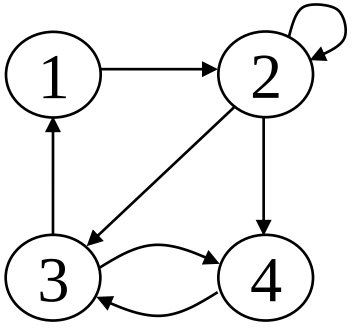
Chapter22. Graph

Graphs

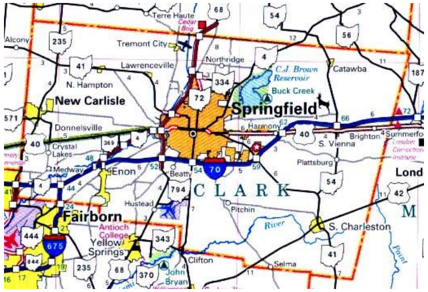
Definition = a set of nodes (vertices) with edges (links) between them.

- $G = (V, E)$: graph
- V = set of vertices $\quad |V| = n$
- E = set of edges $|E| = m$
  - Binary relation on V
  - Subset of $V \times V = \{(u,v): u \in V, v \in V\}$



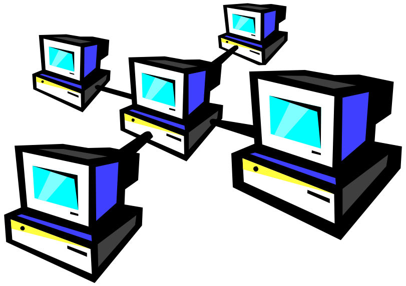
Applications

- Applications that involve not only a set of items, but also the connections between them

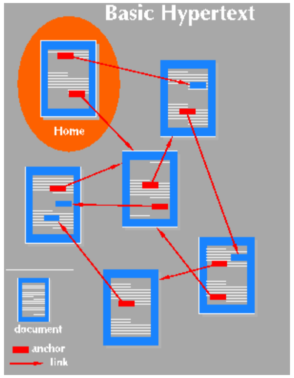


Maps

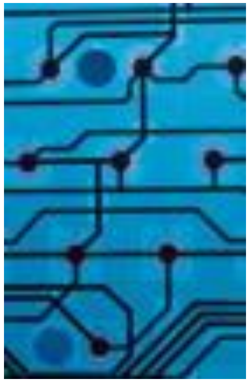
Schedules



Computer networks



Hypertext



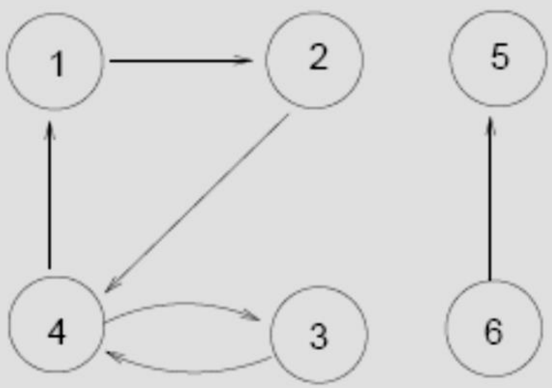
Circuits

Terminology

- Directed vs Undirected graphs

Directed graphs (digraphs)

(ordered pairs of vertices)



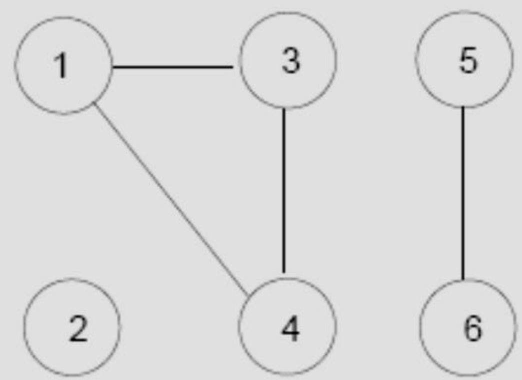
in-degree of v: # edges enetring v

out-degree of v: # edges leaving v

v is **adjacent** to u if there is an edge (u,v)

Undirected graphs

(unordered pairs of vertices)

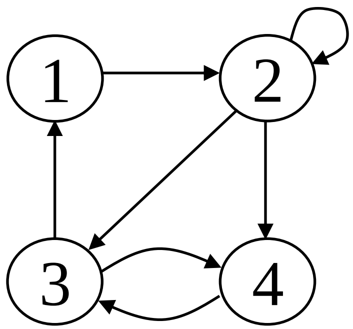


degree of v: # edges incident on v

v is **adjacent** to u and u is **adjacent** to v
if there is and edge (u,v)

Terminology (cont'd)

- Complete graph
  - A graph with an edge between each pair of vertices
- Subgraph
  - A graph (V', E') such that $V' \subseteq V$ and $E' \subseteq E$
- Path from v to w
  - A sequence of vertices $<v_0, v_1, \ldots, v_k>$ such that $v_0=v$ and $v_k=w$
- Length of a path
  - Number of edges in the path

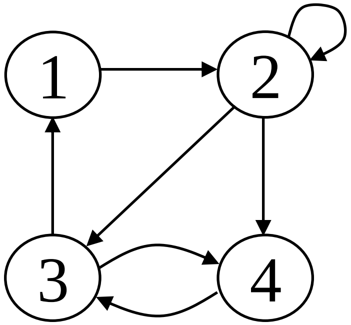


path from v_1 to v_4

$<v_1, v_2, v_4>$

Terminology (cont'd)

- w is reachable from v
  - If there is a path from v to w
- Simple path
  - All the vertices in the path are distinct
- Cycles
  - A path $<v_0, v_1, \ldots, v_k>$ forms a cycle if $v_0=v_k$ and $k\geq 2$
- Acyclic graph
  - A graph without any cycles



cycle from v_1 to v_1

$<v_1, v_2, v_3, v_1>$

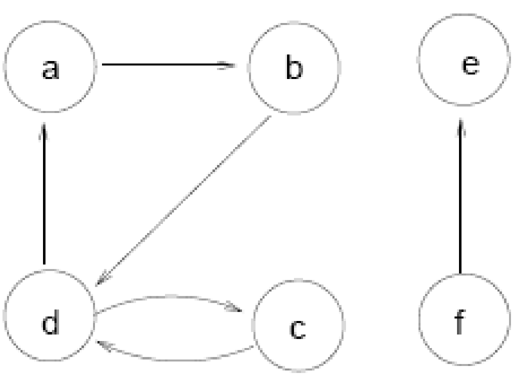
Terminology (cont'd)

Connected and Strongly Connected

directed graphs

strongly connected: every two vertices are reachable from each other

strongly connected components : all possible strongly connected subgraphs

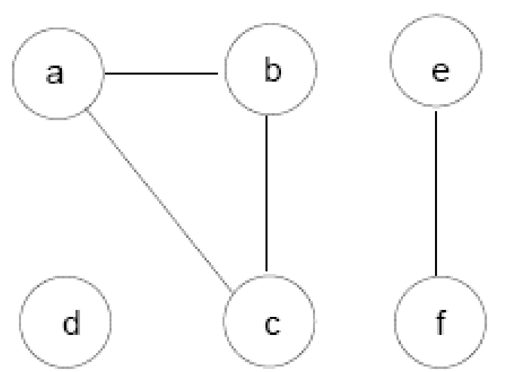


strongly connected components: {a,b,c,d} { e} {f}

undirected graphs

connected: every pair of vertices is connected by a path

connected components: all possible connected subgraphs

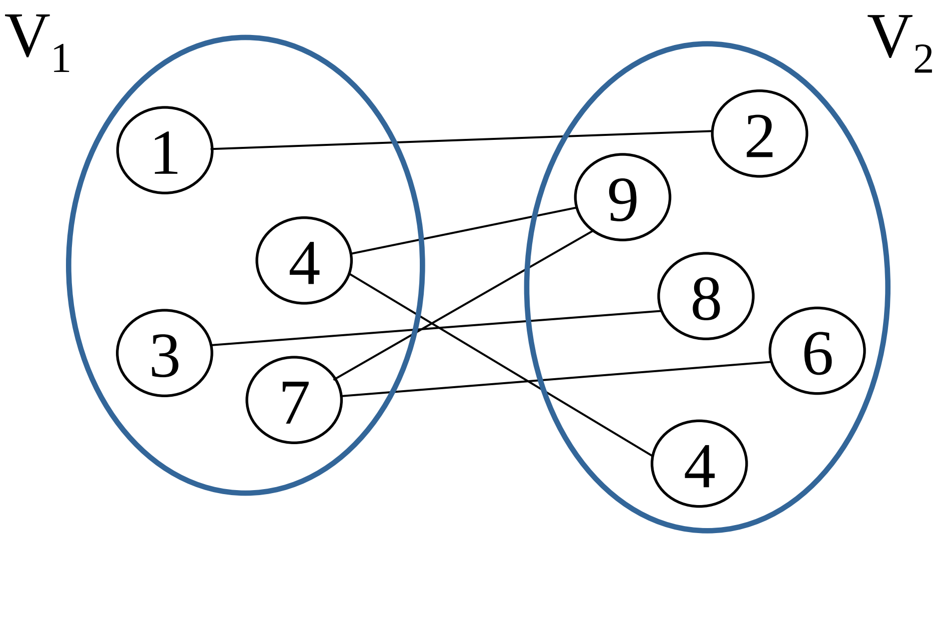


connected components: {a,b,c} {d} {e,f}

Terminology (cont'd)

- A **bipartite graph** is an undirected graph

  G = (V, E) in which $V = V_1 + V_2$ and there are edges only between vertices in V_1 and V_2

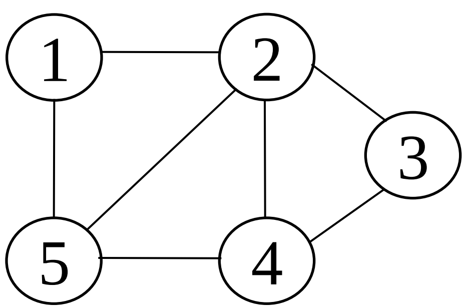


Graphs

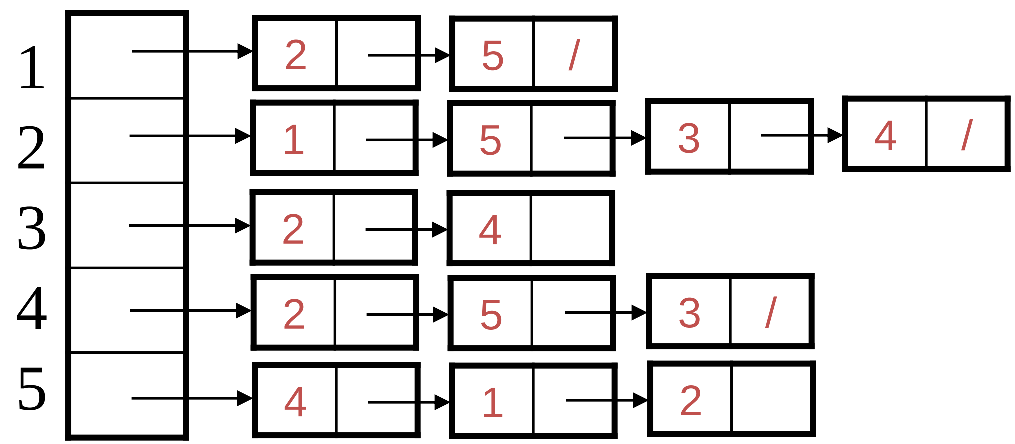
- We will typically express running times in terms of |E| and |V|
  - If $|E| \approx |V|^2$, the graph is *dense*
  - If $|E| \approx |V|$, the graph is *sparse*
- If you know you are dealing with dense or sparse graphs, different data structures may make sense

Graph Representation

- **Adjacency list representation** of G = (V, E)
  - An array of |V| lists, one for each vertex in V
  - Each list Adj[u] contains all the vertices v that are adjacent to u (i.e., there is an edge from u to v)
  - Can be used for both directed and undirected graphs

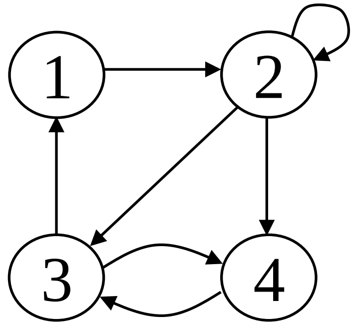


Undirected graph

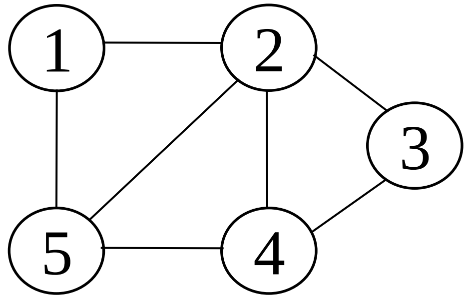


Properties of Adjacency List Representation

- Sum of “lengths” of all adjacency lists
  - Directed graph: $|E|$
    - edge (u, v) appears only once (i.e., in the list of **u**)
  - Undirected graph: $2|E|$
    - edge (u, v) appears twice (i.e., in the lists of both **u** and **v**)



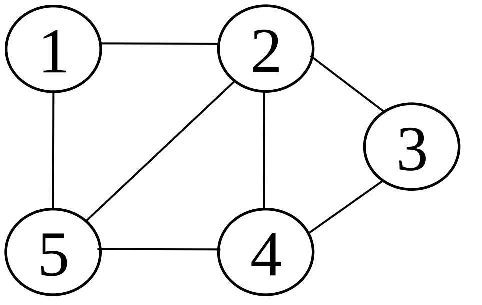
Directed graph



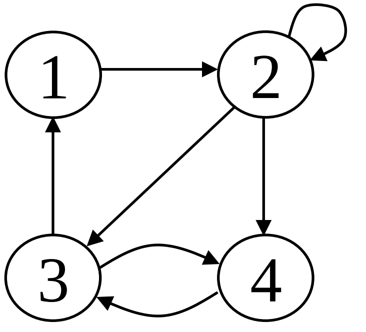
Undirected graph

Properties of Adjacency-List Representation

- Memory required
  - $\Theta(V + E)$
- Preferred when
  - The graph is **sparse**: $|E| << |V|^2$
  - We need to quickly determine the nodes adjacent to a given node.
- Disadvantage
  - No quick way to determine whether there is an edge between node u and v
- Time to determine if $(u, v) \in E$:
  - $O(\text{degree}(u))$
- Time to list all vertices adjacent to u:
  - $\Theta(\text{degree}(u))$



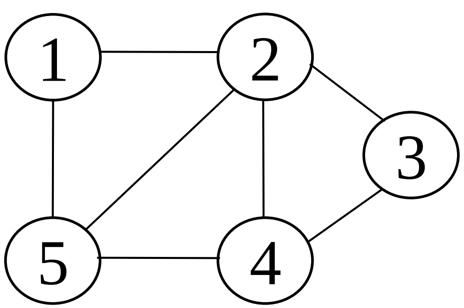
Undirected graph



Directed graph

Graph Representation

- **Adjacency matrix representation** of G = (V, E)
  - Assume vertices are numbered 1, 2, … $|V|$
  - The representation consists of a matrix $A_{|V| \times |V|}$:
  - $a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$



Undirected graph

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

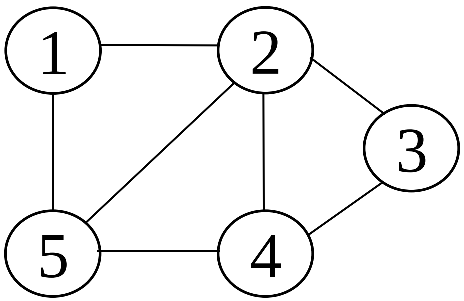
For undirected graphs, matrix A is symmetric:

$a_{ij} = a_{ji}$

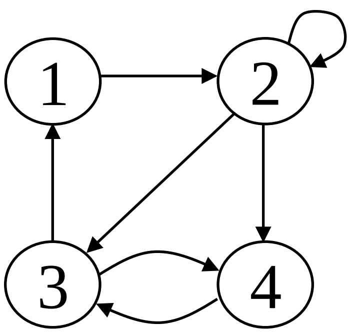
$A = A^T$

Properties of Adjacency Matrix Representation

- Memory required
  - $\Theta(V^2)$, independent on the number of edges in G
- Preferred when
  - The graph is **dense:** $|E|$ is close to $|V|^2$
  - We need to quickly determine if there is an edge between two vertices
- Time to determine if $(u, v) \in E$:
  - $\Theta(1)$
- Disadvantage
  - No quick way to determine the vertices adjacent to another vertex
- Time to list all vertices adjacent to u:
  - $\Theta(V)$



Undirected graph



Directed graph

Graphs: Adjacency Matrix

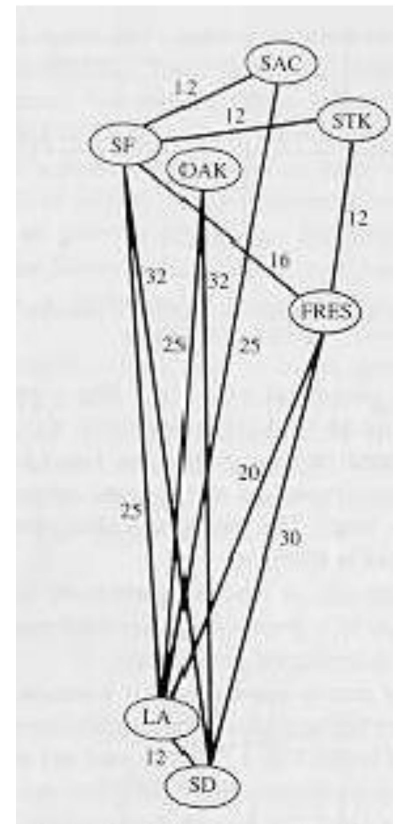
- *How much storage does the adjacency matrix require?*
- Ans: $O(V^2)$
- *What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 4 vertices?*
- Ans: 6 bits
  - Undirected graph $\rightarrow$ matrix is symmetric
  - No self-loops $\rightarrow$ don't need diagonal

Weighted Graphs

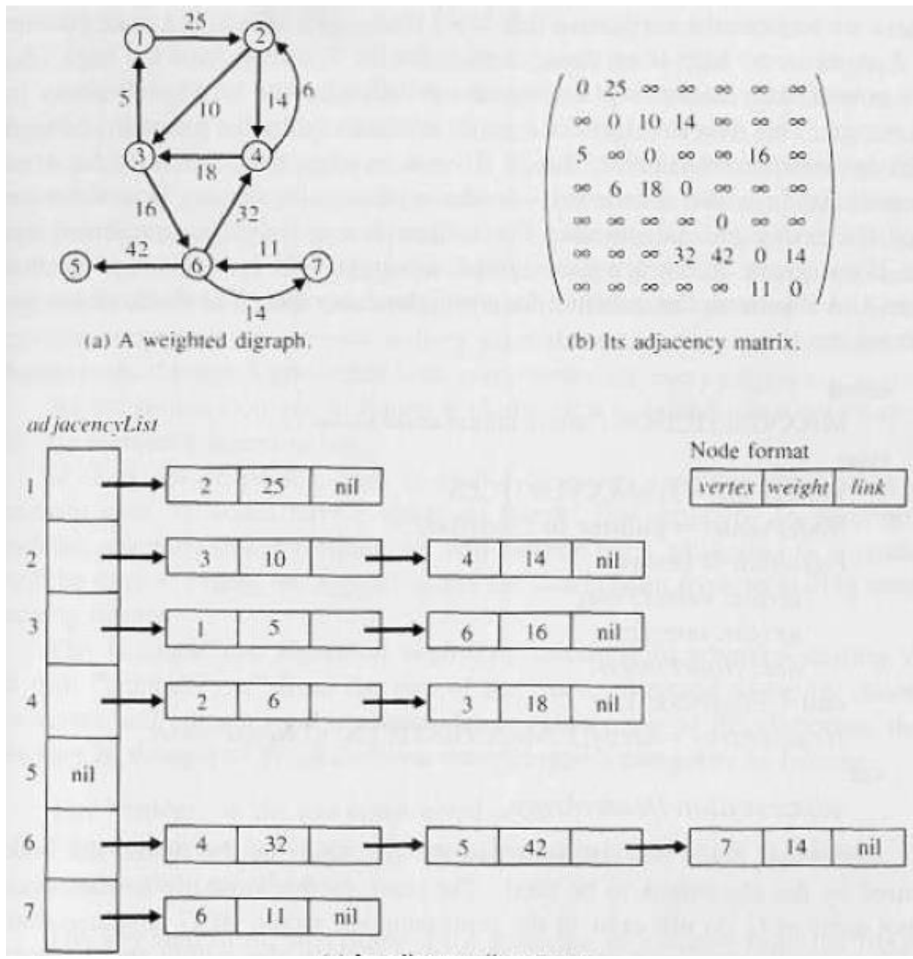
- Graphs for which each edge has an associated weight w(u, v)

  w: $E \rightarrow$ R, weight function

- Storing the weights of a graph
  - Adjacency list:
    - Store w(u,v) along with vertex v in u's adjacency list
  - Adjacency matrix:
    - Store w(u, v) at location (u, v) in the matrix



Weighted Graphs



Graph traversals

- Graph traversal algorithm: visit some or all of the nodes in a graph, labeling them with useful information
  - breadth-first: useful for undirected, yields connectivity and shortest-paths information
  - depth-first: useful for directed, yields numbering used for
    - topological sort
    - strongly-connected component decomposition

Breadth First Search(BFS)

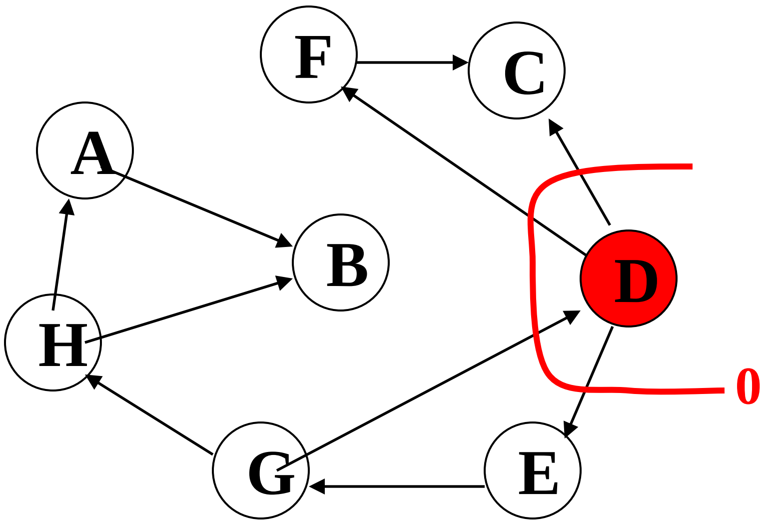
- Given
  - a graph G=(V,E) : set of vertices and edges
  - a distinguished source vertex s
- Breadth first search systematically explores the edges of G to discover every vertex that is reachable from s.
- It also produces a ‘breadth first tree’ with root s that contains all the vertices reachable from s.
- For any vertex v reachable from s, the path in the breadth first tree corresponds to the shortest path in graph G from s to v.
- It works on both directed and undirected graphs. However, we will explore only directed graphs.

BFS

- It is so named because it discovers all vertices at distance k from s before discovering vertices at distance k+1.

- http://en.wikipedia.org/wiki/Image:Animated_BFS.gif

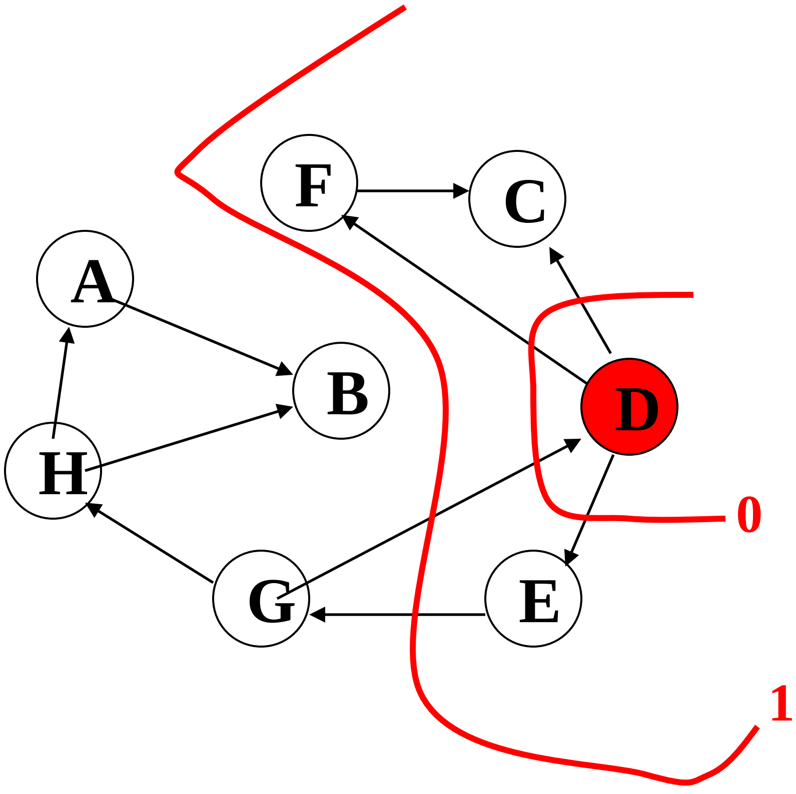
BFS Overview



Breadth-first search starts with given node

Task: Conduct a breadth-first search of the graph starting with node D

BFS Overview



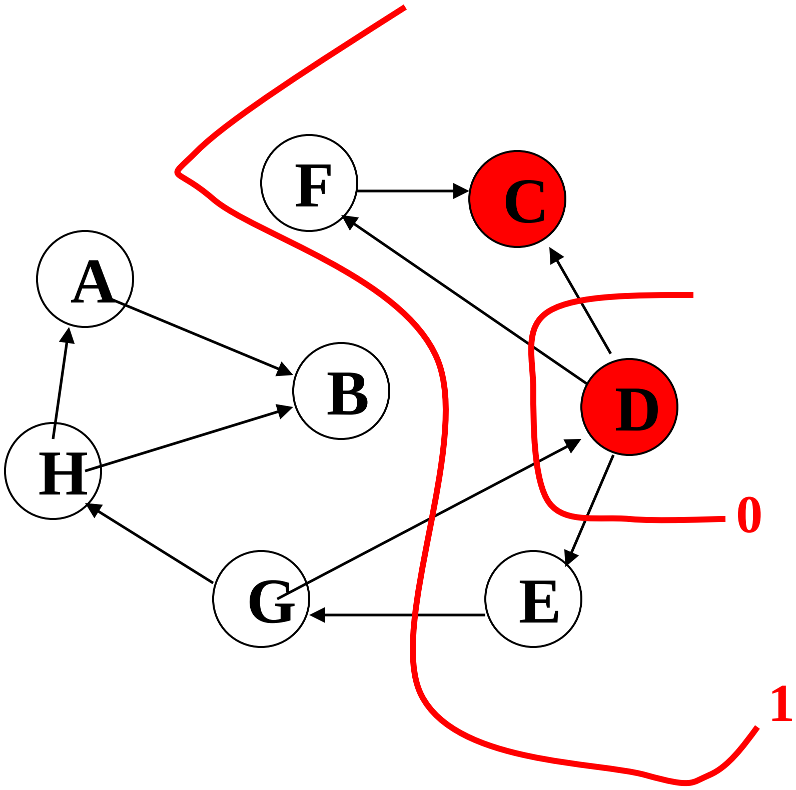
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D

BFS Overview



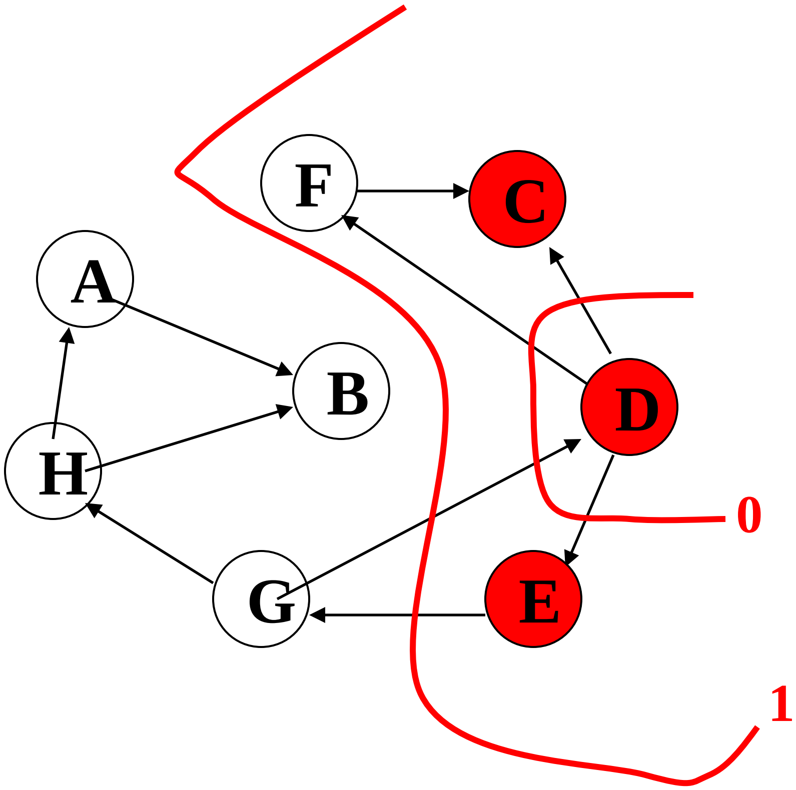
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D, C

BFS Overview



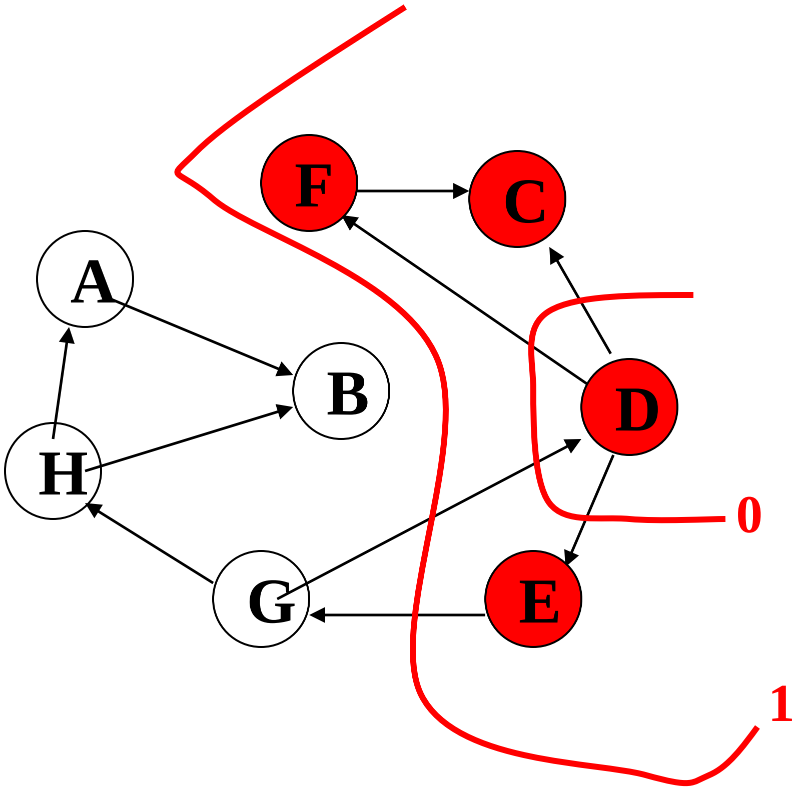
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D, C, E

BFS Overview



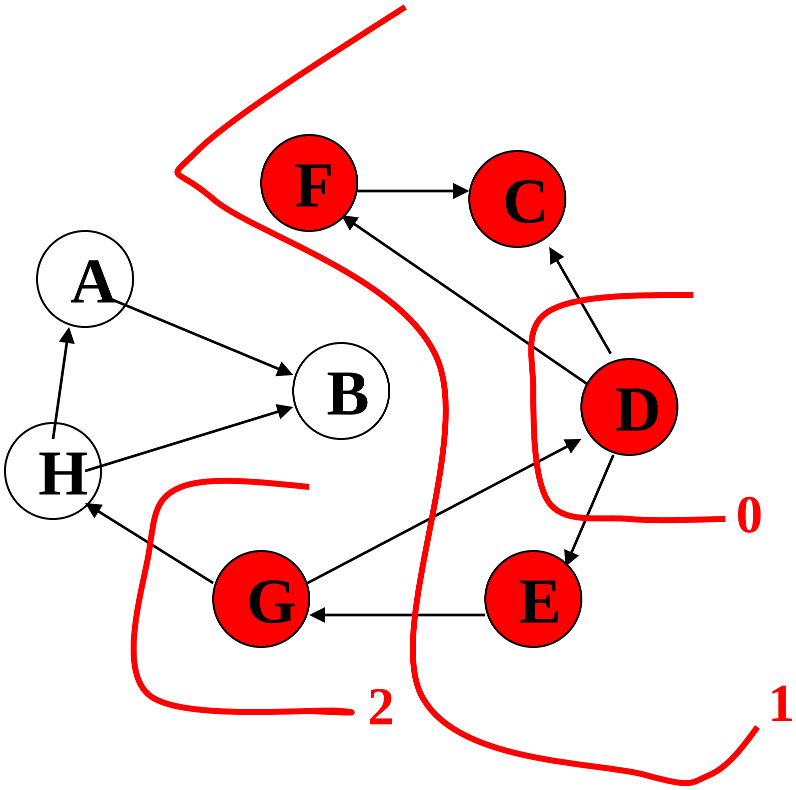
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D, C, E, F

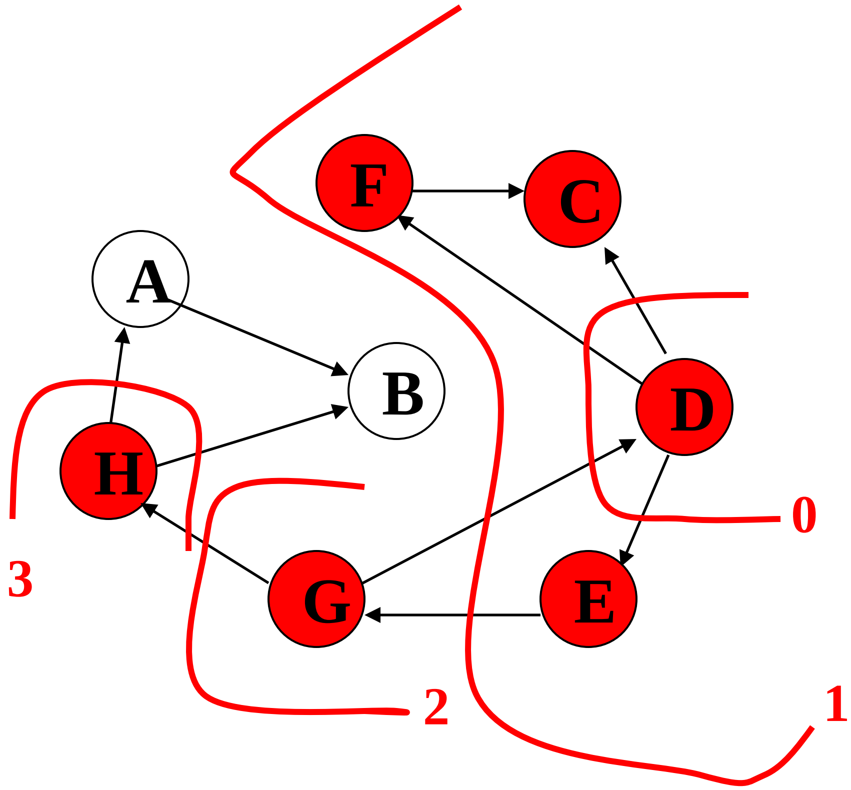
BFS Overview



When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G

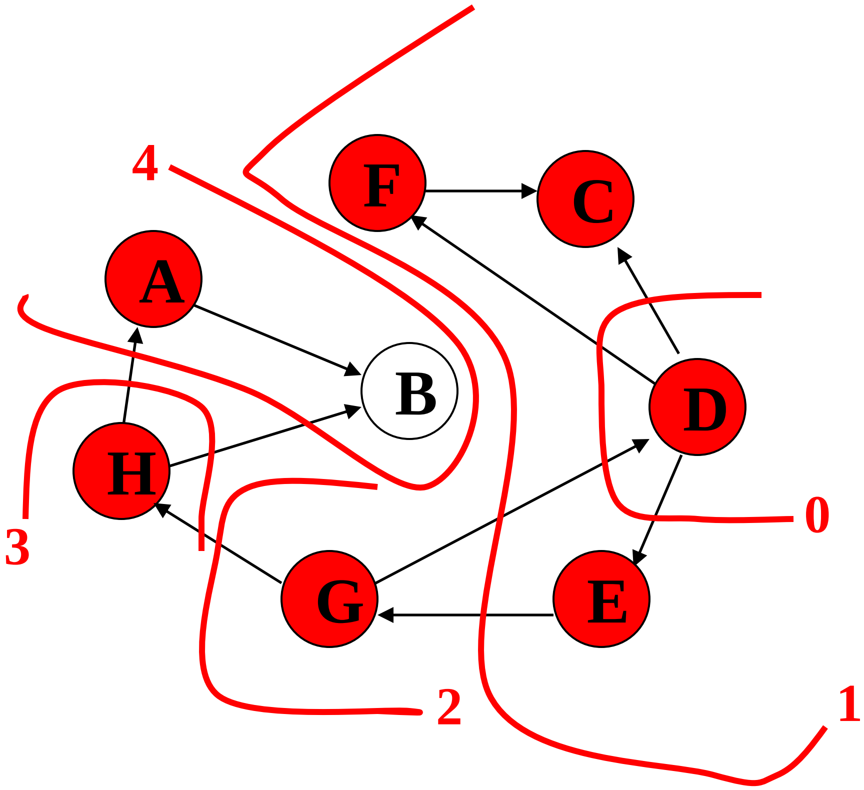
BFS Overview



When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G, H

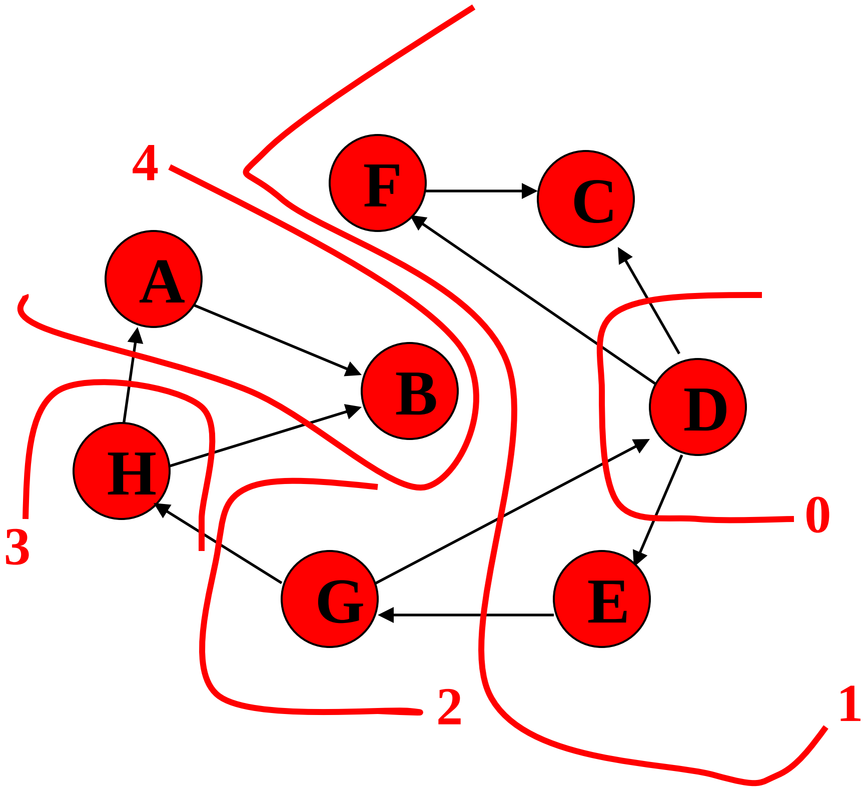
BFS Overview



When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G, H, A

BFS Overview



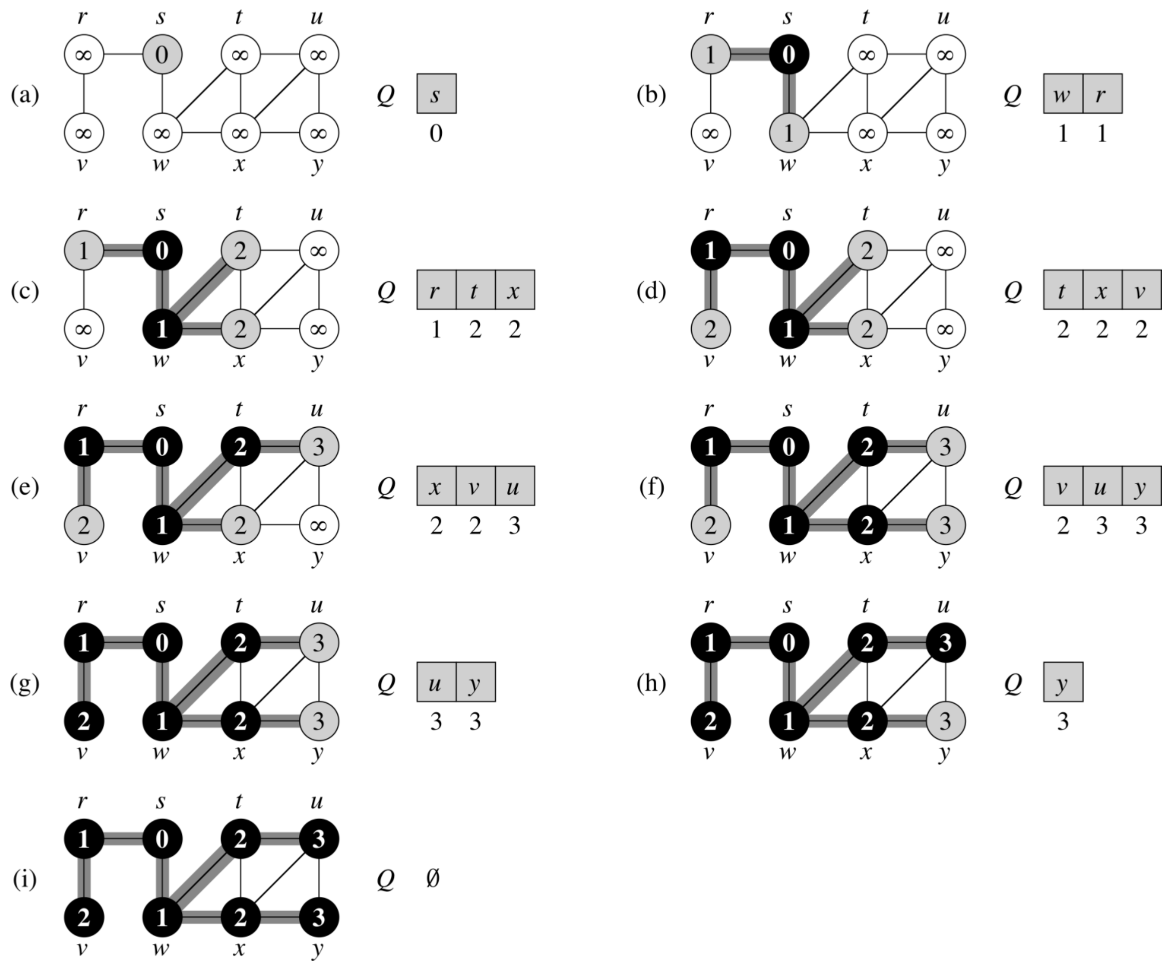
When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G, H, A, B

BFS

```
BFS(G, s)                              // G is the graph and s is the starting node
 1  for each vertex u ∈ V [G] - {s}
 2      do color[u] ← WHITE            // color of vertex u
 3         d[u] ← ∞                    // distance from source s to vertex u
 4         π[u] ← NIL                  // predecessor of u
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← Ø                              // Q is a FIFO - queue
 9  ENQUEUE(Q, s)
10  while Q ≠ Ø                        // iterates as long as there are gray vertices. Lines 10-18
11      do u ← DEQUEUE(Q)
12         for each v ∈ Adj[u]
13             do if color[v] = WHITE  // discover the undiscovered adjacent vertices
14                   then color[v] ← GRAY           // enqueued whenever painted gray
15                        d[v] ← d[u] + 1
16                        π[v] ← u
17                        ENQUEUE(Q, v)
18         color[u] ← BLACK            // painted black whenever dequeued
```

BFS Example



(a) Q s 0
(b) Q w r 1 1
(c) Q r t x 1 2 2
(d) Q t x v 2 2 2
(e) Q x v u 2 2 3
(f) Q v u y 2 3 3
(g) Q u y 3 3
(h) Q y 3
(i) Q Ø
r s t u
v w x y

BFS Analysis

- Enqueue and Dequeue happen only once for each node. : O(V).
- Sum of the lengths of adjacency lists : Θ(E) (for a directed graph)
- Initialization overhead O(V)

Total runtime O(V+E)