CMPE 180A DSA with Python

Graphs and graph traversals

Sanja Damjanovic

Week 8

09/02/2025

Announcements & Info

- The final exam duration: 150 minutes, ~ 30 questions, Week 10
  - Questions of the same type as in the midterm
  - All material included
  - Questions will include also: explain what a function does and what it returns, recognize algorithms, e.g. recognize binary search and its versions
- HW3 & 4: reading file from Google drive (see example in Week 8 Lab)
- Project:
  - Example Classification in Module Week 8
  - Main task is to use Pandas to clean the data set and matplotlib to visualize histograms and graphs as shown in Week 5 Labs

Week 8 Outline

- Graphs:
  - Terminology
  - Building a graph
- Graph traversal algorithms:
  - Breath First Search (BFS)
  - Depth First Search (DFS)
- BFS and DFS application
  - General DFS
  - Path searching using DFS and BFS

Graphs

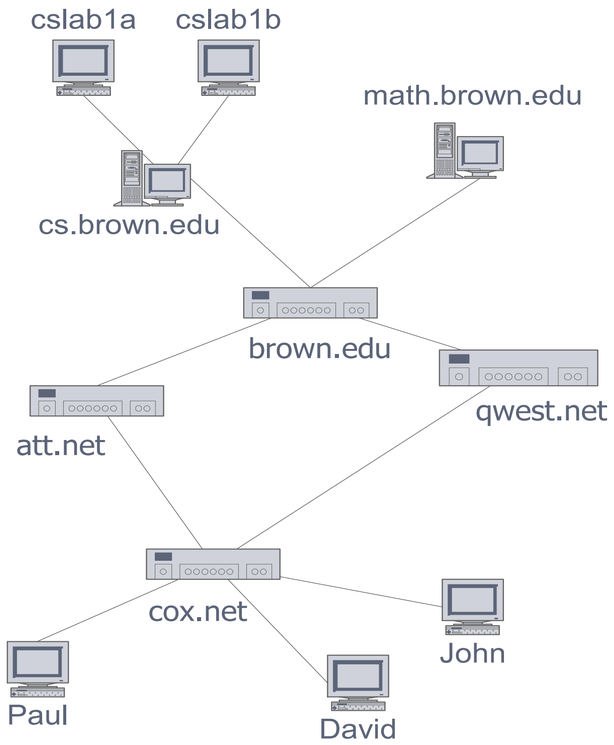
Terminology Overview

- Vertex
- Edge:
  - Undirected
  - Directed (Source,Sink/Destination)
  - Weight
- Path
- Cycle
- Connected component

Graph

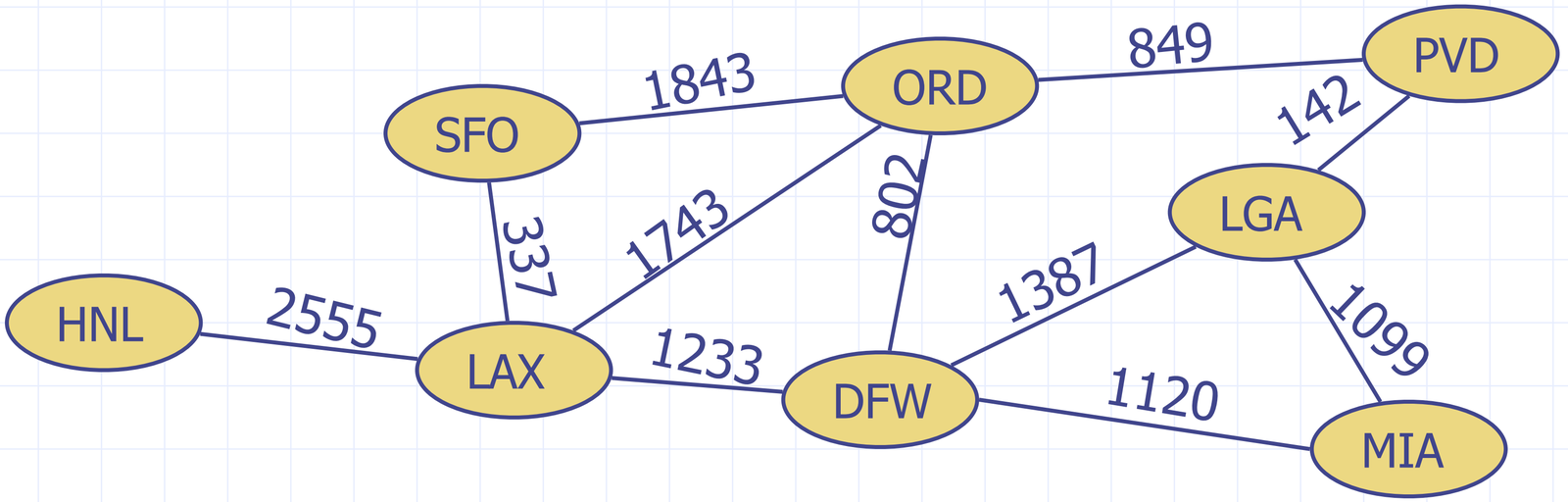
Applications

- social network of friends
- computer network
  - local area networks
  - Internet
- transportation network
  - highway network
  - flight network



Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



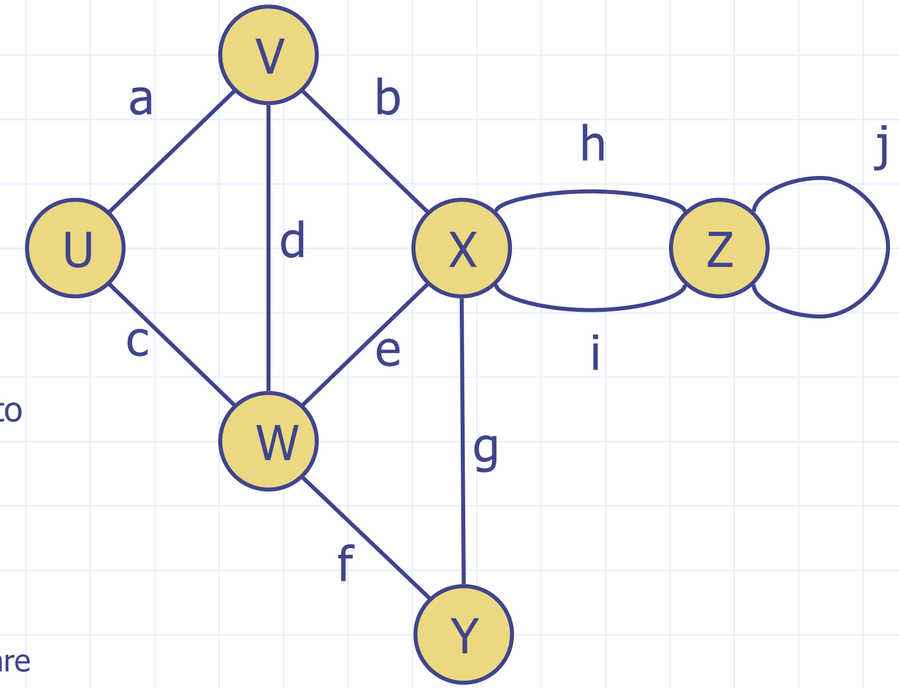
Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network

PVD: Providence, Kent County, Rhode Island
ORD: Orchard Field Airport, Chicago

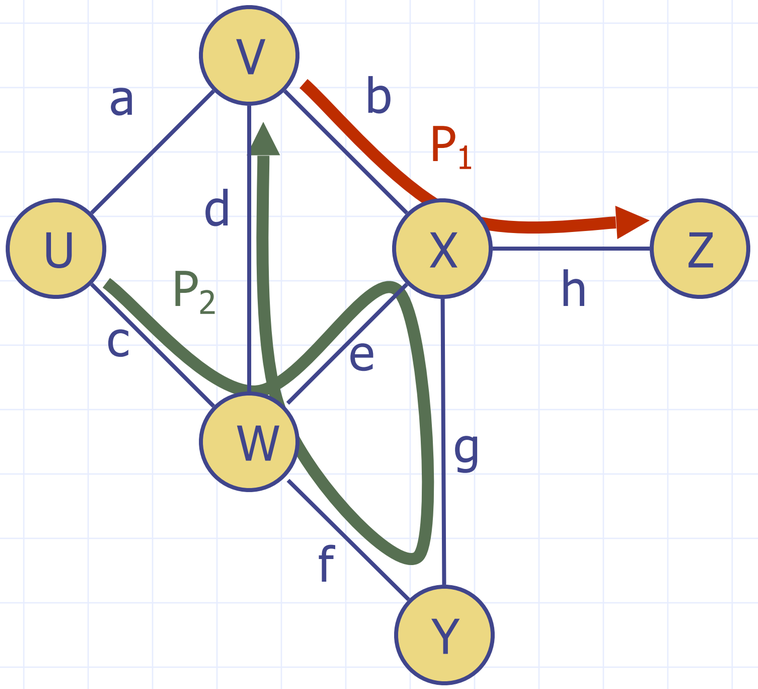
Terminology

- **End vertices (or endpoints)** of an edge
  - the two vertices that are connected by an edge
    - **U** and **V** are the **endpoints** of the edge **a**
- Edges incident on a vertex
  - the number of edges that are incident (connected) to that vertex
    - **a**, **d**, and **b** are **incident** on V
- **Adjacent vertices (neighbors)**
  - two vertices that are connected directly by an edge
    - U and V are adjacent
- **Degree of a vertex**
  - the total number of edges that are connected to the vertex
    - X has degree 5
    - What is the degree of an isolated vertex?
    - Directed graph: in- & out- degree
- **Parallel edges**
  - h and i are parallel edges
  - Example: a transportation network where two cities are connected by multiple highways
- **Self-loop**
  - j is a self-loop
  - What is the degree of vertex **z**?



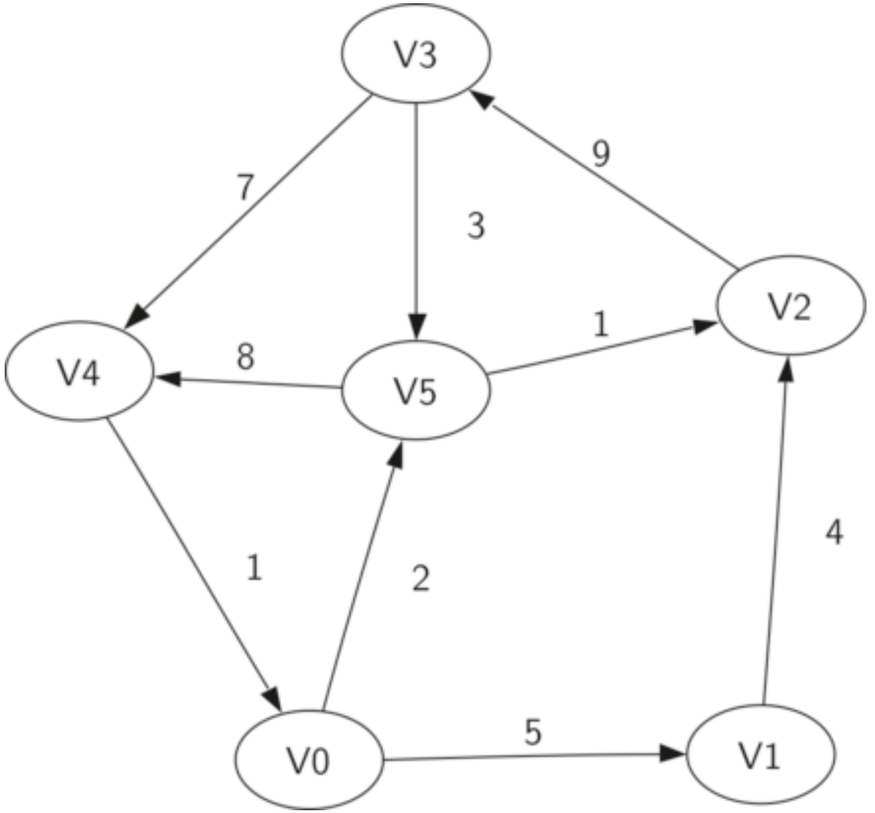
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple

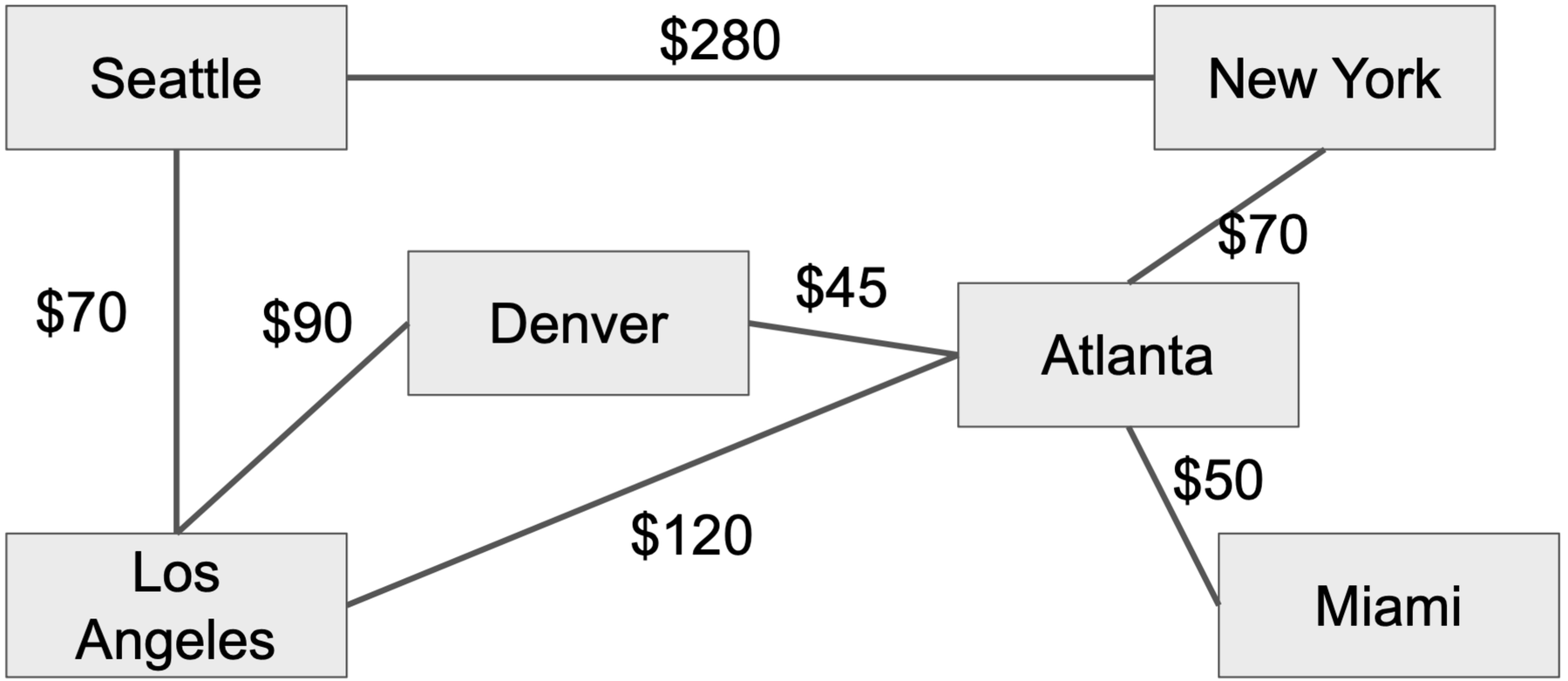


Graph

- Undirected or directed?
- Weighted or unweighted?
- #V=?
- #E=?



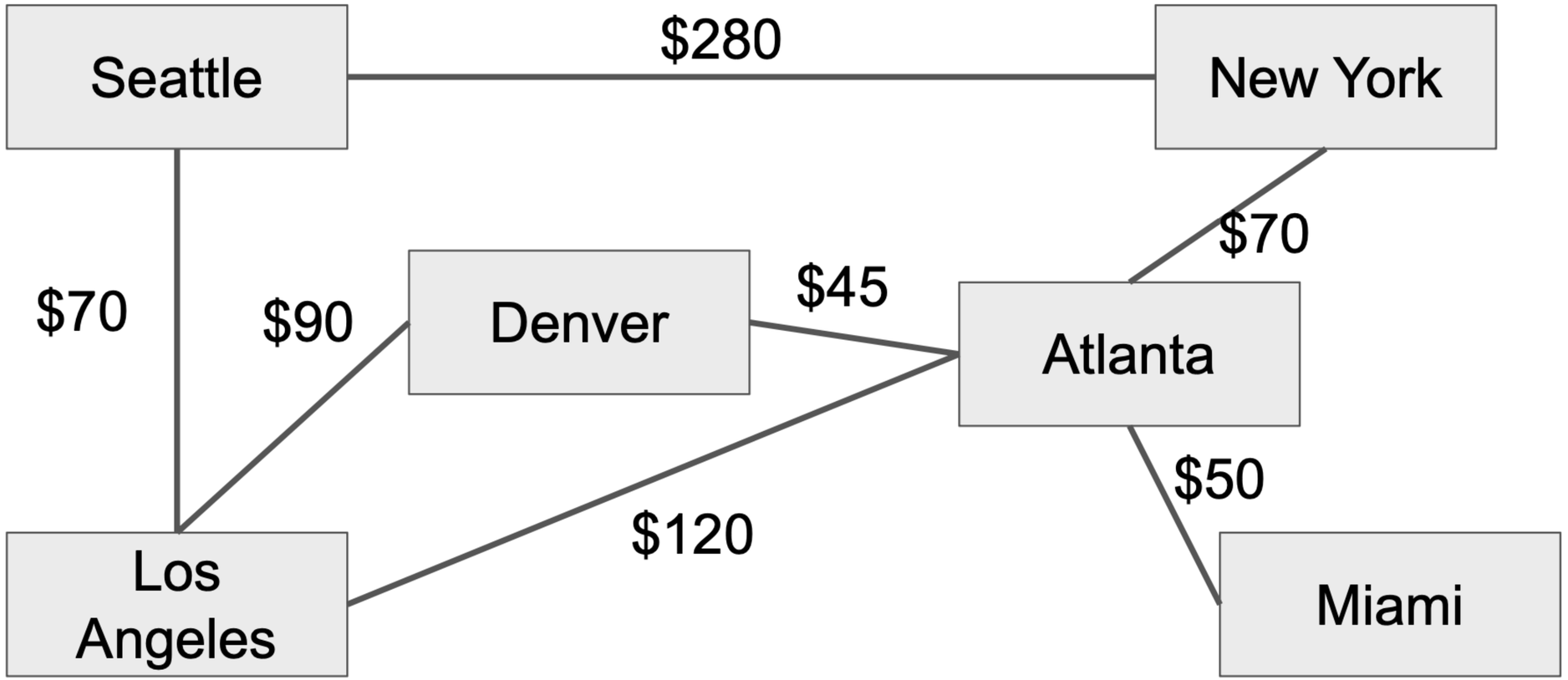
Graphs



What type of a graph is this?

What is the cheapest way to get to Miami from LA?

Graphs



Undirected, weighted

Types of graphs regarding the no. of E

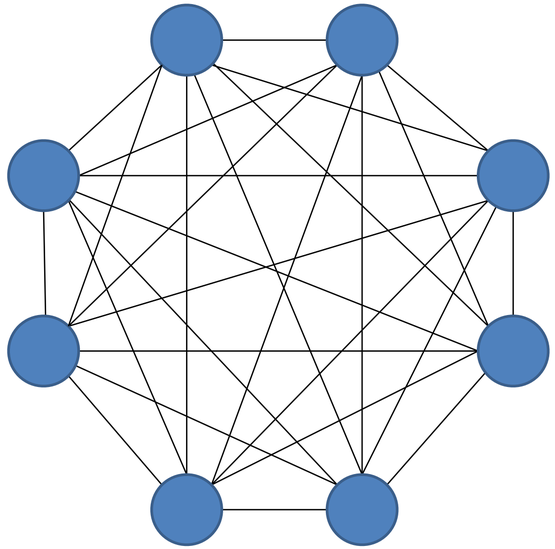
- **Dense**: a large number of edges, close to maximum
  - Q1: A graph consists of V vertices. What is the maximum no of edges in
    - (a) an undirected graph
    - (b) a directed graph?
- **Sparse**: small number of edges, close to minimum
  - Q2: What is the minimum if the graph has V vertices?
- **Complete**: no of edges is equal to the maximum

Types of graphs regarding the no. of E

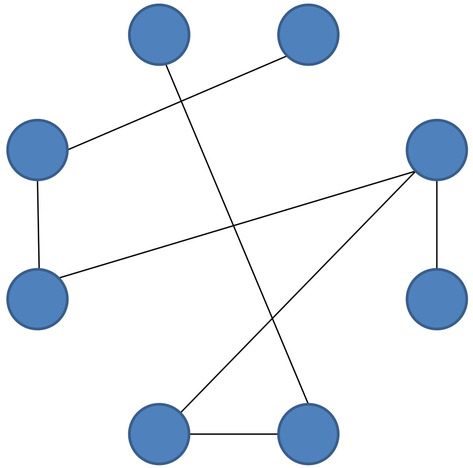
- **Dense**: a large number of edges, close to maximum
  - Q1: What is the maximum no of edges in a graph with V vertices?
    - undirected **V*(V-1)/2**
    - directed **V*(V-1)**
- **Sparse**: small number of edges, close to minimum
  - Q2: What is the minimum if the graph has V vertices? **V-1**
- **Complete**: no of edges is equal to the maximum

Dense and Sparse Graphs

- **Dense Graph –**
  graph where $|E| \sim |V|^2$



- **Sparse Graph –**
  graph where $|E| \sim |V|$



Graph Implementations

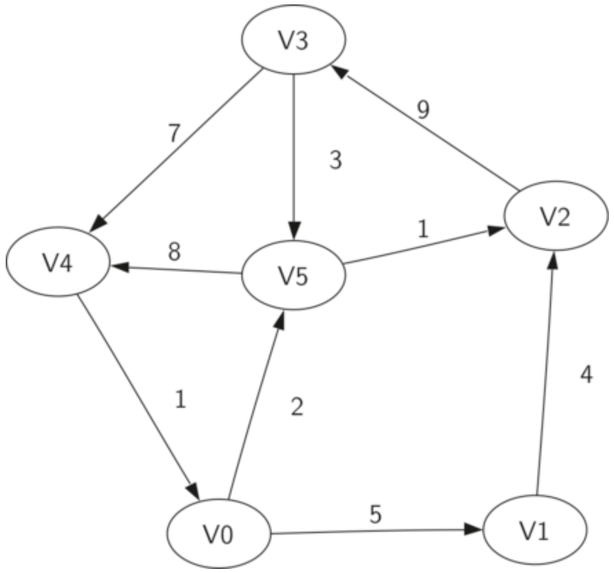
- Adjacency matrix
- Adjacency list

An Adjacency Matrix Representation for a Graph

Directed graph

- The value that is stored in the cell at the intersection of row and column indicates if there is an edge from vertex to vertex . When two vertices are connected by an edge, we say that they are **adjacent**.

- Sparse or dense graphs?

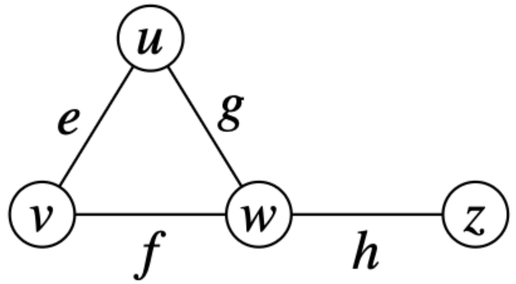


	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

An Adjacency Matrix Representation for a Graph

Undirected graph

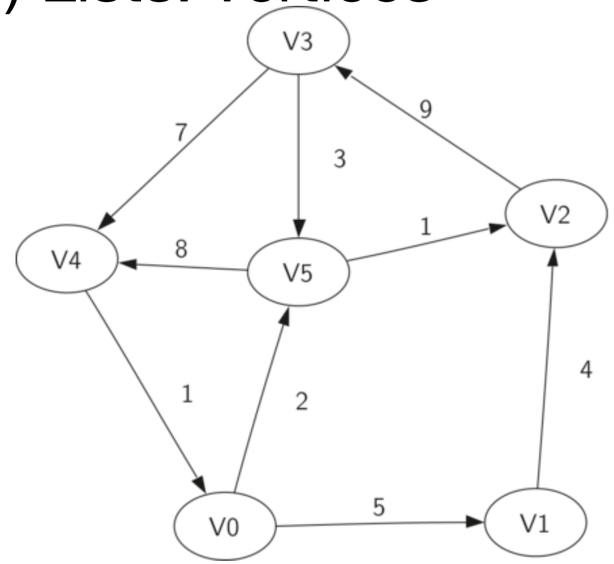
- e.g. edge with weight *e* connects vertices in both directions *u* with *v* and *v* with *u*

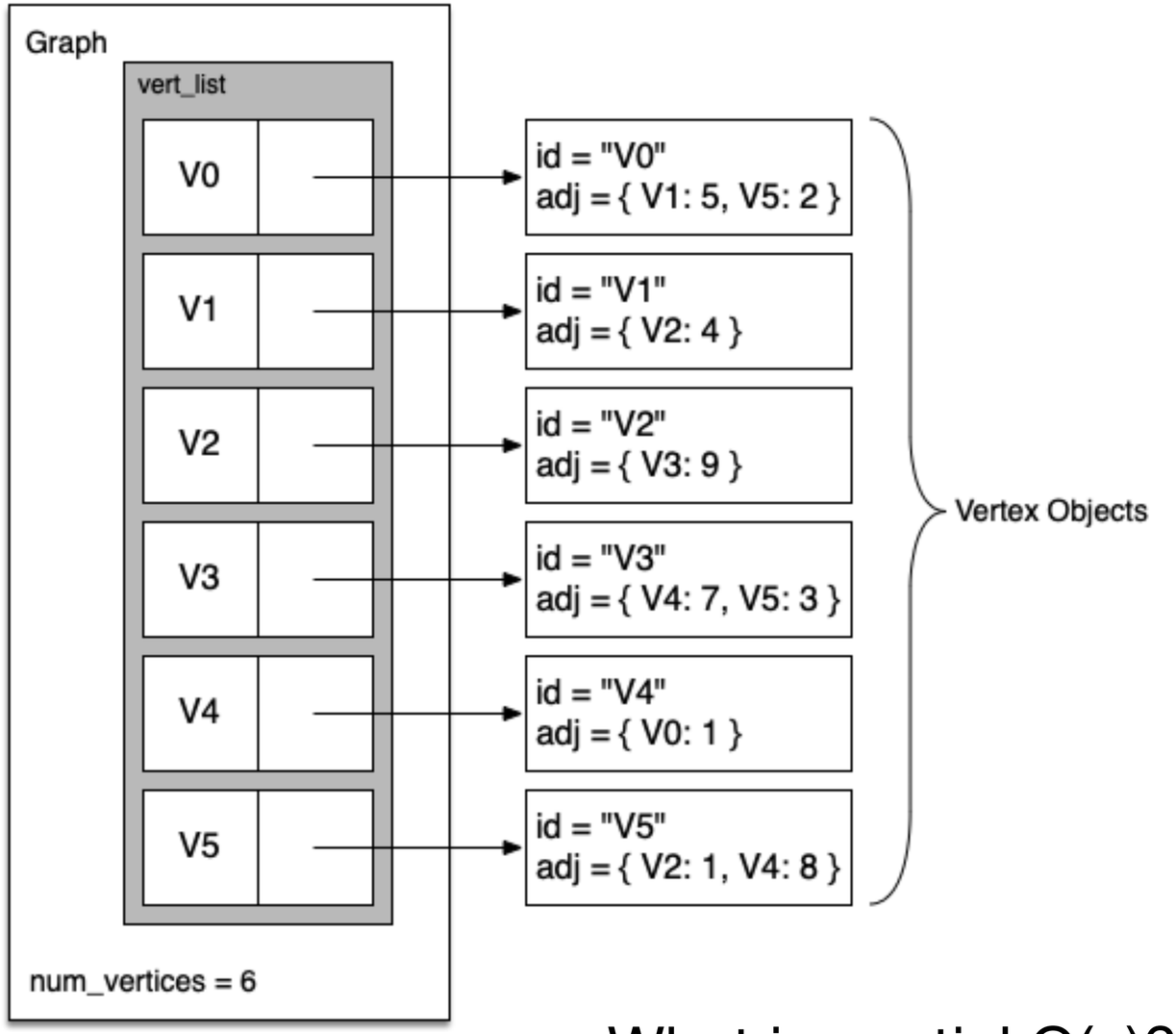


		0	1	2	3
u	0		*e*	*g*	
v	1	*e*		*f*	
w	2	*g*	*f*		*h*
z	3			*h*	

An Adjacency List Structure

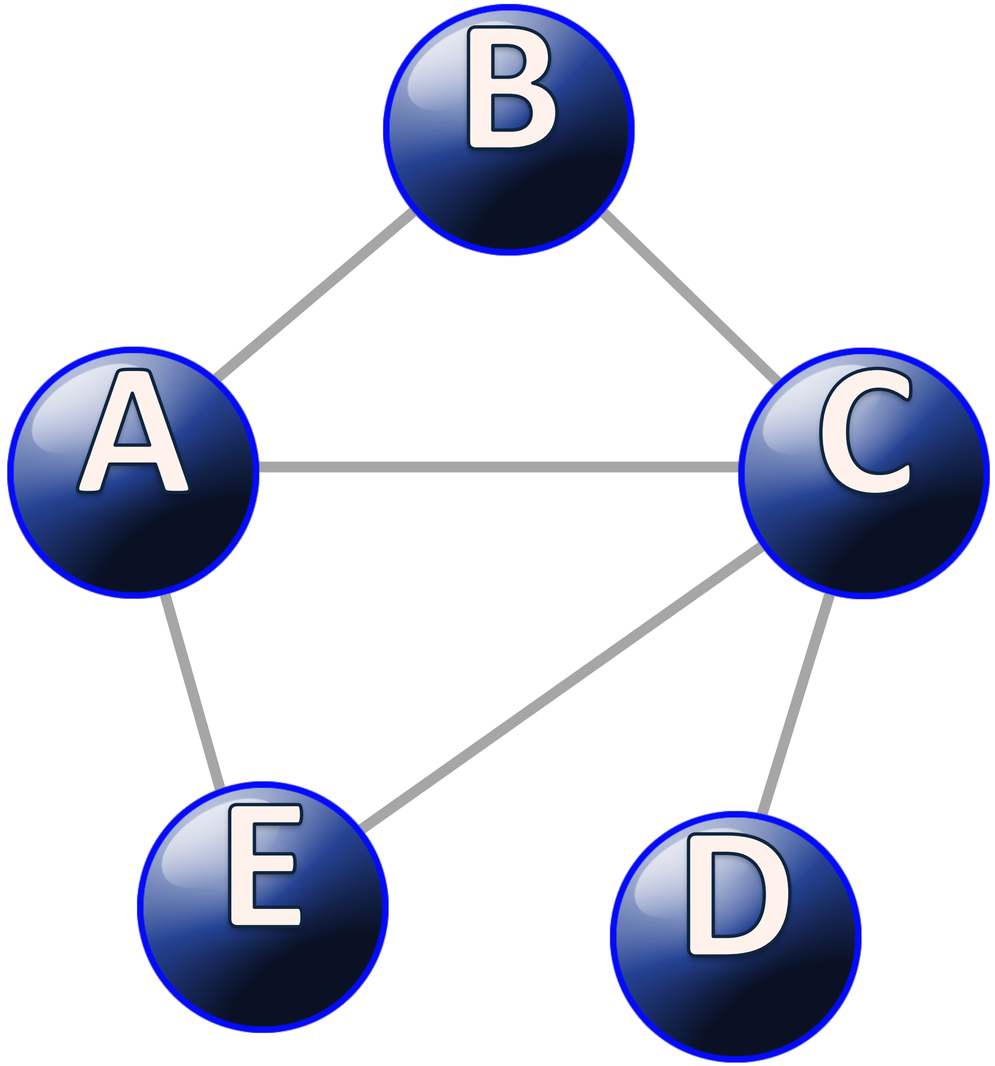
- Which data structures can be used to implement adjacency list?
- A) Dictionaries: incidence (incoming) collection with weights
- B) Lists: vertices





What is spatial O(n)?

Undirected Graph



Create Adjacency List

A: B, C, E

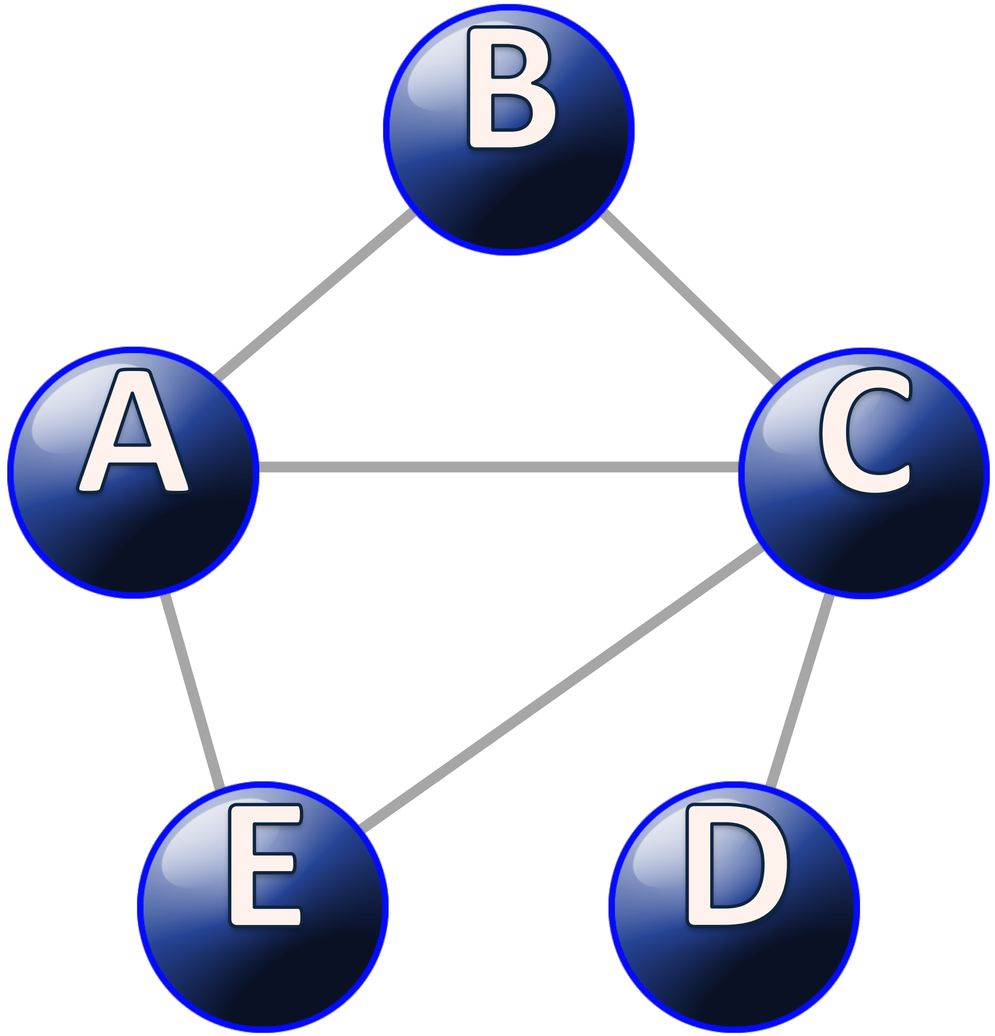
B:

C:

D:

E:

Undirected Graph



Adjacency List

A: B, C, E

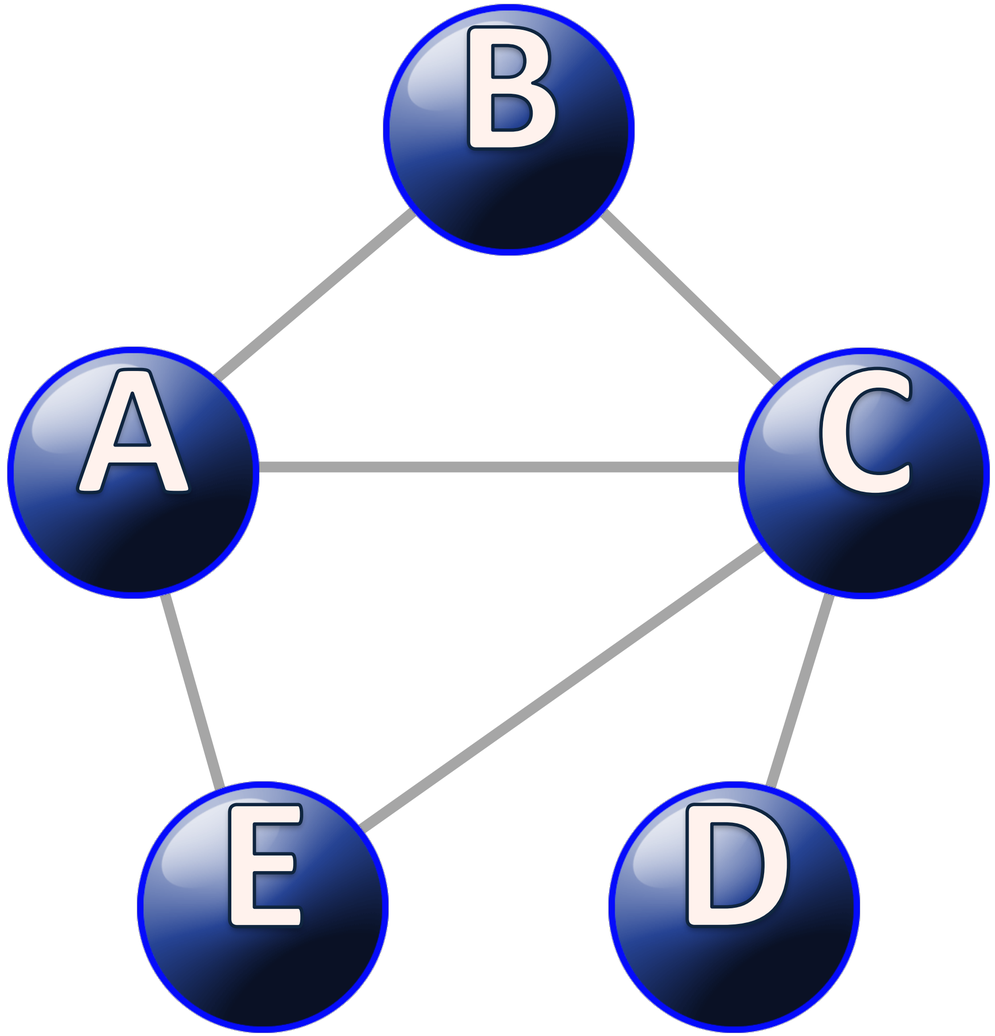
B: A, C

C: A, B, D, E

D: C

E: A, C

Undirected Graph



Adjacency List

A: B, C, E

B: A, C

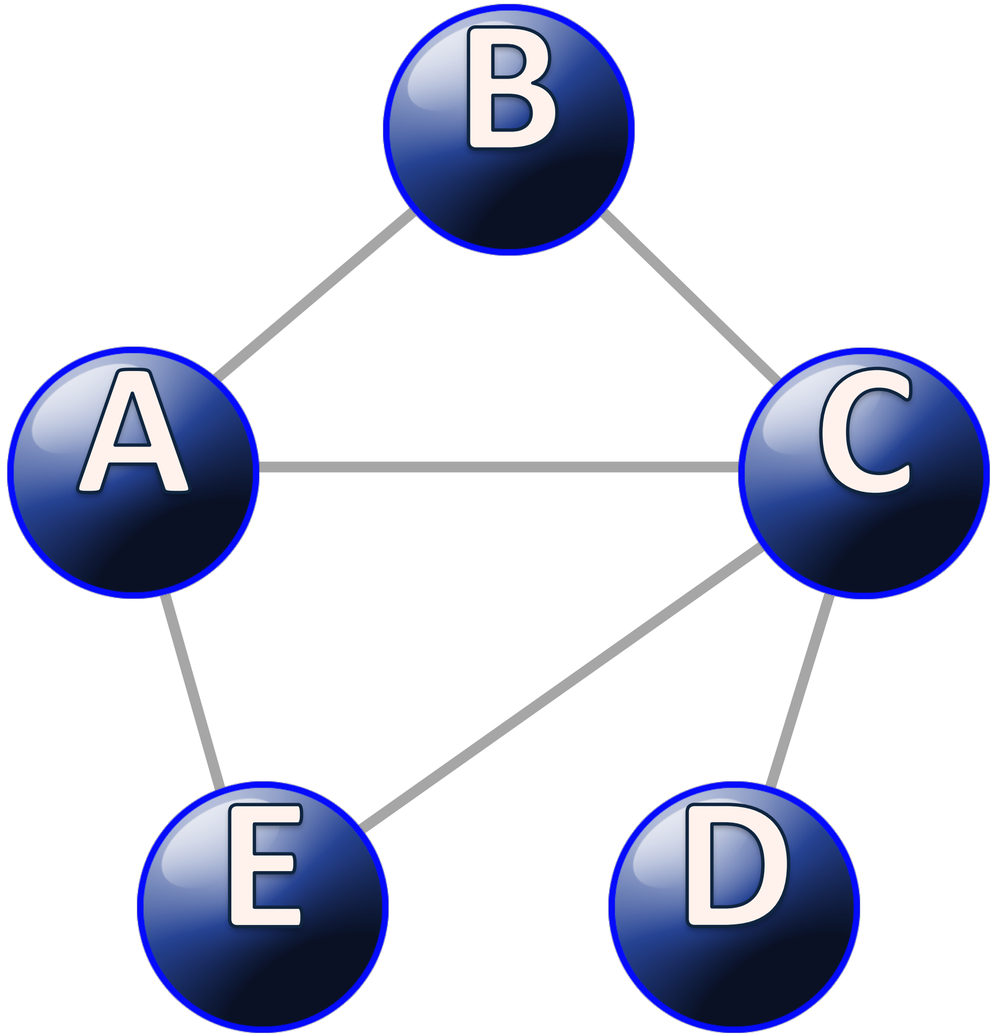
C: A, B, D, E

D: C

E: A, C

Stored in Node A

Undirected Graph



Adjacency List

A: B, C, E

B: A, C

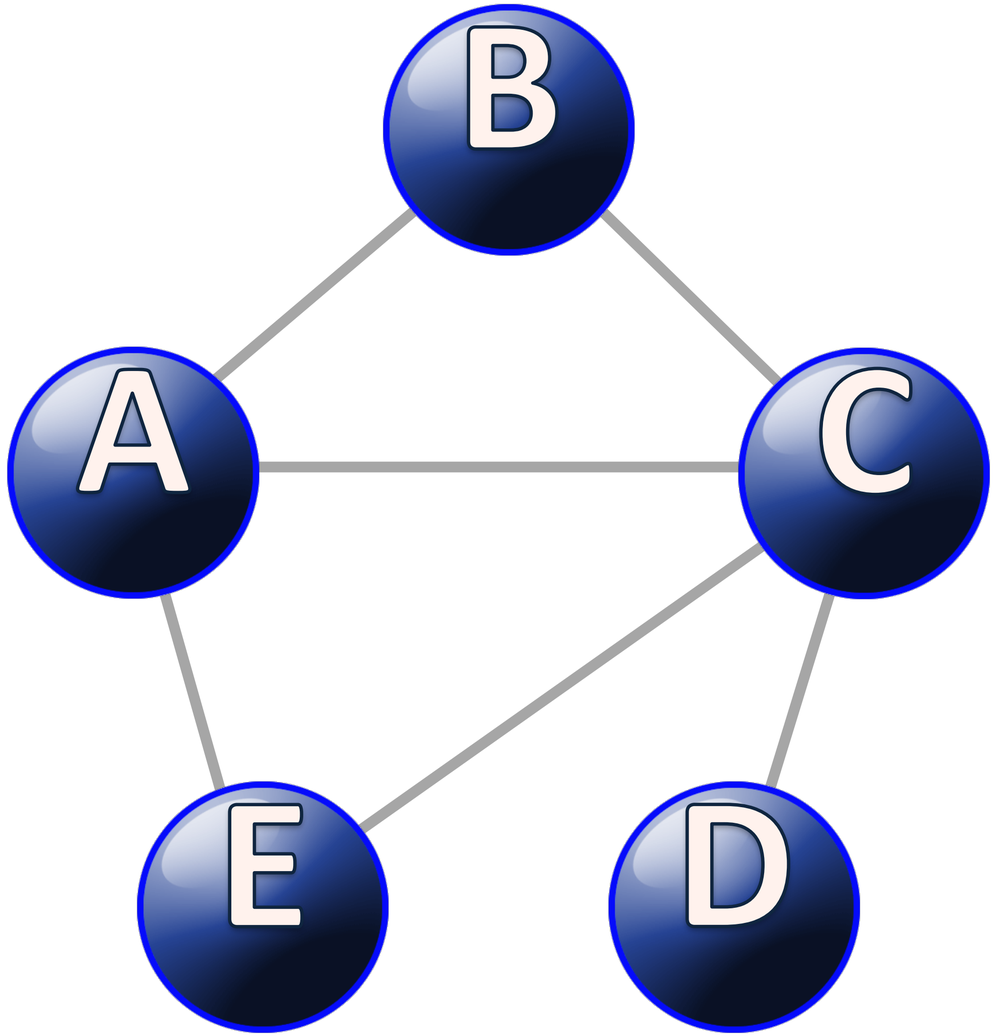
C: A, B, D, E

D: C

E: A, C

Stored in Node B

Undirected Graph



Fill in the Adjacency Matrix:

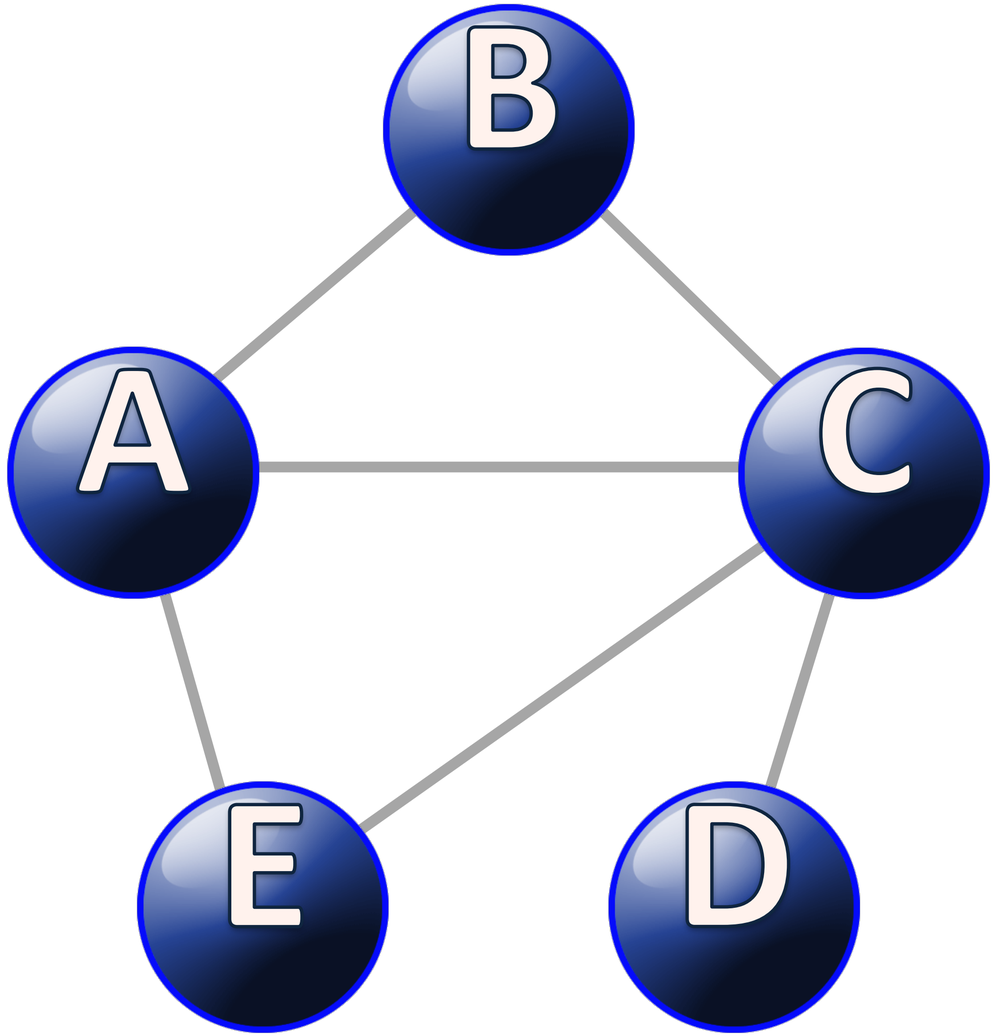
1 - connected, 0- not connected

to

from	A	B	C	D	E
A					
B					
C					
D					
E					

Stored in Graph

Undirected Graph



Adjacency Matrix

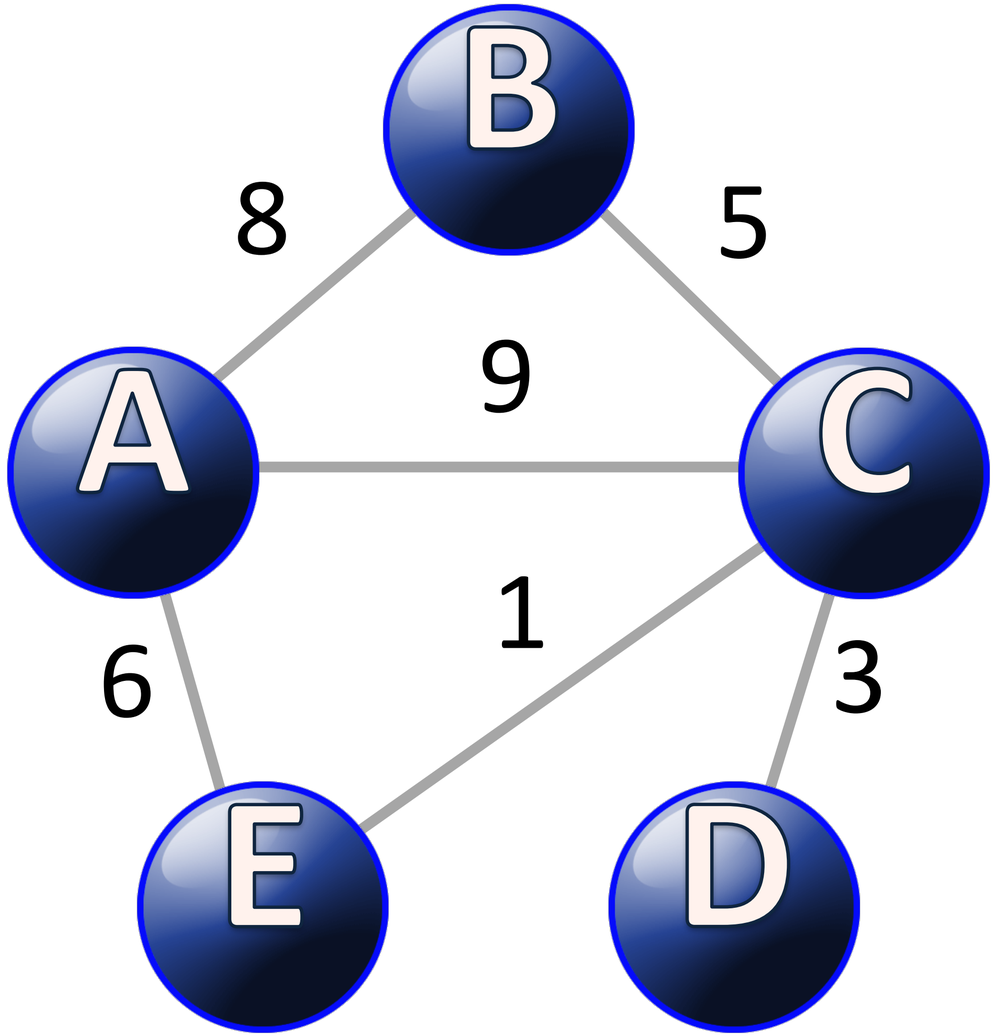
to

from

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

Stored in Graph

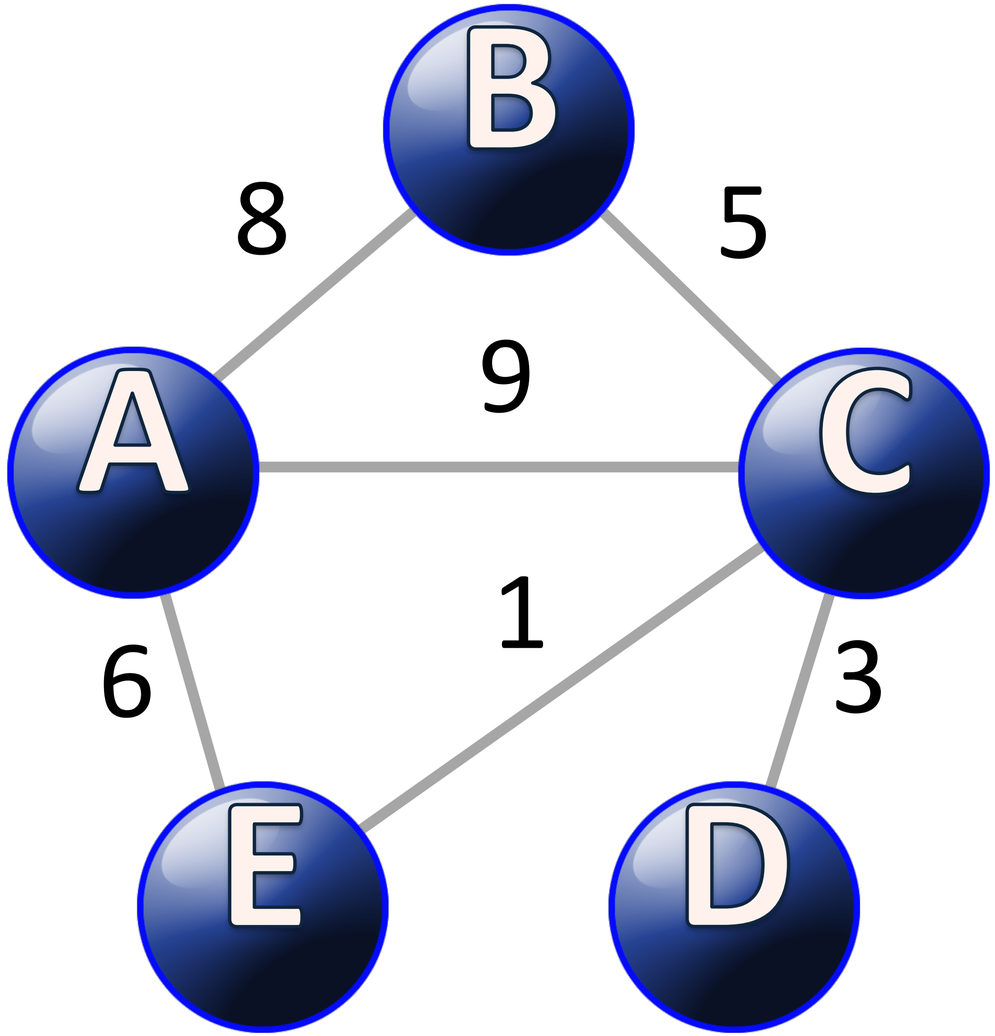
Undirected Graph



Weighted Edges?

Much easier to implement
with
Adjacency Matrix

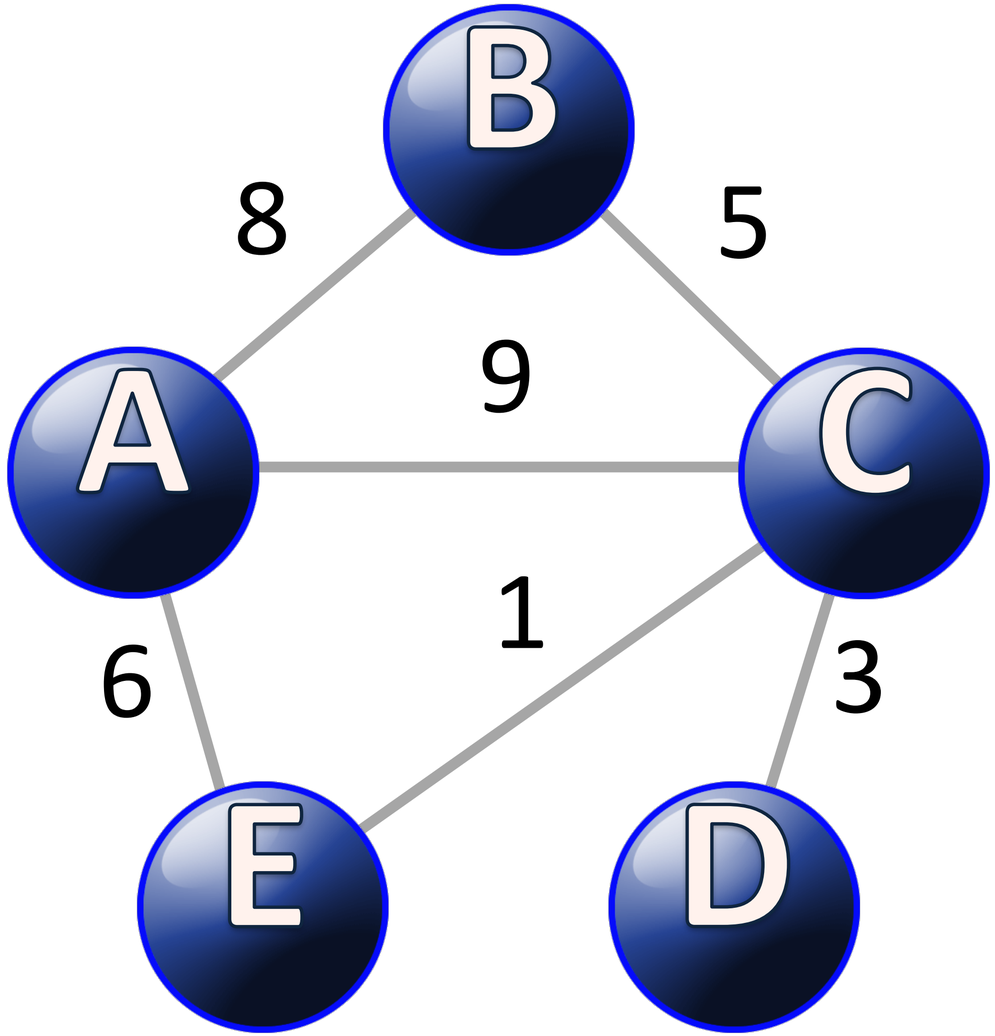
Undirected Weighted Graph



Adjacency Matrix

	A	B	C	D	E
A					
B					
C					
D					
E					

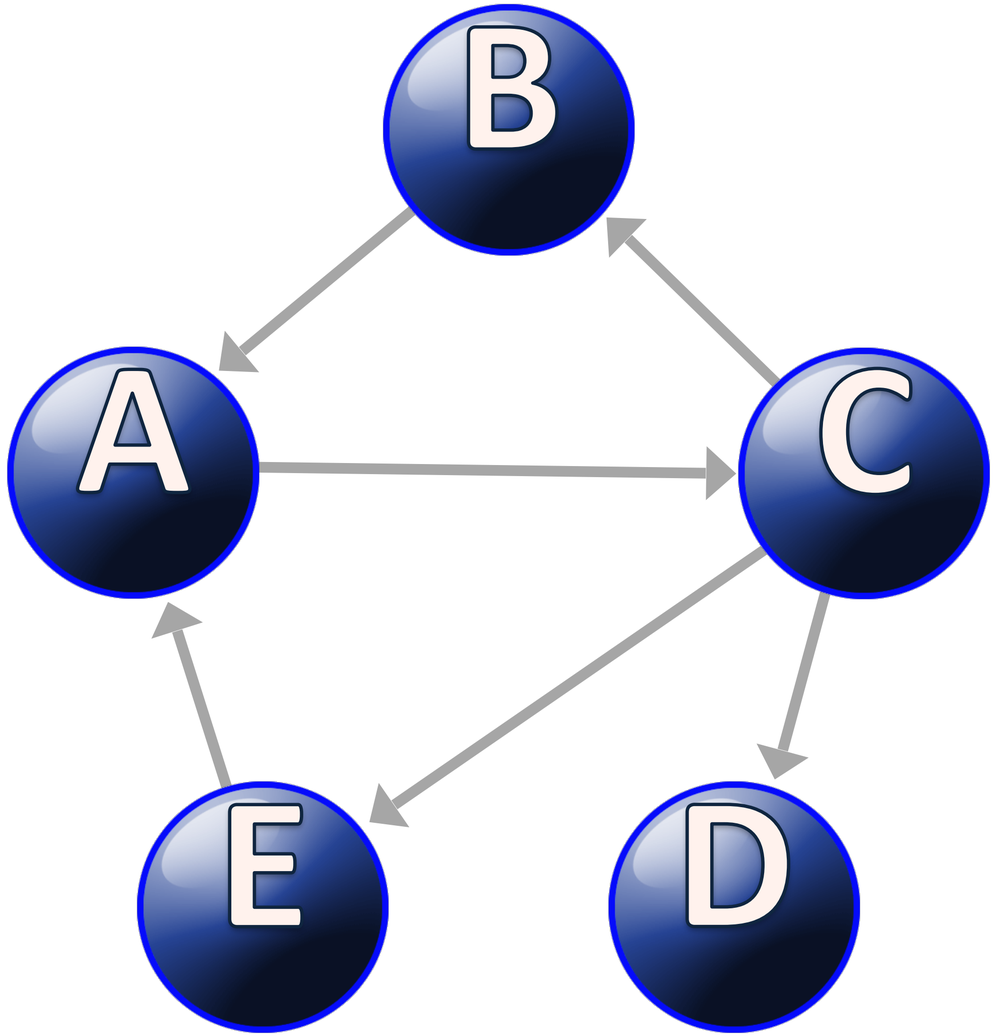
Undirected Weighted Graph



Adjacency Matrix

	A	B	C	D	E
A	0	8	9	0	6
B	8	0	5	0	0
C	9	5	0	3	1
D	0	0	3	0	0
E	6	0	1	0	0

Directed Graph



Create Adjacency List

A:

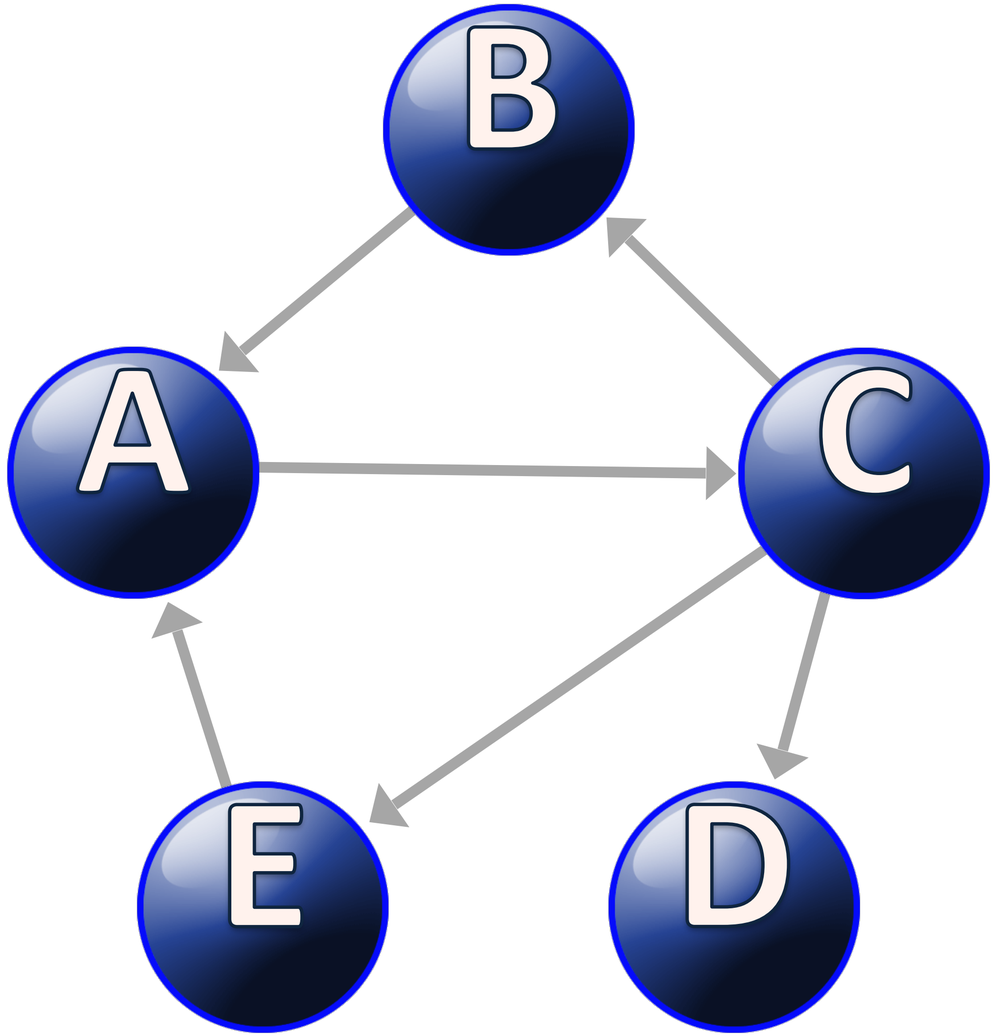
B:

C:

D:

E:

Directed Graph



Adjacency List

A: C

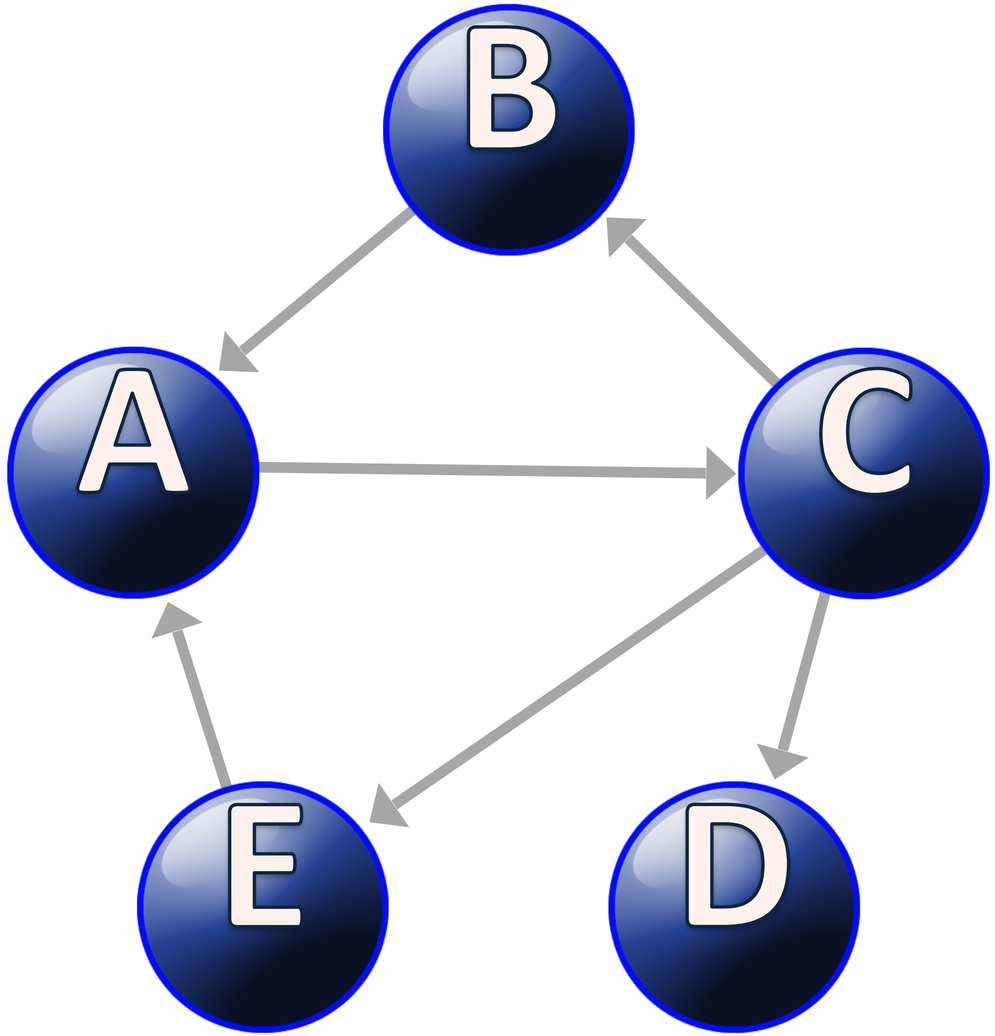
B: A

C: B, D, E

D:

E: A

Directed Graph



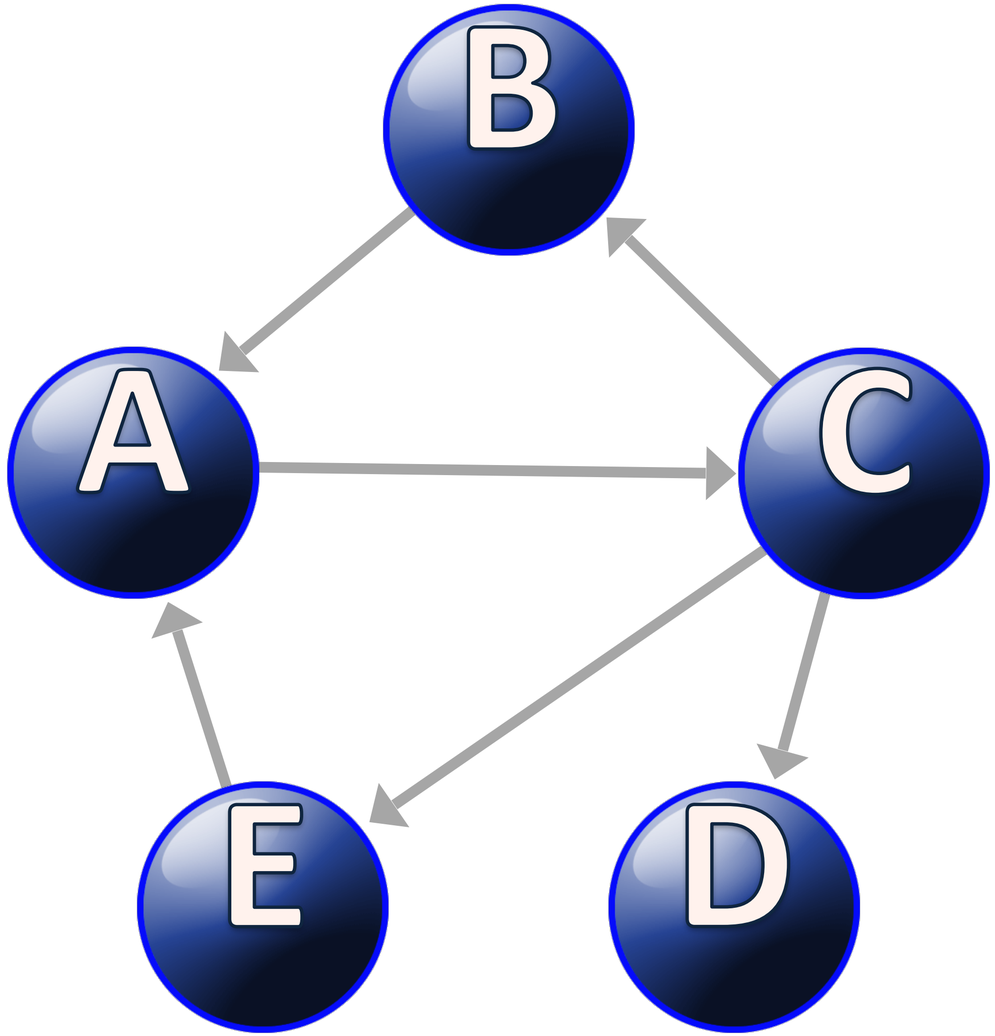
Create Adjacency Matrix

to

from

	A	B	C	D	E
A					
B					
C					
D					
E					

Directed Graph



Adjacency Matrix

to

from

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

Graph

Abstract Data Type

- Graph() : creates a new empty graph
- `add_vertex(vert)` adds an instance of `Vertex` to the graph
- `add_edge(from_vert, to_vert)` adds a new directed edge to the graph that connects two vertices
- `add_edge(from_vert, to_vert, weight)` adds a new weighted directed edge to the graph that connects two vertices
- `get_vertex(vert_key)` finds the vertex in the graph named `vert_key`.
- `get_vertices()` returns the list of all vertices in the graph.
- `in` returns `True` for a statement of the form `vertex in graph` if the given vertex is in the graph, `False` otherwise.

Graph Traversal

Graph Traversal

- A traversal is a systematic procedure for exploring a graph by examining all of its vertices and edges
- The process of visiting each vertex in a graph
- A traversal is efficient if it visits all the vertices and edges in time proportional to their number:
  - in linear time ~O(|V|+|E|)
- Traversal algorithms:
  - Breadth first search (BFS):
    - visiting a sibling before visiting children
    - Implementation: Adding children into a queue
  - Depth first search (DFS):
    - visiting a child before siblings
    - Implementation: Adding children into a stack

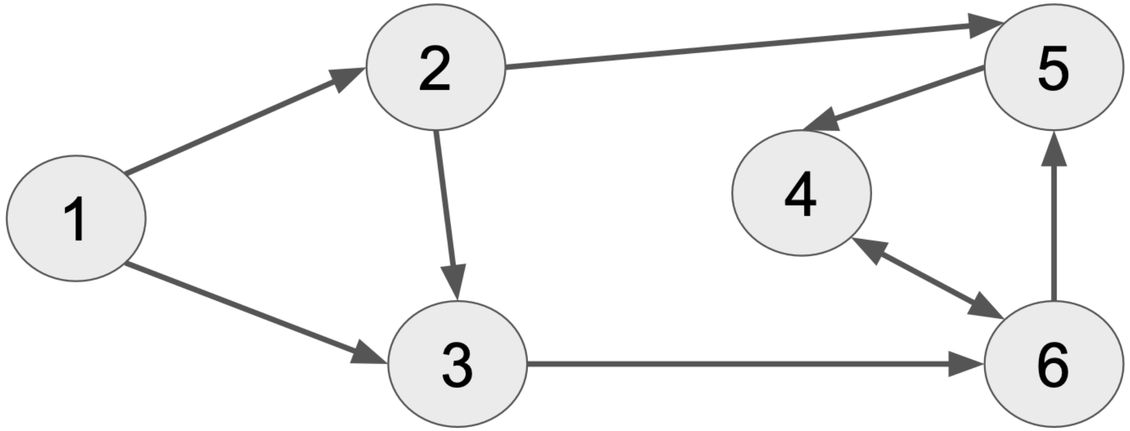
Breath First Search (BFS)

BFS

- Algorithm BFS(G, s):
  - Input: A graph G and a starting vertex s of G
  - Output: A collection of vertices reachable from s, with their discovery edges
    - Initialize an empty queue Q
    - Mark vertex s as visited
    - Enqueue s into Q

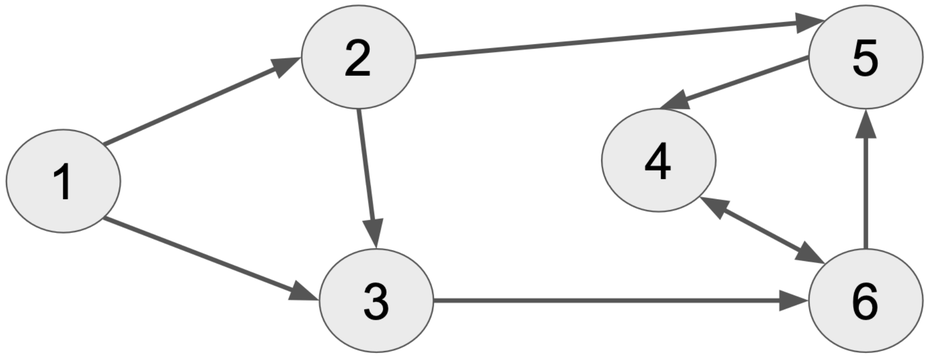
```
while Q is not empty do
        u ← Dequeue(Q)
        for each outgoing edge e = (u, v) of u do
          if vertex v has not been visited then
                Mark vertex v as visited (via edge e)
                Enqueue v into Q
```

Breadth-first search



bfs(node=1, graph)

Breadth-first search

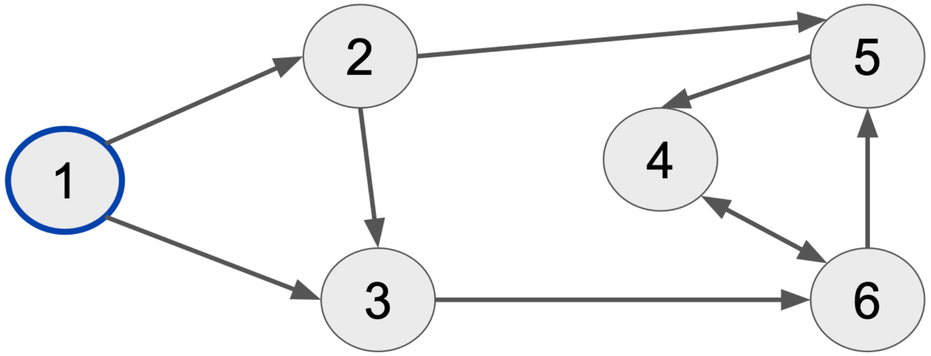


bfs(node=1, graph)

front	rear

Queue

Breadth-first search

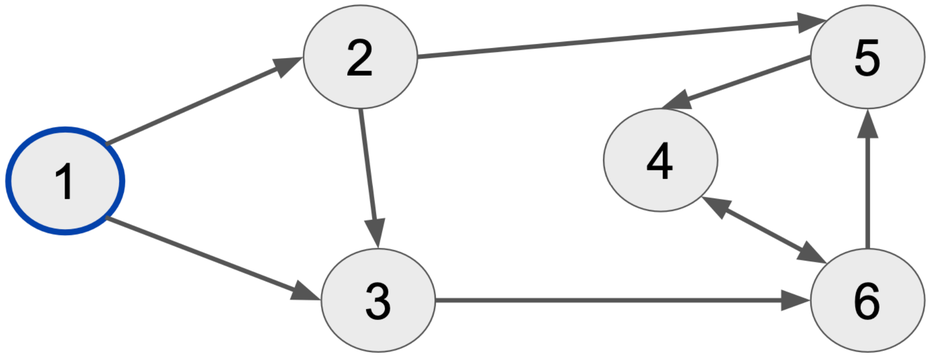


bfs(node=1, graph)

front	rear

Queue

Breadth-first search

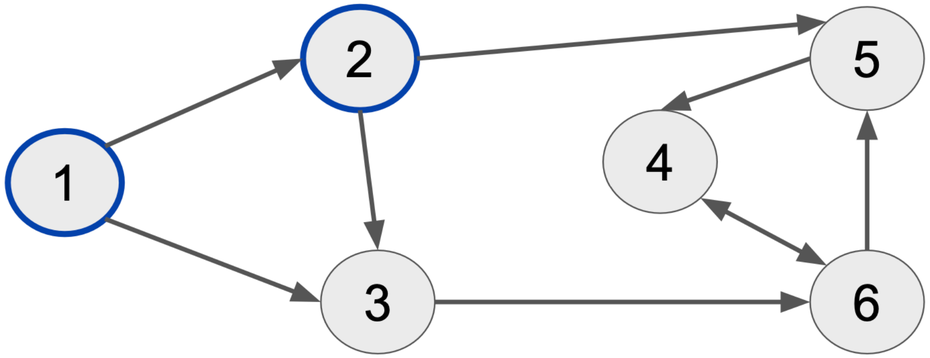


bfs(node=1, graph)

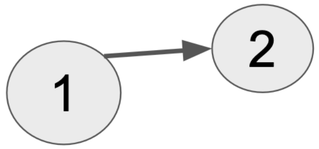
front	2	3	rear

Queue

Breadth-first search



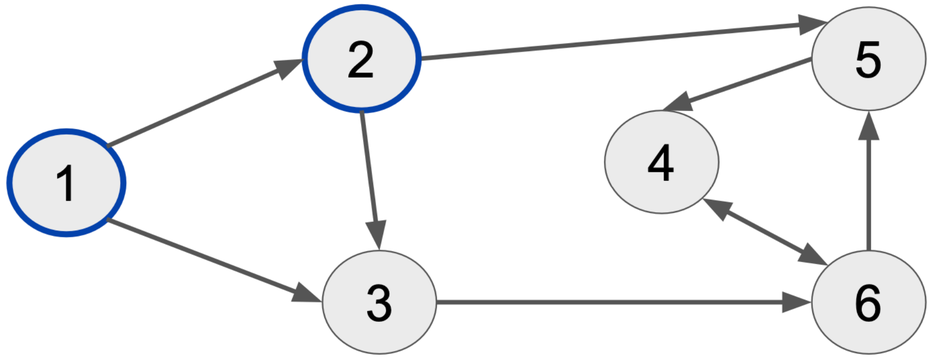
bfs(node=1, graph)



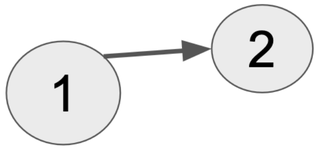
front	3	rear

Queue

Breadth-first search



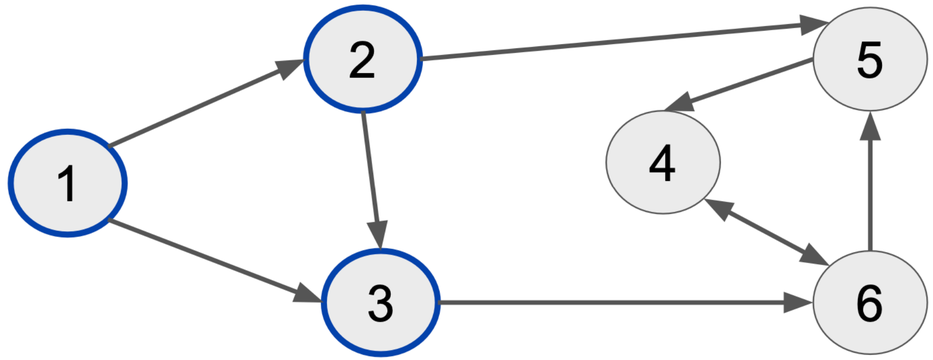
bfs(node=1, graph)



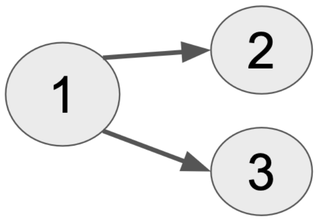
front	3	5	3	rear

Queue

Breadth-first search



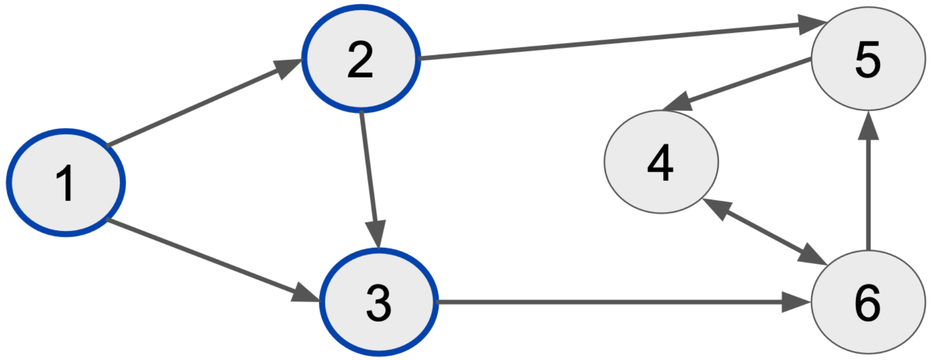
bfs(node=1, graph)



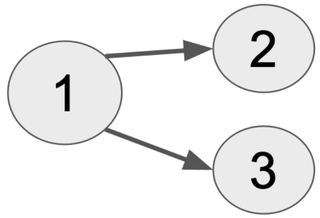
front	5	3	rear

Queue

Breadth-first search



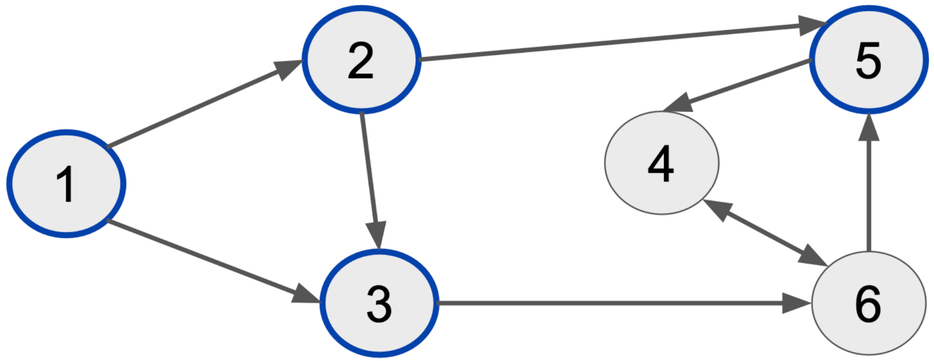
bfs(node=1, graph)



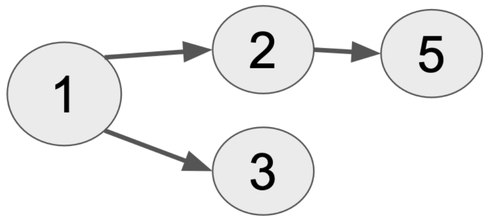
front	5	3	6	rear

Queue

Breadth-first search



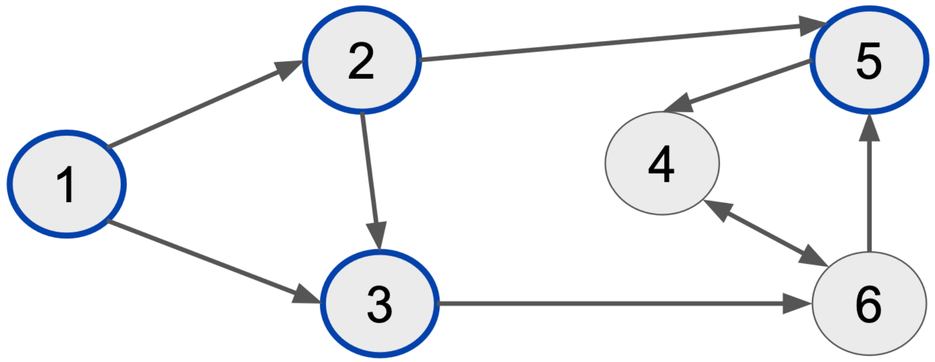
bfs(node=1, graph)



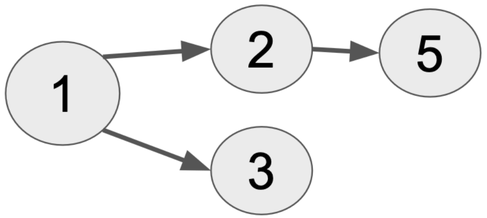
front	3	6	rear

Queue

Breadth-first search



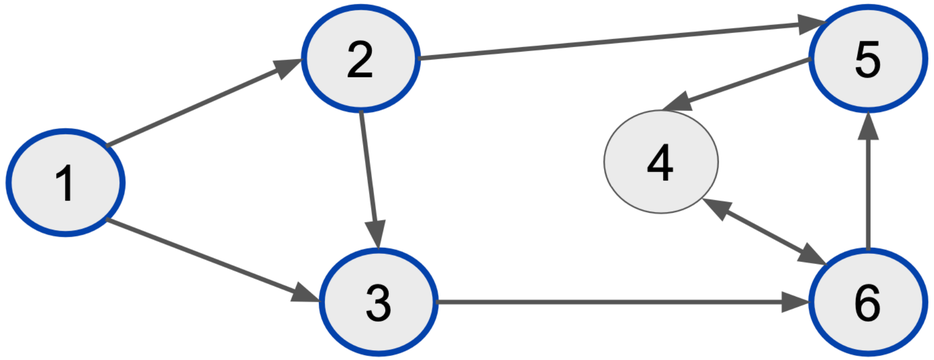
bfs(node=1, graph)



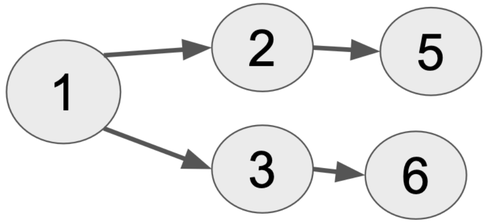
front	3	6	4	rear

Queue

Breadth-first search



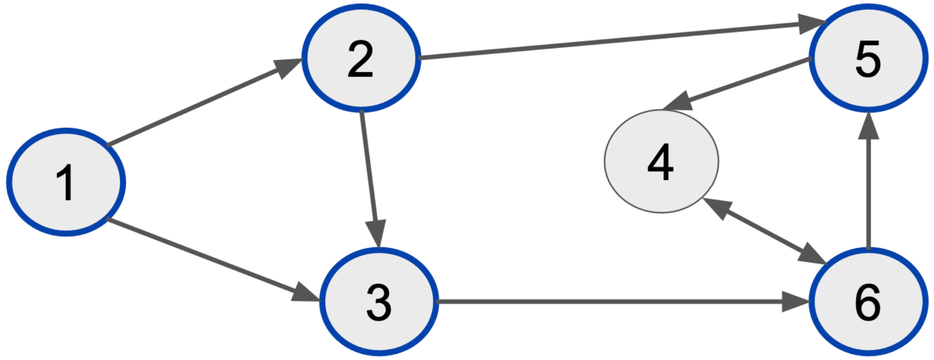
bfs(node=1, graph)



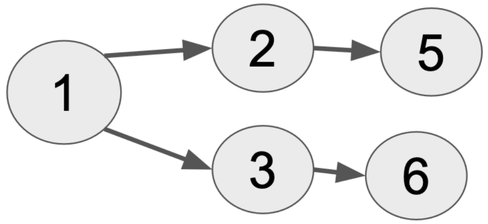
front	4	rear

Queue

Breadth-first search



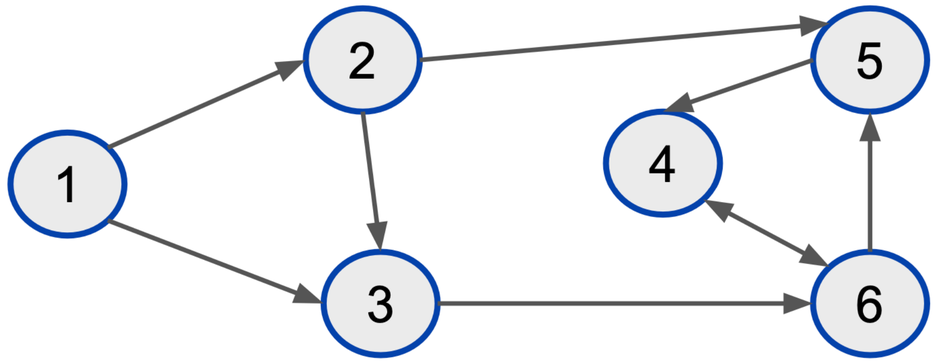
bfs(node=1, graph)



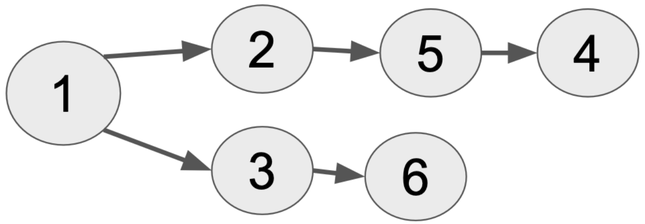
front	4	4	5	rear

Queue

Breadth-first search



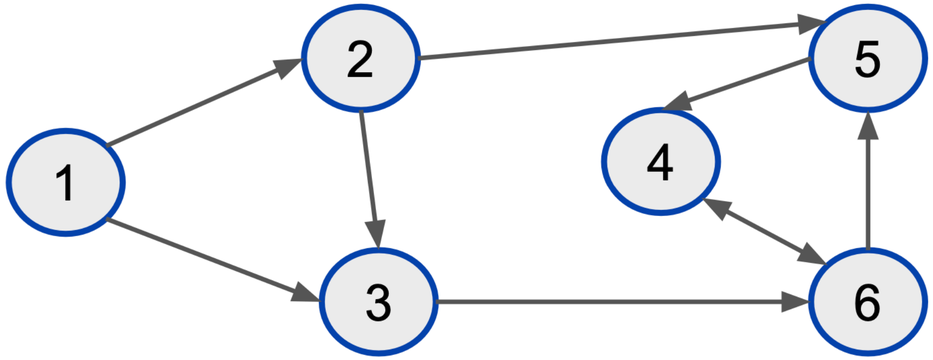
bfs(node=1, graph)



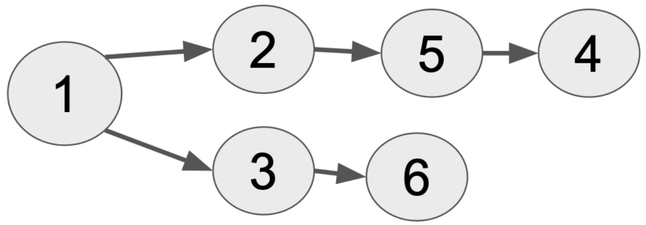
front	4	5	rear

Queue

Breadth-first search



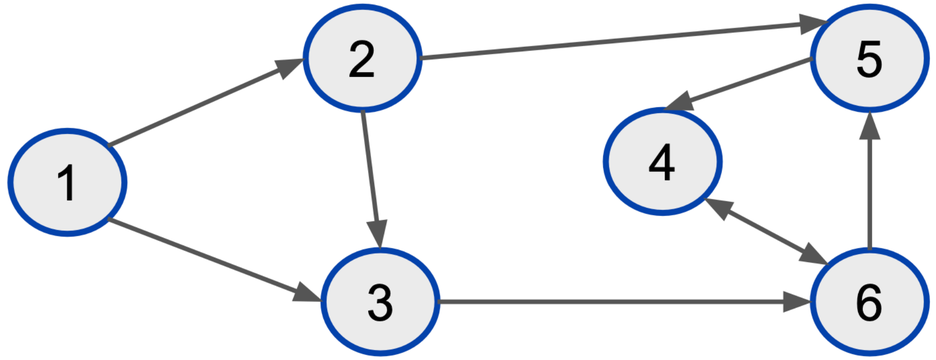
bfs(node=1, graph)



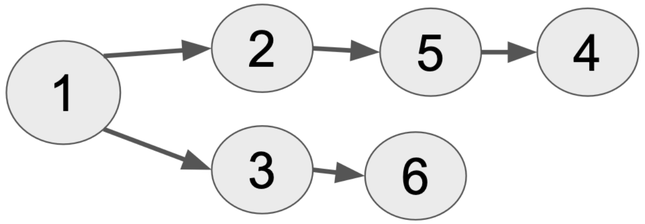
front	4	5	6	rear

Queue

Breadth-first search



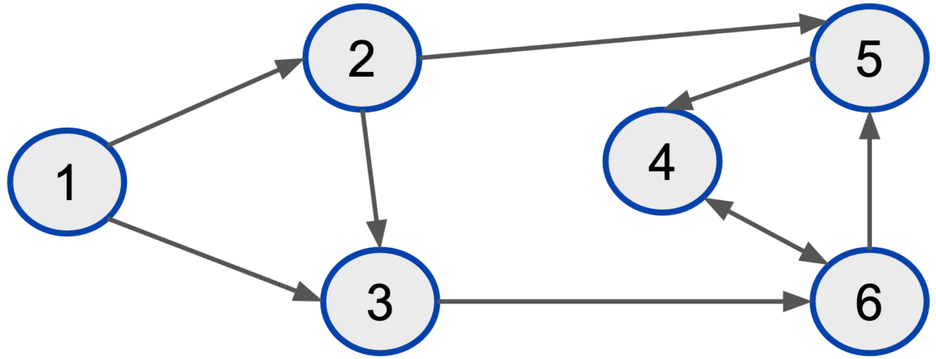
bfs(node=1, graph)



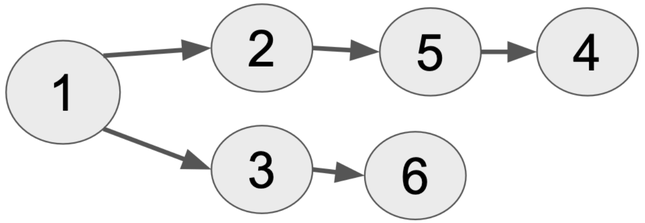
front	5	6	rear

Queue

Breadth-first search



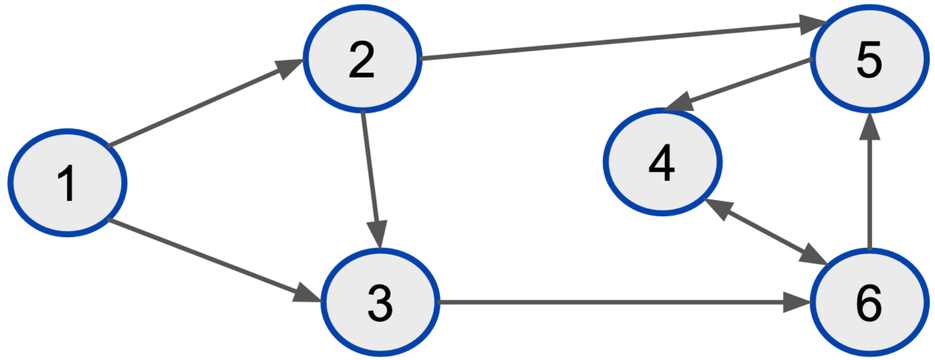
bfs(node=1, graph)



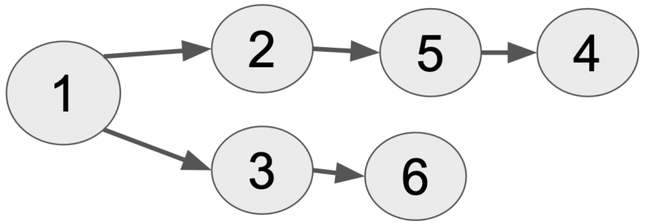
front	6	rear

Queue

Breadth-first search



bfs(node=1, graph)



front	rear

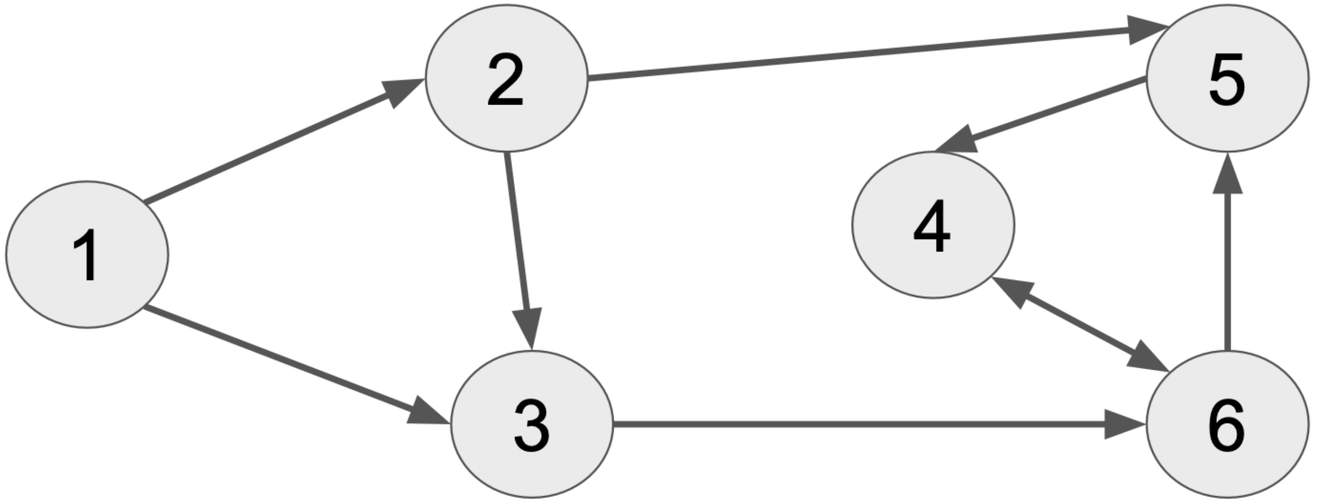
Queue

Depth First Search (DFS)

Depth First Search

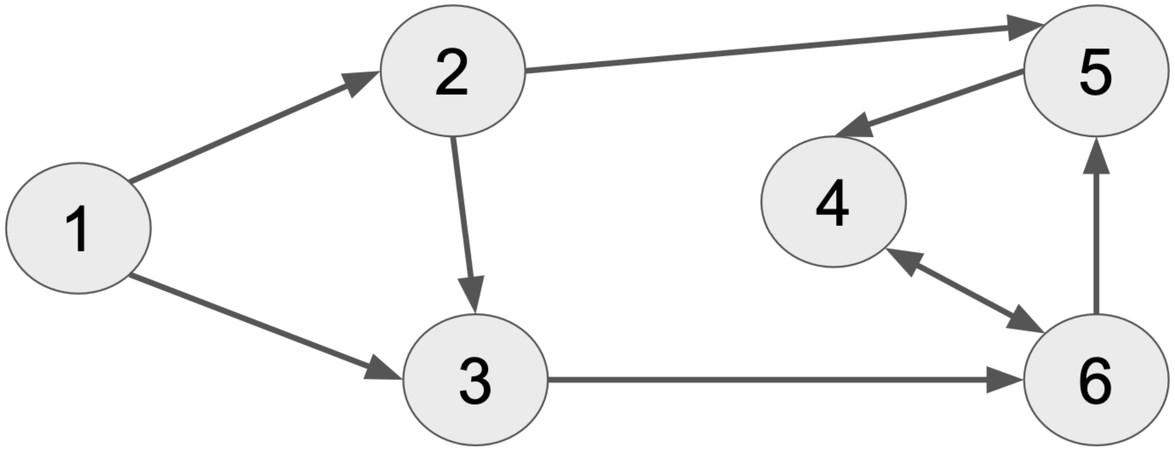
- **Algorithm** DFS(G,u): {We assume u has already been marked as visited}
  - Input: A graph G and a vertex u of G
  - Output: A collection of vertices reachable from u, with their discovery edges
  - for each outgoing edge e = (u,v) of u do
    - if vertex v has not been visited then
    - Mark vertex v as visited (via edge e).
    - Recursively call DFS(G,v).

Depth-first search

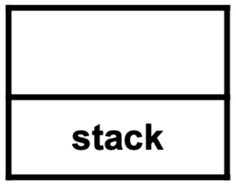


dfs(node=1, graph)

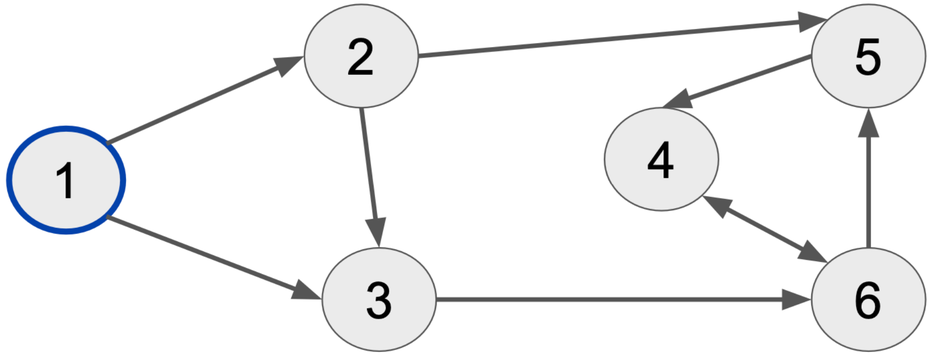
Depth-first search



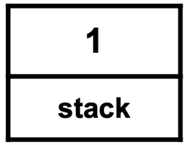
dfs(node=1, graph)



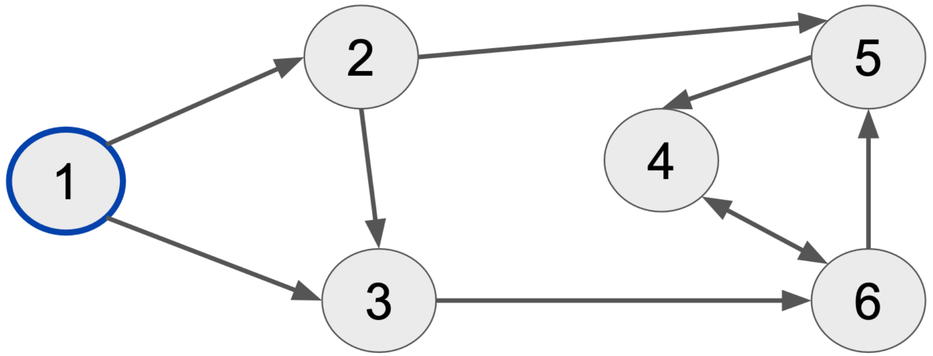
Depth-first search



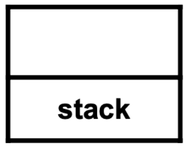
dfs(node=1, graph)



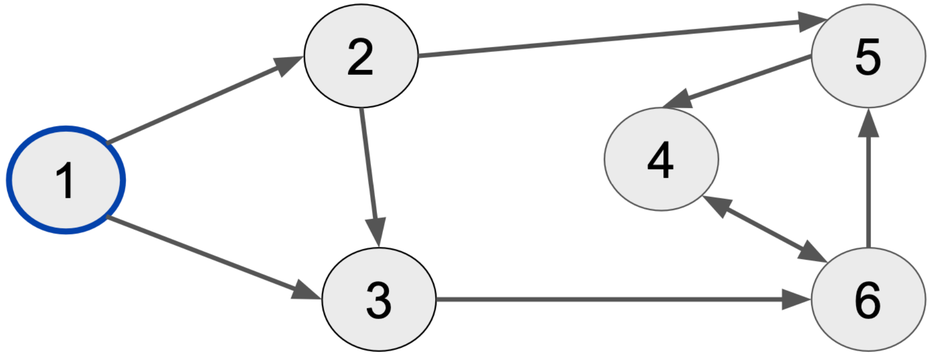
Depth-first search



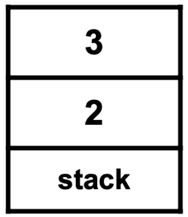
dfs(node=1, graph)



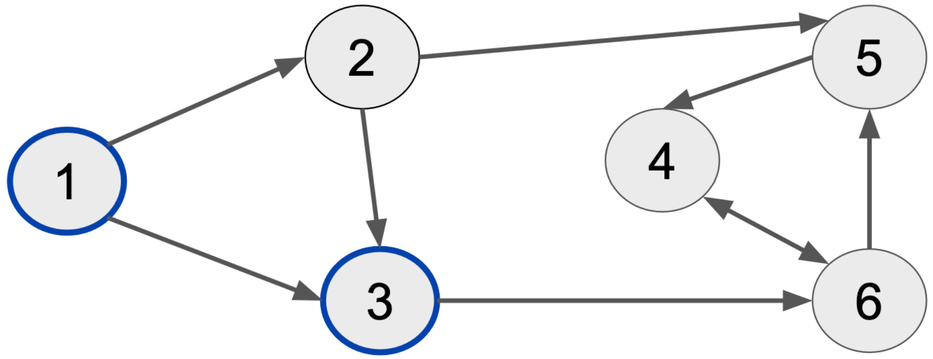
Depth-first search



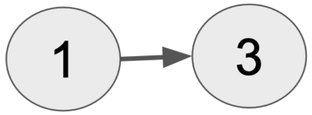
dfs(node=1, graph)

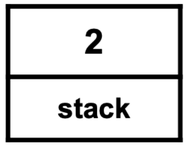


Depth-first search

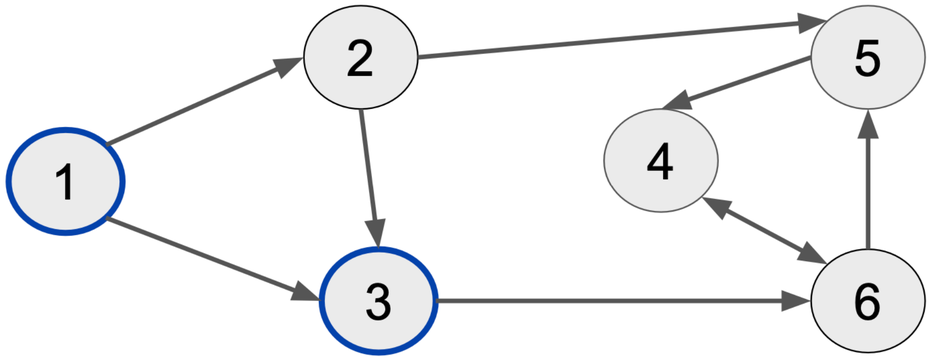


dfs(node=1, graph)

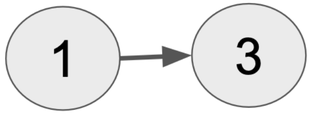


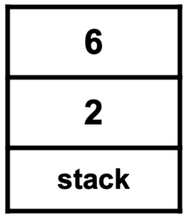


Depth-first search

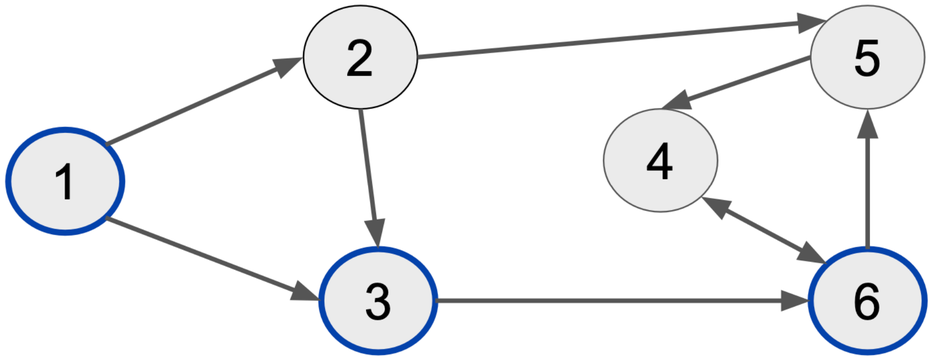


dfs(node=1, graph)

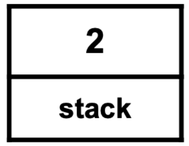




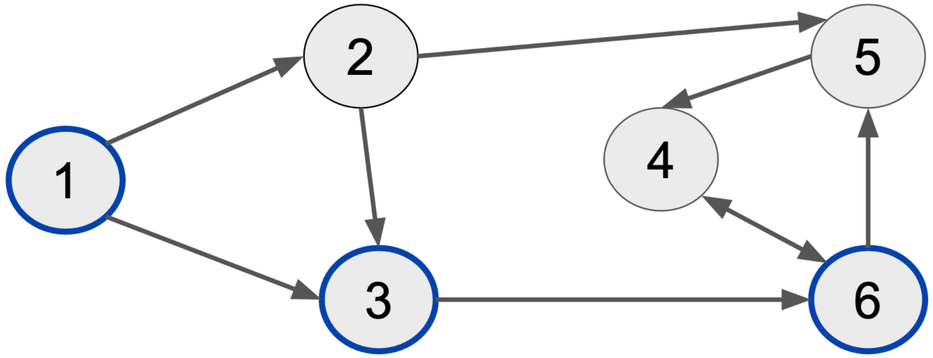
Depth-first search



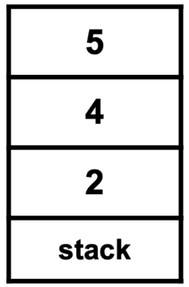
dfs(node=1, graph)



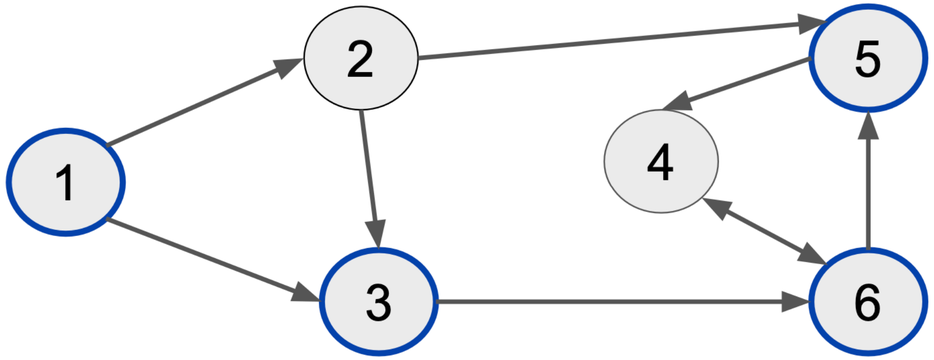
Depth-first search



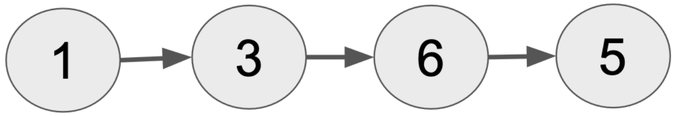
dfs(node=1, graph)

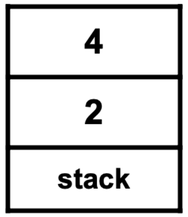


Depth-first search

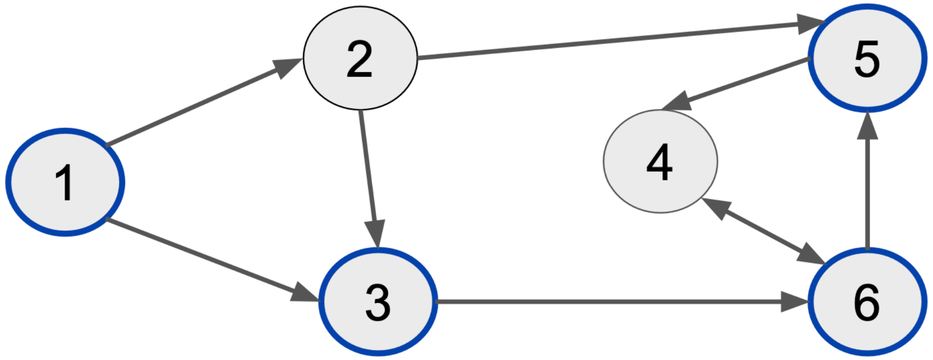


dfs(node=1, graph)

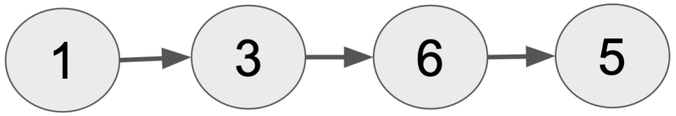


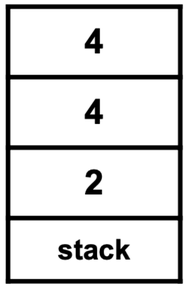


Depth-first search

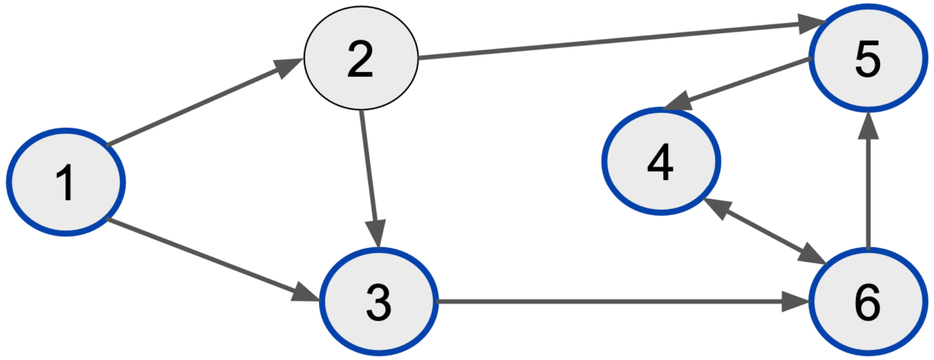


dfs(node=1, graph)

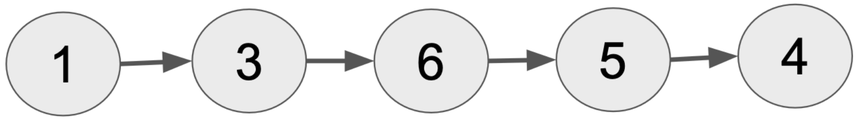


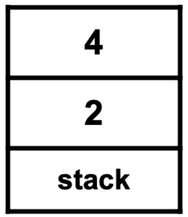


Depth-first search

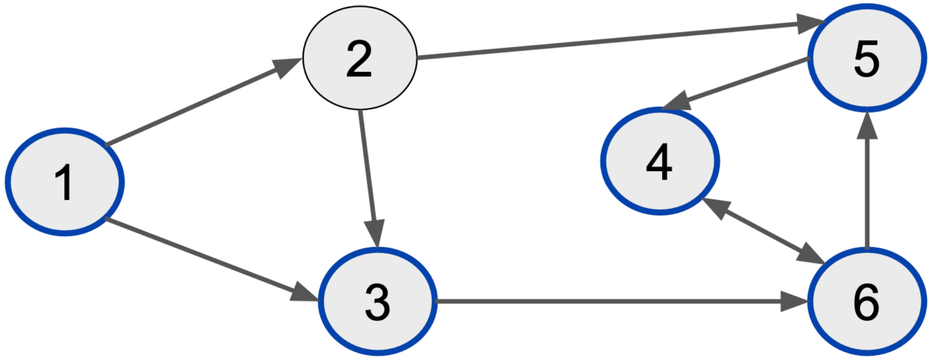


dfs(node=1, graph)

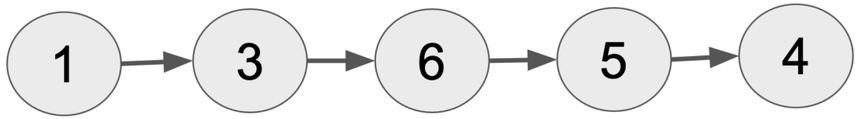


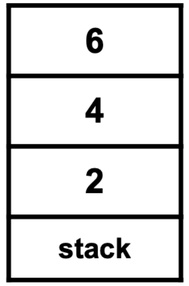


Depth-first search

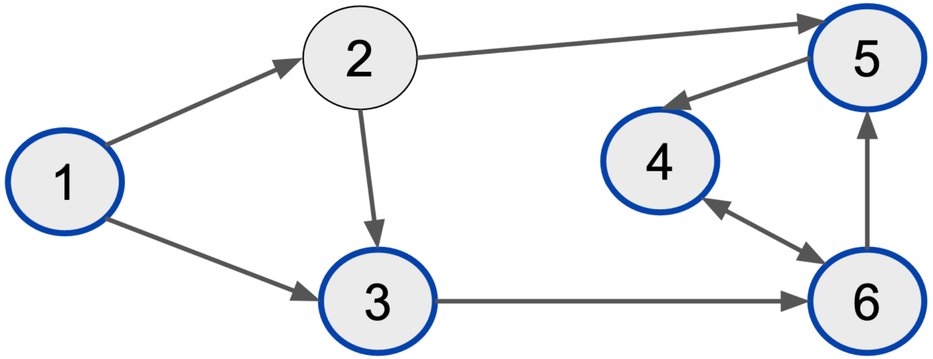


dfs(node=1, graph)

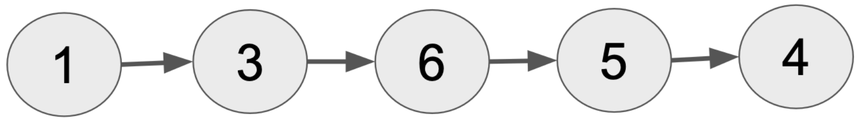


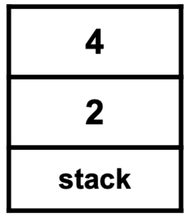


Depth-first search

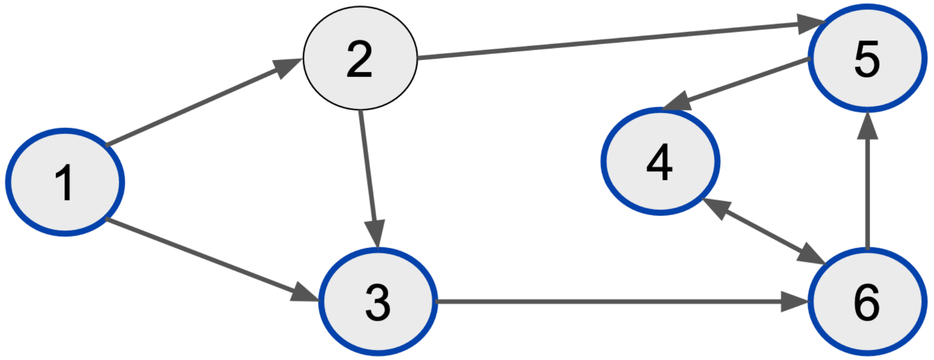


dfs(node=1, graph)

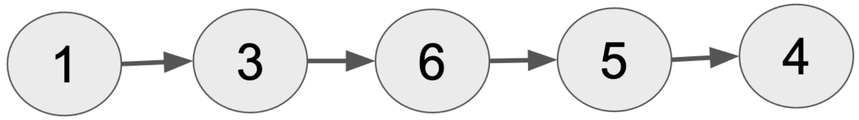


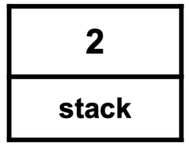


Depth-first search

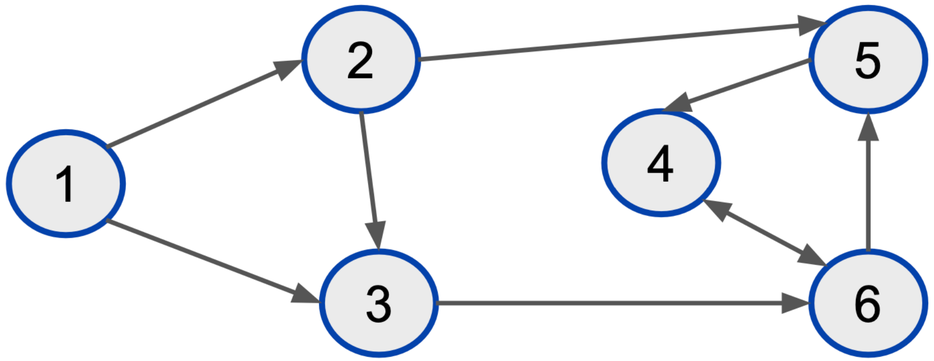


dfs(node=1, graph)

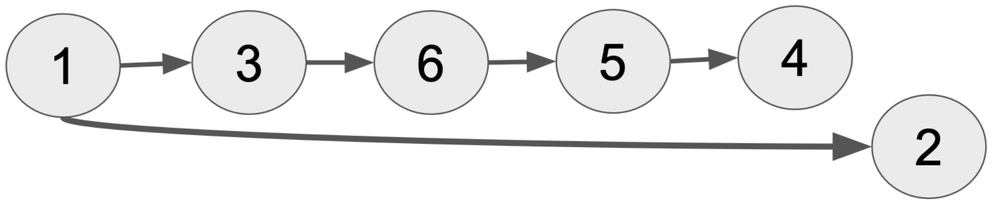




Depth-first search



dfs(node=1, graph)



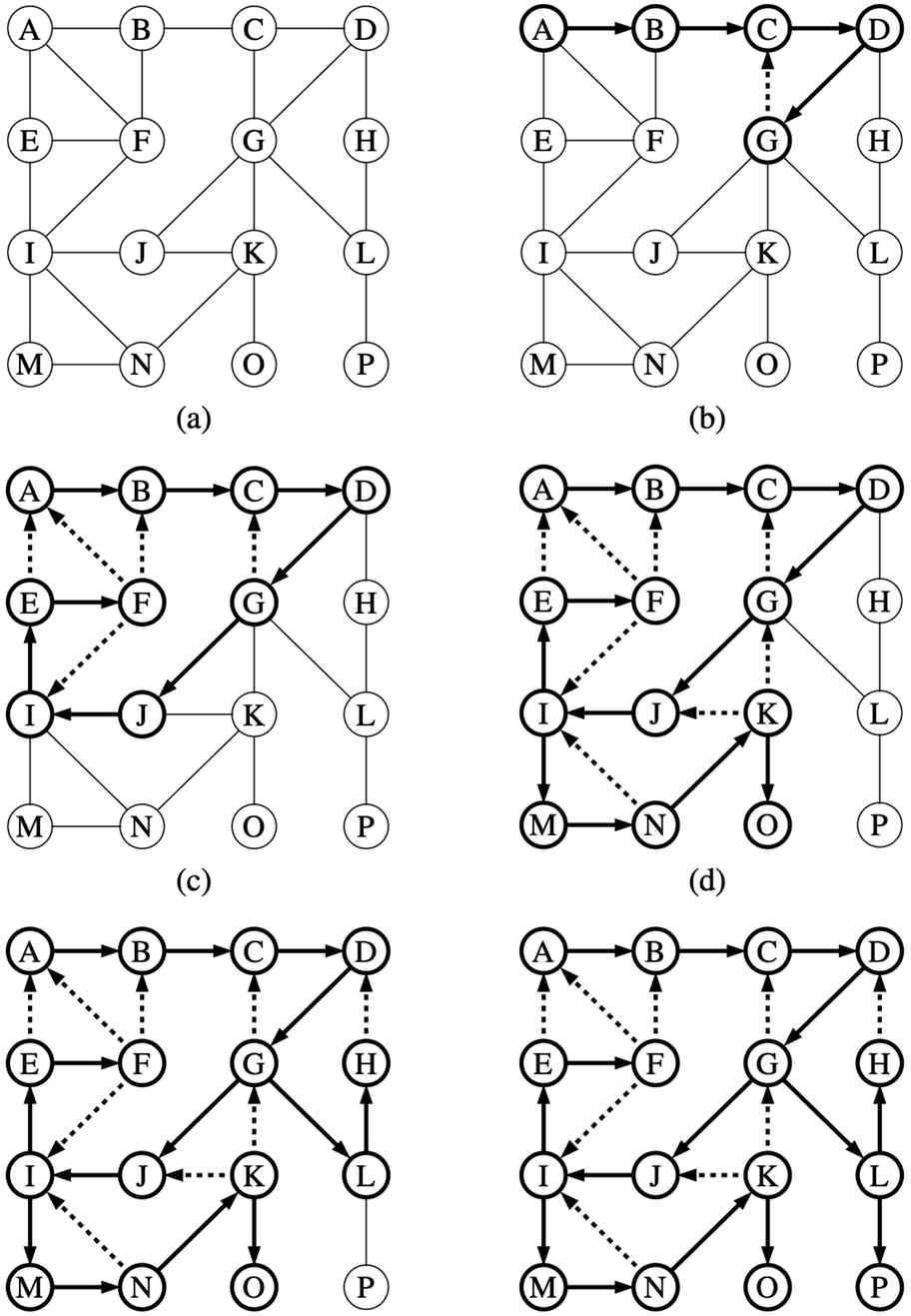
stack

BFS vs DFS

- DFS uses recursion/stack, exploring deeply before backtracking
- BFS uses a queue, exploring level by level
- Both keep track of discovery edges that form a traversal tree

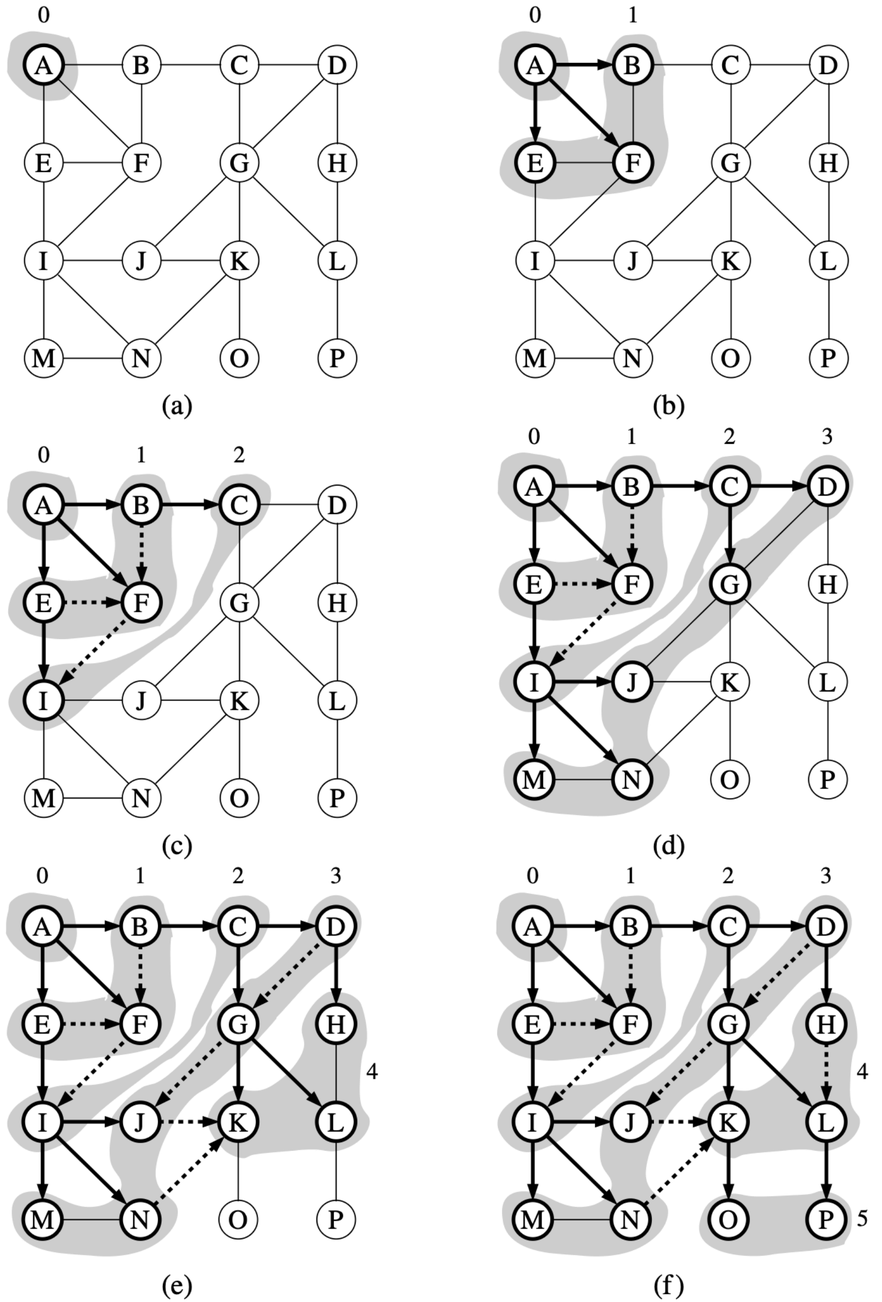
DFS Example

- discovery edges drawn as solid lines
- Examined not taken edges drawn as dashed lines



BFS Example

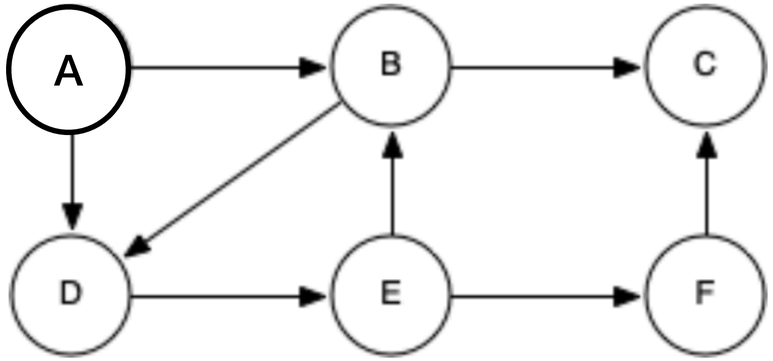
- discovery edges drawn as solid lines
- Examined not taken edges drawn as dashed lines

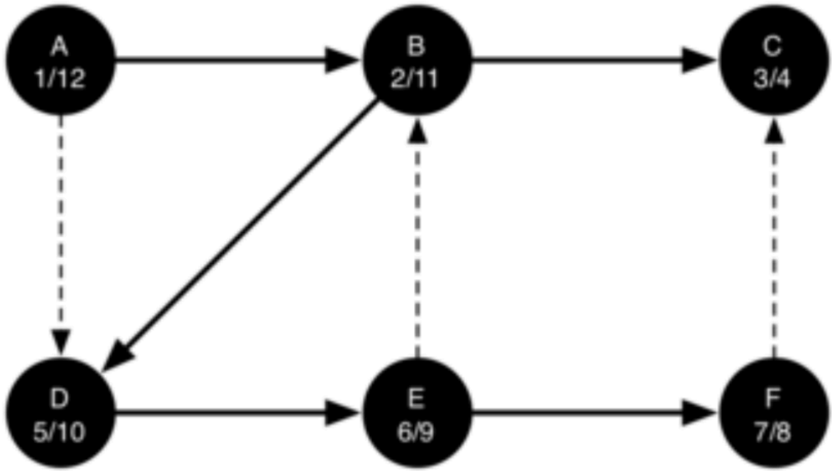


Applications

General Depth-First Search

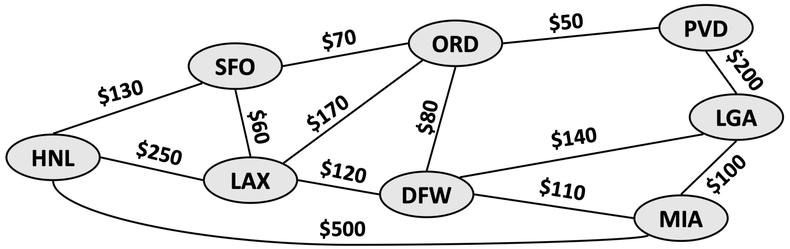
- 8.15. General Depth First Search
- Vertex:
  - name
  - discovery_time/closing_time





Searching for Paths

- Searching for a path from one vertex to another:
  - Any path or if there exists a path
  - Shortest path (minimum number of edges)
  - minimize path *cost* (sum of edge weights)



Finding Paths

- Easiest way: Depth-First Search (DFS) with recursive backtracking
- Find a path between two nodes if it exists:
  - Find all the nodes **reachable** from a node.
  - Where can I travel to, if starting in San Francisco?

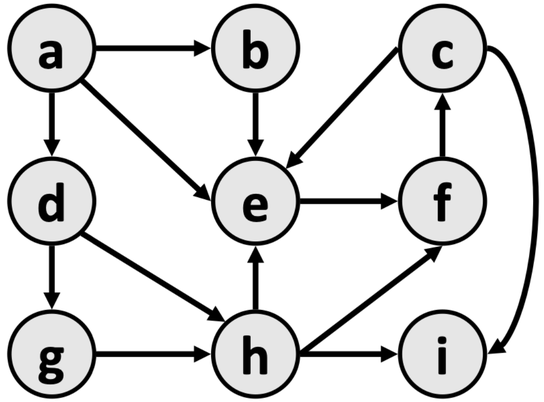
.

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

DFS
Path finding

- DFS from *a* to *h* (assuming A-Z order) visits:
  - **a**
    - **b**
      - **e**
        - **f**
          - **c**
            - **i**
    - **d**
      - **g**
        - **h**
  - path found: **{a, d, g, h}**



DFS

- In an *n*-node, *m*-edge graph, takes O(*m* + *n*) time with an adjacency list
- Visit each edge once, visit each node at most once
- How to modify the pseudocode to look for a specific path?
- DFS Pseudocode:

dfs:

starting from v_1:

    mark v_1 as **seen**.

    for each of v_1's unvisited neighbors *n*:

        dfs(*n*)

DFS that finds path

- **dfs from v_1 to v_2:**

  mark v_1 as **visited**, and **add to path**

  perform a **dfs** from each of v_1's unvisited neighbors n to v_2:

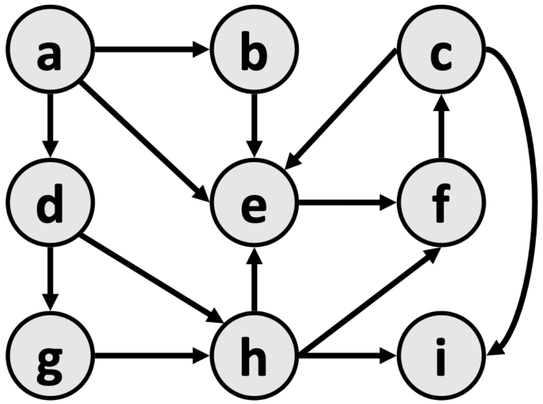
  if **dfs**(n, v_2) succeeds: a path is found!

  if all neighbors fail: **remove v_1 from path**

• To retrieve the DFS path found, pass a collection parameter to each call and choose-explore-unchoose.

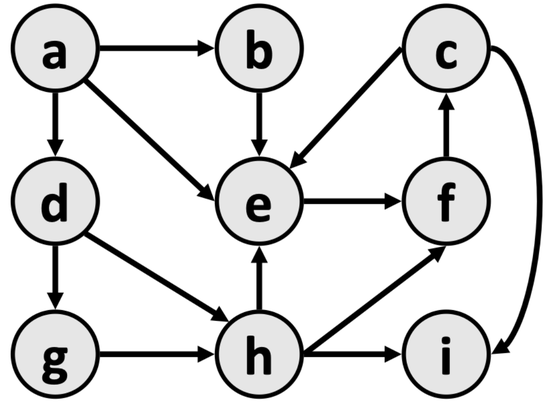
.

DFS



- *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 using ‘choose - explore - unchoose’
- *optimality*: not optimal. DFS is guaranteed to find a path, not necessarily the best/shortest path
  - Example: dfs(a, i) returns {a, b, e, f, c, i} rather than {a, d, h, i}

BFS properties



- 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: 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.

BFS

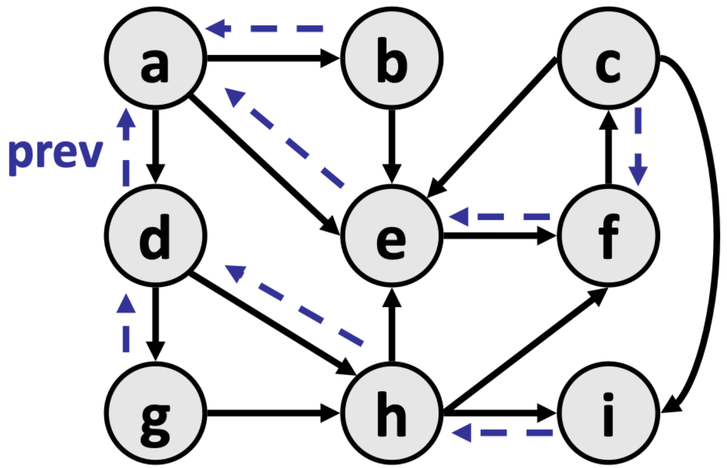
Finding a Shortest Path

- Shortest path questions:
  - Fewest number of steps to complete a task?
  - Least amount of edits between two words?

  - BFS:
    - keeps a queue of nodes
    - dequeue a node, enqueue all its neighbors

.

BFS: finding path



- **bfs** from v_1 to v_2:
  - create a *queue* of vertexes to visit, initially storing just v_1
    - mark v_1 as visited.
  - while *queue* is not empty and v_2 is not seen:
    - dequeue a vertex *v* from it,
    - mark that vertex *v* as **visited**,
    - add each unvisited neighbor ***n*** of *v* to the *queue,* **while setting *n*'s prev to *v***