13
Graphs
Chapter 14

Much of the slides of this lecture are courtesy of Prof.Y.Sahillioğlu and Prof.T.Can of METU CENG.

Definitions

A graph G = (V, E) consists of

- A set of vertices, V, and
- A set of edges, E, where each edge is a pair (v, w) s.t. v, w $\in$ V

Vertices are sometimes called nodes, edges are sometimes called arcs.

If the edge pair is ordered (i.e., (v,w) ≠ (w,v)) then the graph is called a directed graph (also called digraphs).

We also call a normal graph (which is not a directed graph) an undirected graph.

Definitions

Two vertices of a graph are ***adjacent*** if they are joined by an edge.

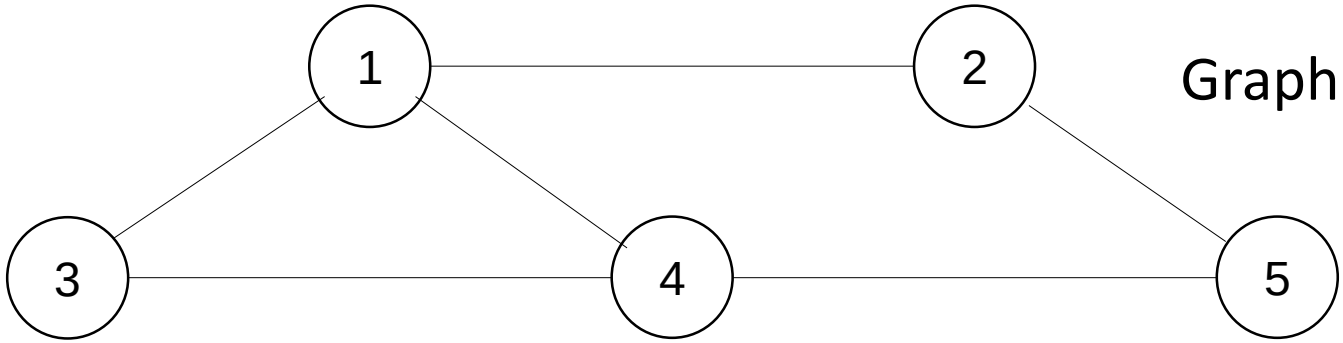
A **path** between two vertices is a sequence of edges that begins at one vertex and ends at another (i.e., w_1, w_2, …, w_N is a path if (w_i, w_{i+1}) $\in$ E for $1 \leq i \leq N-1$).

A **simple path** passes through a vertex only once.

A **cycle** is a path that begins and ends at the same vertex.

A **simple cycle** is a cycle that does not pass through vertices other than the starting vertex more than once.

Definitions



Graph G (undirected)

The graph G= (V,E) has 5 vertices and 6 edges:

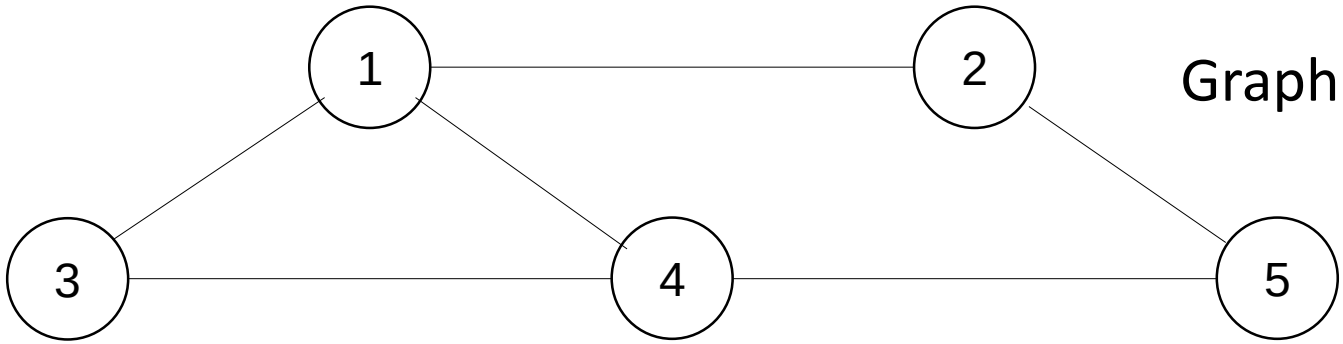
V = {1,2,3,4,5}

E = { (1,2),(1,3),(1,4),(2,5),(3,4),(4,5)}

If it were to be a directed graph, then the set of edges would be:

E ={(1,2),(1,3),(1,4),(2,5),(3,4),(4,5),(2,1),(3,1),(4,1),(5,2),(4,3),(5,4)}

Definitions



Graph G (undirected)

Adjacency

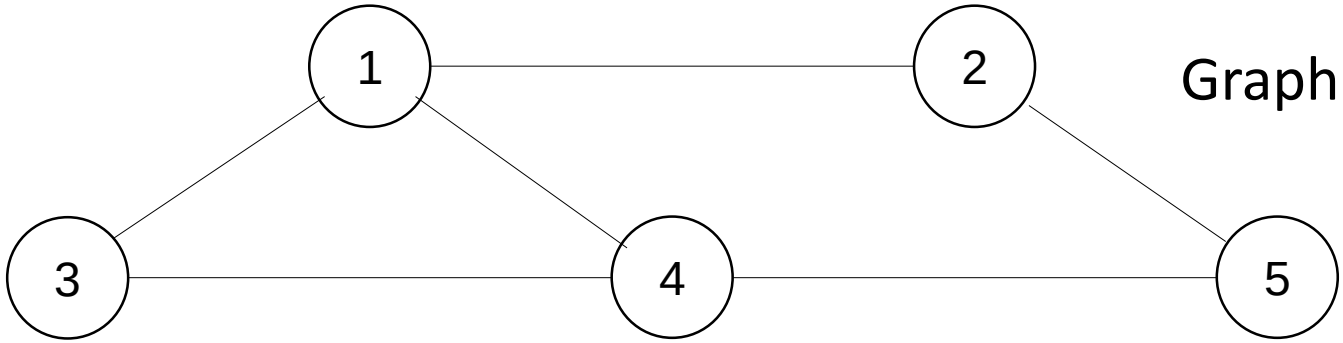
Nodes 1 and 2 are adjacent, i.e., 1 is adjacent to 2, and 2 is adjacent to 1.

Path

1, 2, and 5 forms a simple path

1, 3, 4, 1, 2, and 5 forms a path as well, however, it is not a simple path.

Definitions



Graph G (undirected)

Cycle

1, 3, 4, and 1 forms a simple cycle.

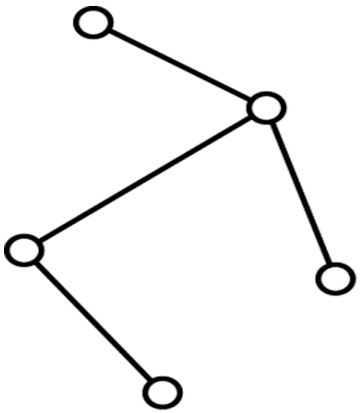
1, 3, 4, 1, 4, and 1 forms a cycle too, however, it is not a simple cycle.

Definitions

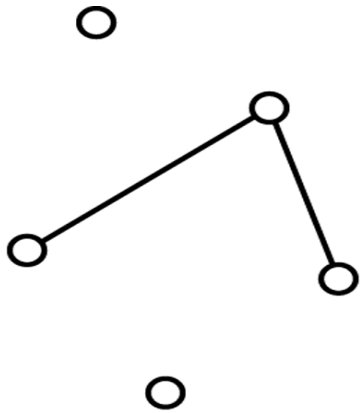
A **connected graph** has a path between each pair of its distinct vertices.

A **complete graph** has an edge between each pair of distinct vertices.

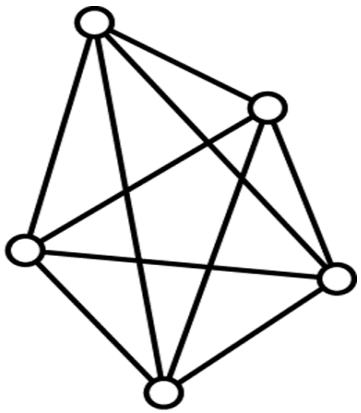
A **complete graph** is also a connected graph. But a **connected graph** may not be a complete graph.



(a) connected



(b) disconnected



(c) complete

Directed Graphs

If the edge pair is ordered then the graph is called a **directed graph** (also called digraphs).

Each edge in a directed graph has a direction, and they are called **directed edge**s.

Definitions given for undirected graphs apply also to directed graphs, with changes that account for direction.

Vertex w is **adjacent to** v iff (v,w) $\in$ E.

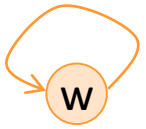
- i.e. there is a direct edge from v to w
- w is successor of v
- v is predecessor of w

Directed Graphs

A **directed path** between two vertices is a sequence of directed edges that begins at one vertex and ends at another (i.e., w_1, w_2, …, w_N is a path if $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq N-1$).

A **cycle** in a directed graph is a path of length at least 1 such that $w_1 = w_N$. This cycle is simple if the path is simple.

Examples: {(w,w)} is a simple cycle and also a path. {(w,w), (w,w)} is a cycle, too but not a simple cycle.

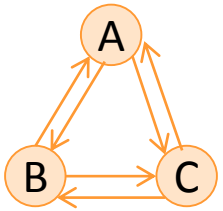


A **directed acyclic graph** (*DAG*) is a type of directed graph having no cycles.

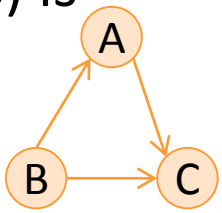
Directed Graphs

An undirected graph is **connected** if there is a path from every vertex to every other vertex.

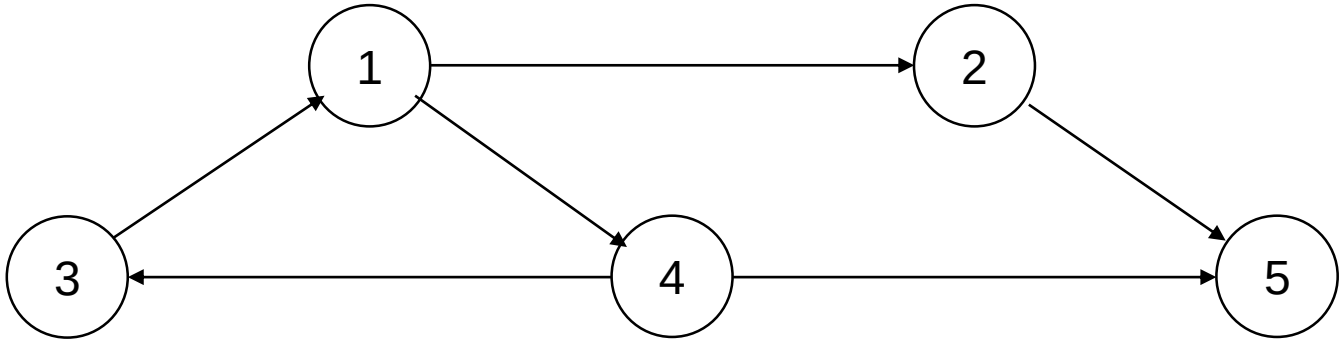
A directed graph with this property is called **strongly connected**.



If a directed graph is not strongly connected, but the underlying graph (by disregarding the directions of the arcs/edges) is connected then the graph is **weakly connected**.



Directed Graphs



The graph G= (V,E) has 5 vertices and 6 edges:

V = {1, 2, 3, 4, 5} E = { (1,2), (1,4), (2,5), (4,5), (3,1), (4,3) }

Adjacency: 2 is adjacent to 1, but 1 is not adjacent to 2.

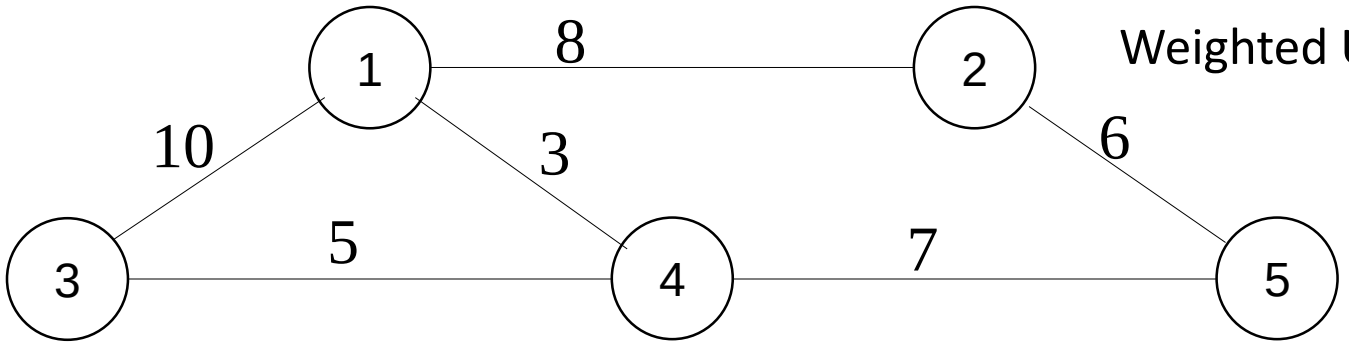
Confused about adjancency? To decide whether two vertices are adjacent, this question can be asked: "Can we go from 2 to 1 in one step?" If the answer is "No." then 1 is not adjacent to 2, it is not reachable from 2 with a single step.

Path: 1, 2, and 5 forms a directed path.

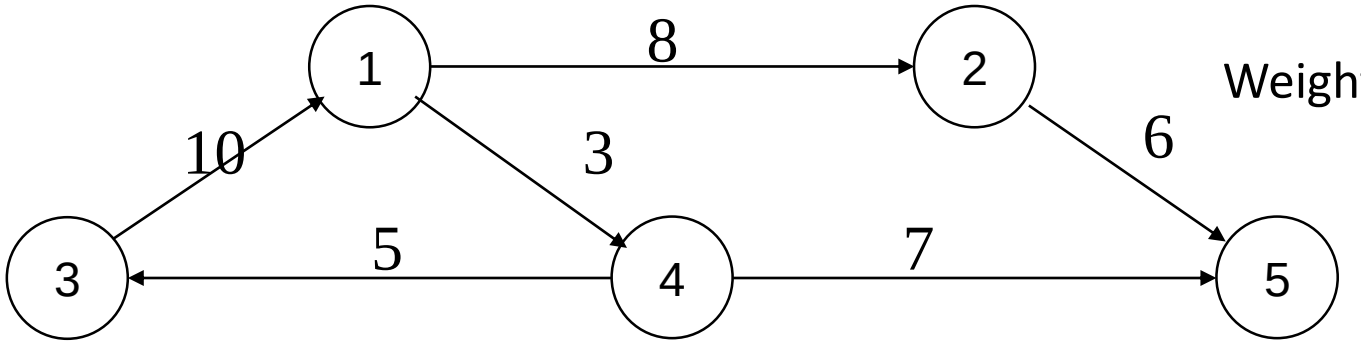
Cycle: 1, 4, 3, and 1 forms a directed cycle.

Weighted Graph

If the edges of a graph are associated with numerical values, then that graph is called a **weighted graph**.



Weighted Undirected Graph



Weighted Directed Graph

Data Structures for Graphs

Four data structures to represent a graph:

- Edge List (unordered list of all edges)
- Adjacency List (for each vertex, a separate list of edges)
- Adjacency Map (for each vertex, a separate map of edges indexed with adjacent vertex)
- Adjancecy Matrix (an nxn matrix for a graph with n vertices)

Adjacency Matrix

An adjacency matrix for a graph with n vertices are labeled as 0, 1, ... , n-1 is an n-by-n matrix such that matrix[i][j] has the value 1 (true) if there is an edge from vertex i to vertex j, and 0 (false) otherwise.

When the graph is weighted, matrix[i][j] hold the weight of edge from vertex i to vertex j. matrix[i][j] hold the value ∞ when there is no edge from vertex i to vertex j.

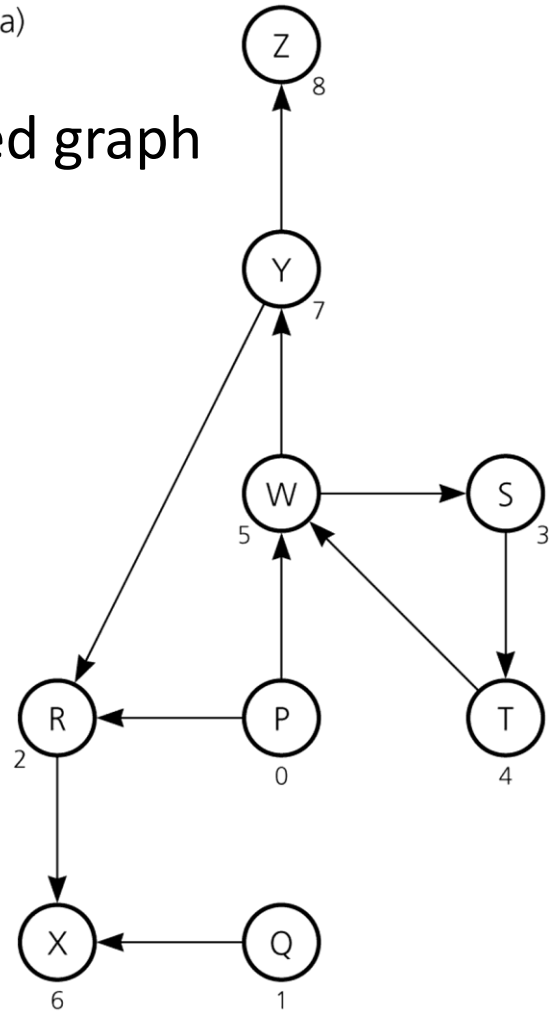
Adjacency matrix for an undirected graph is symmetrical, i.e., matrix[i][j] is equal to matrix[j][i].

Space requirement is $O(|V|^2)$.

Acceptable if the graph is dense.

Adjacency Matrix

A directed graph



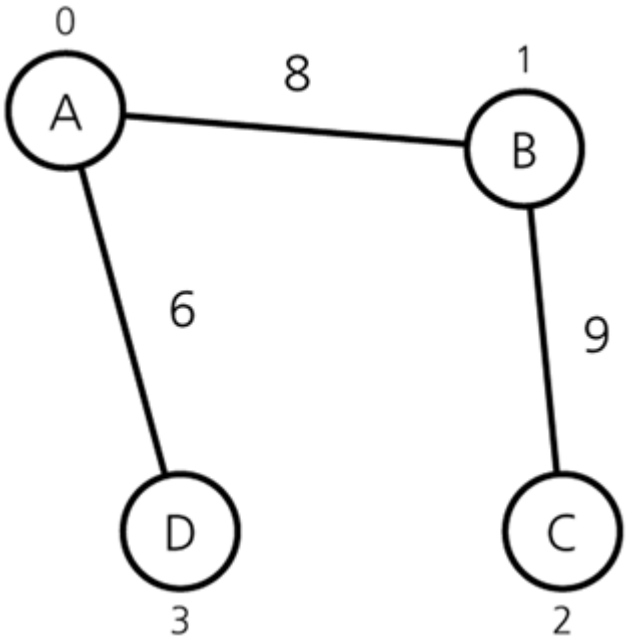
(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Its adjacency matrix

Adjacency Matrix

Undirected Weighted Graph



Adjacency Matrix

		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Adjacency List

An adjacency list for a graph with n vertices numbered 0, 1, ... , n-1 consists of n objects of sequence types (such as lists, linked lists, etc.).

The i^{th} item in that sequence stores a node for vertex j if and only if the graph contains an edge from vertex i to vertex j.

Adjacency list is a better solution if the graph is sparse.

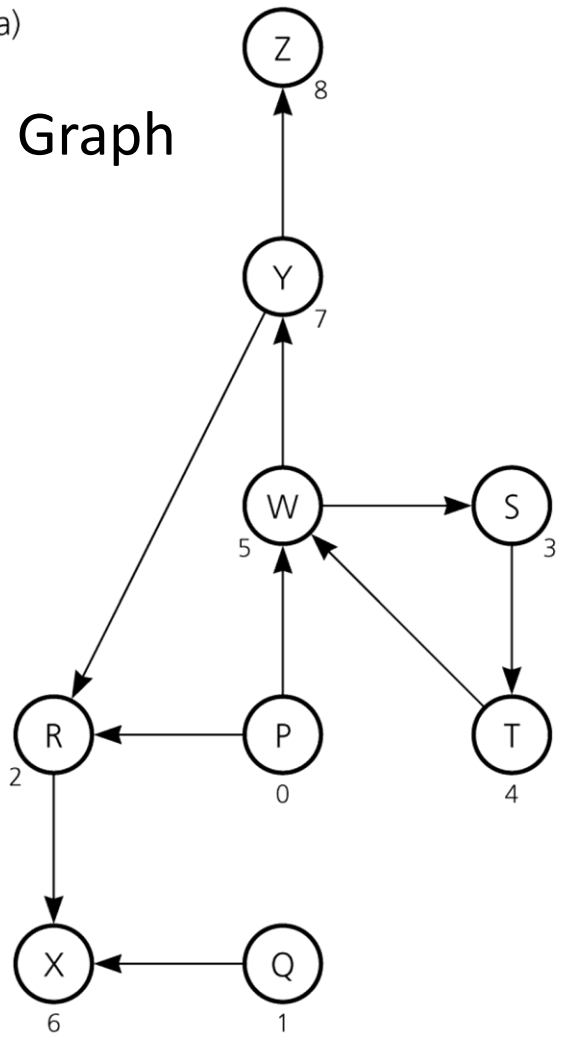
Space requirement is O(|E| + |V|), which is linear in the size of the graph.

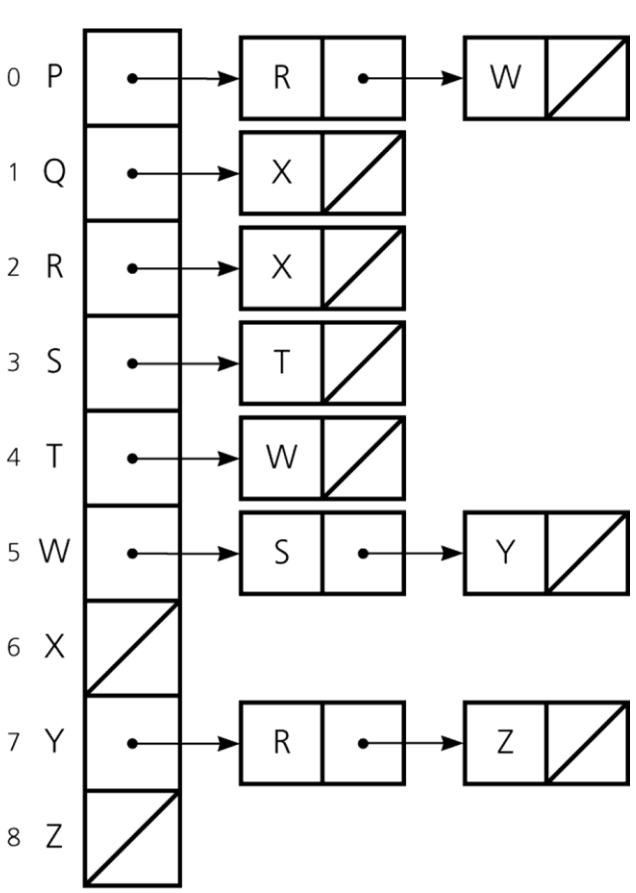
In an undirected graph each edge (v,w) appears in two lists.

Space requirement is doubled.

Adjacency List

Directed Graph

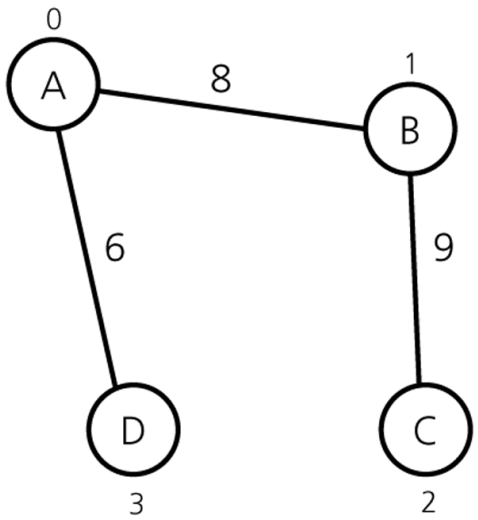




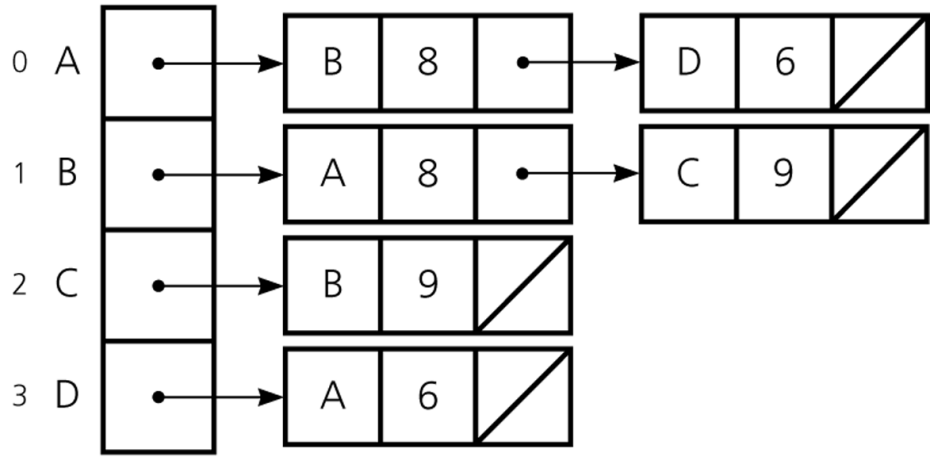
Adjacency List

Adjacency List

Undirected Weighted Graph



Adjacency List



Adjacency Matrix vs. Adjacency List

Adjacency matrix could help better when determining whether there is an edge from vertex i to vertex j.

Adjacency list could be a more efficient choice when finding all vertices adjacent to a given vertex i.

An adjacency list often requires less space than an adjacency matrix.

- Adjacency Matrix: Space requirement is $O(|V|^2)$
- Adjacency List : Space requirement is $O(|E| + |V|)$, which is linear in the size of the graph.

Adjacency matrix is better if the graph is dense (too many edges).

Adjacency list is better if the graph is sparse (few edges).

Graph Traversals

A **graph traversal algorithm** starts from a vertex v, and continues over all of the vertices that are reachable from the vertex v.

A graph traversal algorithm can visit all vertices if and only if the graph is connected.

A **connected component** is the subset of vertices visited during a traversal algorithm that begins at a given vertex.

A graph traversal algorithm must mark each vertex during a visit and must never visit a vertex more than once. Thus, if a graph contains a cycle, the graph traversal algorithm can avoid potential infinite loops.

Graph Traversals

We cover the following graph-traversal algorithms:

- Depth-First Search (DFS) (a.k.a., Depth-first Traversal)
- Breadth-First Search (BFS) (a.k.a., Breadth-first Traversal)

Depth-first Search

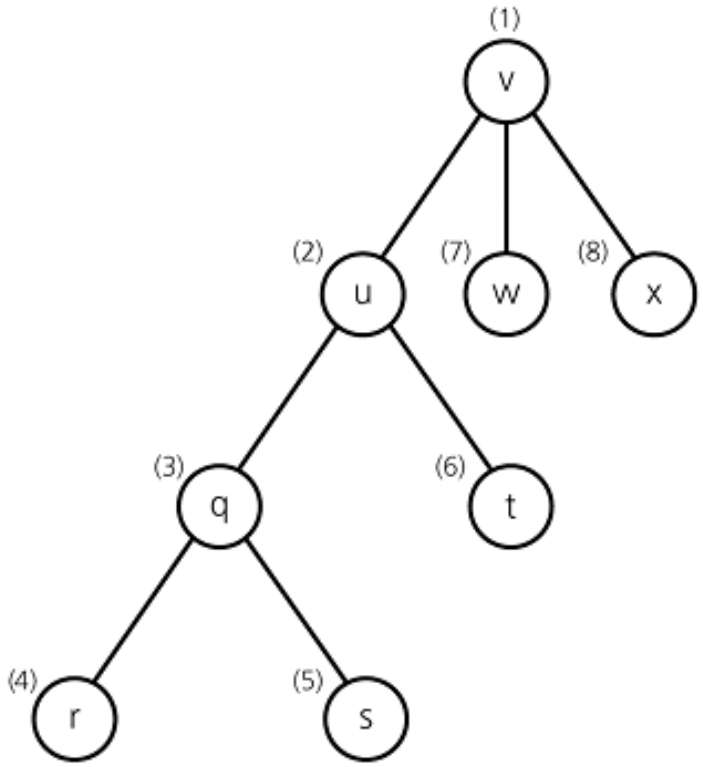
For a given vertex v, the depth-first search algorithm proceeds along a path from v as deeply into the graph as possible before backing up.

That is, after visiting a vertex v, the depth-first search algorithm visits (if possible) an unvisited adjacent vertex to vertex v.

The depth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.

We may visit the vertices adjacent to v in sorted order.

Depth-first Search



A depth-first search of the graph starting from vertex v.

Visit a vertex, then visit a vertex adjacent to that vertex.

If there is no unvisited vertex adjacent to visited vertex, back up to the previous step.

Depth-first Search

Pseudo Code for Depth-first Search

Let G be a graph and u be a vertex of G.

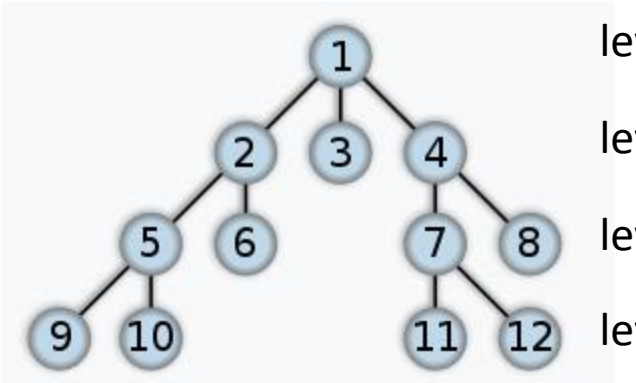
DFS(G, u)

- For each node v adjacent to u:
  - If v is not visited then:
    - Mark v as visited
    - DFS(G, v)

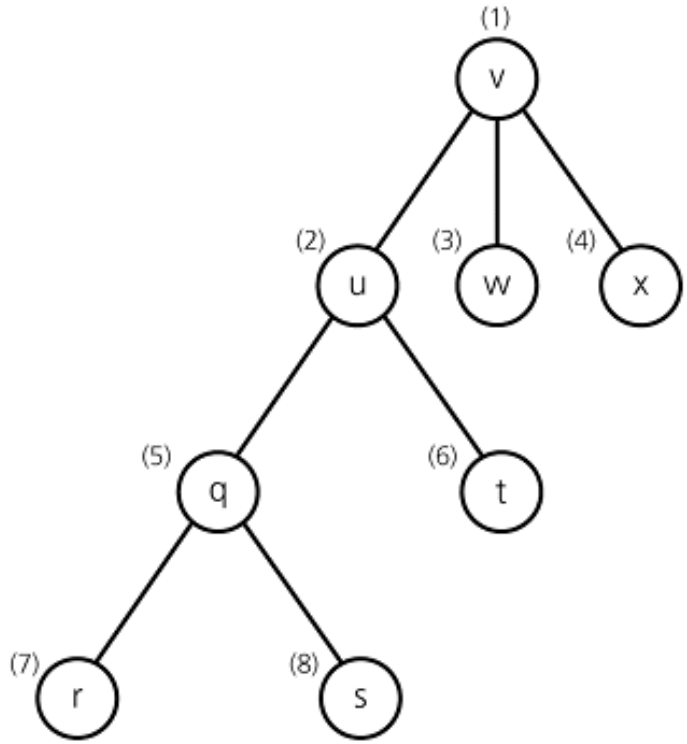
Breadth-first Search

After visiting a given vertex v (level 0), the breadth-first search algorithm visits every vertex adjacent to v (level 1), then other vertices that are adjacent to level 1 vertices are visited (level 2), until when no new vertices are found in a level.

The breadth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.



Breadth-first Search



A breadth-first search of the graph starting from vertex v.

Visit a vertex, then visit all vertices adjacent to that vertex.

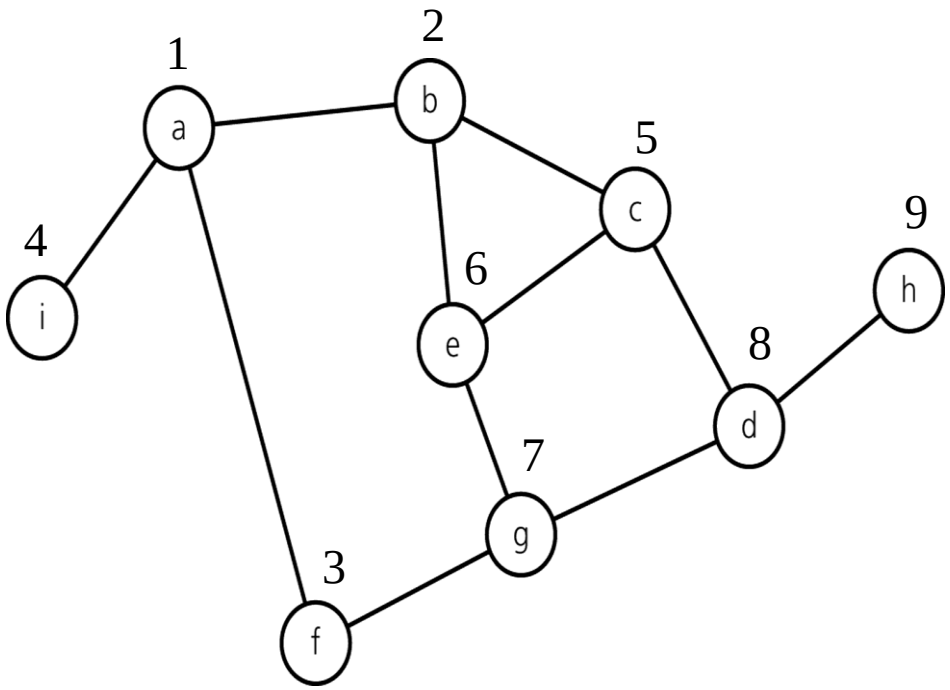
Breadth-first Search

Pseudo Code for Breadth-first Search

Let G be a graph and u be a vertex of G.

```
BFS(G,u):
Initialize and empty queue Q, mark u as visited
Q.enqueue(u)
While Q is not empty:
    x = Q.dequeue()
    for each vertex y adjacent to x:
        if x is not visited:
            Mark x as visited
            Q.enqueue(x)
```

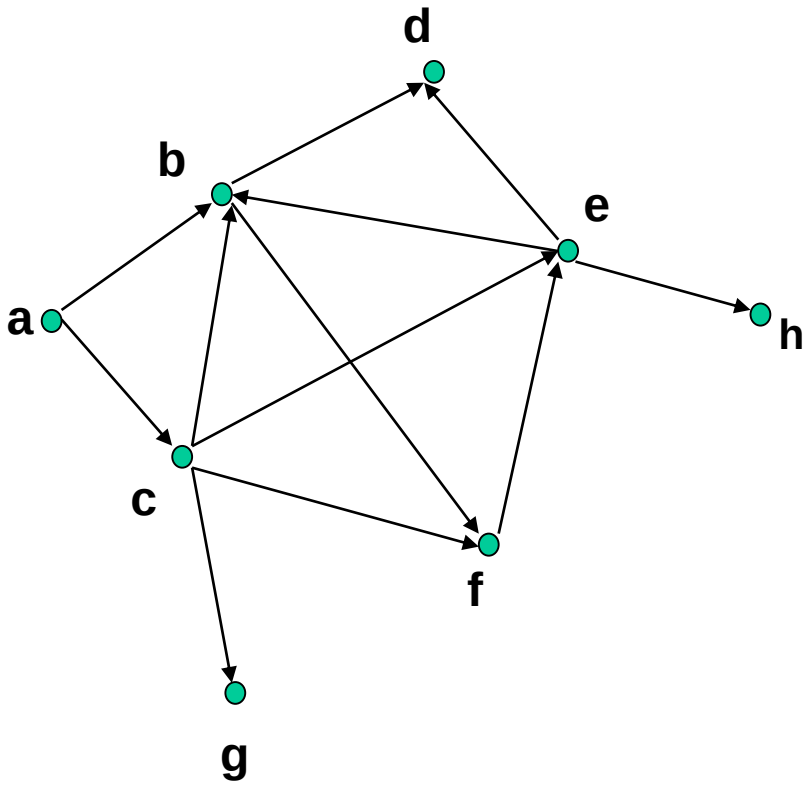
Breadth-first Search



Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

While Q is not empty:
    x = Q.dequeue()
    for each vertex y adjacent to x:
        if x is not visited:
            Mark x as visited
            Q.enqueue(x)

An Exercise



a) Give the sequence of vertices when they are traversed starting from the vertex **a** using the depth-first search algorithm.

b) Give the sequence of vertices when they are traversed starting from the vertex **a** using the breadth-first search algorithm.

Graph Algorithms

Shortest Path Algorithms

- Unweighted shortest paths
- Weighted shortest paths (Dijkstra's Algorithm)

Topological sorting

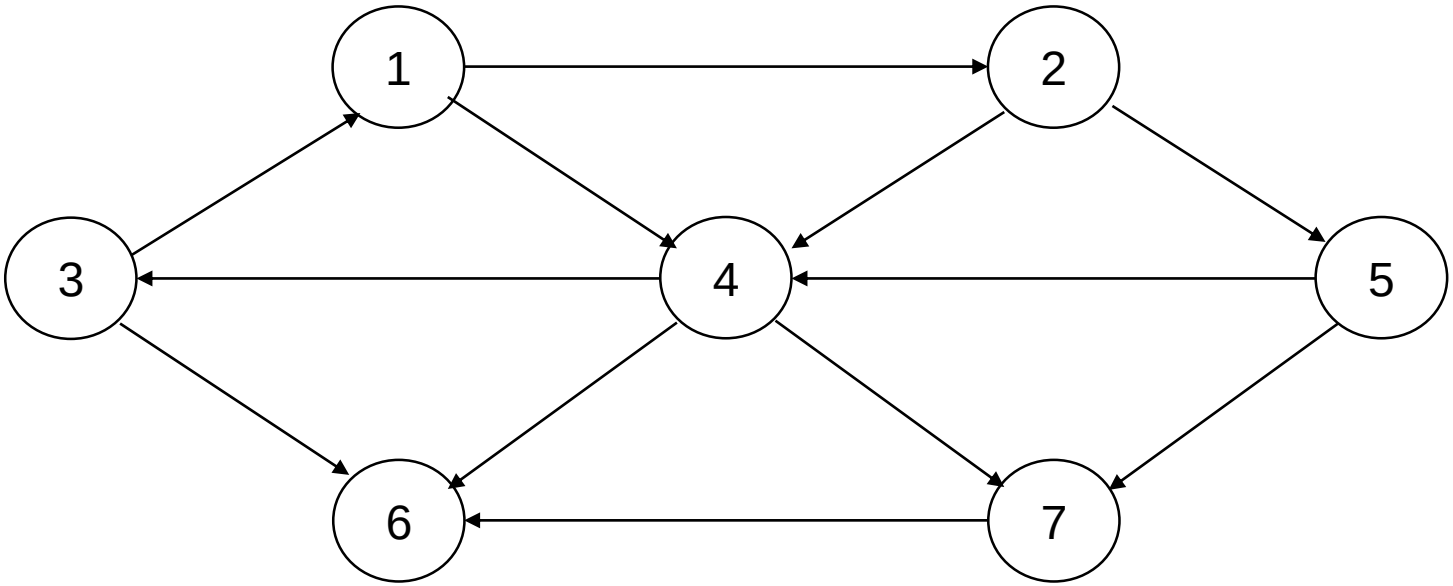
Network Flow Problems

Minimum Spanning Tree

Depth-first search Applications

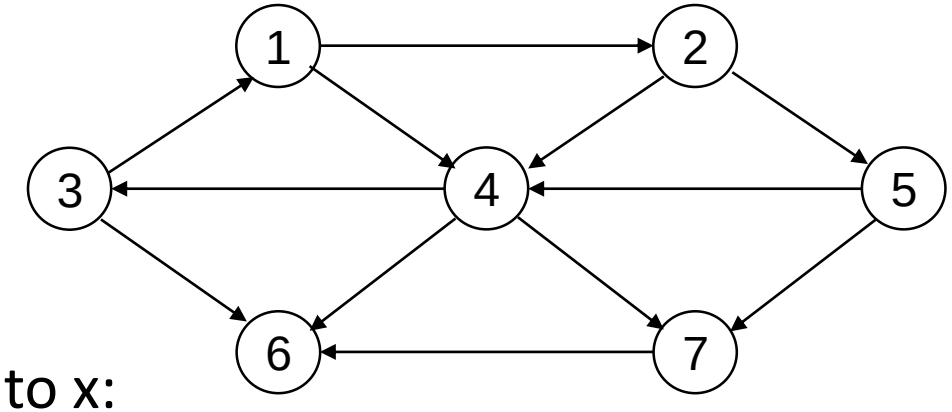
Unweighted Shortest Path

Find the shortest path (measured by number of edges) from a designated vertex S to every vertex.



Unweighted Shortest Path

Let G be a graph and u be the node to start from. Let D_i be the distance between node i and node u. $D_u = 0$, and Di = ∞ where i ≠ u. Let Q be a queue.



Q.enqueue(u)

While Q is not empty:

    x = Q.dequeue()

    For each node y adjacent to x:

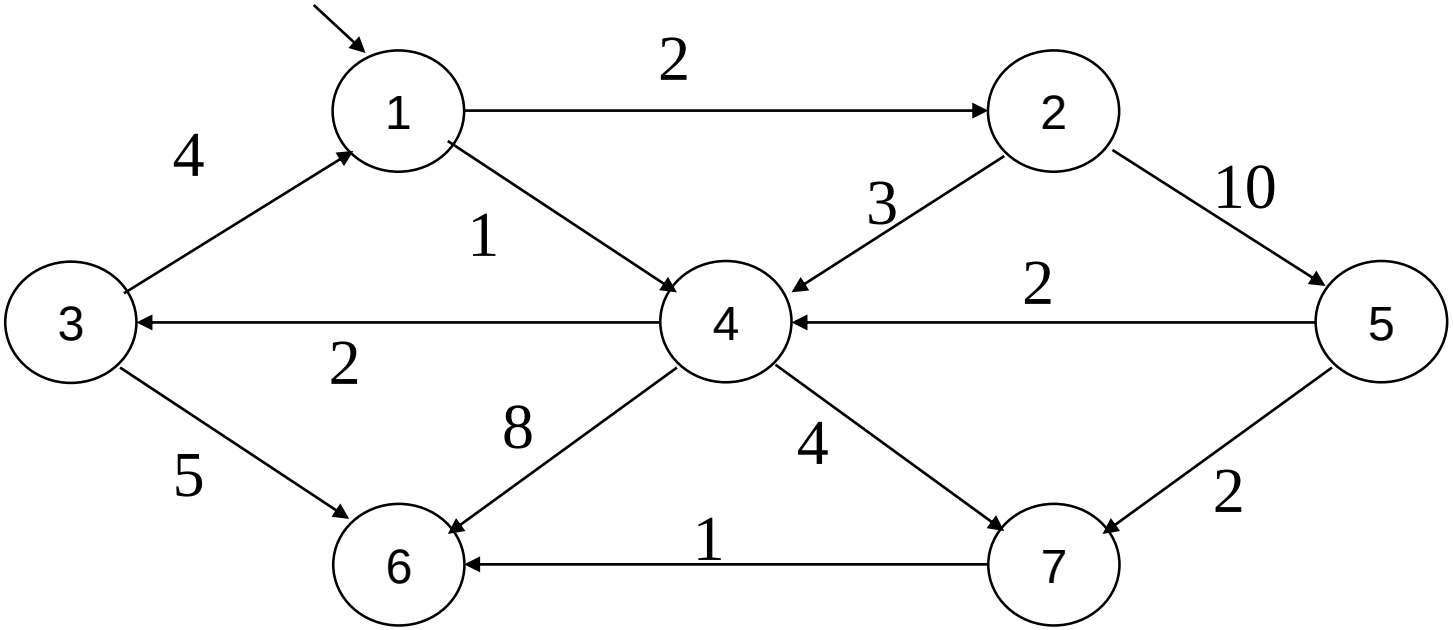
        If D_y = ∞ then:

            $D_y = D_x + 1$

            Q.enqueue(y)

Weighted Shortest Path

Find the shortest path (measured by total cost) from a designated vertex S to every vertex. All edge costs are nonnegative.



Weighted Shortest Path

The method used to solve this problem is known as Dijkstra's algorithm.

An example of a greedy algorithm (i.e., use the local optimum at each step)

Solution is similar to the solution of unweighted shortest path problem.

Weighted Shortest Path

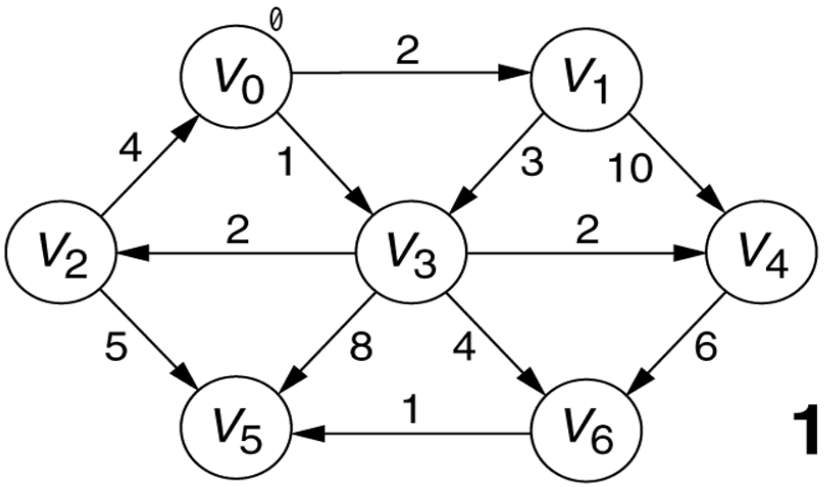
Algorithm ShortestPath(G, s):

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

Output: The length of a shortest path from s to v for each vertex v of G.

Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
  {pull a new vertex u into the cloud}
  u = value returned by Q.remove_min()
  for each vertex v adjacent to u such that v is in Q **do**
    {perform the ***relaxation*** procedure on edge (u, v)}
    if $D[u] + w(u, v) < D[v]$ **then**
      $D[v] = D[u] + w(u, v)$
      Change to $D[v]$ the key of vertex v in Q.
return the label $D[v]$ of each vertex v

Weighted Shortest Path



Priority Queue Q

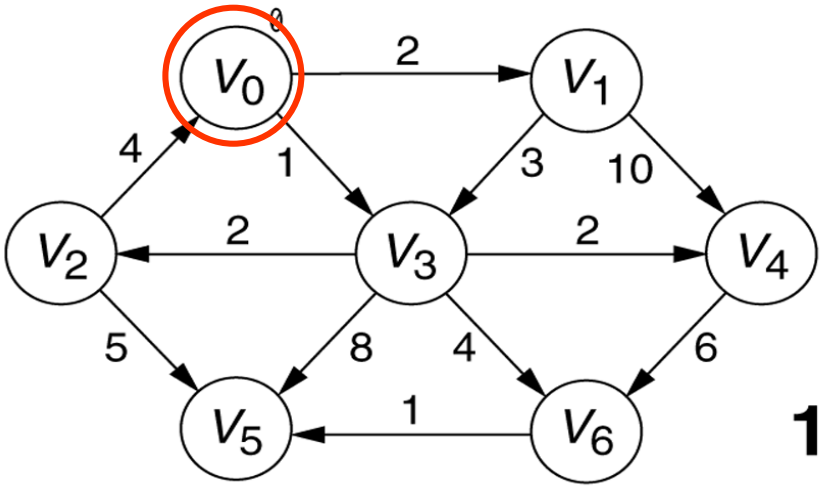
V0	V1	V2	V3	V4	V5	V6

Initial State

Distance Vector D

V0	V1	V2	V3	V4	V5	V6
0	Inf	Inf	Inf	Inf	Inf	Inf

Weighted Shortest Path



Shortest Distance: 0

Priority Queue Q

V0	V1	V2	V3	V4	V5	V6

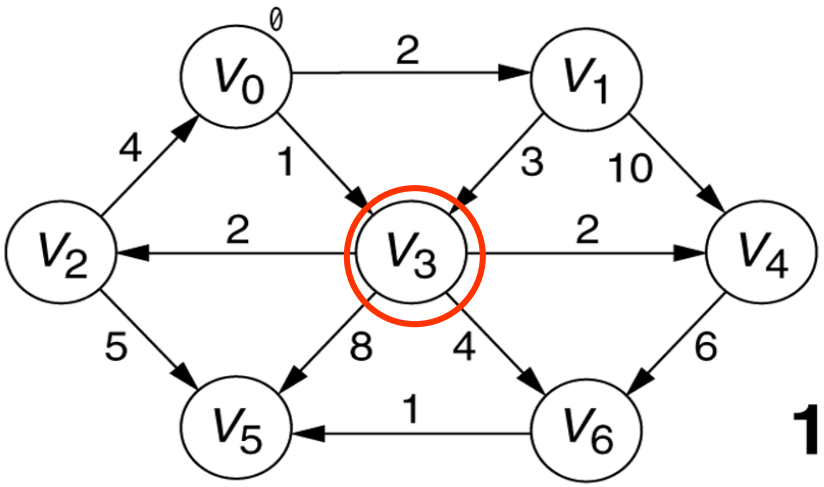
Adjacent and Non-dropped Nodes

V1	V3			

Distance Vector D

V0	V1	V2	V3	V4	V5	V6
0	2	Inf	1	Inf	Inf	Inf

Weighted Shortest Path



Shortest Distance

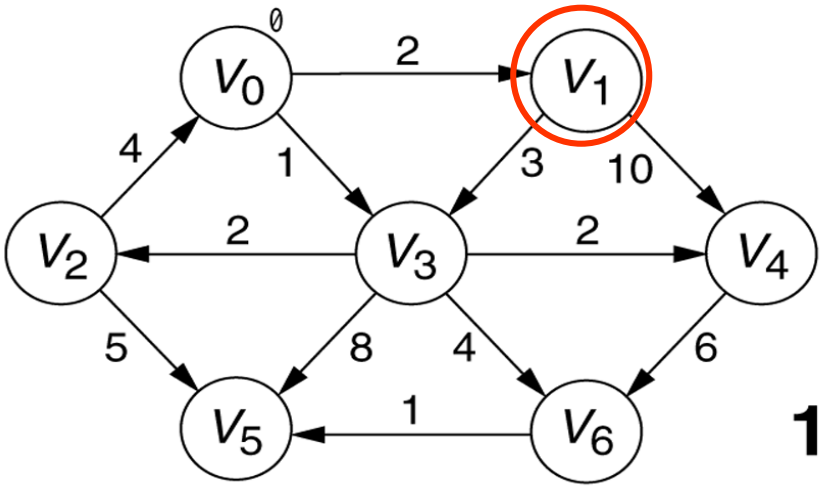
		0			1			
Priority Queue	Q	~~V0~~	V1	V2	~~V3~~	V4	V5	V6

Adjacent and Non-dropped Nodes

V2	V4	V5	V6	

	V0	V1	V2	V3	V4	V5	V6
Distance Vector D	0	2	3	1	3	9	5

Weighted Shortest Path



Shortest Distance

	0	2		1			
Priority Queue Q	~~V0~~	~~V1~~	V2	~~V3~~	V4	V5	V6

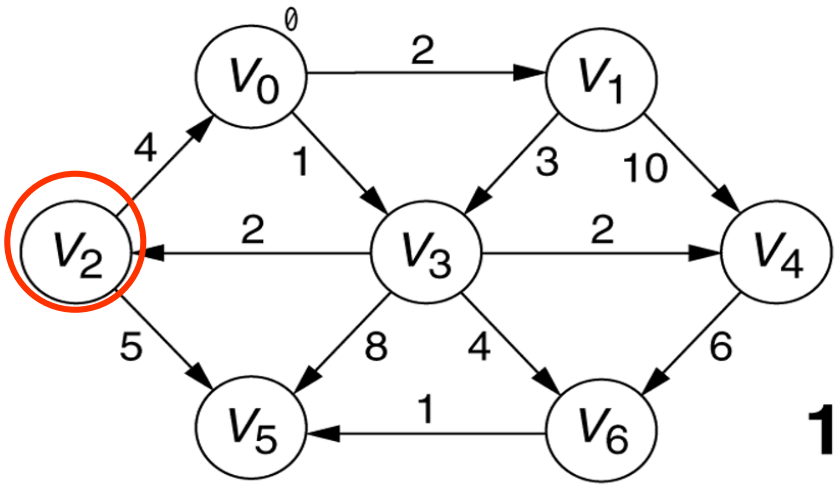
Adjacent and Non-dropped Nodes

V4				

	V0	V1	V2	V3	V4	V5	V6
Distance Vector D	0	2	3	1	3	9	5

No update!

Weighted Shortest Path



Shortest Distance

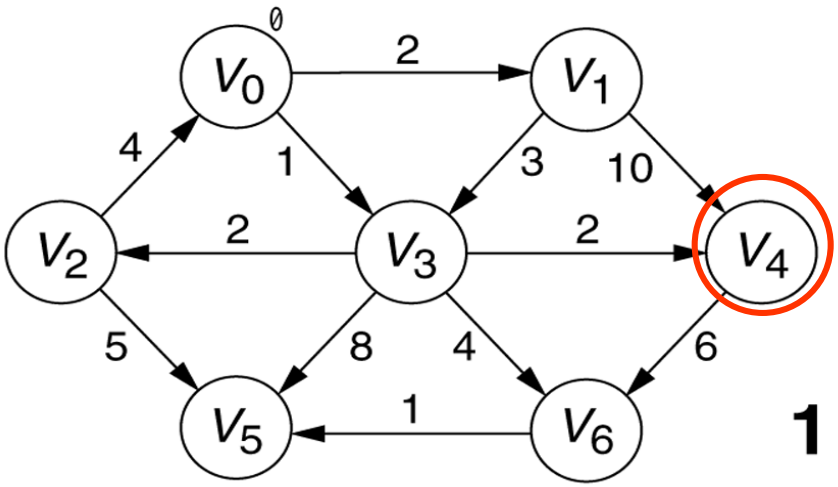
	0	2	3	1			
Priority Queue Q	~~V0~~	~~V1~~	~~V2~~	~~V3~~	V4	V5	V6

Adjacent and Non-dropped Nodes

V5				

	V0	V1	V2	V3	V4	V5	V6
Distance Vector D	0	2	3	1	3	8	5

Weighted Shortest Path



Shortest Distance

	0	2	3	1	3		
Priority Queue Q	~~V0~~	~~V1~~	~~V2~~	~~V3~~	~~V4~~	V5	V6

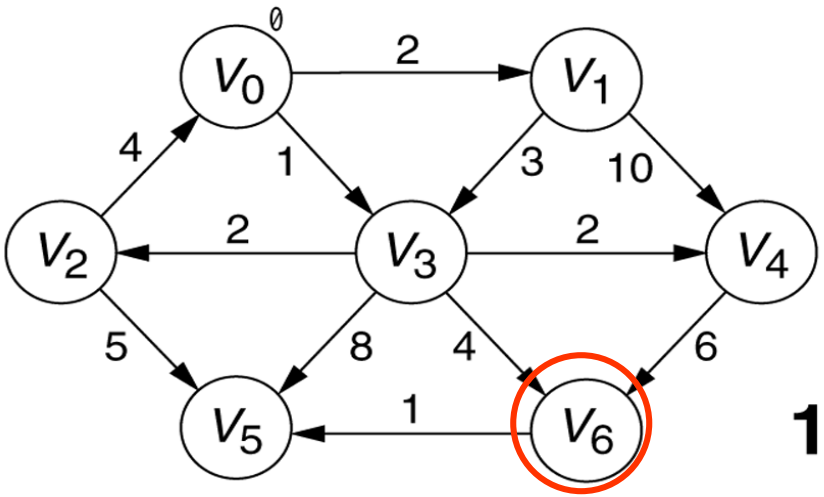
Adjacent and Non-dropped Nodes

V6				

	V0	V1	V2	V3	V4	V5	V6
Distance Vector D	0	2	3	1	3	8	5

No update!

Weighted Shortest Path



Shortest Distance

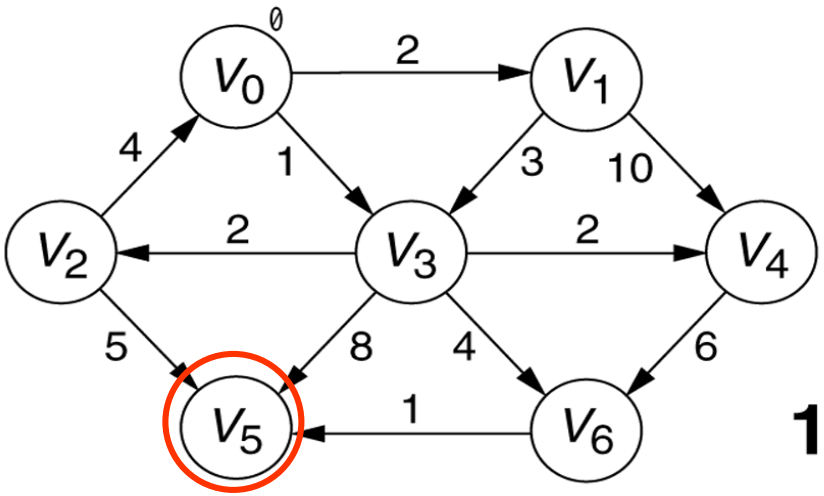
	0	2	3	1	3		5
Priority Queue Q	~~V0~~	~~V1~~	~~V2~~	~~V3~~	~~V4~~	V5	~~V6~~

Adjacent and Non-dropped Nodes

V5				

	V0	V1	V2	V3	V4	V5	V6
Distance Vector D	0	2	3	1	3	6	5

Weighted Shortest Path



Shortest Distance

	0	2	3	1	3	6	5
Priority Queue Q	~~V0~~	~~V1~~	~~V2~~	~~V3~~	~~V4~~	~~V5~~	~~V6~~

Adjacent and Non-dropped Nodes

	V0	V1	V2	V3	V4	V5	V6
Distance Vector D	0	2	3	1	3	6	5