COSC 222 Data Structure

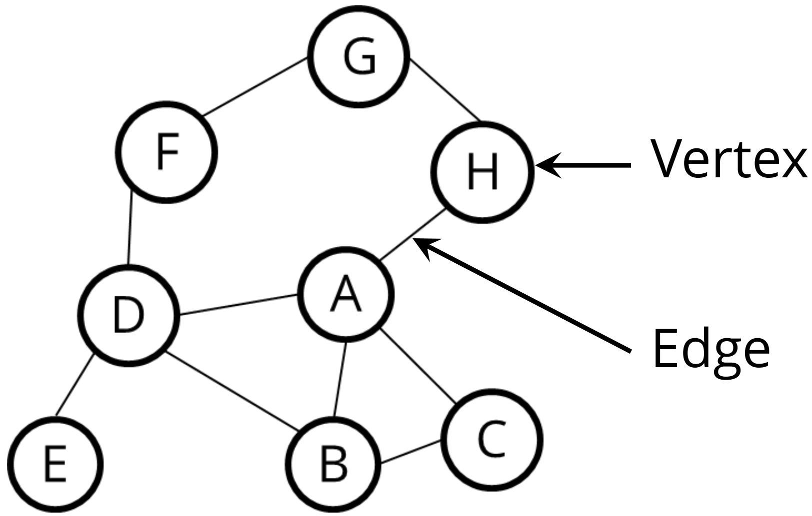
Graphs – Part 1

Exam Dates Reminder

- Midterm exam: **Thursday March 13** (Previously March 11)
  - based on in-class poll taken before mid-semester break
- Final exam: TBD
  - examination period for W2024 T2 is April 12-April 27, 2025.

Graph: formal definition

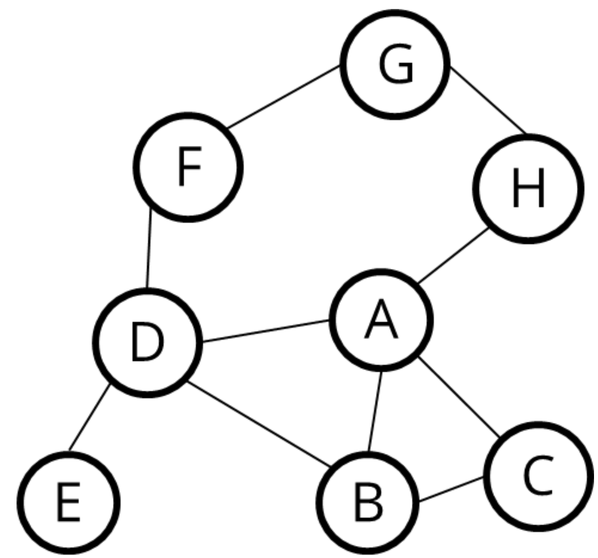
- The words “node” and “vertex” are synonyms
- Vertices are the circle things (A), edges are the lines (A,B)



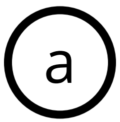
- A graph is a pair G = (V , E), where...
  - V is a set of vertices
  - E is a set of edges (pairs of vertices)

Graph: formal definition

- Path: sequence of edges that connect two vertices in a graph
  - Example: path from A to F: {A, D, F}
- The **length** of a path is the number of edges on the path, which is equal to $N - 1$ where N is number of vertices
- Two nodes in a graph are called **adjacent** or **neighbor** if there's an edge between them
  - Vertex A is adjacent to vertex B
  - Vertex A is adjacent to vertex D

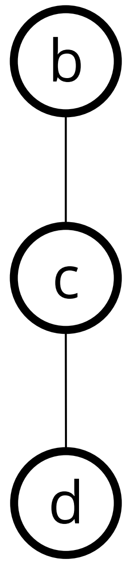


Graph: examples



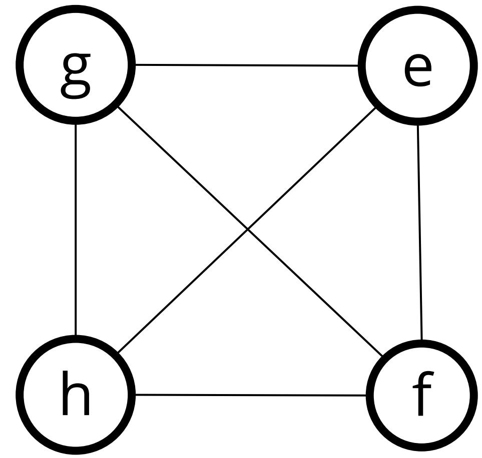
$V = \{a\}$

$E = \{\}$



$V = \{b, c, d\}$

$E = \{(b, c), (c, d)\}$



$V = \{e, f, g, h\}$

$E = \{(e, f), (f, g),$

$(g, h), (h, e),$

$(e, g), (f, h)\}$

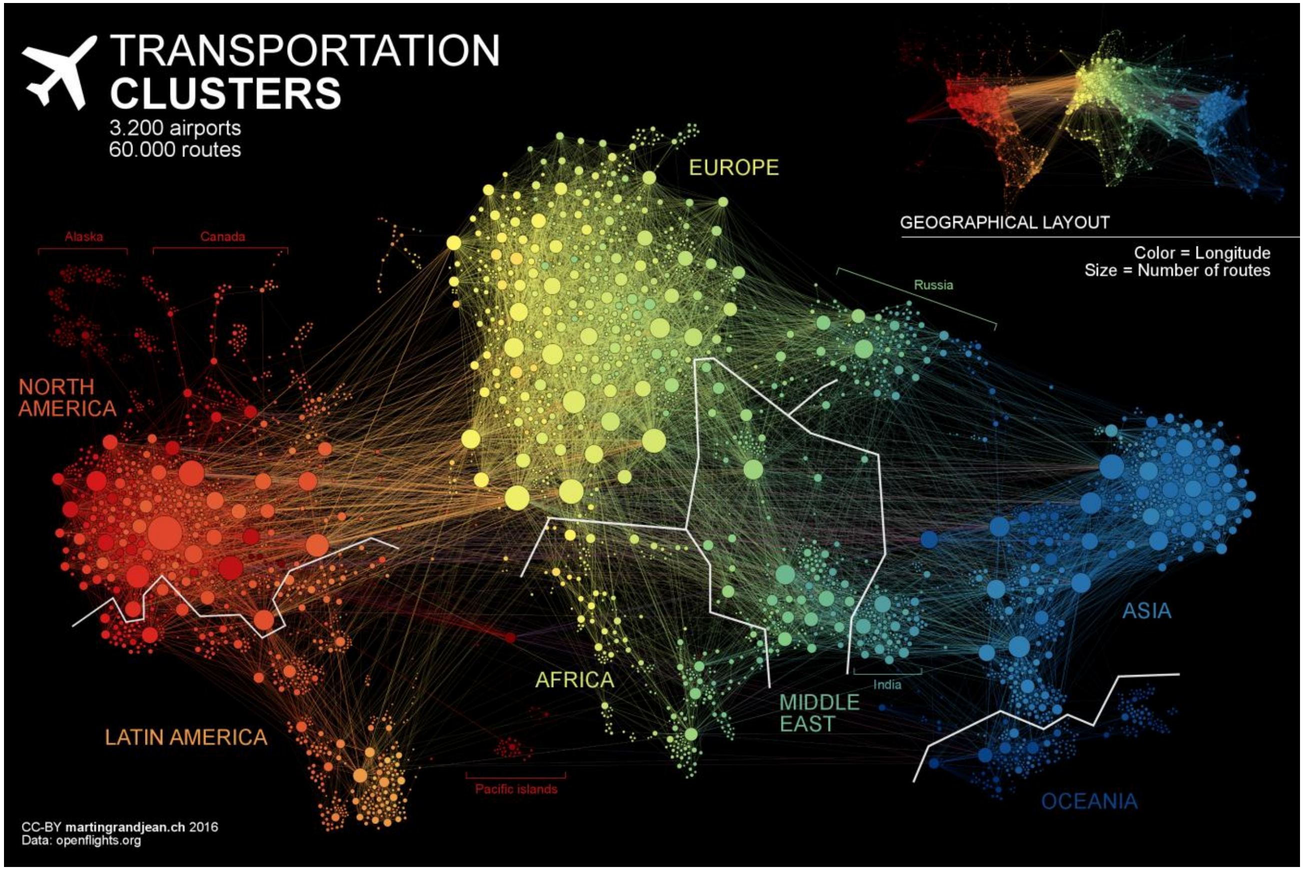
Applications of graphs

- **In a nutshell:**
  - Graphs let us model the “relationship” between items.
  - If that seems like a very general definition, that’s because graphs are a very general concept!

- **Core insight:**
  - Graphs are an abstract concept that appear in many different ways
  - Many problems can be modeled as a graph problem.

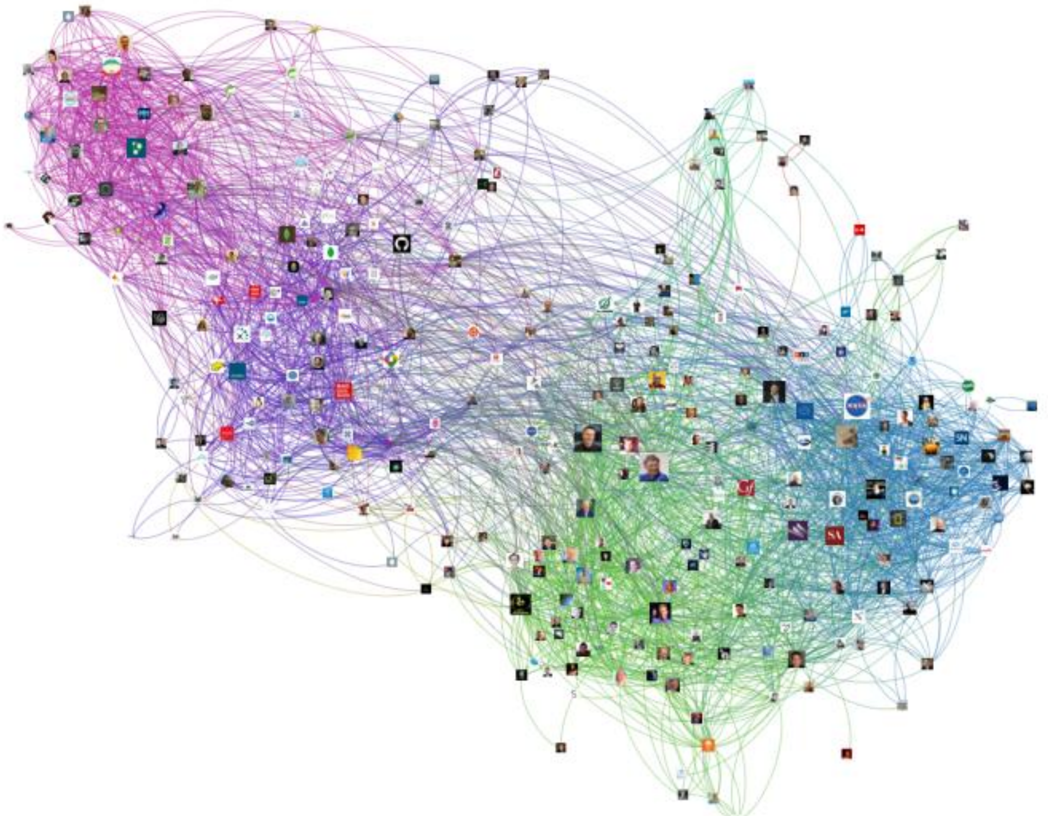
Application: Airline flight graph

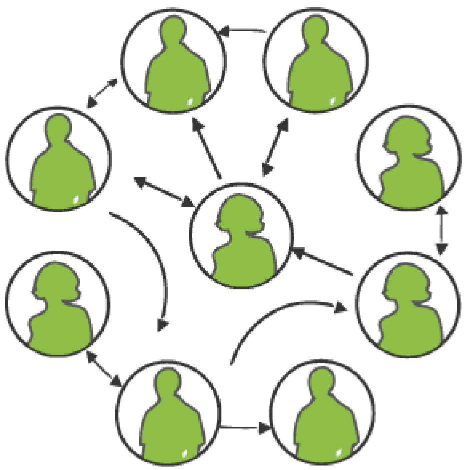
- Questions:
  - What is the cheapest/shortest/etc flight from *A* to *B*?
  - Is the route the airline offering me actually the cheapest route?



Application: Social Media Graph

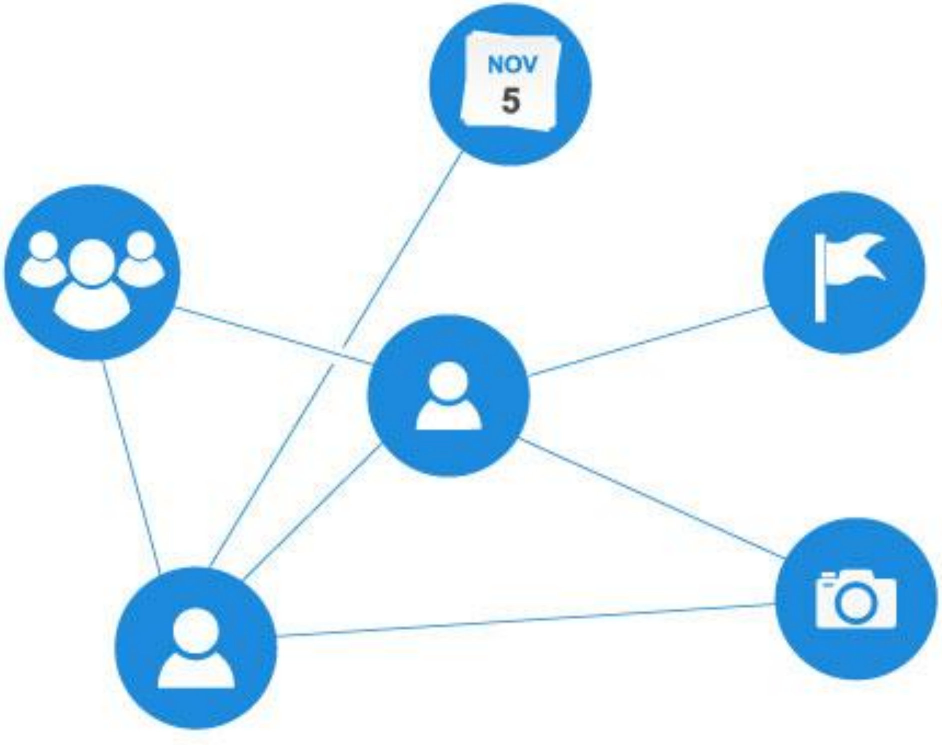
- Social Network Graph
  - who knows whom, who communicates with whom
  - who influences whom, or other relationships





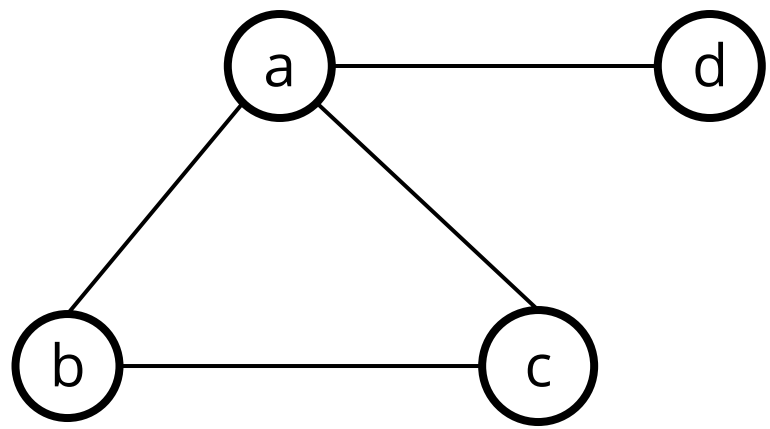
Application: Facebook Graph API

- Facebook's Graph API
  - Everything (e.g., users, photos) are vertices or nodes
  - Every connection or relationship is an edge



Undirected graphs

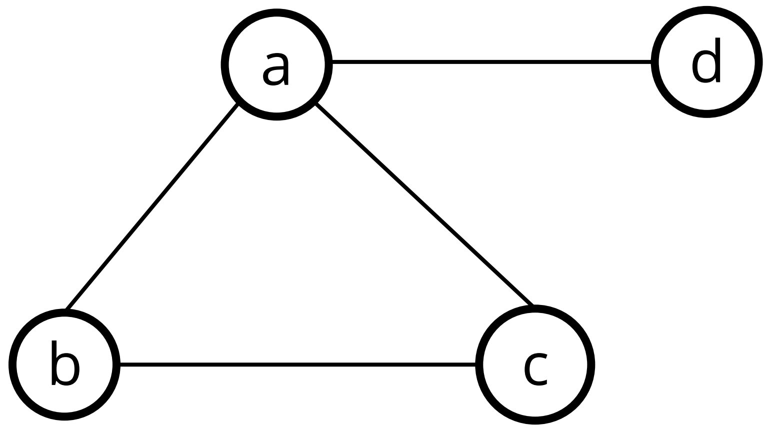
- In a **undirected graph**, edges have no direction: are two-way



- This means that $(x, y) \in E$ implies that $(y, x) \in E$.
- Often, we treat these two pairs as equivalent and only include one of the two permutations.

Degree of a vertex

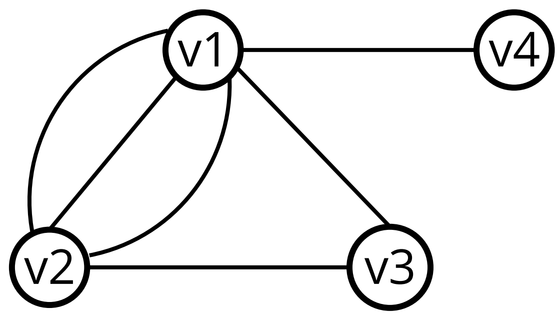
- The **degree** of some vertex *v* is the number of edges containing that vertex.
- So, the degree is the number of “ways out” of that vertex.
- The degree of a graph *G* is equal to it’s largest vertex degree.



- What is the degree of vertex *a* ? 3
- What is the degree of vertex *d* ? 1
- What is the degree of the graph? 3

Degree of a vertex

- Note that some graphs allow for parallel edges.



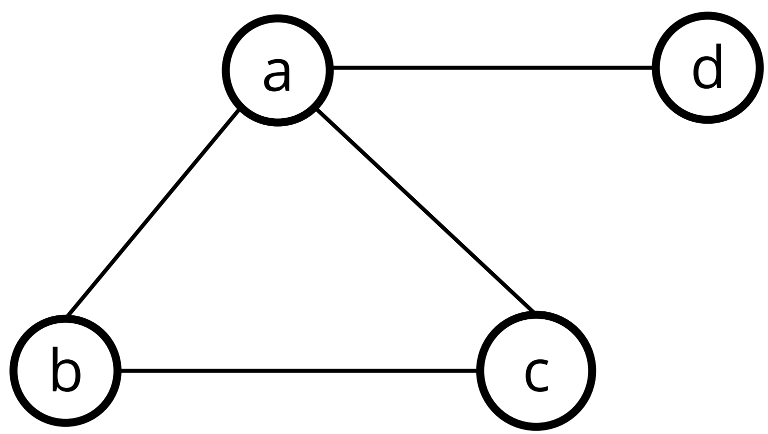
- deg(v1) =?
- deg(v2) =?
- deg(v3) =?
- deg(v4) =?

Handshaking Theorem

- The sum of degrees of the vertices of an **undirected graph** is twice the number of edges
- If G=(V,E) is a graph with E edges, then-

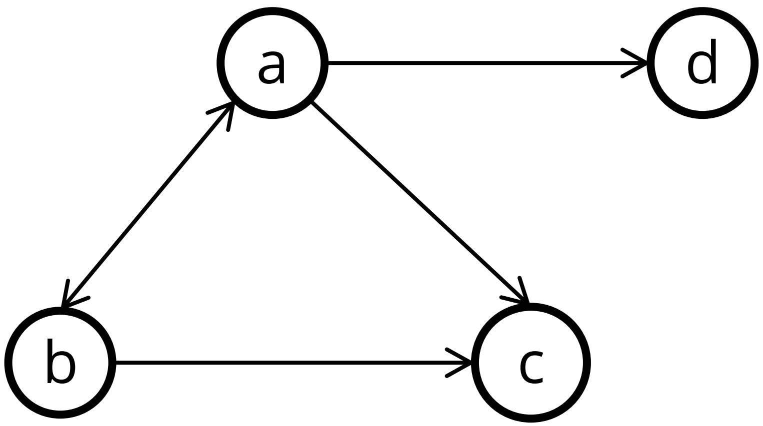
$$\sum \deg_G(V) = 2E$$

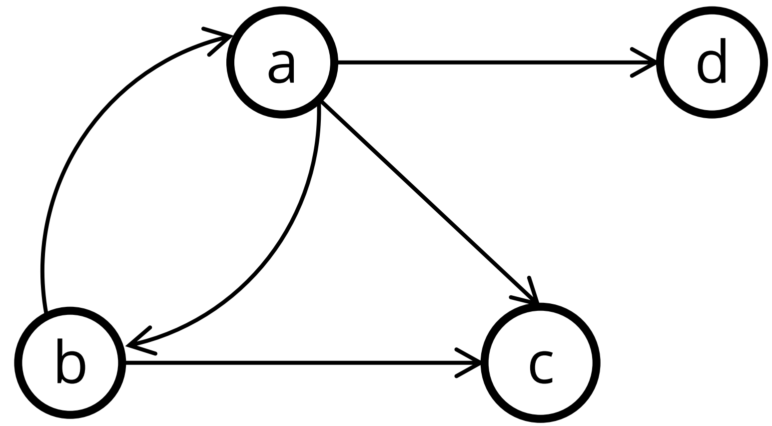
- Each edge contributes twice to the total degree count of all vertices. Thus, both sides of the equation equal to twice the number of edges.



Directed graph

- In a **directed graph**, edges *do* have a direction: are one-way

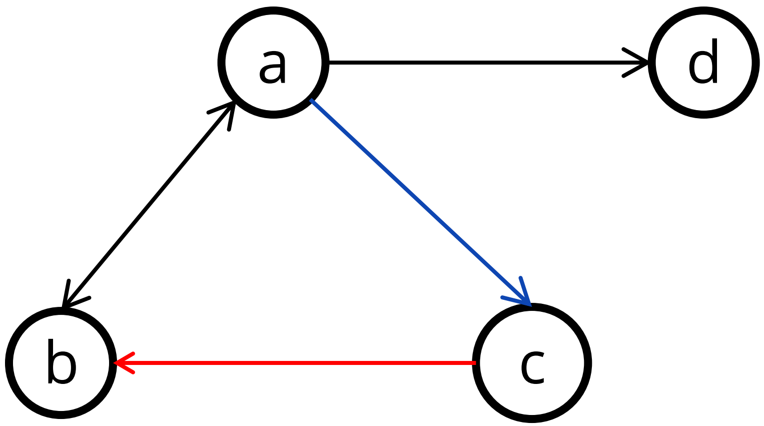




- Now, (x, y) and (y, x) mean different things.

Degree of a vertex: Directed graph

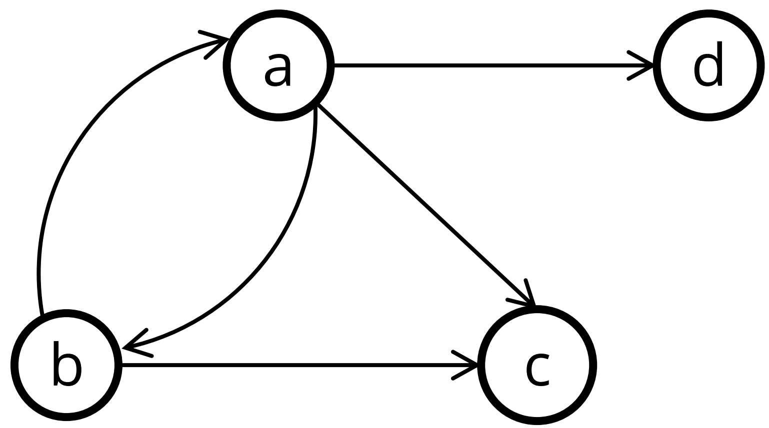
- **In-degree of a vertex**
  - The **in-degree** of *v* is the number of edges that point to *v*.
- **Out-degree of a vertex**
  - The **out-degree** of *v* is the number of edges that start at *v*.



- In-degree vertex for c: 1
- Out-degree vertex for c: 1

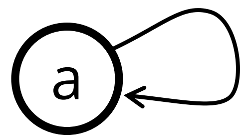
Degree of a vertex

- Let G = (V, E) be a graph with directed edges.
- then |E| = Σ in-deg(V) = Σ out-deg(V)

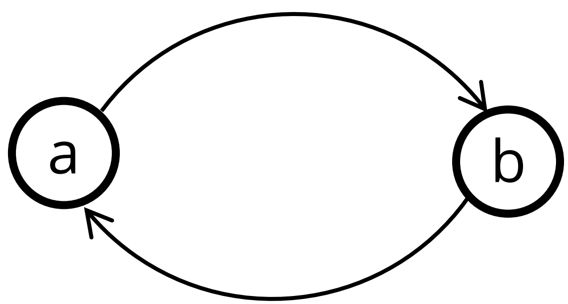


Self-loops and parallel edges

- A **self-loop** is an edge that starts and ends at the same vertex.

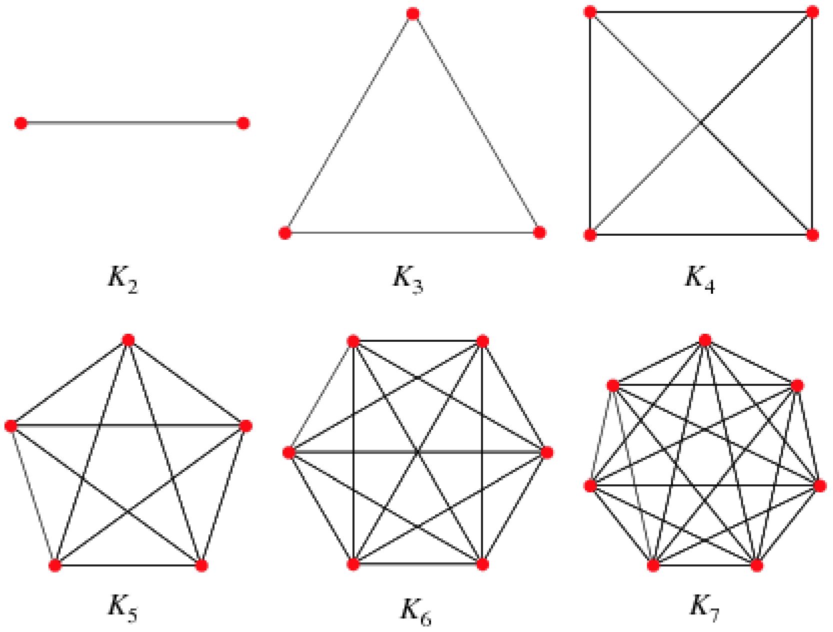


- **Simple graph:** A graph with no self-loops and no parallel edges.



Complete Graphs

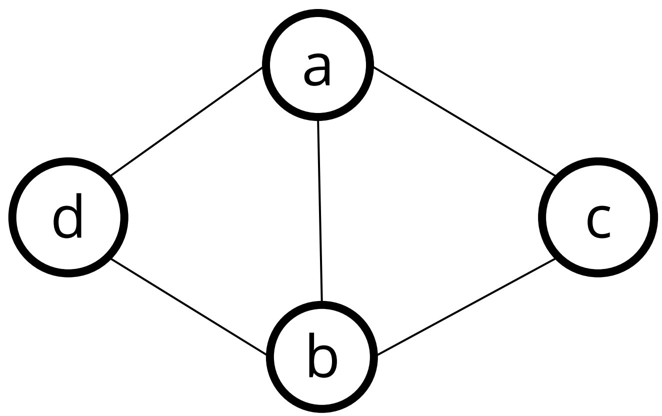
- A complete graph on n vertices, denoted by K_n, is the simple graph that contains exactly one edge between each pair of distinct vertices.



- A complete graph with n vertices is denoted K_n and has n(n-1)/2 undirected edges

Walks

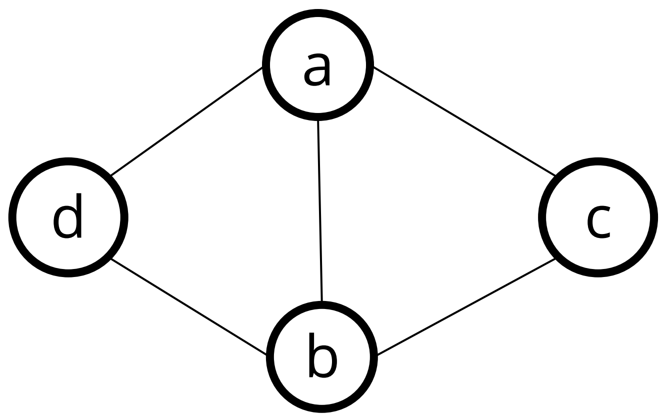
- A **walk** in a graph is a sequence of vertices, each linked to the next vertex by a specified edge of the graph.
  - More intuitively, a walk is one continuous line following the edges.



- In a walk, a Vertex can be repeated, Edges can be repeated
  Example: a $\rightarrow$ b $\rightarrow$ d $\rightarrow$ a $\rightarrow$ c

Paths

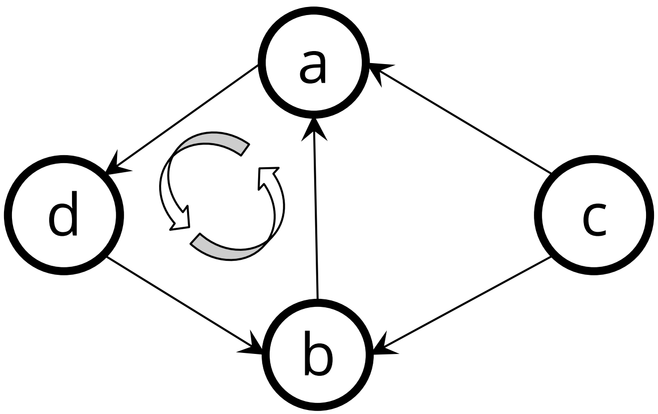
- A **path** is a walk that never visits the same vertex twice.



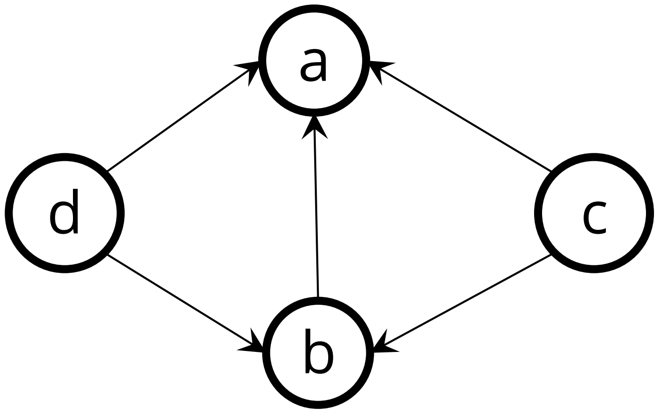
- Vertex not repeated, Edge not repeated
- Example: a $\rightarrow$ b $\rightarrow$ d

Cyclic and acyclic

- A graph with one or more cycles is called a cyclic graph.
- A graph having no cycles is an acyclic graph.



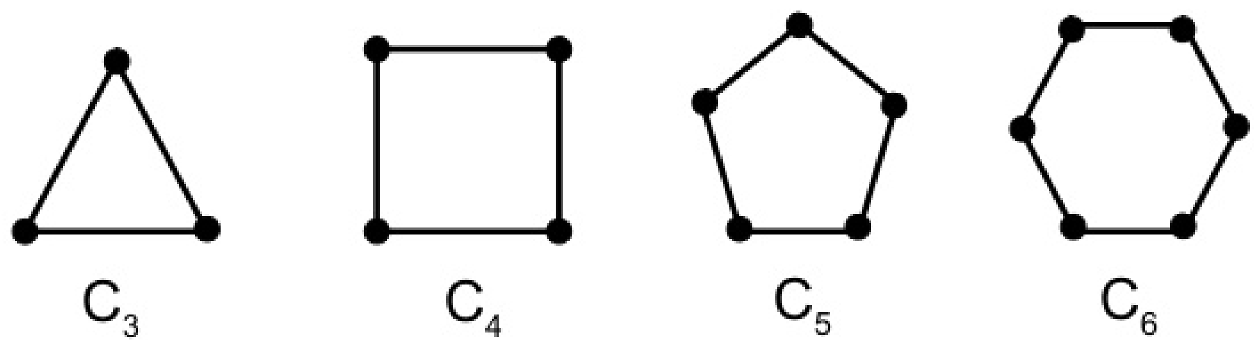
Cyclic graph



Acyclic graph

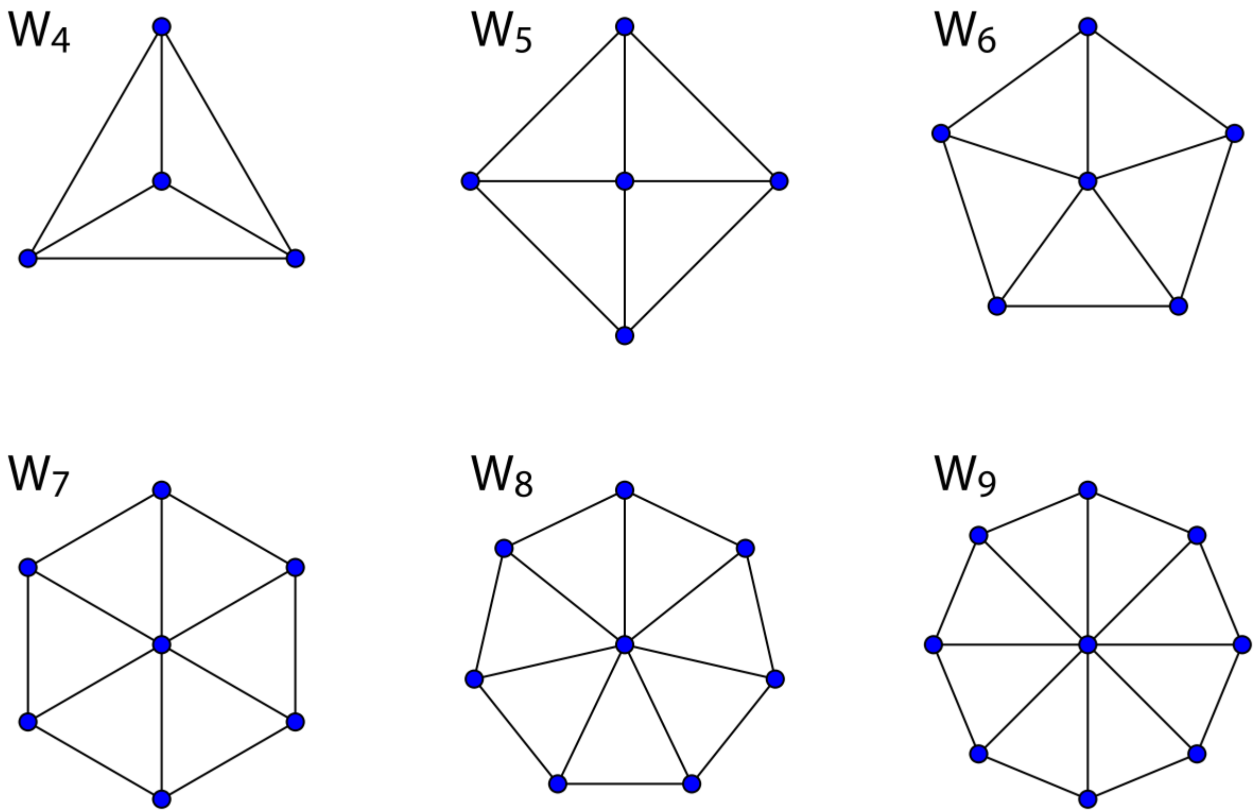
Cycles

- A cycle is a closed path.
- A cycle C_n for $n \geq 3$ consists of n vertices $v_1, v_2, \cdots, v_n$, and edges $\{v_1, v_2\}, \{v_2, v_3\}, \cdots, \{v_{n-1}, v_n\}, \{v_n, v_1\}$.



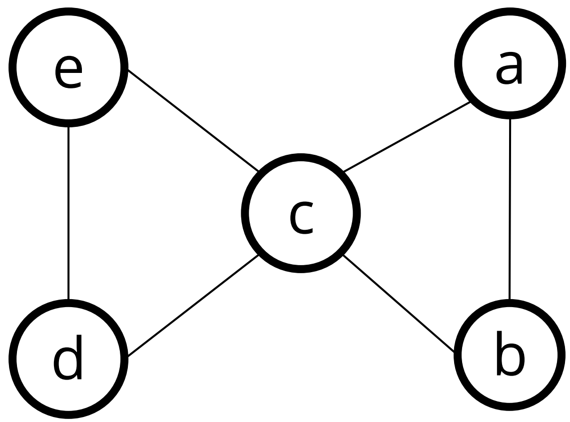
Wheels

- A wheel W_n is obtained by adding an additional vertex to a cycle C_n for $n \geq 3$ and connecting this new vertex to each of the n vertices in C_n by new edges.



Circuits

- A **circuit** in a graph is a path that begins and ends at the same vertex.



- A Vertex may be repeated, but an Edge can not repeated
- Example: a $\rightarrow$ b $\rightarrow$ c $\rightarrow$ d $\rightarrow$ e $\rightarrow$ c $\rightarrow$ a

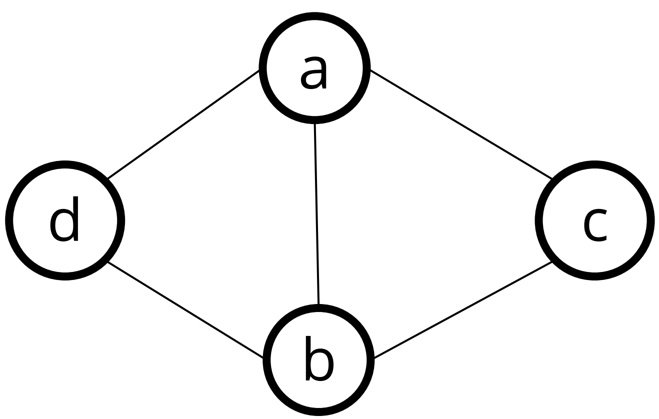
Summary

- Walk: Vertices may repeat. Edges may repeat (Closed or Open)
- Circuit: Vertices may repeat. Edges cannot repeat (Closed)
- Path: Vertices cannot repeat. Edges cannot repeat (Open)
- Cycle: Vertices cannot repeat. Edges cannot repeat (Closed)

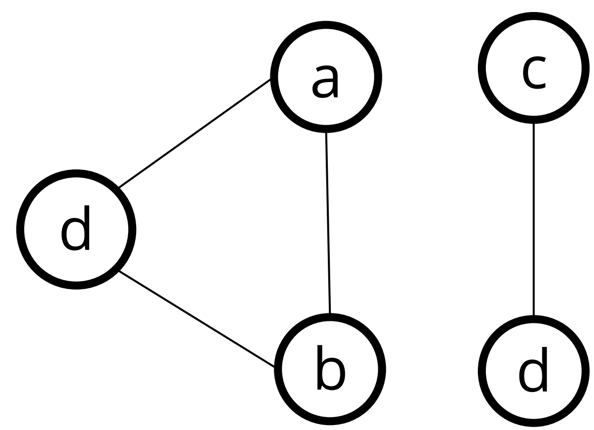
Connected graph

- Vertex *a* is **reachable** from *b* if a path exists from *a* to *b*.
- A graph is **connected** if every vertex is reachable from every other vertex via some path.
- E.g.: if we pick up the graph and shake it, nothing flies off.
- **Strongly connected:** Every vertex has an edge to every other vertex.
  - Same as complete but often associated with directed graphs or graph segments.

Connected



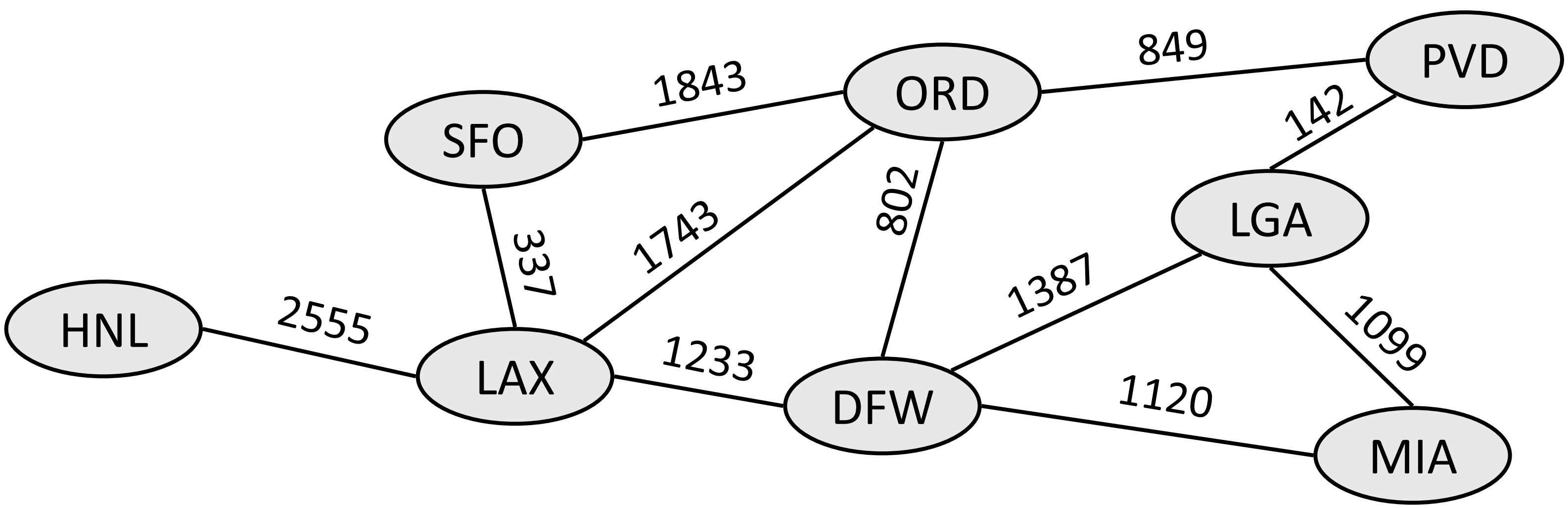
Not connected



Weighted graphs

- **weight**: Cost associated with a given edge.
  - Some graphs have weighted edges, and some are unweighted.
  - Edges in an unweighted graph can be thought of as having equal weight (e.g. all 0, or all 1, etc.)
  - Most graphs do not allow negative weights.

- *Example*: graph of airline flights, weighted by miles between cities:

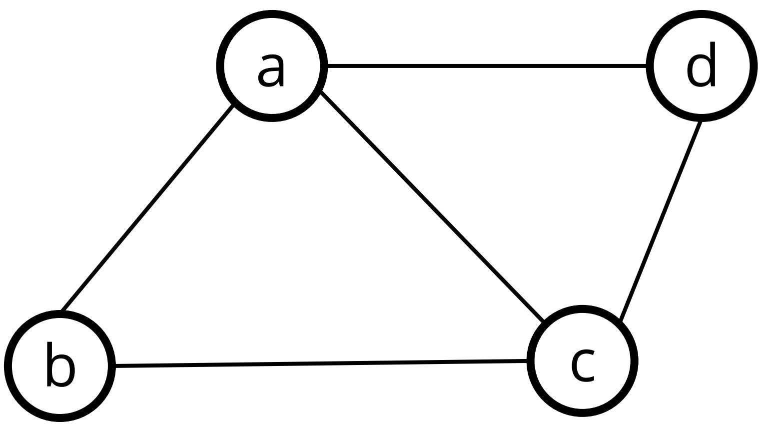


How do we represent graphs in code?

- Two common approaches:
  - Adjacency matrix
  - Adjacency list

Adjacency matrix

- Core idea:
  - Create a $|V| \times |V|$ matrix of booleans or ints
  - The entry A(i, j) is 1 if there is an edge from vertex i to vertex j , and 0 if there isn't.

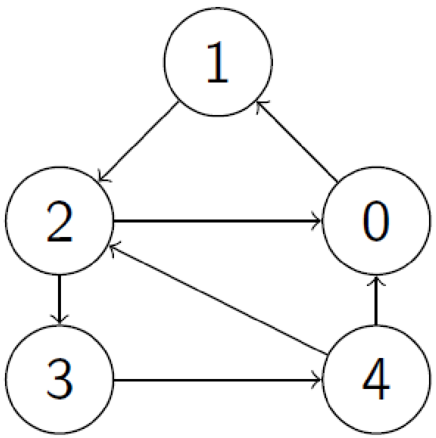


	a	b	c	d
a	0	1	1	1
b	1	0	1	0
c	1	1	0	1
d	1	0	1	0

- Note that the adjacency matrix of an undirected graph is symmetric around the main diagonal.

Adjacency Matrix of a Directed Graph

- Adjacency Matrix:

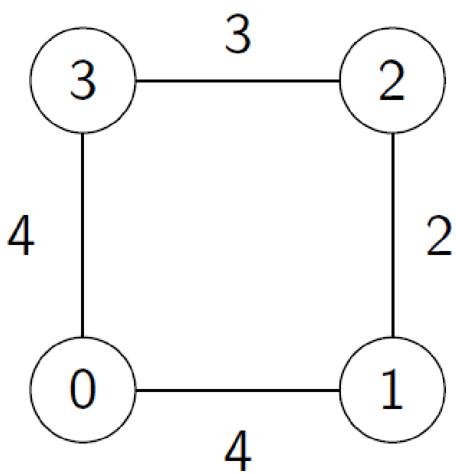


	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	0	0
2	1	0	0	1	0
3	0	0	0	0	1
4	1	0	1	0	0

- Note that the adjacency matrix of a digraph (i.e., directed graph) is not usually symmetric.

Adjacency Matrix of a Weighted Graph

- Weight graph: each edge of a graph has an associated value (i.e., weight).
- If the edges have weights, we can store the weight of edge <i , j> in A(i , j):



Adjacency Matrix:

	0	1	2	3
0	0	4	0	4
1	4	0	2	0
2	0	2	0	3
3	4	0	3	0

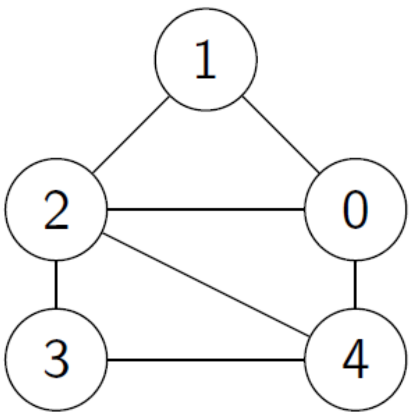
Adjacency Matrix

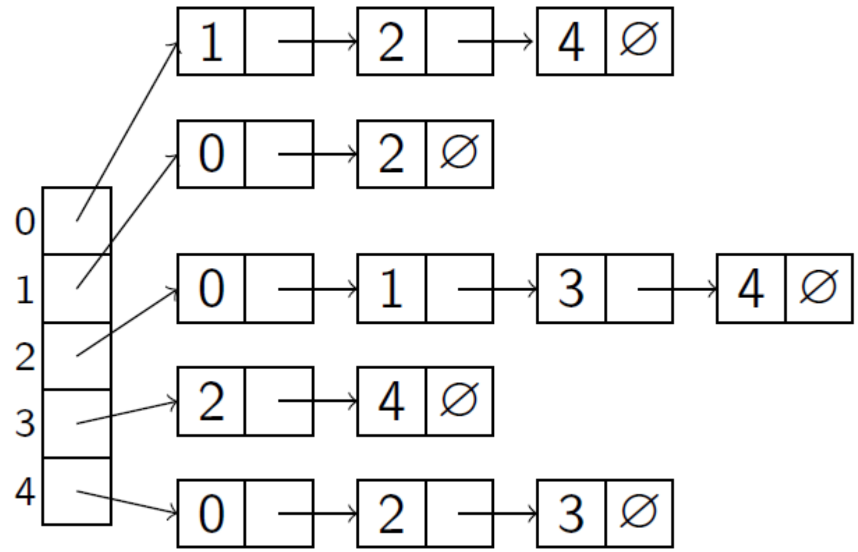
- Primary advantage:
  - Checking to see if there is an edge from vertex i to vertex j = checking entry A(i , j), which takes constant time. (Fast! Easy!)

- Major disadvantage:
  - Usually, a lot of space is wasted storing zeroes.
  - To check adjacent vertices of a given vertex i, we have to look at all entries in row i

Adjacency List of an Undirected Graph

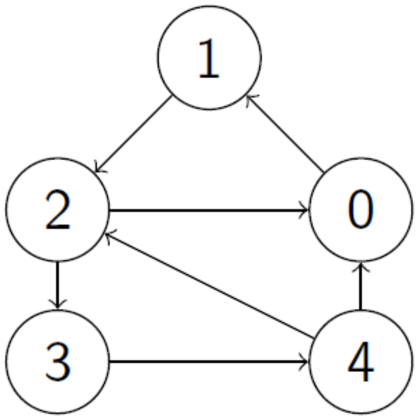
- Adjacency List:

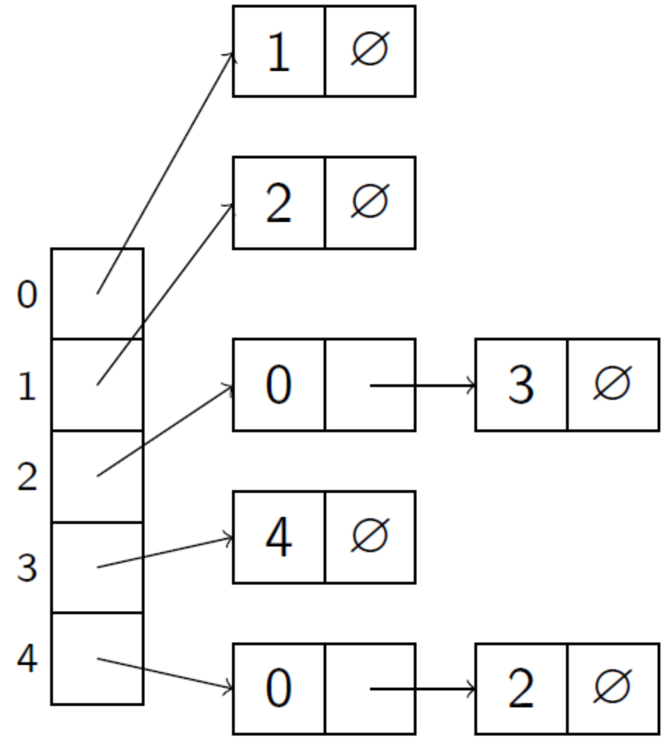




Adjacency List of a Directed Graph

- Adjacency List:

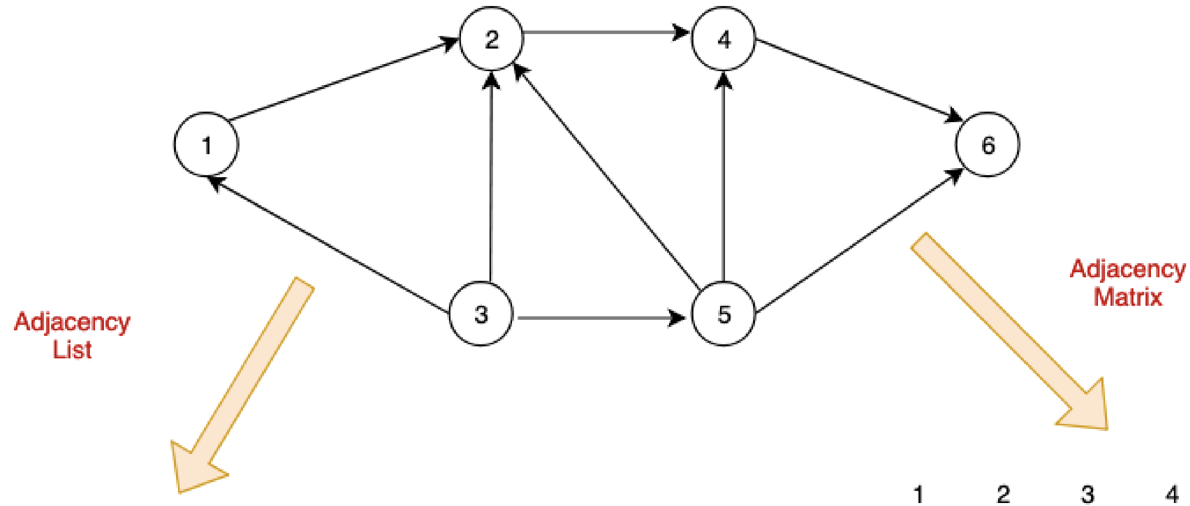


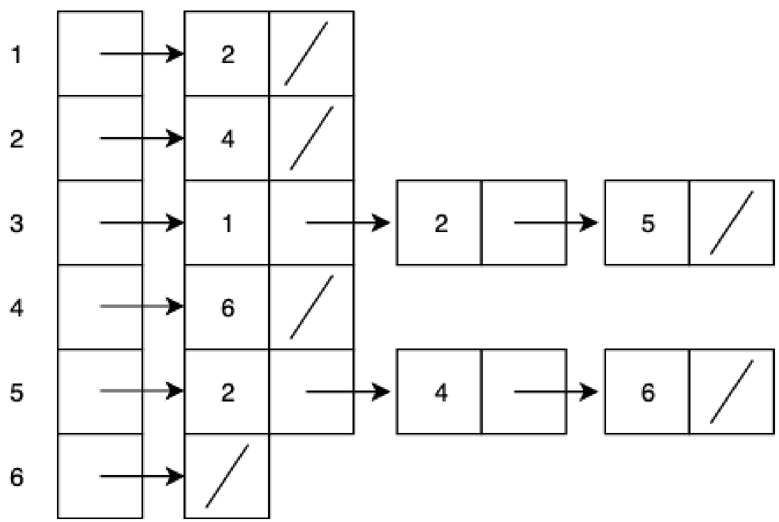


Adjacency List

- We aren't wasting space on edges that aren't there.
- Since vertex i can be connected to all the other vertices, that search can cost O(k), where k is the number of vertices.
- And we don't have to search through edges that aren't there to find all the edges out of a vertex.

Adjacency Matrix vs Adjacency List



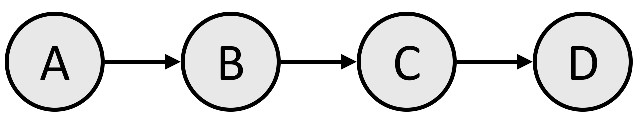


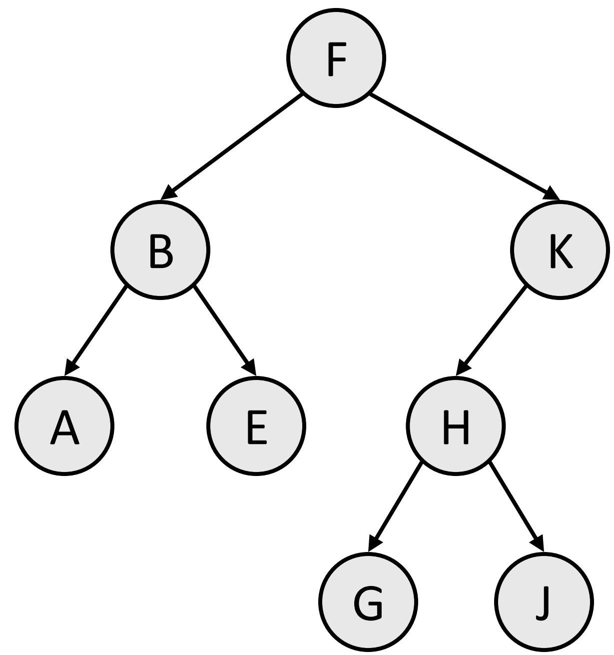
	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	0	1	0	0
3	1	1	0	0	1	0
4	0	0	0	0	0	1
5	0	1	0	1	0	1
6	0	0	0	0	0	0

Linked Lists, Trees, Graphs

- A binary tree is a graph with some restrictions:
  - The tree is an unweighted, acyclic graph
  - There is exactly one path from the root to every node.

- A linked list is also a graph:
  - Unweighted Directed Acyclic Graph.
  - In/out degree of at most 1 for all nodes.





Traversing in a Graph

- The most common operation is to visit all the vertices in a systematic way.
- Two types of traversals:
  - **Depth-first traversal (or depth-first search)**
  - **Breadth-first traversal (or breadth-first search)**
- A traversal starts at a vertex *v* and visits all the vertices *u* such that a path exists from *v* to *u*.

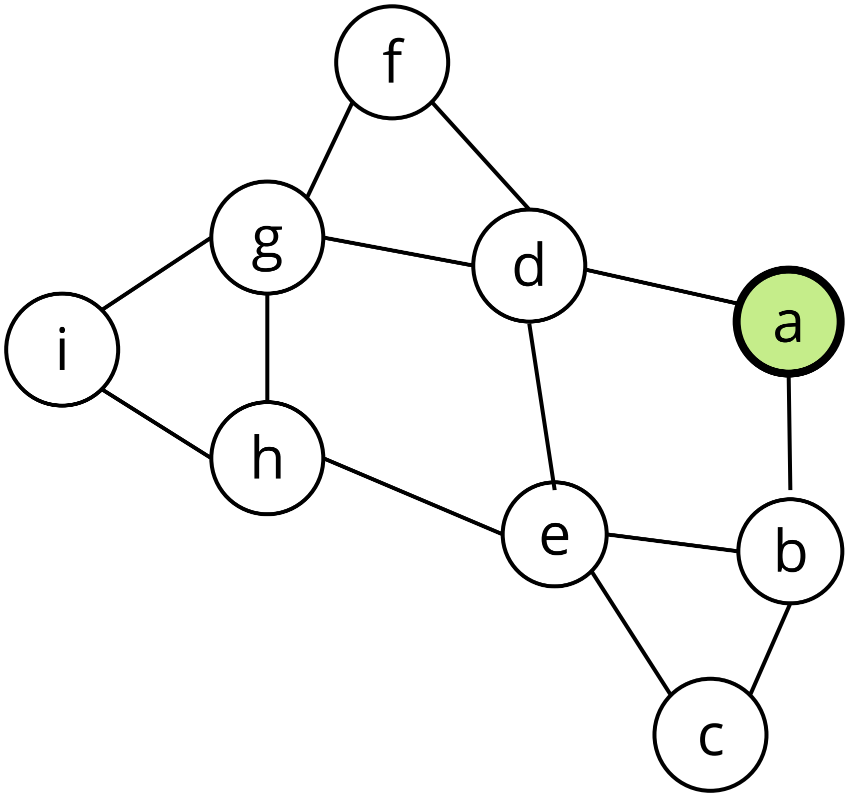
Breadth-first search (BFS)

- A breadth-first traversal visits all nearby vertices first before moving farther away.
- It is a queue-based, iterative traversal.

Pseudocode: Breadth-first search

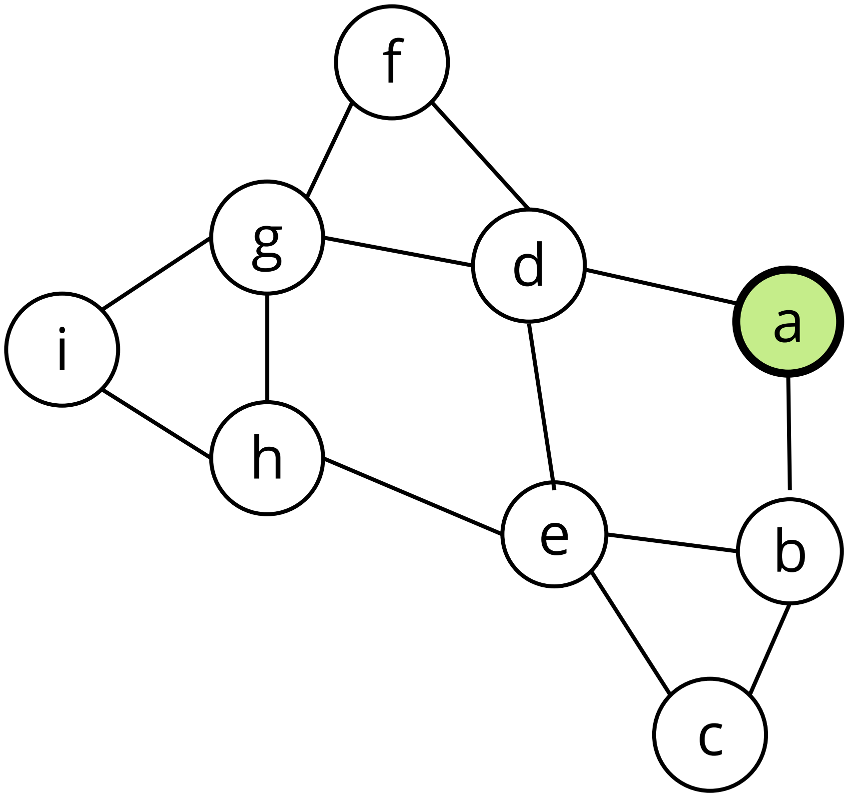
```
search(s):
    visited = empty set
    queue.enqueue(s)
    visited.add(s)

    while (queue is not empty):
        v = queue.dequeue()
        for (w : v.neighbors()):
            if (w not in visited):
                queue.enqueue(w)
                visited.add(w)
```

Breadth-first search (BFS) example



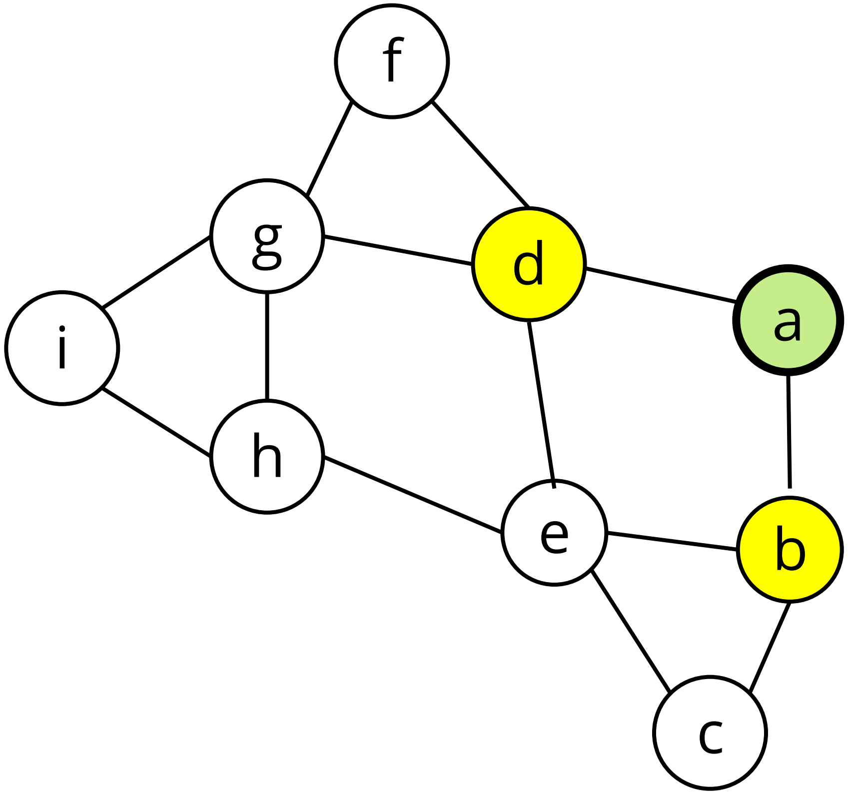
- Current node (v):
- Queue: a
- Visited: a

Breadth-first search (BFS) example



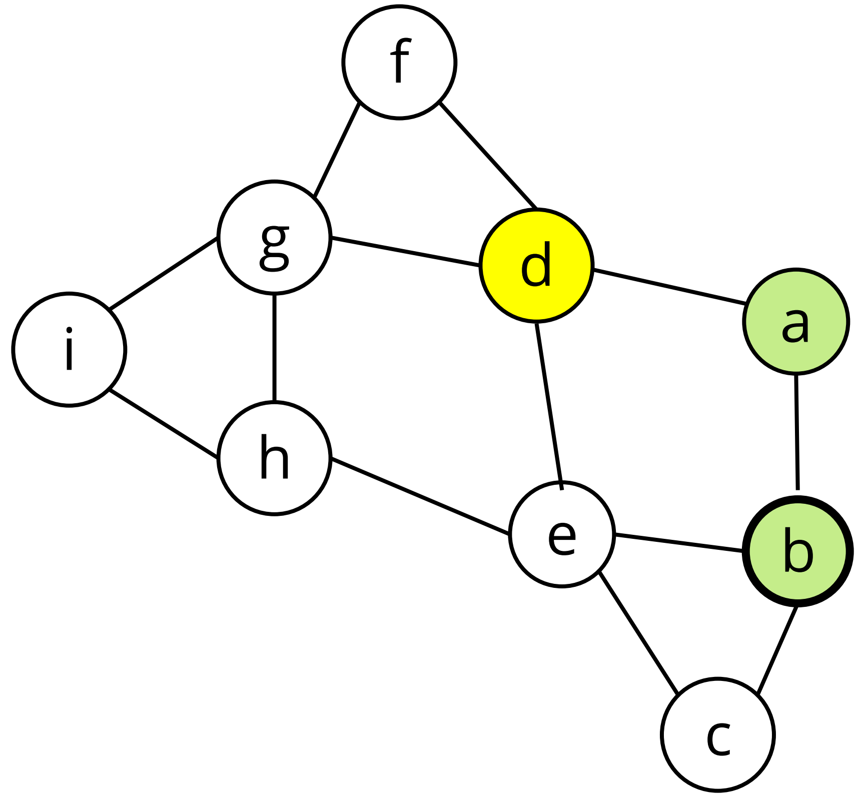
- Current node (v): a
- Queue:
- Visited: a

Breadth-first search (BFS) example



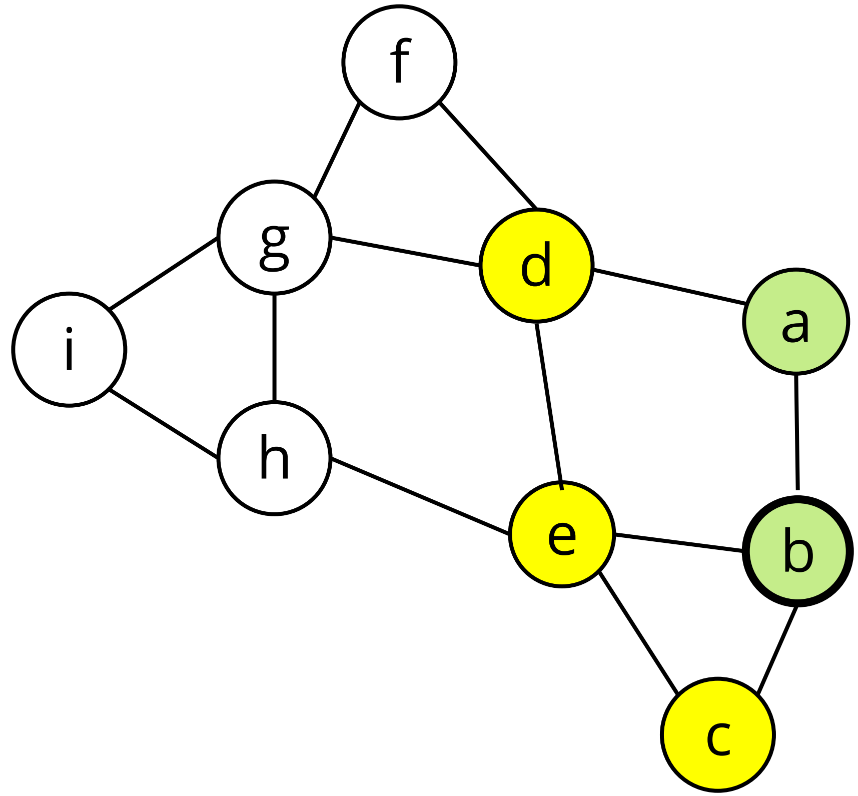
- Current node (v): a
- Queue: b, d
- Visited: a, b, d

Breadth-first search (BFS) example



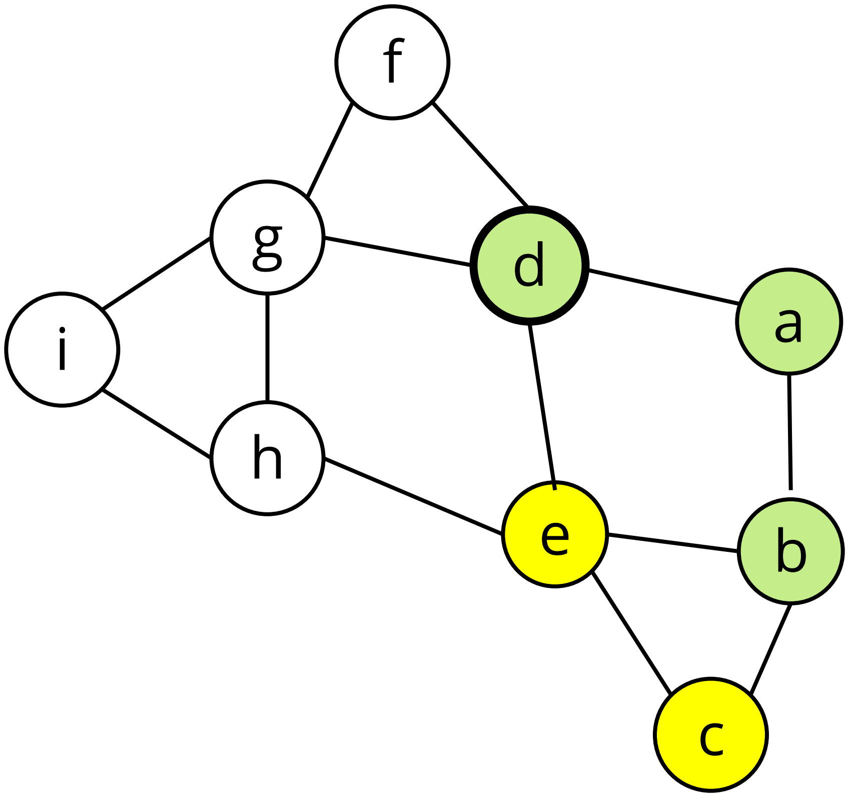
- Current node (v): b
- Queue: d
- Visited: a, b, d

Breadth-first search (BFS) example



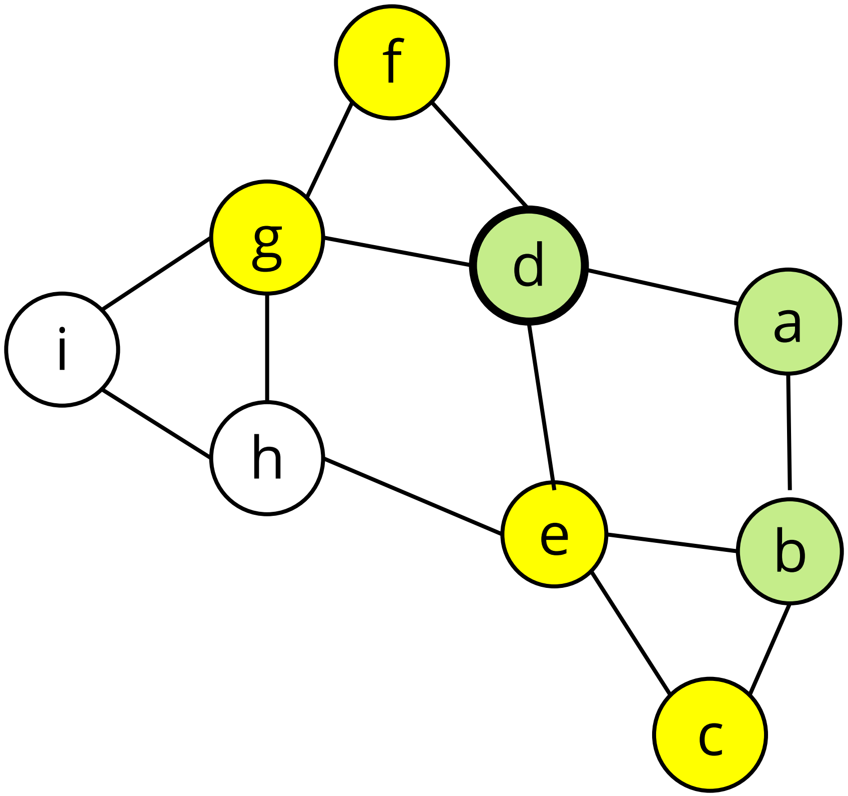
- Current node (v): b
- Queue: d, c, e
- Visited: a, b, d, c, e

Breadth-first search (BFS) example



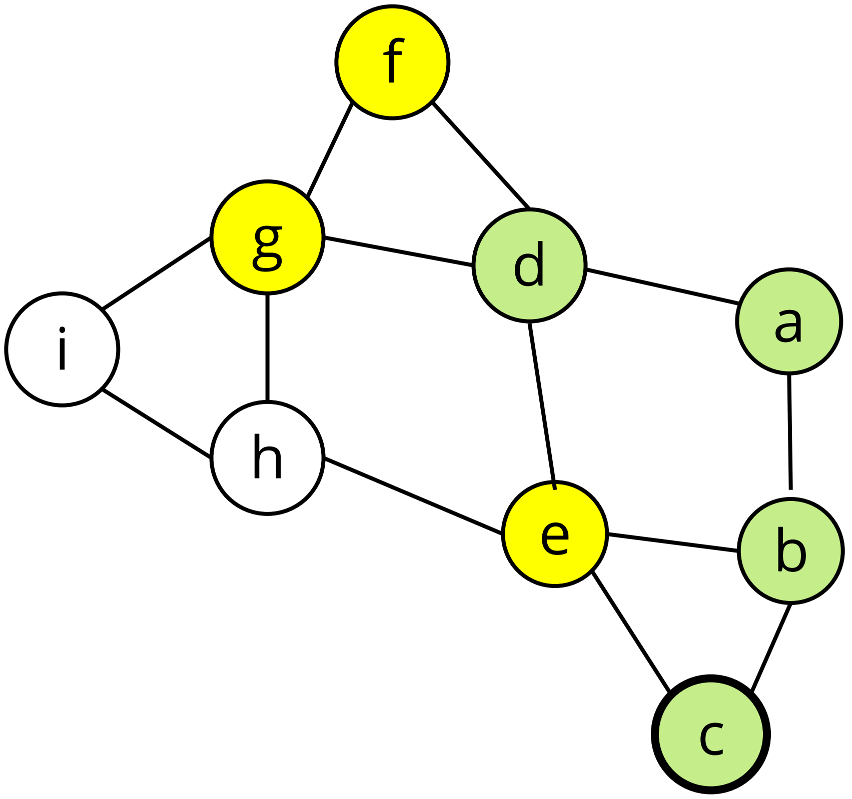
- Current node (v): d
- Queue: c, e
- Visited: a, b, d, c, e

Breadth-first search (BFS) example



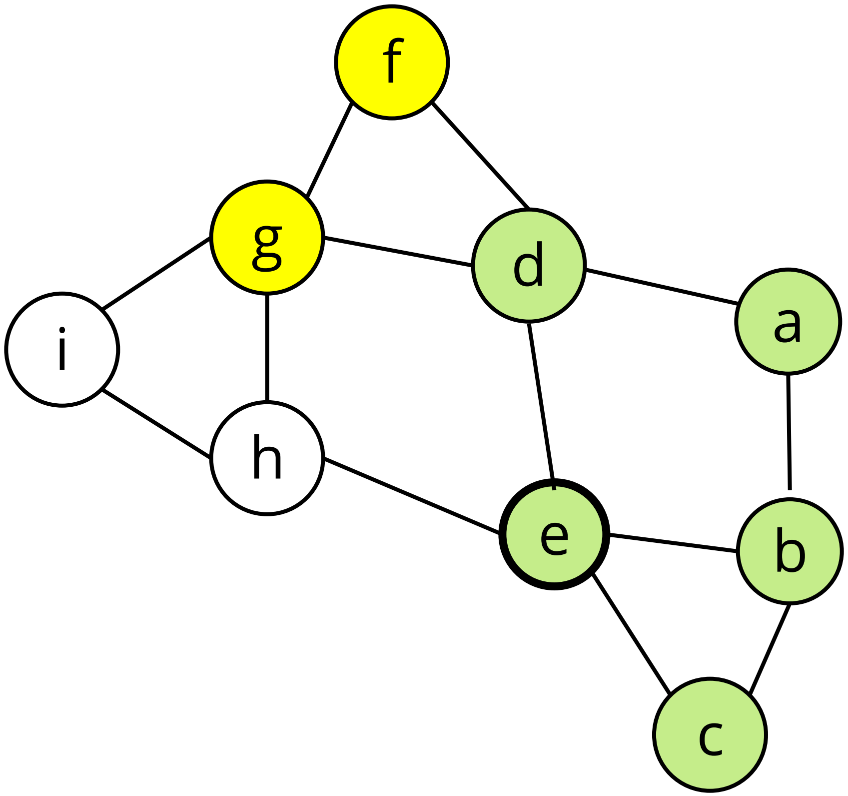
- Current node (v): d
- Queue: c, e, f, g
- Visited: a, b, d, c, e, f, g

Breadth-first search (BFS) example



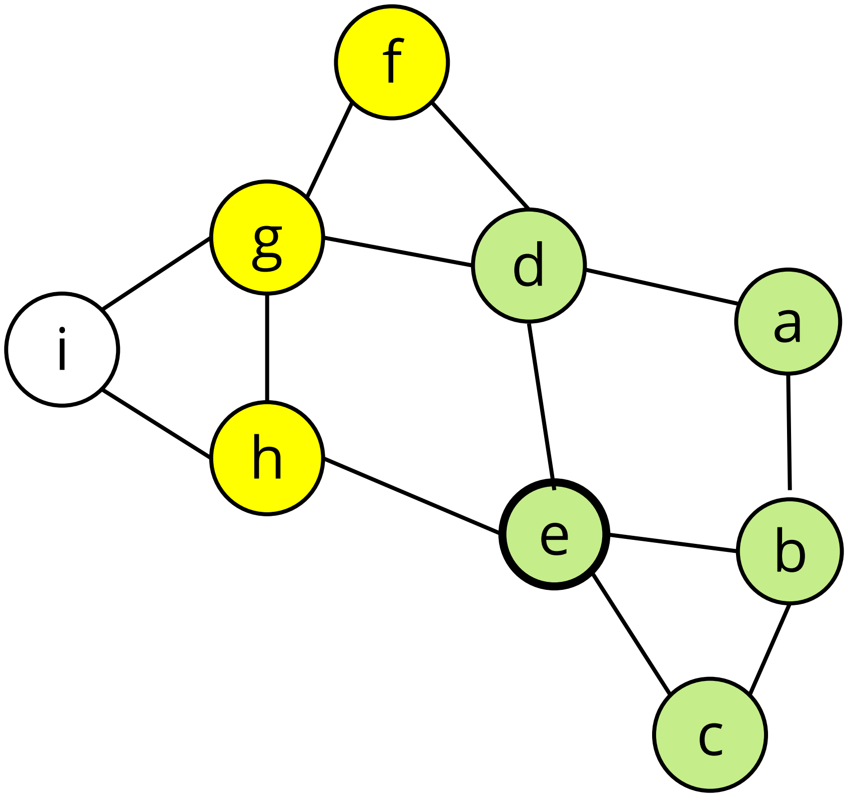
- Current node (v): c
- Queue: e, f, g
- Visited: a, b, d, c, e, f, g

Breadth-first search (BFS) example



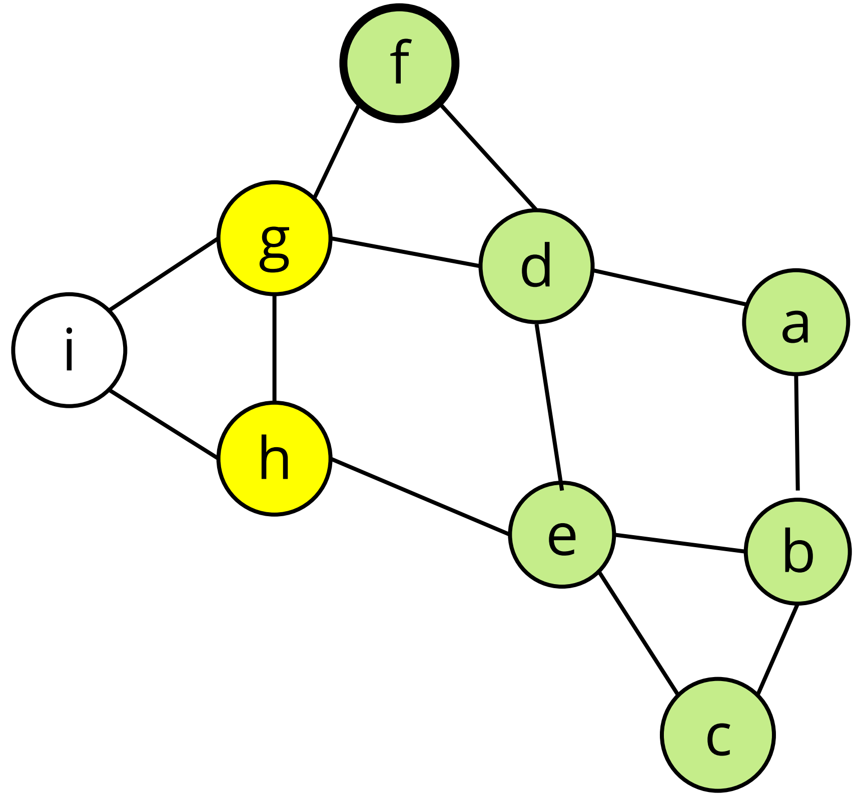
- Current node (v): e
- Queue: f, g
- Visited: a, b, d, c, e, f, g

Breadth-first search (BFS) example



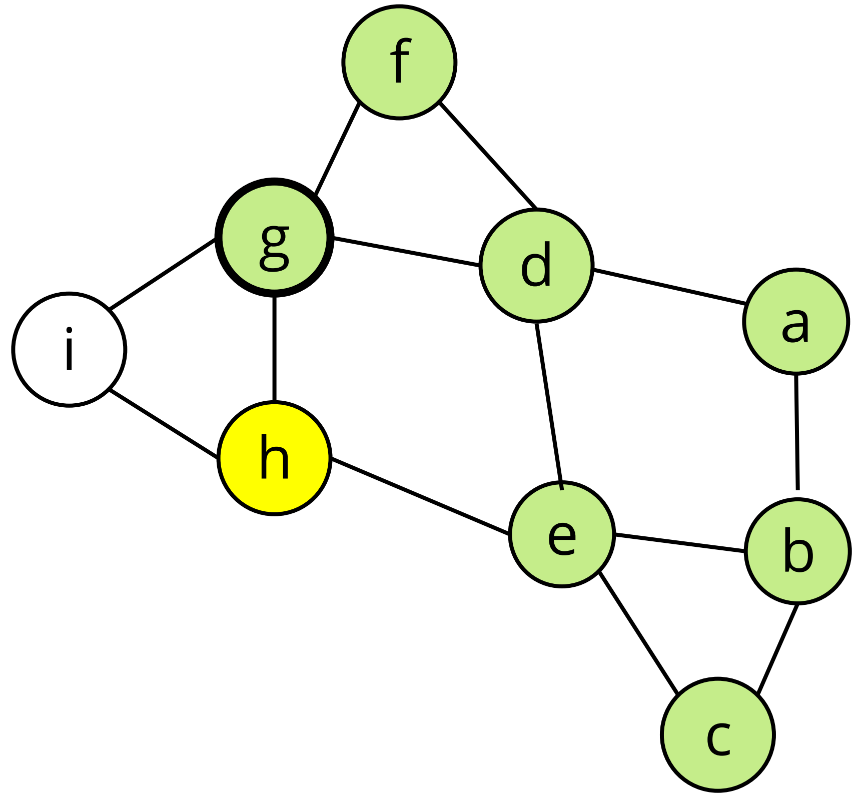
- Current node (v): e
- Queue: f, g, h
- Visited: a, b, d, c, e, f, g, h

Breadth-first search (BFS) example



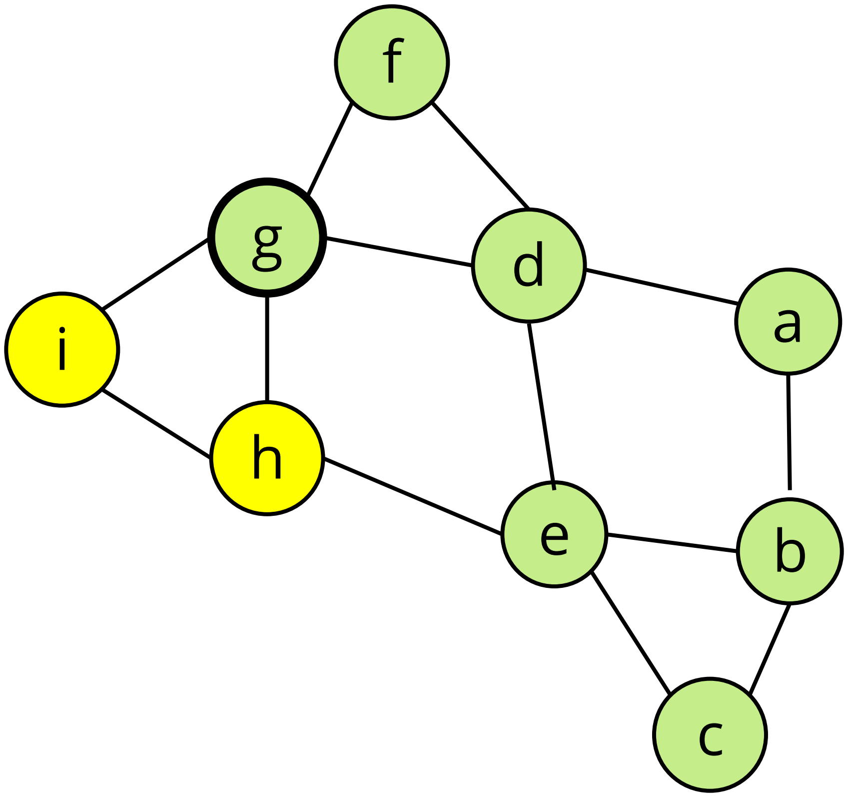
- Current node (v): f
- Queue: g, h
- Visited: a, b, d, c, e, f, g, h

Breadth-first search (BFS) example



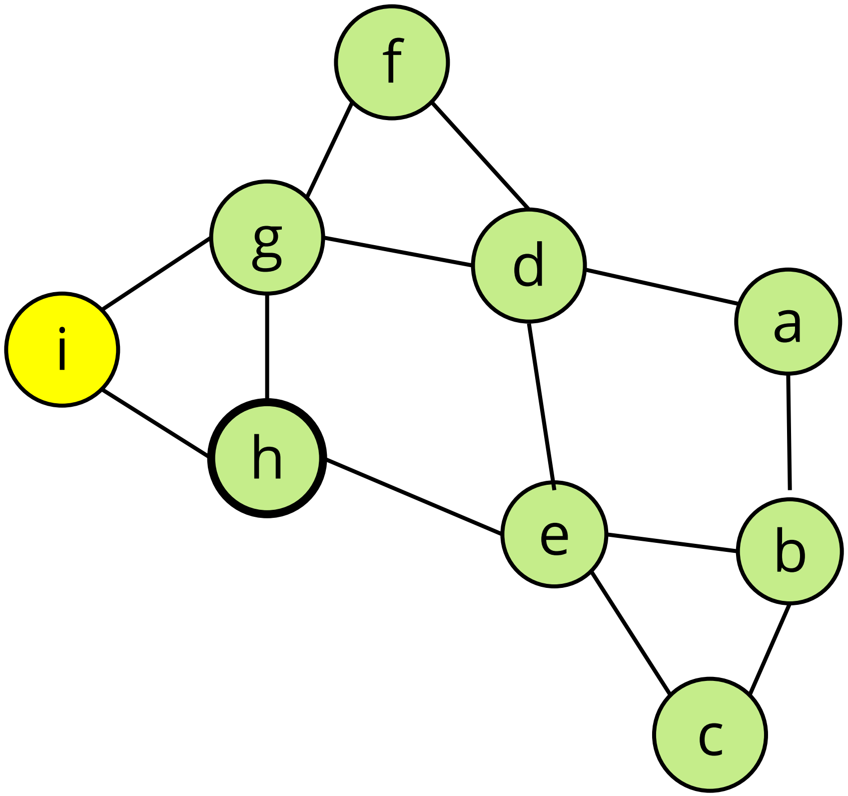
- Current node (v): g
- Queue: h
- Visited: a, b, d, c, e, f, g, h

Breadth-first search (BFS) example



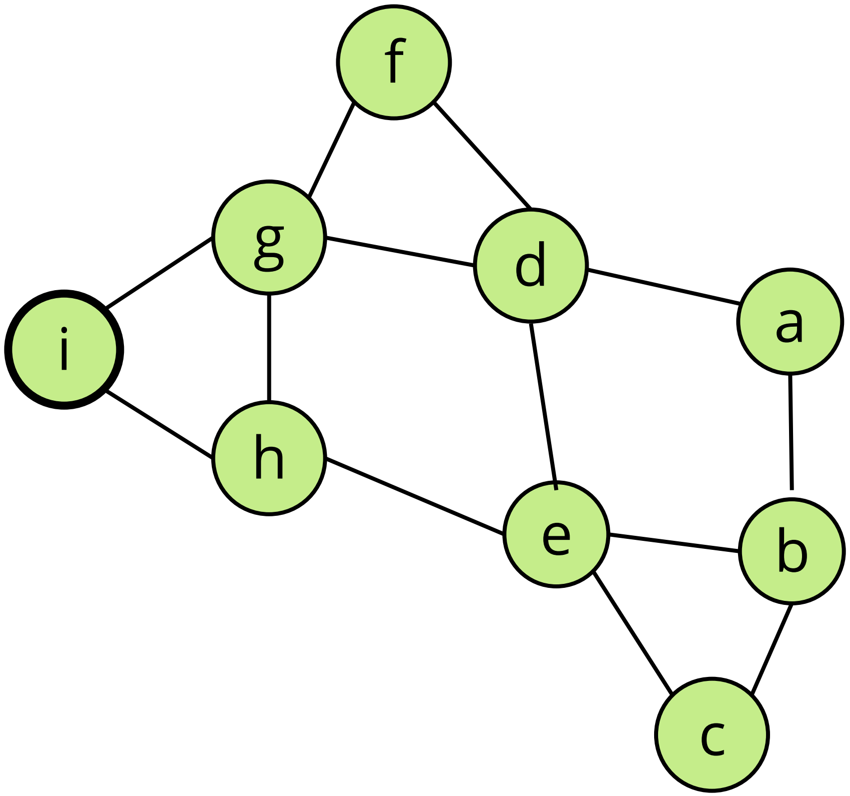
- Current node (v): g
- Queue: h, i
- Visited: a, b, d, c, e, f, g, h, i

Breadth-first search (BFS) example



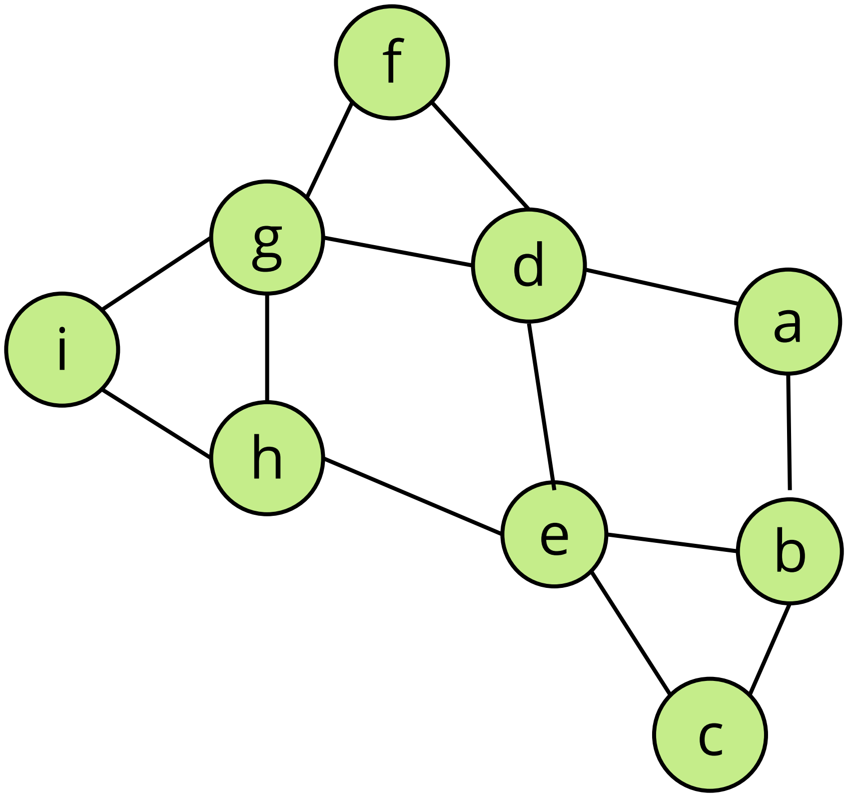
- Current node (v): h
- Queue: i
- Visited: a, b, d, c, e, f, g, h, i

Breadth-first search (BFS) example



- Current node (v): i
- Queue:
- Visited: a, b, d, c, e, f, g, h, i

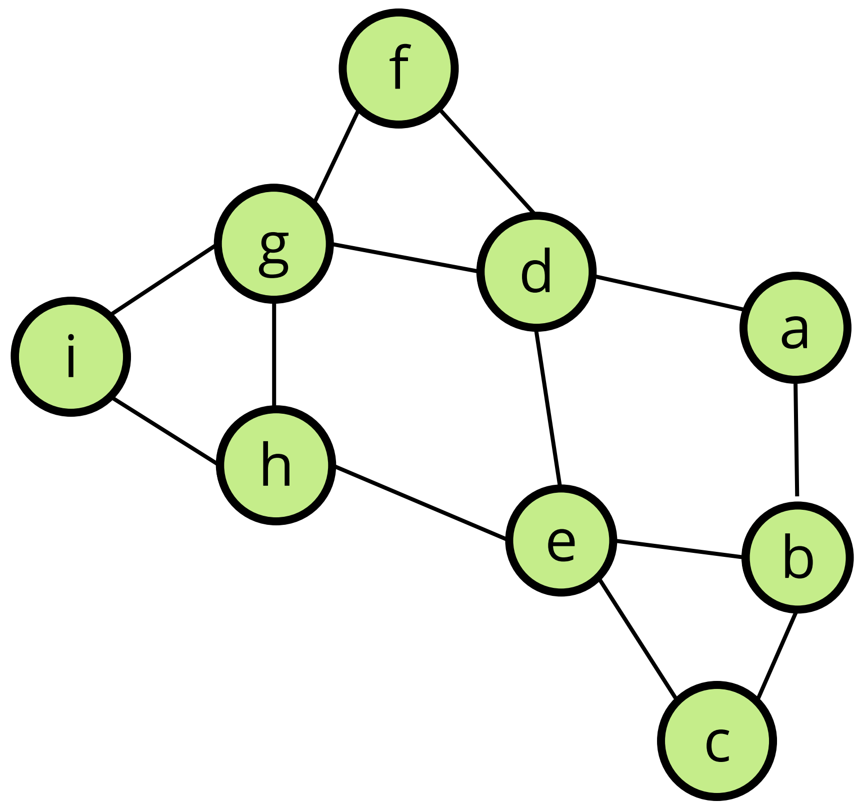
Breadth-first search (BFS) example



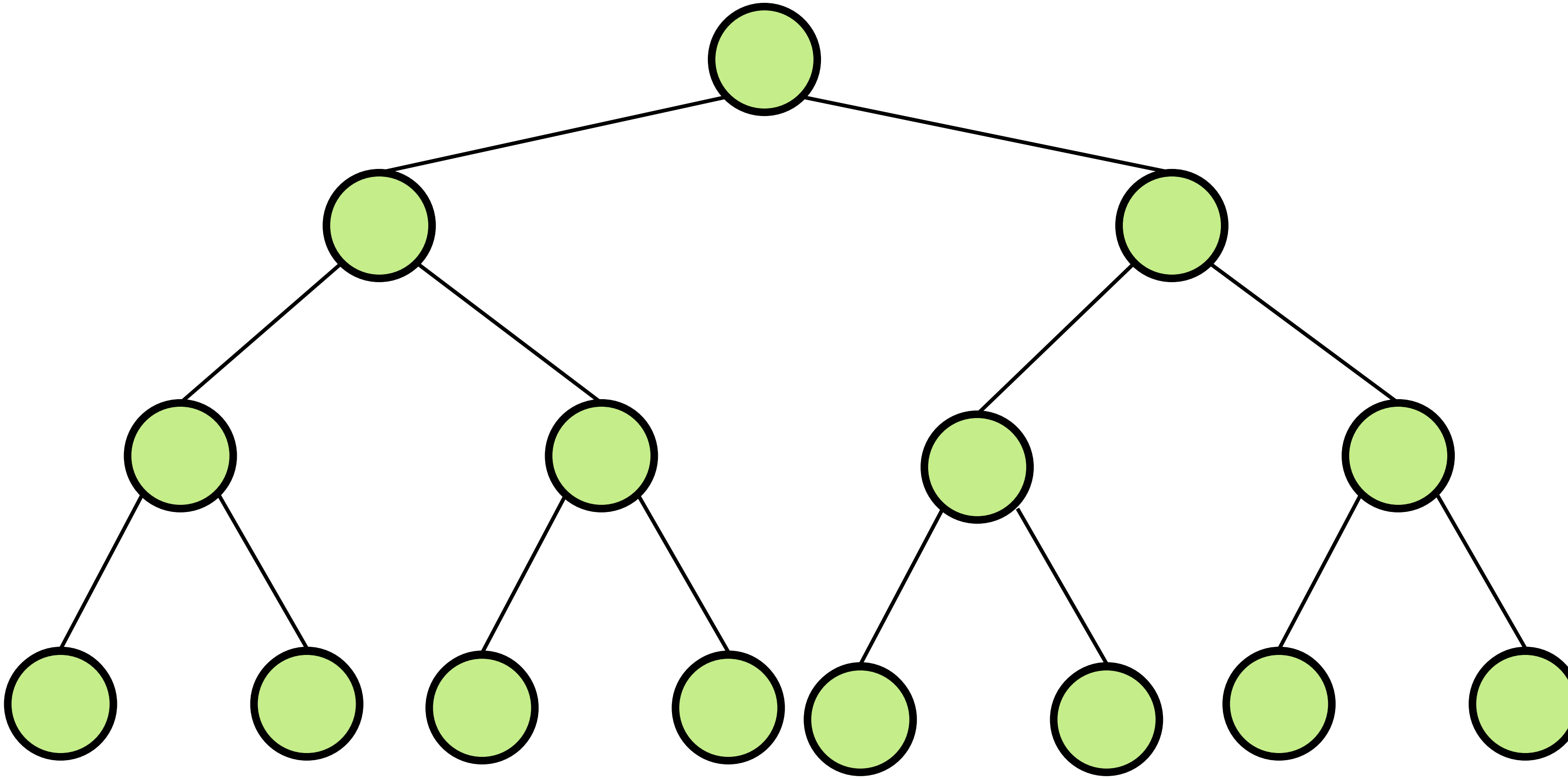
- Current node (v):
- Queue:
- Visited: a, b, d, c, e, f, g, h, i

An interesting property

- Note: We visited the nodes in “rings” – maintained a gradually growing “frontier” of nodes.

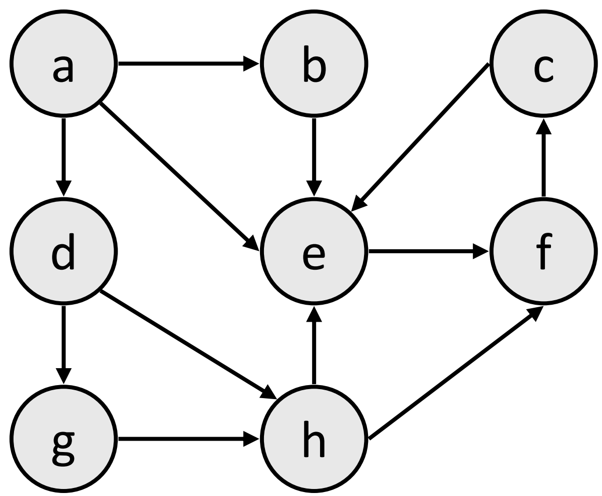


What does this look like for trees?



BFS observations

- Optimality:
  - always finds the shortest path (fewest edges).
  - in unweighted graphs, finds optimal cost path.
  - In weighted graphs, not always optimal cost.



Review

Review

Which is the correct definition of Big-Oh?

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers.

1. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ *such that* $f(n) \geq c \cdot g(n)$, for $n \geq n_0$.
2. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ *such that* $f(n) \leq c \cdot g(n)$, for $n \geq n_0$.
3. We say that $f(n)$ is $O(g(n))$ if there are constants c' > 0 and c'' > 0 and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$, for $n \geq n_0$.

Review

1. $n + \log n^2 + n^3 = O(\_\_)$
2. $2 + \log n + \log^k n + n \log n = O(\_\_)$

3.
```
for (int i = 1; i <= n * n; i++)
      System.out.println("i: " + i);
```
$= O(\_\_)$

4.
```
for(int j = 1; j <= n; j = j * 2)
      System.out.println(“j: " + j);
```
$= O(\_\_)$

5.
```
for (int i = 1; i <= n; i++)
  for(int j = 1; j <= n; j = j++)
      for(int k = 1; k <= n; k = k + 2)
            System.out.println("i: " + i + "j: " + j + "k: " + k);
```
$= O(\_\_)$

Review

True or false?

1. An nlogn function grows less rapidly (with the input size) than a quadratic function.

2. A Stack ADT can only be implemented using arrays.

3. ArrayList <Object> can store a list of any kind of data.

4. A linked list does not require shifting elements when adding a new element to the middle of the list.

Review

1. Which method is used to return the element at the front of the queue without removing it?
   a) enqueue
   b) dequeue
   c) first
   d) top
2. Which of the following are good uses for a Stack ADT?
   a) checking whether an expression contains evenly matched pairs of parenthesis.
   b) evaluating a postfix expression
   c) evaluating a prefix expression
   d) implementing a ring buffer (a FIFO data structure that loops back to zero after is reaches the buffer length)
   e) serving items in order of priority

Questions?