Graphs: Refresher

CS 4104: Data and Algorithm Analysis

Yoseph Berhanu Alebachew

May 11, 2025

Virginia Tech

Table of contents

Motivation

Why Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:

Why Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:
    - Computer networks

Why Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:
    - Computer networks
    - The World Wide Web

Why Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:
  - Computer networks
  - The World Wide Web
  - Social Networks

Why Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:
  - Computer networks
  - The World Wide Web
  - Social Networks
  - Software Systems

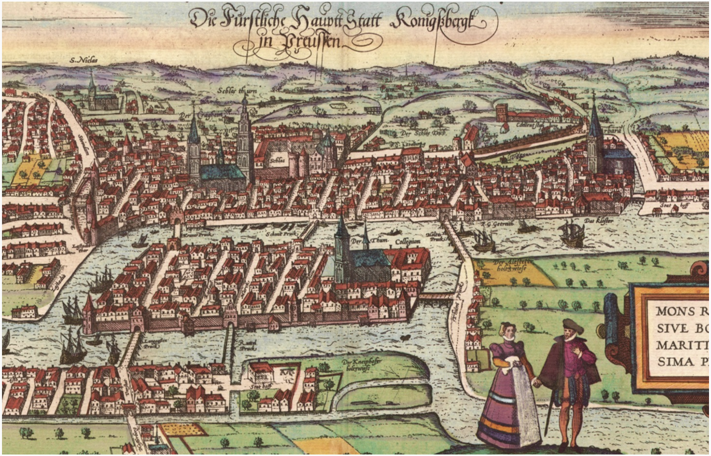
Why Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:
    - Computer networks
    - The World Wide Web
    - Social Networks
    - Software Systems
    - Job scheduling

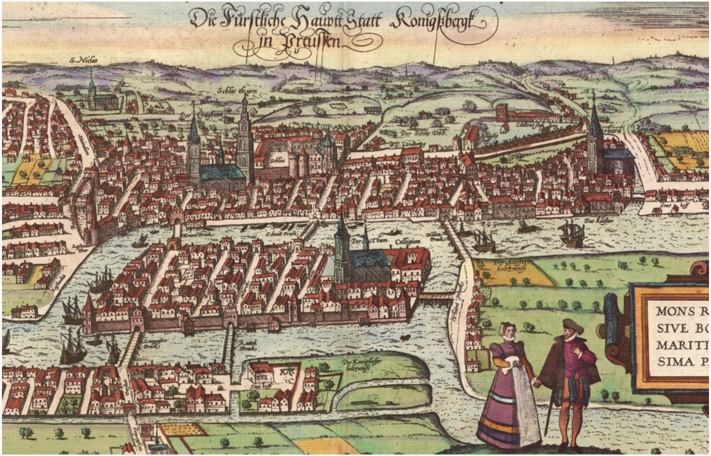
Why Graphs

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications:
  - Computer networks
  - The World Wide Web
  - Social Networks
  - Software Systems
  - Job scheduling
  - Word Morphology, and more
- In CS, we use graph data structure when we want to model non linear data structures

Euler’s Problem

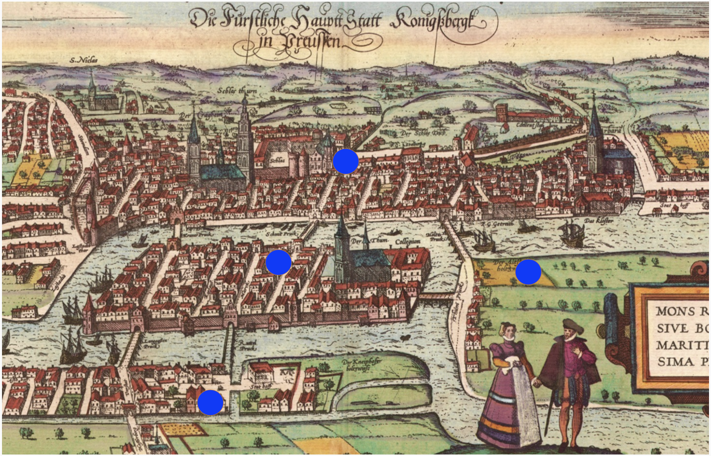


Euler’s Problem

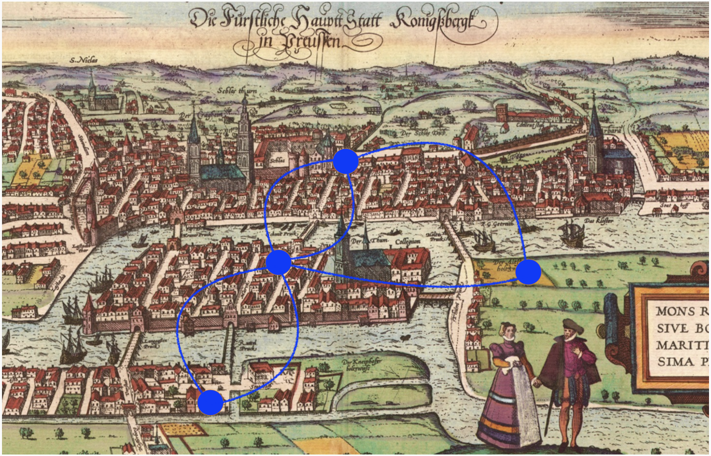


- Devise a walk through the city that crosses each of the bridges exactly once.

Euler's Problem

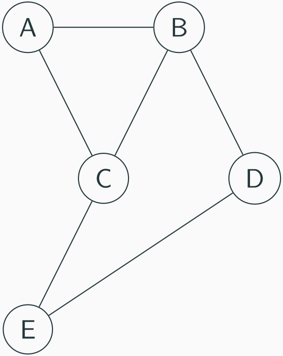


Euler's Problem



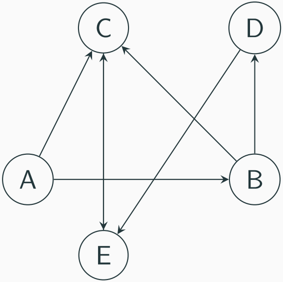
Definition

Definition: Undirected Graph



- **Undirected graph** $G = (V, E)$**:** set V of nodes and set E of edges, where $E \subset V \times V$
- Elements of E are **unordered** pairs.
- Edge (u, v) is incident on u, v; u and v are neighbours of each other.
- Exactly one edge between any pair of nodes.
- G contains no self loops, i.e., no edges of the form (u, u).

Definition: Directed Graph



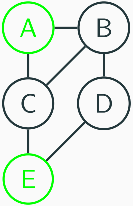
- **Directed graph** $G = (V, E)$**:** set V of nodes and set E of edges, where $E \subset V \times V$
- Elements of E are **ordered** pairs.
- Edge (u, v): u is the tail of the edge e, v is its head; e is directed from u to v.
- A pair of nodes may be connected by two directed edges: (u, v) and (v, u).
- G contains no self loops, i.e., no edges of the form (u, u).

Definition: Paths

- A $v_1 - v_k$ path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.

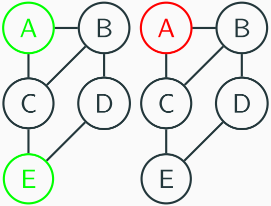
Definition: Paths

- A $v_1 - v_k$ path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is simple if all its nodes are distinct.



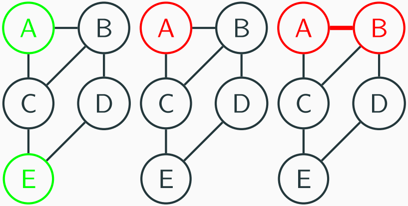
Definition: Paths

- A $v_1 - v_k$ path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is simple if all its nodes are distinct.



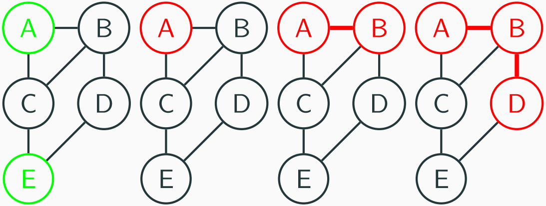
Definition: Paths

- A $v_1 - v_k$ path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is simple if all its nodes are distinct.



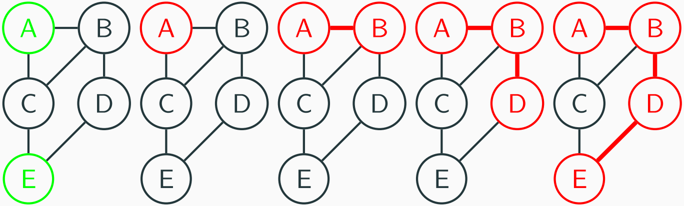
Definition: Paths

- A $v_1 - v_k$ path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is simple if all its nodes are distinct.



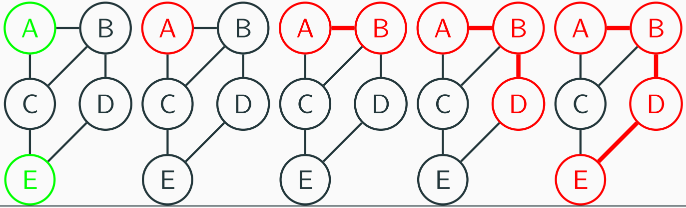
Definition: Paths

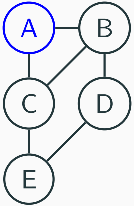
- A $v_1 - v_k$ path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is simple if all its nodes are distinct.



Definition: Paths

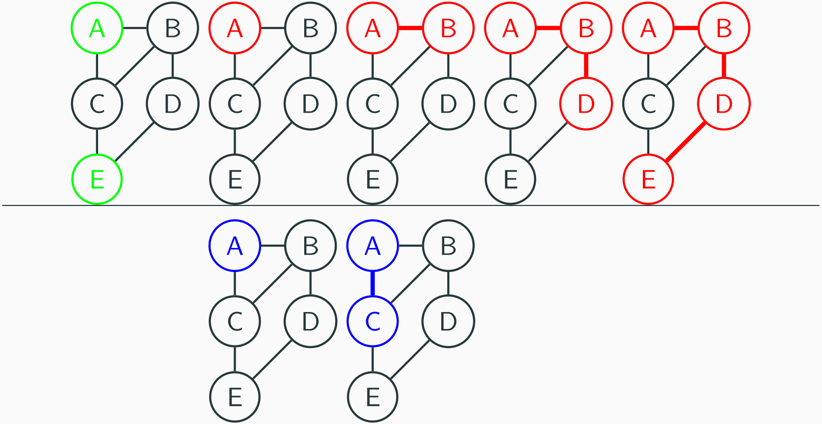
- A $v_1 - v_k$ path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is simple if all its nodes are distinct.





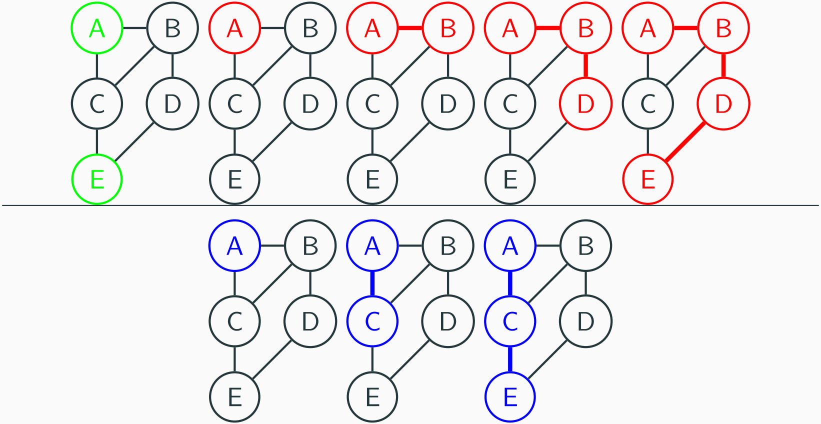
Definition: Paths

- A $v_1 - v_k$ path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is simple if all its nodes are distinct.



Definition: Paths

- A $v_1 - v_k$ path in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, ..., v_{k-1}, v_k \in V$ such that every consecutive pair of nodes $v_i, v_{i+1}, 1 \leq i < k$ is connected by an edge in E.
- A path is simple if all its nodes are distinct.

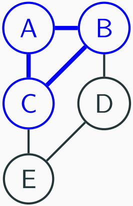


Definition: Paths

- A cycle is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.

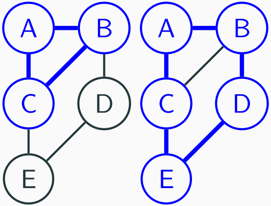
Definition: Paths

- A cycle is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.



Definition: Paths

- A cycle is a path where $k > 2$, the first $k - 1$ nodes are distinct, and $v_1 = v_k$.



Definition: Connectivity

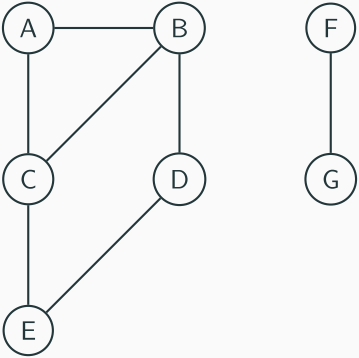
- An undirected graph G is connected if for every pair of nodes $u, v \in V$, there is a path from u to v in G.

Definition: Connectivity

- An undirected graph G is connected if for every pair of nodes $u, v \in V$, there is a path from u to v in G.
- Distance $d(u, v)$ between two nodes u and v is the minimum number of edges in any $u - v$ path.

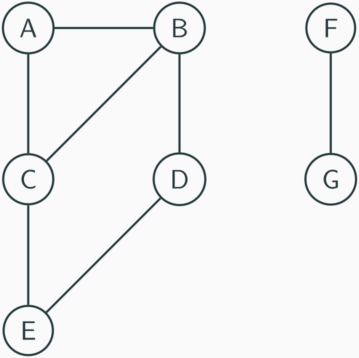
Definition: Connectivity

- An undirected graph G is connected if for every pair of nodes $u, v \in V$, there is a path from u to v in G.
- Distance $d(u, v)$ between two nodes u and v is the minimum number of edges in any $u - v$ path.



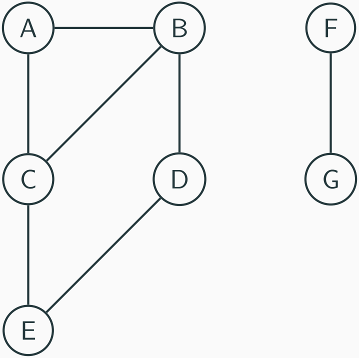
Definition: Connectivity

- An undirected graph G is connected if for every pair of nodes $u, v \in V$, there is a path from u to v in G.
- Distance $d(u, v)$ between two nodes u and v is the minimum number of edges in any $u - v$ path.

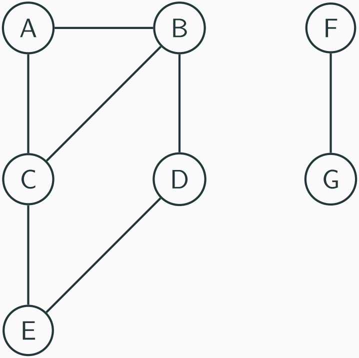


- Similar definitions carry over to directed graphs as well.

Example: Connectivity

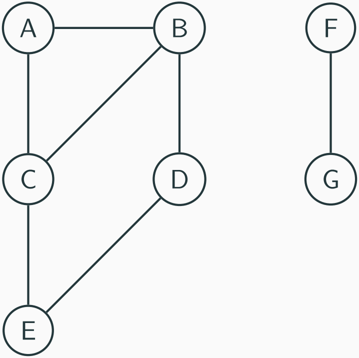


Example: Connectivity



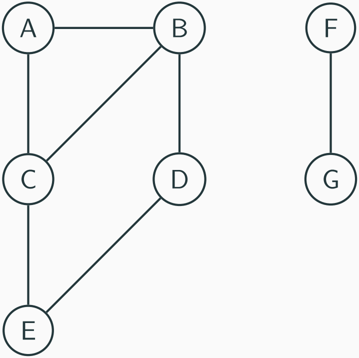
- Questions

Example: Connectivity



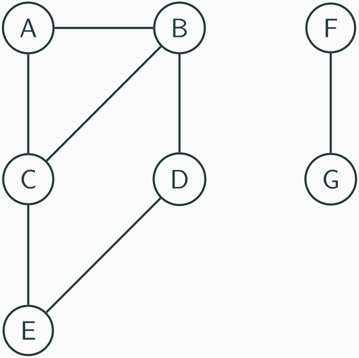
- Questions
    - Is there a path between F and C:

Example: Connectivity



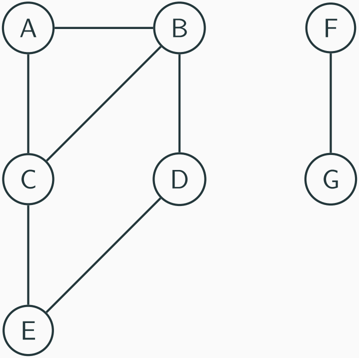
- Questions
  - Is there a path between F and C: **No**

Example: Connectivity



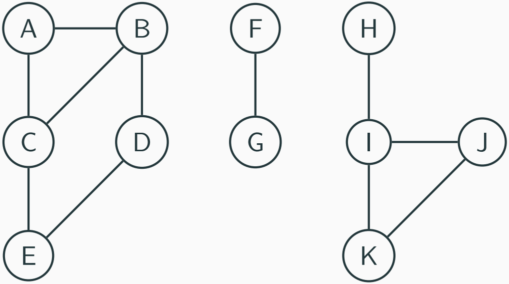
- Questions
  - Is there a path between F and C: **No**
  - What's the distance between D and A :

Example: Connectivity



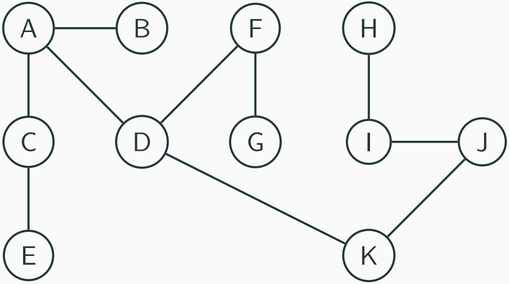
- Questions
    - Is there a path between F and C: **No**
    - What's the distance between D and A : **2**

Example: Connectivity



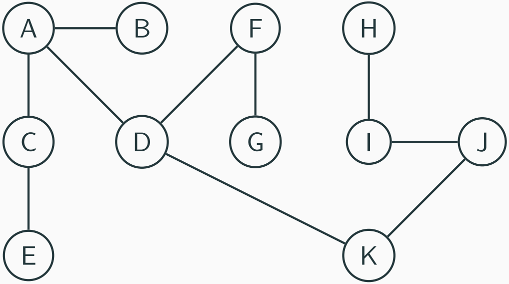
- The **connected component of the graph** containing E is the set of all nodes u such that there is a path between E and u path in the graph.
- Algorithm for the S-T Connectivity problem: compute the connected component of G that contains S and check if T is in that component.

Definition: Tree



- A connected graph G is said to be a **Tree** if there are nos cycles in the graph

Definition: Tree



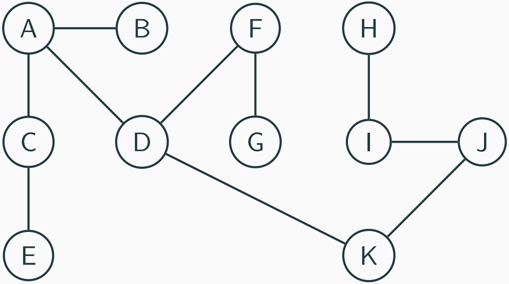
- A connected graph G is said to be a **Tree** if there are nos cycles in the graph
- If two of the following are true the third is true.
  - G is a Tree
  - G is connected
  - G does not have a cycle

Traversal

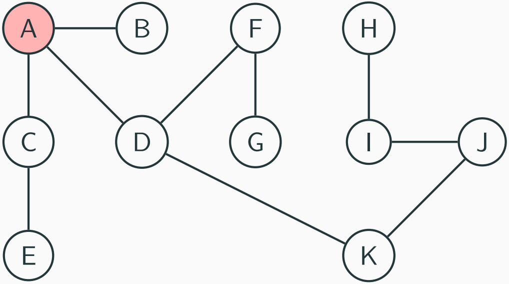
BFS: Intuition

- Breadth-First Search (BFS) is an algorithm for traversing or searching tree or graph data structures.
- It starts at the root (or an arbitrary node in the case of a graph) and explores all neighbors at the present depth prior to moving on to nodes at the next depth level.
- BFS uses a queue to keep track of the next node to explore, ensuring all nodes at the current depth are visited before moving deeper.
- BFS is useful for:
  - Finding the shortest path in unweighted graphs.
  - Finding all nodes within one connected component.

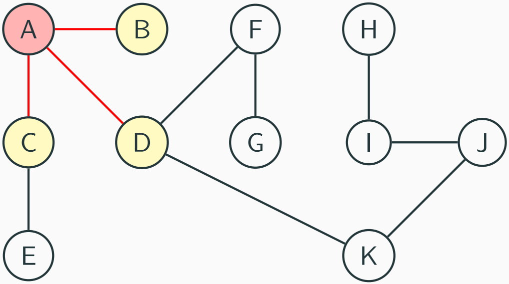
BFS: Example



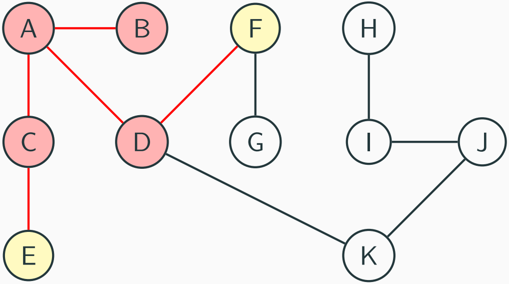
BFS: Example (Step 1)



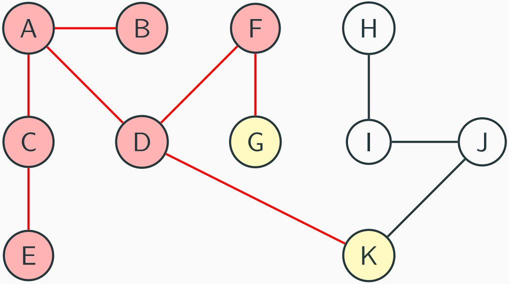
BFS: Example (Step 2)



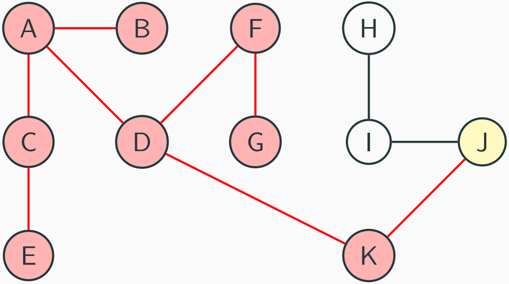
BFS: Example (Step 3)



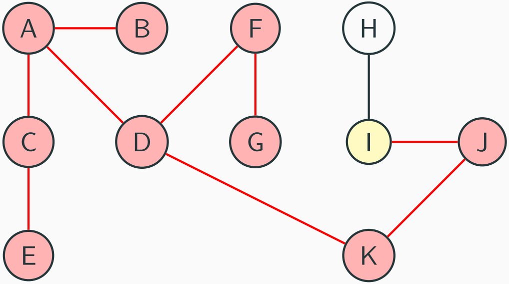
BFS: Example (Step 4)



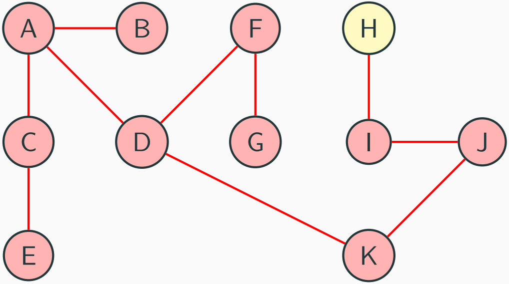
BFS: Example (Step 5)



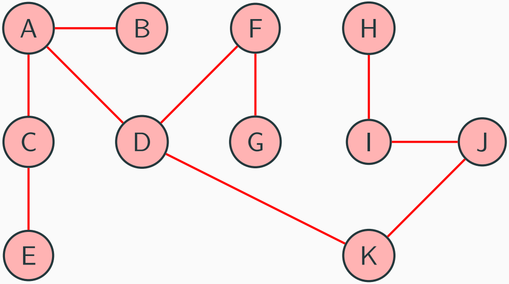
BFS: Example (Step 6)



BFS: Example (Step 7)



BFS: Example (Step 8)



BFS: Algorithm

Algorithm 1 Breadth-First Search (BFS)

Input: Graph $G = (V, E)$, starting node s
Output: Set of visited nodes

function BFS(G, s):
    let Q be a queue
    initialize Q with s
    mark s as visited
    while Q is not empty **do**
        $v \leftarrow$ **dequeue** Q
        for each neighbor w of v **do**
            if w is not visited **then**
                mark w as visited
                enqueue w into Q
            end if
        end for
    end while

BFS: Analysis

- **Time Complexity**: $O(V + E)$
  - Each vertex is enqueued and dequeued at most once.
  - Each edge is considered once when exploring the vertex at one end of the edge.
  - Therefore, the total work done is proportional to the sum of the number of vertices and edges.
- **Space Complexity**: $O(V)$
  - We need to store the visited status of each vertex, which requires $O(V)$ space.
  - The queue can grow to at most $O(V)$ size if all vertices are at the same level.

BFS: Use Cases

- **Shortest path in unweighted graphs**:
  - BFS finds the shortest path (minimum number of edges) from the source node to all other nodes.
  - Useful in scenarios like finding the shortest route in a road network where all roads have the same length.
- **Finding connected components in a graph**:
  - BFS can be used to explore all nodes in a connected component starting from any node in the component.
  - Helps in identifying and counting isolated subgraphs within a larger graph.
- **Level-order traversal of a tree**:
  - In trees, BFS is used for level-order traversal, visiting nodes level by level.
  - Commonly used in scenarios like breadth-first search in AI and game development.

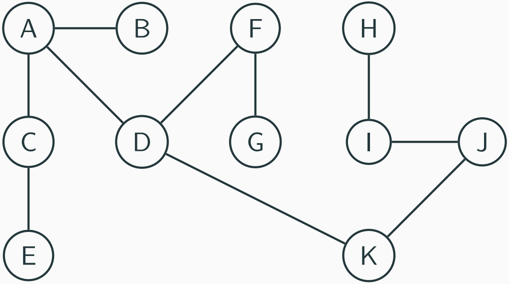
BFS: Use Cases

- **Web crawling**:
  - BFS is used to crawl web pages starting from a given URL and exploring all reachable pages within a certain depth.
  - Ensures that all pages at the current depth are visited before moving to deeper levels.
- **Social network analysis**:
  - BFS can help in exploring social networks to find shortest connections between individuals.
  - Useful for analyzing degrees of separation and influence spread in networks like Facebook or LinkedIn.

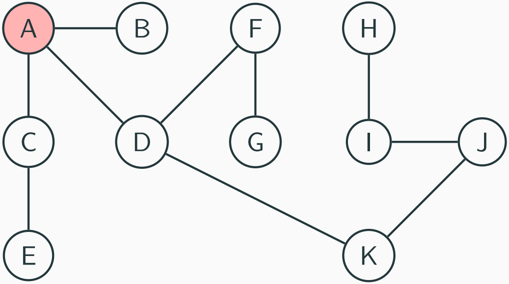
DFS: Intuition

- Depth-First Search (DFS) is an algorithm for traversing or searching tree or graph data structures.
- It starts at the root (or an arbitrary node in the case of a graph) and explores as far as possible along each branch before backtracking.
- DFS uses a stack (or recursion) to keep track of the path being explored.
- DFS is useful for:
    - Pathfinding in mazes.
    - Topological sorting.
    - Detecting cycles in graphs.

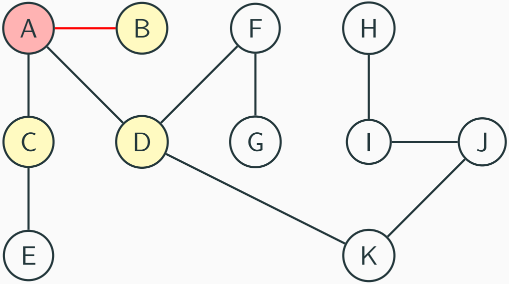
DFS: Example



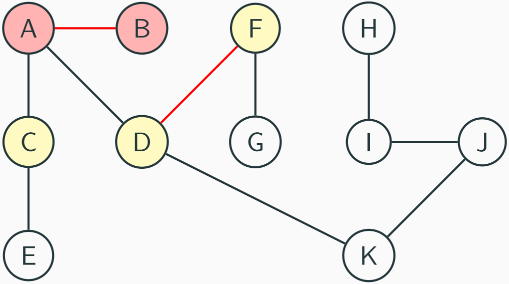
DFS: Example (Step 1)



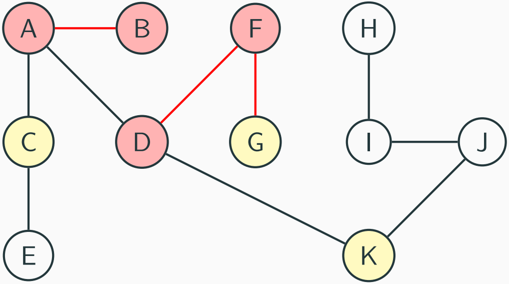
DFS: Example (Step 2)



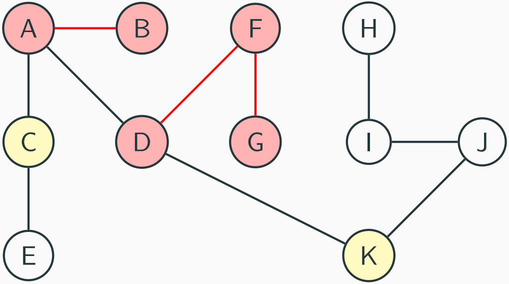
DFS: Example (Step 3)



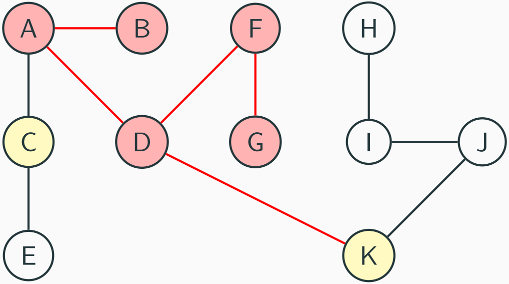
DFS: Example (Step 4)



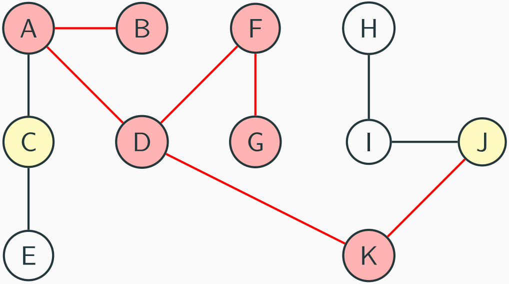
DFS: Example (Step 5)



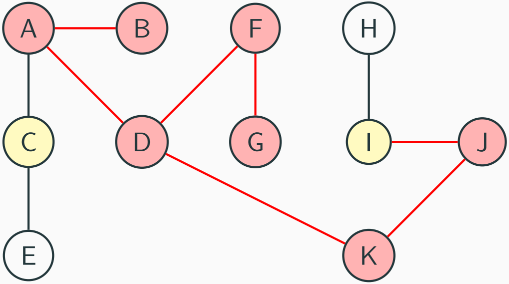
DFS: Example (Step 6)



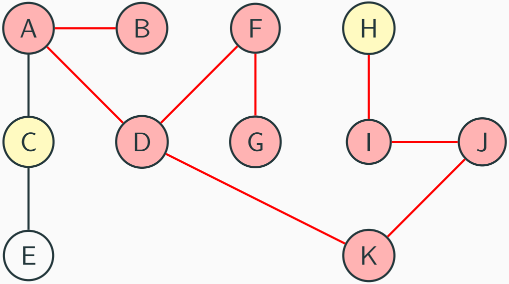
DFS: Example (Step 7)



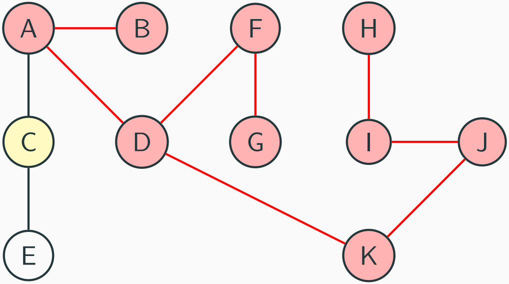
DFS: Example (Step 8)



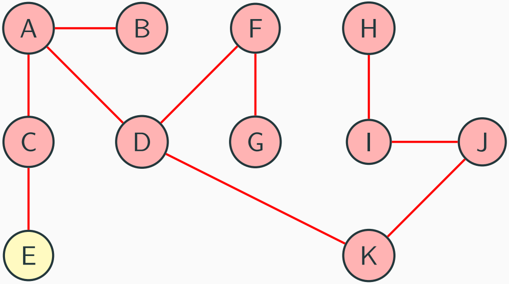
DFS: Example (Step 9)



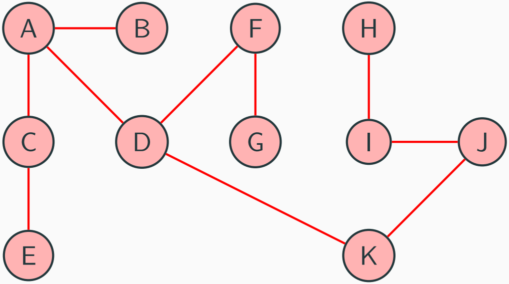
DFS: Example (Step 10)



DFS: Example (Step 11)



DFS: Example (Step 12)



DFS: Algorithm

Algorithm 2 Depth-First Search (DFS)

```
Input: Graph G = (V, E), starting node s
Output: Set of visited nodes

function DFS(G, s):
    initialize an empty stack S
    push s onto S
    mark s as visited
    while S is not empty do
        v ← pop S
        for each neighbor w of v do
            if w is not visited then
                mark w as visited
                push w onto S
            end if
        end for
    end while
```

DFS: Analysis

- **Time Complexity**: $O(V + E)$
  - Each vertex is pushed and popped from the stack at most once.
  - Each edge is explored once when visiting the vertex at one end of the edge.
  - Therefore, the total work done is proportional to the sum of the number of vertices and edges.
- **Space Complexity**: $O(V)$
  - We need to store the visited status of each vertex, which requires $O(V)$ space.
  - The stack can grow to at most $O(V)$ size in the worst case (when the graph is a single path).

DFS: Use Cases

- **Pathfinding in mazes**:
  - DFS is useful for exploring all possible paths in a maze or labyrinth.
  - It helps in finding a path from the start to the end by exploring deeper into the maze.
- **Topological sorting**:
  - DFS is used in topological sorting of directed acyclic graphs (DAGs).
  - It helps in ordering tasks or vertices such that for every directed edge uv, vertex u comes before v.
- **Detecting cycles in graphs**:
  - DFS can detect cycles in both directed and undirected graphs.
  - By keeping track of visited nodes and the recursion stack, DFS identifies back edges that form cycles.

DFS: Use Cases

- **Finding connected components**:
  - DFS is used to find all vertices in a connected component of an undirected graph.
  - Helps in identifying and counting isolated subgraphs within a larger graph.
- **Solving puzzles with only one solution**:
  - Puzzles like Sudoku can be solved using DFS by exploring possible solutions depth-wise.
  - Ensures all potential paths are explored until the correct solution is found.

BFS vs DFS

- Both visit the same set of nodes but in a different order.
- Both traverse all the edges in the connected component but in a different order.
- BFS trees have root-to-leaf paths that look as short as possible
- Paths in DFS trees tend to be long and deep.

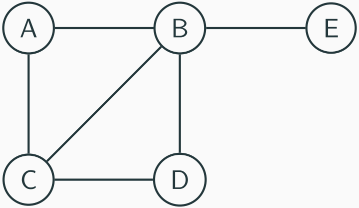
Implementation

Implementation: Representing Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- **Adjacency matrix**: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 if and only if the graph contains the edge (i, j).
- **Adjacency list**: array Adj, where Adj[v] stores a linked list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in $Adj[u]$ and $Adj[v]$.

Operation/Space	Adj. matrix	Adj. list
Is (i, j) an edge?	$O(1) time$	$O(n_i)$
Iterate over all edges incident on node i	$O(n) time$	$O(n_i)$
Space	$O(n^2)$	$O(n + m)$

Graph Representations: Example



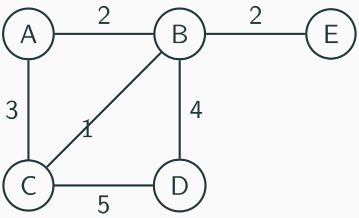
Adjacency matrix representation

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	1
C	1	1	0	1	0
D	0	1	1	0	0
E	0	1	0	0	0

Adjacency list representation

V	**Neighbors**
A	B, C
B	A, C, D, E
C	A, B, D
D	B, C
E	B

Graph Representations: Example 2



Adjacency matrix representation

	A	B	C	D	E
A	0	2	3	0	0
B	2	0	1	4	2
C	3	1	0	5	0
D	0	4	5	0	0
E	0	2	0	0	0

Adjacency list representation

V	**Neighbors**
A	B (2), C (3)
B	A (2), C (1), D (4), E (2)
C	A (3), B (1), D (5)
D	B (4), C (5)
E	B (2)

Implementation: Traversal

- "Implementation" of BFS and DFS: fully specify the algorithms and data structures so that we can obtain provably efficient times.
- Inner loop of both BFS and DFS: process the set of edges incident on a given node and the set of visited nodes.
- How do we store the set of visited nodes? Order in which we process the nodes is crucial.
    - BFS: store visited nodes in a queue (first-in, first-out).
    - DFS: store visited nodes in a stack (last-in, first-out)

Conclusion

Summary

- We discussed the motivation behind graph data structures
- Problems that can be solved with graph modeling
- Definitions and properties
- Graph representations
- Graph traversal algorithms

Next class

- Greedy Algorithms
    - Both with linear and graphs data structures
    - More graph examples

Acknowledgement

- Parts of the slides adopted from T. M. Murali @ VT