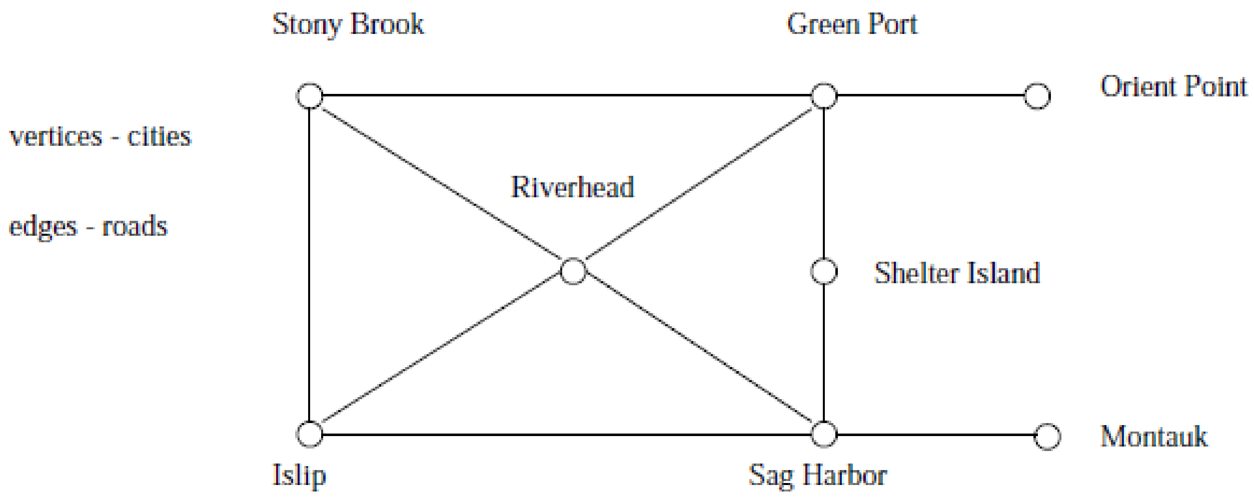
Elementary Graph Algorithms

Graphs

- Graphs are one of the unifying themes of computer science. A graph $G = (V, E)$ is defined by a set of vertices V , and a set of edges E consisting of ordered or unordered pairs of vertices from V .
- Thus a graph $G = (V, E)$
  - V = set of vertices
  - E = set of edges = subset of $V \times V$
  - Thus $|E| = O(|V|^2)$

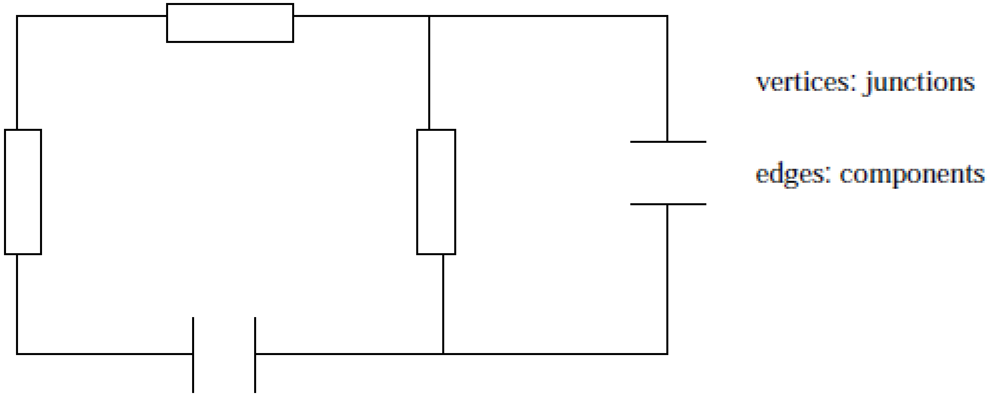
Examples

- **Road Networks**
  - In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges.



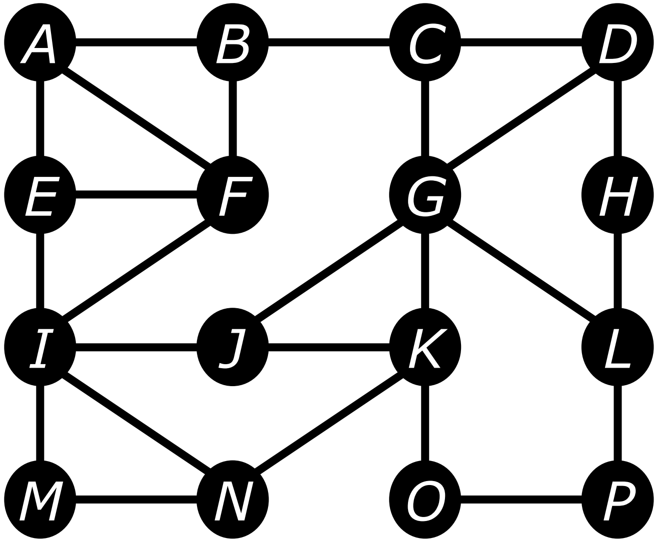
Examples

- **Electronic Circuits**
  - In an electronic circuit, with junctions as vertices and components as edges.



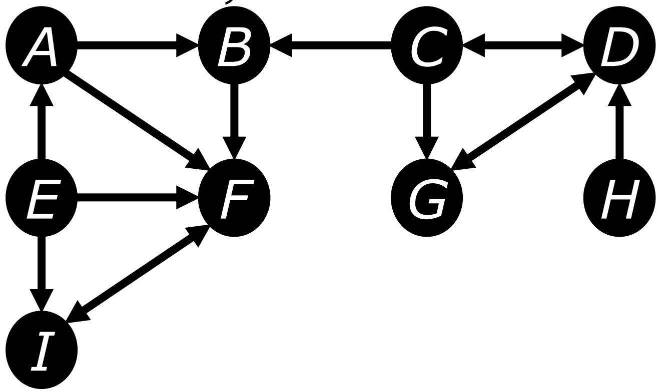
Graph Variations

- Variations:
  - A *connected graph* has a path from every vertex to every other
  - In an *undirected graph:*
    - Edge (u,v) = Edge (v,u)
    - No self-loops



Graph Variations

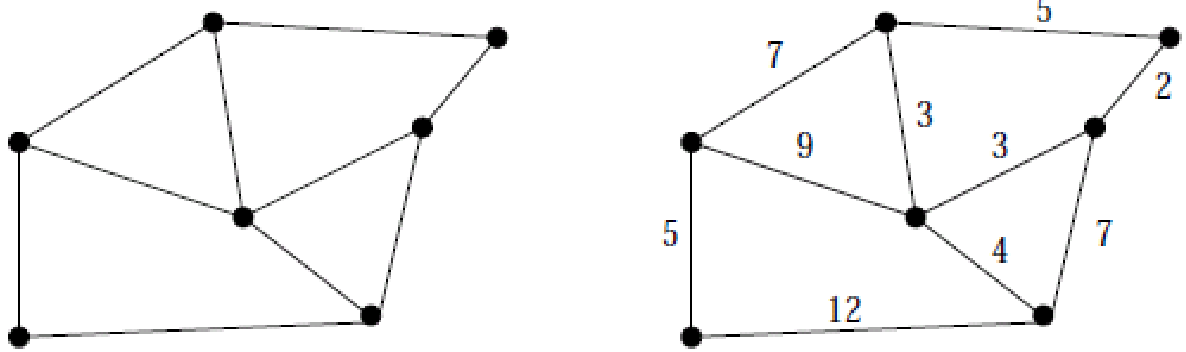
- In a *directed* graph:
  - Edge (u,v) goes from vertex u to vertex v, notated $u \to v$
  - (u,v) is different from (v,u)



- Road networks *between cities are typically undirected.*
- Street networks *within* cities are almost always directed because of one-way streets.
- Most graphs of graph-theoretic interest are undirected.

Graph Variations

- More variations:
  - A *weighted graph* associates weights with either the edges or the vertices
    - E.g., a road map: edges might be weighted with their length, drive-time or speed limit.
  - In *unweighted* graphs, there is no cost distinction between various edges and vertices.

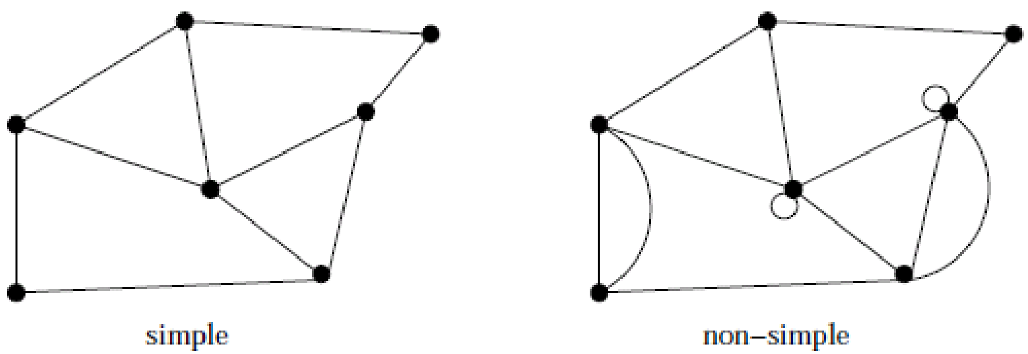


unweighted weighted

- A *multigraph* allows multiple edges between the same vertices
  - E.g., the call graph in a program (a function can get called from multiple points in another function)

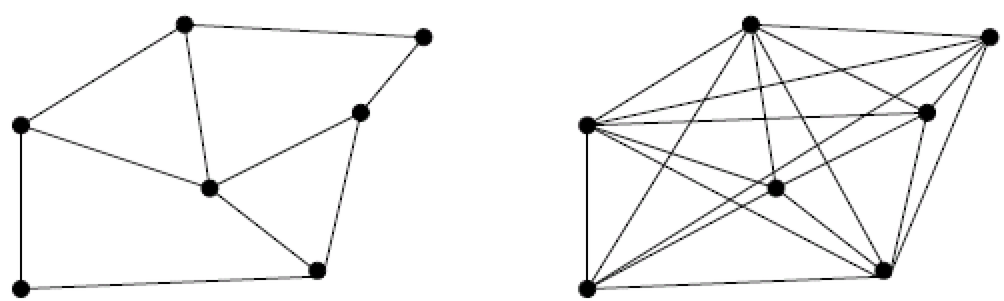
Simple vs. Non-simple Graphs

- Certain types of edges complicate the task of working with graphs. A *self-loop* is an edge (x; x) involving only one vertex.
- An edge (x; y) is a *multi-edge* if it occurs more than once in the graph.
- Any graph which avoids these structures is called *simple.*



Spars Vs Dense Graphs

- Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
- Graphs are usually sparse due to application-specific constraints.
- Road networks must be sparse because of road junctions.



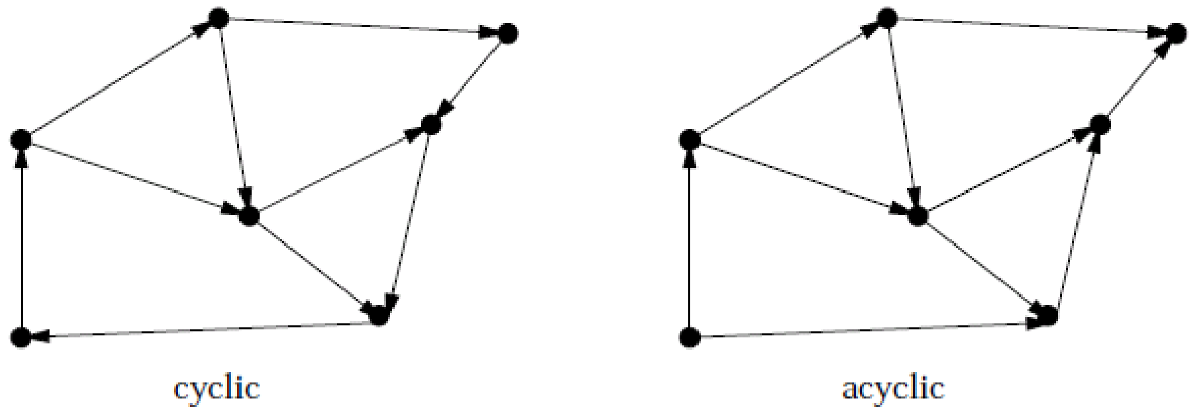
sparse dense

Spars Vs Dense Graphs

- Typically dense graphs have a quadratic number of edges while sparse graphs are linear in size.

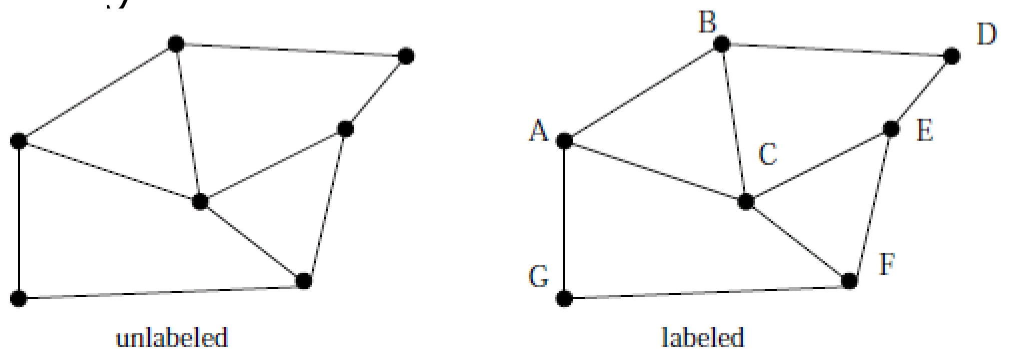
Cyclic vs. Acyclic Graphs

- An *acyclic* graph does not contain any cycles. *Trees* are connected acyclic *undirected* graphs.
- Directed acyclic graphs are called *DAGs*. They arise naturally in scheduling problems, where a directed edge (x; y) indicates that x must occur before y



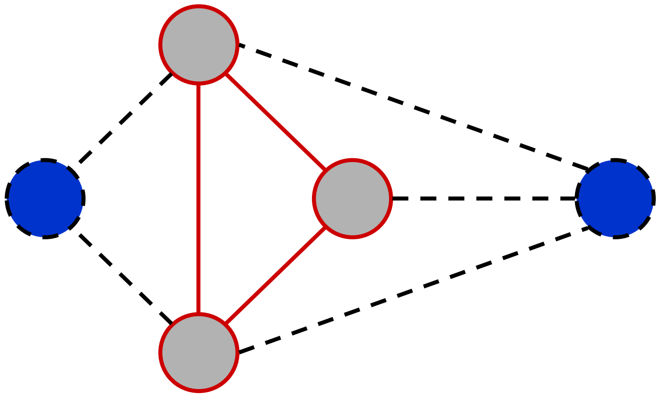
Labeled vs. Unlabeled Graphs

- In *labeled graphs*, each vertex is assigned a unique name or identifier to distinguish it from all other vertices.
- An important graph problem is *isomorphism testing*, determining whether the topological structure of two graphs are in fact identical if we ignore any labels.

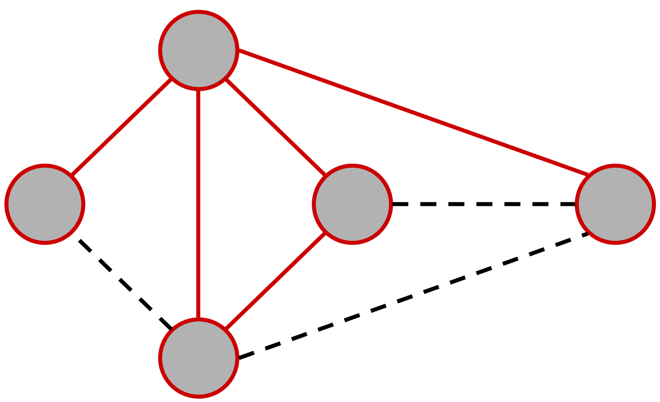


Subgraphs

- A subgraph S of a graph G is a graph such that
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



Subgraph

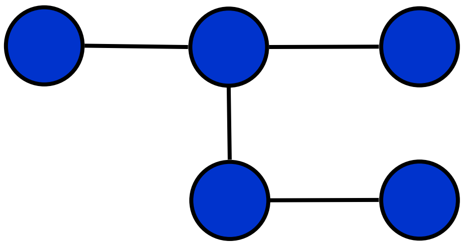


Spanning subgraph

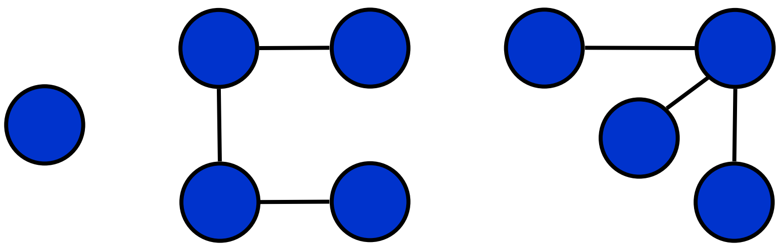
Trees and Forests

- A tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



Tree



Forest

Graph in Applications

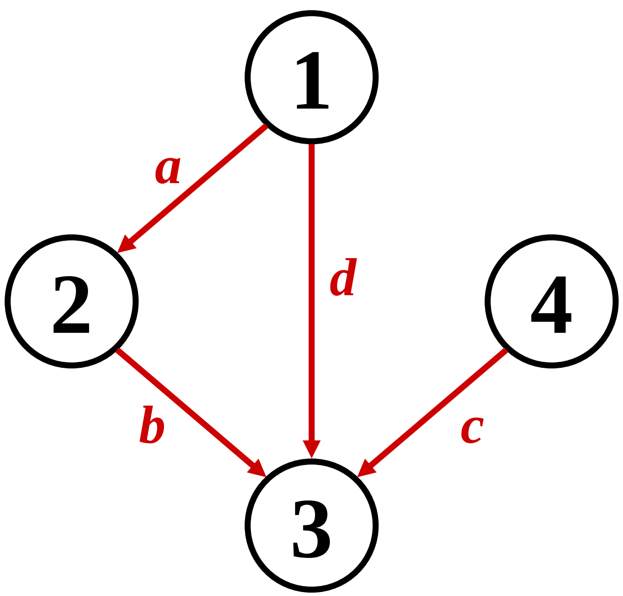
- Internet: web graphs
  - Each page is a vertex
  - Each edge represent a hyperlink
  - Directed
- GPS: highway maps
  - Each city is a vertex
  - Each freeway segment is an undirected edge
  - Undirected
- Graphs are ubiquitous in computer science

Representing Graphs

- Assume V = $\{1, 2, \ldots, n\}$
- An *adjacency matrix* represents the graph as a n x n matrix A:
  - $A[i, j] = 1$ if edge $(i, j) \in E$ (or weight of edge)
    $= 0$ if edge $(i, j) \notin E$

Graphs: Adjacency Matrix

- Example:

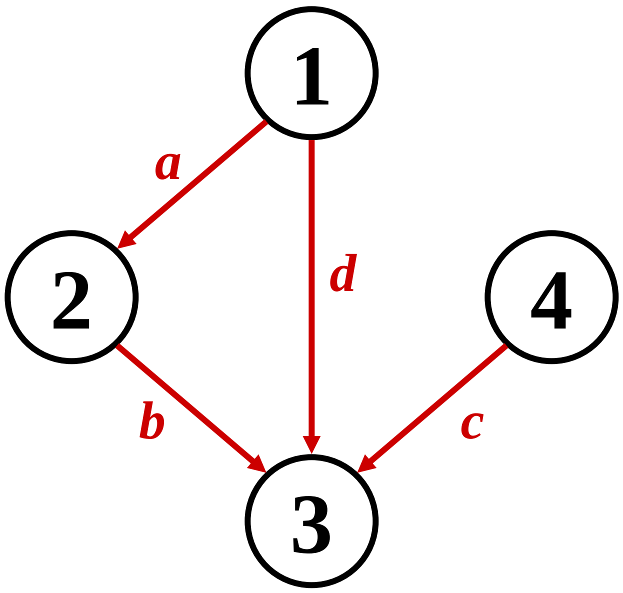


A	1	2	3	4
1				
2				
3				
4				

? ?

Graphs: Adjacency Matrix

- Example:



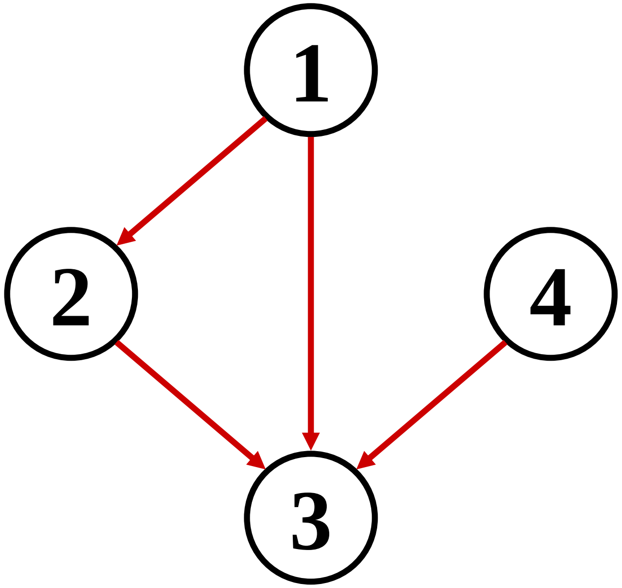
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

Graphs: Adjacency Matrix

- The adjacency matrix is a dense representation
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs
- Most large interesting graphs are sparse
  - E.g., planar graphs, in which no edges cross
  - For this reason the *adjacency list* is often a more appropriate respresentation

Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to v
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}



- Variation: can also keep a list of edges coming *into* vertex

Graphs: Adjacency List

- How much storage is required?
  - The *degree* of a vertex *v* in an undirected graph = # incident edges
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is
    $$\Sigma \text{ out-degree}(v) = |E|$$
    takes $\Theta(V + E)$ storage
  - For undirected graphs, # items in adj lists is
    $$\Sigma \text{ degree}(v) = 2\,|E|$$
    also $\Theta(V + E)$ storage
- So: Adjacency lists take $O(V+E)$ storage

Graph Representation

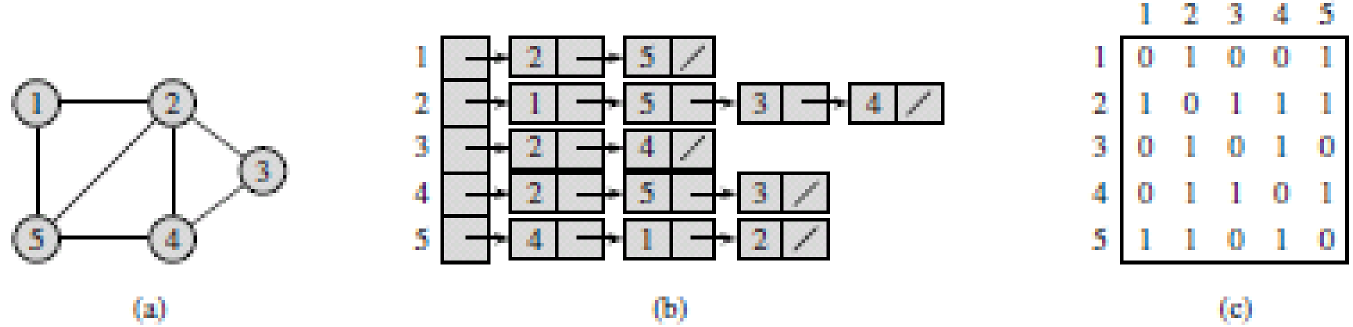


Figure 22.1 Two representations of an undirected graph. (a) An undirected graph G having five vertices and seven edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

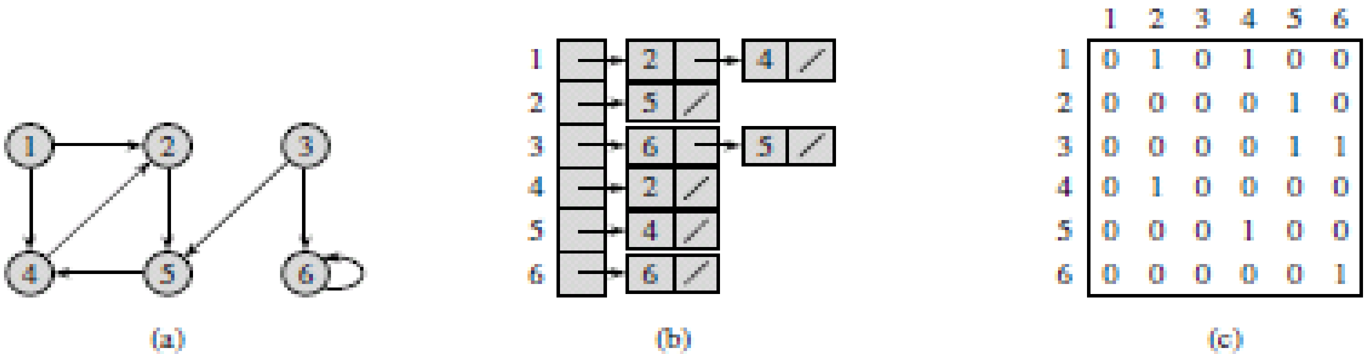


Figure 22.2 Two representations of a directed graph. (a) A directed graph G having six vertices and eight edges. (b) An adjacency-list representation of G. (c) The adjacency-matrix representation of G.

Google Problem:Graph Searching

- How to search and explore in the Web Space?
- How to find all web-pages and all the hyperlinks?
  - Start from one vertex "www.google.com"
  - Systematically follow hyperlinks from the discovered vertex/web page
  - Build a search tree along the exploration

General Graph Searching

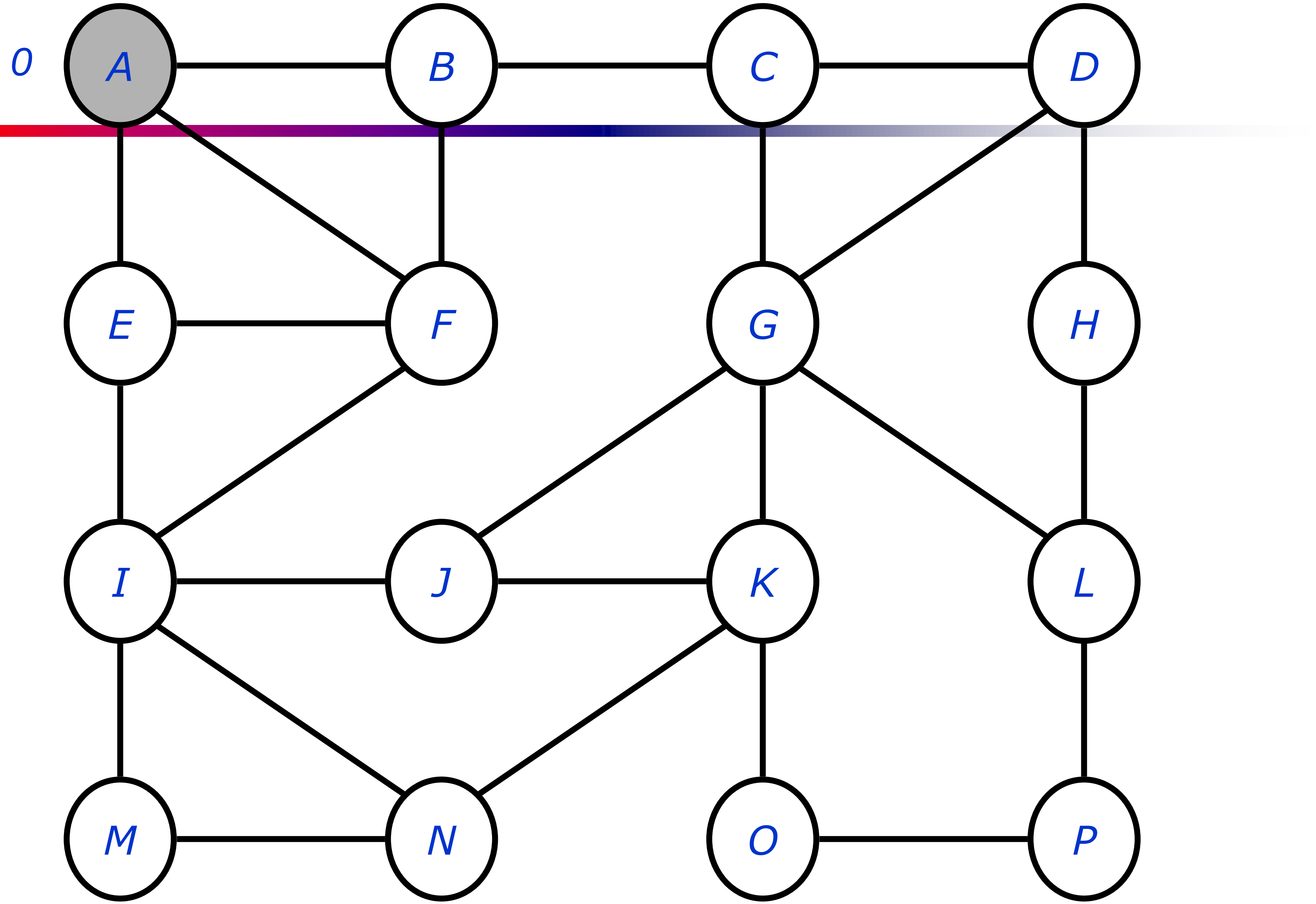
- Given: a graph $G = (V, E)$, directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected

Breadth-First Search

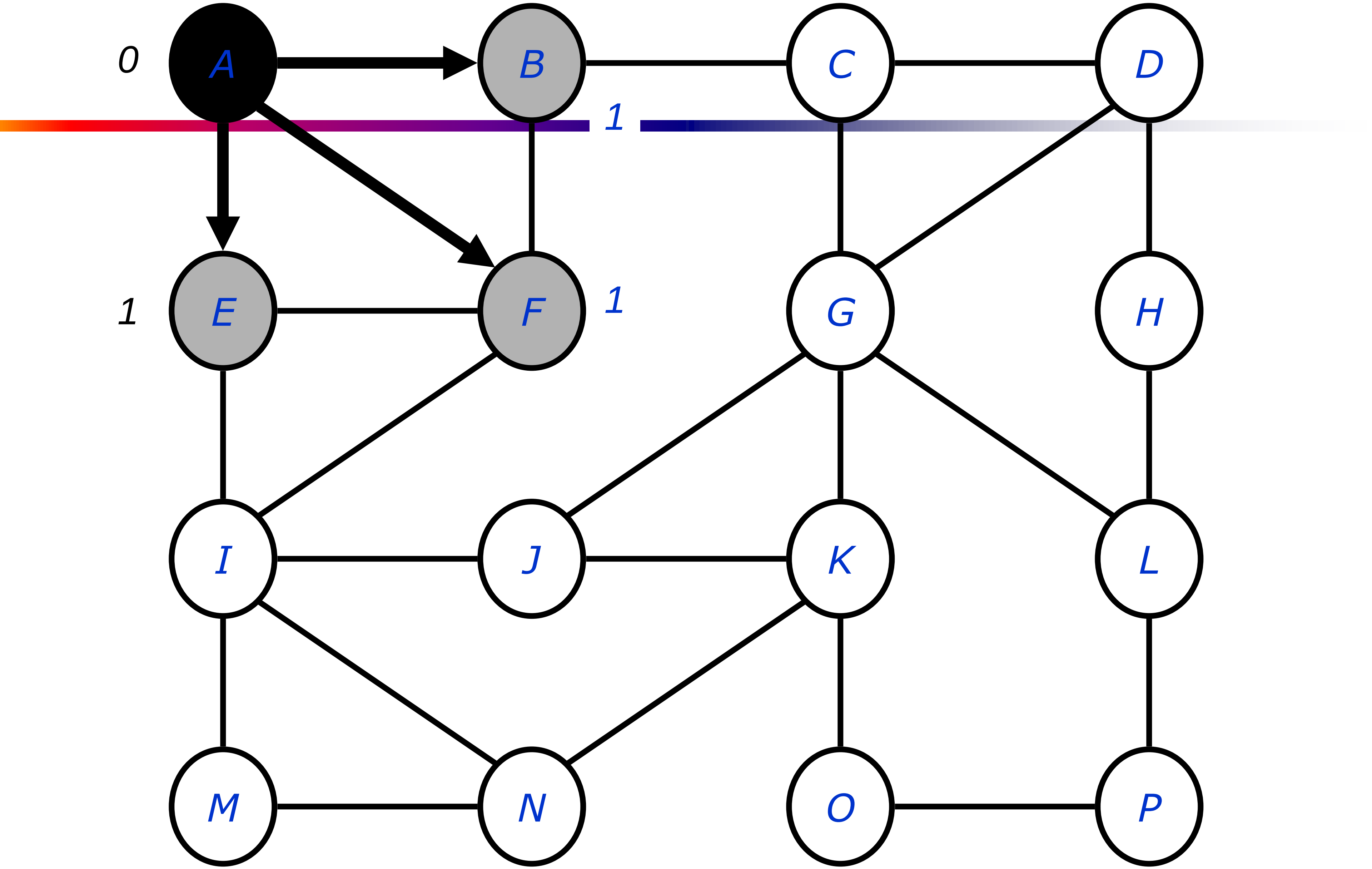
- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

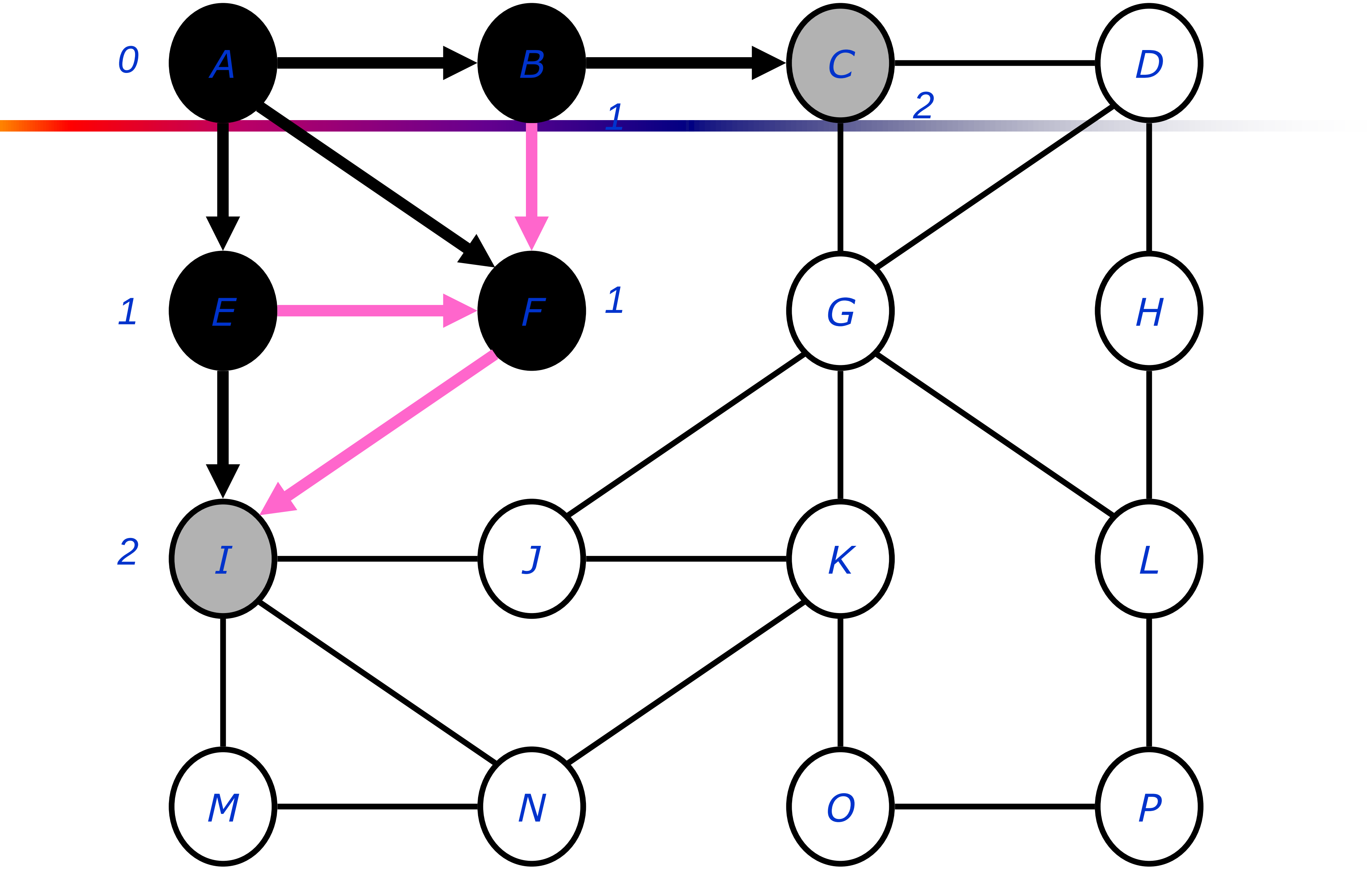
Breadth-First Search

- Again will associate vertex "colors" to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

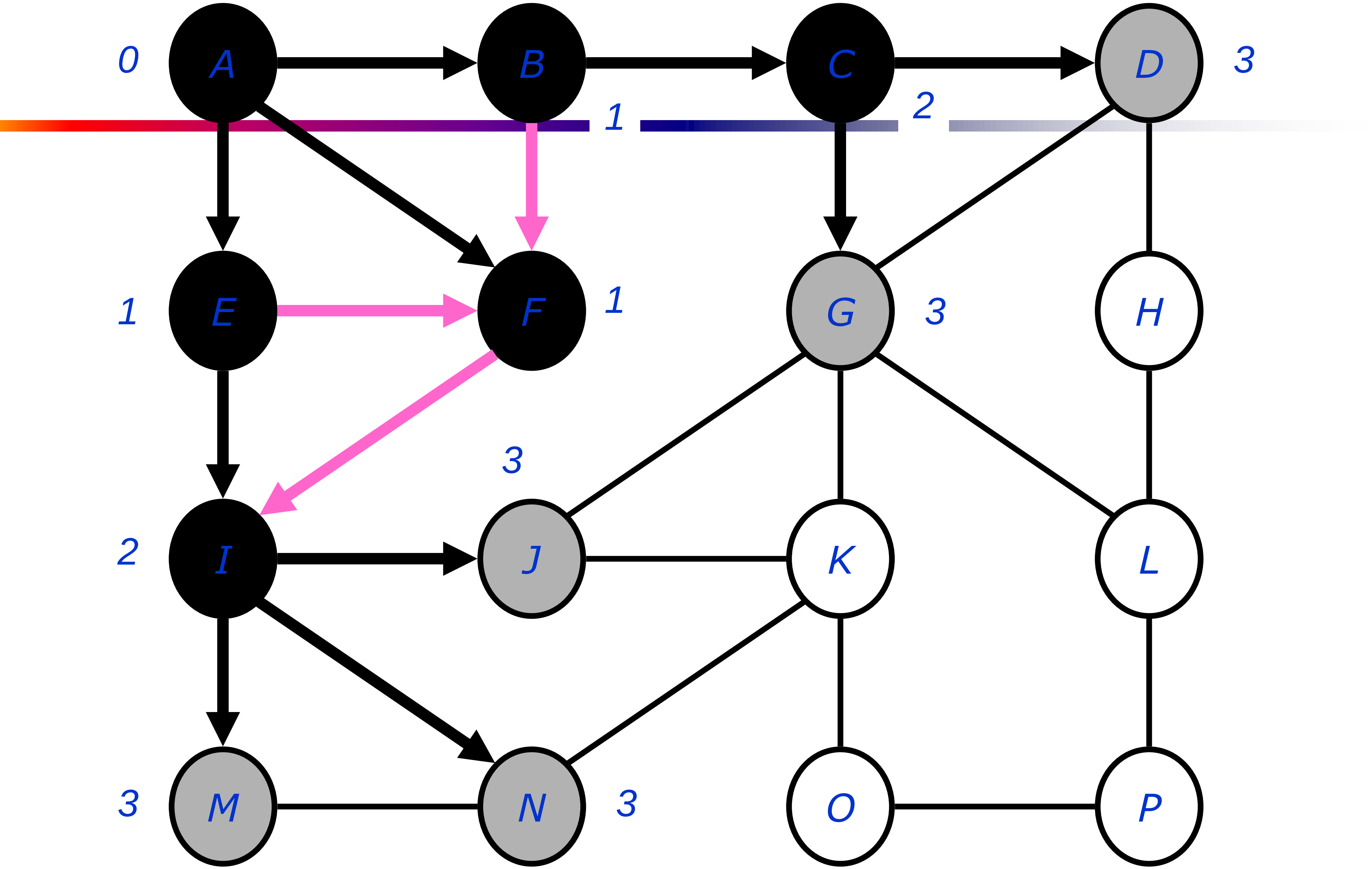


0
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

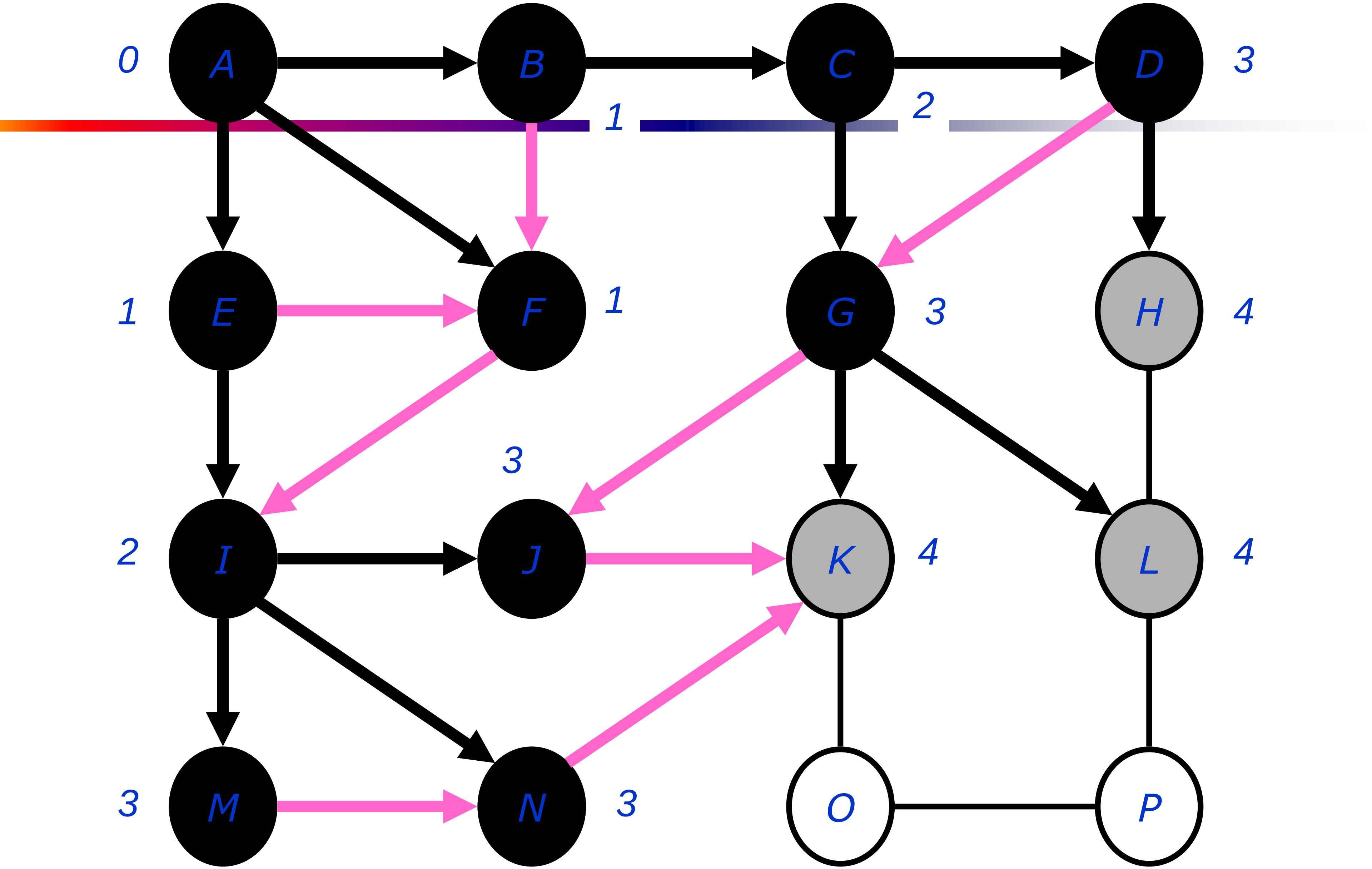

0
A
B
C
D
1
E
F
1
G
H
1
I
J
K
L
M
N
O
P



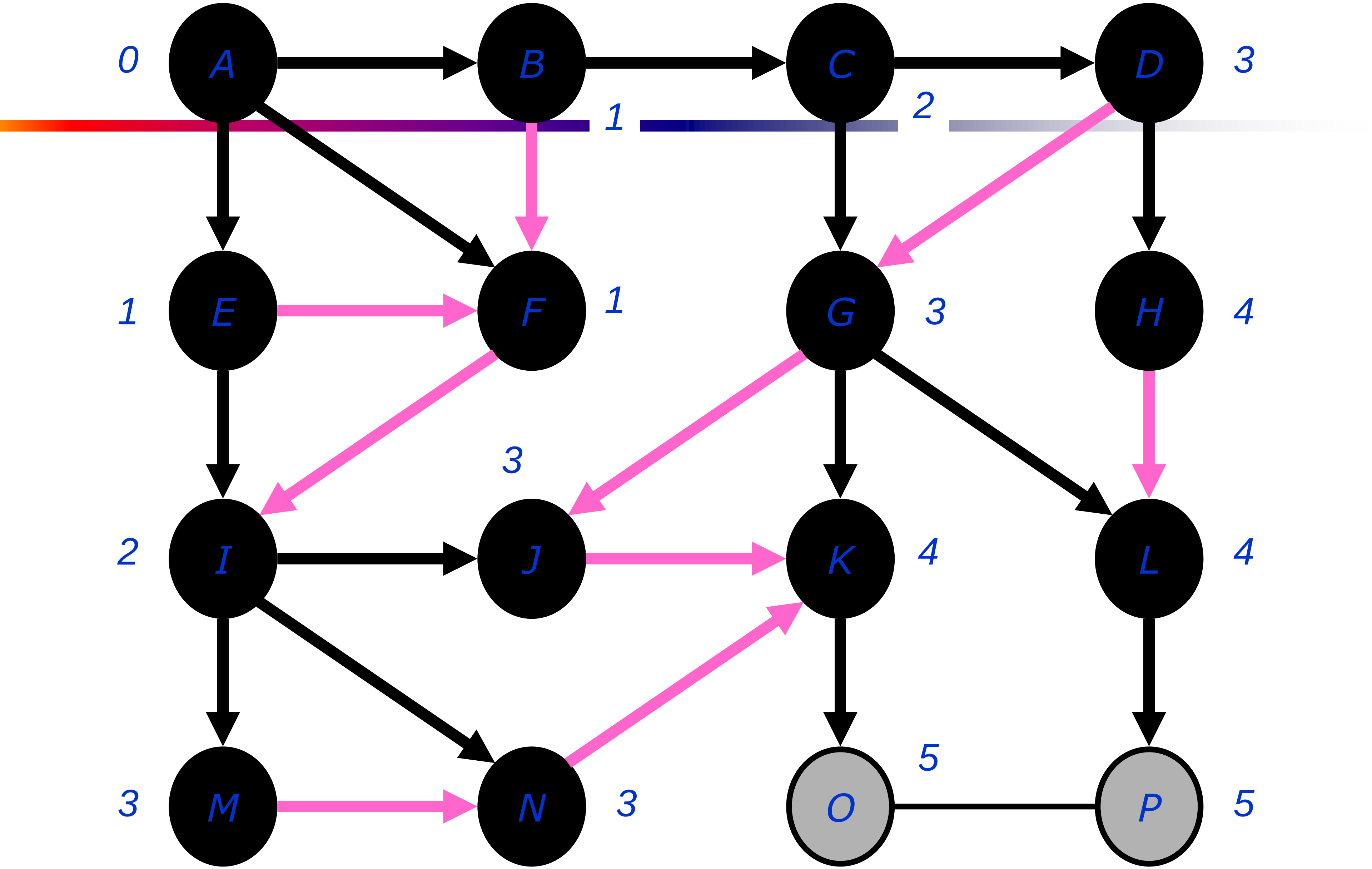
0
A
B
C
D
1
2
1
E
F
1
G
H
2
I
J
K
L
M
N
O
P



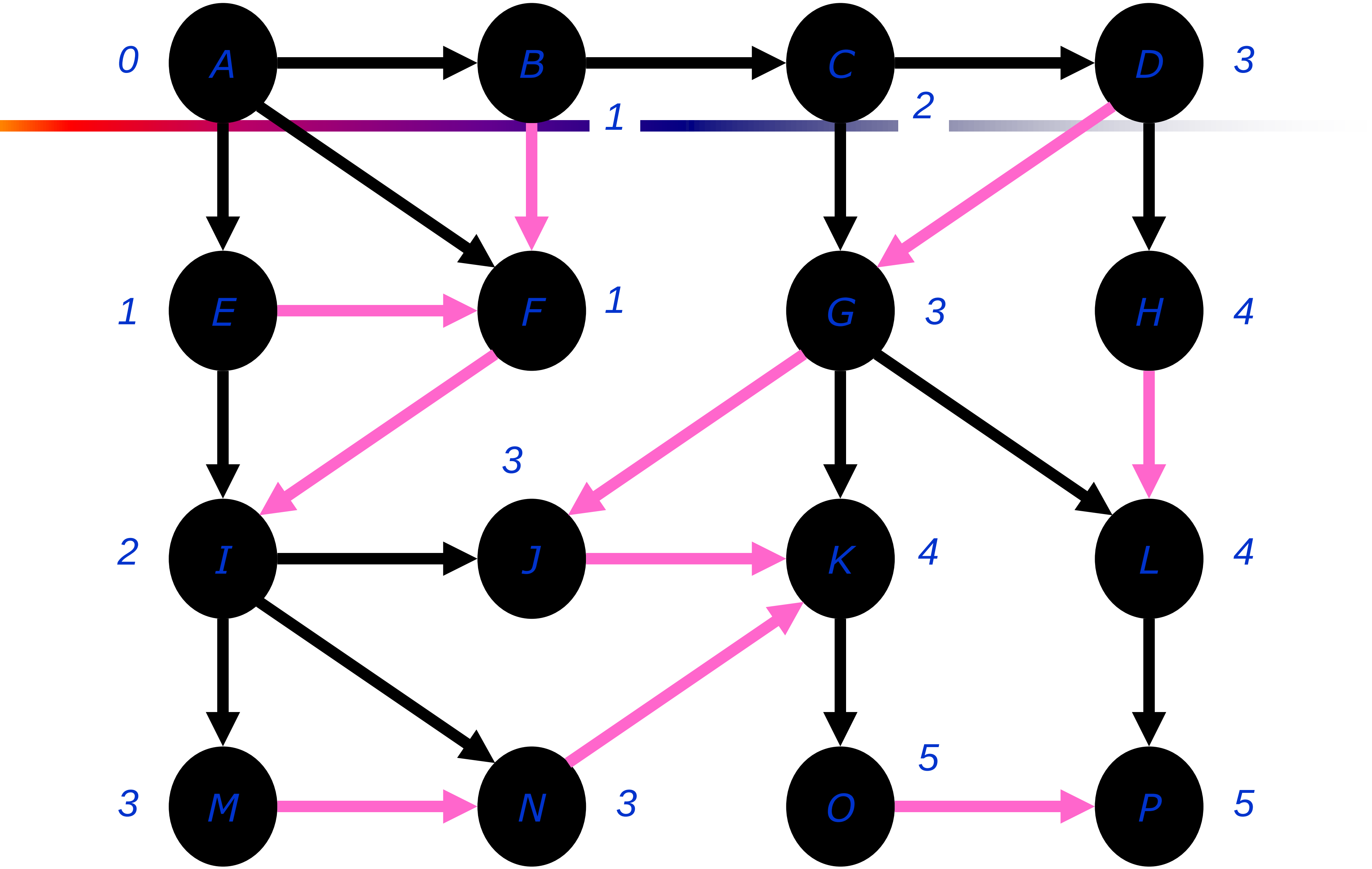
0
A
B
C
D
3
1
2
1
E
F
1
G
3
H
3
2
I
J
K
L
3
M
N
3
O
P

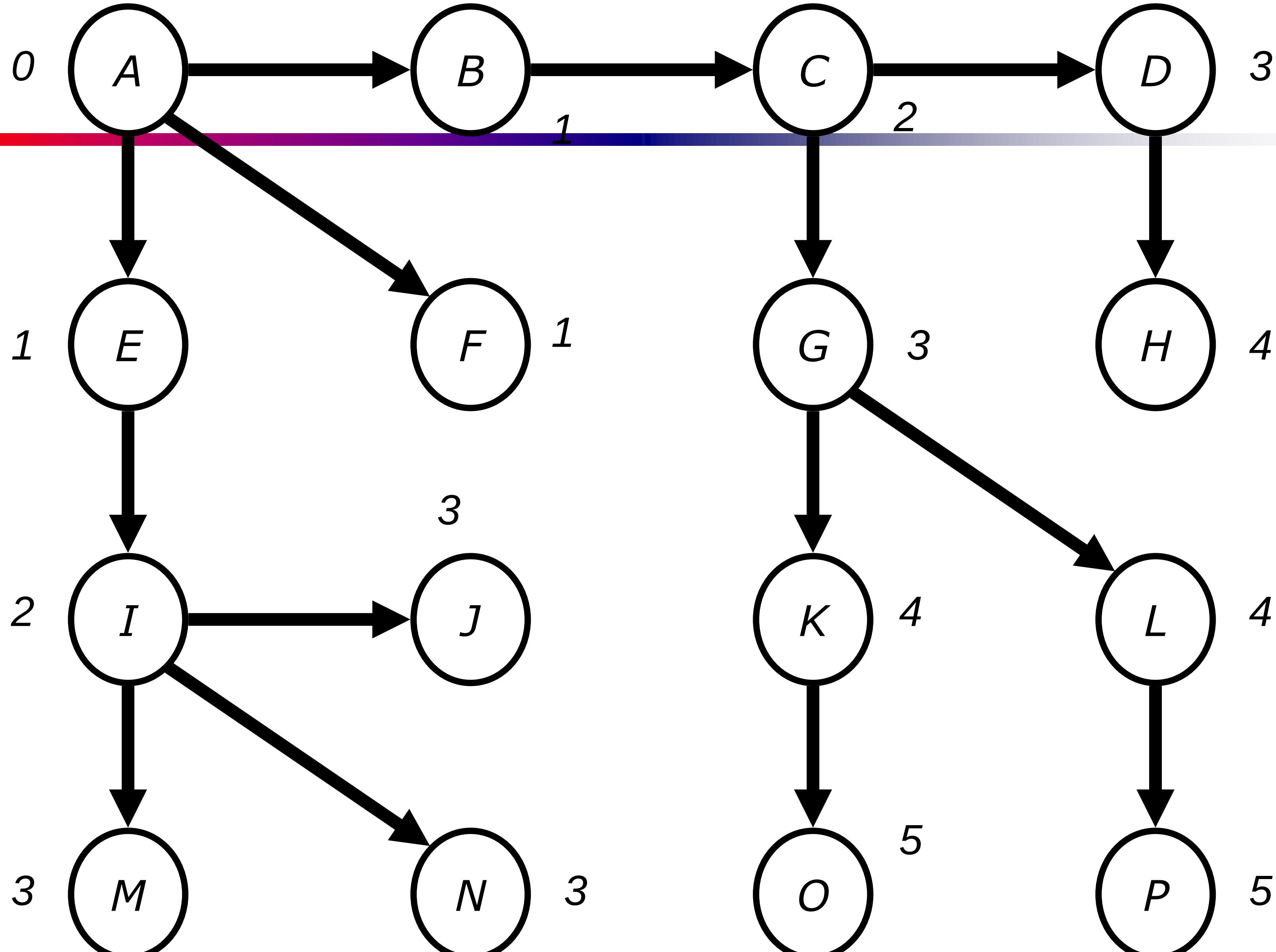


0
A
B
C
D
3
1
2
1
E
F
1
G
3
H
4
3
2
I
J
K
4
L
4
3
M
N
3
O
P



A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
0
1
2
3
1
1
3
4
3
2
4
4
3
3
5
5


0
A
B
C
D
3
1
2
E
F
G
H
1
1
3
4
3
I
J
K
L
2
4
4
M
N
O
P
3
3
5
5

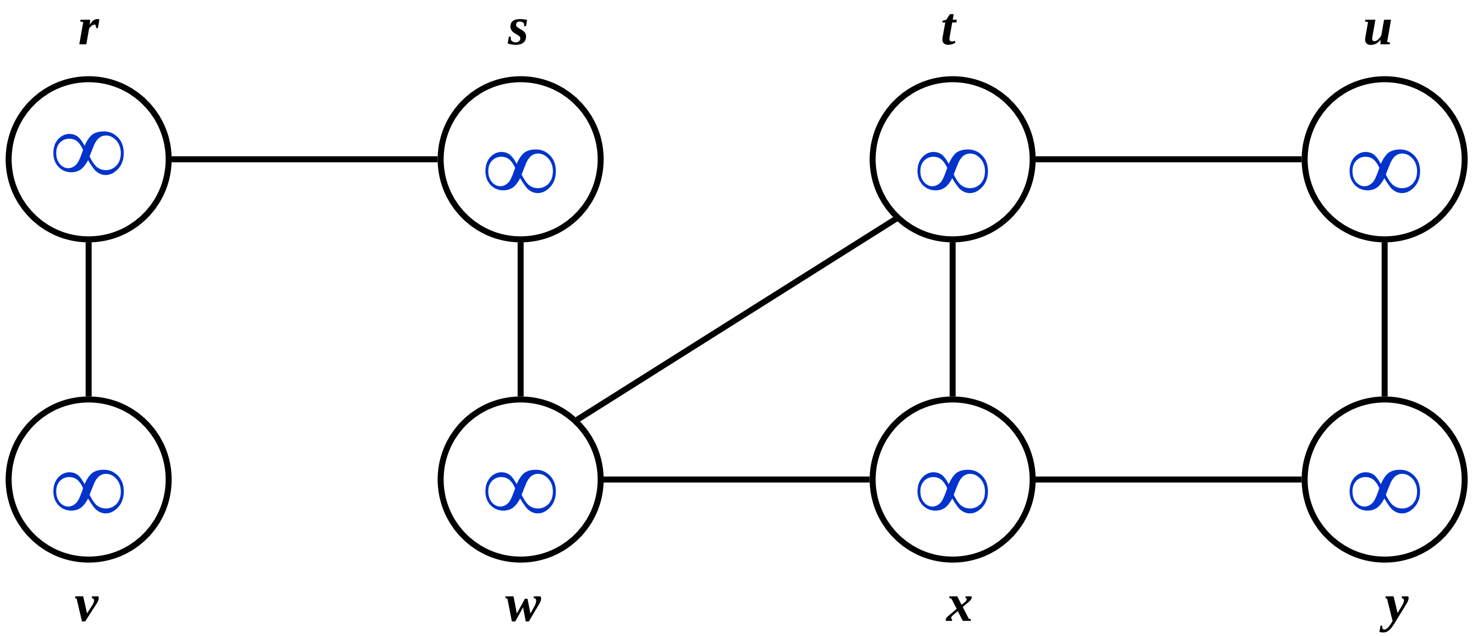


0
A
B
C
D
3
1
2
1
E
F
1
G
3
H
4
3
2
I
J
K
4
L
4
3
M
N
3
O
5
P
5

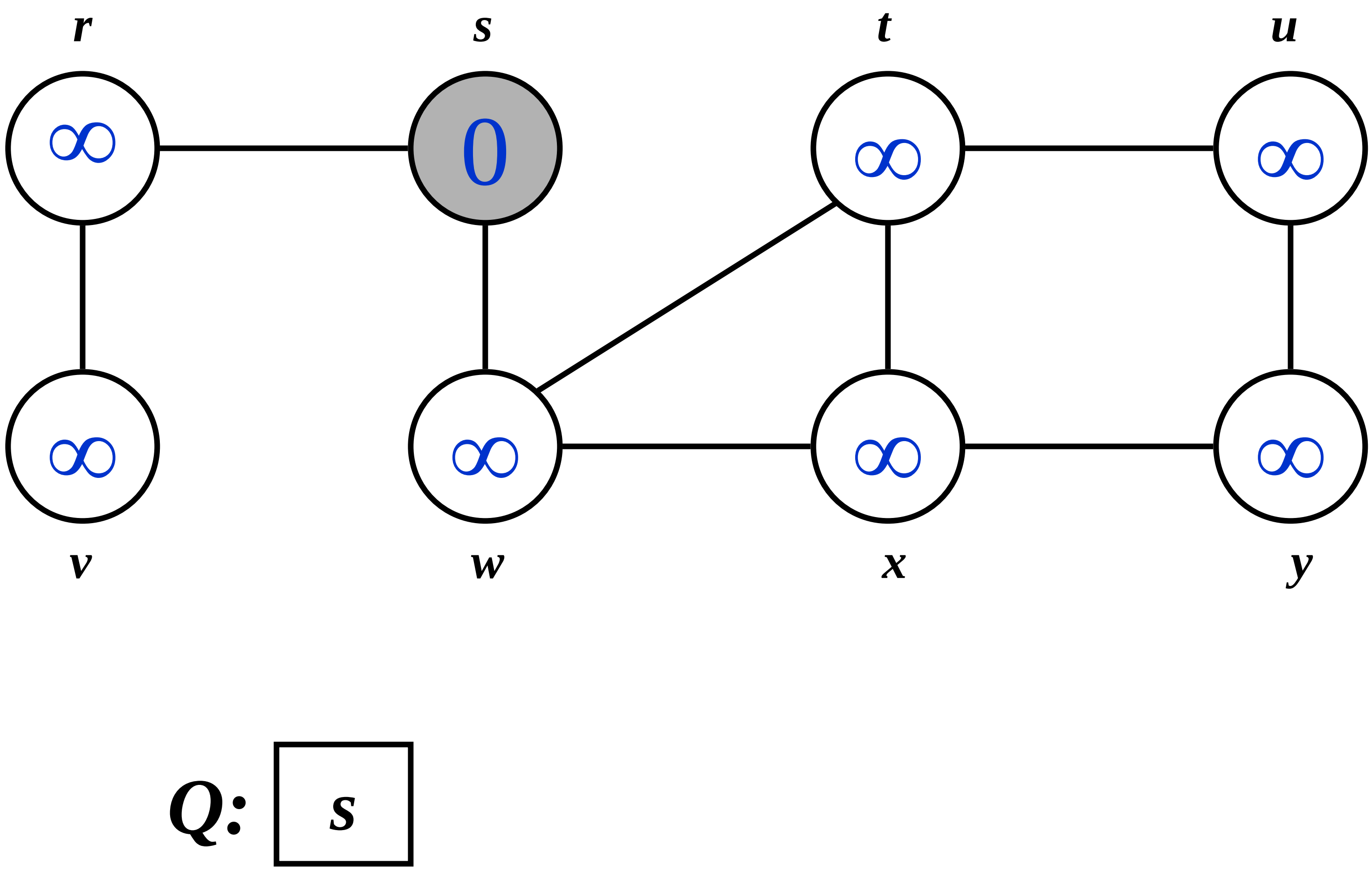
Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};        // Q is a queue (duh); initialize to s
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

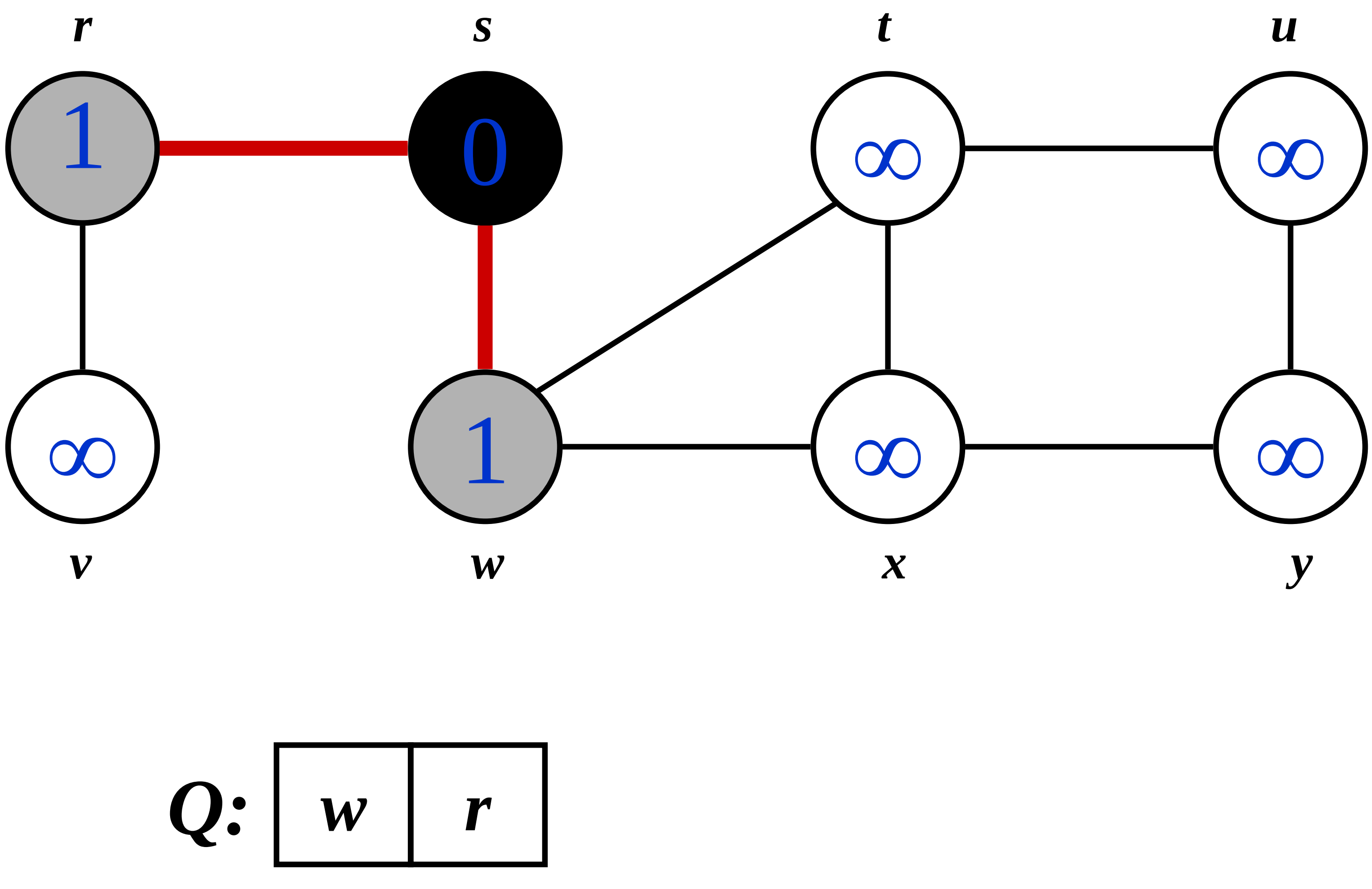
Breadth-First Search: Example



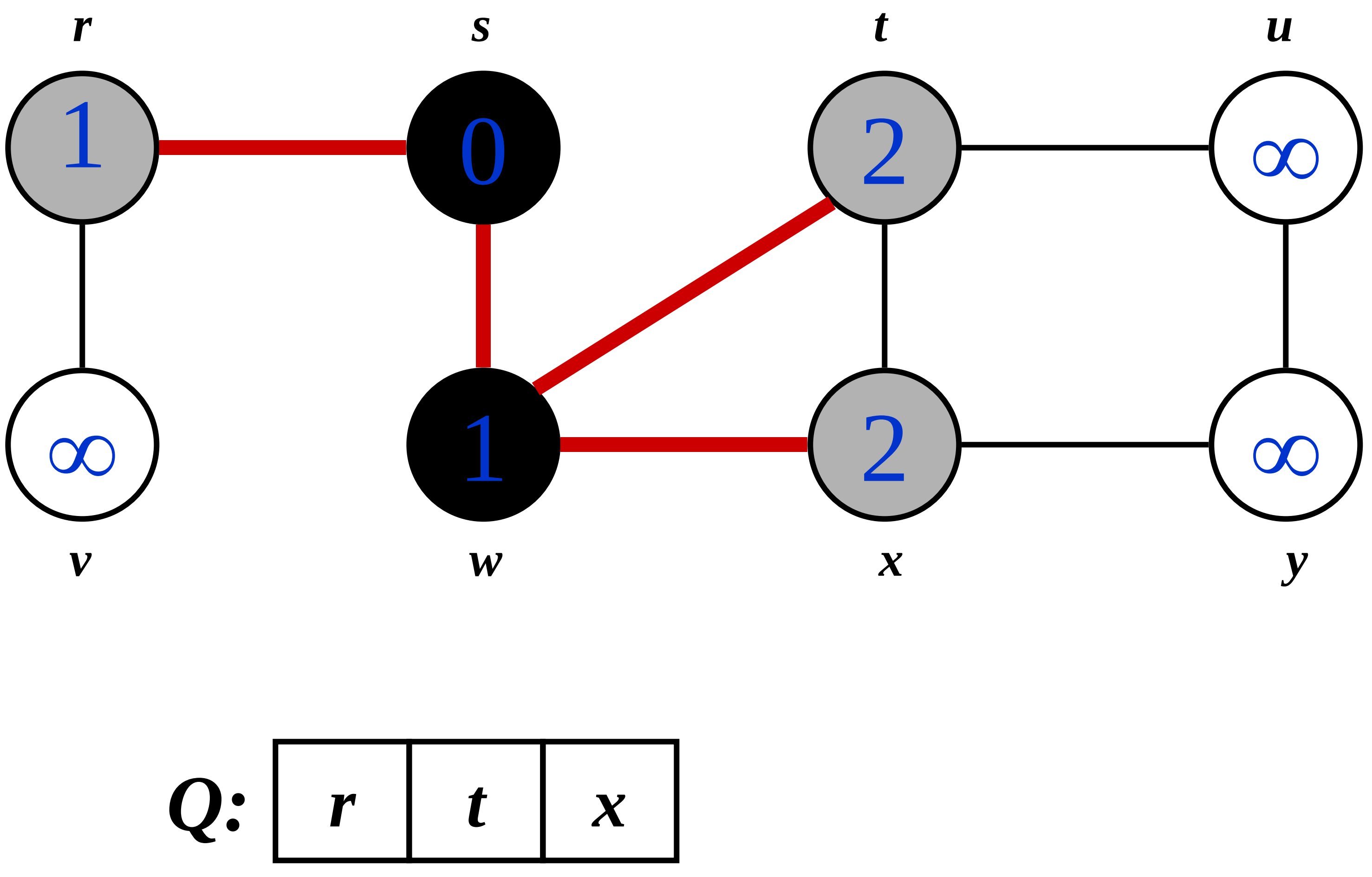
Breadth-First Search: Example



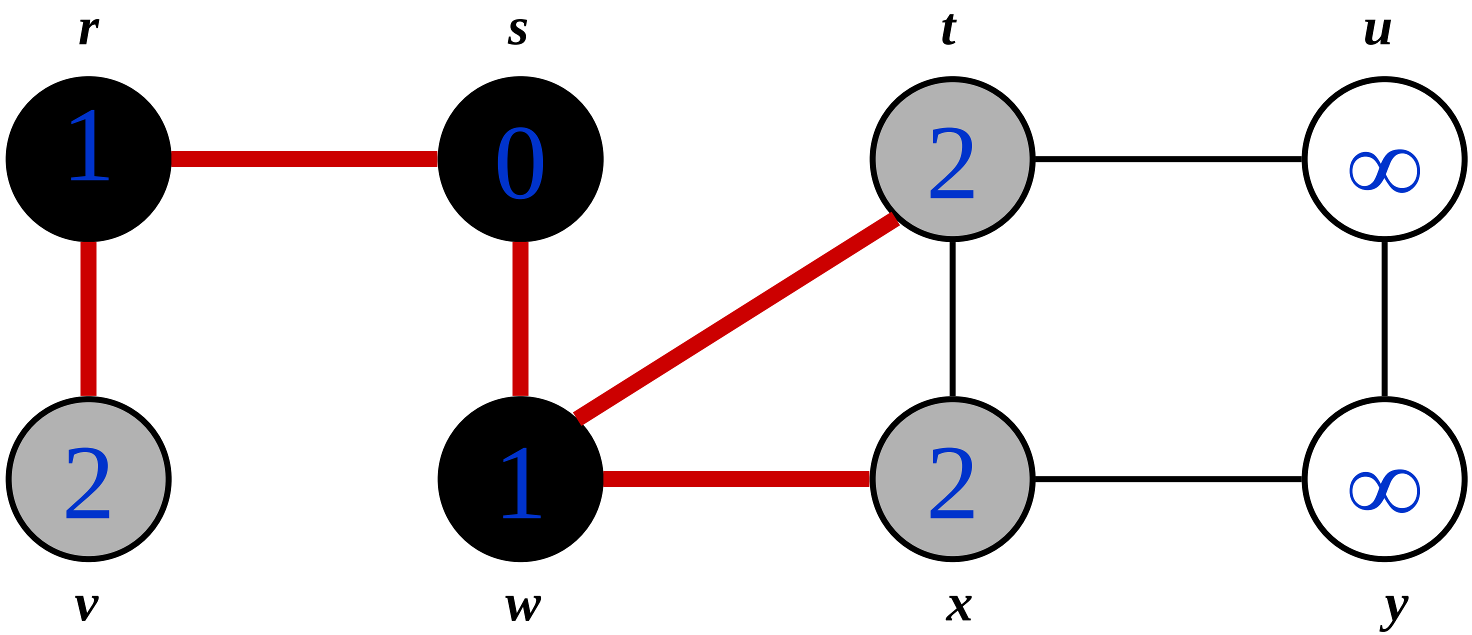
Breadth-First Search: Example



Breadth-First Search: Example



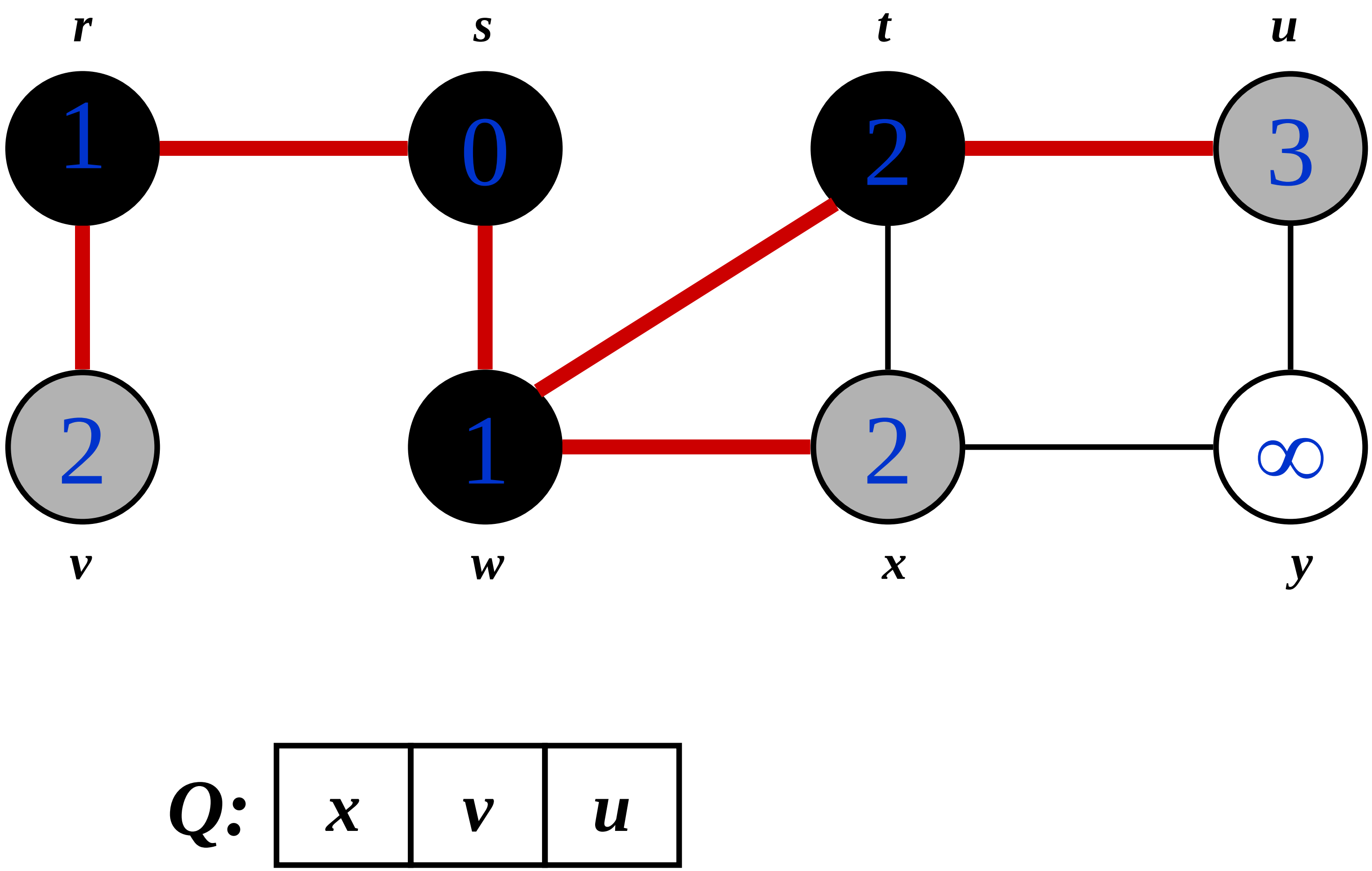
Breadth-First Search: Example



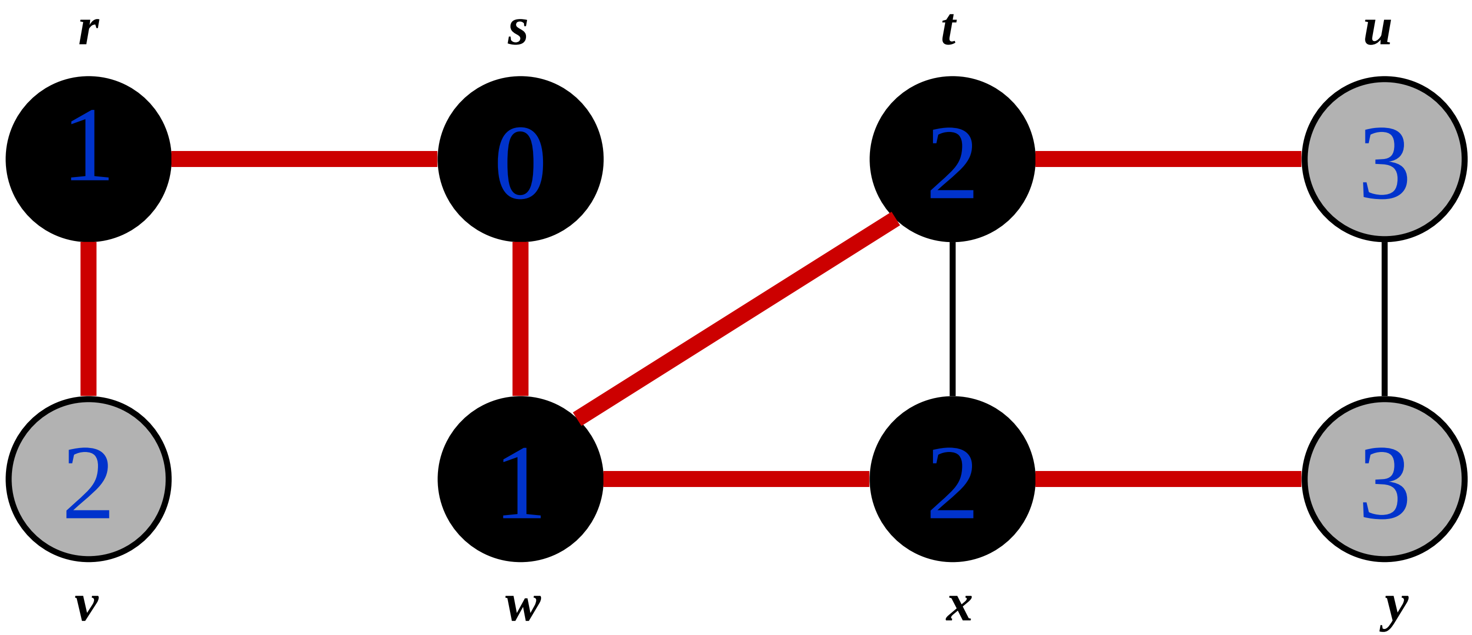
Q:

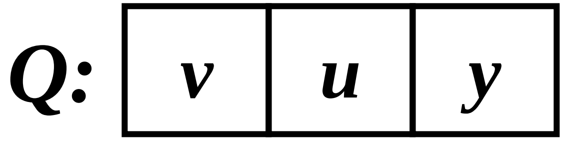
t	*x*	*v*

Breadth-First Search: Example

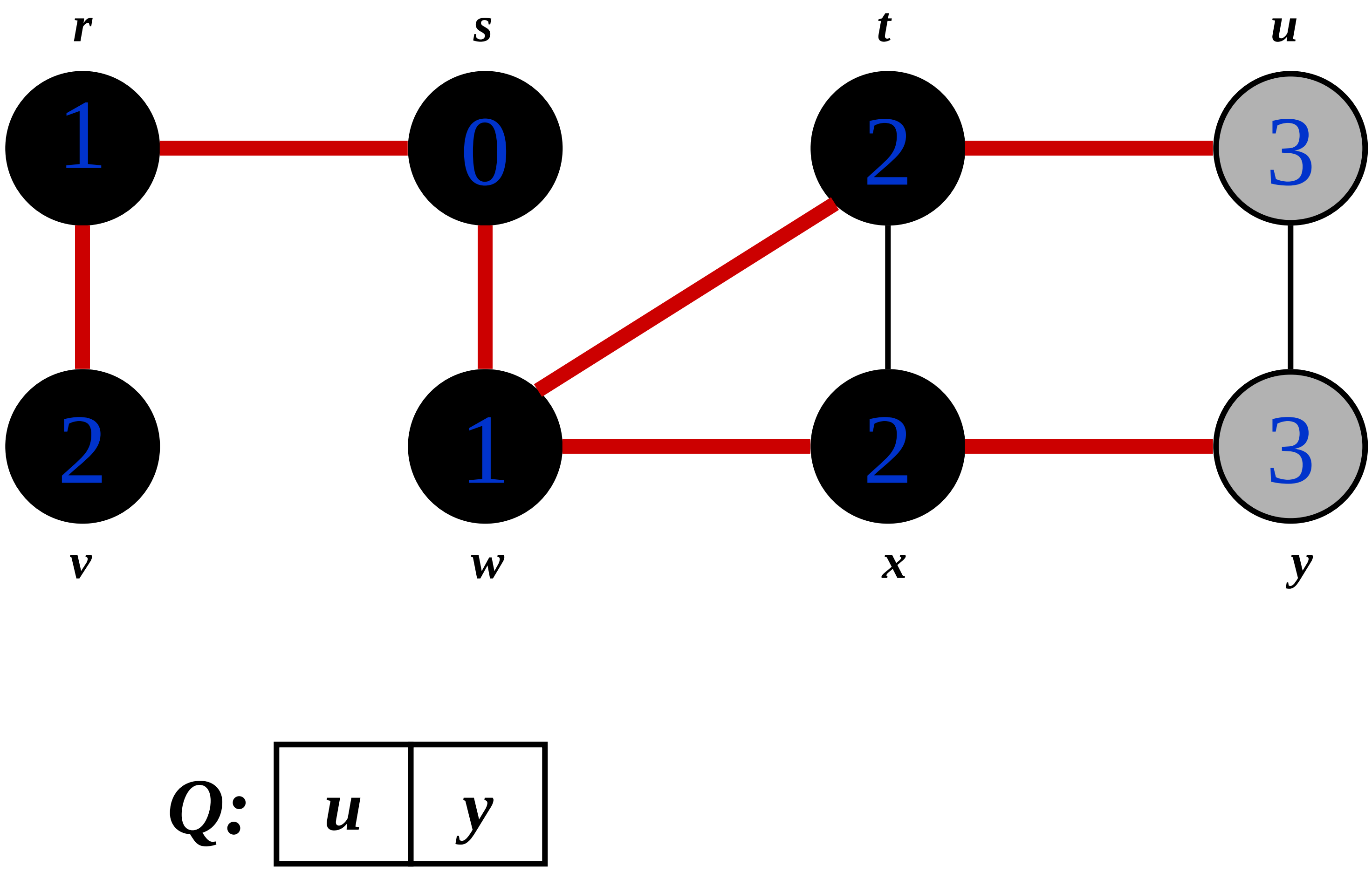


Breadth-First Search: Example

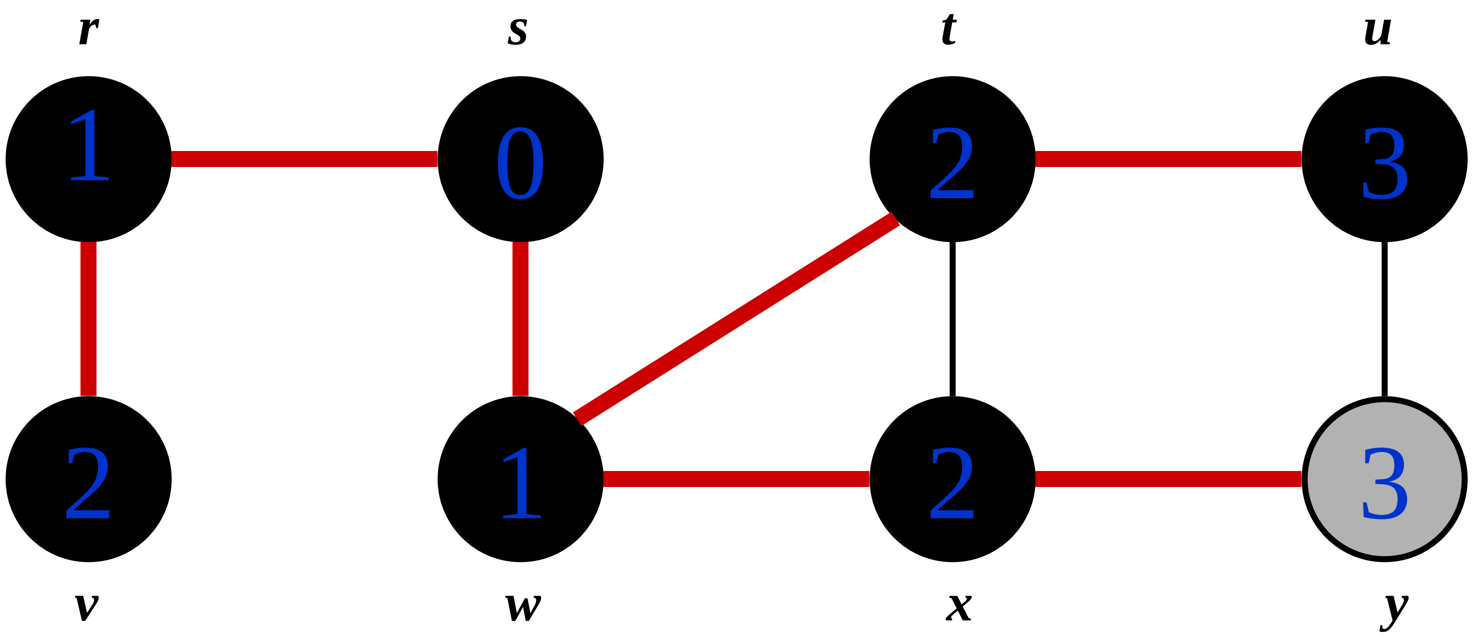




Breadth-First Search: Example

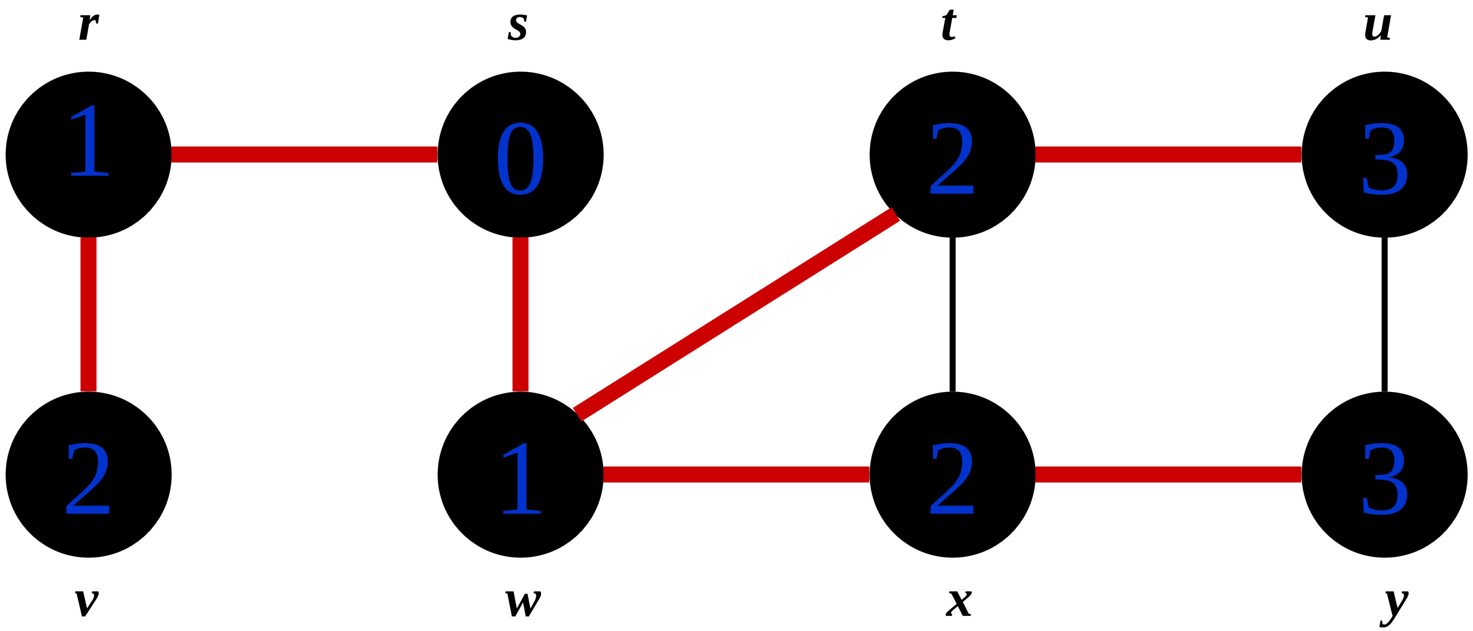


Breadth-First Search: Example



Q: y

Breadth-First Search: Example



Q: Ø

BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

Touch every vertex: O(V)

u = every vertex, but only once

So v = every vertex that appears in some other vert's adjacency list

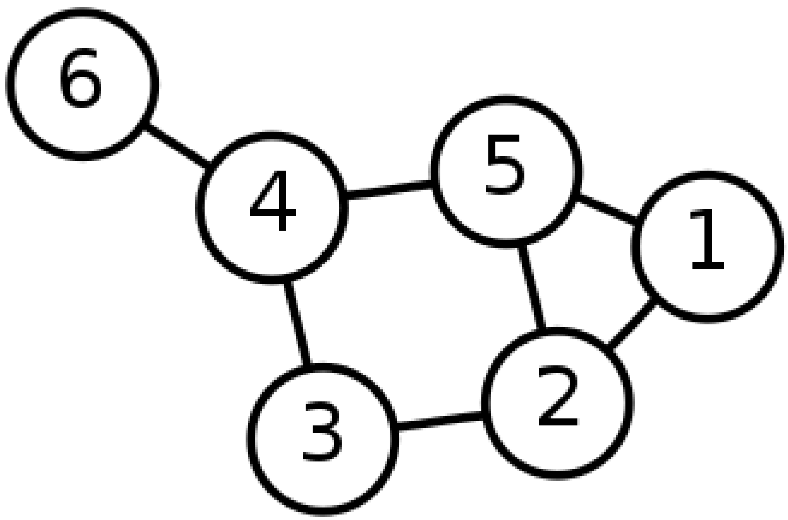
What will be the running time?

Total running time: O(V+E)

Dijkstra’s Algorithm

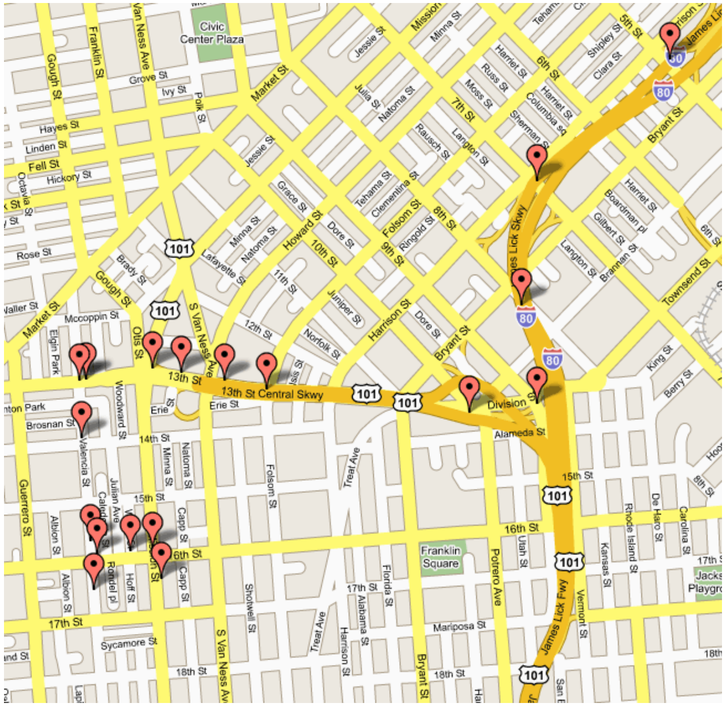
Single-Source Shortest Path Problem

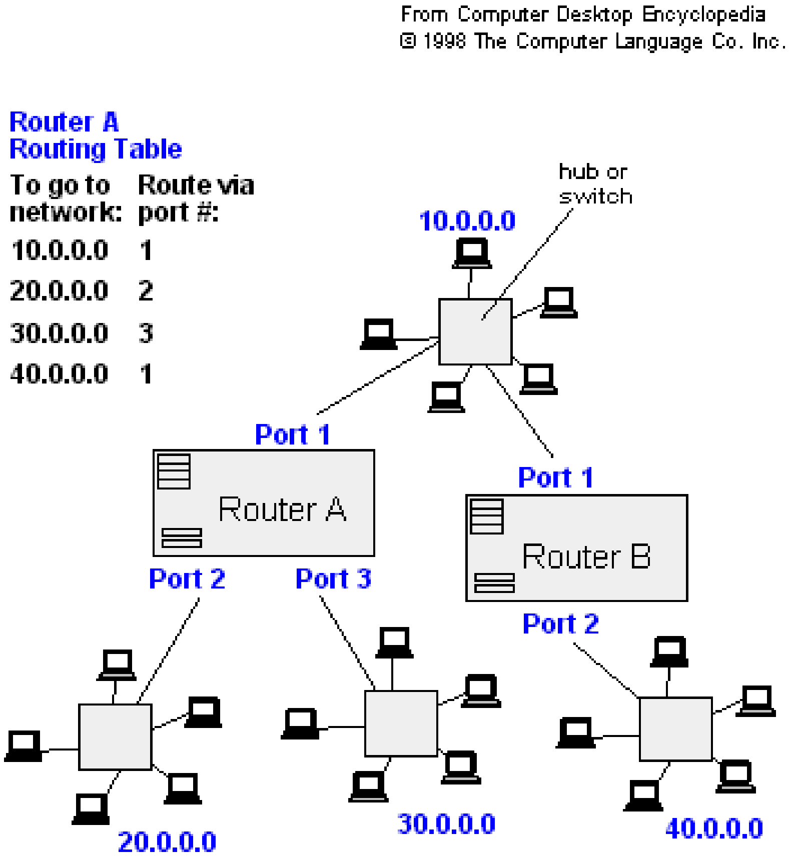
<u>Single-Source Shortest Path Problem</u> - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Applications

- Maps (Map Quest, Google Maps)
- Routing Systems





Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Approach

- The algorithm computes for each vertex u the distance to u from the start vertex v, that is, the weight of a shortest path between v and u.
- the algorithm keeps track of the set of vertices for which the distance has been computed, called the cloud C
- Every vertex has a label D associated with it. For any vertex u, D[u] stores an approximation of the distance between v and u. The algorithm will update a D[u] value when it finds a shorter path from v to u.
- When a vertex u is added to the cloud, its label D[u] is equal to the actual (final) distance between the starting vertex v and vertex u.

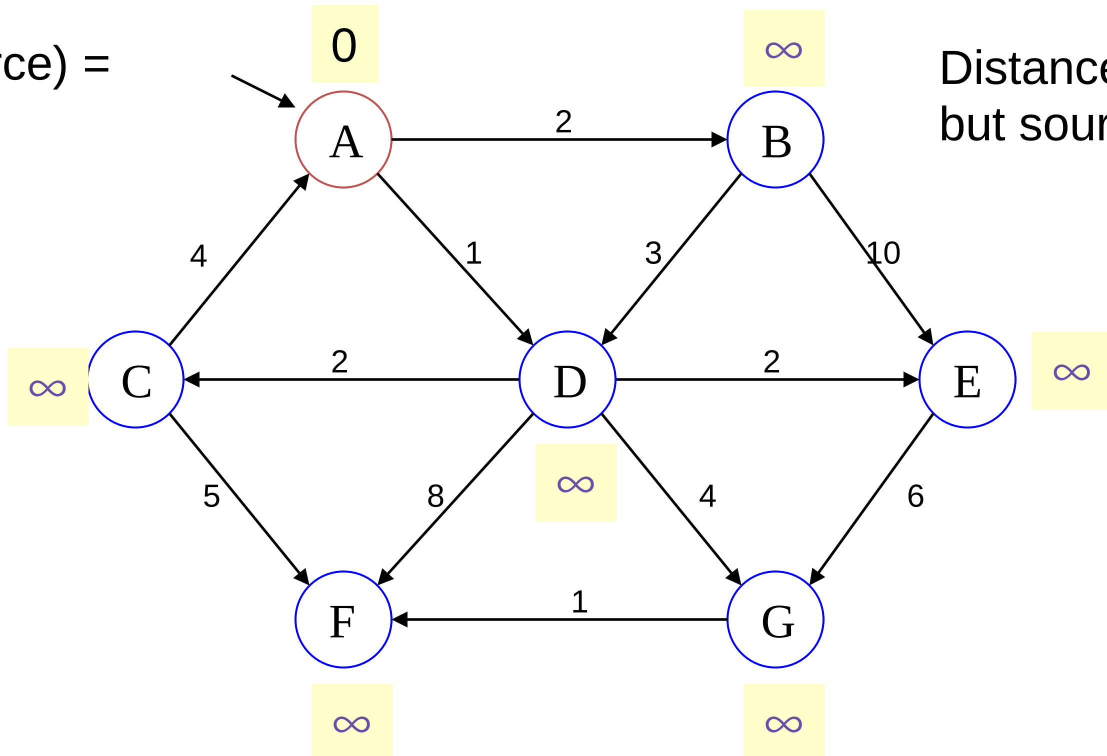
Dijkstra pseudocode

```
Dijkstra(v1, v2):
  for each vertex v:                    // Initialization
     v's distance := infinity.
     v's previous := none.
  v1's distance := 0.
  List := {all vertices}.

  while List is not empty:
     v := remove List vertex with minimum distance.
     mark v as known.
     for each unknown neighbor n of v:
        dist := v's distance + edge (v, n)'s weight.

        if dist is smaller than n's distance:
           n's distance := dist.
           n's previous := v.

  reconstruct path from v2 back to v1,
  following previous pointers.
```

Example: Initialization

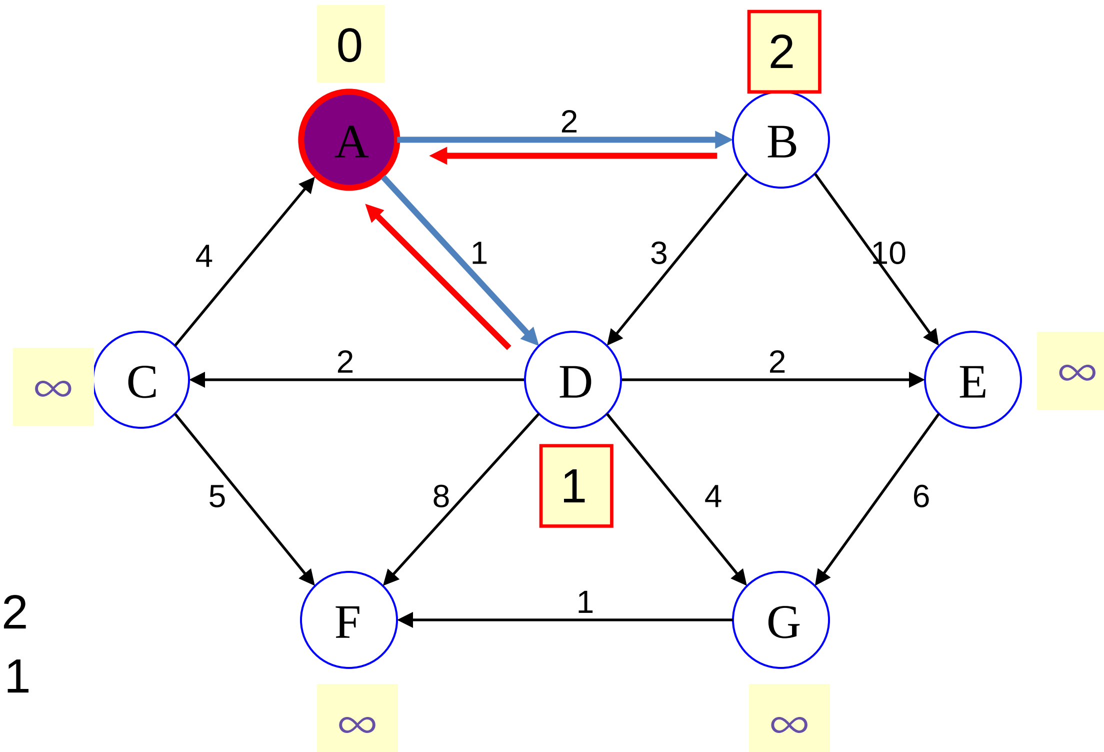
Distance(source) = 0

Distance (all vertices but source) = ∞



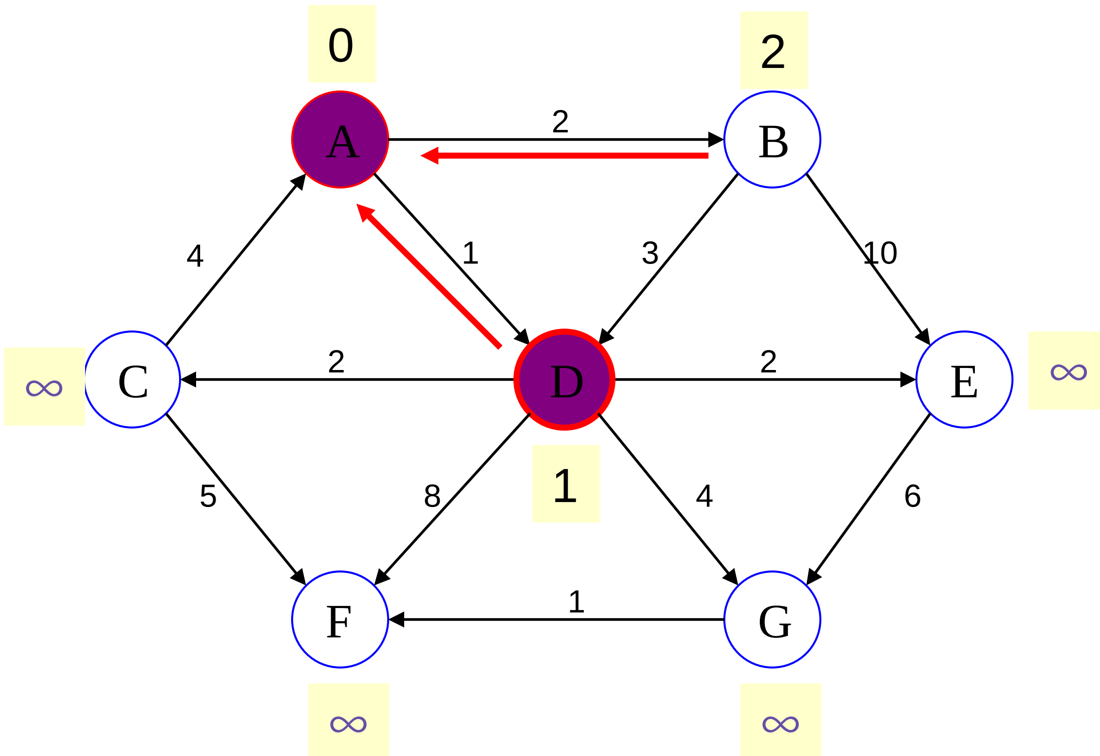
Pick vertex in List with minimum distance.

Example: Update neighbors' distance



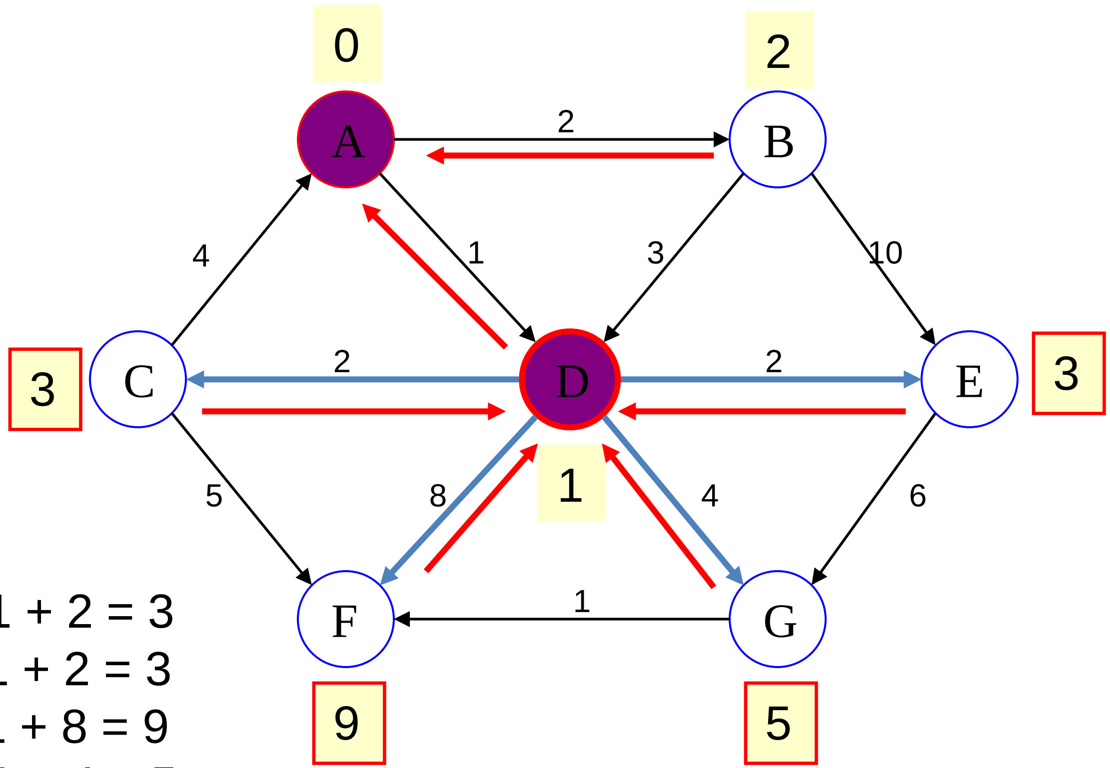
Distance(B) = 2
Distance(D) = 1

Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

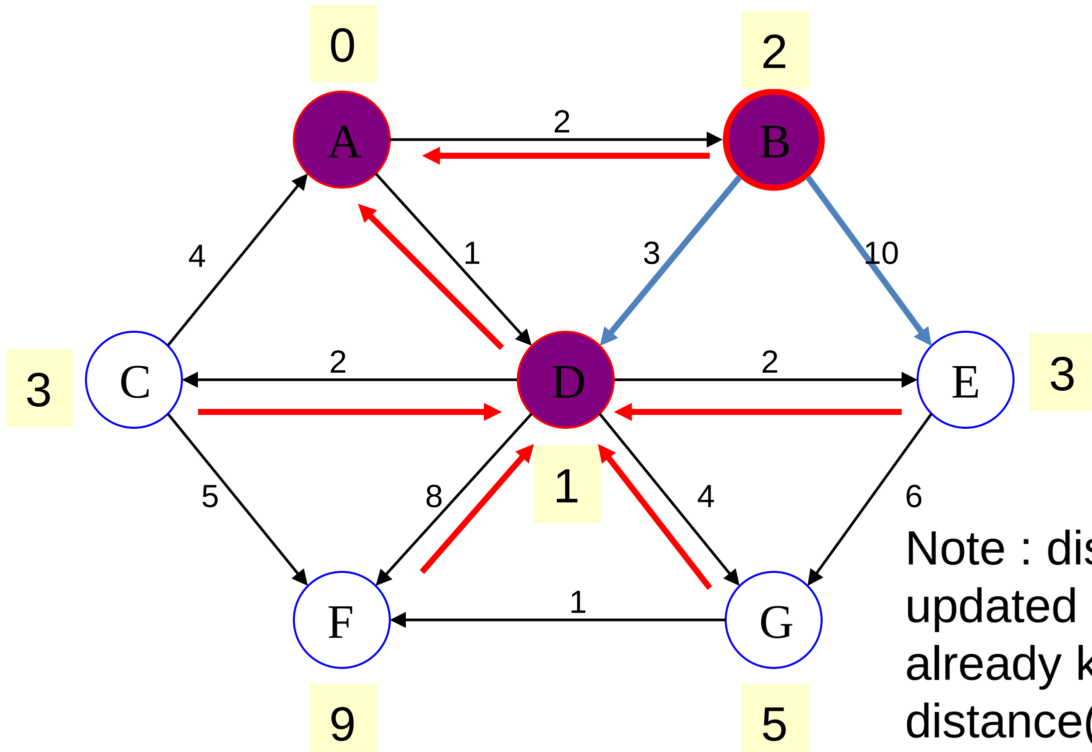
Example: Update neighbors



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example: Continued...

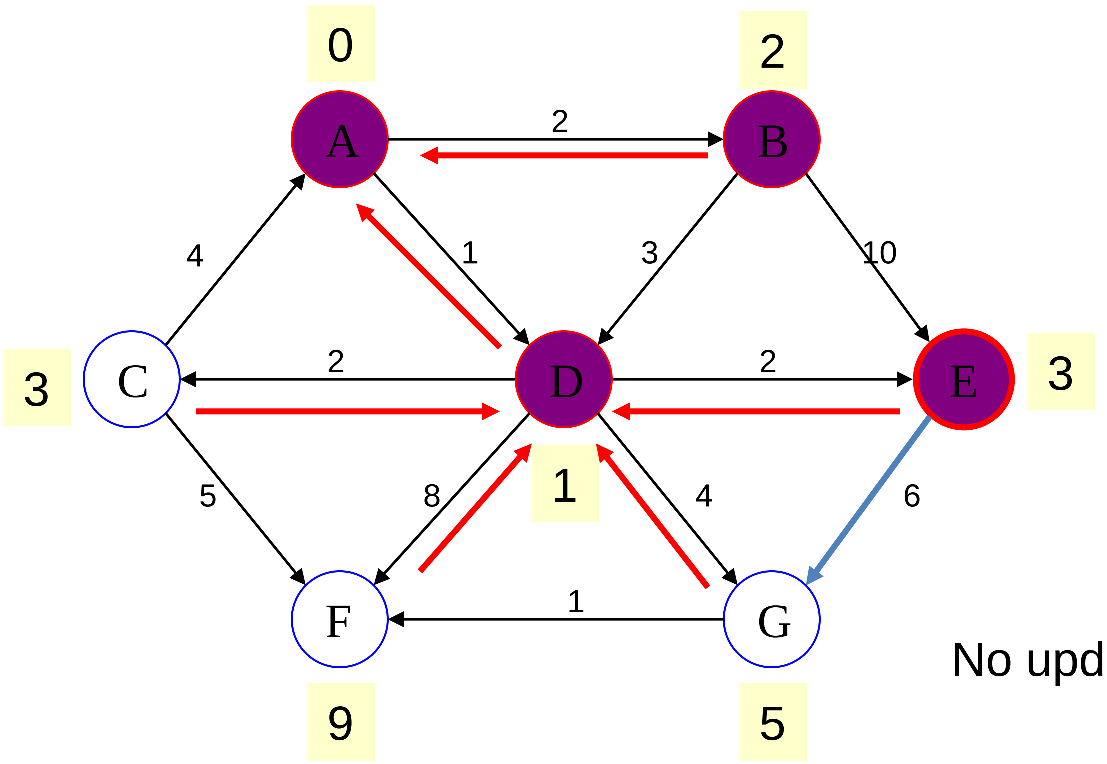
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

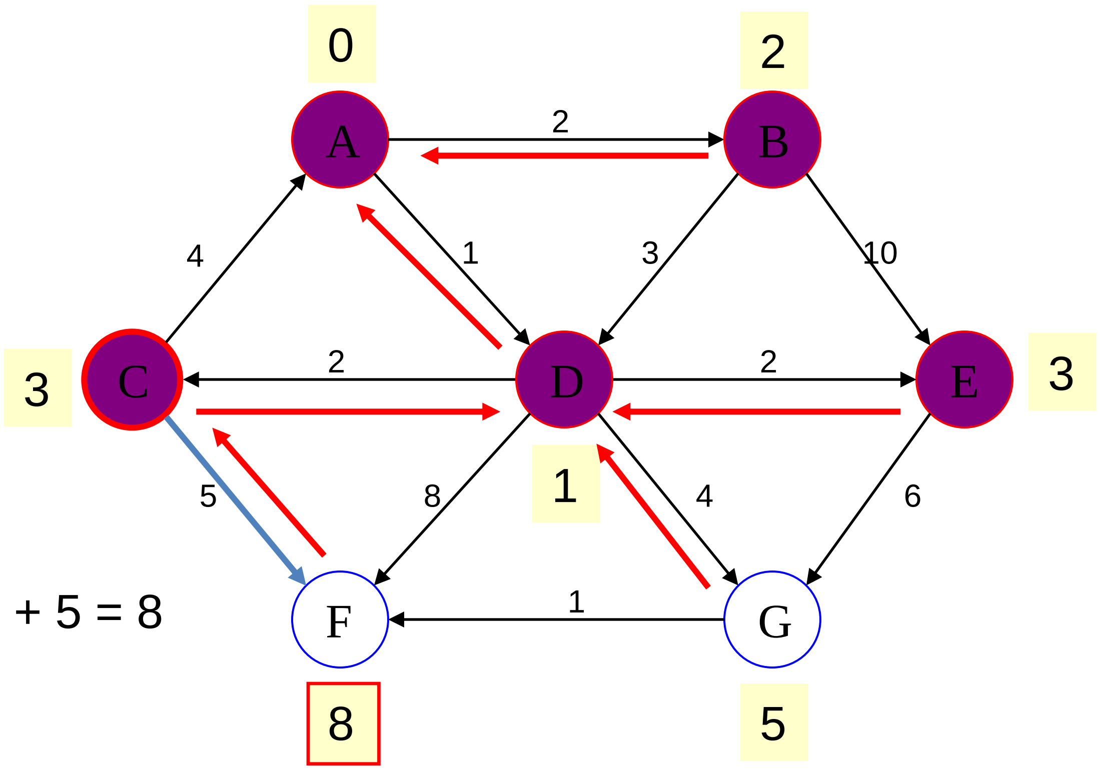
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

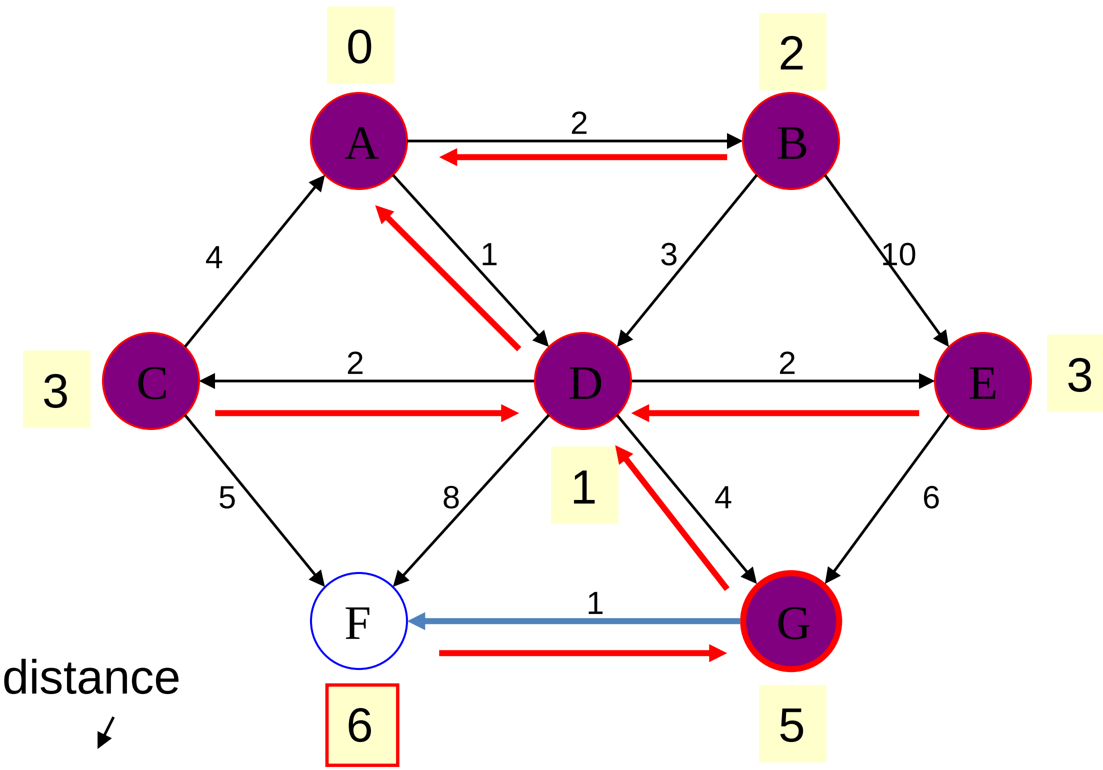
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

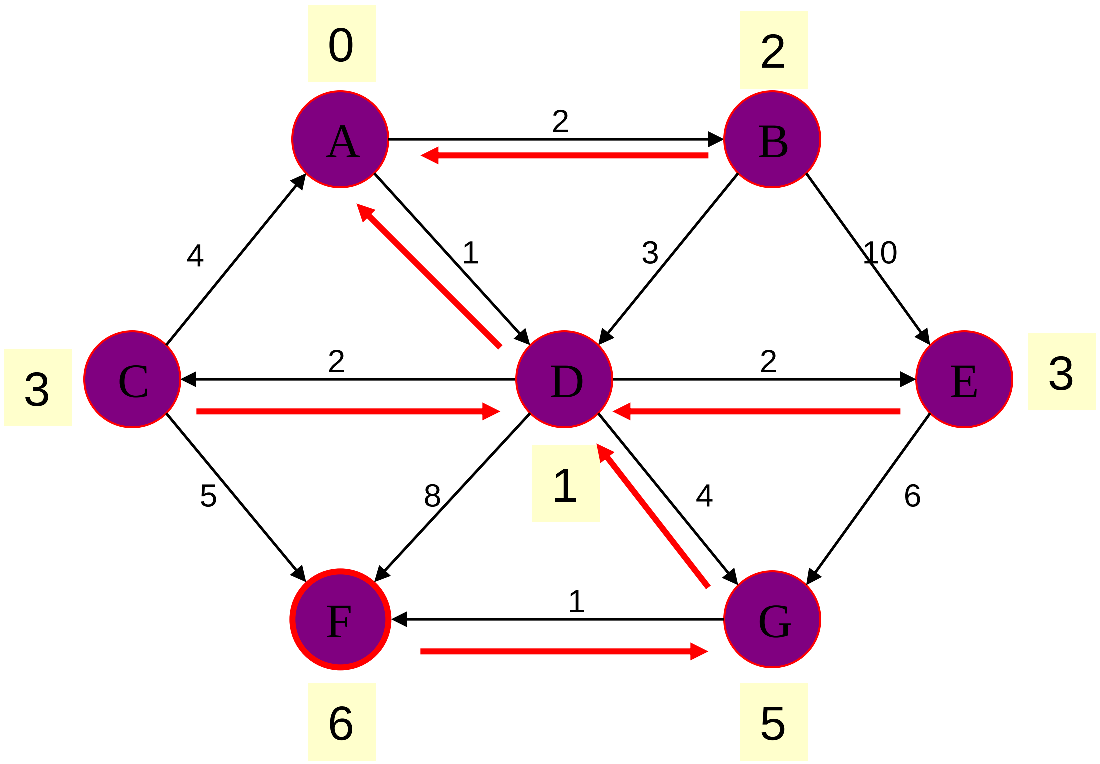
Pick vertex List with minimum distance (G) and update neighbors



Previous distance

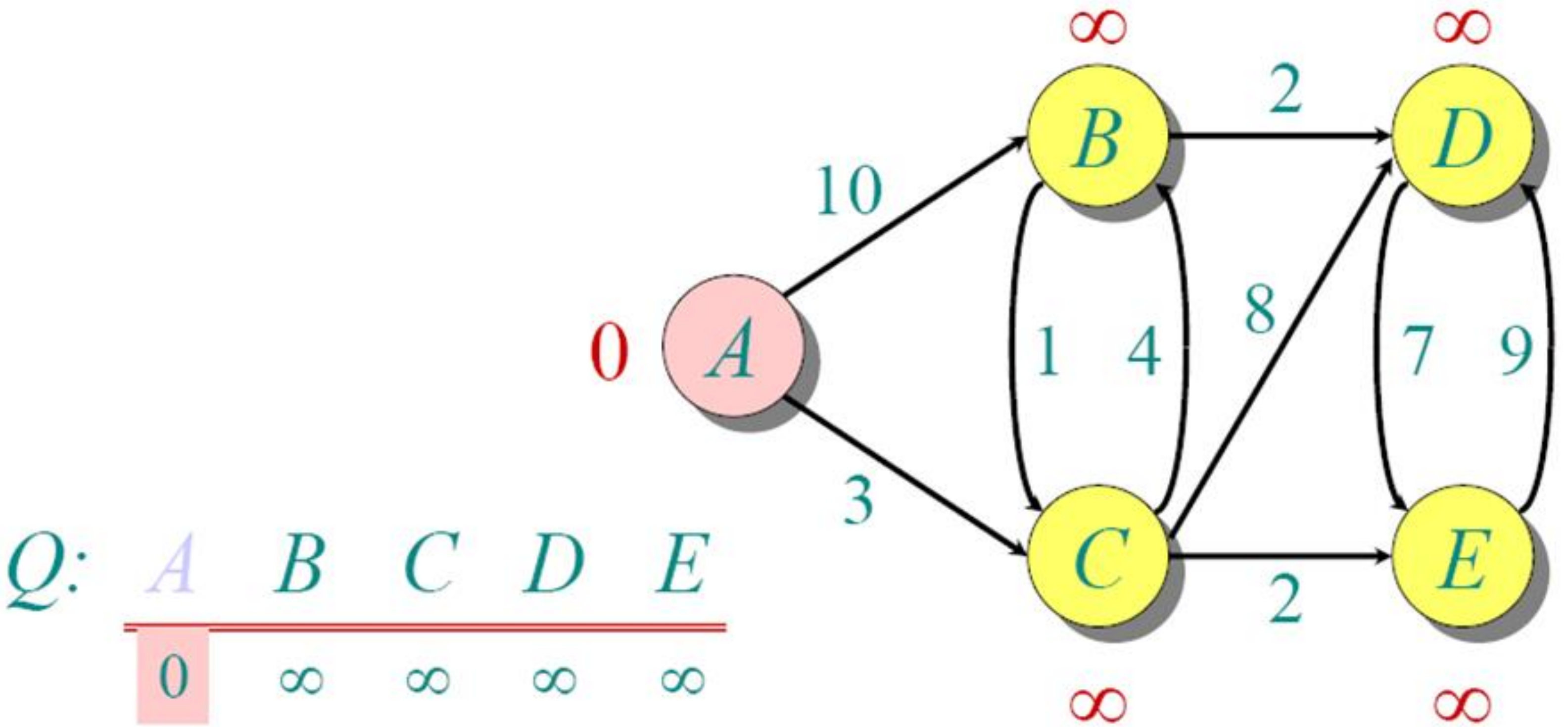
Distance(F) = min (8, 5+1) = 6

Example (end)

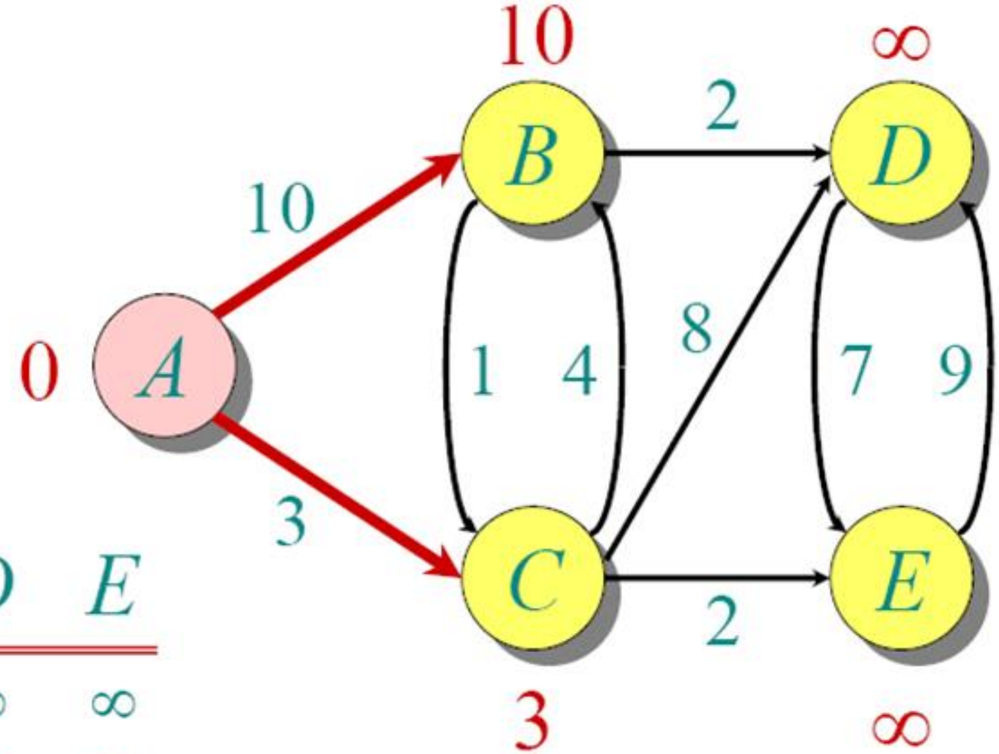


Pick vertex not in S with lowest cost (F) and update neighbors

Another Example



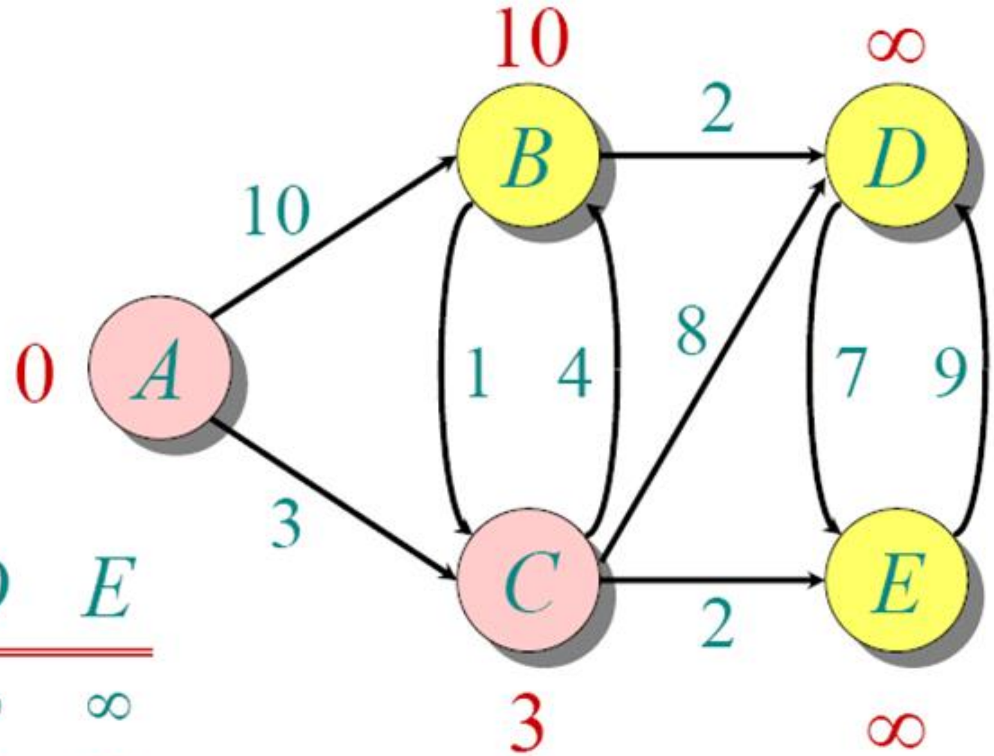
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

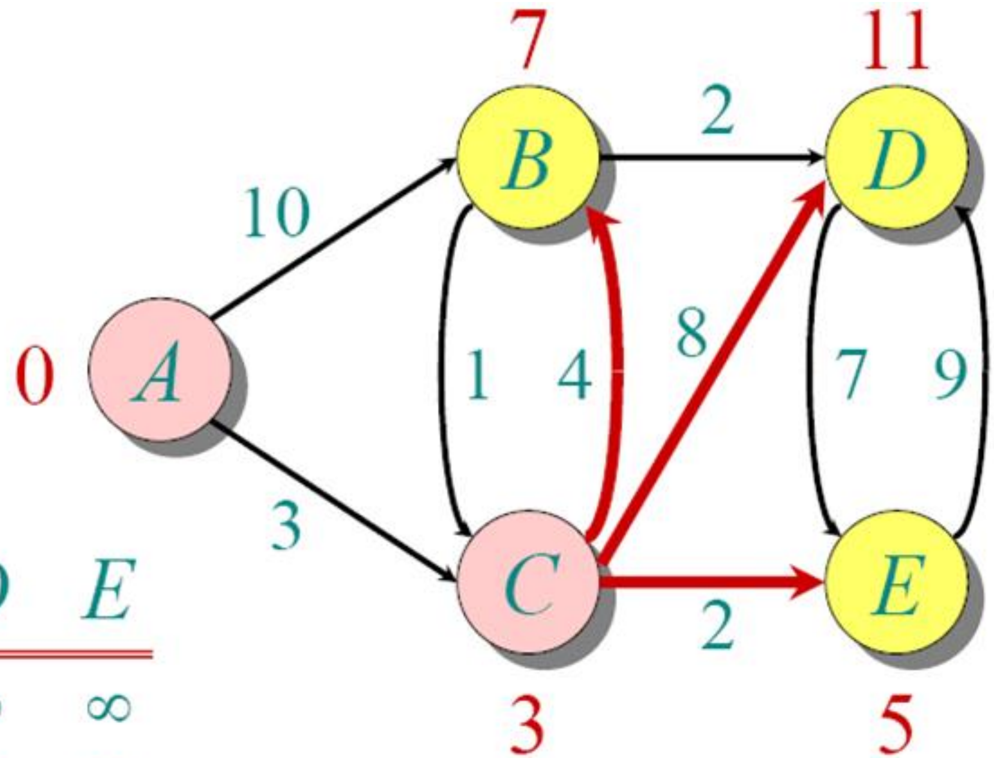
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

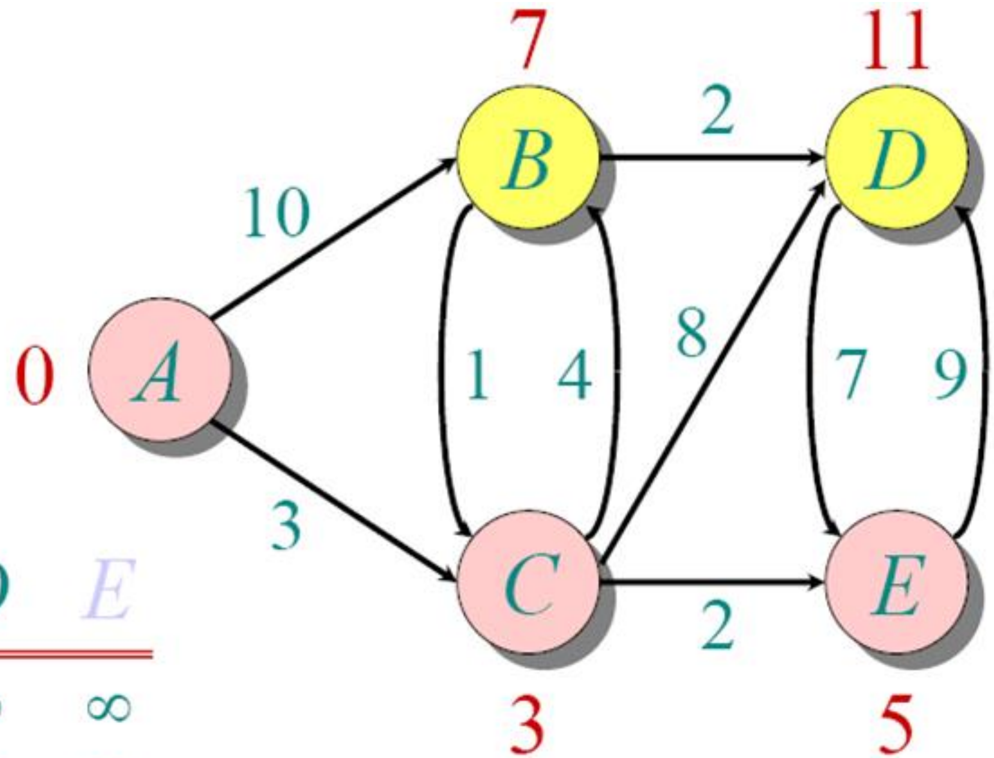
Another Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	5

S: $\{ A, C \}$

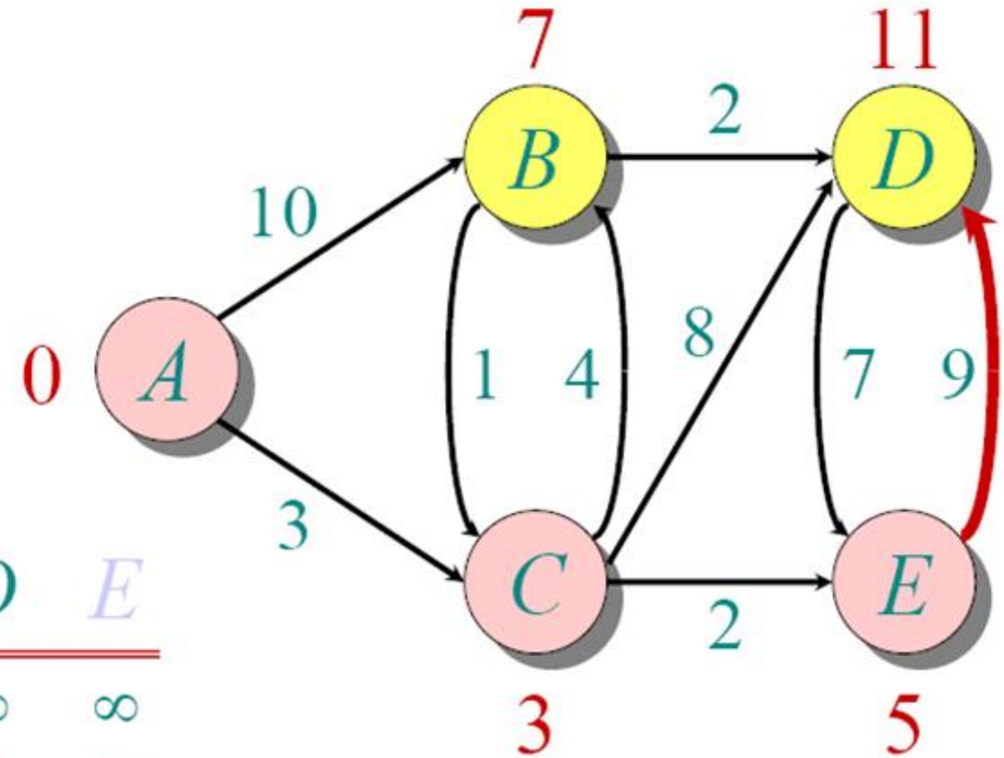
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

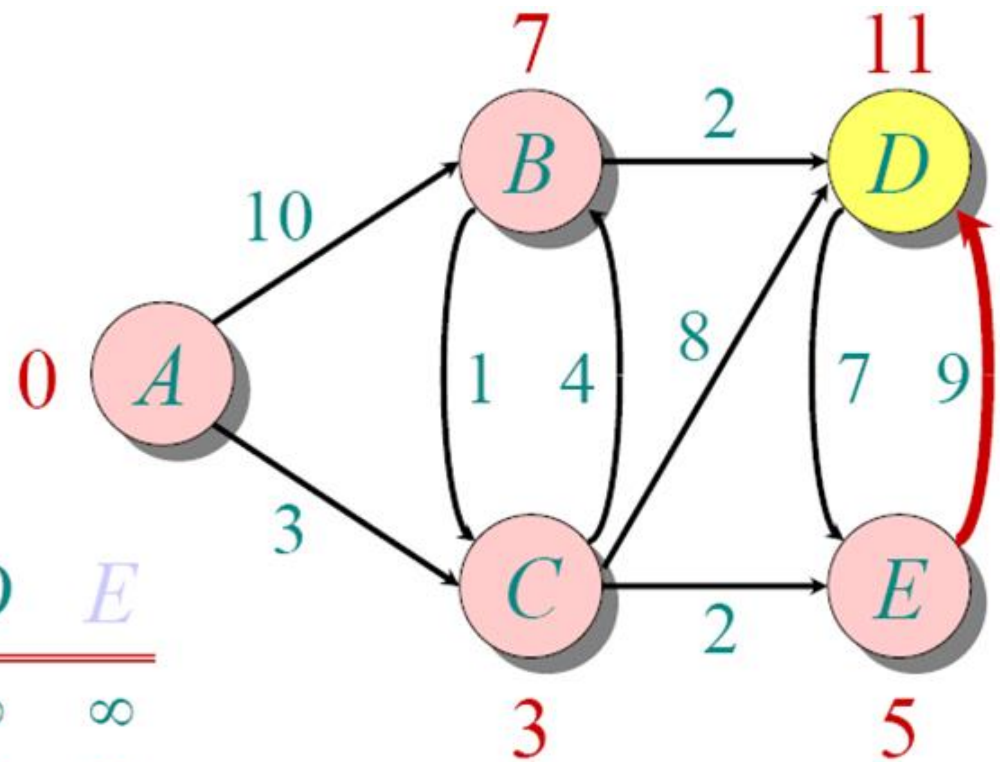
Another Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		7		11	

S: { A, C, E }

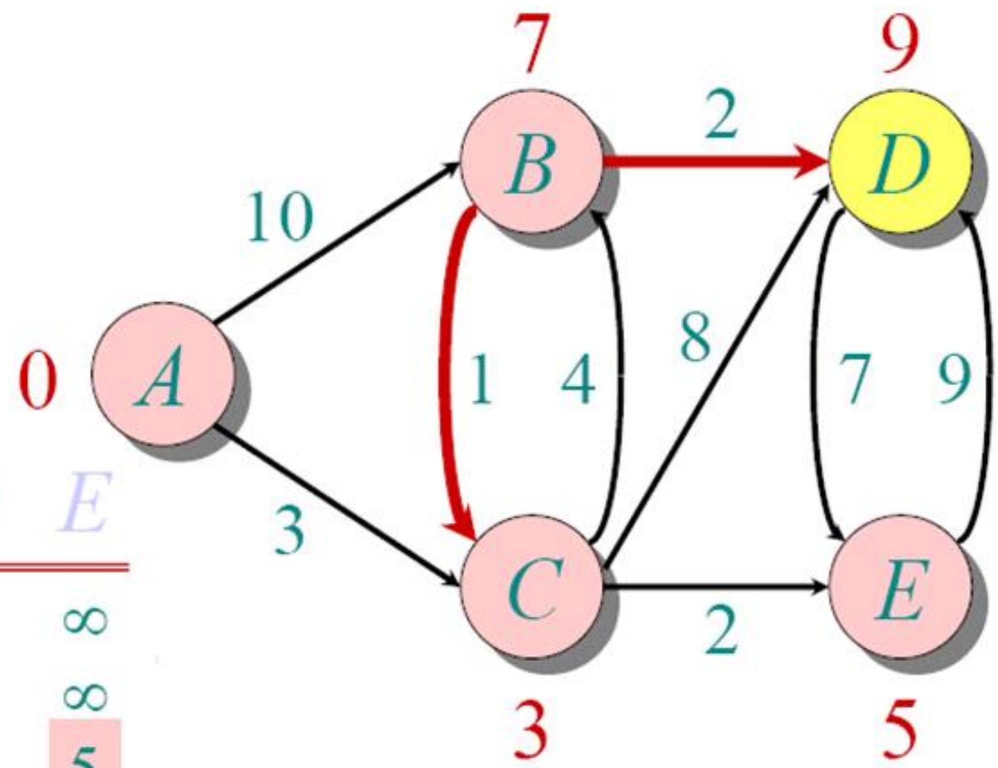
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E*, *B* }

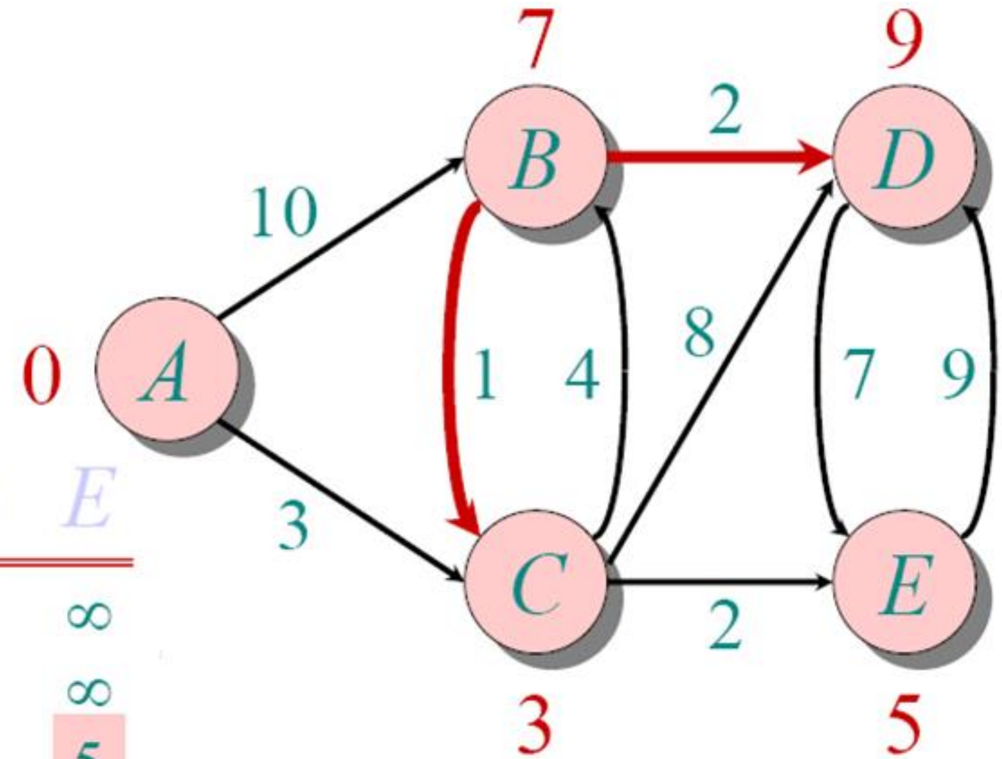
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Another Example

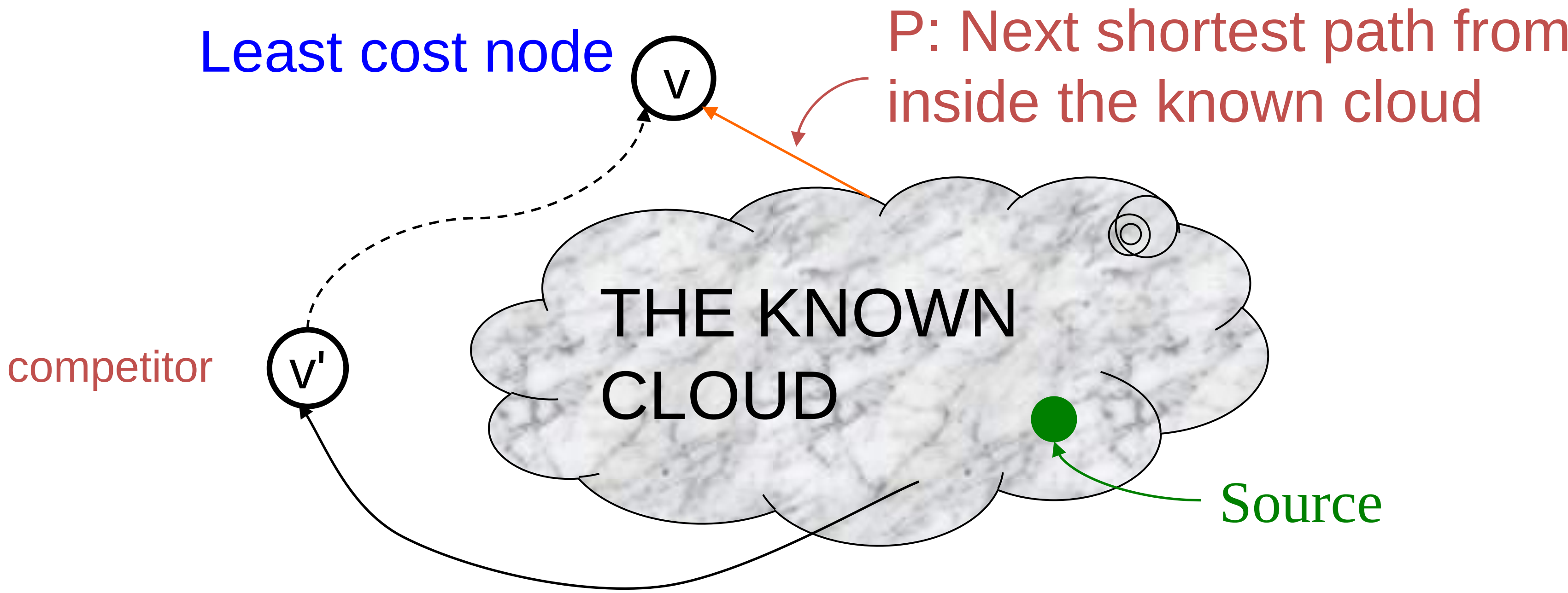


Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	

S: { A, C, E, B, D }

Correctness :"Cloudy" Proof

When a vertex is added to the cloud, it has shortest distance to source.

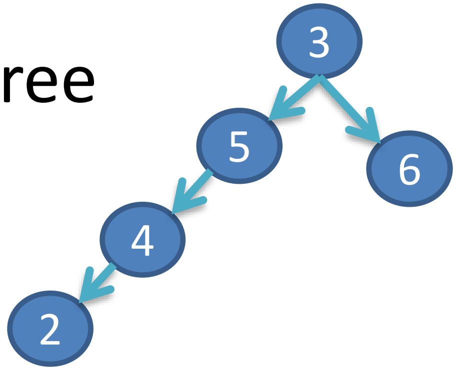


- If the path to v is the next shortest path, the path to v' must be at least as long. Therefore, any path through v' to v cannot be shorter!

Minimum Spanning trees

What are trees in undirected graphs?

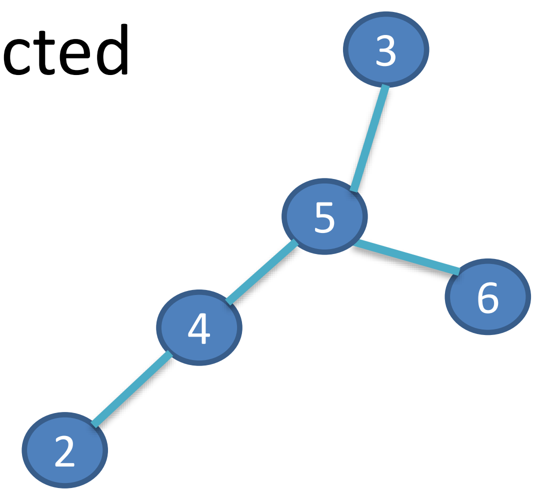
- We are familiar with a rooted directed tree, such as:
  - Binary search tree
  - Heap



- They look like:
  - You know which one the root is
  - Who the parent is
  - Who the children are

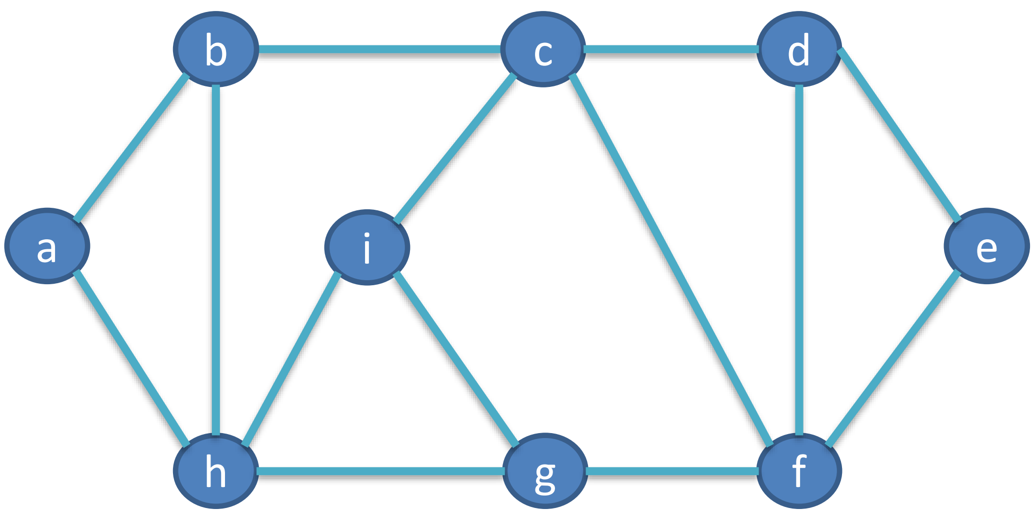
What are trees in undirected graphs?

- However, in undirected graphs, there is another definition of trees
- Tree
  - A undirected graph (V, E), where E is the set of undirected edges
  - All vertices are connected
  - $|E|=|V|-1$



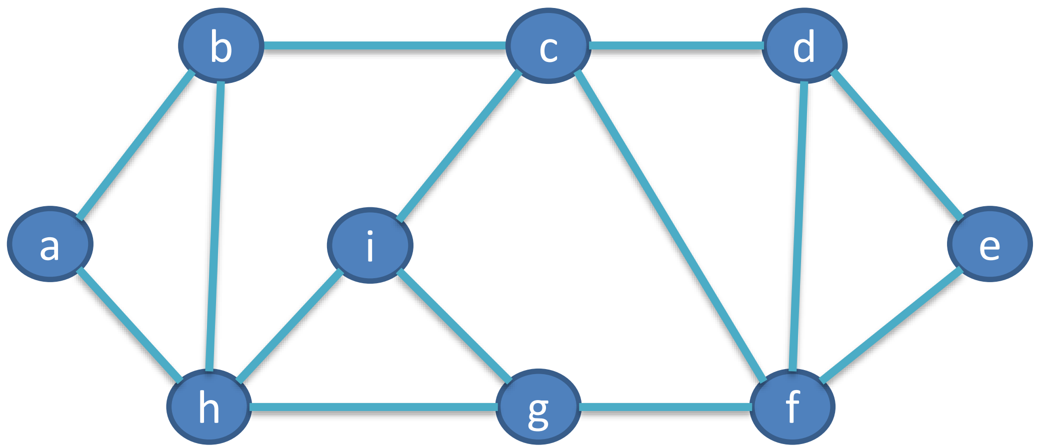
What is a spanning tree?

- Unless otherwise specified, spanning tree is about a **tree of a undirected graph**
- A spanning tree of graph G=(V, E) is:
  - A graph G'=(V', E'), where $V' = V, E' \subseteq E$
  - G' is a tree
- Given the following graph, are they spanning trees?



What is a spanning tree?

- Given the following graph, are they spanning trees?

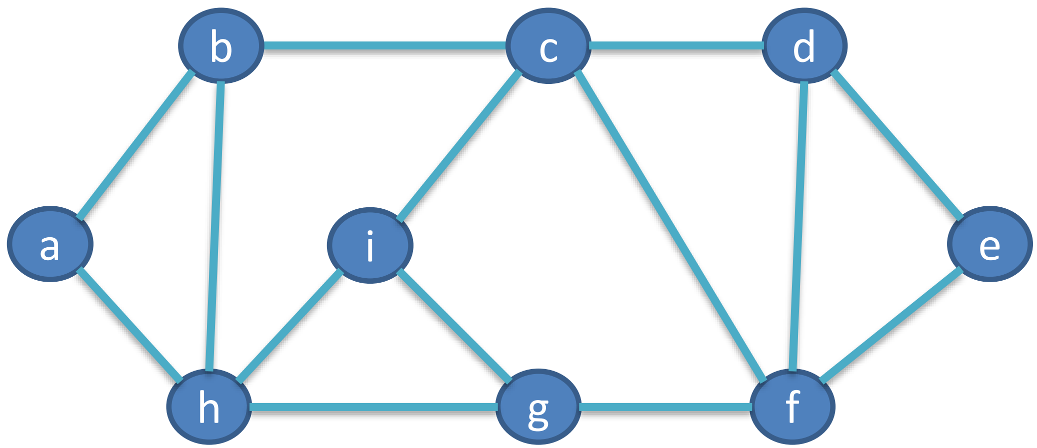


Is the red sub graph a tree? Yes

Is the red sub graph a spanning tree? No. $V' \neq V$

What is a spanning tree?

- Given the following graph, are they spanning trees?

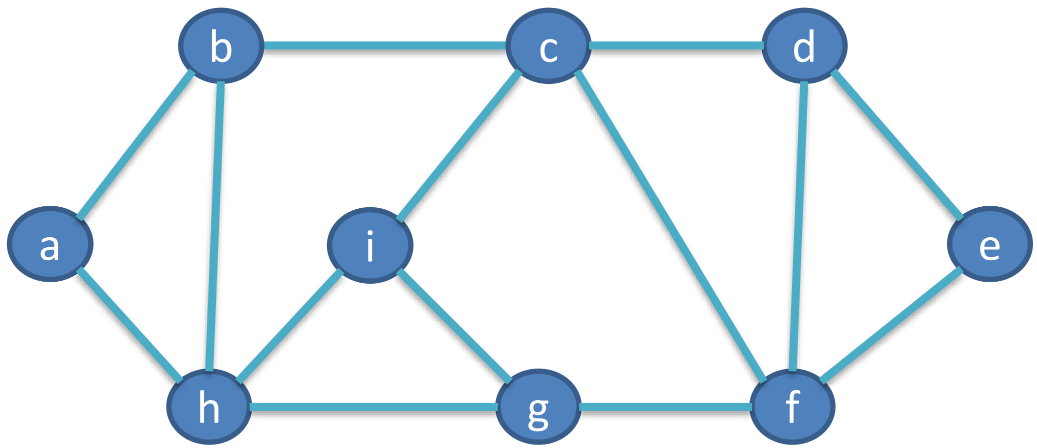


Is the red sub graph a tree? No. $|E| \neq |V'| - 1$

Is the red sub graph a spanning tree? No. It is not a tree.

What is a spanning tree?

- Given the following graph, are they spanning trees?

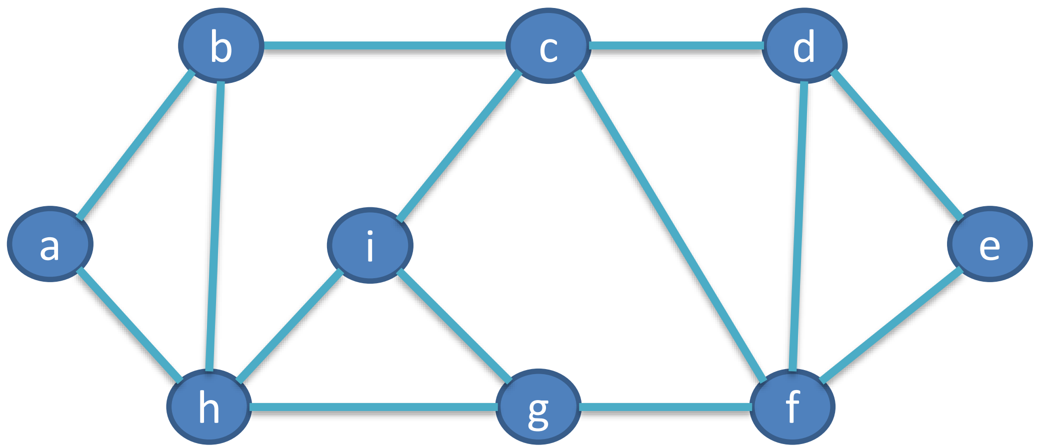


Is the red sub graph a tree? No. Vertices are not all connected

Is the red sub graph a spanning tree? No. It is not a tree.

What is a spanning tree?

- Given the following graph, are they spanning trees?

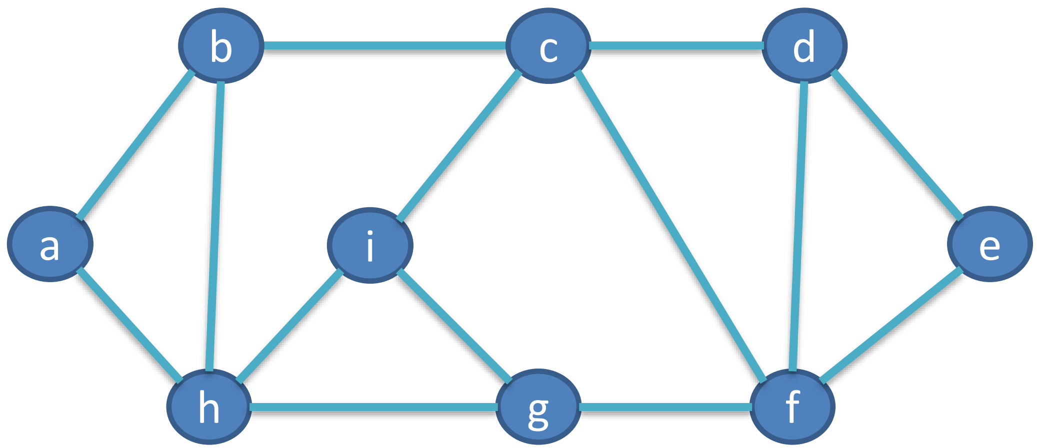


Is the red sub graph a tree? Yes

Is the red sub graph a spanning tree? Yes

What is a spanning tree?

- Given the following graph, are they spanning trees?



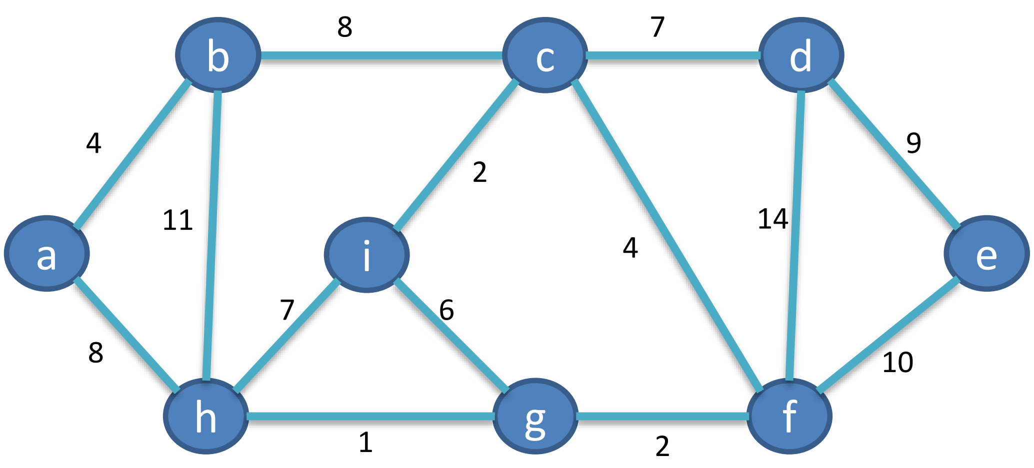
Is the red sub graph a tree? Yes

Is the red sub graph a spanning tree? Yes

There could be more than one spanning trees

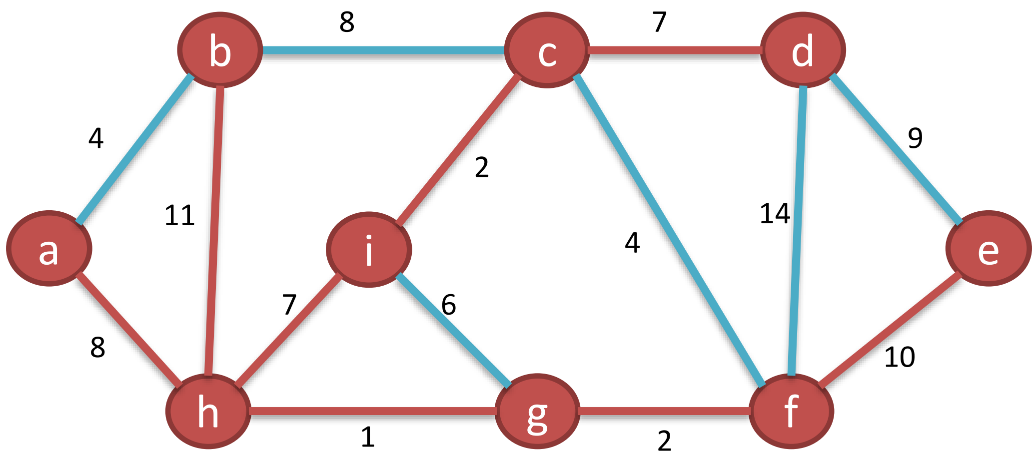
What is a minimum spanning tree?

- When edges are weighted
- A minimum spanning tree of G is:
  - A spanning tree of G
  - With a minimum sum of a edge weights sum among all spanning trees



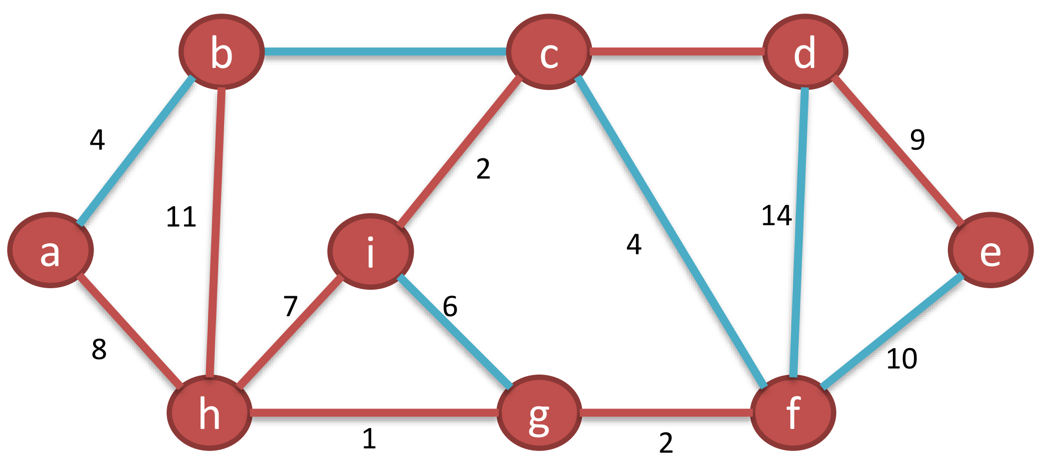
What is a minimum spanning tree?

- Spanning trees:



The weight sum of this spanning tree?

48

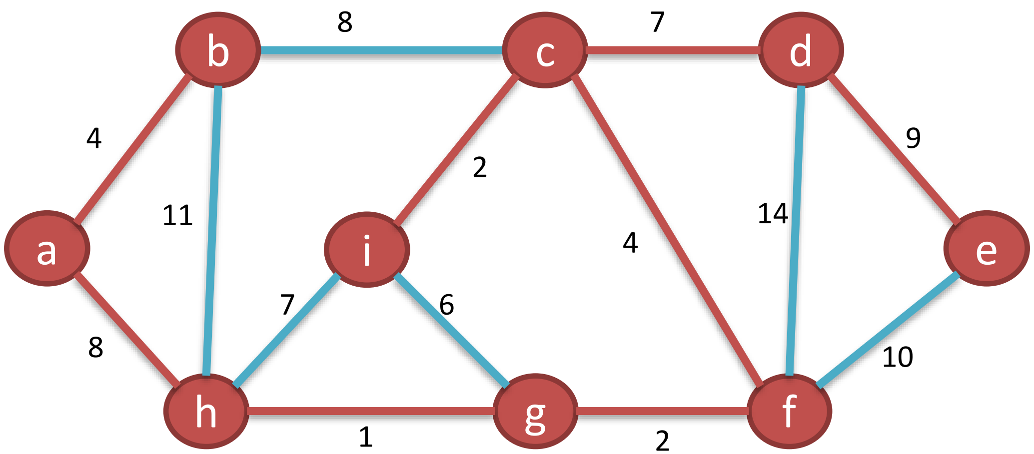


The weight sum of this spanning tree?

47

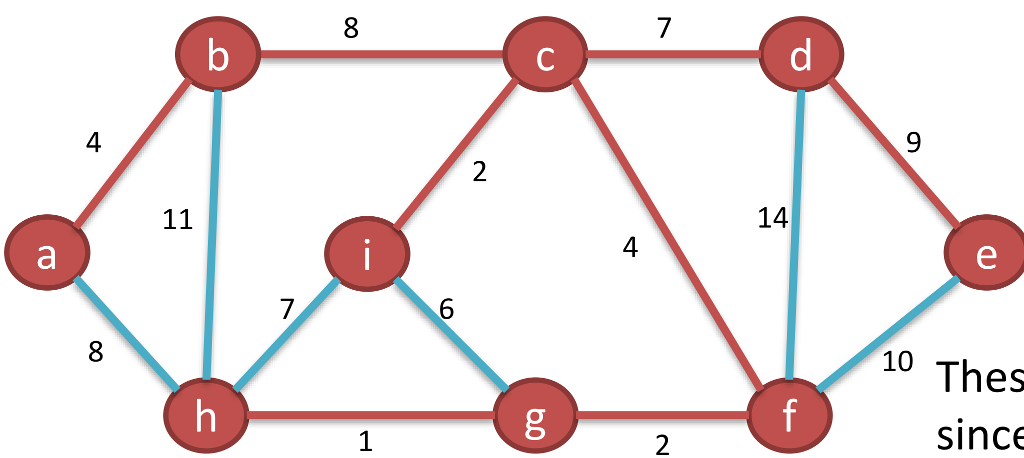
What is a minimum spanning tree?

- Spanning trees:



The weight sum of this spanning tree?

37



The weight sum of this spanning tree?

37

These 2 are minimum spanning trees since there is no spanning tree with a smaller total weight

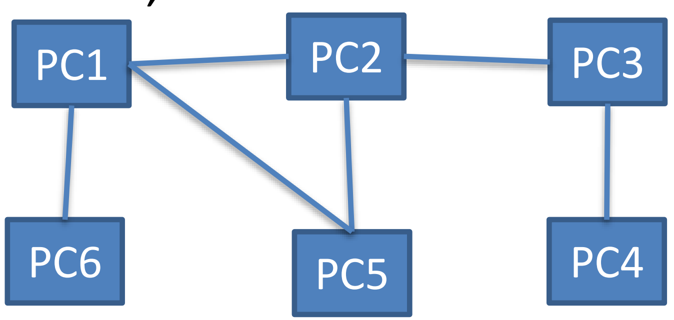
Minimum spanning tree problem

- **Minimum spanning tree (MST) problem**: given a undirected graph G, find a spanning tree with the minimum total edge weights.
- There are 2 classic greedy algorithms to solve this problem
  - Kruskal’s algorithm
  - Prim’s algorithm

Minimum spanning tree problem

- A application example of MST in computer network
  - What if PC3 wants to communicate with PC7?
  - By a routing protocol:
    - PC3 broadcasts this requirement to PC2, PC4
    - PC2 broadcasts it to PC1, PC5
    - PC1 broadcasts it to PC5, PC6

      …
    - PC5 broadcasts it to PC2



  - At least, a package is in a deadlock: PC2->PC1->PC5-PC2->PC1-> …………….

Find and use a MST can solve this problem, since there is no cycle

Minimum spanning tree problem

- The MST algorithm can be considered as choose |V|-1 edges from E to form a minimum spanning tree
- Both Kruskal's algorithm and Prim's algorithm proposed a greedy choice about how to choose a edge
- They have greedy choice property because of textbook **Corollary 23.2**

Kruskal's algorithm

- Input: G=(V, E)
- Output: a minimum spanning tree (V, A)
- **Kruskal** (G=(V, E))

```
Set each vertices in V as a vertex set;
Set A as an empty set;
while (there are more than 1 vertex set){
    Add a smallest edge (u, v) to A where u, v are not in the same set;
    Merge the set contains u with the set contains v;
}
```

- In most of the implementations, edges are sorted on weights at the beginning, and then scan them once
- The running time = O(|E|lg|V|)

Kruskal's algorithm

Kruskal (G=(V, E))

Set each vertices in V as a vertex set;

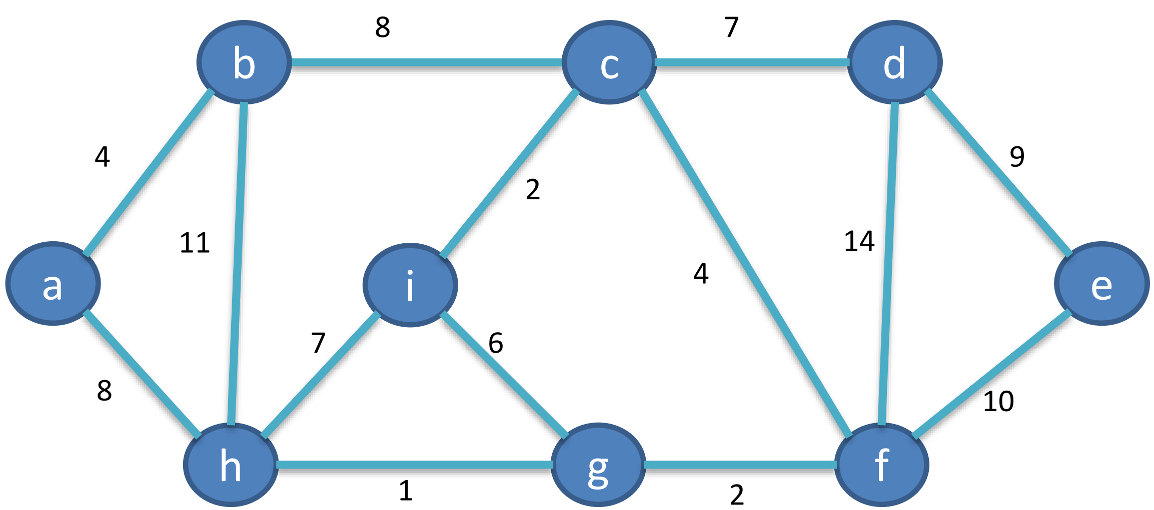
Set A as an empty set;

while (there are more than 1 vertex set){

Add a smallest edge (u, v) to A where u, v are not in the same set;

Merge the set contains u with the set contains v;

}



Total weight: 37

Edge	Weight
(h, g) ✔	1
(i, c) ✔	2
(g, f) ✔	2
(c, f) ✔	4
(a, b) ✔	4
(i, g)	6
(h, i)	7
(c, d) ✔	7
(a, h) ✔	8
(b, c)	8
(d, e) ✔	9
(e, f)	10
(b, h)	11
(d, f)	14

Prim's algorithm

- Input: G=(V, E)
- Output: a minimum spanning tree (V, A)
- **Prim**(G=(V, E))

  Set A as an empty set;

  Choose a vertex as the current MST;

  while (not all the vertices are added into the MST){

    Choose the smallest edge crossing current MST vertices and other vertices, and put it to MST ;

  }
- The running time = $O(|E|\lg|V|)$

Prim’s algorithm

Prim(G=(V, E))

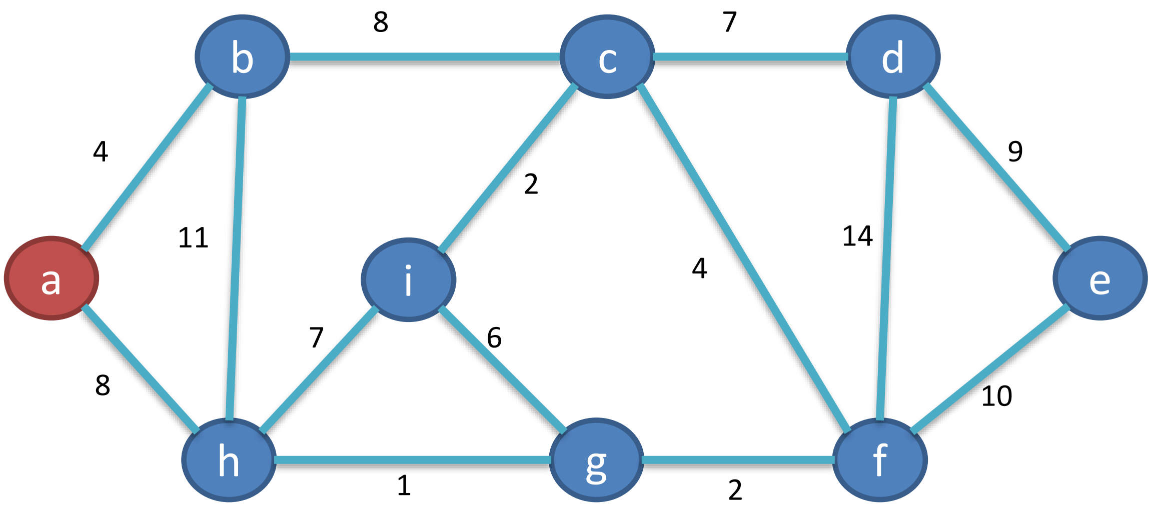
  Set A as an empty set;

  Choose a vertex as the current MST;

  while (not all the vertices are added into the MST){

    Choose the smallest safe edge, and put it to MST;

  }



Total weight: 37

Summary

- Prim’s algorithm and Kruskal’s algorithm have the same asymptotically running time