CSE 373

Graphs 1: Concepts,
Depth/Breadth-First Search

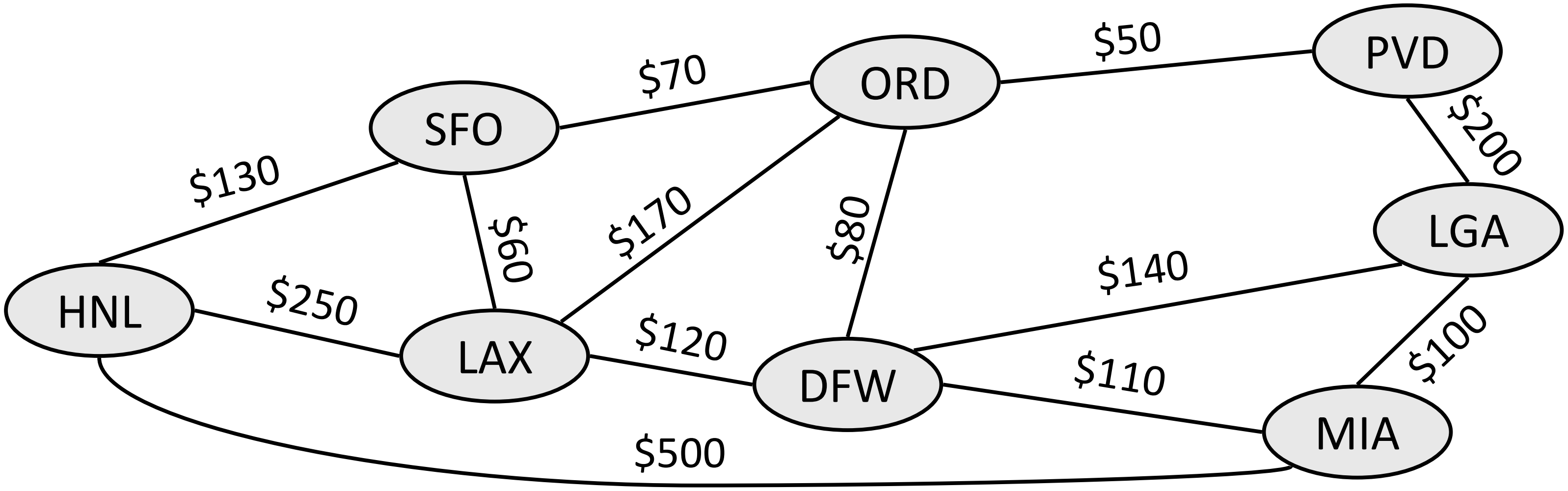
reading: Weiss Ch. 9

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

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).

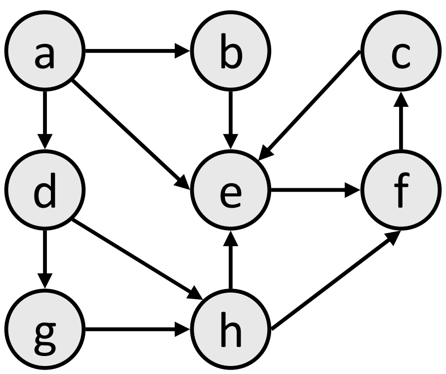
- What is the shortest path from MIA to SFO?
  Which path has the minimum cost?



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}



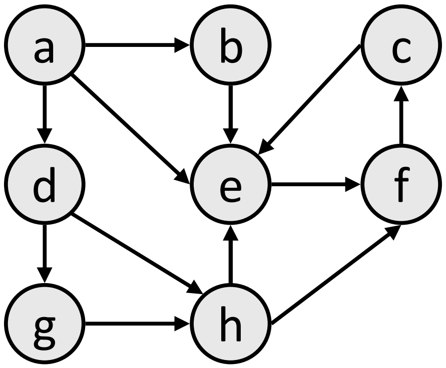
DFS pseudocode

function **dfs**(v_1, v_2):
  dfs(v_1, v_2, { }).

function **dfs**(v_1, v_2, *path*):
  path += v_1.
  mark v_1 as visited.
  if v_1 is v_2:
    a path is found!

  for each unvisited neighbor *n* of v_1:
    if dfs(*n*, v_2, *path*) finds a path: a path is found!

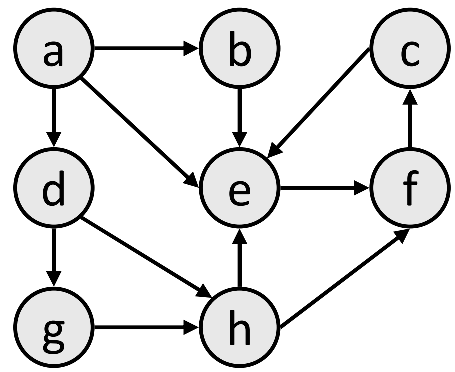
  path -= v_1.   // path is not found.



- 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.



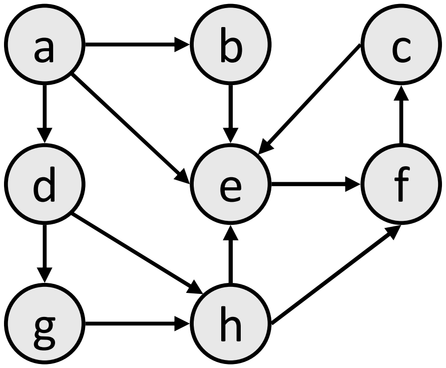
- *retrieval*: It is easy to retrieve exactly what the path is (the sequence of edges taken) if we find it

- *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}**



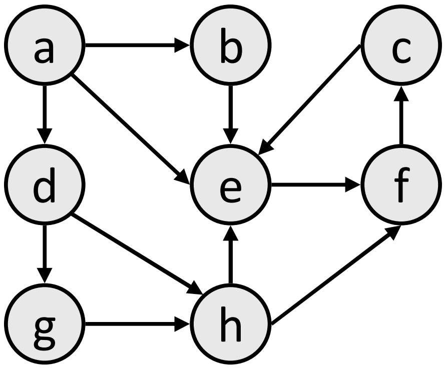
BFS pseudocode

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).

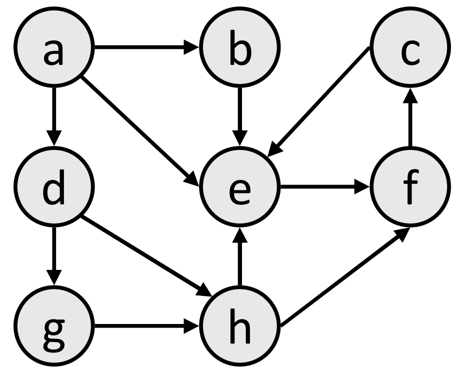
    // path is not found.



- 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?)

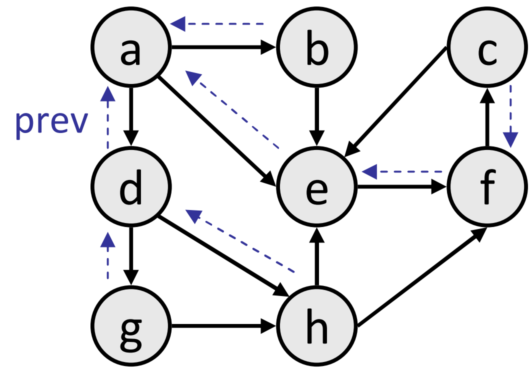
BFS that finds path

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.