Unit #9: Graphs

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2

Unit Outline

- Topological Sort: Sorting vertices
- Graph ADT and Graph Representations
- Graph Terminology
- More Graph Algorithms
  - Shortest Path (Dijkstra's Algorithm)
  - Minimum Spanning Tree (Kruskal's Algorithm)

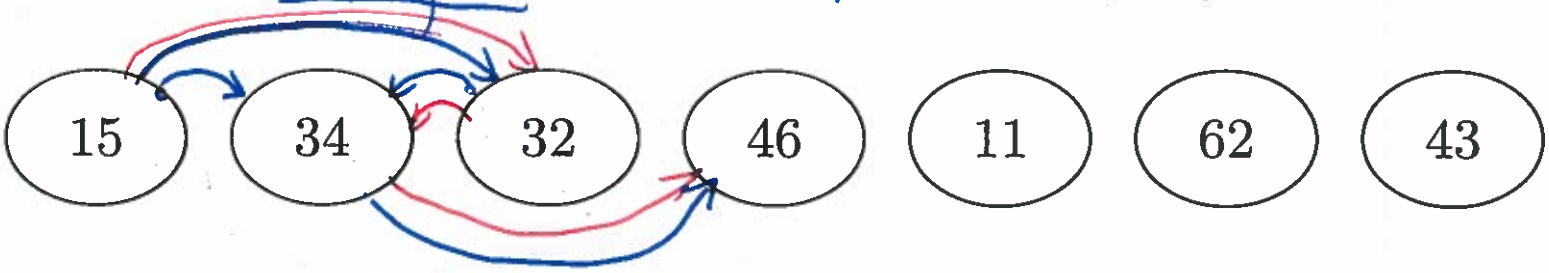
Learning Goals

- Describe the properties and possible applications of various kinds of graphs (e.g., simple, complete), and the relationships among vertices, edges, and degrees.
- Prove basic theorems about simple graphs (e.g., handshaking theorem).
- Convert between adjacency matrices/lists and their corresponding graphs.
- Determine whether two graphs are isomorphic.
- Determine whether a given graph is a subgraph of another.
- Perform breadth-first and depth-first searches in graphs.
- Execute Dijkstra's shortest path algorithm and Kruskal's minimum spanning tree algorithm on a given graph.

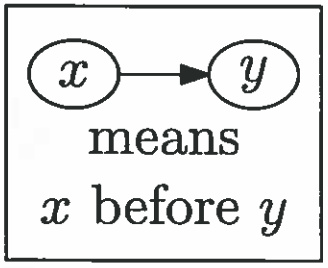
Sorting Total Orders

→ operation (<) such that for any two items x, y, we know if x<y, or y<x.

We can totally order the complete set

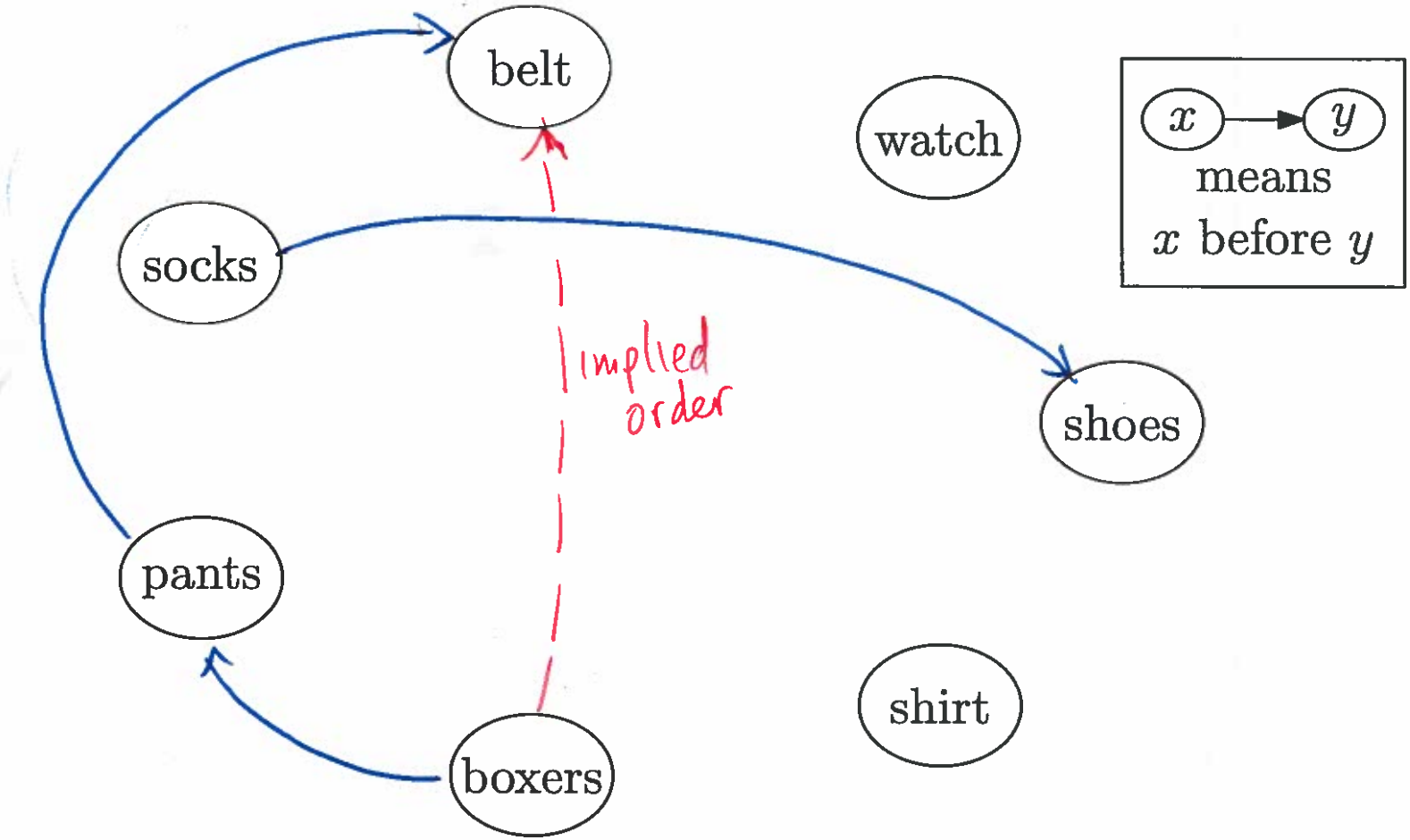


there is a subset of edges, that forms a path through all of all vertices in order in the graph.

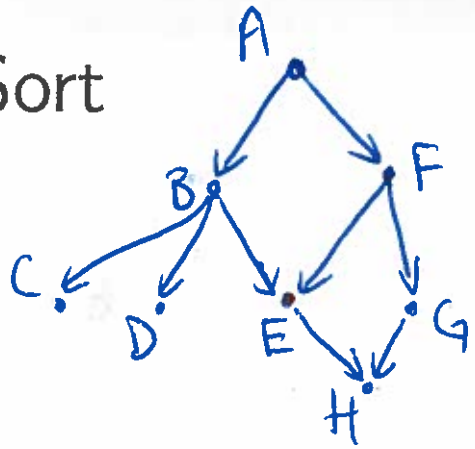


What property does the comparison-based sorting algorithm need to achieve?

Partial Order: Getting Dressed



Topological Sort



means x comes before y.

A topological sort is a total order of the vertices of a directed acyclic graph (DAG) $G = (V, E)$ such that if (u, v) is an edge of G then u appears before v in the order.

Valid?

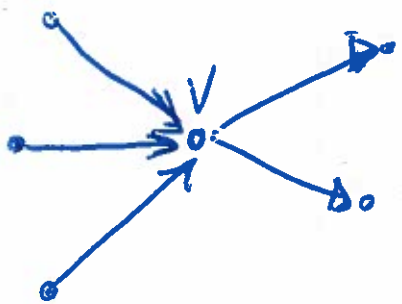
A) A, B, C, D, E, F, G, H

B) A, B, C, D, F, G, E, H ✓

C) A, F, G, B, E, H, C, D ✓

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

Topological Sort Algorithm I



in-degree(v) = 3
out-degree(v) = 2

Node:
- value
- in-degree
- list of sub-nodes

Let h = 8 = # of vertices, m = 9 = # of edges, and V = set of all vertices.

1. Find each vertex's *in-degree* (# of inbound edges).
2. While there are vertices remaining:
   - 2.1 Pick a vertex $v \in V$ with in-degree zero and output it.
   - 2.2 Reduce the in-degree of all vertices that v has an edge to.
   - 2.3 Remove v from the list of vertices.

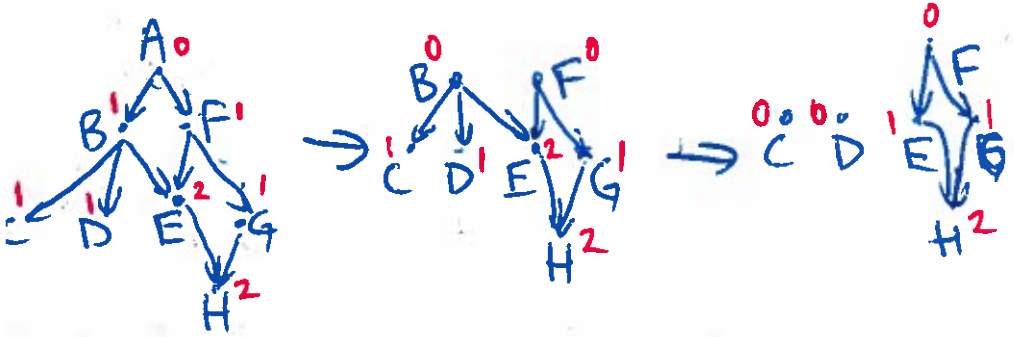
(means nothing comes "before" v.)

Runtime?

1 → $\Theta(n)$, $\Theta(m)$, $\Theta(n^2)$

n [2.1 → $\Theta(n)$
2.2 → $\Theta(m)$
2.3 → $\Theta(1)$] } $\Theta(n^2)$

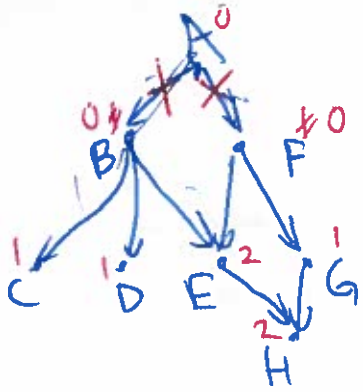
$\Theta(\text{outdegree}(v))$



B // F

Output: A B

Topological Sort Algorithm II



Let $n = \#$ of vertices, $m = \#$ of edges, and $V =$ set of all vertices.

1. Find each vertex's in-degree.
2. Initialize a queue to contain all in-degree zero vertices. — Q initially has (A)
3. While there are vertices in the queue:
   - 3.1 Dequeue a vertex v (with in-degree zero) and output it. — $\Theta(1)$
   - 3.2 Reduce the in-degree of all vertices that v has an edge to. — $\Theta(\text{outdegree}(v))$
   - 3.3 Enqueue any of these vertices that now have in-degree zero. — $\Theta(1)$

(n times)

Runtime? $\Theta(n+m)$ — m could be very large (n^2)

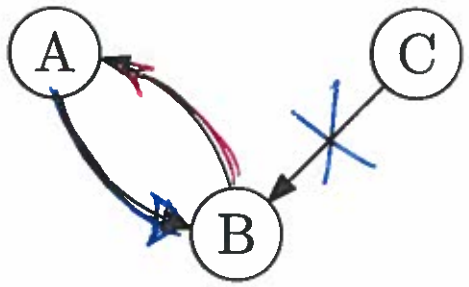
Graph ADT

A graph is a useful formalism for representing relationships among things.

A graph is represented as a pair of sets: $G = (V, E)$ where $|V| = n$ and $|E| = m$.

- V is a set of vertices: $\{v_1, v_2, \ldots, v_n\}$.
- E is a set of edges: $\{e_1, e_2, \ldots, e_m\}$ where each e_i is a pair of vertices: $e_i \in V \times V$.

Total order? No.
Partial order? Cycle → No.

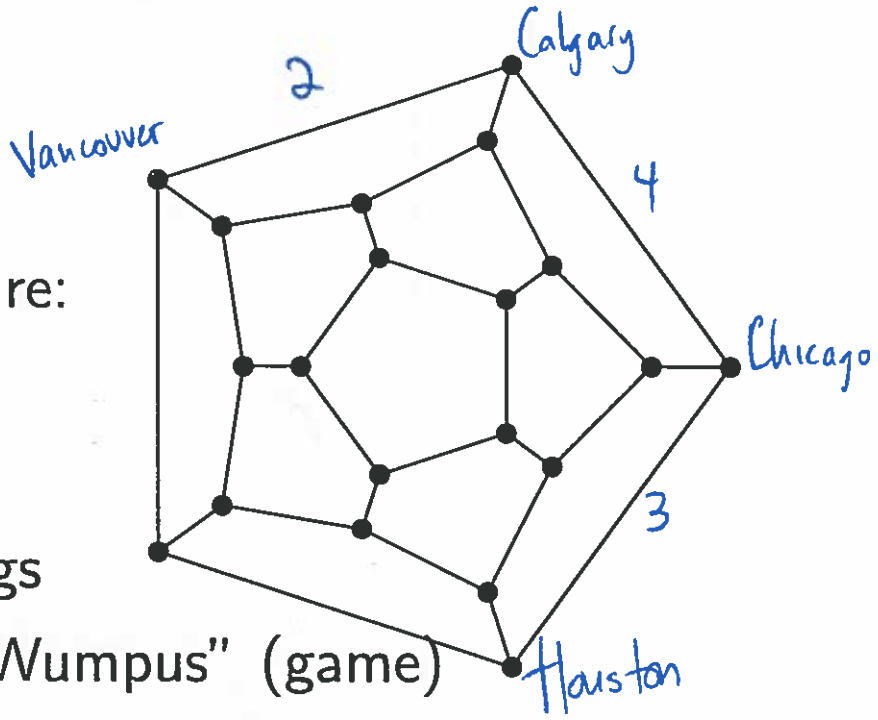


$V = \{A, B, C\}$
$E = \{(A, B), (B, A), (C, B)\}$

Operations may include:

- Create a graph (with a certain number of vertices).
- Insert or delete a given edge or vertex.
- Iterate over vertices adjacent to a given vertex.
- Ask if an edge exists that connects two given vertices.

Graph Applications



Storing things that are graphs by nature:

- Road networks
- Airline flights
- Relationships among people/things
- Room connections in "Hunt the Wumpus" (game)

Compilers:

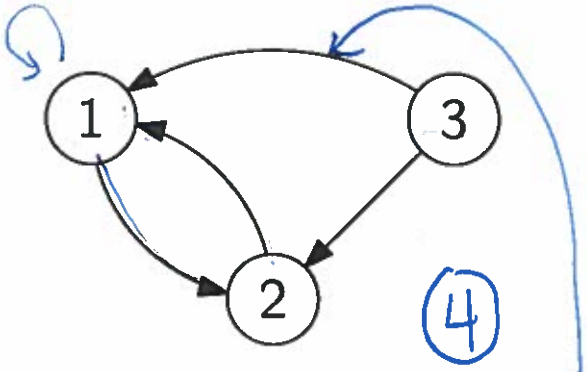
- Call graph - Which functions call other functions?
- Control flow graph - Which fragments of code can follow others?
- Dependency graphs - Which variables depend on others?

Others:

- Circuits, class hierarchies, meshes, networks of computers, ...

Graph Representation Using an Adjacency Matrix

A $|V| \times |V|$ array A where $A[u, v] = 1$ if and only if $(u, v) \in E$.



	1	2	3
1	0	1	0
2	1	0	0
3	1	1	0

edge from ③ to ①

n = # vertices = $|V|$

m = # edges = $|E|$

The runtime to:

- Iterate over n vertices is: $\Theta(n)$
- Iterate over m edges (in a graph with n vertices) is: $\Theta(n^2)$ — $\Theta(m)$
- Iterate over all vertices adjacent to a vertex v is: $\Theta(n)$ ⇒ go across a row (size: n)
- Check whether an edge (u, v) exists is: $\Theta(1)$

edge(3,1) → is there an edge from 3 to 1.

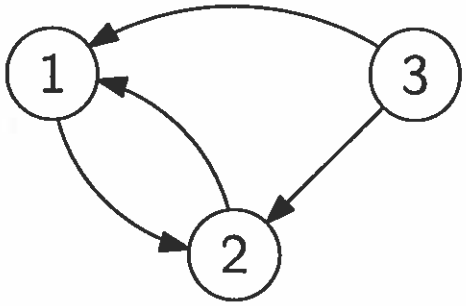
Memory requirements: $\Theta(n^2)$

A[3rd row down, 1st column]

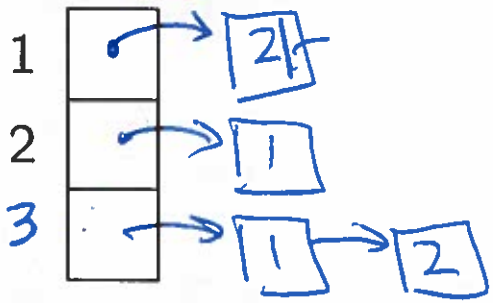
A[3,1] → * arrays are 0-based index *

Graph Representation Using an Adjacency List

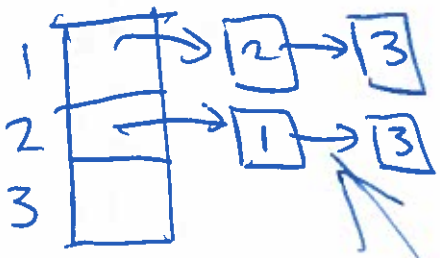
Adjacency List: An array L of $|V|$ lists, such that $L[u]$ contains v if and only if $(u, v) \in E$.



outgoing edges



incoming edges



more memory

The runtime to:

- Iterate over n vertices is: $\Theta(n)$
- Iterate over m edges is: $\Theta(n+m)$ → fast for sparse graphs
- Iterate over all vertices adjacent to a vertex v is: $\Theta(\text{outdegree}(v))$
- Check whether an edge (u, v) exists is: $\Theta(\text{outdegree}(u))$ $\Theta(\text{indegree}(v))$

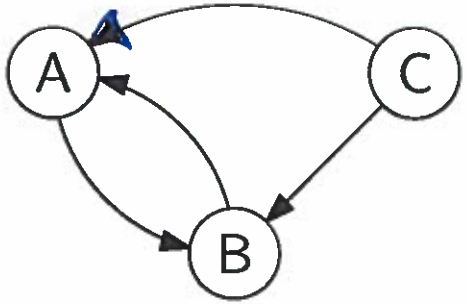
3 → 1

Memory requirements: $\Theta(n+m)$

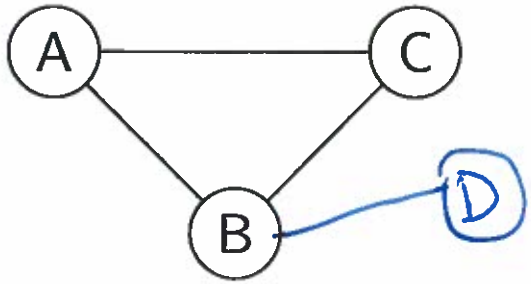
↳ plus same memory for nodes (pointers)

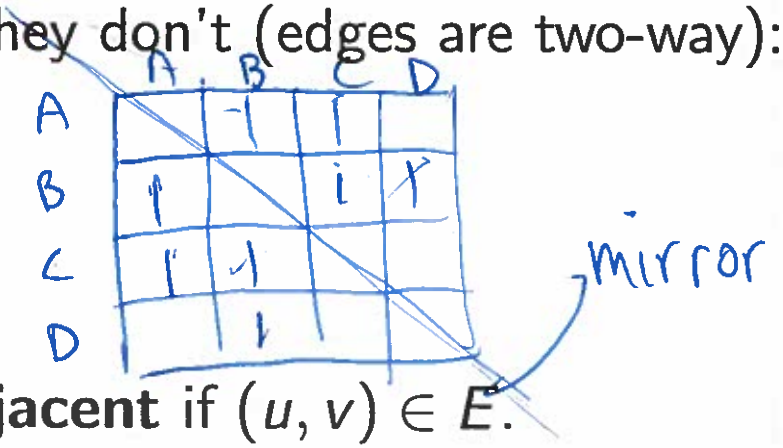
Directed vs. Undirected Graphs

In **directed** graphs, edges have a specific direction:



In **undirected** graphs, they don't (edges are two-way):

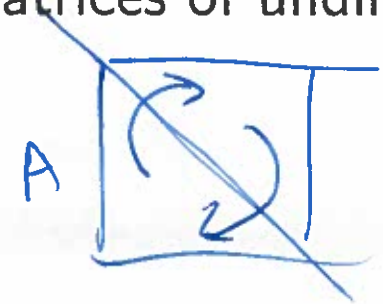




Vertices u and v are **adjacent** if $(u, v) \in E$.

What property do adjacency matrices of undirected graphs have?

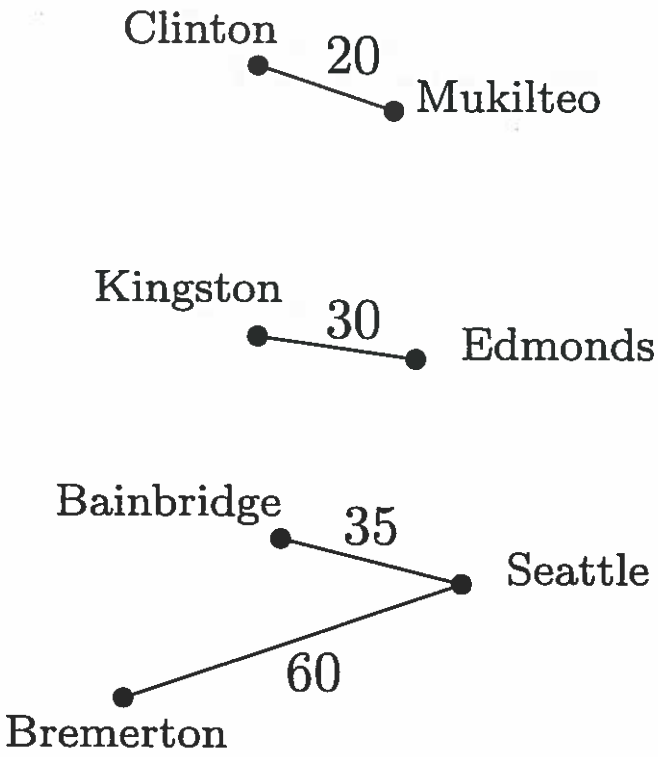
they are symmetric



$A[x, y] = A[y, x]$

Weighted Graphs

Each edge has an associated weight or cost. For example:



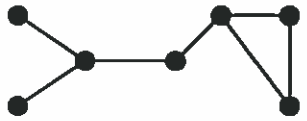
How can we store weights in an adjacency matrix?

In an adjacency list?

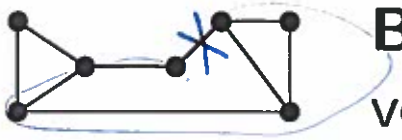
data type includes vertex value/key plus edge weights.

Graph Connectivity

• - vertex
— edges



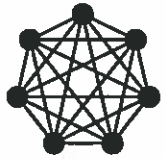
Connected: undirected and there is a path between any two vertices.



Biconnected: connected even after removing one vertex.

Strongly connected: directed and there is a path from any one vertex to any other.

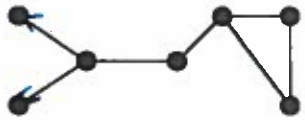
Weakly connected: directed and there is a path between any two vertices, ignoring direction.

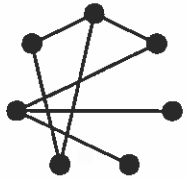


Complete graph: an edge between every pair of vertices

Isomorphism and Subgraphs

Isomorphic: Two graphs are isomorphic if they have the same structure (ignoring vertex names).

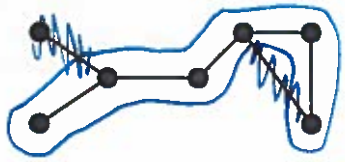




if we reorganize what this graph looks like, it's equivalent

$G_1 = (V_1, E_1)$ is isomorphic to $G_2 = (V_2, E_2)$ if there is a one-to-one and onto function (i.e., bijection) $f : V_1 \to V_2$ such that $(u, v) \in E_1$ iff $(f(u), f(v)) \in E_2$.

Subgraph: One graph is a subgraph of another if it is some part of the other graph.



$G_1 = (V_1, E_1)$ is a subgraph of $G_2 = (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

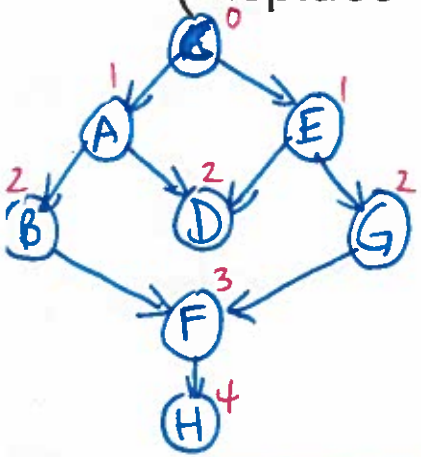
Note: We sometimes say that H is a subgraph of G if H is isomorphic to a subgraph (in the above sense) of G.

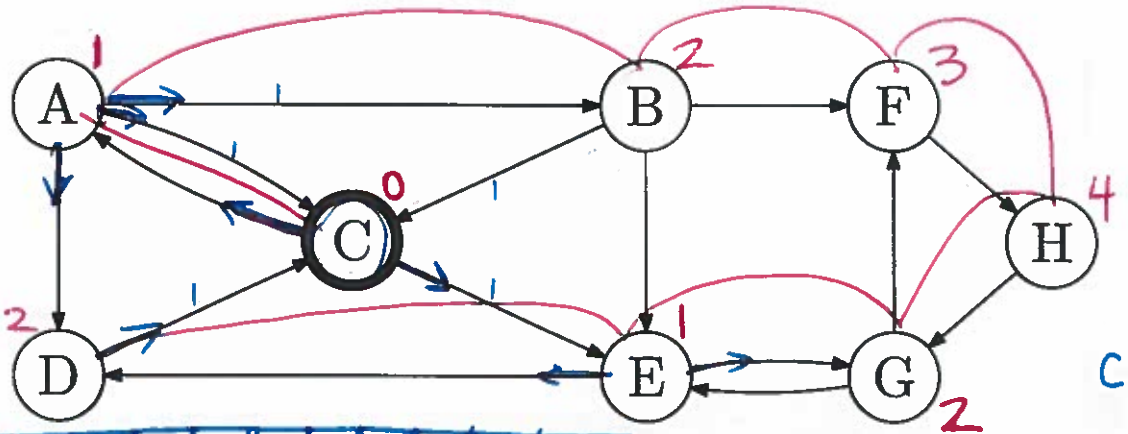
Unweighted Single-Source Shortest Path Problem

radiate outwards ← graph ← starting vertex (c)

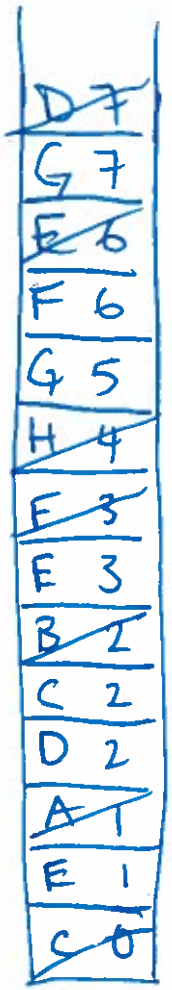
```
BreadthFirstSearch(G, s)
    Q.enqueue([s,0])
    while Q is not empty:
        [v,d] = Q.dequeue()
        if v is unmarked:
            mark v with distance d
            for each edge (v,w):
                Q.enqueue([w,d+1])
```

(Replace the queue with a stack to get a depth-first search.)





Q: C A E D C B D G C F E E F H G
0 1 1 2 2 2 2 2 3 3 3 3 3 4 5



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

General Breadth-First Search (BFS) Algorithm

```
BFS(v) using a starting vertex s in graph G:

  Add s to queue.
  While queue not empty:
       Dequeue vertex v.
       Process (e.g., print) v.  (If in search mode,
          return the information when the target is
          found, and terminate the algorithm.)
       Enqueue all unvisited neighbours of v.
```

We need to mark each vertex as being visited (so far), or not.

For a directed graph, u is a neighbour of v if (v, u) is an edge.

Application: Model a maze as a graph, and use BFS to find the shortest path/solution. (Koffman, p. 727+).

General Depth-First Search (DFS) Algorithm

```
DFS(v) using a starting vertex v in graph G:
    Call DFS(v)

DFS(v):
  Process (e.g., print) v.  (If in search mode, return
        the information when the target is found, and
        terminate the algorithm.)
  For each unvisited neighbour c of v:
       DFS(c)
```

We need to mark each vertex as being visited (so far), or not.

Application: Model a course prerequisite chart as a graph, and perform a topological sort (see Koffman, p. 731+).

Application: Solve a maze.