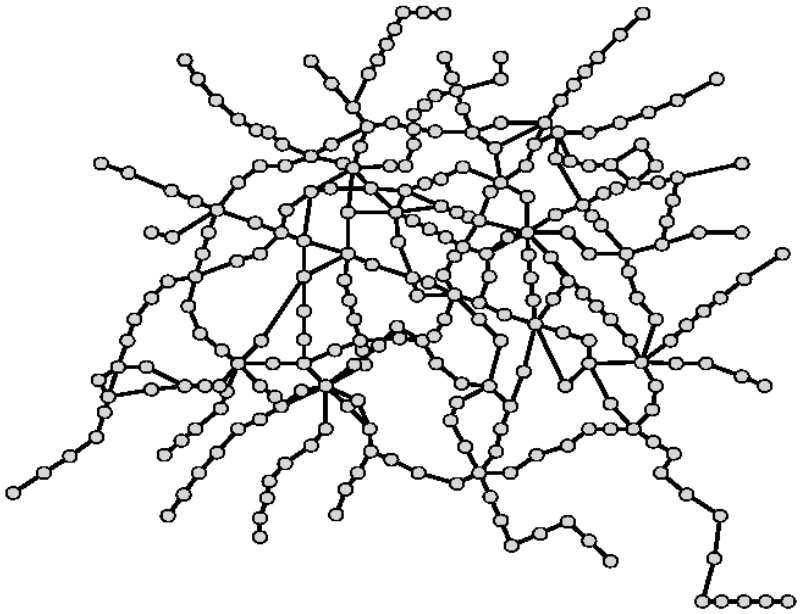
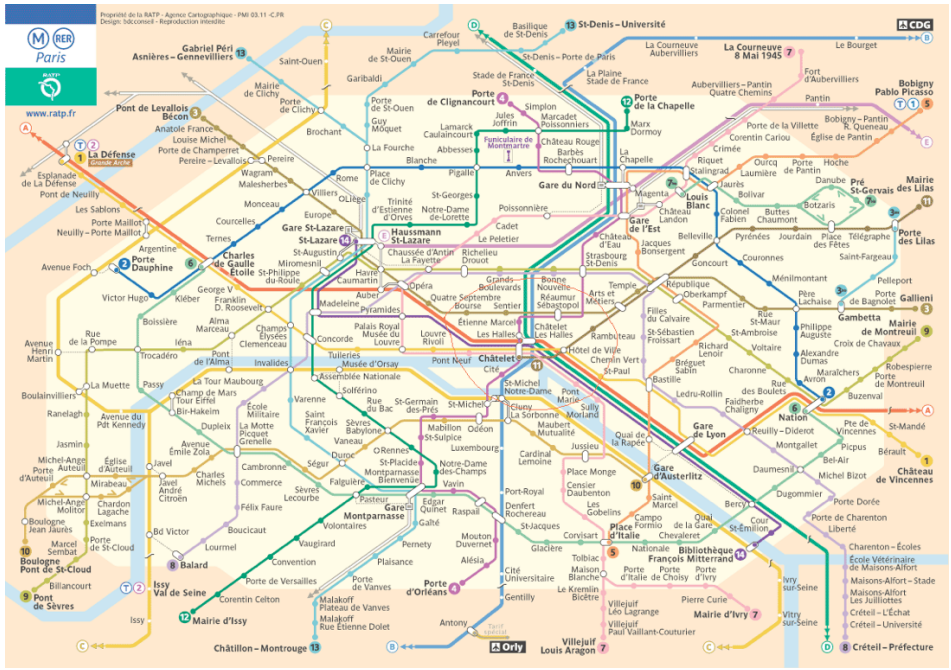
Intro to Graph Concepts & Algorithms

Ye Yang
Stevens Institute of Technology

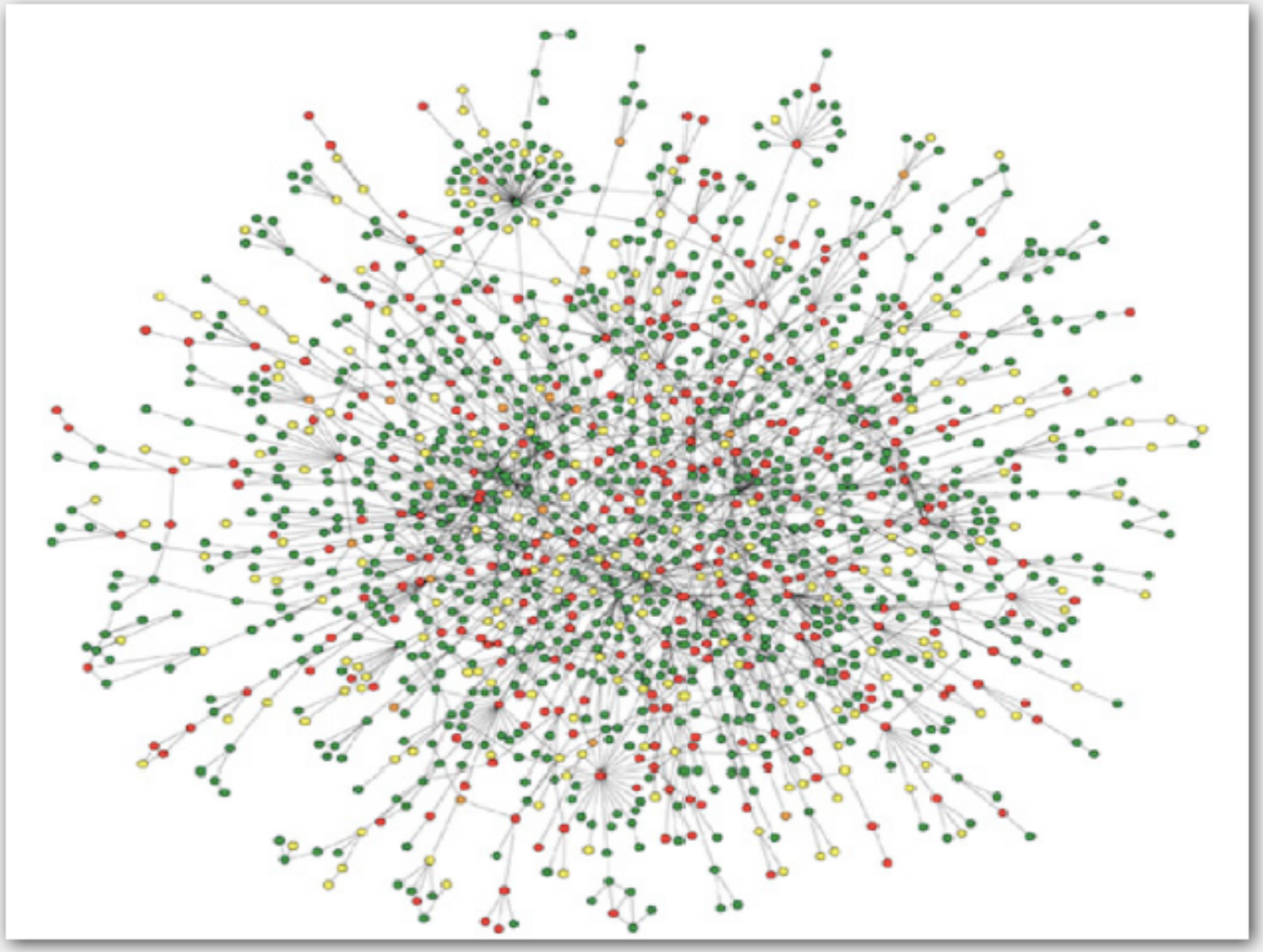
Graphs

- Set of vertices connected pairwise by edges.
- Why study graph algorithms?
  - Interesting and broadly useful abstraction.
  - Challenging branch of computer science and discrete math.
  - Hundreds of graph algorithms known.
  - Thousands of practical applications.





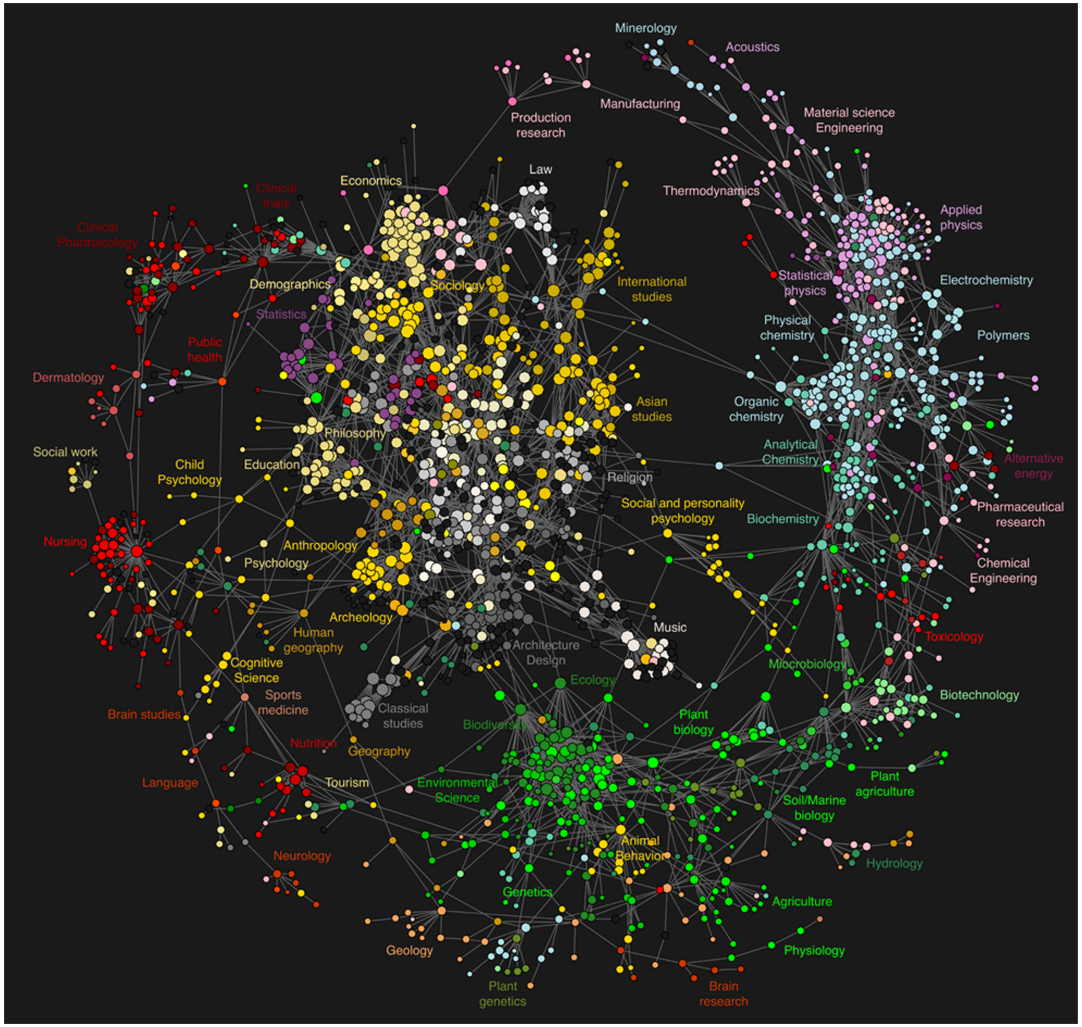
Protein-protein interaction network



Reference: Jeong et al, Nature Review | Genetics

Map of science clickstreams

---- http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803



10 million Facebook friends

"Visualizing Friendships" by Paul Butler

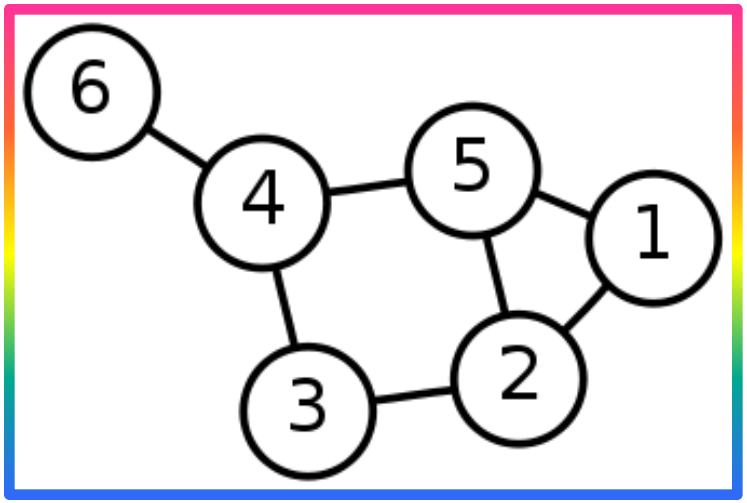
Graph applications

graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
chemical compound	molecule	bond

Graph Terminology

Undirected Graphs

- A Graph G, consists of (undiredted, unweighted):
  - a set of vertices, V
  - a set of edges, E
  - where each edge is associated with a pair of vertices.
- We write: G = (V, E)



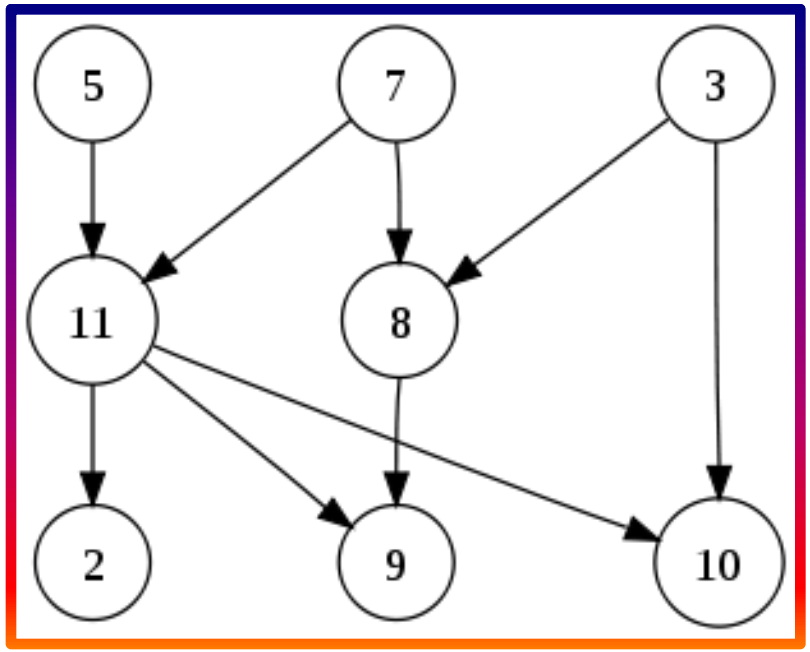
A labeled simple graph:

Vertex set $V = \{1, 2, 3, 4, 5, 6\}$

Edge set $E = \{\{1,2\}, \{1,5\}, \{2,3\}, \{2,5\}, \{3,4\}, \{4,5\}, \{4,6\}\}$.

Directed Graphs

- Same as above, but where each edge is associated with an *ordered* pair of vertices.



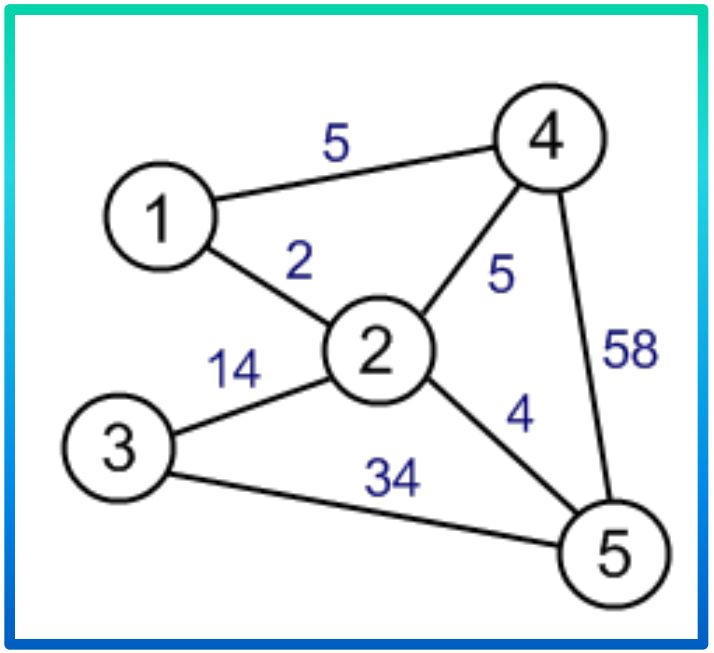
A labeled simple graph:

Vertex set $V = \{2,3,5,7,8,9,10,11\}$

Edge set $E = \{\{3,8\}, \{3,10\}, \{5,11\}, \{7,8\}, \{7,11\}, \{8,9\}, \{11,2\},\{11,9\},\{11,10\}\}$.

Weighted Graphs

- Same as above, but where each edge *also* has an associated real number with it, known as the edge **weight**.



A labeled weighted graph:

Vertex set $V = \{1,2,3,4,5\}$

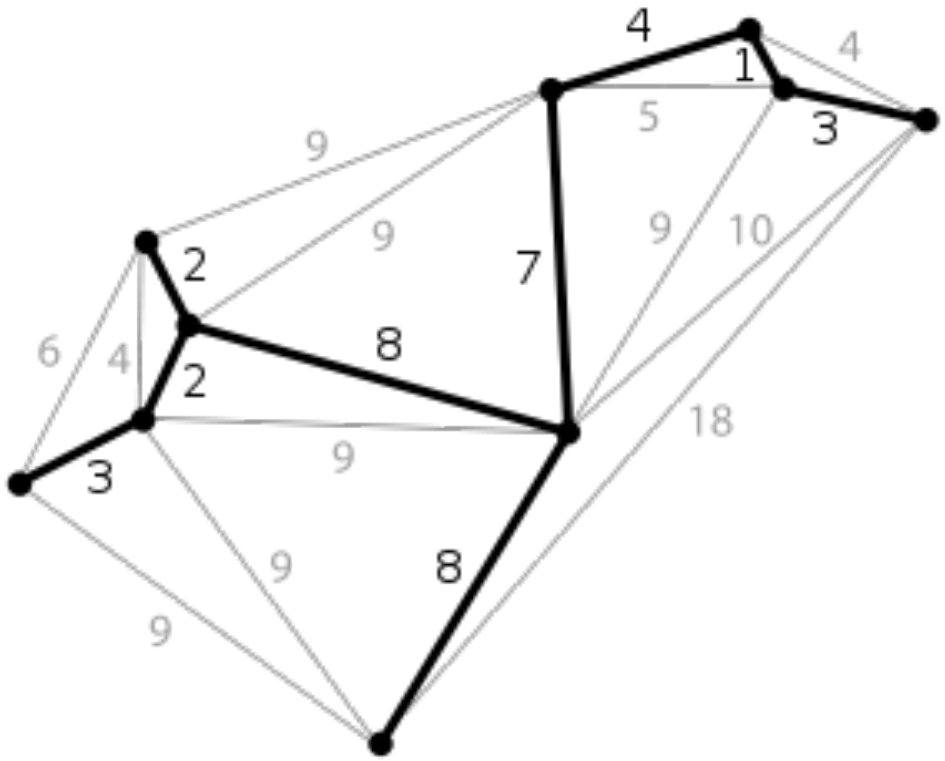
Graph Density

- Graph density is a measure of the number of edges
  - A *sparse* graph has $O(|V|)$ edges
  - A *dense* graph has $\Theta(|V|^2)$ edges

- Anything in between is either *sparsish* or *densy* depending on the context.

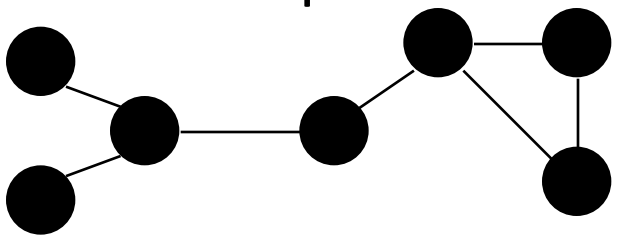
Connected Graph

- A connected graph is one where any pair of vertices in the graph is connected by at least one path.

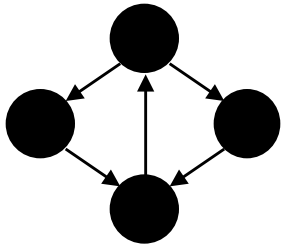


Connectivity

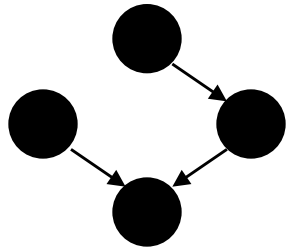
Undirected graphs are *connected* if there is a path between any two vertices



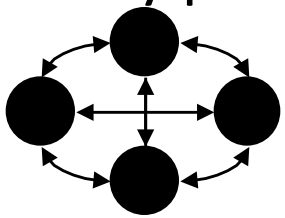
Directed graphs are *strongly connected* if there is a path from any one vertex to any other



Directed graphs are *weakly connected* if there is a path between any two vertices, *ignoring direction*



A *complete* graph has an edge between every pair of vertices



Graph Representation

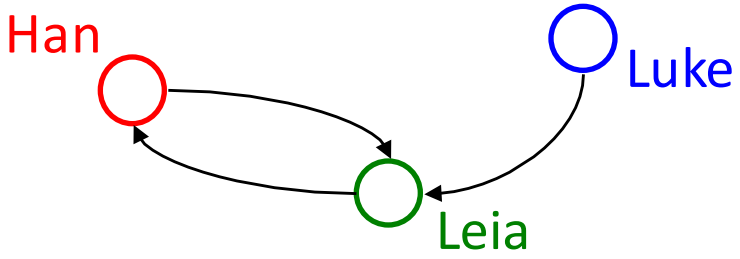
How do we represent graph internally?

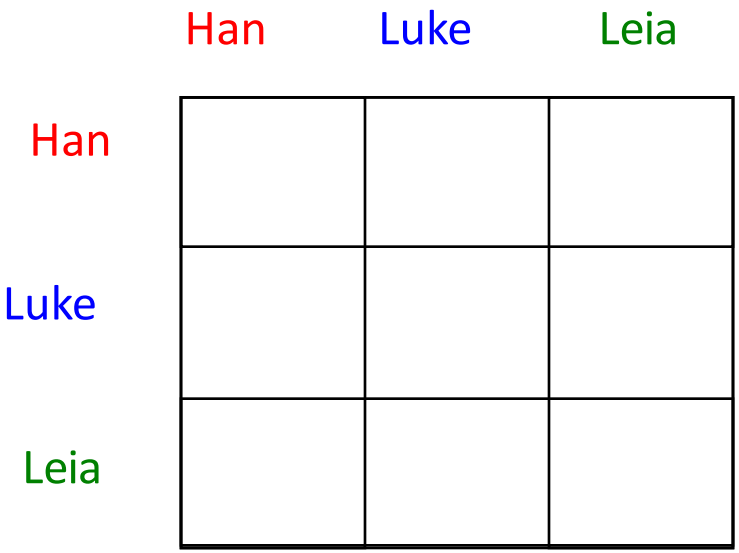
Two Representations of Graphs

1. Adjacency Matrix A:
   - Use matrix entries to represent edges in the graph
   - A[i][j]=1 if (i,j) is an edge; 0 otherwise
   - Using quadratic space
     - Space cost, $O(|V|^2)$
2. Adjacency Lists:
   - Use an array of lists to represent edges in the graph
   - L[i] is the linked list containing all neighbors of node i
   - Using linear space
     - Space cost, $O(|E|)$

Representation 1: Adjacency Matrix

A `|V| x |V|` array in which an element `(u, v)` is true if and only if there is an edge from `u` to `v`





Runtime:

iterate over vertices

iterate ever edges

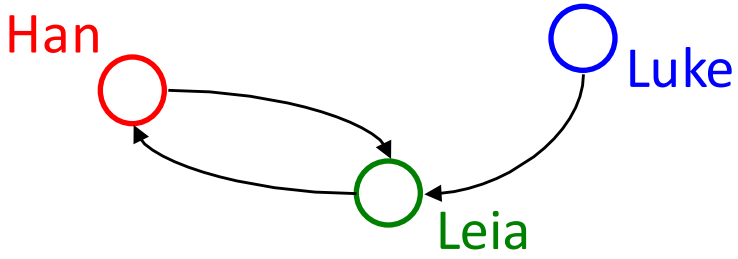
iterate edges adj. to vertex

edge exists?

Space requirements:

Representation 2: Adjacency Lists

A $|V|$-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices



Runtime:

iterate over vertices

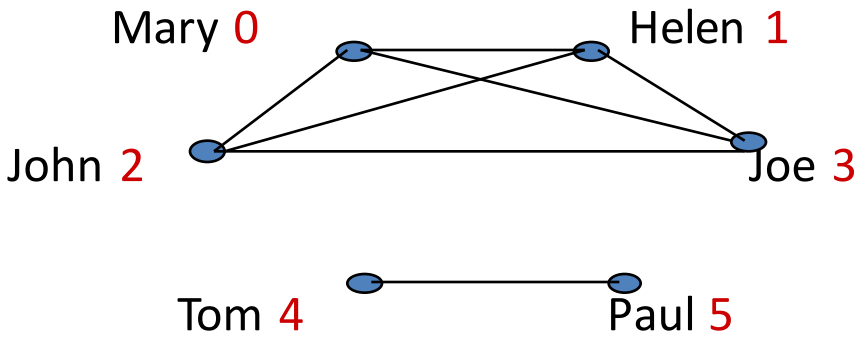
iterate ever edges

iterate edges adj. to vertex

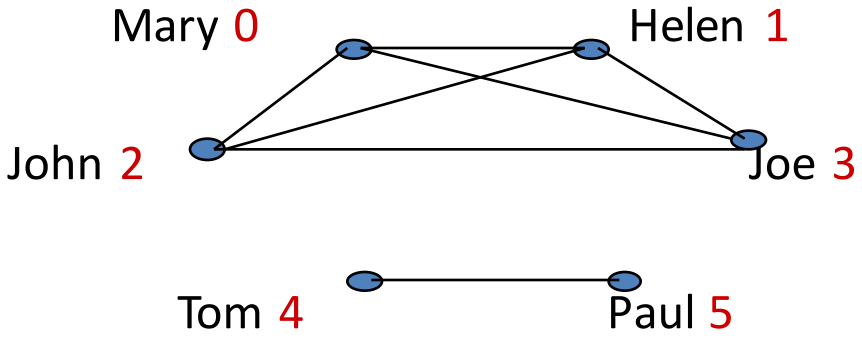
edge exists?

space requirements:

Exercise: Graph Representations



Exercise Solution: Graph Representations



Adjacency Matrix:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Linked Lists:

L[0]: 1, 2, 3
L[1]: 0, 2, 3
L[2]: 0, 1, 3
L[3]: 0, 1, 2
L[4]: 5
L[5]: 4

Some graph-processing problems

- Path. Is there a path between s and t?
- Shortest path. What is the shortest path between s and t?

- Cycle. Is there a cycle in the graph?
- Euler tour. Is there a cycle that uses each edge exactly once?
- Hamilton tour. Is there a cycle that uses each vertex exactly once?

- Connectivity. Is there a way to connect all of the vertices?
- MST. What is the best way to connect all of the vertices?
- Biconnectivity. Is there a vertex whose removal disconnects the graph?

- Planarity. Can you draw the graph in the plane with no crossing edges?
- Graph isomorphism. Do two adjacency lists represent the same graph?

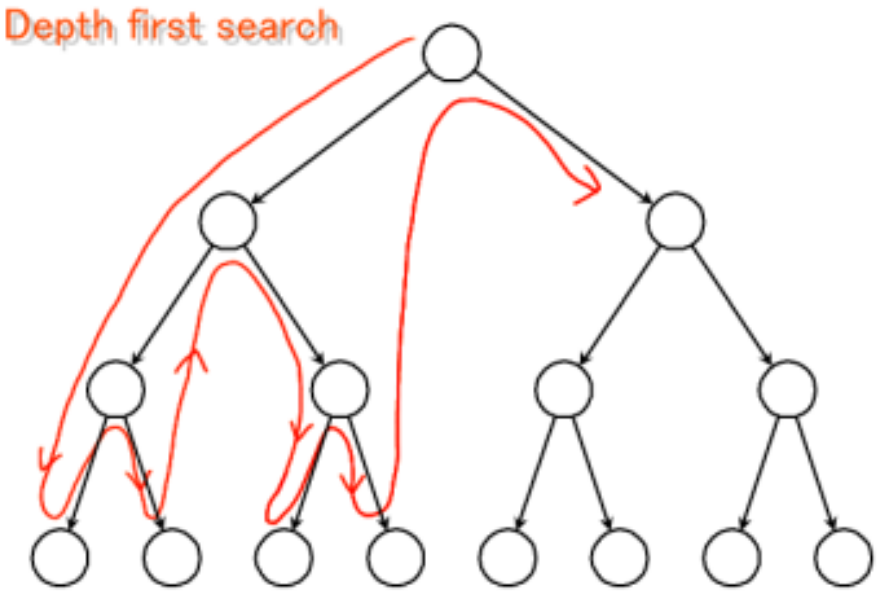
- Challenge. Which of these problems are easy? difficult? intractable?

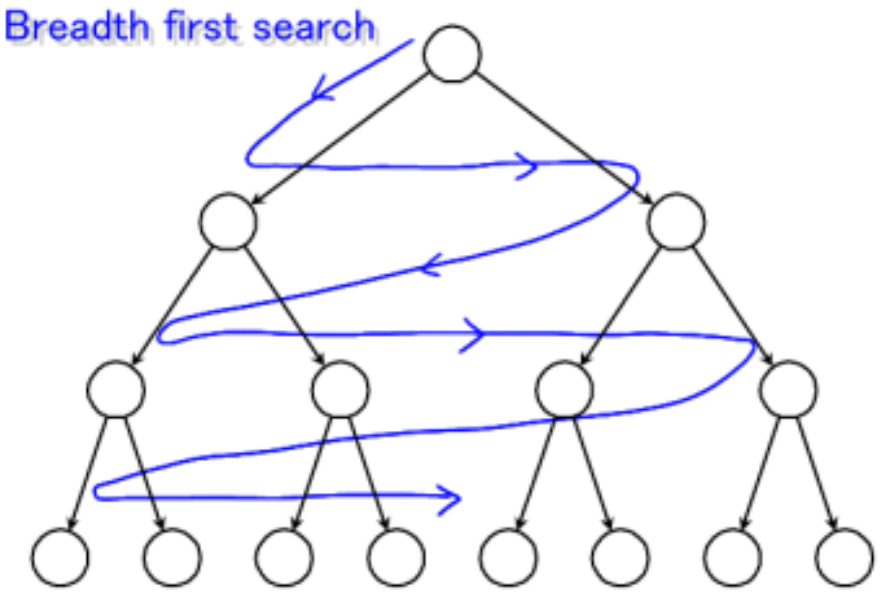
Graph Traversals

Searching a graph

- With certain modifications, any tree search technique can be applied to a graph
  - This includes depth-first, breadth-first, and other types of searches
- The difference is that a graph may have cycles
  - We don't want to search around and around in a cycle
- To avoid getting caught in a cycle, we must keep track of which nodes we have already explored
- There are two basic techniques for this:
  - Keep a set of already explored nodes, or
  - Mark the node itself as having been explored
    - Marking nodes is not always possible (may not be allowed)

Graph Traversals





- Both take time: O(V+E)

Depth-First-Search (DFS)

- What is the idea behind DFS?
  - Travel as far as you can down a path
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a

stack

Ideas: Depth-first search

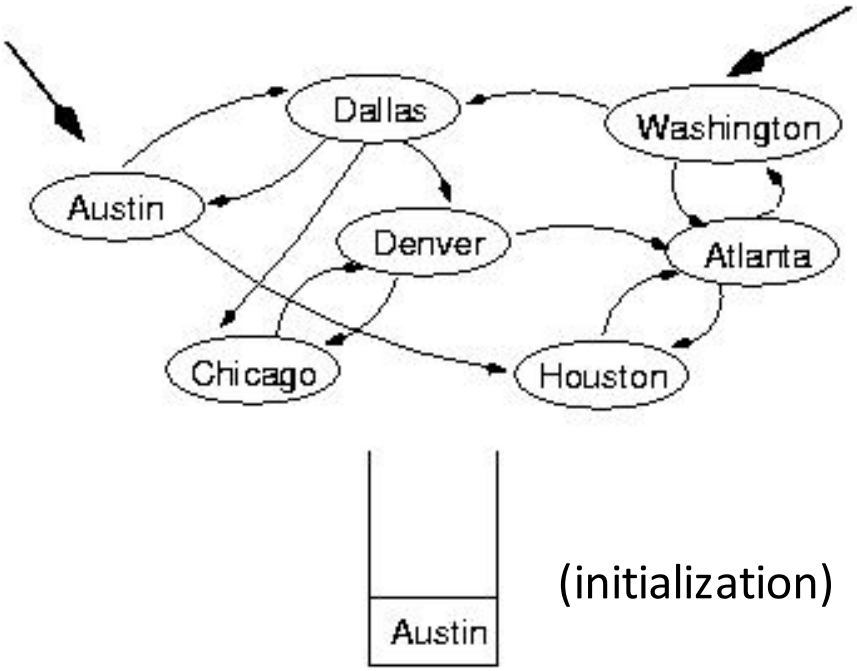
- **Here is how to do DFS on a tree:**

```
Put the root node on a stack;
while (stack is not empty) {
    remove a node from the stack;
    if (node is a goal node) return success;
    put all children of the node onto the stack;
}
return failure;
```

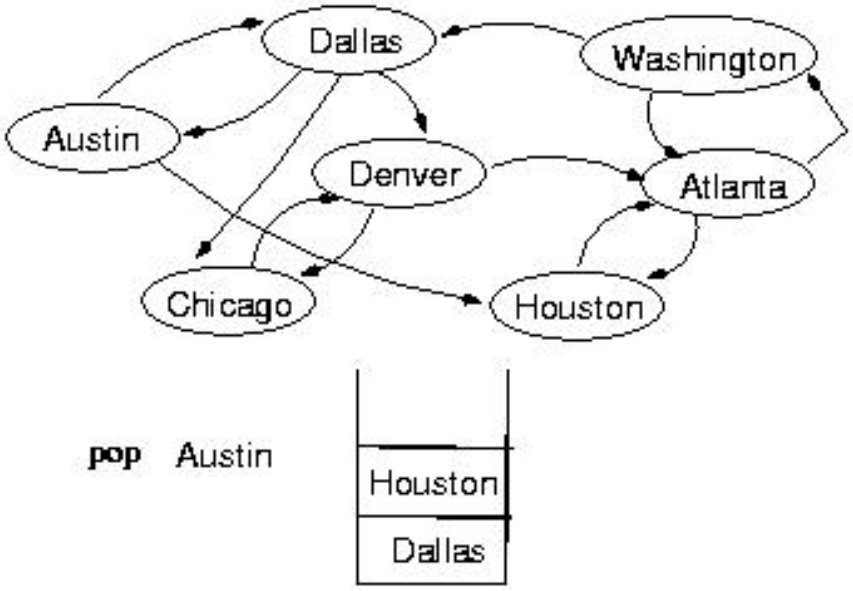
- **Here is how to do DFS on a graph:**

```
Put the starting node on a stack;
while (stack is not empty) {
    remove a node from the stack;
    if (node has already been visited) continue;
    if (node is a goal node) return success;
    put all adjacent nodes of the node onto the stack;
}
return failure;
```

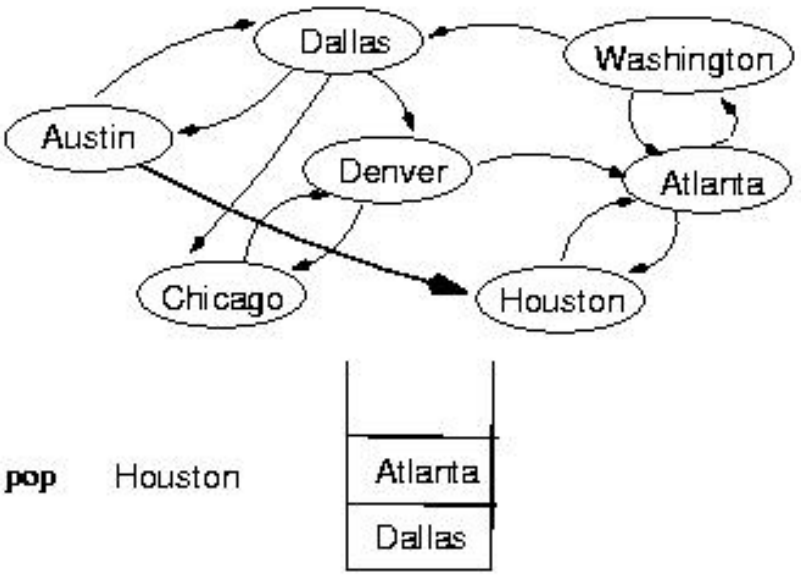
Example: Depth-First-Search (DFS)

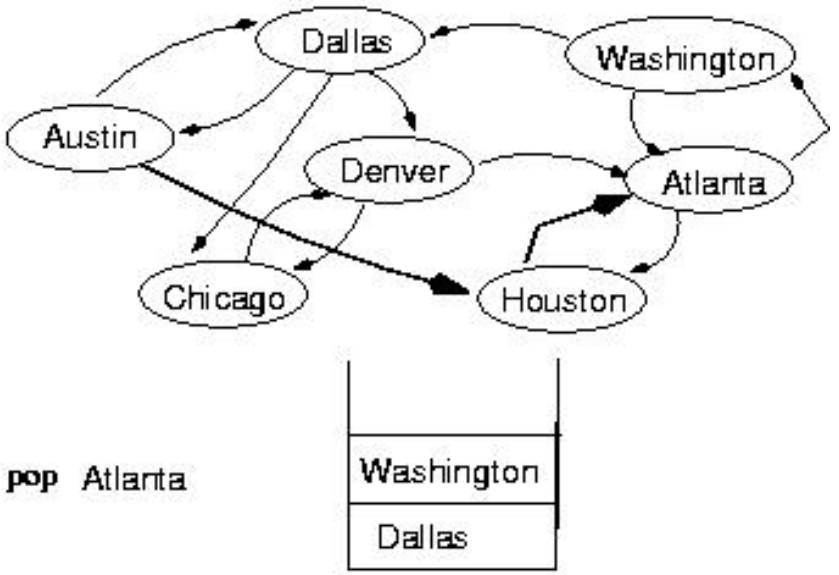


(initialization)

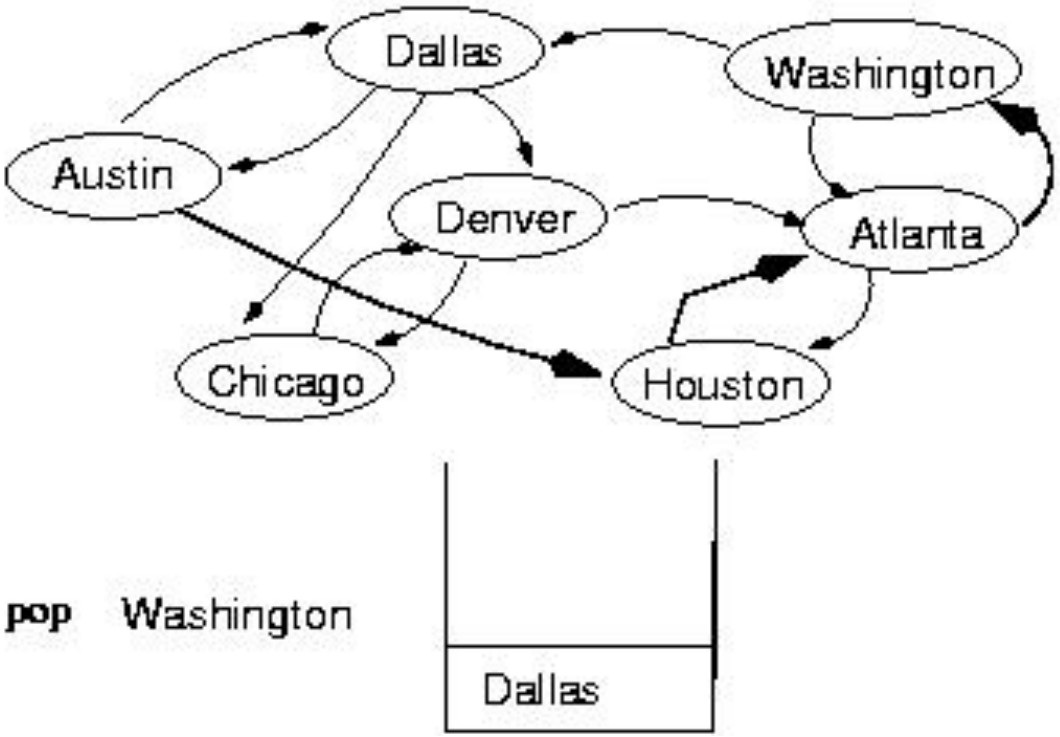


Example: Depth-First-Search (DFS)





Example: Depth-First-Search (DFS)

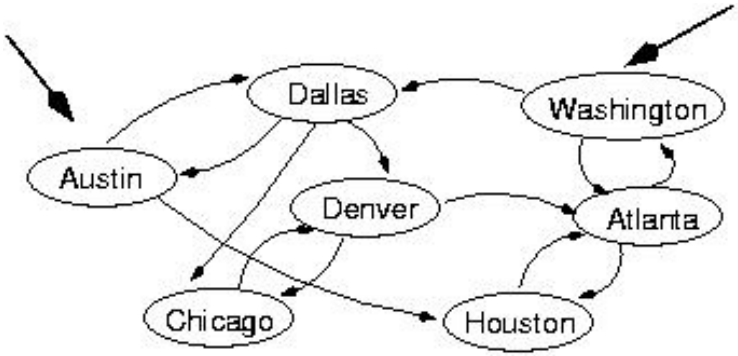


Breadth-First-Searching (BFS)

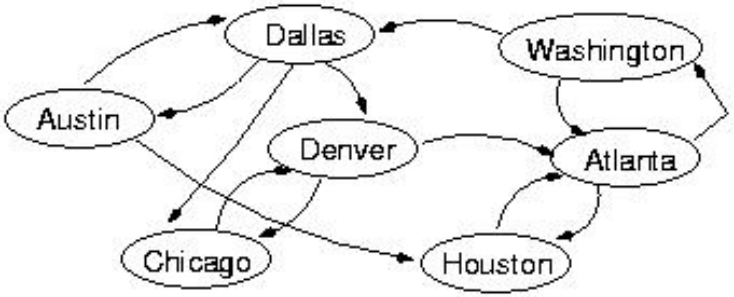
- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

Use of a queue

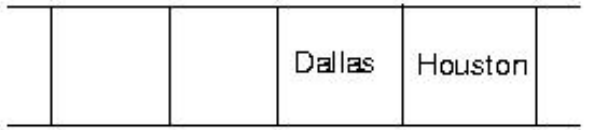
- It is very common to use a queue to keep track of:
  - nodes to be visited next, or
  - nodes that we have already visited.
- Typically, use of a queue leads to a *breadth-first* visit order.
- Breadth-first visit order is "cautious" in the sense that it examines every path of length i before going on to paths of length i+1.

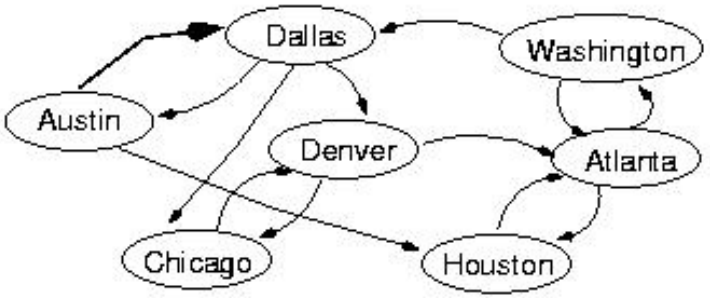


(initialization)

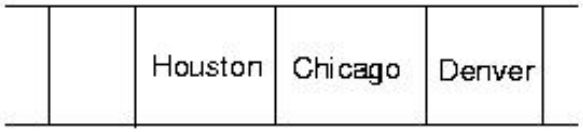


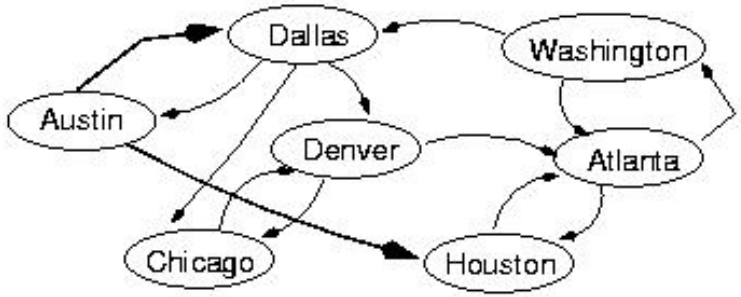
dequeue Austin



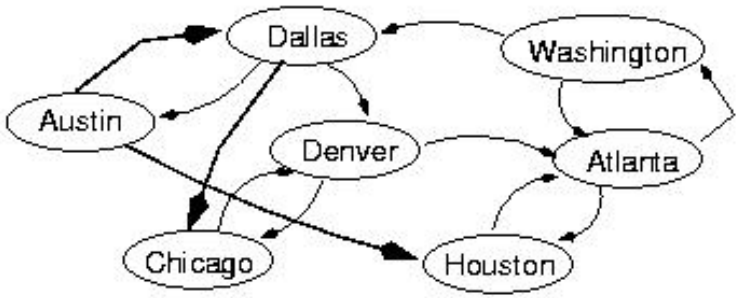


dequeue Dallas

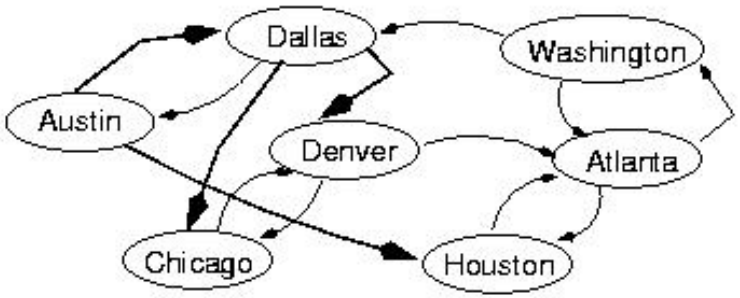




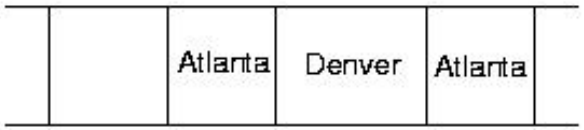
dequeue Houston

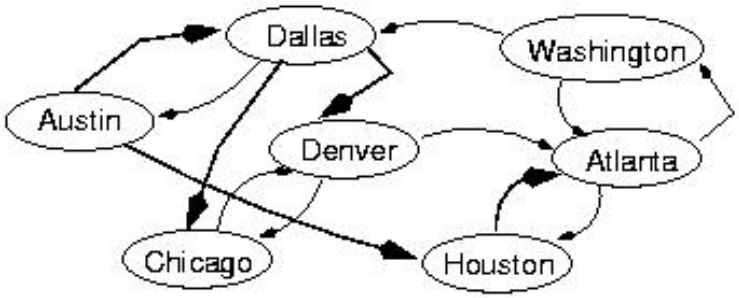


dequeue Chicago

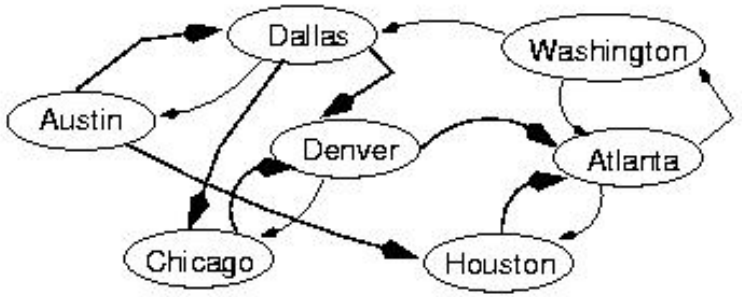


dequeue Denver

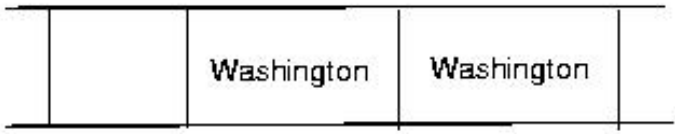


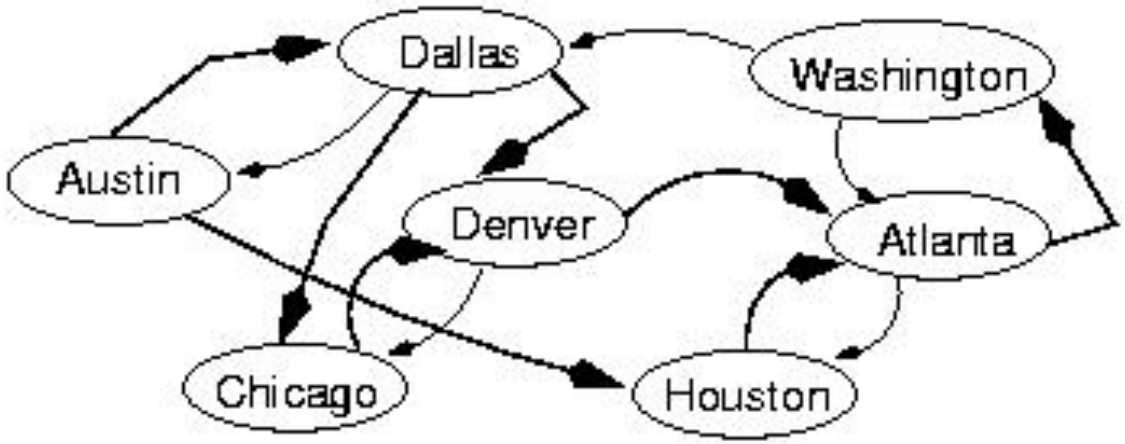


dequeue Atlanta



dequeue Denver,
next: Atlanta

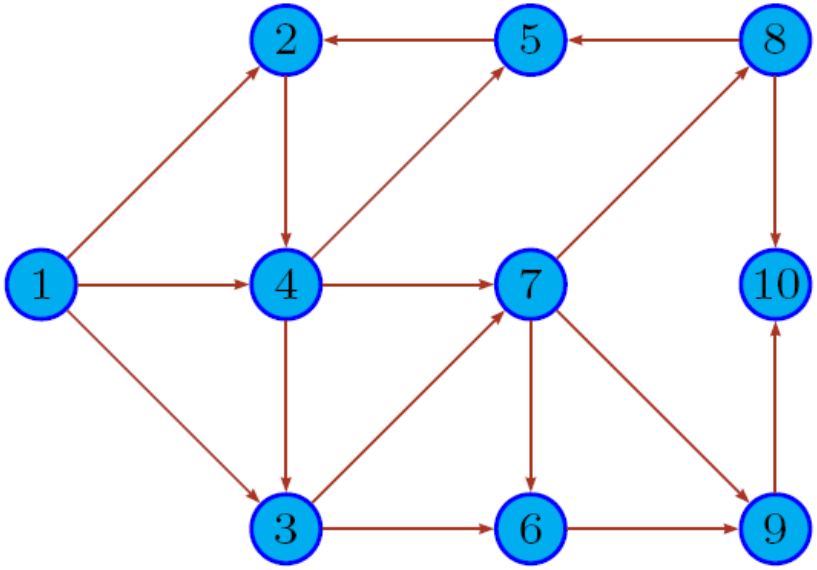



Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston


dequeue
Washington
Washington

Exercise

- Perform DFS and BFS starting from 1. Show the state of the queue and nodes visited at each stage.



Shortest Path

Shortest-path

- Suppose we want to find the shortest path from node X to node Y
- It turns out that, in order to do this, we need to find the shortest path from X to *all* other nodes
  - Why?
  - If we don't know the shortest path from X to Z, we might overlook a shorter path from X to Y that contains Z
- Dijkstra's Algorithm finds the shortest path from a given node to *all* other reachable nodes

Dijkstra's Algorithm for Single Source Shortest Path

- Classic algorithm for solving shortest path in weighted graphs (with *only positive* edge weights)
- Similar to breadth-first search, but uses a priority queue instead of a FIFO queue:
  - Always select (expand) the vertex that has a lowest-cost path to the start vertex
  - a kind of "greedy" algorithm
- Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges

Pseudocode for Dijkstra

Initialize the cost of each *vertex* to ∞

cost[s] = 0;

heap.insert(s);

While (! heap.empty())

    n = heap.deleteMin()

    For (each vertex a which is adjacent to n along edge e)

    if (cost[n] + edge_cost[e] < cost[a]) then

        cost [a] = cost[n] + edge_cost[e]

        previous_on_path_to[a] = n;

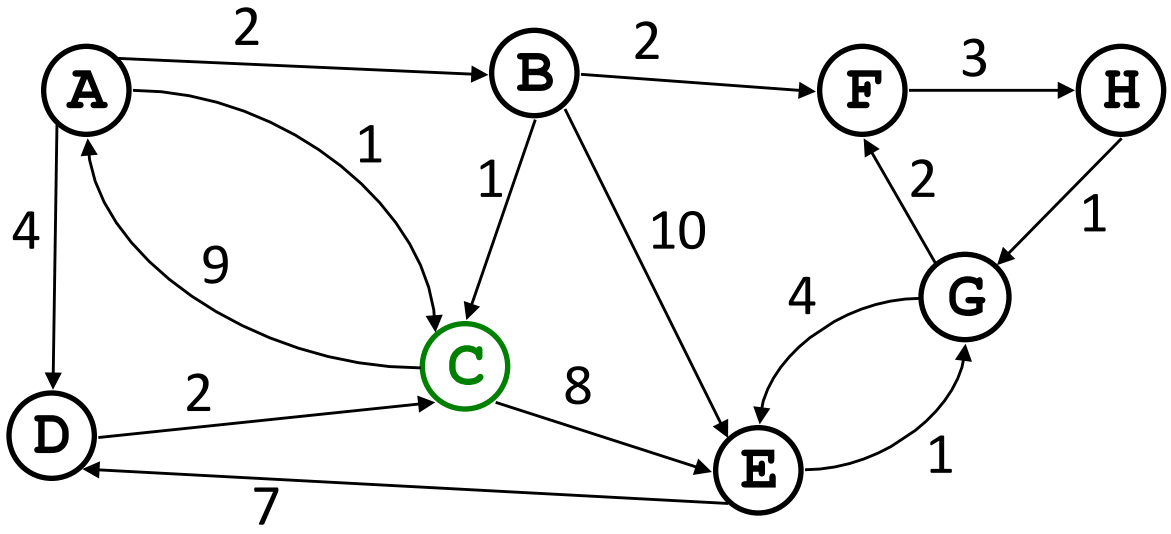
        if (a is in the heap) then heap.decreaseKey(a)

                else heap.insert(a)

Important Features

- Once a vertex is removed from the head, the cost of the shortest path to that node is known
- While a vertex is still in the heap, another shorter path to it might still be found
- The shortest path itself from s to any node a can be found by following the pointers stored in previous_on_path_to[a]

Dijkstra's Algorithm in Action



vertex	known	cost
A		
B		
C		
D		
E		
F		
G		
H		

Exercise: Shortest Path

- Run Dijkstra's shortest path algorithm on the following digraph using C as the source. Give the order in which the vertices are removed from the priority queue. Note that A is not reachable from C, so it is never added (and thus never removed) from the priority queue.