below. There may be multiple sorts; this is okay since a is not related to f, either vertex can come first.



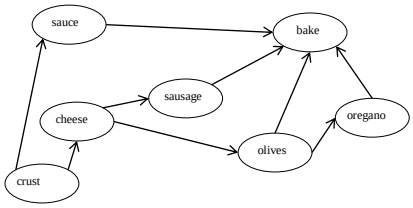
Topological Sort

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

- Algorithm for Topological-Sort:
  - Call DFS(G) to compute f(v), the finish time for each vertex.
  - As each vertex is finished insert it onto the front of the list.
  - Return the list.
- Time is $\Theta(V+E)$, time for DFS.

Topological Sort Example

- Making Pizza



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

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.