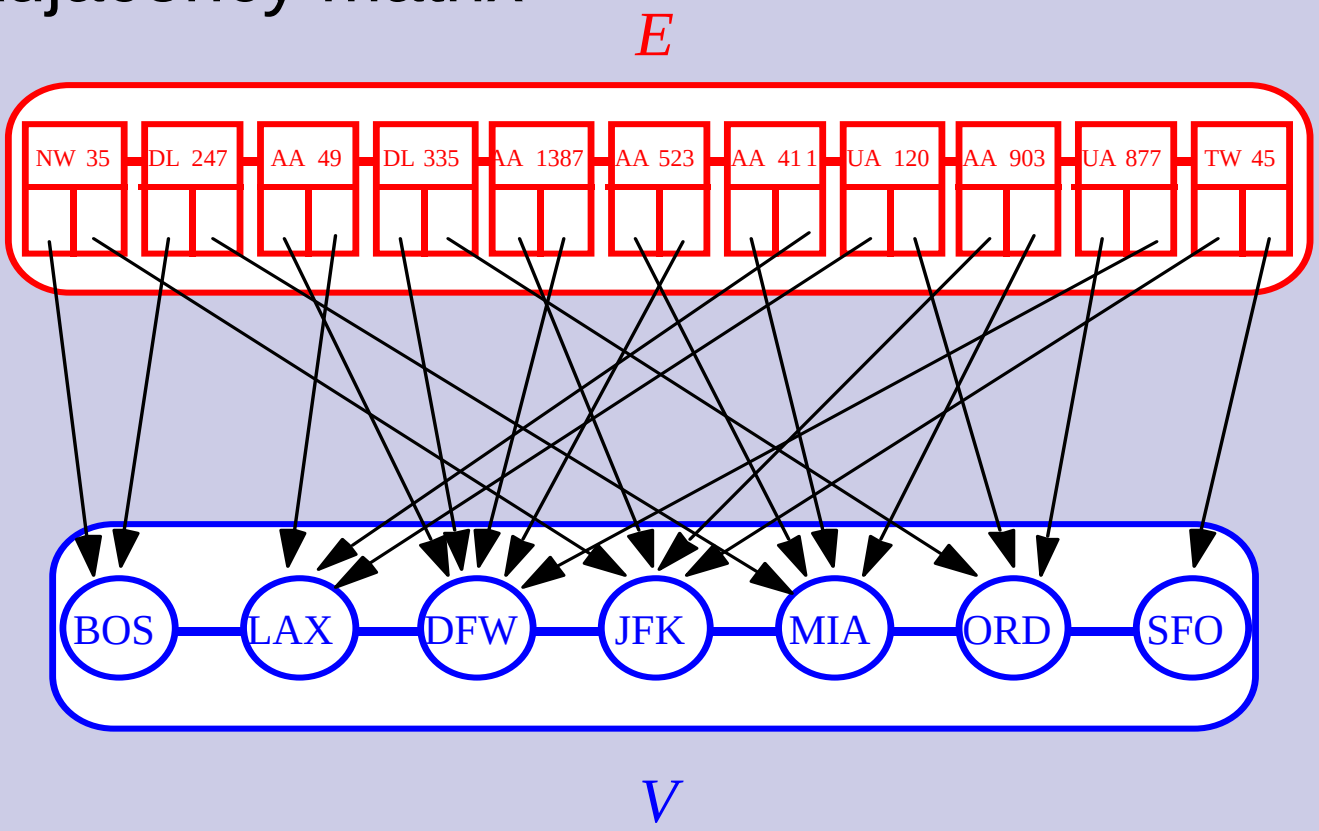
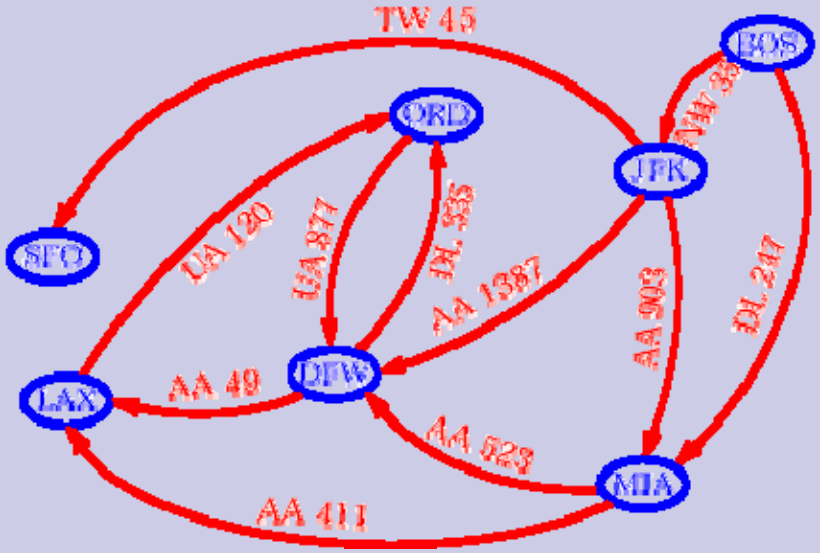
Data Structures for Graphs

- Edge list
- Adjacency lists
- Adjacency matrix



Data Structures for Graphs

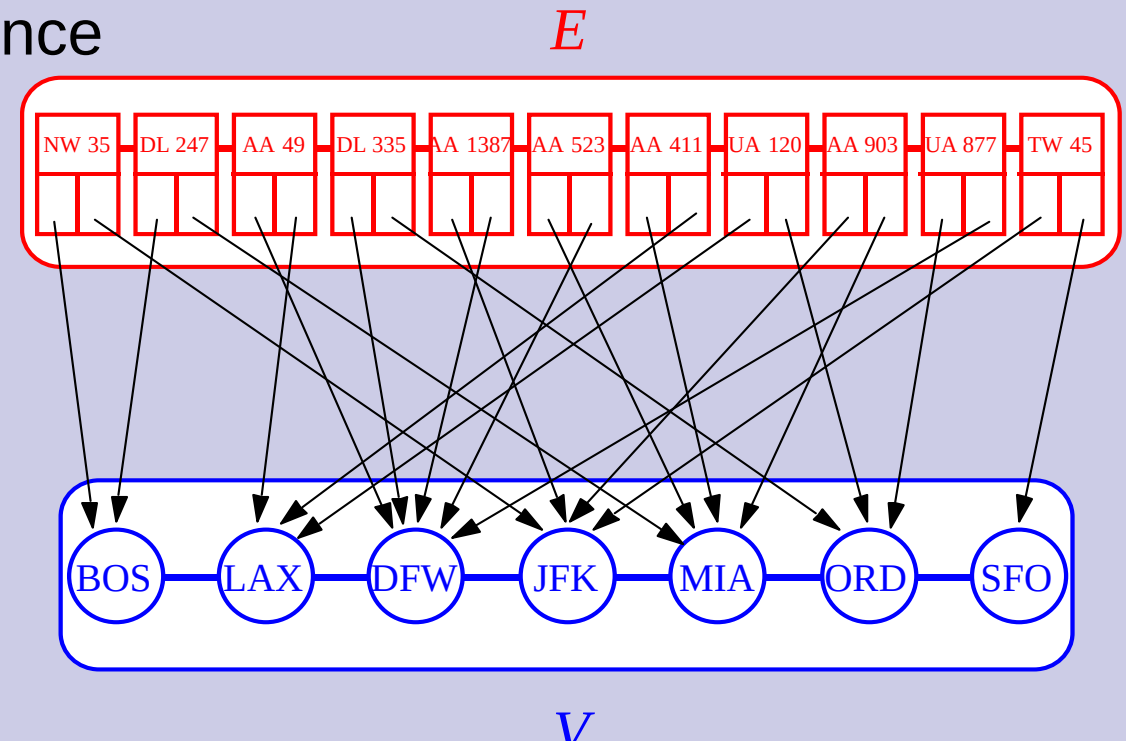
- A Graph! How can we represent it?
- To start with, we store the vertices and the edges into two containers, and each edge object has references to the vertices it connects.



Additional structures can be used to perform efficiently the methods of the Graph ADT

Edge List

- The edge list structure simply stores the vertices and the edges in two unsorted sequences.
- Easy to implement.
- Finding the edges incident on a given vertex is inefficient since it requires examining the entire edge sequence

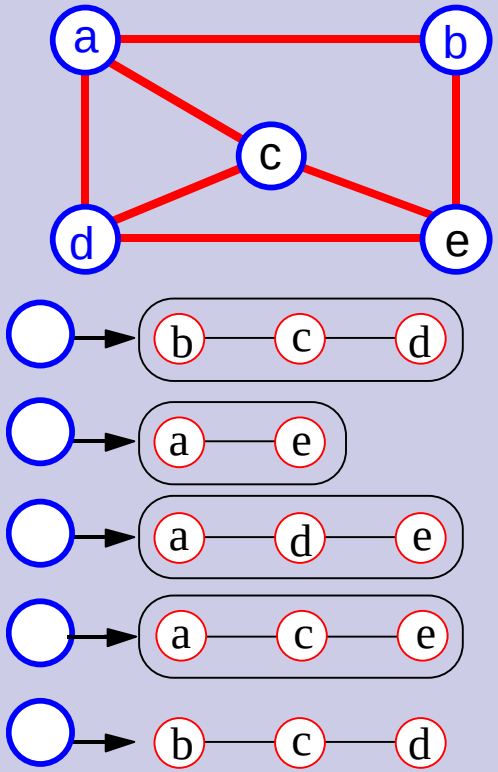


Performance of Edge List

Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected	O(1)
incidentEdges, inIncidentEdges, outIncidentEdges, adjacent Vertices , inAdjacentVertices, outAdjacentVertices,areAdjacent, degree, inDegree, outDegree	O(m)
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	O(1)
removeVertex	O(m)

Adjacency List (traditional)

- adjacency list of a vertex v: sequence of vertices adjacent to v
- represent the graph by the adjacency lists of all the vertices

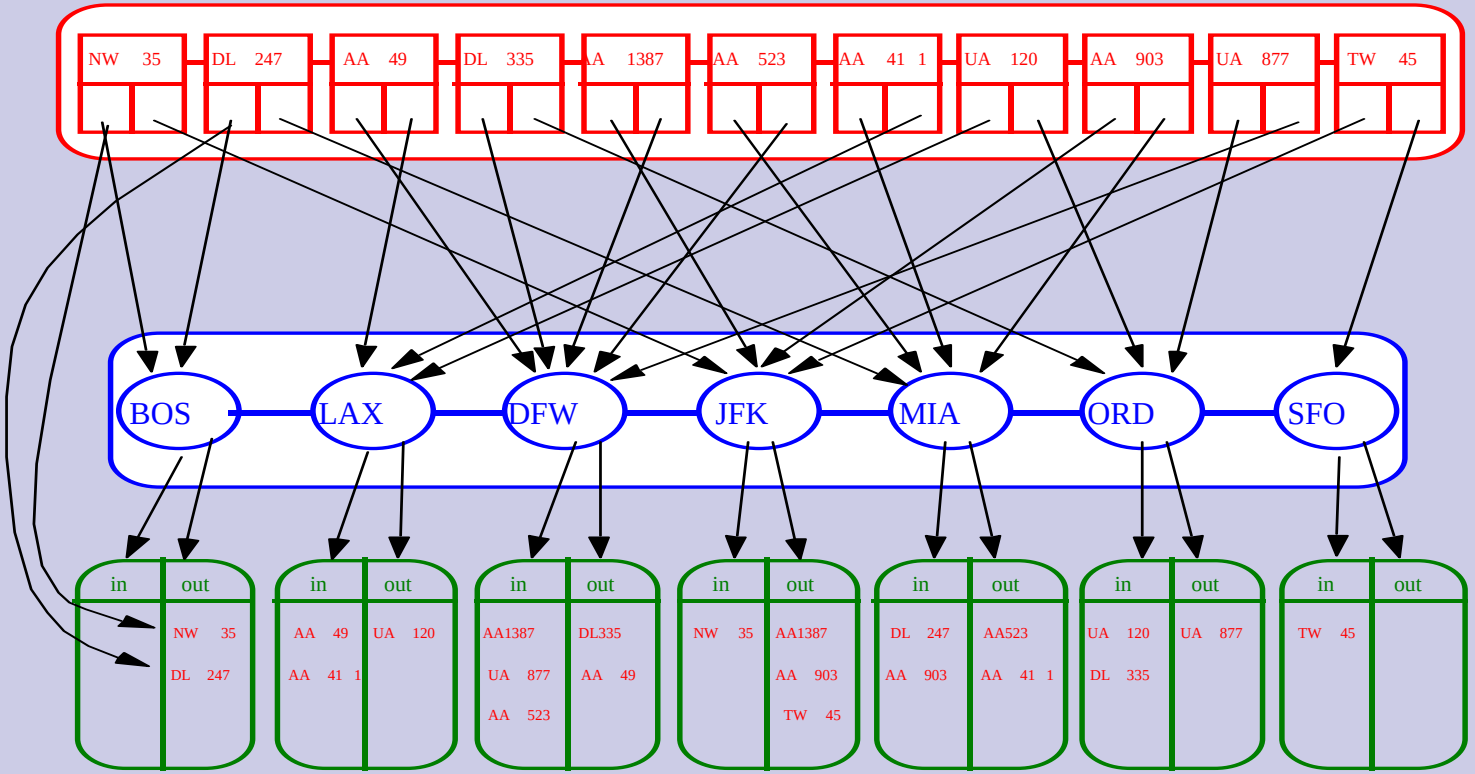


Space =

$$\Theta(N + \Sigma \deg(v)) = \Theta(N + M)$$

Adjacency List (modern)

- The adjacency list structure extends the edge list structure by adding adjacency lists to each vertex.

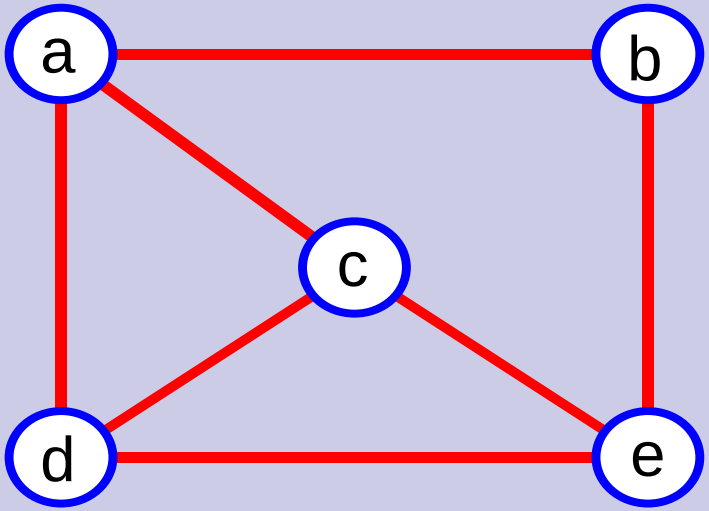


The space requirement is O(n + m).

Performance of the Adjacency List Structure

size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	O(1)
incidentEdges(v), inIncidentEdges(v), outIncidentEdges(v), adjacentVertices(v), inAdjacentVertices(v), outAdjacentVertices(v)	O(deg(v))
areAdjacent(u, v)	O(min(deg(u),deg(v)))
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection,	O(1)
removeVertex(v)	O(deg(v))

Adjacency Matrix (traditional)



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

- matrix M with entries for all pairs of vertices
- M[i,j] = true means that there is an edge (i,j) in the graph.
- M[i,j] = false means that there is no edge (i,j) in the graph.
- There is an entry for every possible edge, therefore:

$$\text{Space} = \Theta(N^2)$$

Adjacency Matrix (modern)

- The adjacency matrix structures augments the edge list structure with a matrix where each row and column corresponds to a vertex.

	0	1	2	3	4	5	6
0	Ø	Ø	NW 35	Ø	DL 247	Ø	Ø
1	Ø	Ø	Ø	AA 49	Ø	DL 335	Ø
2	Ø	AA 1387	Ø	Ø	AA 903	Ø	TW 45
3	Ø	Ø	Ø	Ø	Ø	UA 120	Ø
4	Ø	AA 523	Ø	AA 411	Ø	Ø	Ø
5	Ø	UA 877	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø

BOS 0 DFW 1 JFK 2 LAX 3 MIA 4 ORD 5 SFO 6

Performance of Adjacency Matrix

Operation	Time
size, isEmpty, replaceElement, swap	O(1)
numVertices, numEdges	O(1)
vertices	O(n)
edges, directedEdges, undirectedEdges	O(m)
elements, positions	O(n+m)
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	O(1)
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices,	O(n)
areAdjacent	O(1)
insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	O(1)
insertVertex, removeVertex	$O(n^2)$

Graph Searching Algorithms

- Systematic search of every edge and vertex of the graph
- Graph G = (V,E) is either directed or undirected
- Today's algorithms assume an adjacency list representation
- Applications
  - Compilers
  - Graphics
  - Maze-solving
  - Mapping
  - Networks: routing, searching, clustering, etc.

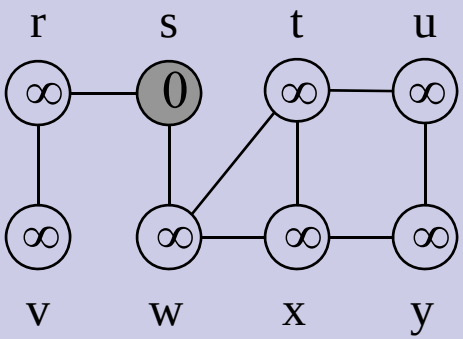
Breadth First Search

- A **Breadth-First Search (BFS)** traverses a **connected component** of a graph, and in doing so defines a **spanning tree** with several useful properties
- BFS in an **undirected** graph G is like wandering in a labyrinth with a string.
- The starting vertex *s,* it is assigned a distance 0.
- In the first round, the string is unrolled the length of one edge, and all of the vertices that are only one edge away from the anchor are visited (**discovered**), and assigned distances of 1

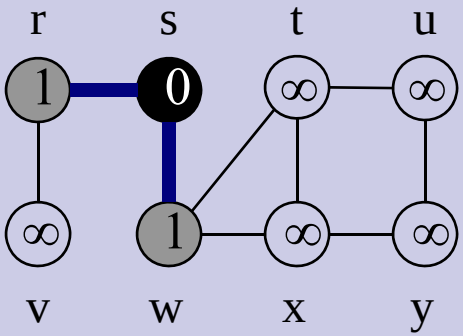
Breadth-First Search (2)

- In the second round, all the new vertices that can be reached by unrolling the string 2 edges are visited and assigned a distance of 2
- This continues until every vertex has been assigned a level
- The label of any vertex *v* corresponds to the length of the shortest path (in terms of edges) from *s* to *v*

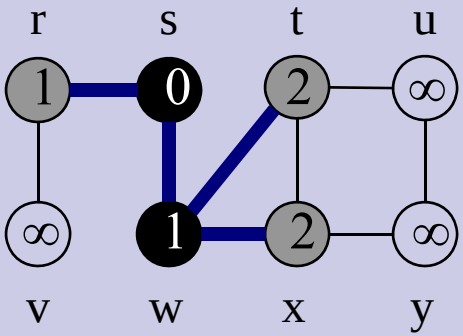
BFS Example



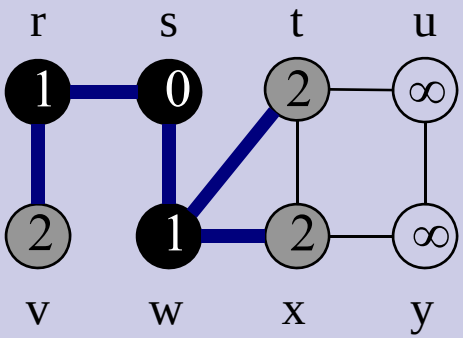
Q | s
0



Q | w | r
1 1

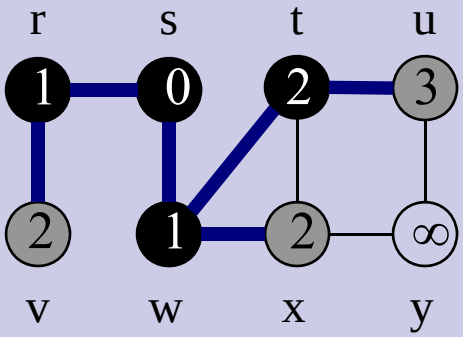


Q | r | t | x
1 2 2

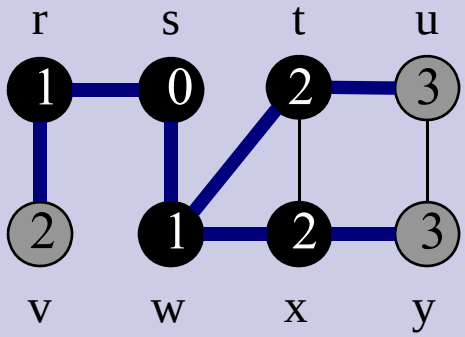


Q | t | x | v
2 2 2

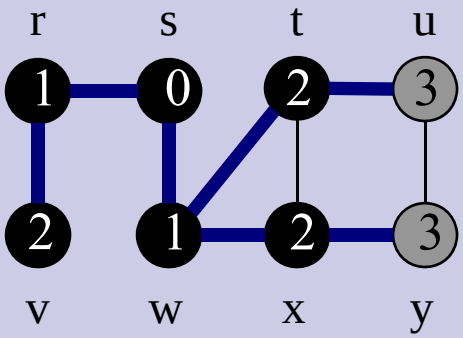
BFS Example



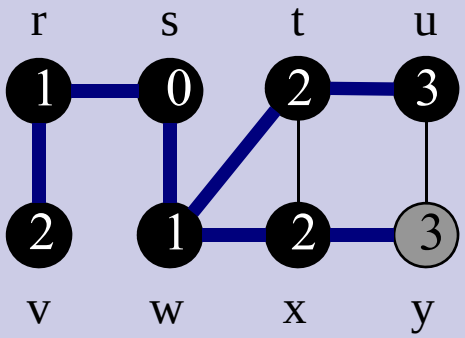
Q | x | v | u
2 2 3



Q | v | u | y
2 3 3

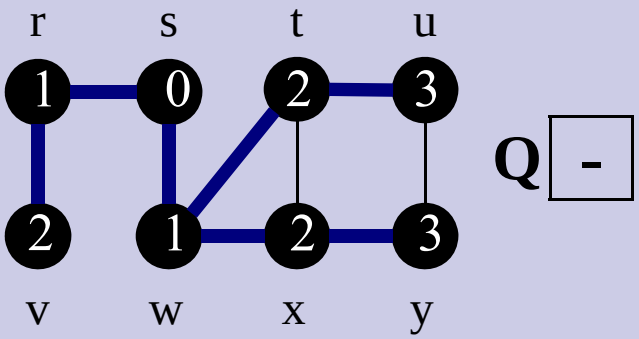


Q | u | y
3 3

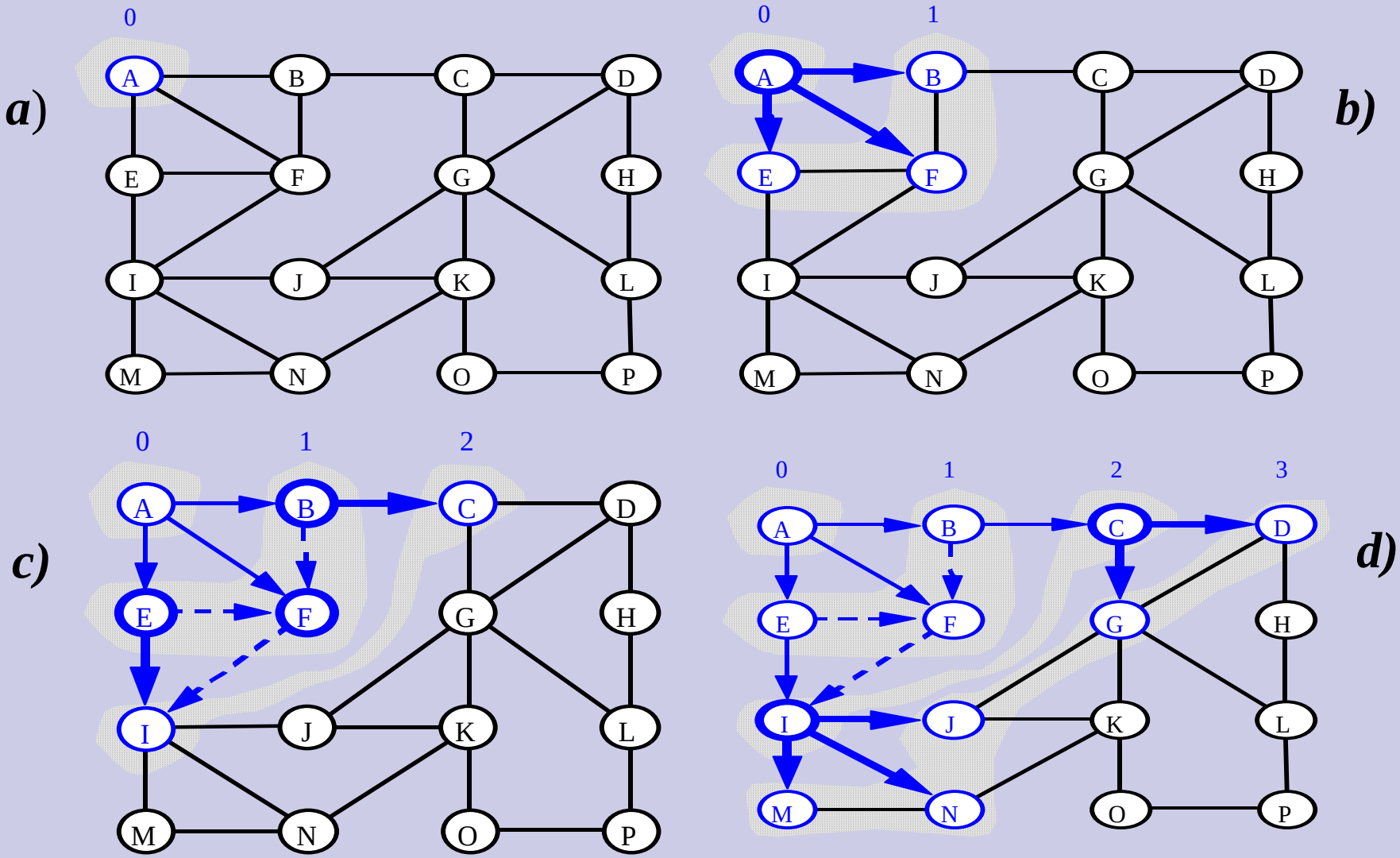


Q | y
3

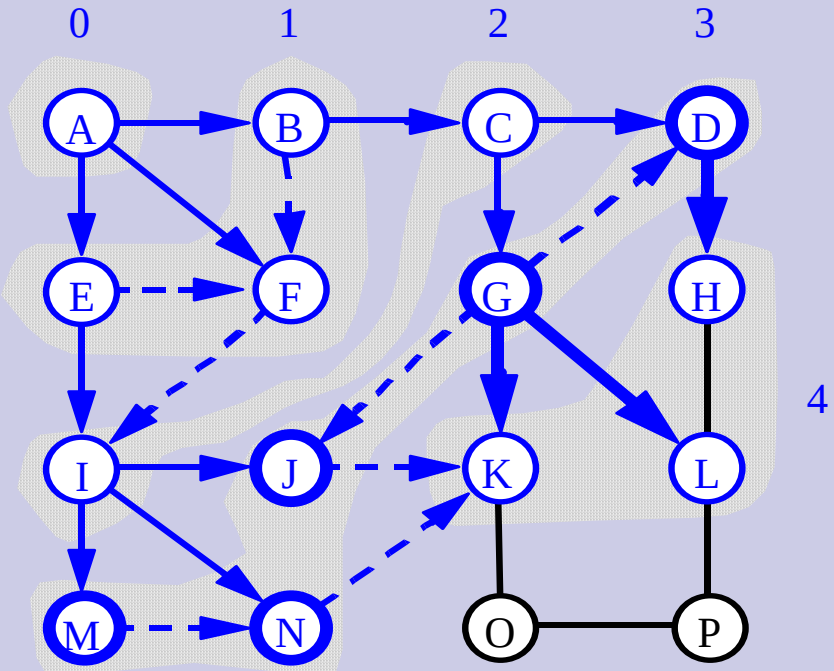
BFS Example: Result

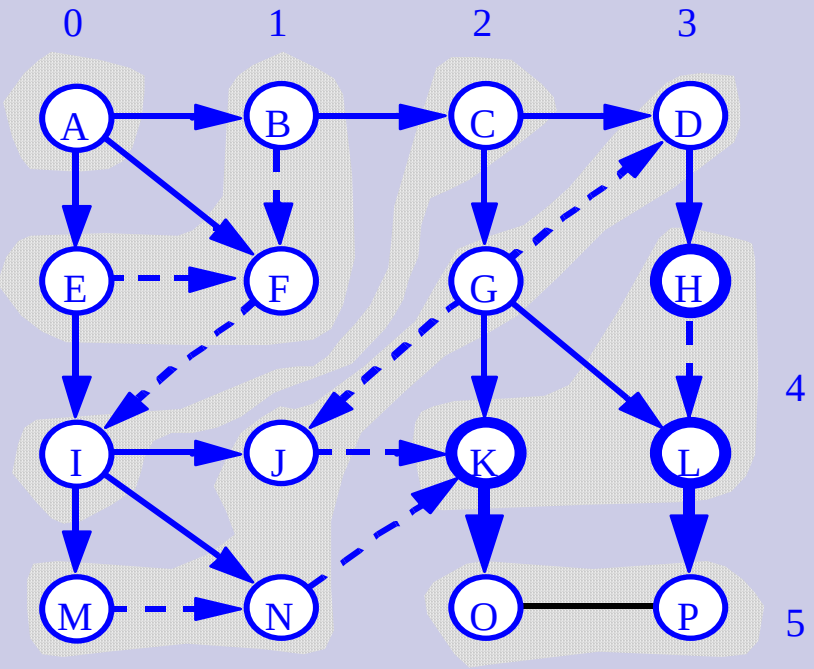


BFS - A Graphical representation



More BFS





BFS Algorithm

```
BFS(G,s)
01 for each vertex u ∈ V[G]-{s}
02    color[u] ← white
03    d[u] ← ∞
04    π[u] ← NIL
05 color[s] ← gray
06 d[s] ← 0
07 π[u] ← NIL
08 Q ← {s}
09 while Q ≠∅ do
10    u ← head[Q]
11    for each v ∈ Adj[u] do
12       if color[v] = white then
13          color[v] ← gray
14          d[v] ← d[u] + 1
15          π[v] ← u
16          Enqueue(Q,v)
17    Dequeue(Q)
18    color[u] ← black
```

Init all vertices

Init BFS with *s*

Handle all *u*'s children before handling any children of children

BFS Running Time

- Given a graph G = (V,E)
  - Vertices are enqueued if there color is white
  - Assuming that en- and dequeuing takes O(1) time the total cost of this operation is O(V)
  - Adjacency list of a vertex is scanned when the vertex is dequeued (and only then…)
  - The sum of the lengths of all lists is Θ(E). Consequently, O(E) time is spent on scanning them
  - Initializing the algorithm takes O(V)
- **Total running time O(V+E)** (linear in the size of the adjacency list representation of G)

BFS Properties

- Given an undirected graph G = (V,E), BFS **discovers all vertices reachable from a source vertex *s***
- For each vertex v at level i, the path of the BFS tree T between s and v has i edges, and any other path of G between ***s*** and v has at least i edges.
- If (u, v) is an edge then the level numbers of u and v differ by at most one.
- It computes the **shortest distance** to all reachable vertices

Breadth First Tree

- Predecessor subgraph of G

$$G_\pi = (V_\pi, E_\pi)$$

$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$$

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$$

- G_π is a breadth-first tree
  - V_π consists of the vertices reachable from s, and
  - for all $v \in V_\pi$, there is a unique simple path from *s* to *v* in G_π that is also a shortest path from *s* to *v* in G
- The edges in G_π are called tree edges
- For any vertex *v* reachable from *s*, the path in the breadth first tree from s to v, corresponds to a **shortest path** in G