Lecture 15: Graph Traversals

CSE 332: Data Structures & Parallelism

Winston Jodjana

Summer 2023

Take Handouts!

(Raise your hand if you need one)

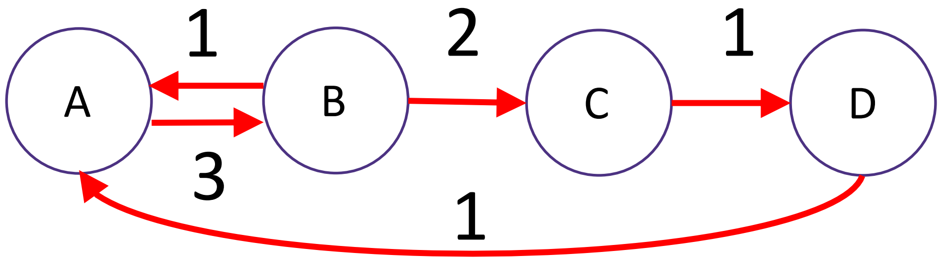
Announcements

- P2
  - Due Tomorrow, late Thursday
  - Most people use late days
- EX09: Hashing
  - Due this Friday
- EX10: Sorting
  - Due this Friday

Today

- Graph Terminologies
  - Paths vs Cycles
  - Connected vs Unconnected
  - Sparse vs dense
- Graph Datastructures
  - Adjacency Matrix
  - Adjacency List
- Graph Traversals
  - DFS (Iterative + Recursive)
  - BFS
- Graph Shortest Paths
  - Dijkstra's

Graphs: (Walks) vs Paths vs Cycles



- Walk: Sequence of adjacent vertices
  - e.g., ABA, ABCD, ABC, etc.
- Path (or Simple Path): A walk that doesn't repeat a vertex
  - e.g., ABCD, ABC, AB
  - NOT ABA
- Cycle: A walk that doesn't repeat a vertex except the first and last vertex
  - e.g., ABCDA
  - NOT ABCD

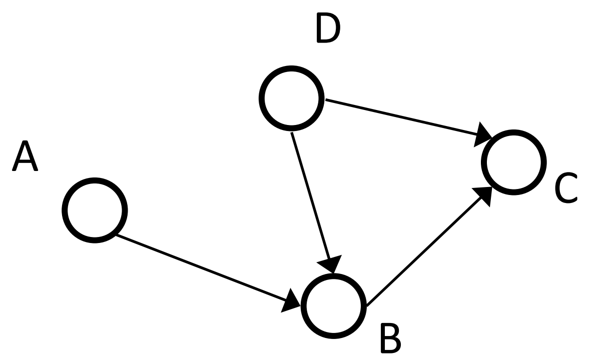
____ Length: Number of edges in ____

____ Cost: Sum of weights of each edge in ____

Graphs: Paths vs Cycles Example

- Is there a path from A to D?

- Does the graph contain any cycles?



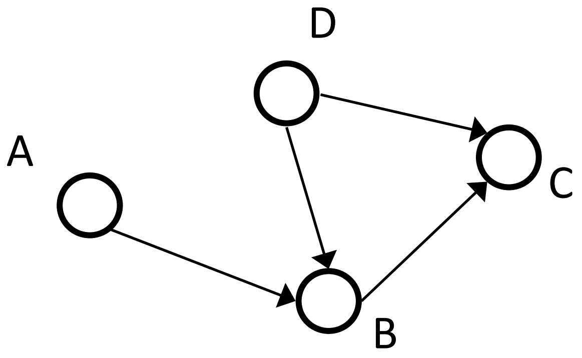
- What if undirected?

Graphs: Paths vs Cycles Example (Soln.)

- Is there a path from A to D?

No

- Does the graph contain any cycles? No

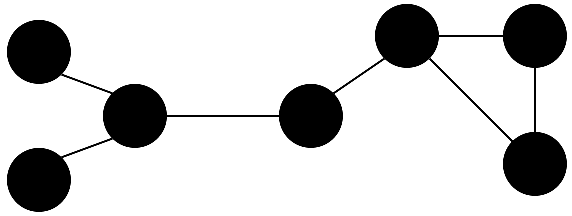


- What if undirected?

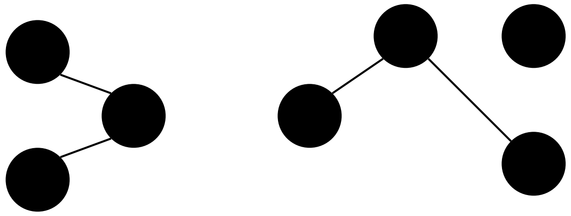
Yes, Yes

Graphs: Undirected Graph Connectivity

- An undirected graph is connected if for all pairs of vertices (v, u), there exists a path from v to u



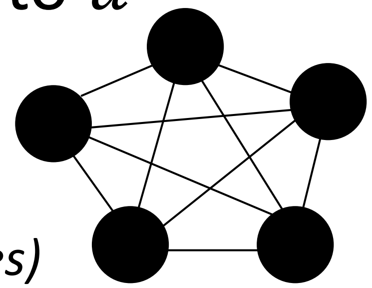
Connected graph



Disconnected graph

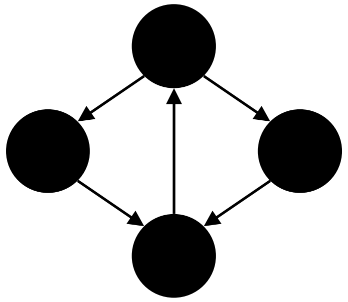
- An undirected graph is complete, a.k.a. fully connected if for all pairs of vertices (v, u), there exists an edge from v to u

(plus self-edges)

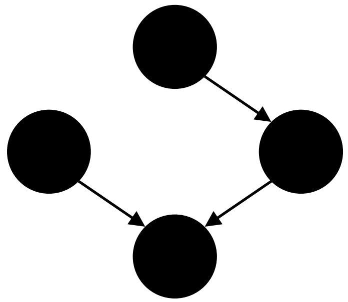


Graphs: Directed Graph Connectivity

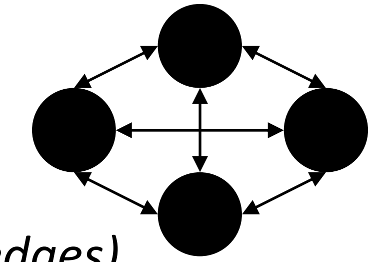
- A directed graph is strongly connected if there is a path from every vertex to every other vertex



- A directed graph is weakly connected if there is a path from every vertex to every other vertex ignoring direction of edges



- A directed graph is complete a.k.a. fully connected if for all pairs of vertices (v, u), there exists an edge from v to u



(plus self-edges)

Graphs: Practical Examples

For undirected graphs: connected?

For directed graphs: strongly connected? weakly connected?

weighted?

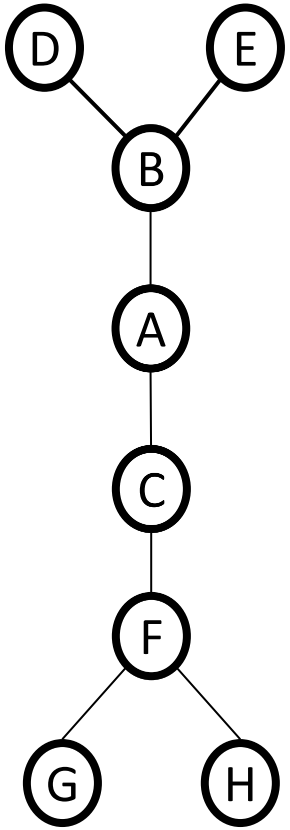
- Web pages with links
- Facebook friends
- Methods in a program that call each other
- Road maps (e.g., Google maps)
- Airline routes
- Course pre-requisites
- …

Graphs: Trees

- When talking about graphs, we say a tree is a graph that is:
  - undirected
  - acyclic
  - connected

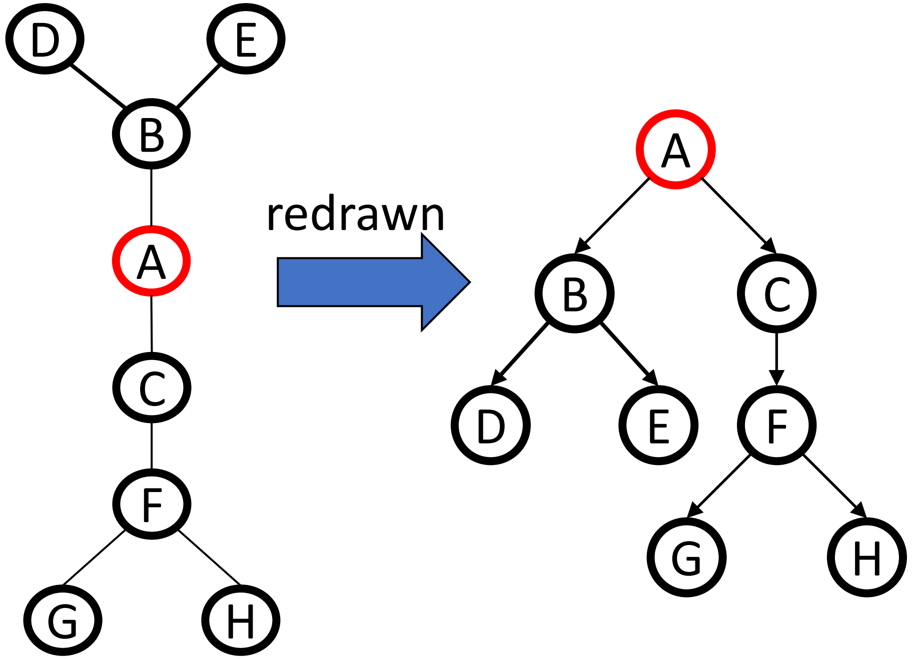
- So all trees are graphs, but not all graphs are trees

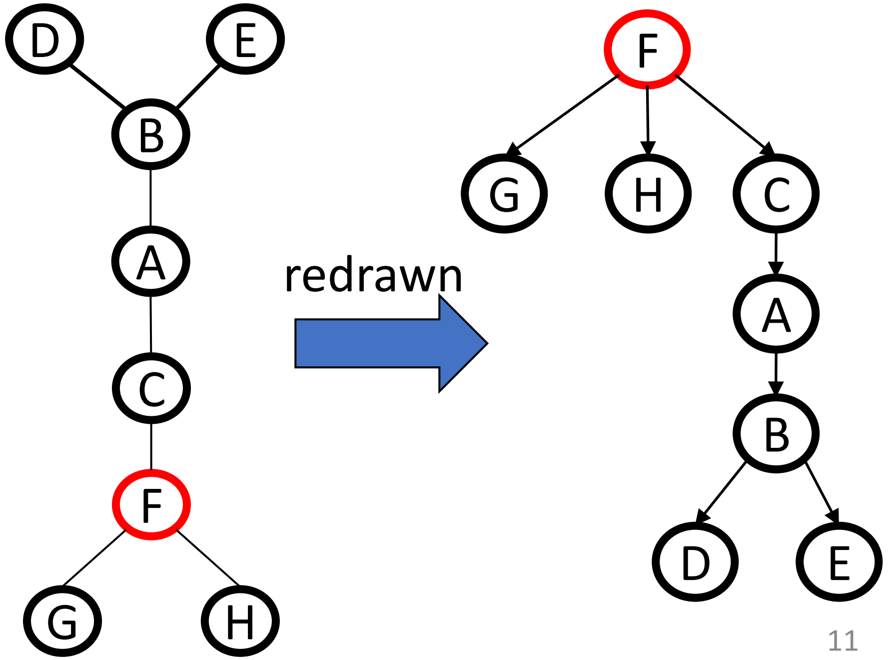
- How does this relate to the trees we know and love?...



Graphs: Rooted Trees

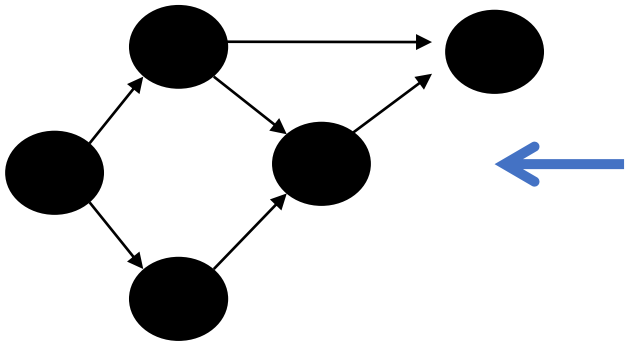
- We are more accustomed to rooted trees where:
  - We identify a unique (“special”) root
  - We think of edges as **directed**: parent to children
- Given a tree, once you pick a root, you have a unique rooted tree (just drawn differently and with undirected edges)





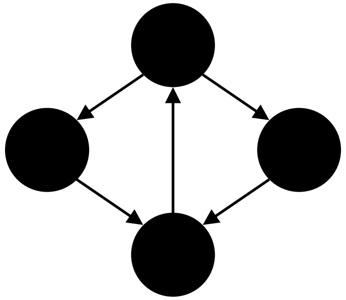
Graphs: Directed Acyclic Graphs (DAGs)

- A DAG is a directed graph with no cycles (Acyclic)
  - Every rooted directed tree is a DAG
    - But not every DAG is a rooted directed tree:



Not a rooted directed tree,
Has a cycle (in the undirected sense)

- Every DAG is a directed graph
  - But not every directed graph is a DAG:



Graphs: Number of Vertices vs Edges (Math)

- Correct Mathematical Notation:
  - Number of Vertices = $|\{v_1, v_2, \dots, v_n\}| = |V|$
  - Number of Edges = $|\{e_1, e_2, \dots, e_m\}| = |E|$
- Common Notation: V or E
- Given $|V|$ vertices, what is:
  - Minimum number of Edges?
  - Maximum for undirected?
  - Maximum for directed?

Graphs: Number of Vertices vs Edges (Math)

- Correct Mathematical Notation:
  - Number of Vertices = $|\{v_1, v_2, \dots, v_n\}| = |V|$
  - Number of Edges = $|\{e_1, e_2, \dots, e_m\}| = |E|$
- Common Notation: V or E
- Given $|V|$ vertices, what is:
  - Minimum number of Edges?
    - 0
  - Maximum for undirected?
    - $\frac{V(V+1)}{2}$ (with self-edges) or $\frac{V(V+1)}{2} - V$ (no self-edges)
  - Maximum for directed?
    - V^2

Graphs: Sparse vs Dense Graphs

- In a graph,
  - Undirected, $0 \leq |E| < |V|^2$
  - Directed: $0 \leq |E| \leq |V|^2$
- So: $|E| \in \mathcal{O}(|V|^2)$
- Sparse: when $|E| \in \Theta(|V|)$ i.e., "few edges"
- Dense: when $|E| \in \Theta(|V|^2)$ i.e., "many edges"

Sparse **Dense**

0 edges $\mathcal{O}(|V|)$ edges $\mathcal{O}(|V|^2)$ edges

Any Questions?

Today

- Graph Terminologies
  - Paths vs Cycles
  - Connected vs Unconnected
  - Sparse vs dense
- Graph Datastructures
  - Adjacency Matrix
  - Adjacency List
- Graph Traversals
  - DFS (Iterative + Recursive)
  - BFS
- Graph Shortest Paths
  - Dijkstra's

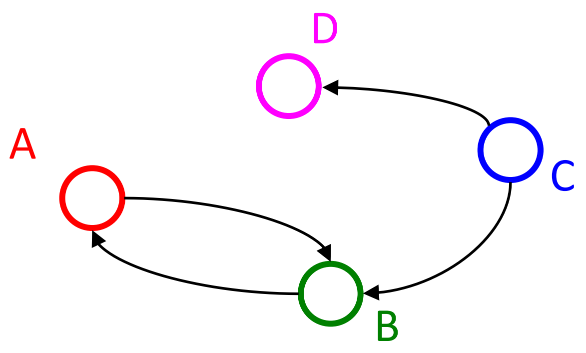
Graphs: The Data Structure

- Many data structures, tradeoffs
- Exploits graph properties
- Common operations:
  - "Is (v, u) an edge?"
  - "What are the neighbors of v?"

- Two standards:
  - Adjacency Matrix
  - Adjacency List

Graphs: Adjacency Matrix

- Assign each node a number from 0 to $|V| - 1$
- A $|V|$ by $|V|$ matrix `M` (2-D array) of Booleans
- `M[v][u]==true` means there is an edge from `v` to `u`



To / From

From \ To	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

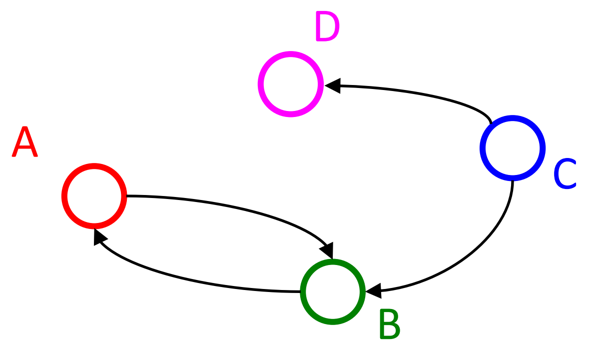
Any Questions?

Adjacency Matrix: Properties

- Running time to:
  - Get a vertex's out-bound edges:
  - Get a vertex's in-bound edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
- Better for Sparse or Dense Graphs?

To

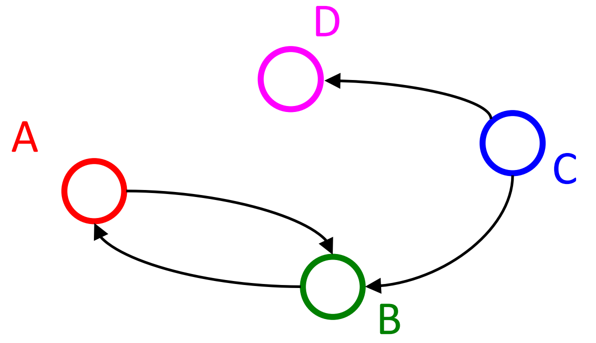
From	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F



Adjacency Matrix: Properties (Soln.)

- Running time to:
  - Get a vertex's out-bound edges: $\mathcal{O}(|V|)$
  - Get a vertex's in-bound edges: $\mathcal{O}(|V|)$
  - Decide if some edge exists: $\mathcal{O}(1)$
  - Insert an edge: $\mathcal{O}(1)$
  - Delete an edge: $\mathcal{O}(1)$
- Space requirements: $\mathcal{O}(|V|^2)$
- Better for Sparse or Dense Graphs? Dense

From \ To	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F



Adjacency Matrix: Adaptability

- How does it work for undirected graph?

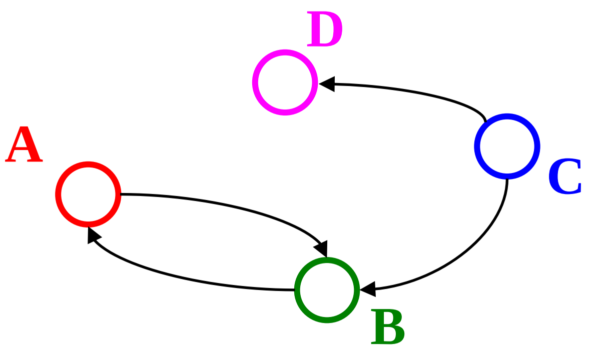
- How does it work for weighted graph?

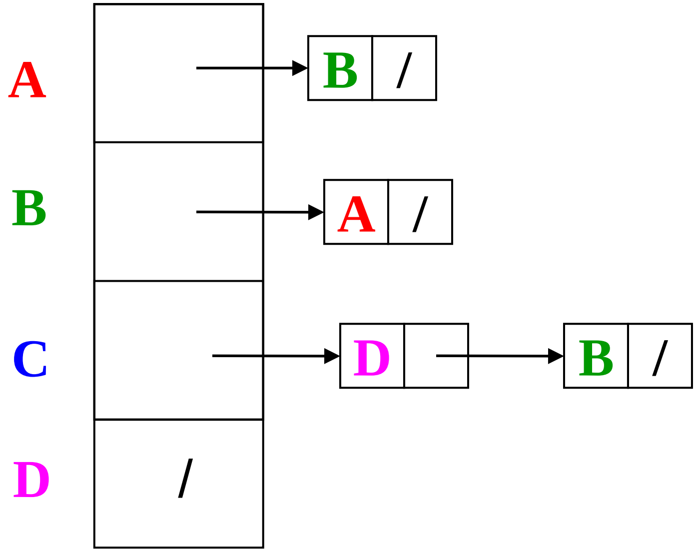
Adjacency Matrix: Adaptability (Soln.)

- How does it work for undirected graph?
  - Symmetric in diagonal axis (e.g., `M[v][u]==true`, then `M[u][v]==true`)
- How does it work for weighted graph?
  - Instead of boolean, use integer
  - "not an edge" can be 0, -1, infinite, etc.

Graphs: Adjacency List

- Assign each node a number from 0 to $|V| - 1$
- An array `arr` of length $|V|$ where `arr[i]` stores a (linked) list of all adjacent vertices

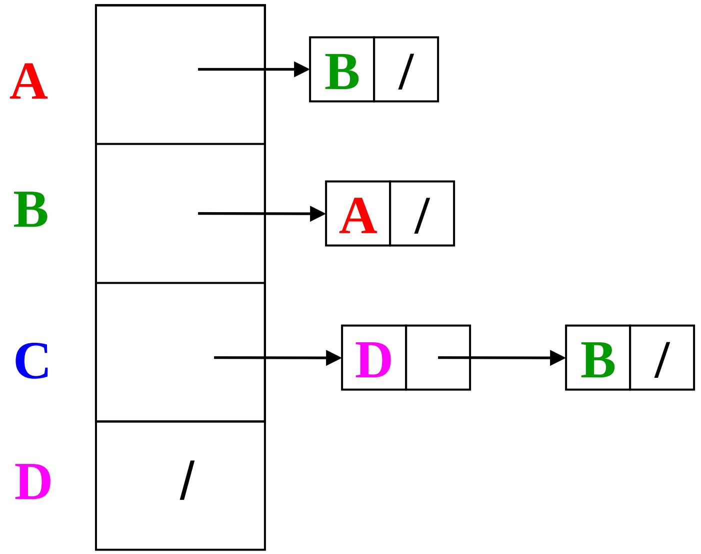


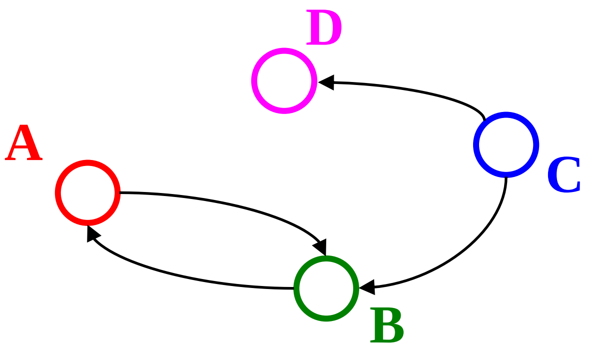


Any Questions?

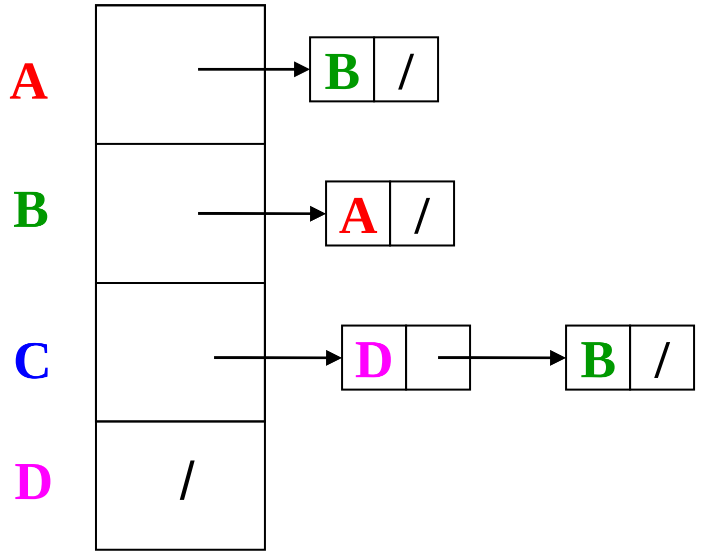
Adjacency List: Properties

- Running time to:
  - Get a vertex's out-bound edges:
  - Get a vertex's in-bound edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
- Better for Sparse or Dense Graphs?

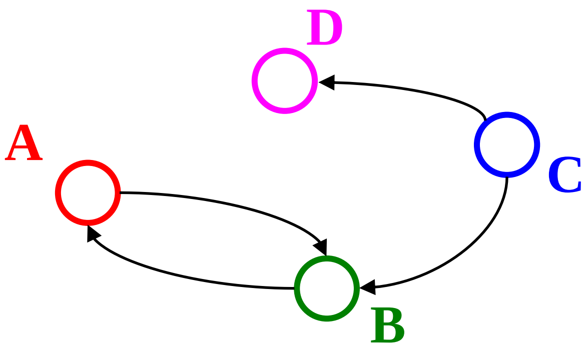




Adjacency List: Properties (Soln.)



- Running time to:
  - Get a vertex's out-bound edges:
    - $\mathcal{O}(d)$, where d is out-degree of vertex
  - Get a vertex's in-bound edges:
    - $\mathcal{O}(|V| + |E|)$, note: can keep 2nd "reverse" adjacency list for faster
  - Decide if some edge exists:
    - $\mathcal{O}(d)$, where d is out-degree of source vertex
  - Insert an edge:
    - $\mathcal{O}(1)$, unless you need to check for duplicates then $\mathcal{O}(d)$
  - Delete an edge:
    - $\mathcal{O}(d)$



- Space requirements: $\mathcal{O}(|V| + |E|)$
- Better for Sparse or Dense Graphs? Sparse

Any Questions?

Matrix vs List, which is better?

- Graphs are often sparse:
  - Streets form grids
    - every corner is not connected to every other corner
  - Airlines rarely fly to all possible cities
    - or if they do it is to/from a hub rather than directly to/from all small cities to other small cities

- Adjacency lists should generally be your default choice
  - Slower performance compensated by greater space savings

Today

- Graph Terminologies
  - Paths vs Cycles
  - Connected vs Unconnected
  - Sparse vs dense
- Graph Datastructures
  - Adjacency Matrix
  - Adjacency List
- Graph Traversals
  - DFS (Iterative + Recursive)
  - BFS
- Graph Shortest Paths
  - Dijkstra's

Graphs: Algorithms

Okay, we can represent graphs

Now let’s implement some useful and non-trivial algorithms

- Graph Traversals: Depth-first graph search (DFS) & Breadth-first graph search (BFS)
- Shortest paths: Find the shortest or lowest-cost path from x to y
  - Related: Determine if there even is such a path

Graphs: Traversals

Problem: In a graph `G`, find all nodes from a node `src`

- i.e., Is there a path from `src` to specific nodes?

Useful for doing something (processing) at a node (e.g., print the node)

Basic Idea:

- Keep following nodes
- "mark" nodes after visiting them such that it processes each node once

Traversal: Abstract "Pseudocode"

```
traverseGraph(Node src) {
  Set pending = new DataStructure();
  pending.add(src)
  mark src as visited
  while(pending is not empty) {
      v = pending.remove()
      for each node u adjacent to v // i.e., all of v's neighbour(s)
            if(u is not marked) {
                  mark u
                  pending.add(u)
            }
  }
}
```

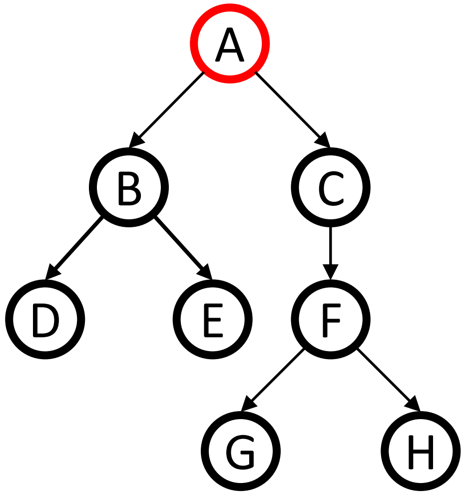
Traversal: Algorithms

- Depth-First Search
  - Uses a Stack
  - (Recursively) Explore far away from `src` first
- Breadth-First Search
  - Uses a Queue
  - Explore everything near `src` first

Traversal: Internet Warning

You know the drill.

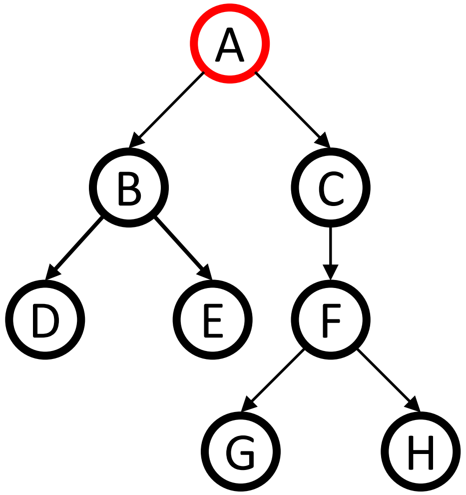
Traversal: Iterative DFS (Less common)



Order Processed:

```
IterativeDFS(Node src) {
   s = new Stack()
   s.push(src)
   mark src as visited
   while(s is not empty) {
        v = s.pop() // and "process"
        for each node u adjacent to v
        if(u is not marked)
                mark u as visited
                s.push(u)
        }
   }
}
```

Traversal: Iterative DFS (Less common) (Soln.)



Order Processed:

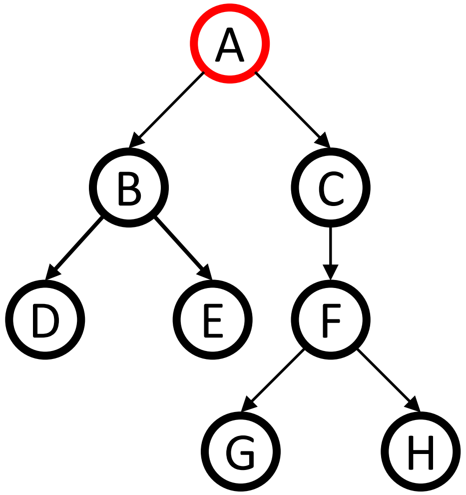
A, C, F, H, G, B, E, D

A, B, D, E, C, F, G, H

etc.

```
IterativeDFS(Node src) {
   s = new Stack()
   s.push(src)
   mark src as visited
   while(s is not empty) {
        v = s.pop() // and "process"
        for each node u adjacent to v
        if(u is not marked)
                mark u as visited
                s.push(u)
        }
   }
}
```

Traversal: Recursive DFS (More common)



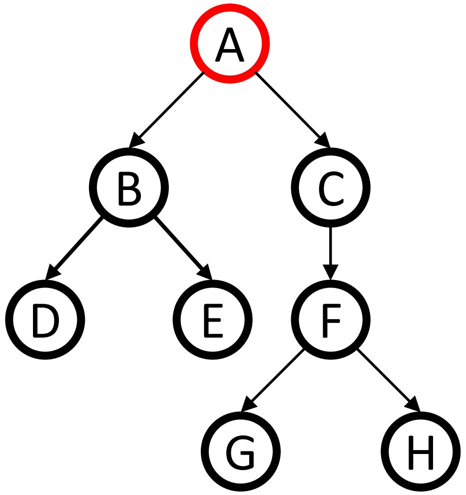

```
RecursiveDFS(Node v) {
   mark v as visited // and "process"
   for each node u adjacent to v
        if u is not marked
                RecursiveDFS(u)
}
```

Order Processed:

Same as before!

Any Questions?

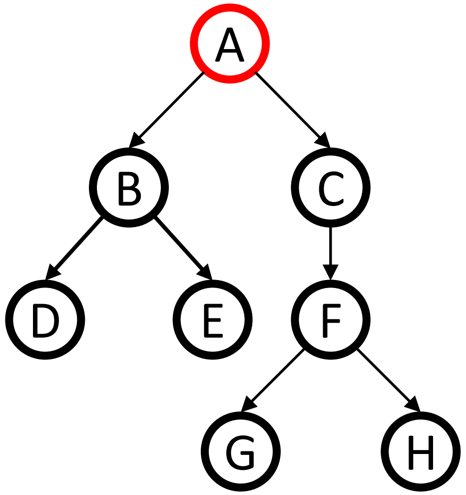
Traversal: BFS (Soln.)



Order Processed:

```
BFS(Node src) {
   s = new Queue()
   s.enqueue(src)
   mark src as visited
   while(s is not empty) {
        v = s.dequeue() // and "process"
        for each node u adjacent to v
        if(u is not marked)
               mark u as visited
               s.enqueue(u)
        }
   }
}
```

Traversal: BFS (Soln.)



Order Processed:

A, B, C, D, E, F, G, H

etc., any level-order traversal

```
BFS(Node src) {
   s = new Queue()
   s.enqueue(src)
   mark src as visited
   while(s is not empty) {
       v = s.dequeue() // and "process"
       for each node u adjacent to v
       if(u is not marked)
              mark u as visited
              s.enqueue(u)
       }
   }
}
```

Traversal: DFS vs BFS

- Depth-First Search (DFS):
  - Memory: Generally, DFS uses less memory compared to BFS as it only needs to store the nodes along the current branch.
  - Applications: Topological Sorting, Cycle Detection, etc.
- Breadth-First Search (BFS):
  - Memory: BFS tends to use more memory than DFS, as it needs to store all nodes at the current level before moving to the next level.
  - Applications: Shortest Paths
- 3rd Option: Iterative Deep DFS (IDDFS)
  - Use DFS with increasing depth limits
  - Good memory + finds shortest path

Traversal: Saving the Path

- Old Problem: Is there a path from `src` to specific nodes?
- New Problem: What is the path from `src` to specific nodes?

Q: How do we output the actual path?

A:

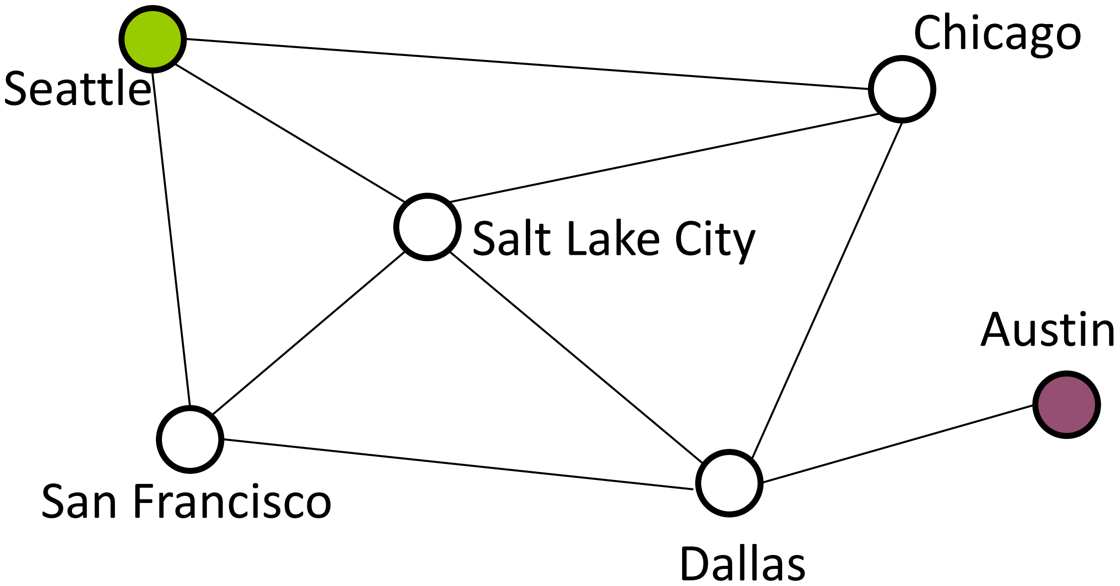
- When marking, store the predecessor (previous) node along the path
- When you're done search, follow the pred backwards to where you started (and then reverse it to get the path)

BFS with Path Saving

```
IterativeDFS(Node src) {
  s = new Queue()
  s.enqueue(src)
  src.pred = null // same as marking src as visited
  while(s is not empty) {
      v = s.dequeue() // and "process"
      for each node u adjacent to v
      if(u is not marked)
            u.pred = v // previous node of u in the path is v
            s.enqueue(u)
      }
  }
}
```

Traversal: BFS Shortest Path Example

What is the shortest path from Seattle to Austin?



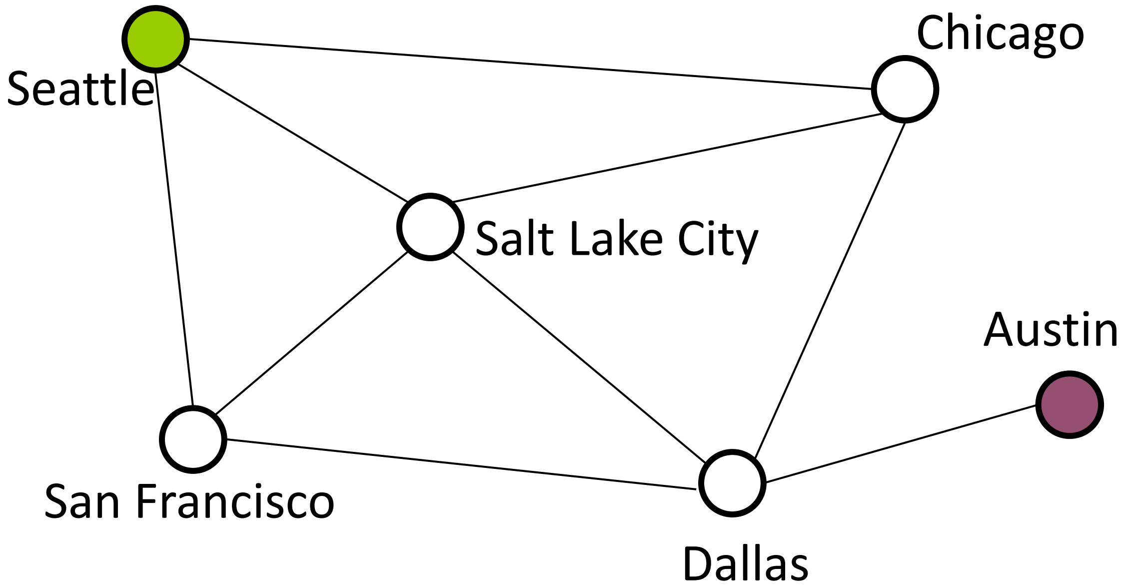
Traversal: BFS Shortest Path Example (Soln.)

What is the shortest path from Seattle to Austin?

Seattle -> Chicago -> Dallas -> Austin

Seattle -> Salt Lake City -> Dallas -> Austin

Seattle -> San Francisco -> Dallas -> Austin



Any Questions?

Today

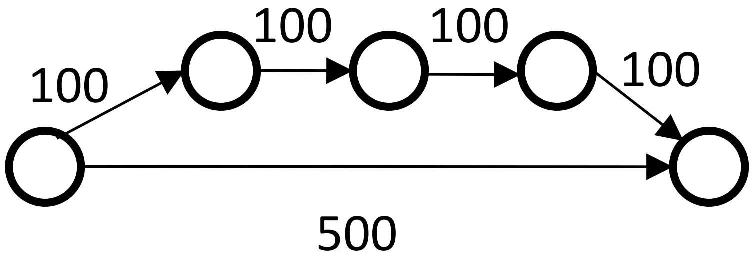
- Graph Terminologies
  - Paths vs Cycles
  - Connected vs Unconnected
  - Sparse vs dense
- Graph Datastructures
  - Adjacency Matrix
  - Adjacency List
- Graph Traversals
  - DFS (Iterative + Recursive)
  - BFS
- Graph Shortest Paths
  - Dijkstra's

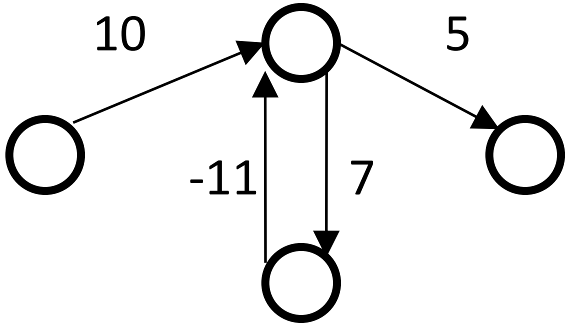
Shortest Path: Applications

- Google Maps
- Network routing
- Driving directions
- Cheap flight tickets
- Critical paths in project management (see textbook)
- etc.

Shortest Path: Weighted Graphs

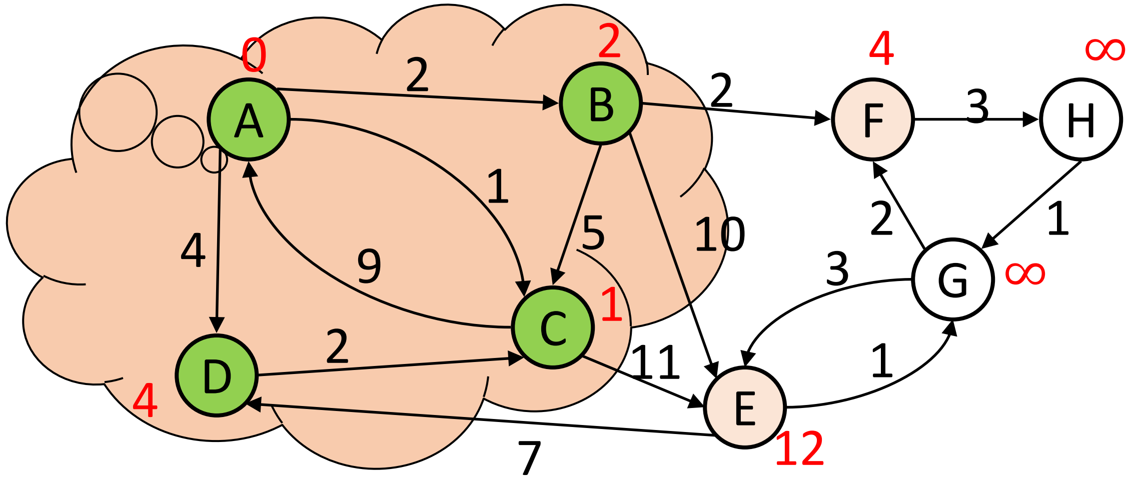
New Problem: What is the shortest path from `src` to specific nodes in a weighted graph?





- Why BFS won’t work: Shortest path may not have the fewest edges
  - Annoying when this happens with costs of flights
- We will assume there are no negative weights
  - Problem is ill-defined if there are negative-cost cycles
  - Some algorithms are wrong (e.g, Dijkstra's Algorithm) if edges can be negative

Shortest Path: Dijkstra's Algorithm



- Initially, start node ($\mathtt{A}$) has cost 0 and all other nodes have cost ∞
- At each step:
  - Pick closest unknown vertex $\mathtt{v}$
  - Add it to the "cloud" of known vertices
  - Update distances for nodes with edges from $\mathtt{v}$
- That’s it! (Have to prove it produces correct answers)