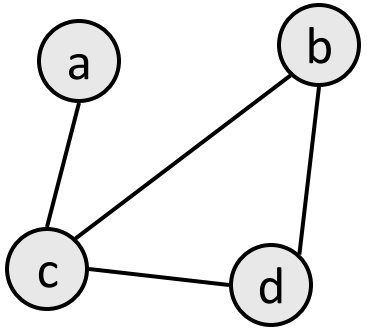
Graph Traversal

DFS & BFS

Graphs

- **graph**: A data structure containing:
  - a set of **vertices** V, *(sometimes called nodes)*
  - a set of **edges** E, where an edge represents a connection between 2 vertices.
    - Graph $G = (V, E)$
    - an edge is a pair (v, w) where v, w are in V



- the graph at right:
  - $V = \{a, b, c, d\}$
  - $E = \{(a, c), (b, c), (b, d), (c, d)\}$

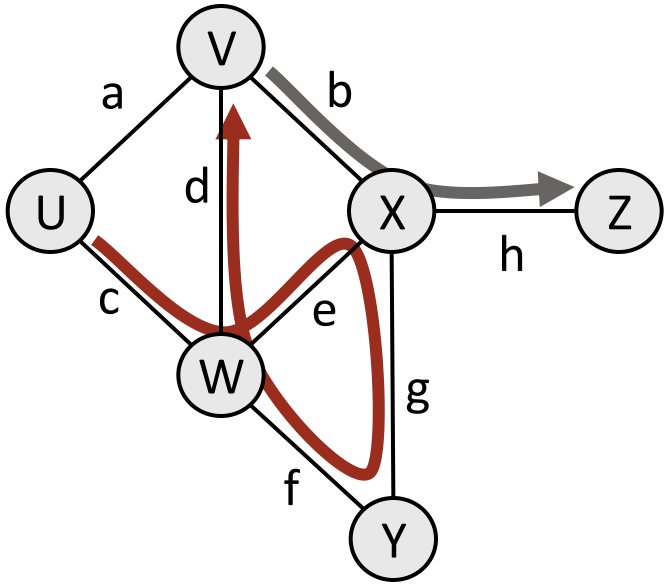
- **degree**: number of edges touching a given vertex.
  - at right: a=1, b=2, c=3, d=2

Paths

- **path**: A path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*.
  - can be represented as vertices visited, or edges taken
  - example, one path from *V* to *Z*: {b, h} or {V, X, Z}
  - What are two paths from U to Y?

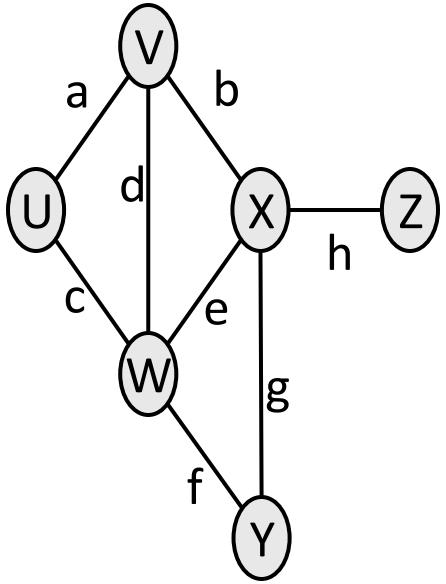
- **path length**: Number of vertices or edges contained in the path.

- **neighbor** or **adjacent**: Two vertices connected directly by an edge.
  - example: V and X



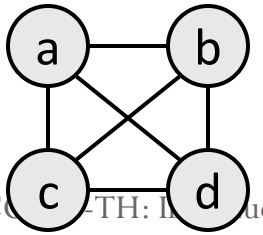
Reachability, connectedness

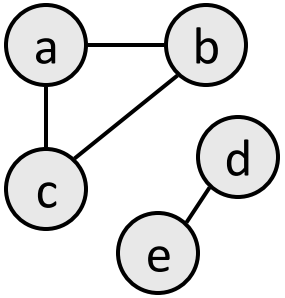
- **reachable**: Vertex *a* is *reachable* from *b* if a path exists from *a* to *b*.



- **connected**: A graph is *connected* if every vertex is reachable from any other.
  - Is the graph at top right connected?

- **strongly connected**: When every vertex has an edge to every other vertex.



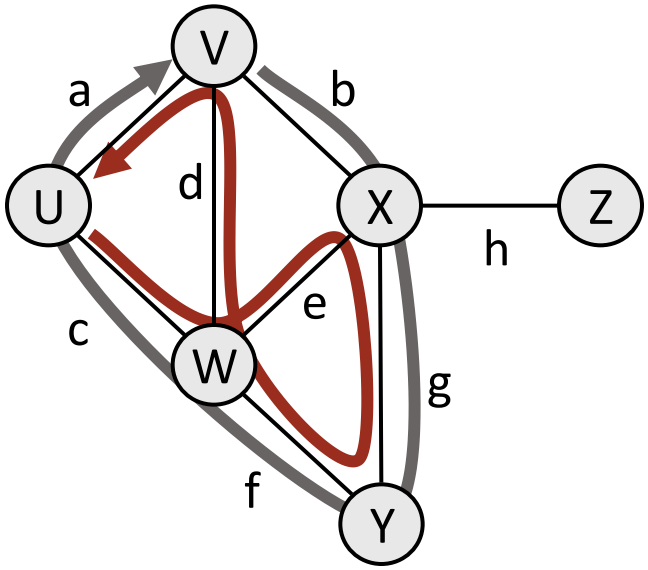


Loops and cycles

- **cycle**: A path that begins and ends at the same node.
  - example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  - example: {c, d, a} or {U, W, V, U}.

  - **acyclic graph**: One that does not contain any cycles.

- **loop**: An edge directly from a node to itself.
  - Many graphs don't allow loops.



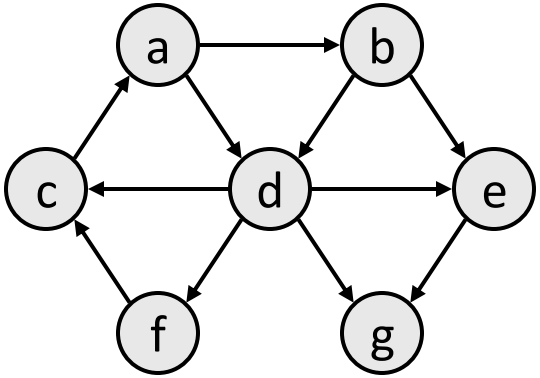
Weighted graphs

- **weight**: Cost associated with a given edge.
  - Some graphs have weighted edges, and some are unweighted.
  - Edges in an unweighted graph can be thought of as having equal weight (e.g. all 0, or all 1, etc.)
  - Most graphs do not allow negative weights.

- *example*: graph of airline flights, weighted by miles between cities:

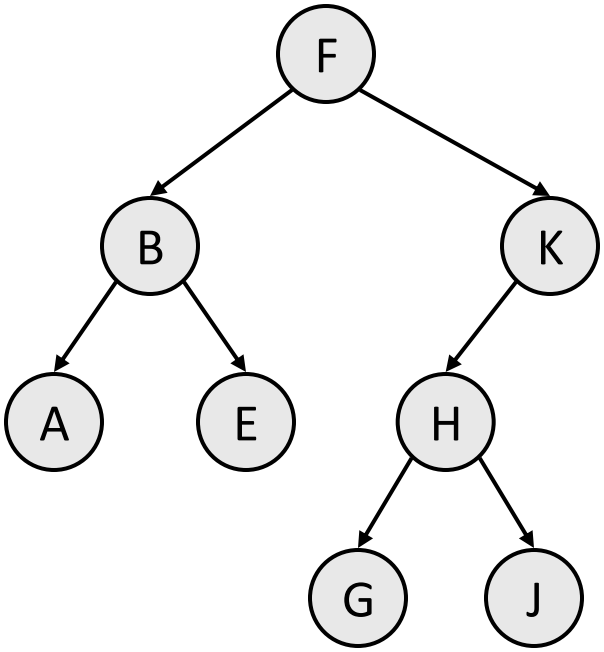
Directed graphs

- **directed graph** ("digraph"): One where edges are *one-way* connections between vertices.
  - If graph is directed, a vertex has a separate in/out degree.
  - A digraph can be weighted or unweighted.
  - Is the graph below connected? Why or why not?

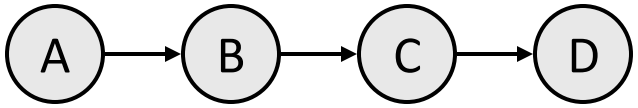


Linked Lists, Trees, Graphs

- A *binary tree* is a graph with some restrictions:
  - The tree is an unweighted, directed, acyclic graph (DAG).
  - Each node's in-degree is at most 1, and out-degree is at most 2.
  - There is exactly one path from the root to every node.



- A *linked list* is also a graph:
  - Unweighted DAG.
  - In/out degree of at most 1 for all nodes.

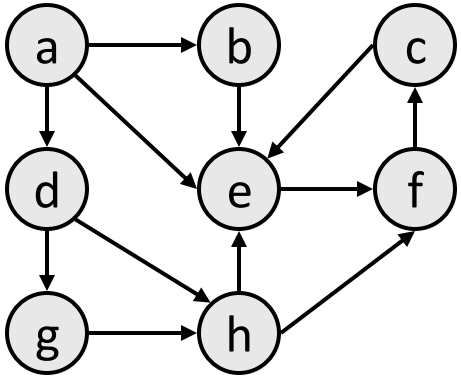


Searching for paths

- Searching for a path from one vertex to another:
  - Sometimes, we just want *any* path (or want to know there *is* a path).
  - Sometimes, we want to minimize path *length* (# of edges).
  - Sometimes, we want to minimize path *cost* (sum of edge weights).

Depth-first search

- **depth-first search** (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.
  - Often implemented recursively.
  - stack is used to implement
  - Many graph algorithms involve *visiting* or *marking* vertices.



Adjacency List

Node	Adjacent Nodes
a	b, d, e
b	e
c	e
d	g , h
e	f
f	c
g	h
h	f

DFS (Start with node a)

- Stack: Push a
- Pop a and process “a” and push all adjacent nodes of a
- Stack: b, d, e
- Pop e (as per the LIFO) and process “e”
- Push adjacent nodes of e
- Stack: b,d,f
- Pop f and process “f” and push adjacent nodes of f.

Continued…

- Stack: b,d,c
- Pop c and process “c”. Push adjacent nodes of c. But node “e” is already visited and processed. So don’t push
- Stack: b,d
- Pop d and process “d”. Push g and h.
- Stack: b, g,h
- Pop h and process “h”. But f can not be pushed.
- Pop g and process “g”. But h can not be pushed.
- Pop b and process “b”. Nothing need to push now.
- Entire Processed Nodes: a, e, f, c, d, h, g, b

Hence it can be concluded that all nodes are reachable from “a”

DFS pseudocode

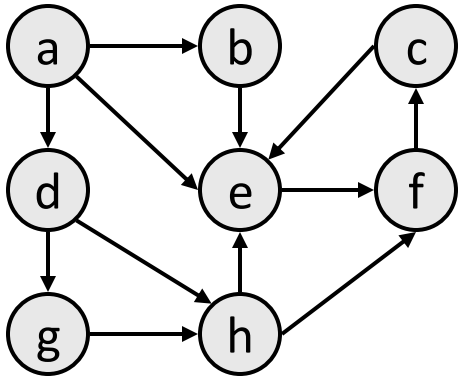
- n ← number of nodes
- Initialize visited[ ] **to false** (0)
- **for**(i=0;i<n;i++)
- visited[i] = 0;
- **void** DFS(vertex i) [DFS starting from i]
- {
- visited[i]=1;
- **for each** w adjacent **to** i
- **if**(!visited[w])
- DFS(w);
- }

DFS observations

- *discovery*: DFS is guaranteed to find *a* path if one exists.
- *retrieval*: It is easy to retrieve exactly what the path is (the sequence of edges taken) if we find it
- *optimality*: not optimal. DFS is guaranteed to find a path, not necessarily the best/shortest path

Breadth-first search

- **breadth-first search** (BFS): Finds a path between two nodes by taking one step down all paths and then immediately backtracking.
  - Often implemented by maintaining a queue of vertices to visit.
- BFS always returns the shortest path (the one with the fewest edges) between the start and the end vertices.



Adjacency List

Node	Adjacent Nodes
a	b, d, e
b	e
c	e
d	g , h
e	f
f	c
g	h
h	f

BFS Traversal (a as starting vertex & f as ending vertex)

- Insert a into the queue.
- Queue: a
- Origin: ϕ
- Delete a from queue and insert adjacent nodes.
- Queue: b, d, e
- Origin: a, a, a
- Delete b(as per FIFO) and insert adjacent nodes.
- Queue: d, e [e is already visited so not pushed]
- Origin: a , a

Continued..

- Delete d and insert adjacent nodes
- Queue: e, g, h
- Origin: a, d, d
- Delete e and insert adjacent nodes
- Queue: g, h, f [as we have reached f, we stop]
- Origin: d, d, e.
- Lets backtrack for the path f<-e<-a (Recursively (node<-origin))

BFS pseudocode(For all node Traversal)

```
function bfs(v1, v2):
    queue := {v1}.
    mark v1 as visited.

    while queue is not empty:
        v := queue.removeFirst().
        if v is v2:
            a path is found!

        for each unvisited neighbor n of v:
                mark n as visited.
            queue.addLast(n).

    // path is not found.
```

Sample code for BFS function

```
void BFS(int v)
{  int i;
insert_queue(v);
state[v] = waiting;
while(!isEmpty_queue())
{
v = delete_queue( );
printf("%d ",v);
state[v] = visited;
for(i=0; i<n; i++)
{
if(adj[v][i] == 1 && state[i] == initial)
{
insert_queue(i);
state[i] = waiting;
}
}
}
}
```

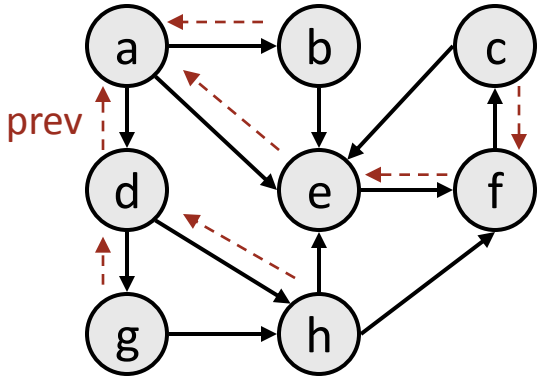
BFS that finds path(For all node Traversal)

function **bfs**(v_1, v_2):
    queue := $\{v_1\}$.
    mark v_1 as visited.

    while *queue* is not empty:
        v := *queue*.removeFirst().
        if v is v_2:
            a path is found! *(reconstruct it by following .prev back to v_1.)*

        for each unvisited neighbor n of v:
                mark n as visited. *(set n.prev = v.)*
            queue.addLast(n).

    // path is not found.



- By storing some kind of "previous" reference associated with each vertex, you can reconstruct your path back once you find v_2.

BFS observations

- *optimality*:
  - always finds the shortest path (fewest edges).
  - in unweighted graphs, finds optimal cost path.
  - In weighted graphs, *not* always optimal cost.

- *retrieval*: harder to reconstruct the actual sequence of vertices or edges in the path once you find it
  - conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a path array/list in progress
  - solution: We can keep track of the path by storing predecessors for each vertex (each vertex can store a reference to a *previous* vertex).

- DFS uses less memory than BFS, easier to reconstruct the path once found; but DFS does not always find shortest path. BFS does.

DFS, BFS runtime

- What is the expected runtime of DFS and BFS, in terms of the number of vertices V and the number of edges E ?

- Answer: O(|V| + |E|)
  - where |V| = number of vertices, |E| = number of edges
  - Must potentially visit every node and/or examine every edge once.
- What is the space complexity of each algorithm?
  - (How much memory does each algorithm require?)

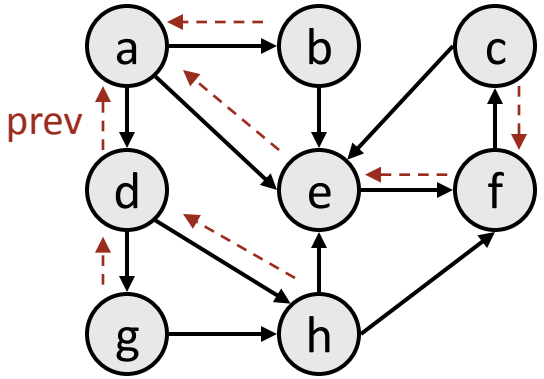
BFS that finds path

function **bfs**(v_1, v_2):
    queue := $\{v_1\}$.
    mark v_1 as visited.

    while *queue* is not empty:
        v := *queue*.removeFirst().
        if v is v_2:
            a path is found! *(reconstruct it by following .prev back to v_1.)*

        for each unvisited neighbor n of v:
                mark n as visited. *(set n.prev = v.)*
            queue.addLast(n).

    // path is not found.



- By storing some kind of "previous" reference associated with each vertex, you can reconstruct your path back once you find v_2.

REST OF THE TOPICS

- Minimal spanning trees: Prim's Algorithm, Kruskal's Algorithm: **Follow Greedy Algorithms lecture**
- Shortest path algorithms: Dijkstra's Algorithm: **Follow Greedy Algorithms lecture**
- Shortest path algorithms: Floyd's Algorithm, Floyd-Warshall Algorithm: **Follow Dynamic Programming lecture**