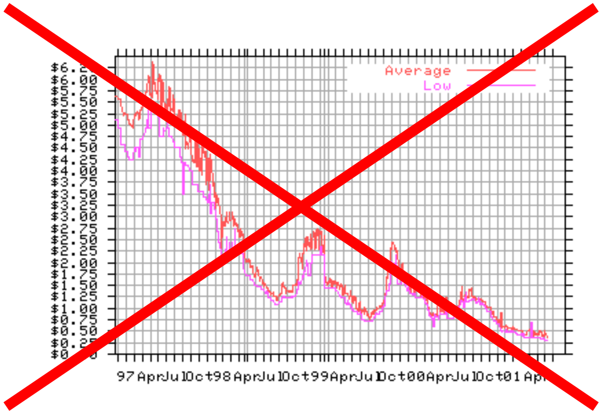
CSE 373

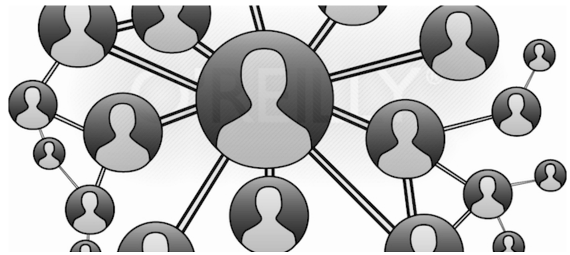
Graphs 1: Concepts, Depth/Breadth-First Search

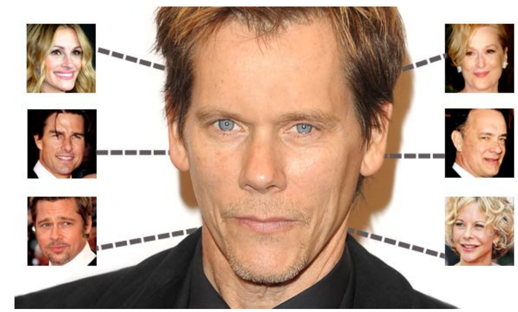
reading: Weiss Ch. 9

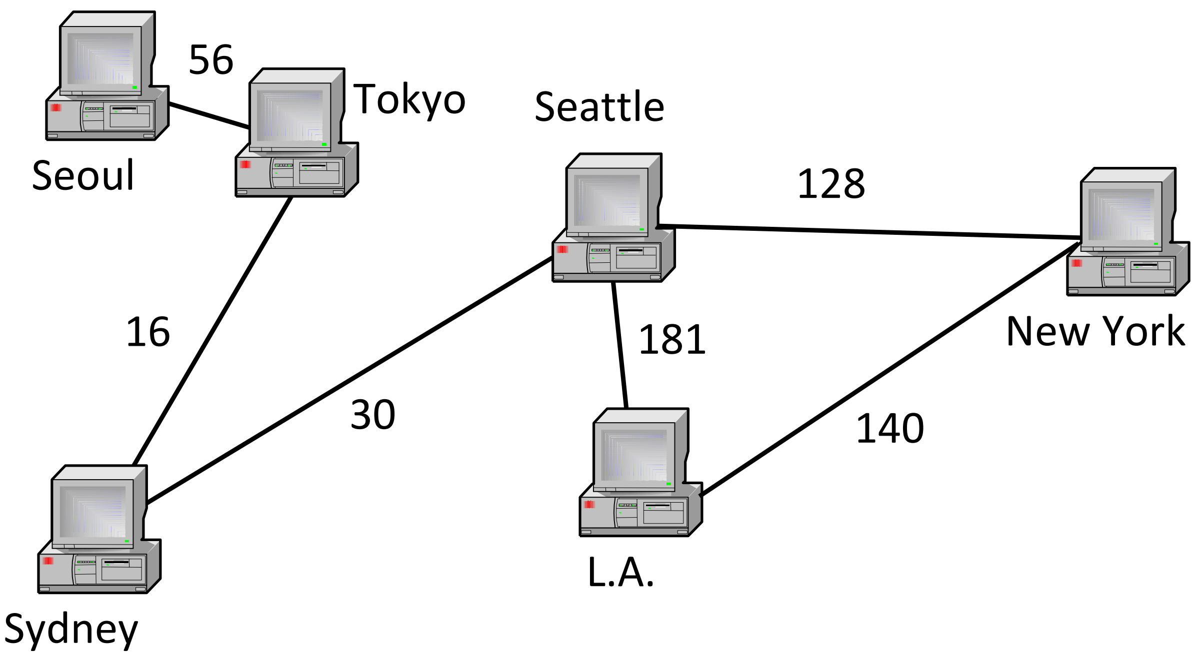
slides created by Marty Stepp
http://www.cs.washington.edu/373/

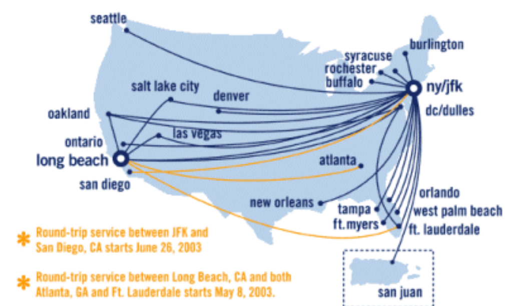
What is a graph?





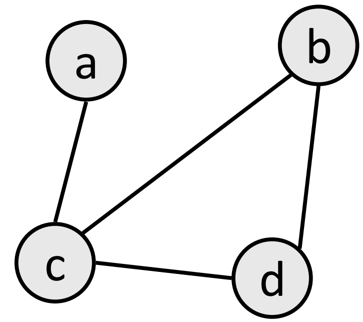






Graphs

- **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

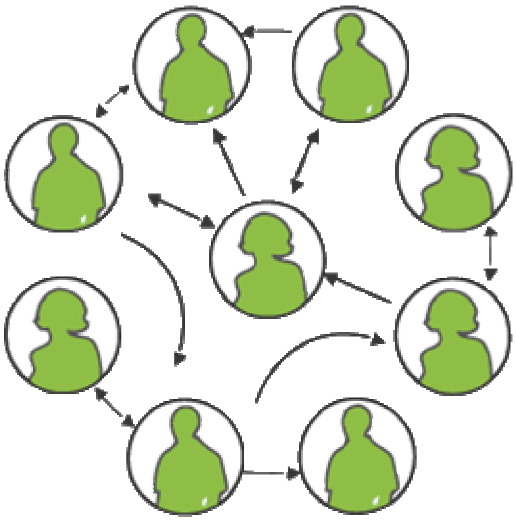


- the graph at right:
  - $V = \{a, b, c, d\}$
  - $E = \{(a, c), (b, c), (b, d), (c, d)\}$

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

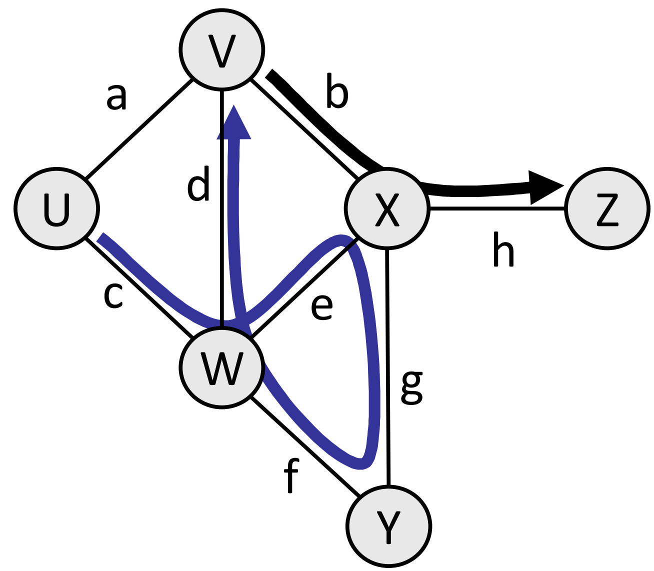
Graph examples

- For each, what are the vertices and what are the edges?
  - Web pages with links
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Facebook friends
  - Course pre-requisites
  - Family trees
  - Paths through a maze



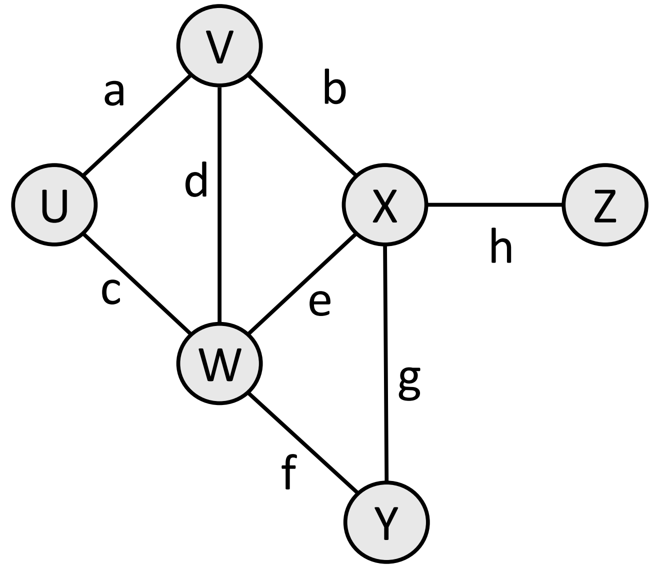
Paths

- **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



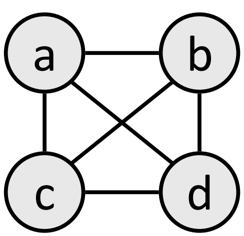
Reachability, connectedness

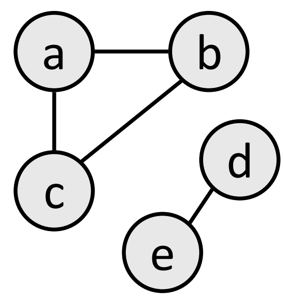
- **reachable**: Vertex *a* is *reachable* from *b* if a path exists from *a* to *b*.



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

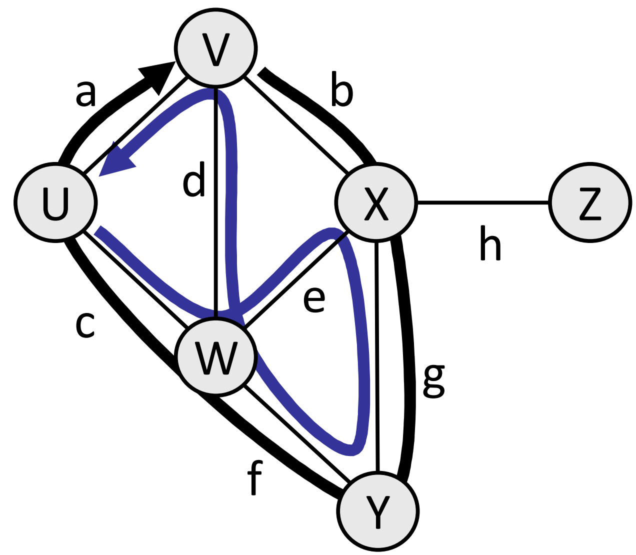
- **strongly connected**: When every vertex has an edge to every other vertex.





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.

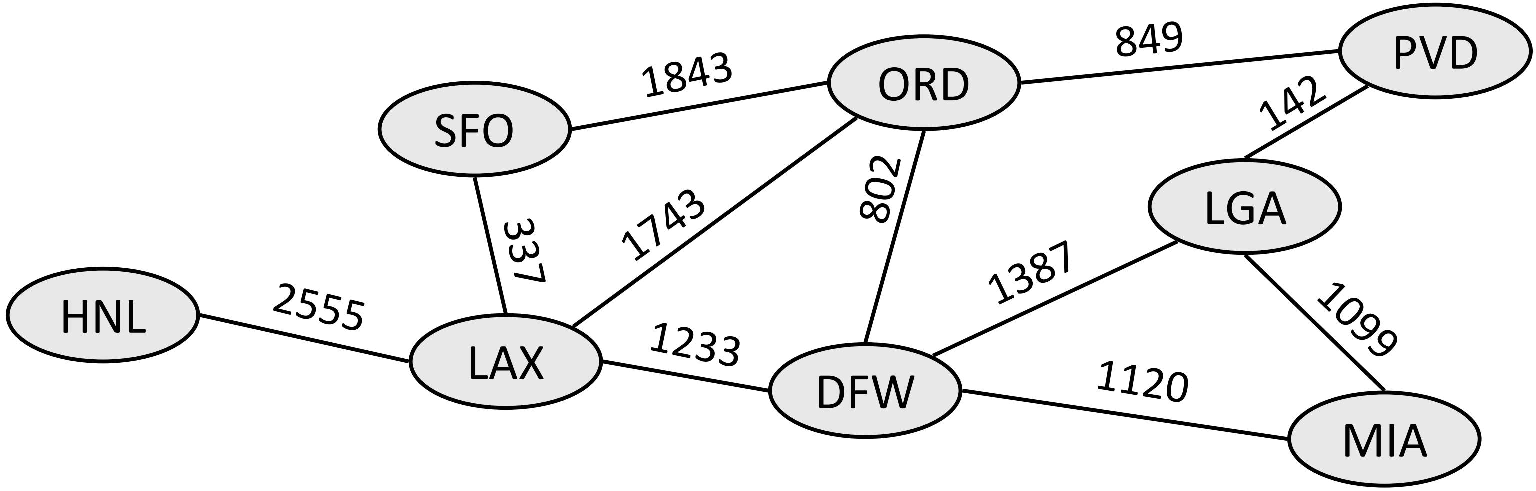


- **loop**: An edge directly from a node to itself.
  - Many graphs don't allow loops.

Weighted graphs

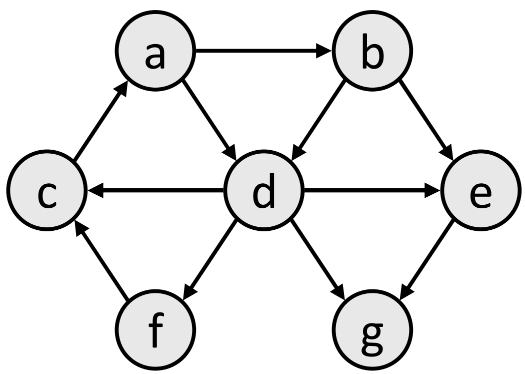
- **weight**: Cost associated with a given edge.
  - Some graphs have weighted edges, and some are unweighted.
  - Edges in an unweighted graph can be thought of as having equal weight (e.g. all 0, or all 1, etc.)
  - Most graphs do not allow negative weights.

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



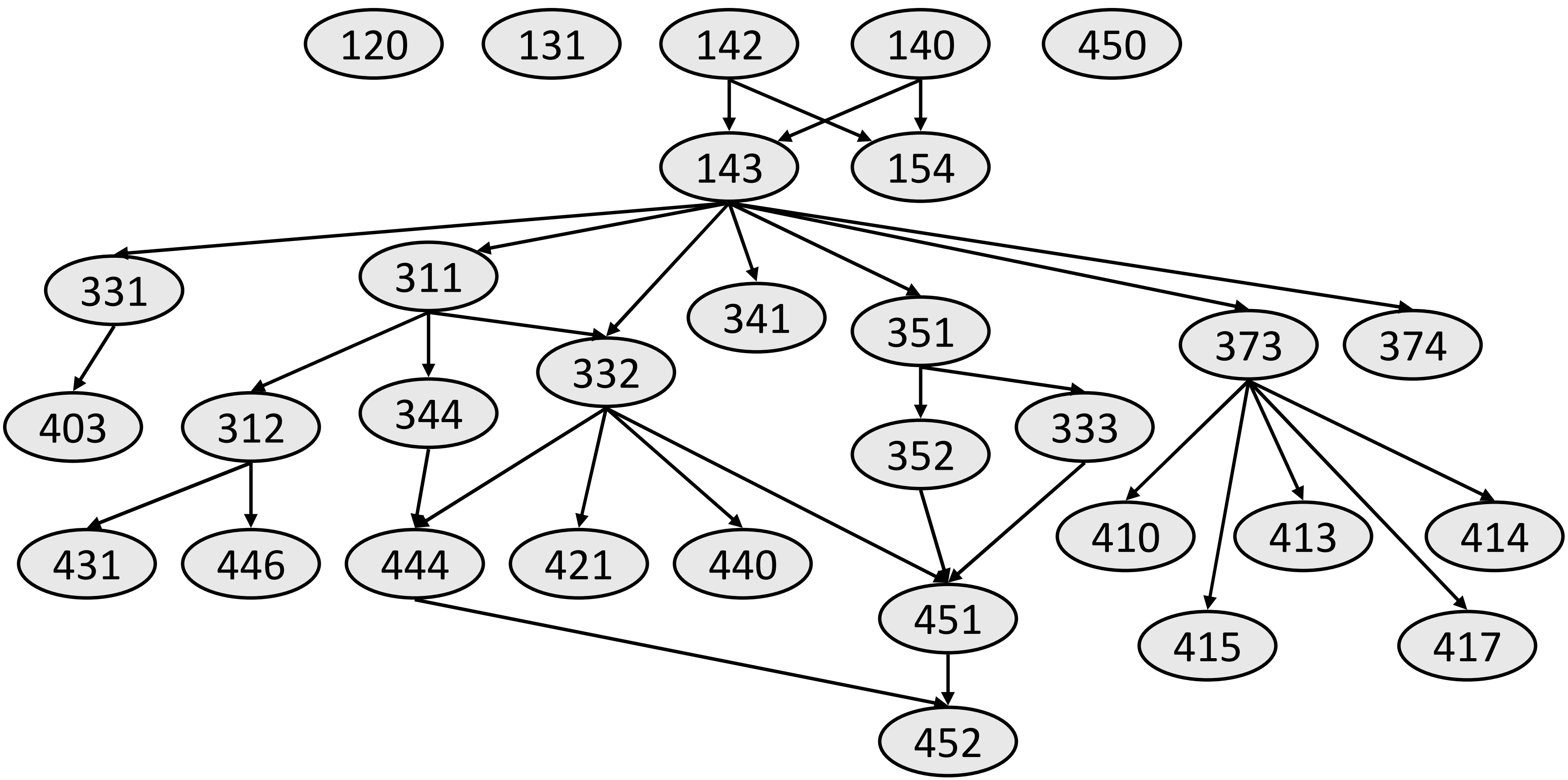
Directed graphs

- **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?



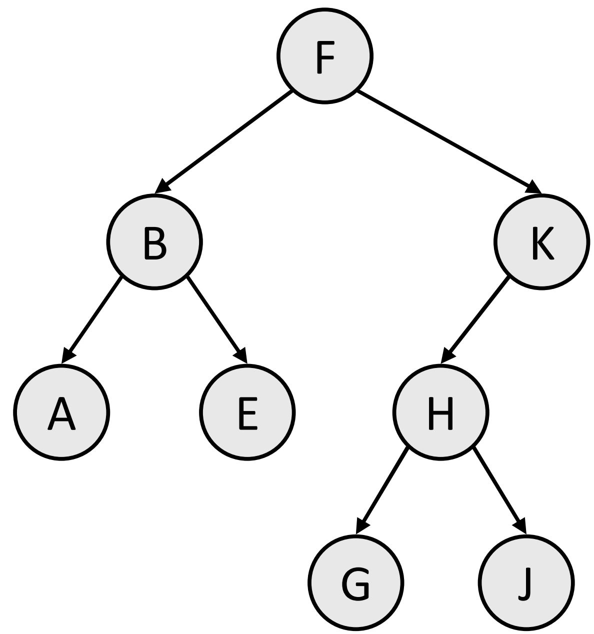
Digraph example

- Vertices = UW CSE courses (incomplete list)
- Edge (a, b) = *a* is a prerequisite for *b*

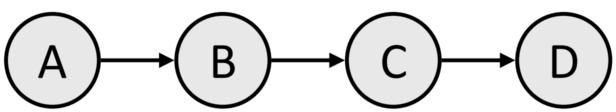


Linked Lists, Trees, Graphs

- 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.



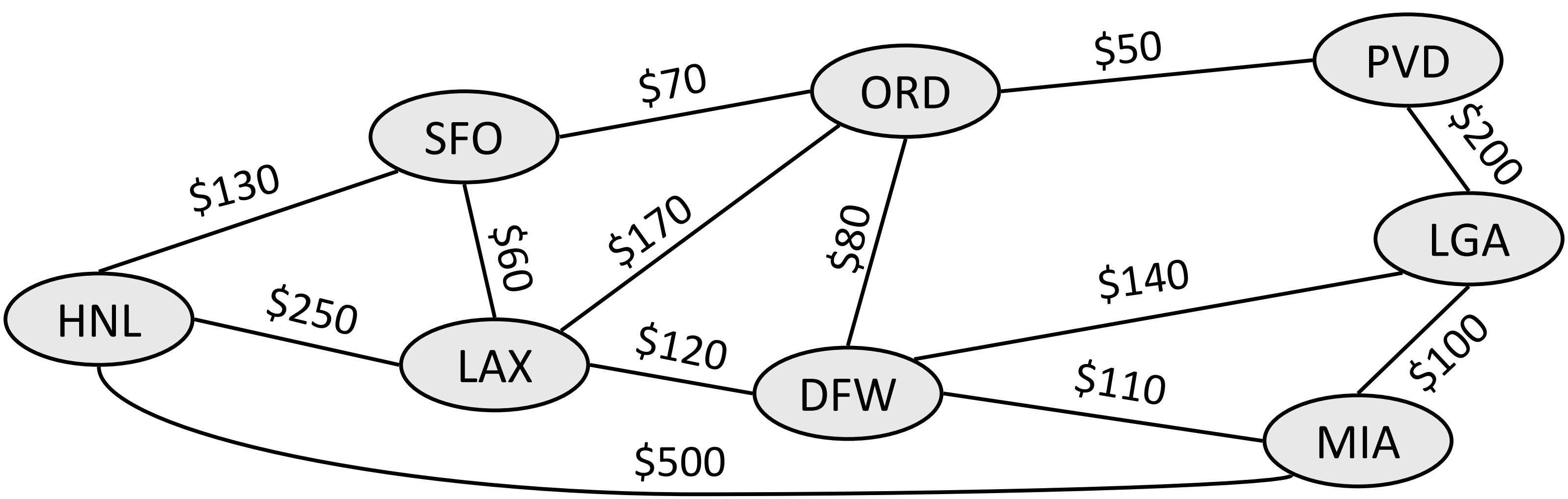
- A *linked list* is also a graph:
  - Unweighted DAG.
  - In/out degree of at most 1 for all nodes.



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).

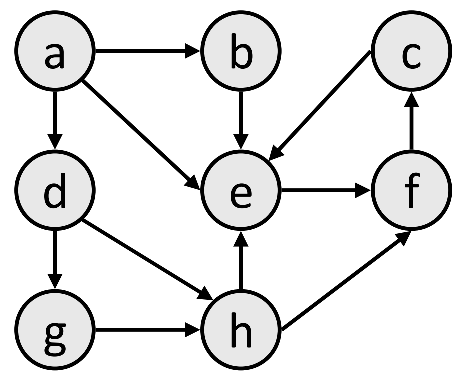
- What is the shortest path from MIA to SFO?
  Which path has the minimum cost?



Depth-first search

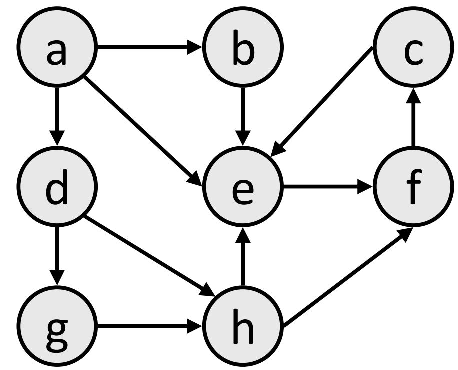
- **depth-first search** (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.
  - Often implemented recursively.
  - Many graph algorithms involve *visiting* or *marking* vertices.

- Depth-first paths from *a* to all vertices (assuming ABC edge order):
  - to b: {a, b}
  - to c: {a, b, e, f, c}
  - to d: {a, d}
  - to e: {a, b, e}
  - to f: {a, b, e, f}
  - to g: {a, d, g}
  - to h: {a, d, g, h}



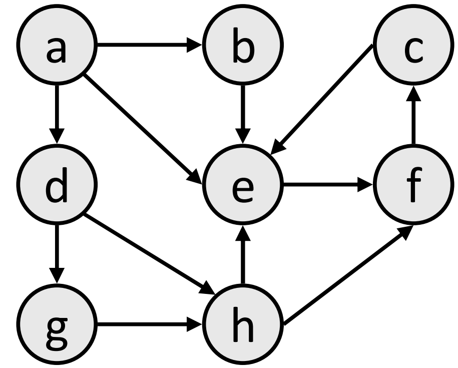
DFS pseudocode

```
function dfs(v1, v2):
   dfs(v1, v2, { }).

function dfs(v1, v2, path):
   path += v1.
   mark v1 as visited.
   if v1 is v2:
      a path is found!

   for each unvisited neighbor n of v1:
      if dfs(n, v2, path) finds a path: a path is found!

   path -= v1.   // path is not found.
```



- The *path* param above is used if you want to have the path available as a list once you are done.
  - Trace dfs(*a*, *f*) in the above graph.

DFS observations

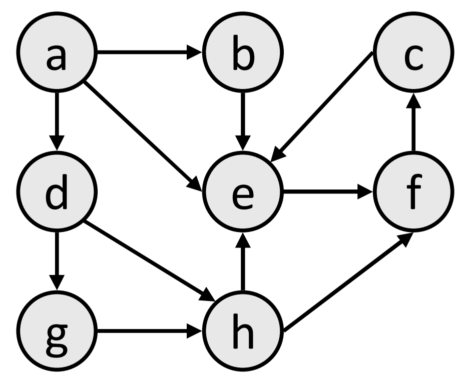


- *discovery*: DFS is guaranteed to find *a* path if one exists.
- *retrieval*: It is easy to retrieve exactly what the path is (the sequence of edges taken) if we find it
- *optimality*: not optimal. DFS is guaranteed to find a path, not necessarily the best/shortest path
  - Example: dfs(a, f) returns {a, d, c, f} rather than {a, d, f}.

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.
  - to b: {a, b}
  - to c: **{a, e, f, c}**
  - to d: {a, d}
  - to e: **{a, e}**
  - to f: **{a, e, f}**
  - to g: {a, d, g}
  - to h: **{a, d, h}**



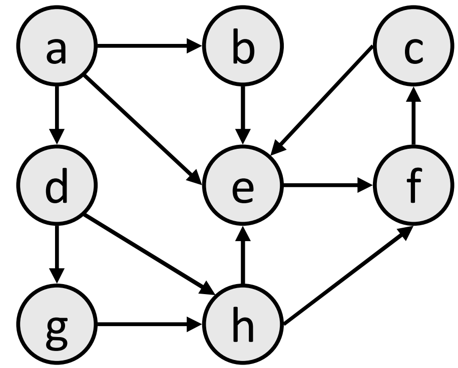
BFS pseudocode

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!

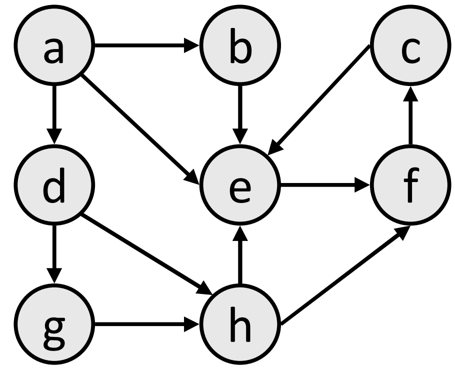
    for each unvisited neighbor n of v:
      mark n as visited.
      queue.addLast(n).

  // path is not found.



- Trace bfs(a, f) in the above graph.

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

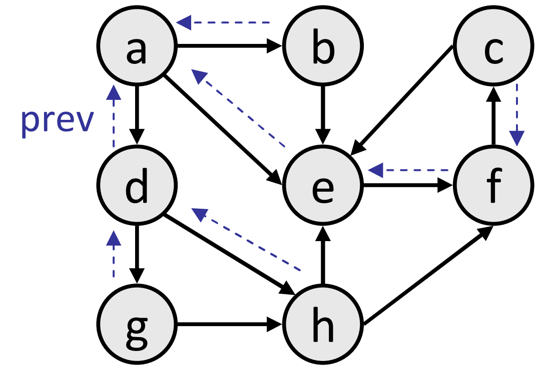
- What is the expected runtime of DFS and BFS, in terms of the number of vertices V and the number of edges E ?

- Answer: O(|V| + |E|)
  - where |V| = number of vertices, |E| = number of edges
  - Must potentially visit every node and/or examine every edge once.

  - why not O(|V| * |E|) ?

- 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.