
Estruturas de Informação

Graphs

Fátima Rodrigues

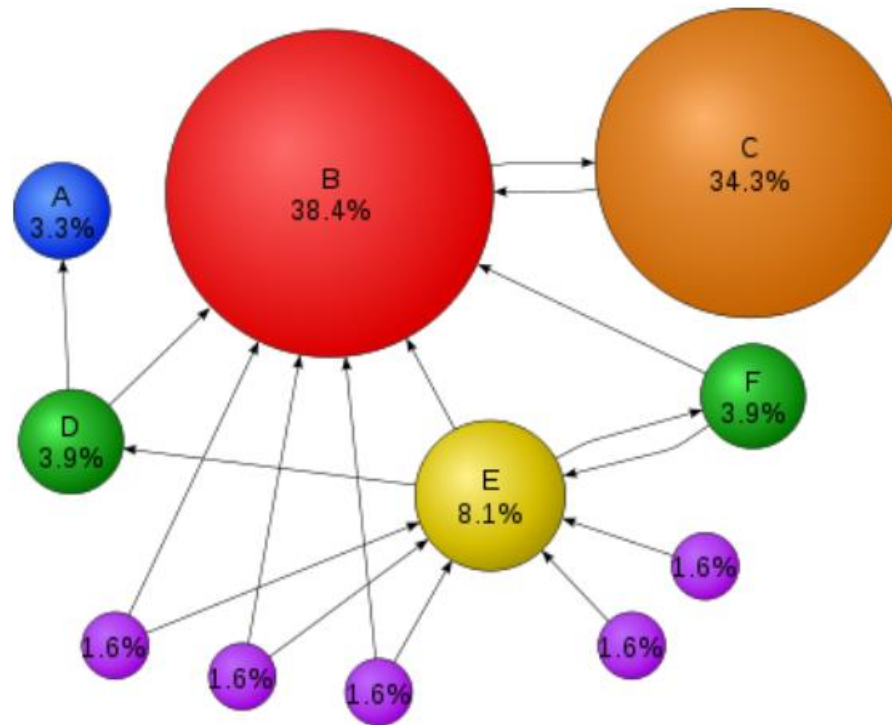
mfc@isep.ipp.pt

Departamento de Engenharia Informática (DEI/ISEP)

Why do we care about graphs?

Many Applications

- Social Networks – Facebook
- Google – Relevance of webpages
- Delivery Networks/Scheduling/Routing – UPS
- Task Scheduling in Projects
- ...



Graphs

Formal definition

A graph is a pair (V, E) where:

- V is a collection of nodes, called **Vertices**
- E is a collection of pairs of vertices, called **Edges**

To each graph edge there is associated a pair of graph vertices

$$\forall_{e \in E} \quad e \rightarrow (u, v) \quad u, v \in V$$

Informal definition

Graphs represent general relationships or connections

- Each node may have many predecessors
- There may be multiple paths (or no path) from one node to another
- Can have cycles or loops

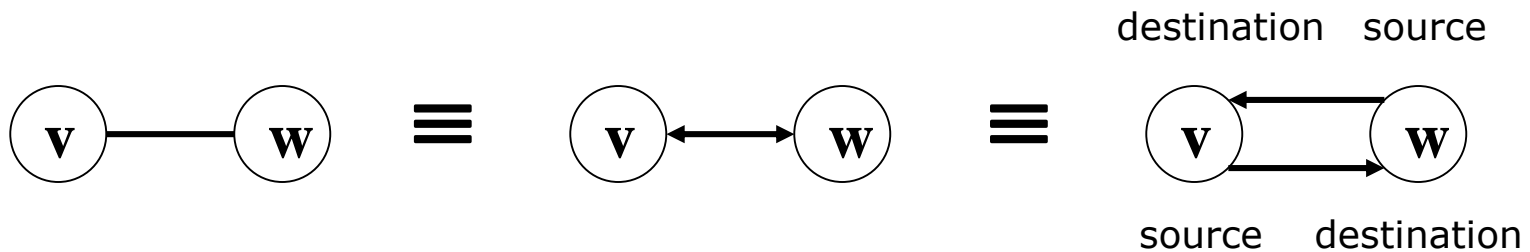
Graphs: Vertices and Edges

A graph is composed of vertices and edges

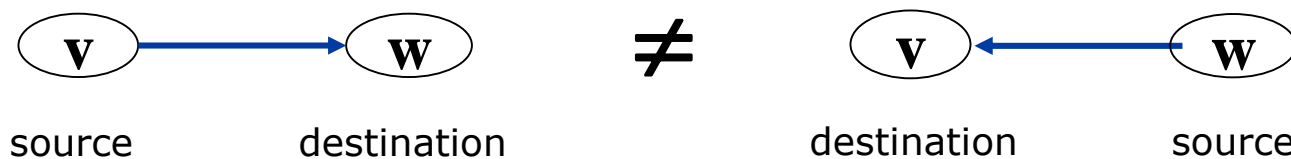
- Vertices (**nodes**):
 - Represent objects, states, positions, place holders
 - Set $\{v_1, v_2, \dots, v_n\}$
 - Each vertex is **unique** \rightarrow no two vertices represent the same object/state
- Edges (**arcs**):
 - Can be directed or undirected
 - Can be weighted (or labeled) or unweighted

Directed and Undirected Edges

- An undirected edge $e = (v_i, v_j)$ indicates that the relationship, connection, etc. is bi-direction:
 - Can go from v_i to v_j (i.e., v_i is related to v_j) and vice-versa

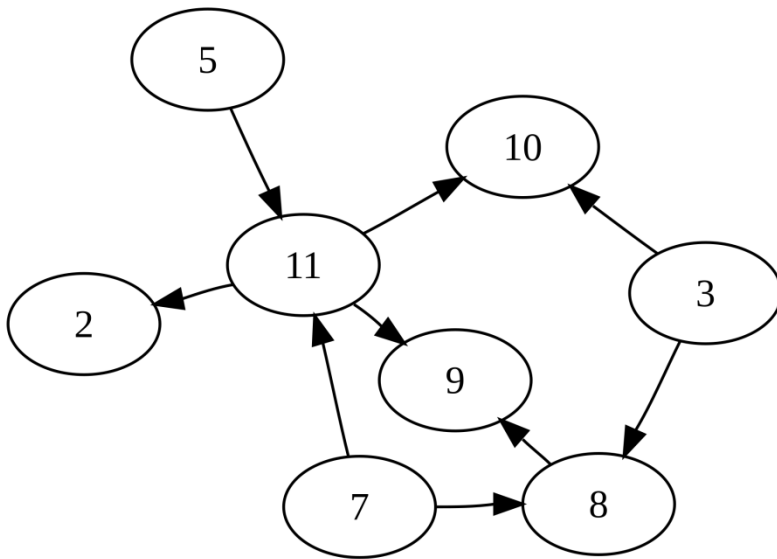


- A directed edge $e = (v_i, v_j)$ specifies a one-directional relationship or connection:
 - Can only go from v_i to v_j

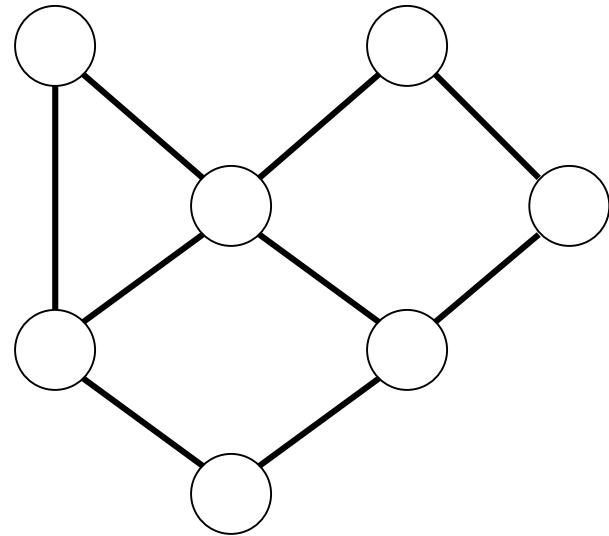


Graphs: Directed and Undirected

- A graph will have either directed or undirected edges, but **not both**



Ex. route network



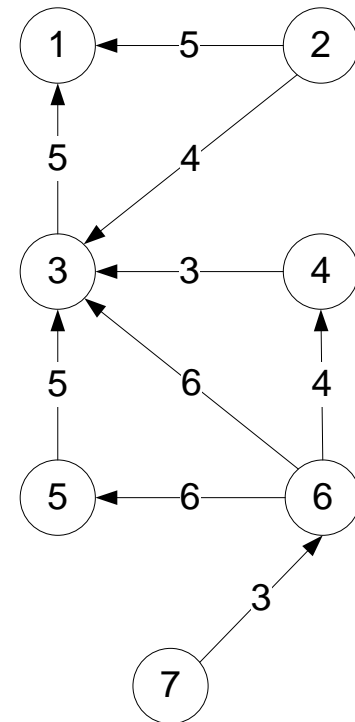
Ex. friends network

Valorised graph

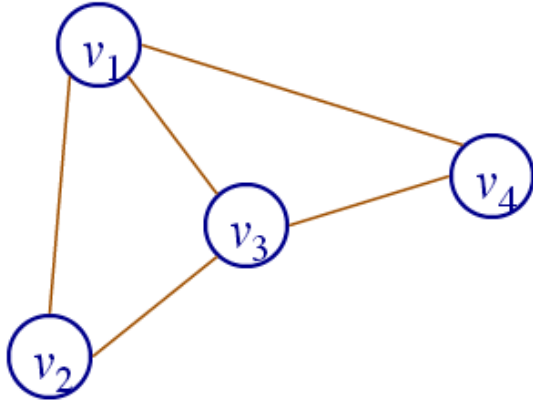
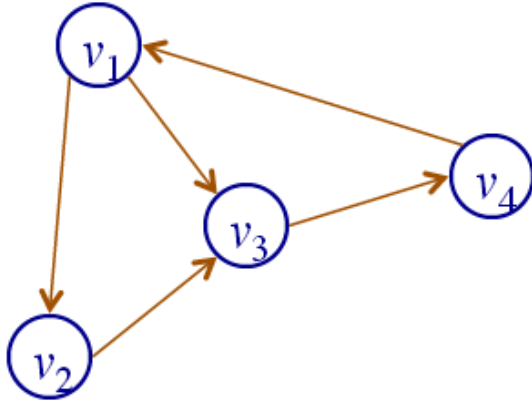
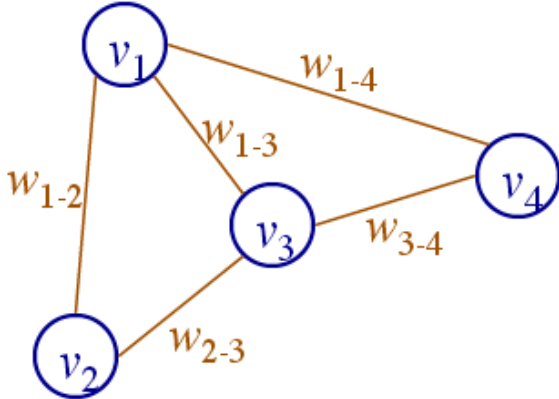
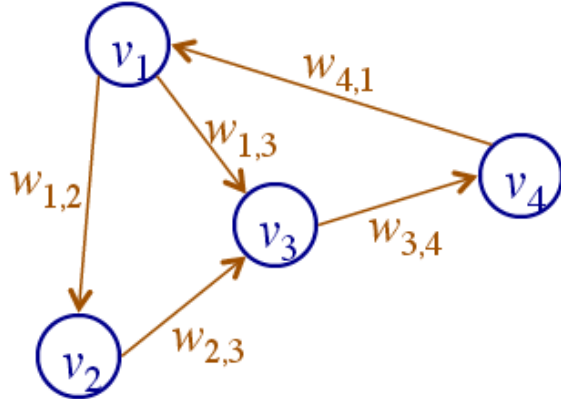
Graph that all its branches have an associated value

These values can represent:

- Costs, distances, or search limitations
- Traffic time
- Waiting time
- Transmission reliability
- Probability of failure occur
- Capacity
- Others

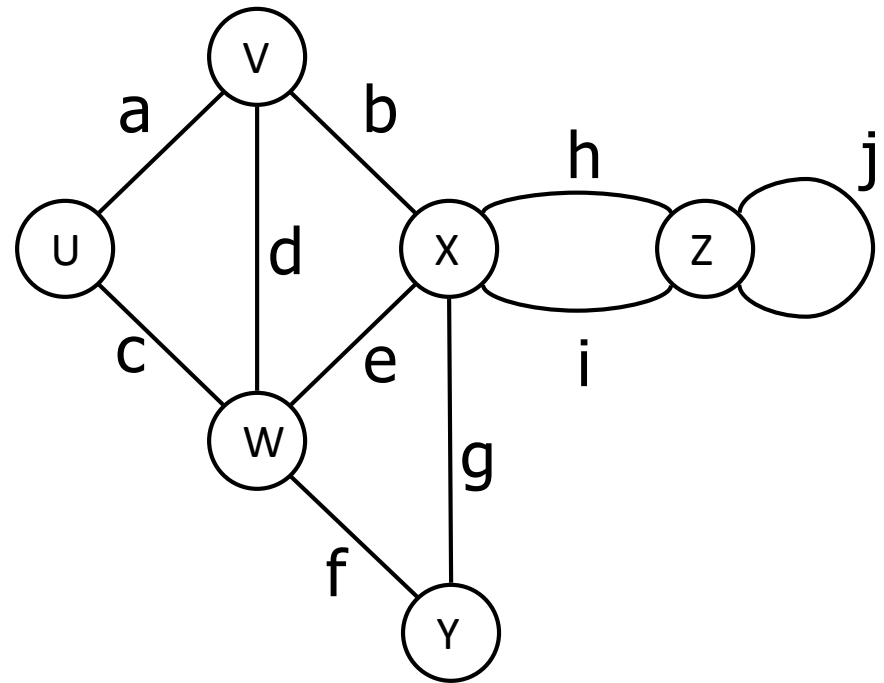


Graphs: Types of Edges

	Undirected	Directed
Unweighted	 <p>An undirected graph with four vertices labeled v_1, v_2, v_3, and v_4. Every vertex is connected to every other vertex by a single edge, forming a complete graph K_4.</p>	 <p>A directed graph with four vertices labeled v_1, v_2, v_3, and v_4. The edges are directed as follows: $v_1 \rightarrow v_2$, $v_1 \rightarrow v_3$, $v_2 \rightarrow v_3$, $v_3 \rightarrow v_4$, and $v_4 \rightarrow v_1$.</p>
Weighted	 <p>An undirected graph with four vertices labeled v_1, v_2, v_3, and v_4. The edges are weighted as follows: w_{1-2} between v_1 and v_2, w_{1-3} between v_1 and v_3, w_{1-4} between v_1 and v_4, w_{2-3} between v_2 and v_3, and w_{3-4} between v_3 and v_4.</p>	 <p>A directed graph with four vertices labeled v_1, v_2, v_3, and v_4. The edges are directed and weighted as follows: $w_{1,2}$ from v_1 to v_2, $w_{1,3}$ from v_1 to v_3, $w_{2,3}$ from v_2 to v_3, $w_{3,4}$ from v_3 to v_4, and $w_{4,1}$ from v_4 to v_1.</p>

Graph Terminology

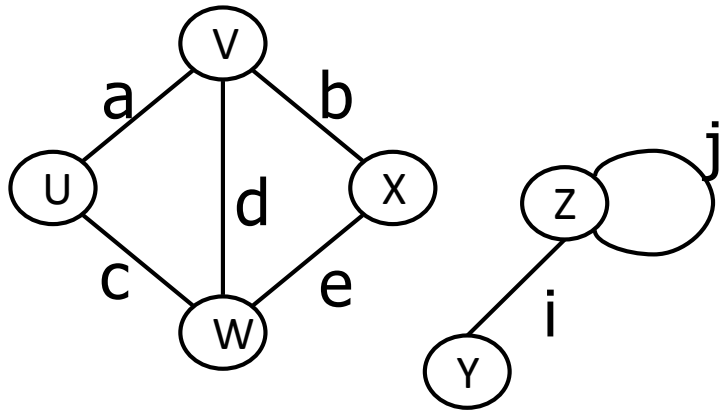
- End vertices (or endpoints) of an edge
 - u and v are the **endpoints** of a
- Edges incident on a vertex
 - a, d, and b are **incident** on v
- Adjacent vertices
 - u and v are **adjacent**
- Degree of a vertex
 - x has **degree 5**
- Parallel edges
 - h and i are **parallel edges**
- Self-loop
 - j is a **self-loop**



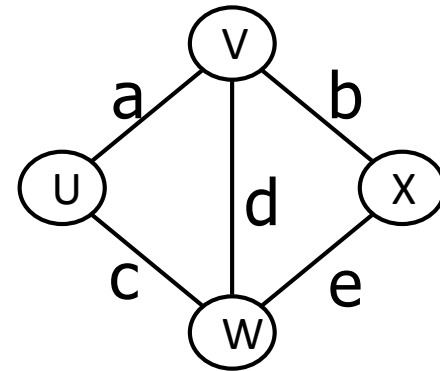
Graph Terminology

- **Connected Graph**

- Graph where **any two vertices are connected** by some path



Not connected graph



Connected graph

- **Subgraph** (V', E') of a Graph (V, E)

- V' is a subset of V , E' is a subset of E , and both endpoints of edges in E' are in V'

- **Spanning subgraph** of G is subgraph that contains all vertices of G

Graph Terminology (cont.)

- **Path**

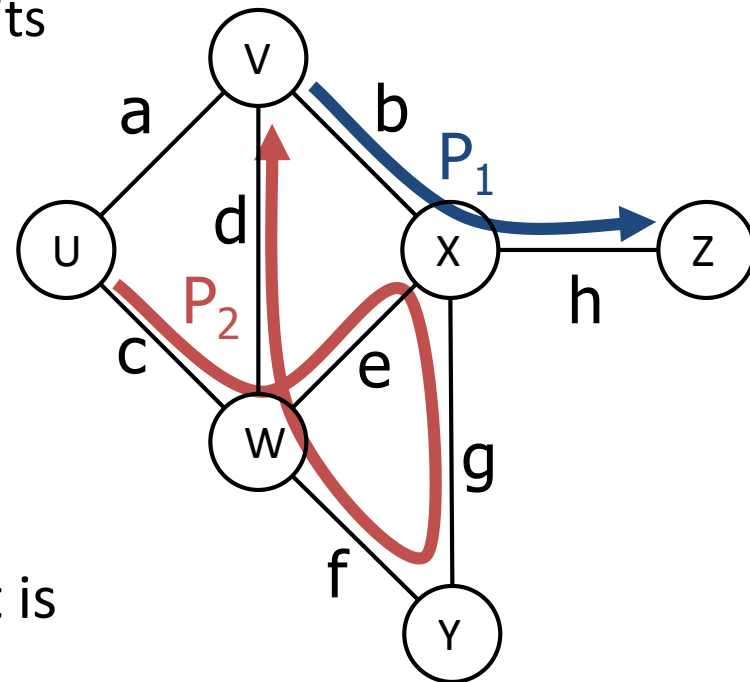
- sequence of **alternating** vertices and edges
- **begins** and **ends** with a vertex
- each **edge** is preceded and followed by its **endpoints**

- **Simple path**

- path such that all its vertices and edges are distinct

- Examples

- $P_1 = (V, b, X, h, Z)$ is a simple path
- $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ is a path that is not simple



Graph Terminology (cont.)

- **Cycle**

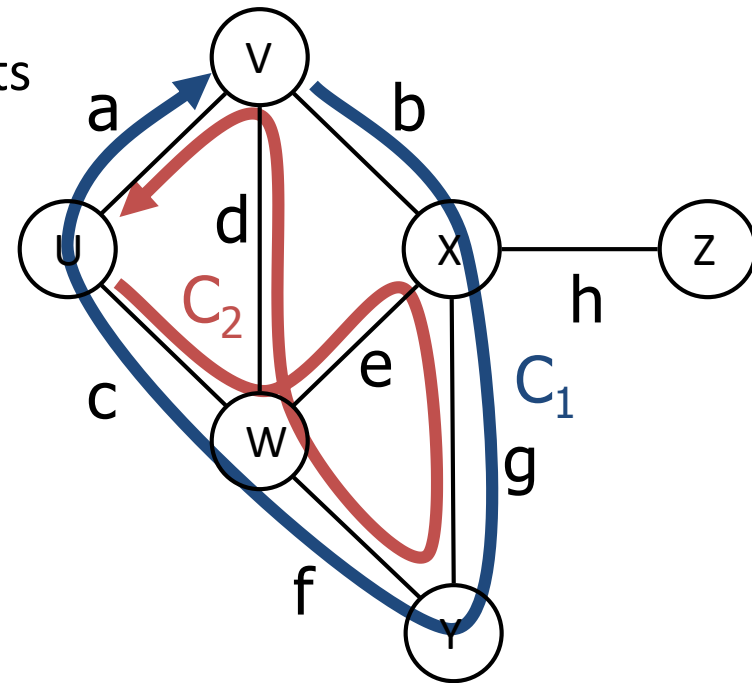
- circular sequence of alternating vertices and edges
- each edge is preceded and followed by its endpoints

- **Simple cycle**

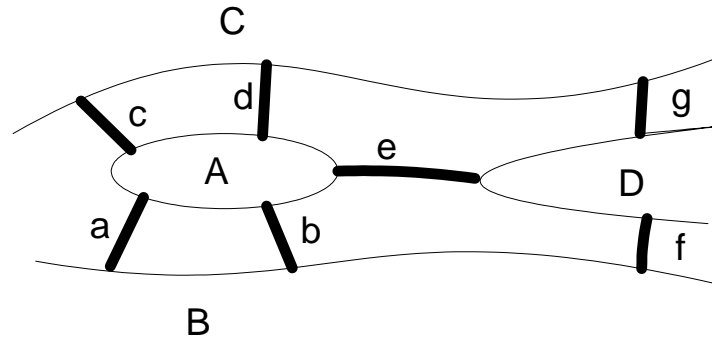
- cycle such that all its vertices and edges are distinct

- Examples

- $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ is a simple cycle
- $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ is a cycle that is not simple

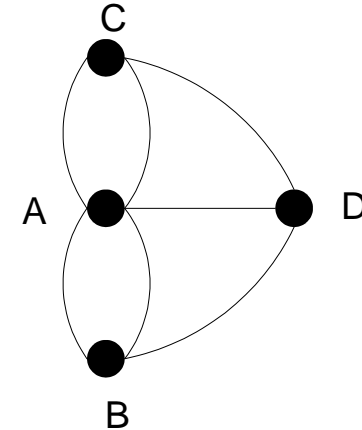
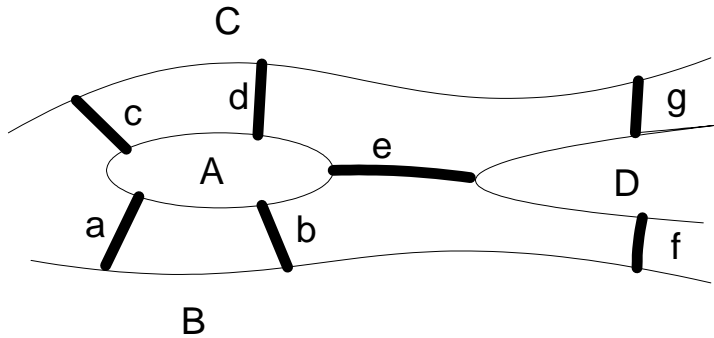


Euler Cycle and the 7 Bridges of Koenigsberg



- The year is 1735. City of Koenigsberg (today Kaliningrado) has a funny layout of 7 bridges across the river
- Citizens of Koenigsberg are wondering if it's possible to walk **across each bridge exactly once and return to same starting point?**
- They think that it's impossible, but no one can prove it

Euler Cycle



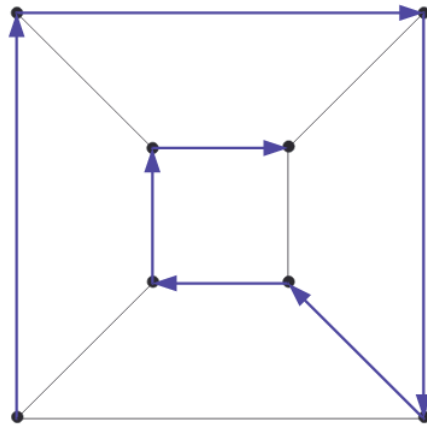
This problem was solved by Euler in 1736 and marks the beginning of Graph Theory

Euler proved

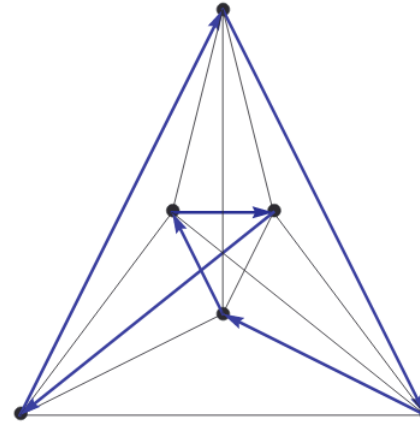
- An **undirected** and **connected** graph has an Euler Cycle iff all the vertices have an even degree
- A **directed** and **strongly connected** graph has an Euler Cycle iff $d_{\text{in}}(V) = d_{\text{out}}(V)$ for each vertex V

Hamilton Path/Cycle

- A simple path/cycle that visits **all the vertices** of the graph **exactly once**



Hamilton path



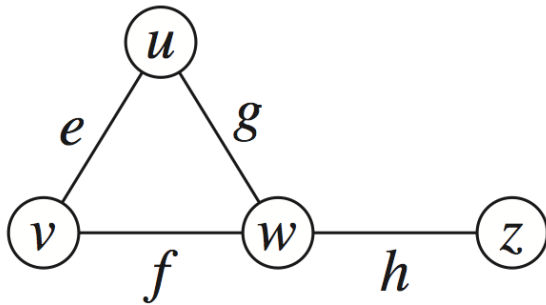
Hamilton cycle

- Unlike the Euler circuit problem, finding Hamilton circuits is hard
- There is no simple set of necessary and sufficient conditions, and no simple algorithm
- The best algorithms known for finding a Hamilton circuit in a graph or determining that no such circuit exists have **exponential worst-case time complexity** (in the number of vertices of the graph)

Graph Representations

Adjacency Matrix Structure

- Represents a graph as a 2-D matrix
- Vertices are indices for rows and columns of the matrix
- Total Space: $O(V^2)$



		0	1	2	3
$u \longrightarrow$	0		e	g	
$v \longrightarrow$	1	e		f	
$w \longrightarrow$	2	g	f		h
$z \longrightarrow$	3			h	

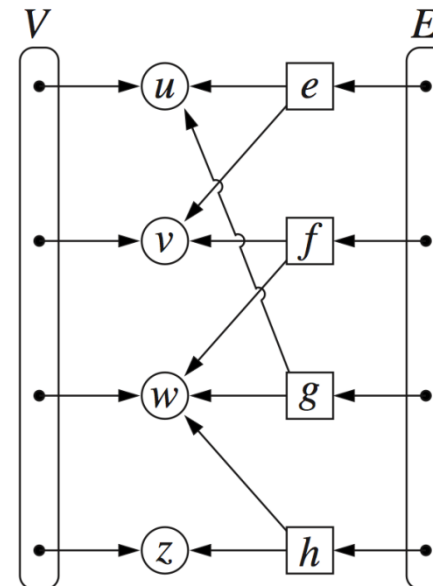
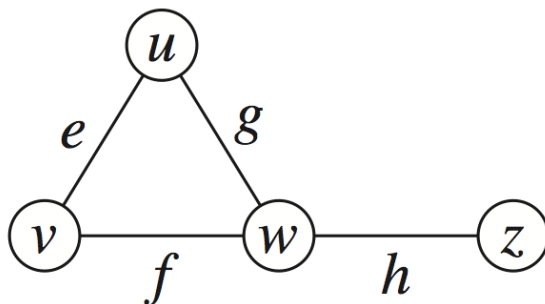
Therefore adjacency matrix should be used only for **dense graphs**

graph is **dense** if $|E| \approx |V|^2$

graph is **sparse** if $|E| \approx |V|$

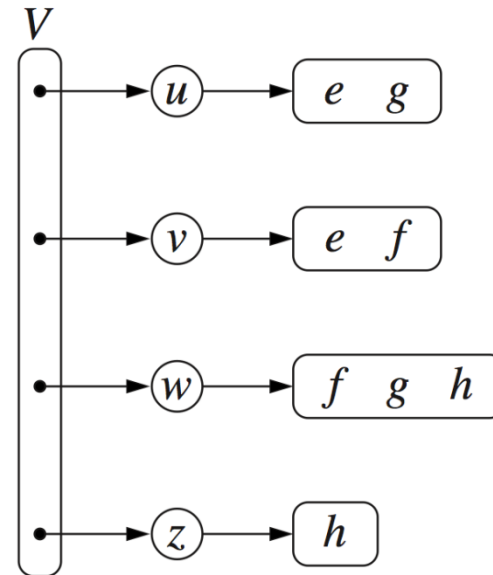
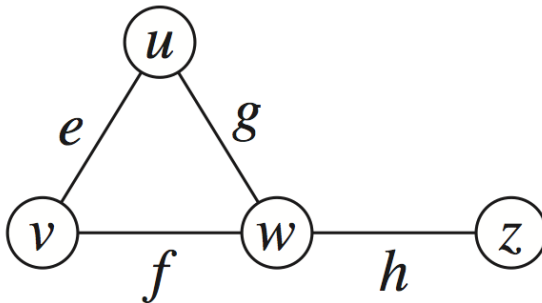
Edge List Structure

- Vertex objects stored in unsorted sequence
 - Space $O(V)$
- Edge objects stored in unsorted sequence
 - Space $O(E)$
- Edge objects has reference to origin and destination vertex object
- Total space: $O(V+E)$



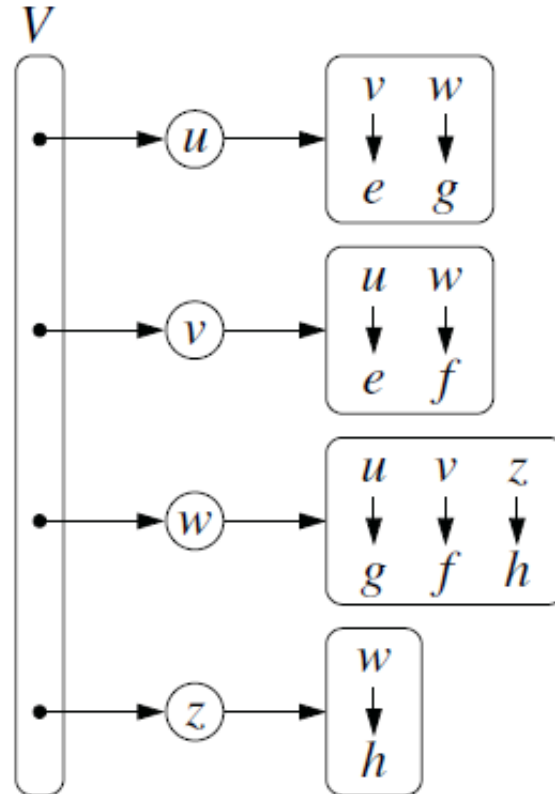
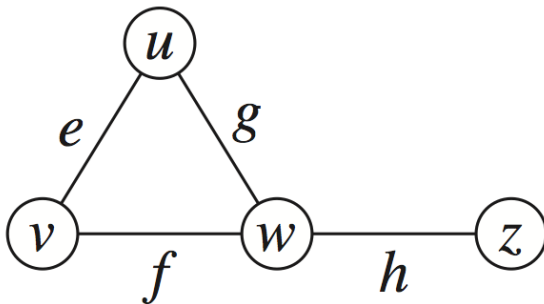
Adjacency List Structure

- Each vertex v_i lists the set of its neighbors
 - sequence of references to its adjacent vertices
- More space-efficient for a sparse graph: Total space $O(V+E)$



Adjacency Map Structure

- Replaces the neighbour list with a Map:
 - with the adjacent vertex serving as a key: **vertex v_i**
 - Its value: **the edge (i,j)**
- This allows more efficient access to a specific edge (i,j) in $O(1)$ expected time
- Total space: $O(V+E)$



Graph ADT

```
public interface Graph <V,E> {  
    int numVertices();  
    Iterable<Vertex<V,E>> vertices();  
    int numEdges();  
    Iterable<Edge<V,E>> edges();  
    Edge<V,E> getEdge(Vertex<V,E> vorig, Vertex<V,E> vdest);  
    Vertex<V,E>[] endVertices(Edge<V,E> e);  
    Vertex<V,E> opposite(Vertex<V,E> v, Edge<V,E> e);  
    int outDegree(Vertex<V,E> v) ;  
    int inDegree(Vertex<V,E> v) ;  
    Iterable<Edge<V,E>> outgoingEdges (Vertex<V,E> v);  
    Iterable<Edge<V,E>> incomingEdges(Vertex<V,E> v);  
    Vertex<V,E> insertVertex(V vInf);  
    Edge<V,E> insertEdge(V vorigInf, V vdestInf, E eInf, double eWeight);  
    void removeVertex(V vInf);  
    void removeEdge(Edge<V,E> e);  
}
```

Asymptotic performance of graph data structures

	Edge List	Adjacency List	Adjacency Map	Adjacency Matrix
Space	$V + E$	$V + E$	$V + E$	V^2
numVertices(), numEdges()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(V)$	$O(V)$	$O(V)$	$O(V)$
getEdge(u, v)	$O(E)$	$O(\min(d_u, d_v))$	$O(1)$	$O(1)$
outDegree(v) inDegree(v)	1	$O(1) / O(V \times E)$	$O(1) / O(V \times E)$	$O(V)$
outgoingEdges(v) incomingEdges(v)	$O(E)$	$O(d_v) / O(V \times E)$	$O(d_v) / O(V)$	$O(V)$
insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(V^2)$
removeVertex(v)	$O(E)$	$O(d_v)$	$O(d_v)$	$O(1)$
insertEdge(u, v, x) removeEdge(x)	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Graph Reachability

Reachability

A common question to ask about a graph is reachability:

- Single-source:
 - Which vertices are “reachable” from a given vertex v_i ?
- All-pairs:
 - For all pairs of vertices v_i and v_j , is v_j “reachable” from v_i ?
 - Solves the single source question for all vertices

All-Pairs Reachability: Adjacency Matrix

To compute all-pairs reachability, it is necessary:

- Start with the **adjacency matrix** of the graph
 - 1: indicates that there is an edge from v_i to v_j
 - 0: no edge from v_i to v_j
- Calculate the **transitive closure** of the graph with **Floyd Warshall's algorithm**
 - **transitive closure** is a matrix with the same vertices as the original graph and an arc between the pairs of vertices that have a path to join them

Floyd-Warshall algorithm - Basic idea

A path exists between two vertices i, j , iff

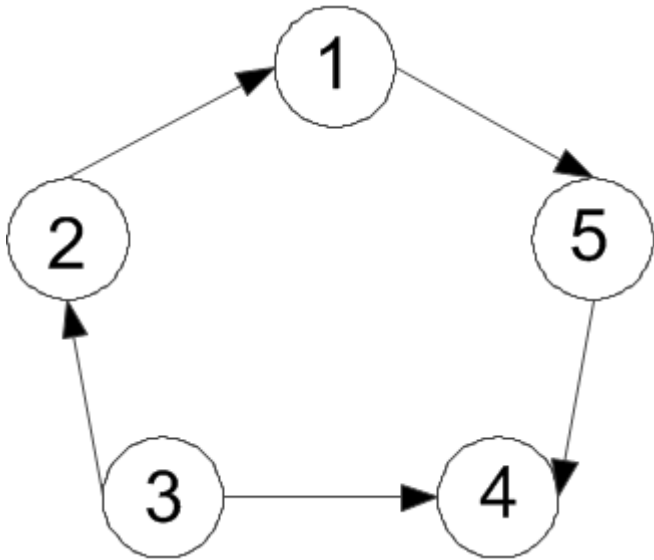
- there is an edge from i to j or
- there is a path from i to j going through vertex 1; or
- there is a path from i to j going through vertex 1 and/or 2; or
- there is a path from i to j going through vertex 1, 2, and/or 3; or
- ...
- there is a path from i to j going through any of the other vertices

On the k^{th} iteration, the algorithm determine if a path exists, between two vertices i, j using just vertices among $1, \dots, k$ allowed as intermediate

$$T_{i,j}^{(k)} = \begin{cases} T_{i,j} & \text{if } k = 0 \\ T_{i,j}^{(k-1)} \vee (T_{i,k}^{(k-1)} \wedge T_{k,j}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall algorithm: Transitive Closure

T^0 matrix is equal to the adjacency matrix – matrix with a path of length 1


$$T^0 =$$

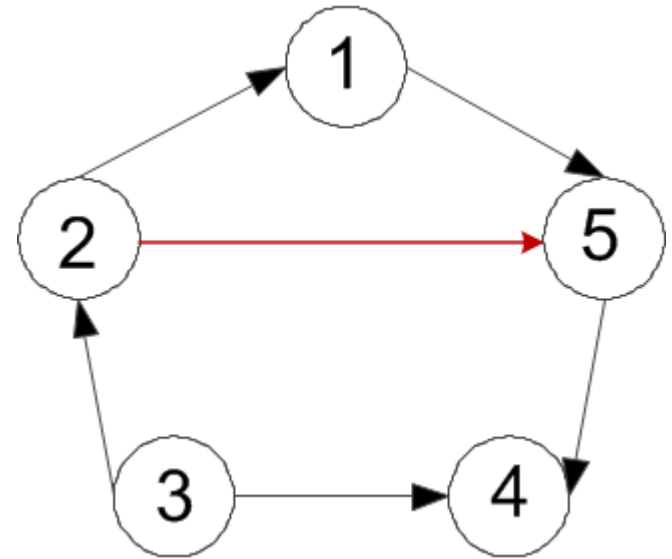
	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	0
3	0	1	0	1	0
4	0	0	0	0	0
5	0	0	0	1	0

$$T_{2,5}^{(1)} = T_{2,5}^0 \vee (T_{2,1}^0 \wedge T_{1,5}^0) = 0 \vee (1 \wedge 1) = 1$$

Floyd-Warshall algorithm: Transitive Closure

$T^1 =$

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



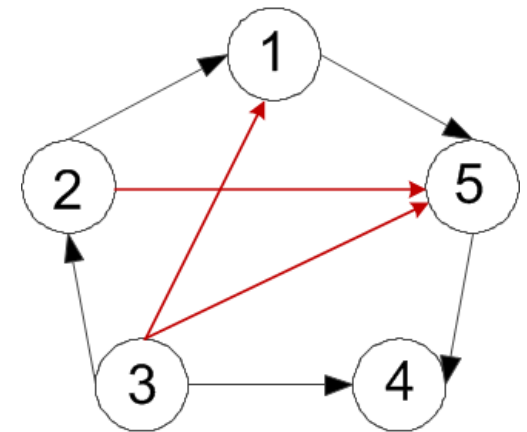
Floyd-Warshall algorithm: Transitive Closure

		1	2	3	4	5			1	2	3	4	5
$T^1 =$	1	0	0	0	0	1	$T^2 =$	1	0	0	0	0	1
	2	1	0	0	0	1		2	1	0	0	0	1
	3	0	1	0	1	0		3	1	1	0	1	1
	4	0	0	0	0	0		4	0	0	0	0	0
	5	0	0	0	1	0		5	0	0	0	1	0

$$T_{3,1}^{(2)} = T_{3,1}^1 \vee (T_{3,2}^1 \wedge T_{2,1}^1) = 0 \vee (1 \wedge 1) = 1$$

$$T_{3,5}^{(2)} = T_{3,5}^1 \vee (T_{3,2}^1 \wedge T_{2,5}^1) = 0 \vee (1 \wedge 1) = 1$$

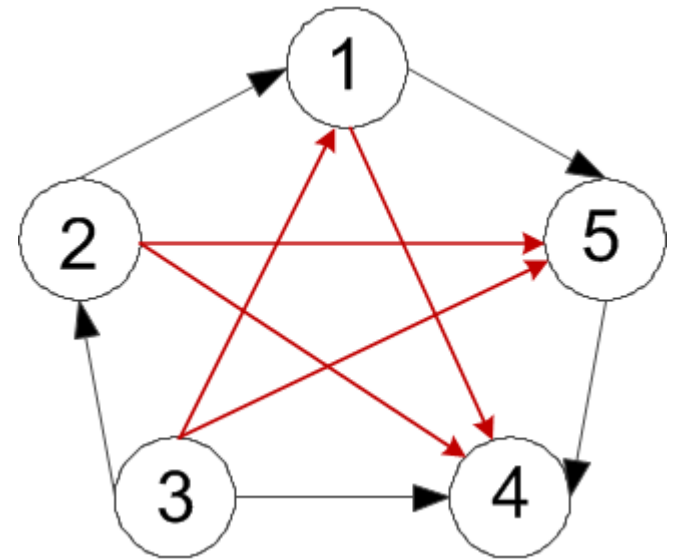
The addition of the vertex 3 and 4 doesn't add new paths, so $T^2 = T^3 = T^4$



Floyd-Warshall algorithm: Transitive Closure

The addition of vertex 5 allows to add edges (1,4) e (2,4)

		1	2	3	4	5
$T^5 =$	1	0	0	0	1	1
	2	1	0	0	1	1
	3	1	1	0	1	1
	4	0	0	0	0	0
	5	0	0	0	1	0



The final matrix has a 1 in row i and column j , if vertex v_j is reachable from vertex v_i via some path

Floyd-Warshall algorithm

```
Algorithm void transitiveClosure (Graph<V,E> g) {  
    for (k ← 0; k < n; k++)  
        for (i ← 0; i < n; i++) {  
            if (i != k && T[i,k] = 1)  
                for (j ← 0; j < n; j++)  
                    if (i != j && k != j && T[k,j] = 1 )  
                        T[i,j] = 1  
        }  
    }
```

Time Complexity: $O(?)$

All-Pairs Reachability: Weighted Graph

- **Key difference:** adjacency graph now has weights instead of binary values
- In place of logical operations (AND, OR) use arithmetic operations (addition)

$$D_{i,j}^{(k)} = \begin{cases} w_{i,j} & \text{if } k = 0 \\ \min(D_{i,j}^{(k-1)}, D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

The final matrix gives, in row i and column j , the **length of the minimum path** between vertices i , j , if vertex v_j is reachable from vertex v_i via some path

Graph Traversals

- For solving most problems on graphs
 - Need to systematically visit all the vertices and edges of a graph
- There are two standard graph traversal techniques that provide an efficient way to “visit” each vertex and edge exactly once:
 - Breadth-First Search (BFS)
 - Depth-First Search (DFS)
- Graph Traversals (BFS, DFS):
 - Starts at some source vertex S
 - Discover every vertex that is **reachable** from S

Breadth-First Search – Basic Idea

1. Choose a **starting vertex**, its level is called the current level
2. From each node N in the current level, in the order in which the level nodes were visited, visit all the **unvisited neighbours** of N. The newly visited nodes from this level form a new level that becomes the next current level
3. Repeat step 2 until no more nodes can be visited
4. If there are still unvisited nodes, repeat from Step 1

BFS → For each vertex visit all its edges (**neighbours**)

Breadth-First Search - Algorithm

```
Algorithm Deque<V> BFS(Graph<V,E> G, Vertex<V,E> vOrig){  
    Add vOrig-element to qbfs  
    Add vOrig to qaux  
    Make vOrig as visited  
    while (!qaux is Empty){  
        vOrig ← Remove first vertex from qaux  
        for (each of vOrig's outgoing edges, e = (VOrig,vAdj)) {  
            if (vertex vAdj has not been visited) {  
                Add vAdj-element to qbfs;  
                Add vAdj to qaux;  
                Make vAdj as visited;  
            }  
        }  
    }  
    return qbfs;  
}
```

Time Complexity: $O(?)$

Depth-First Search - Basic Idea

1. choose a starting vertex, distance $d = 0$
2. Examine **One edge** leading from vertex (at distance d) to adjacent vertices (at distance $d+1$)
3. Then, examine **One edge** leading from vertices at distance $d+1$ to distance $d+2$, and so on,
 4. until no new vertex is discovered, or dead end
5. Then, **backtrack** one distance back up, and try other edges, and so on
6. Until finally backtrack to starting vertex, with no more new vertex to be discovered

Depth-First Search - Algorithm

Algorithm void DFS(Graph<V,E> G, Vertex<V,E> vOrig, Deque<V> qdfs){

 Push vOrig-element to qdfs

 Make vOrig as visited

 for (each of vOrig's outgoing edges, e = (vOrig,vAdj)) {

 if (vertex vAdj has not been visited)

 Recursively call DFS(G, vAdj, qdfs);

 }

}

Time Complexity: $O(?)$

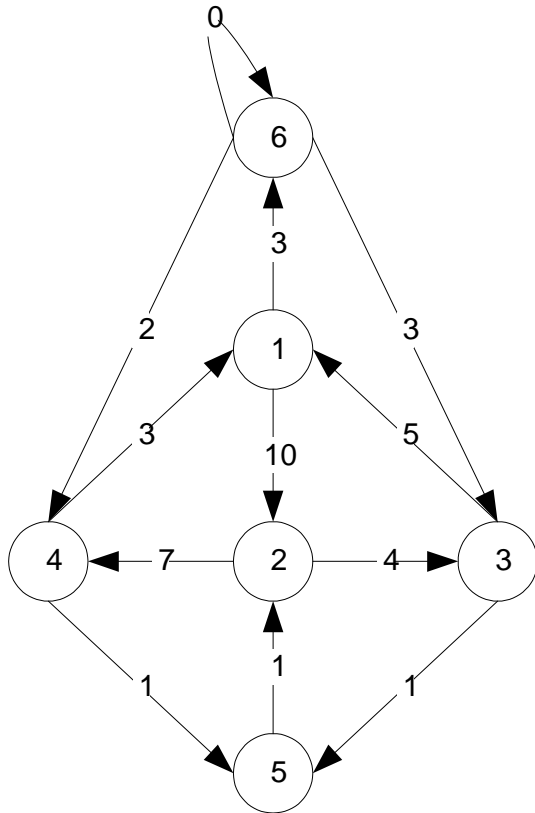
Graphs

Paths and Shortest Paths

Paths and Circuits

- **Path** in a graph is an alternating sequence of adjacent vertices and edges, that does not contain a repeated edge
- **Simple path** is a path that does not contain a repeated vertex
- **Circuit** is a closed path that does not contain a repeated edge
 - **Simple circuit** is a circuit which does not have a repeated vertex except for the first and last
 - **Euler circuit** is a circuit that contains every vertex and every edge of a graph. Every edge is traversed exactly once
 - **Hamiltonian circuit** is a simple circuit that contains all vertices of the graph (and each exactly once)

All Simple paths between two vertices



Adjacency List

1 → **2 , 6**
2 → **3 , 4**
3 → **1 , 5**
4 → **5**
5 → **2**
6 → **3, 4, 6**

All paths between Vertices 1 - 5:

- 1, 2, 3, 5
- 1, 2, 4, 5
- 1, 6, 3, 5
- 1, 6, 4, 5

Count all paths between two vertices - Algorithm

```
Algorithm void numberOfPaths (Graph<V,E> G, Vertex<V,E> vOrig,
                             Vertex<V,E> vDst, boolean[] visited, int cont){
    Make vOrig as visited
    for (each of vOrig's outgoing edges, e = (vOrig,vAdj)) {
        if (vertex vAdj == vDst)
            cont++
        else
            if (vertex vAdj has not been visited)
                Recursively call numberOfPaths(G, vAdj, vDst, cont)
    }
    Make vOrig as not visited
}
```

Time Complexity: $O(?)$

All paths between two vertices - Algorithm

```
Algorithm void allPaths (Graph<V,E> G, Vertex<V,E> vOrig,
                        Vertex<V,E> vDst, boolean [] visited, Deque<V> path,
                        ArrayList<Deque<V>> paths){

    make vOrig as visited
    push vOrig onto path
    for (each of vOrig's outgoing edges, e = (vOrig,vAdj)){
        if (vertex vAdj == vDst){
            push vDst onto path
            add path to paths
            pop last Vertex from path }
        else
            if (vertex vAdj has not been visited)
                recursively call allPaths(G,vAdj,vDst,path,paths)
    }
    make vOrig as not visited
    pop last Vertex from path
}
```

Time Complexity: $O(?)$

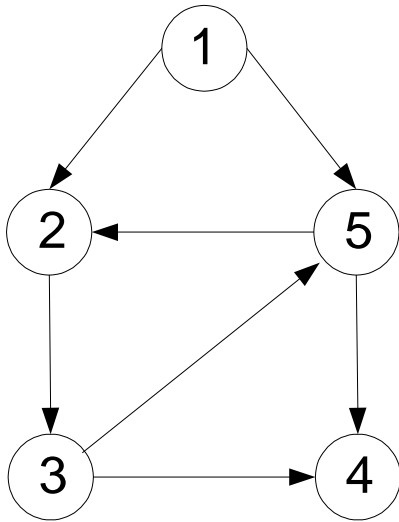
Shortest path problem

Given a directed graph $G(V,E)$ find the shortest path from a given start vertex S to all other vertices (*Single Source Shortest Paths*) where the length of a path is the sum of its edge weights

- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex S
- Assumptions:
 - the graph is connected
 - the edges are undirected
 - the edge weights are **nonnegative**

Single Source Shortest Paths on unweighted Graph

Given a graph $G = (V, E)$ directed, unweighted and an initial vertex s , find all shortest paths between this and any other vertex of the graph



Shortest paths
starting at vertex 1:

- 1-2
- 1-5
- 1-2-3
- 1-5-4

Weight path

1

1

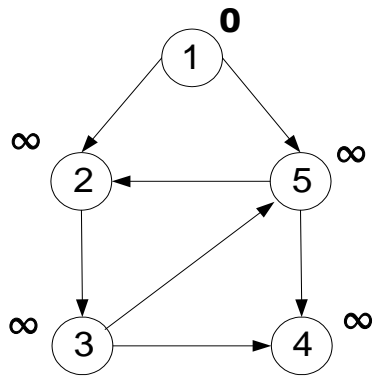
2

2

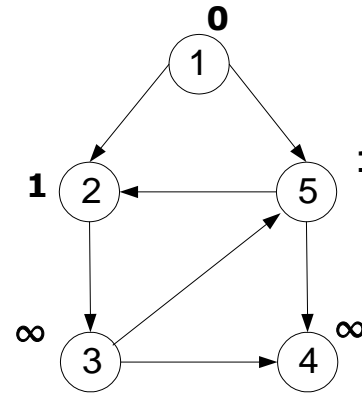
Dijkstra's Algorithm for an unweighted Graph

- The main idea is to perform a **breadth-first search** starting at the source vertex S
- The algorithm uses two vectors which records for the others vertices:
 - the **distance** from each vertex to the initial vertex (dist)
 - the **predecessor** on the shortest path (path)
- The algorithm starts to mark the initial vertex S with length 0
- then processes its adjacent vertices that are marked with a path of length 1, and continues making a breadth-first search

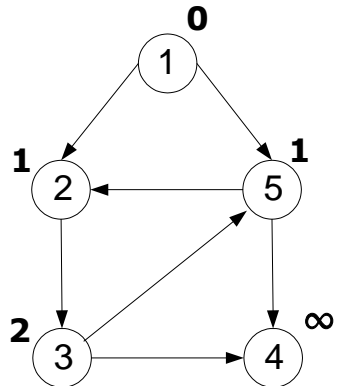
Dijkstra's Algorithm exemplification



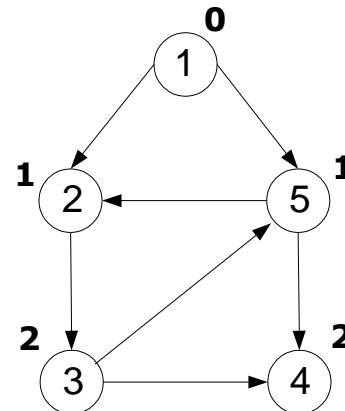
dist	0	∞	∞	∞	∞
	1	2	3	4	5
path	0	0	0	0	0
	1	2	3	4	5
queue		1			



dist	0	1	∞	∞	1
	1	2	3	4	5
path	0	1	0	0	1
	1	2	3	4	5
queue		2	5		



dist	0	1	2	∞	1
	1	2	3	4	5
path	0	1	2	0	1
	1	2	3	4	5
queue		5	3		



dist	0	1	2	2	1
	1	2	3	4	5
path	0	1	2	5	1
	1	2	3	4	5
queue		3	4		

.... ends

queue					
-------	--	--	--	--	--

Dijkstra's Algorithm for an unweighted Graph

```
Algorithm void shortestPathEdges(Graph<V,E> g, Vertex<V,E> vOrig,
                                int[] path, double[] dist){
    for (all V vertices in g) { dist[V]= $\infty$  path[V]=-1 }
    make vOrig as visited
    add vOrig to queue-aux
    while (!queue-aux is Empty){
        vOrig  $\leftarrow$  Remove first vertex from queue-aux
        for (each of vOrig's outgoing edges, e = (vOrig,vAdj))
            if (dist[vAdj] =  $\infty$ ) {
                dist[vAdj] = dist[vOrig] + 1
                path[vAdj] = vOrig
                Add vAdj to queue-aux
            }
    }
}
```

Time Complexity: $O(?)$

Dijkstra's Algorithm for a weighted Graph

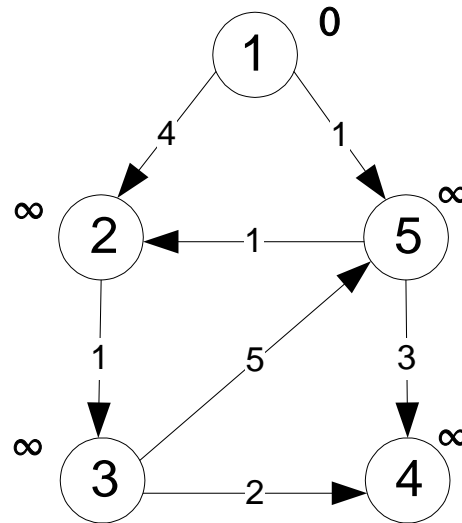
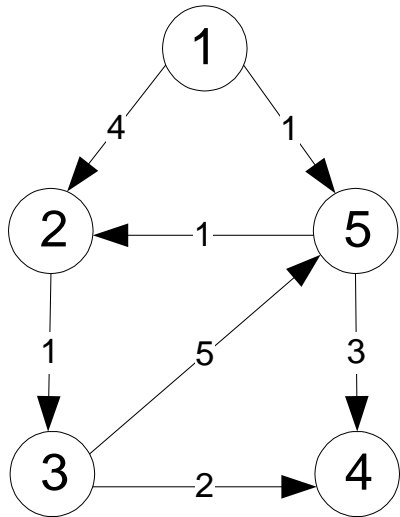
The problem of computing the shortest path in a weighted graph is solved by making minor modifications to the previous algorithm

- As in that algorithm, a **visited map** that maps vertices to their distances from the source vertex S is kept
- Instead of adding 1 to the distance, it is added the weight of the edge traversed
- **Crucial modification**: at each iteration choose the vertex with the smallest distance to the source vertex

This approach is a simple, but nevertheless powerful, is an example of the **greedy-method design pattern**

Restriction: is only valid if there are no branches with **negative costs**

Dijkstra's Algorithm for a weighted graph exemplification



path

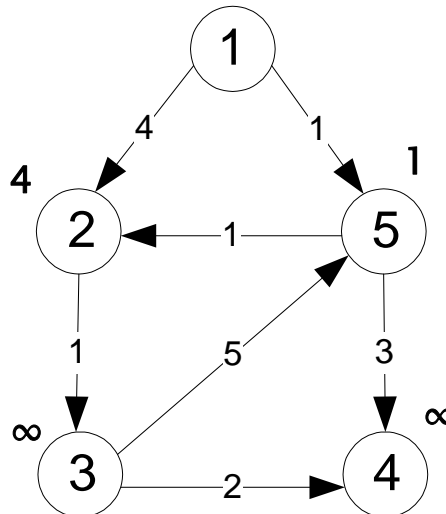
0	0	0	0	0
1	2	3	4	5

dist

0	∞	∞	∞	∞
1	2	3	4	5

visited

0	0	0	0	0
1	2	3	4	5



path

0	1	0	0	1
1	2	3	4	5

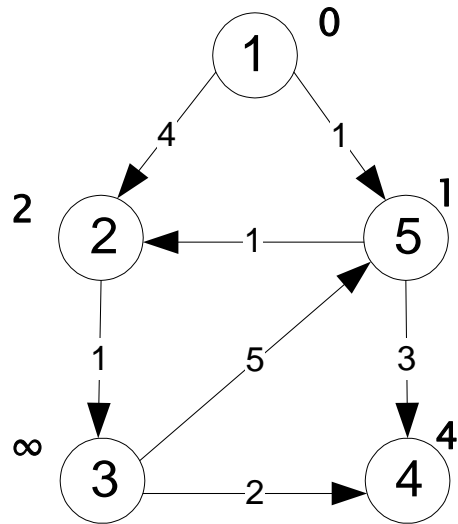
dist

0	4	∞	∞	1
1	2	3	4	5

visited

1	0	0	0	0
1	2	3	4	5

Dijkstra's Algorithm for a weighted graph exemplification



path

0	5	0	5	1
---	---	---	---	---

1 2 3 4 5

dist

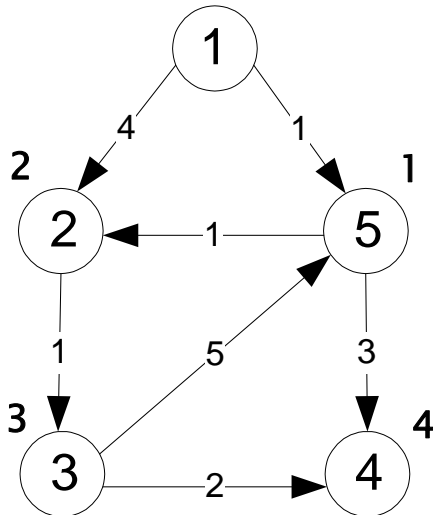
0	2	∞	4	1
---	---	----------	---	---

1 2 3 4 5

visited

1	0	0	0	1
---	---	---	---	---

1 2 3 4 5



path

0	5	2	5	1
---	---	---	---	---

1 2 3 4 5

dist

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

1 2 3 4 5

visited

1	1	0	0	1
---	---	---	---	---

1 2 3 4 5

path

0	5	2	5	1
---	---	---	---	---

1 2 3 4 5

dist

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

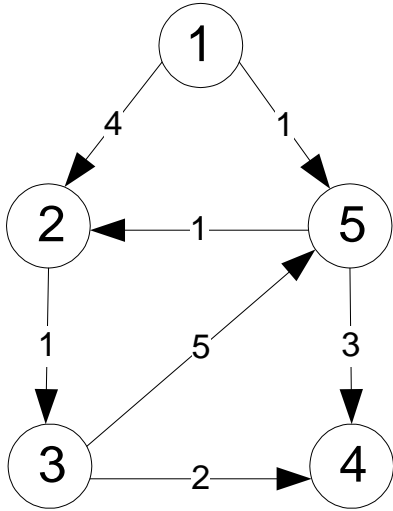
1 2 3 4 5

... visited

1	1	1	1	1
---	---	---	---	---

1 2 3 4 5

Single Source Shortest Paths on a weighted Graph



path	0	5	2	5	1
	1	2	3	4	5
dist	0	2	3	4	1
	1	2	3	4	5

Shortest paths

Source Vertex 1 :

— 1-5-2

— 1-5-2-3

— 1-5-4

— 1-5

Weight

2

3

4

1

Dijkstra's Algorithm for a weighted Graph

```
Algorithm void shortestPathLength(Graph<V,E> g, Vertex<V,E> vOrig,
                                boolean[] visited, int[] path, double[] dist){

    for (all V vertices in g) {dist[V]=∞ path[V]=-1 visited[v]=false }
    dist[vOrig]=0
    while (vOrig != -1){
        make vOrig as visited
        for (each of vOrig's outgoing edges, edge = (vOrig,vAdj))
            if (!visited[vAdj] && dist[vAdj]>dist[vOrig]+edge.getWeight()){
                dist[vAdj] = dist[vOrig]+edge.getWeight()
                path[vAdj] = vOrig
            }
        vOrig = getVertMinDist(dist, visited)
    }
}
```

Time Complexity: O(?)

Graphs with negative weight edges

Applications:

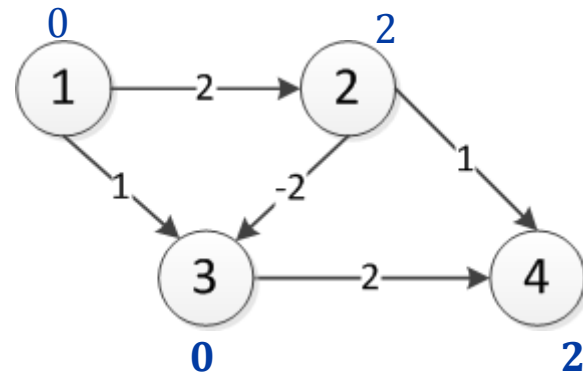
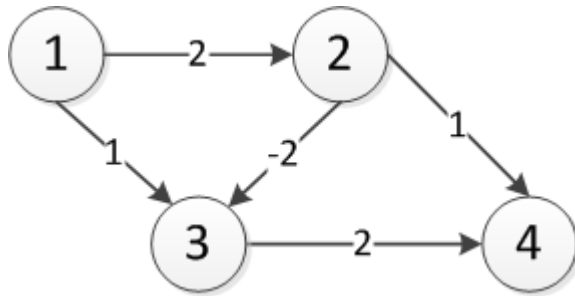
- Distance vector routing protocols in networking
- Currency Exchange
- Chemical processes

Dijkstra's algorithm fails for graphs with **negative weight** edge

Negative weight edges

Why Dijkstra's algorithm doesn't work for Negative-Weight Edges?

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance
- When a vertex with a negative incident edge is selected late, it could alter distances of vertices already processed



- Dijkstra's algorithm would visit 3 then 2 and leave vertices 3 and 4 with a wrong distance

Bellman-Ford Algorithm

Bellman-Ford algorithm computes the *Single Source Shortest Path* in graphs with **negative weight** edge but with **no negative cycles**

The algorithm shares the notion of **edge relaxation** from Dijkstra's algorithm, but does not use it in conjunction with the **greedy method**

Basic Idea

- Assuming there are **no negative cycles**, every shortest path is **Simple** and contains at most **$n-1$ edges**
- Therefore the Bellman-Ford algorithm correctly identifies all shortest paths from a source vertex S in at most **$n-1$ iterations**

Bellman-Ford Algorithm – Basic Idea

- Iterate over the number of vertices
- Keep track of the current shortest path (distance and parent) for each vertex
- For each iteration, “relax” all edges
 - Consider whether this edge can be used to improve the current shortest path of the vertex at its endpoint
 - Because every shortest path is simple, after at most $n-1$ iterations, every shortest path is determined
- To check for negative cycles, after completing $n-1$ iterations, simply scan all edges one more time to see if there is a vertex that could still be improved
 - If so, that means there exists a path longer than $n-1$ edges to achieve the shortest path, which implies a negative cycle

Bellman-Ford Algorithm

```
Algorithm boolean BellmanFord (Graph<V,E> g, Vertex<V,E> vOrig,  
                             int[] path, double [] dist){  
  
    for (all V vertices in g) { dist[V]= $\infty$  path[V]=-1 }  
    dist[vOrig]=0  
  
    for (all vOrig vertices in g) {  
        for (each of vOrig's outgoing edges, e = (vOrig,vAdj))  
            if (dist[vAdj] > dist [vOrig]+edge.getWeight ) { //relaxation  
                dist[vAdj] = dist [vOrig]+ edge.getWeight  
                path[vAdj] = vOrig  }  
    }  
  
    for (every edge(vOrig,vAdj) in g) //determine a negative cycle  
        if (dist[vAdj] > dist[vOrig]+edge.getWeight)  
            return false  
  
    return true  
}
```

Time Complexity (?)

Graphs Circuits

Circuits

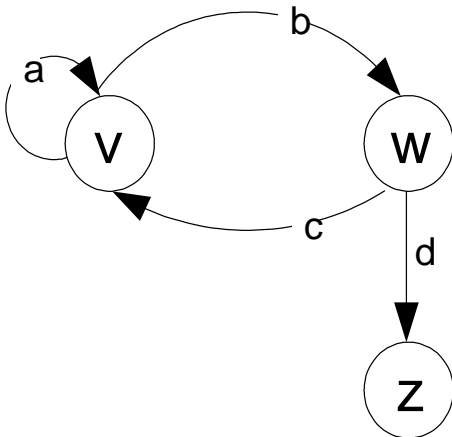
Circuit (or cycle) is a closed path that does not contain a **repeated edge**

Simple circuit is a circuit which does not have a **repeated vertex** ($v_0, v_1, v_2, \dots, v_k$) except for the **first** and **last** $v_k = v_0$

If (u, v, w, \dots, z, u) is a cycle, (v, w, \dots, z, u, v) is also a cycle

Any cyclic permutation of a cycle is an original equivalent cycle

Example:



The paths:

- (v, v) is a cycle length 1
- (v, w, v) is a cycle length 2
- (w, v, w) is an equivalent cycle to the above
- (w, v, v, w) is not a cycle - Vertex v is repeated

Cycle Detection - Basic Idea

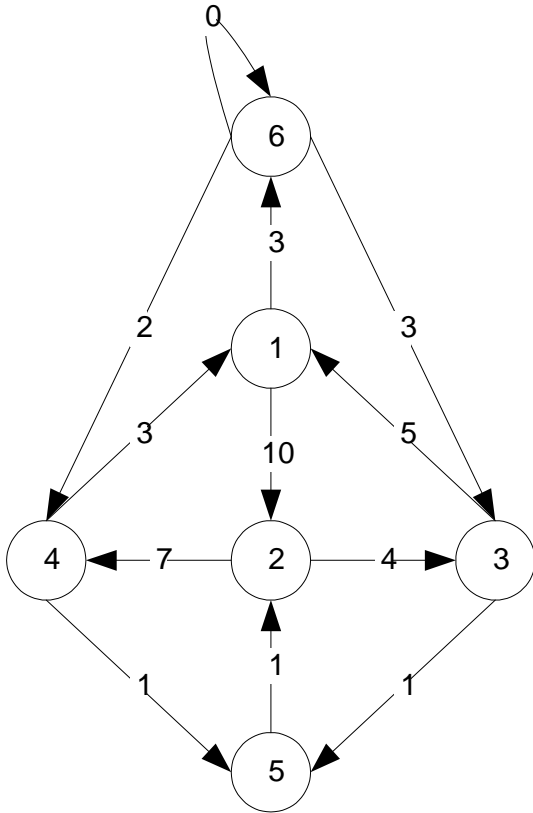
- In a DFS of an acyclic graph, a vertex whose adjacent vertices have all been visited, can be seen again without implying a cycle
- However, if a vertex is seen a second time before all of its adjacent vertices have been visited, then there must be a cycle
- Suppose there is a cycle containing vertex A. Then this means that A must be reachable from one of its adjacent vertices. So when the DFS is visiting that adjacent vertex, it will see A again, before it has finished visiting all of A's adjacent - So there is a cycle
- In order to detect cycles, we use a modified DFS called a colored DFS
 - All nodes are initially marked white
 - When a node is encountered, it is marked grey
 - and when its descendants are completely visited, it is marked black
- If a grey node is ever encountered, then there is a cycle

Cycle Detection - Algorithm

```
Algorithm boolean colorDFS(Graph<V,E> G, Vertex<V,E> vOrig,
                                int[] color) {
    make vOrig gray
    for (each of vOrig's outgoing edges, e = (vOrig,vAdj)) {
        if (vertex vAdj is gray)
            return false
        if (vertex vAdj is white)
            return colorDFS(G, vAdj, color)
    }
    make vOrig black
    return true
}
```

Time Complexity (?)

All cycles in a Graph



Adjacency List

1 → 2, 6

2 → 3, 4

3 → 1, 5

4 → 1, 5

5 → 2

6 → 3, 4, 6

All cycles (– 9 –) :

- 1-2-3-1

- 1-2-4-1

- 1-6-3-1

- 1-6-3-5-2-4-1

- 1-6-4-1

- 1-6-4-5-2-3-1

- 2-3-5-2

- 2-4-5-2

- 6-6

All cycles in a Graph – Algorithm

```
Algorithm ArrayList<Deque<V>> allCycles (Graph<V,E> g){  
  
    for (all Vertex vOrig in g) {  
        for (all V Vertices previous to vOrig) { visited[V]=true }  
        allPaths(g, vOrig, vOrig, visited, path, paths);  
    }  
    return paths;  
}
```

Euler Circuit

An **Euler path** is a path that uses every edge of a graph exactly once. An Euler path starts and ends at **different** vertices

An **Euler circuit** is a circuit that uses every edge of a graph exactly once. An Euler circuit starts and ends at the **same** vertex

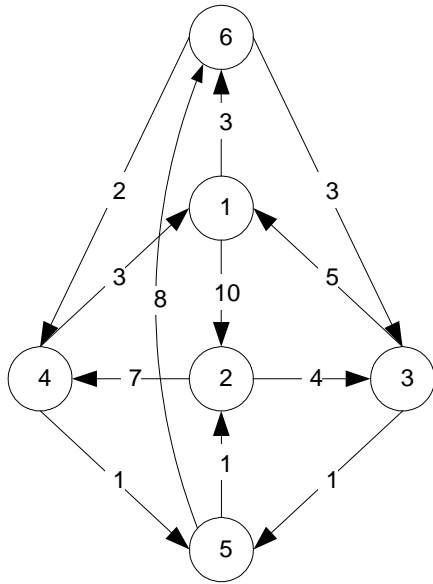
Accordingly to Euler theorem:

- An **undirected** and **connected** graph has an Euler Cycle iff all the vertices have an even degree
- A **directed** and **strongly connected** graph has an Euler Cycle iff $d_{\text{in}}(V) = d_{\text{out}}(V)$ for each vertex V

Euler Circuit – Basic Idea

- Every vertex of G has degree ≥ 2 so, G necessarily has some **simple cycle** C_1
 - if C_1 contains all edges of G - C_1 is the Euler circuit
 - if not, remove from G all the edges of C_1
 - The resulting graph has all vertices still with even degree
 - So, it is possible to determine new cycle C_i
 - and repeat this process until no more edges in G
- At the end, the edges of G are partitioned into **simple cycles**
- Because G is connected, each cycle C_i has at least one vertex in common which permits to make the union of all cycles and obtain the **Euler cycle** that contains **all edges of G exactly once**

Hierholzer Algorithm - Exemplification

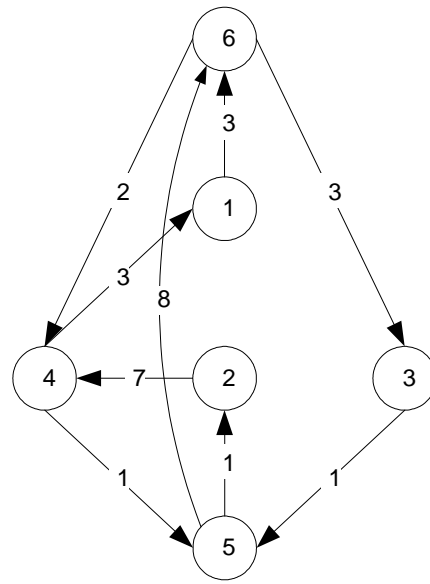


Adjacency List

1 → **2, 6**
2 → **3, 4**
3 → **1, 5**
4 → **1, 5**
5 → **2, 6**
6 → **3, 4**

1st iteration:

1 - 2 - 3 - 1

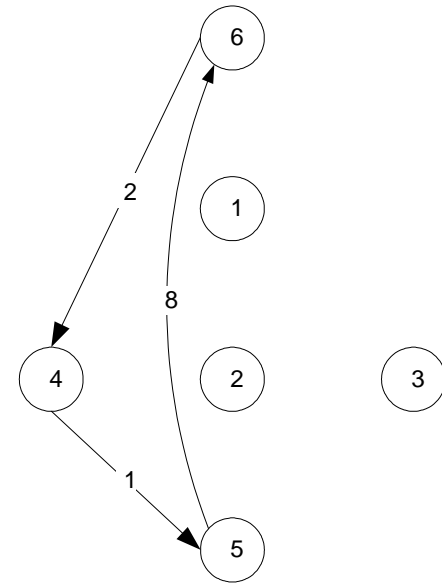


Adjacency List

1 → **6**
2 → **4**
3 → **5**
4 → **1, 5**
5 → **2, 6**
6 → **3, 4**

2nd iteration:

2 - 4 - 1 - 6 - 3 - 5 - 2



Lista de adjacências

1 →
2 →
3 →
4 → **5**
5 → **6**
6 → **4**

3th iteration:

4 - 5 - 6 - 4

Hierholzer Algorithm - Exemplification

Found cycles

1 – 2 – 3 – 1

2 – 4 – 1 – 6 – 3 – 5 – 2

4 – 5 – 6 – 4

Euler Circuit = Union of all cycles

1 – 2 – 3 – 1

2 – 4 – 1 – 6 – 3 – 5 – 2

4 – 5 – 6 – 4

Euler Circuit:

1 – 2 – 4 – 5 – 6 – 4 – 1 – 6 – 3 – 5 – 2 – 3 – 1

Graphs

Topological Sort

Topological Sort

Topological sort of a direct acyclic graph $G(V,E)$ is a **linear ordering** of all the vertices in the graph V_1, V_2, \dots, V_n such that vertex V_i comes before vertex V_j if there is an edge $(V_i, V_j) \in G$

- Topological Sort is only possible in a **Direct Acyclic Graph (DAG)**
- There can be multiple topological sorts of a graph G

Applications of Topological Sort include the following:

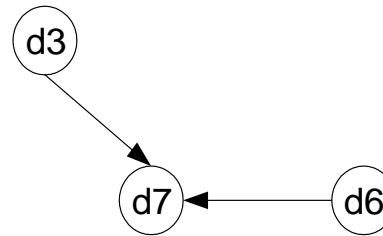
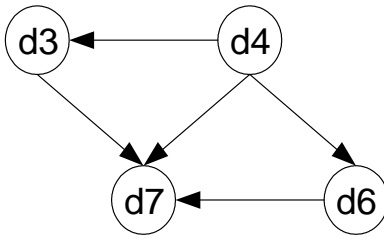
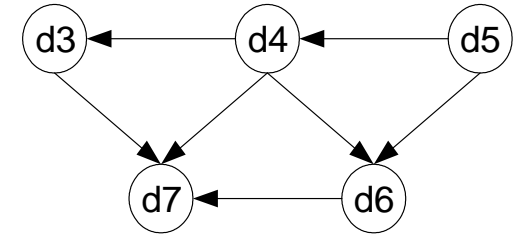
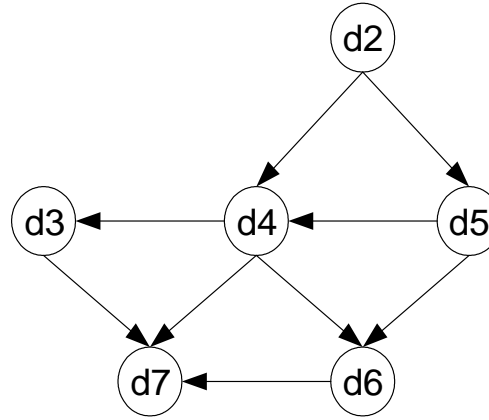
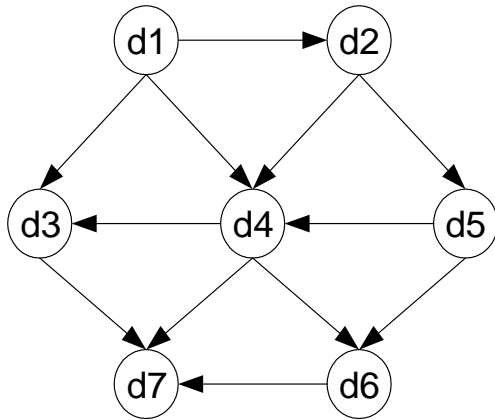
- Prerequisites between courses of an academic program
- Inheritance between classes of an object-oriented program
- Scheduling constraints between the tasks of a project

Topological Sort – Basic Idea

Lemma: All Directed Acyclic Graph (DAG) has at least one vertex with **input degree zero**

- Starts to process all vertices with the **input degree zero**
- After these vertices are processed the input degree of its adjacent vertices are decremented
- The adjacent vertices which input degree becomes zero are the next vertices to be processed
- And so on...

Topological Sort – Exemplification



The graph has two Topological Sorts:

$d_1, d_2, d_5, d_4, d_3, d_6, d_7$

$d_1, d_2, d_5, d_4, d_6, d_3, d_7$

Topological Sort – First Algorithm

```
Algorithm boolean topologicalSort (Graph<V,E> g, Deque<V> topsort) {  
  for (all V vertices in g) {  
    degreeIn[V] = g.inDegree(V)  
    if (degreeIn[V] == 0) Add vertex V to queue-aux }  
  numVerts=0  
  while (!queue-aux is Empty){  
    vOrig ← remove first vertex from queue-aux  
    add vOrig-Element to topsort  
    numVerts++  
    for (each of vOrig's outgoing edges, e = (vOrig,vAdj)){  
      degreeIn[vAdj]--  
      if (degreeIn[vAdj] == 0) add vertex vAdj to queue-aux  
    }  
  }  
  if (numVerts < g.numVertices()) //Graph has a cycle  
    return false  
  return true }  
Time Complexity (?)
```


Topological Sort – Version more simple

- Use the **DFS algorithm** to process the vertices not yet visited
- As each vertex ends (has no adjacent vertices) it is placed in a stack (which guarantees that it appears after all its predecessors)
- At the end of the algorithm, the stack contains the topological sort of the graph

Note

- This algorithm can be initiated by any vertex, regardless of its Input degree
- Therefore, it is possible to find different correct Topological Sorts for the same Directed Acyclic Graph

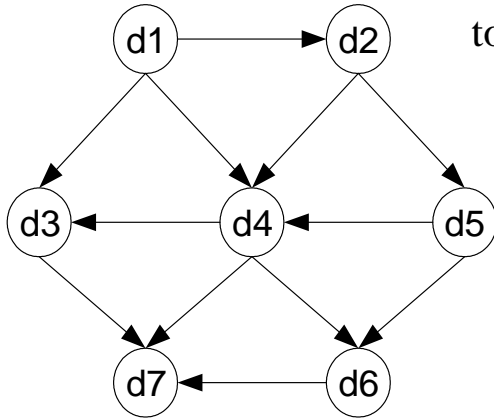
Topological Sort – Algorithm

```
Algorithm Queue<V> topologSort(Graph<V,E> g) {  
    for (all V Vertices in g) { visited[V]=false }  
    for (each of vOrig of graph g)  
        if (vertex vOrig has not been visited)  
            topologSort(G, vOrig, visited, topsort);  
    return topsort  
}
```

```
Algorithm void topologSort(Graph<V,E> g, Vertex<V,E> vOrig,  
                           boolean[] visited, Deque<V> topsort) {  
    make vOrig as visited  
    for (each of vOrig's outgoing edges, edge = (vOrig,vAdj))  
        if (!visited[vAdj])  
            topologSort(g,vAdj,visited,topsort)  
    push vOrig into topsort  
}
```

Time Complexity: $O(?)$

Topological Sort – Exemplification



topologSort (1, vector, topsort)
 topologSort (2, vector, topsort)
 topologSort (4, vector, topsort)
 topologSort (3, vector, topsort)
 topologSort (7, vector, topsort)

visited	1	1	1	1	0	0	1
	1	2	3	4	5	6	7

topsort (7)

topsort (7, 3)

Adjacency List

1 → 2, 3, 4
 2 → 4, 5
 3 → 7
 4 → 3, 6, 7
 5 → 4, 6
 6 → 7
 7 →

topologSort (6, vector, topsort)

visited	1	1	1	1	0	1	1
	1	2	3	4	5	6	7

topsort (7, 3, 6)

topsort (7, 3, 6, 4)

topologSort (5, vector, topsort)

visited	1	1	1	1	1	1	1
	1	2	3	4	5	6	7

topsort (7, 3, 6, 4, 5)

topsort (7, 3, 6, 4, 5, 2)

topsort (7, 3, 6, 4, 5, 2, 1)

Graphs

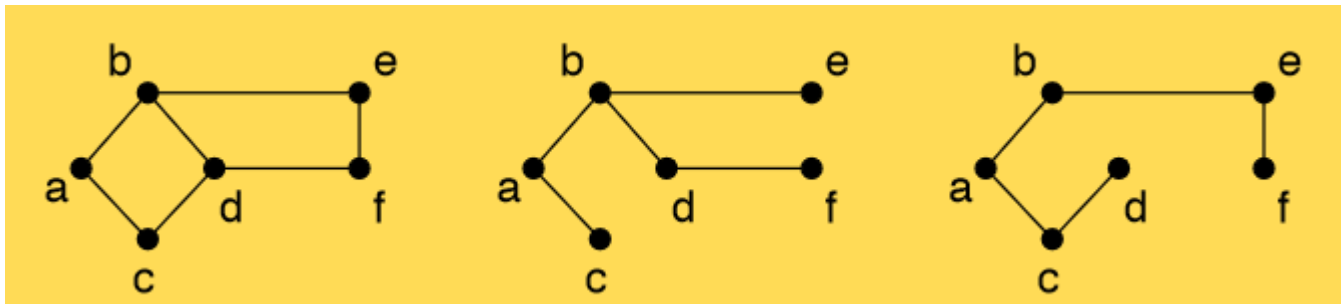
Minimum Spanning Trees

Minimum Spanning Tree (MST)

Given an undirected, connected graph $G=(V,E)$ with positive edge weights, a **minimum spanning tree** $T=(V,E')$ is a tree that connects all the vertices of the graph G with the minimum cost (weights)

$|E'| = |V|-1 \rightarrow$ The number of edges of the MST is one less than the number of vertices, so that there are no cycles

A graph can have many spanning trees, but all have V vertices and $|V|-1$ edges



Minimum Spanning Tree

MST is a fundamental problem with diverse applications

- network design: telephone, electrical, hydraulic, computer
- Cluster analysis

Two algorithms to find the MST of an undirected, connected graph:

Kruskal's algorithm

- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle

Prim's algorithm

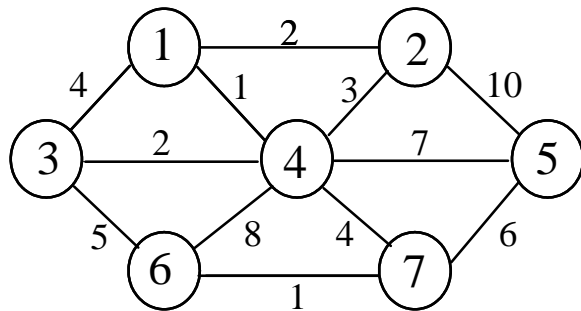
- Start with any vertex S and greedily grow a tree T from S
- At each step, add the cheapest edge to T that has exactly one endpoint in T

Kruskal's algorithm – Basic Idea

Given an undirected, connected graph $G = (V, E)$ with positive edge weights

- At the beginning each vertex is a tree
- All edges of the graph are **inserted in a vector** in order of **increasing weight**
- each edge is removed from the vector and if it joins **two vertices belonging to different trees**, it is added to the MST
- this operation is repeated until all the vertices are linked
- When the algorithm terminates there is only one tree – the **minimum spanning tree**

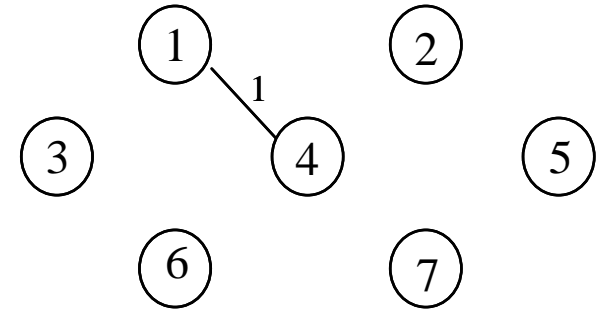
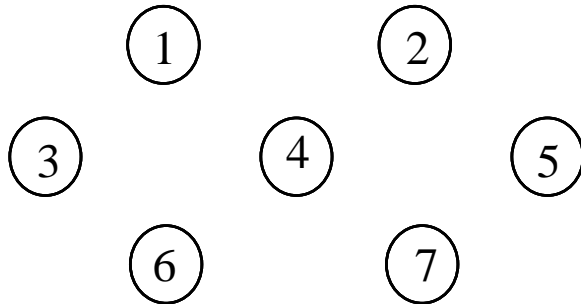
Kruskal's algorithm – Exemplification



Ordered set of edges:

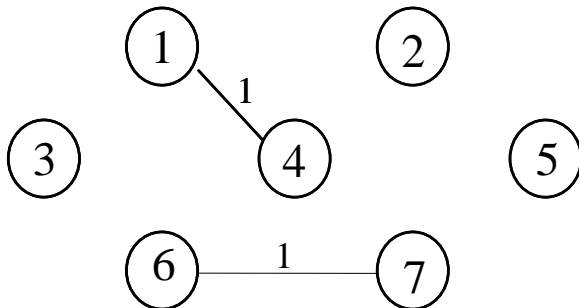
$\{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5) \}$

$Q = \{ \}$



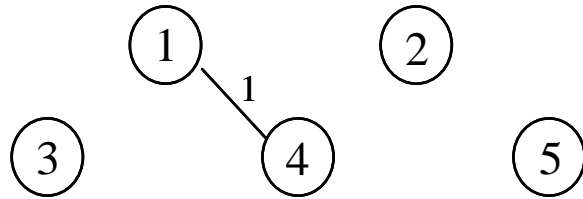
$Q = \{ \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\} \}$

$Q = \{ \{1,4\}, \{2\}, \{3\}, \{5\}, \{6\}, \{7\} \}$



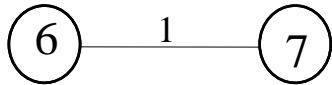
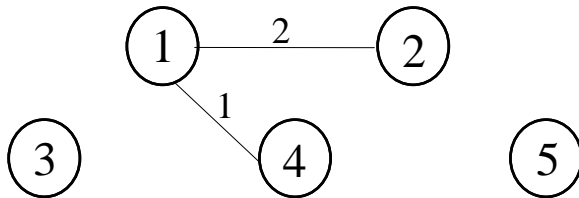
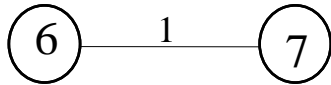
$Q = \{ \{1,4\}, \{6,7\}, \{2\}, \{3\}, \{5\} \}$

Kruskal's algorithm – Exemplification

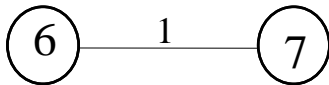
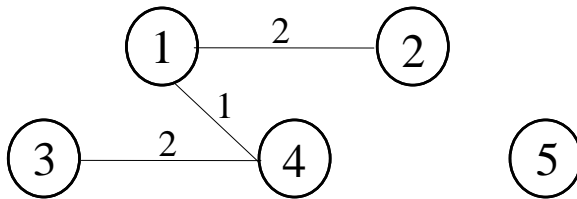


Ordered set of edges:

$\{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5) \}$

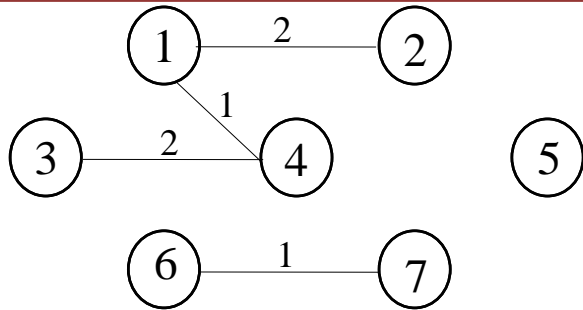


$Q = \{ \{1,4,2\}, \{6,7\}, \{3\}, \{5\} \}$



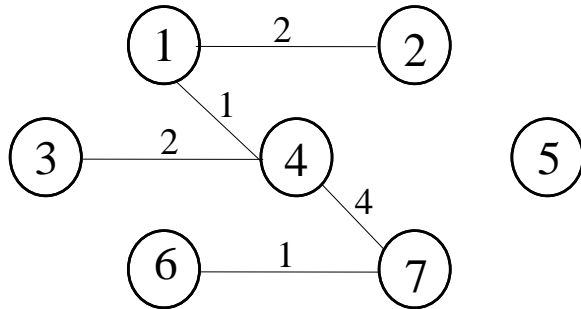
$Q = \{ \{1,4,2,3\}, \{6,7\}, \{5\} \}$

Kruskal's algorithm – Exemplification

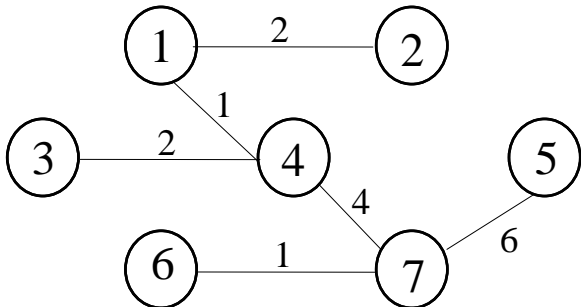


Ordered set of edges:

$\{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5) \}$

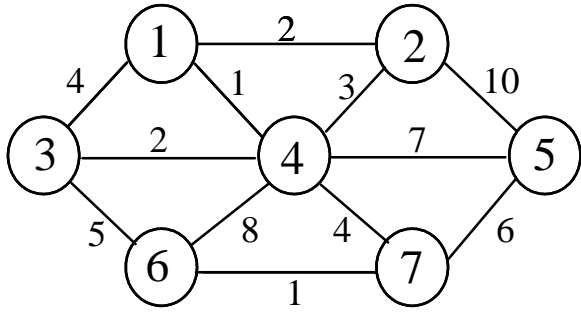


$Q = \{ \{1,4,2,3,6,7\}, \{5\} \}$



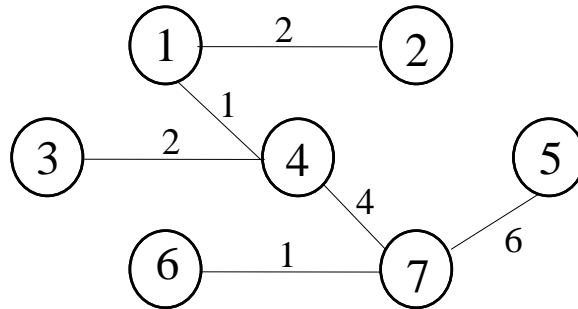
$Q = \{ \{1,4,2,3,6,7,5\} \}$

Kruskal's algorithm – Exemplification



Ordered set of edges:

$\{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5) \}$



Minimum spanning tree

$$A = \{(1,4), (7,6), (1,2), (4,3), (4,7), (5,7)\}$$

The weight of the tree is the sum of its edges $W(A) = 16$

Kruskal's – Algorithm

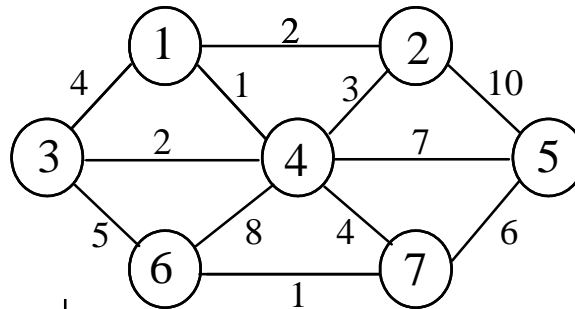
```
Algorithm Graph<V,E> kruskal1 (Graph<V,E> g) {  
    for (all V vertices in g)  
        Add vertex V to mst  
    for (all E edges in g)  
        Add edge into a vector-edges  
    Sort vector-edges by ascending order of weight  
  
    for (each Edge e=(VOrig,vAdj) of vector-edges)  
        if (numberOfPaths(g,vOrig,vAdj) == 0)  
            add e=(vOrig,vAdj) to mst  
  
    return mst;  
}
```

Time Complexity (?)

Prim's algorithm – Basic Idea

- The main idea is to perform a **breadth-first search** starting at any source vertex S
 - The algorithm uses three vectors which records for the vertices:
 - the **distance** from each vertex to its predecessor vertex (dist)
 - the **predecessor** vertex (path)
 - If the vertex has been processed (visited)
 - The algorithm starts to mark the initial vertex S with length 0
 - Then, update the weights of all its adjacent vertices not yet visited
 - Next, Choose the vertex not yet processed with less weight
- Repeat the process until all vertices have been processed

Prim's algorithm – Exemplification



Initial Vertex: 7

path	0	0	0	7	7	7	0
------	---	---	---	---	---	---	---

1 2 3 4 5 6 7

dist	∞	∞	∞	4	6	1	0
------	----------	----------	----------	---	---	---	---

1 2 3 4 5 6 7

visited	0	0	0	0	0	0	1
---------	---	---	---	---	---	---	---

1 2 3 4 5 6 7

minimum(dist)=1, V = 6

path	∞	∞	6	7	7	7	0
------	----------	----------	---	---	---	---	---

1 2 3 4 5 6 7

dist	∞	∞	5	4	6	1	0
------	----------	----------	---	---	---	---	---

1 2 3 4 5 6 7

visited	0	0	0	0	0	1	1
---------	---	---	---	---	---	---	---

1 2 3 4 5 6 7

minimum(dist)=4, V = 4

path	4	4	4	7	7	7	0
------	---	---	---	---	---	---	---

1 2 3 4 5 6 7

dist	1	3	2	4	6	1	0
------	---	---	---	---	---	---	---

1 2 3 4 5 6 7

visited	0	0	0	1	0	1	1
---------	---	---	---	---	---	---	---

1 2 3 4 5 6 7

minimum(dist)=1, V = 1

path	4	1	4	1	7	7	0
------	---	---	---	---	---	---	---

1 2 3 4 5 6 7

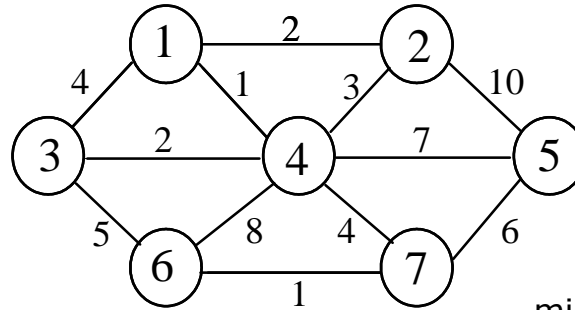
dist	1	2	2	1	6	1	0
------	---	---	---	---	---	---	---

1 2 3 4 5 6 7

visited	1	0	0	1	0	1	1
---------	---	---	---	---	---	---	---

1 2 3 4 5 6 7

Prim's algorithm – Exemplification



minimum(dist)=2, $V = 2$

$V = 2 \rightarrow$ doesn't change dist vector

path	4	1	4	1	7	7	0
	1	2	3	4	5	6	7

dist	1	2	2	1	6	1	0
	1	2	3	4	5	6	7

visited	1	1	0	1	0	1	1
	1	2	3	4	5	6	7

minimum(dist)=2, $V = 3$

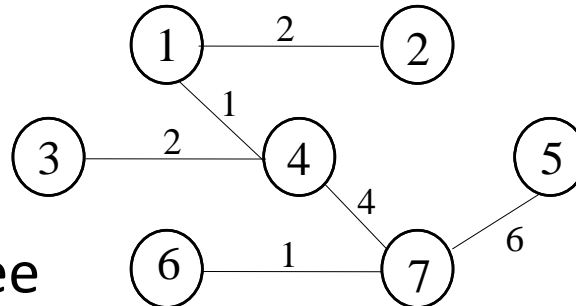
Adj[3] = {1, 4, 6} \rightarrow already processed
 \rightarrow doesn't change dist vector

visited	1	1	1	1	0	1	1
	1	2	3	4	5	6	7

minimum(dist)=6, $V = 5$

Adj[5] = {2, 4, 7} \rightarrow already processed
 \rightarrow doesn't change dist vector

visited	1	1	1	1	1	1	1
	1	2	3	4	5	6	7



Minimum Spanning Tree

Prim's Algorithm

```
Algorithm void prim(Graph<V,E> g, Vertex<V,E> vOrig, Graph<V,E> mst){  
  
    for (all V vertices in g) {dist[V]= $\infty$  path[V]=-1 visited[v]=false }  
    dist[vOrig]=0  
    while (vOrig != -1){  
        make vOrig as visited  
        for (each of vOrig's outgoing edges, edge = (vOrig,vAdj))  
            if (!visited[vAdj] && dist[vAdj]>dist[vOrig]+edge.getWeight()){  
                dist[vAdj] = dist[vOrig]+edge.getWeight()  
                path[vAdj] = vOrig  
            }  
        vOrig = getVertMinDist(dist, visited)  
    }  
    mst=buildMst(path,dist)  
}
```

Time Complexity: $O(?)$

Graphs

Project Scheduling: PERT/CPM

Project scheduling techniques – PERT/CPM

Critical Path Method (CPM) and Program Evaluation Review Technique (PERT) are two management techniques developed in the late 1950s to plan, schedule, and control **large, complex projects** with many activities

These approaches differ primarily on how the duration and the cost of activities are processed. In the case of CPM were assumed to be **deterministic** while in PERT these are described **probabilistically**

Both approaches use a network representation - **graph** to display the relationships between project activities

Questions addressed by PERT/CPM

PERT/CPM help to address questions such as:

1. What is the total time required to complete the project (the expected total time for PERT)?
2. What are the start and the completion times for individual activities?
3. How much delay can be tolerated for non-critical activities without incurring a delay in the estimated project completion time?
4. Which critical activities must be completed as scheduled to meet the estimated project completion time?

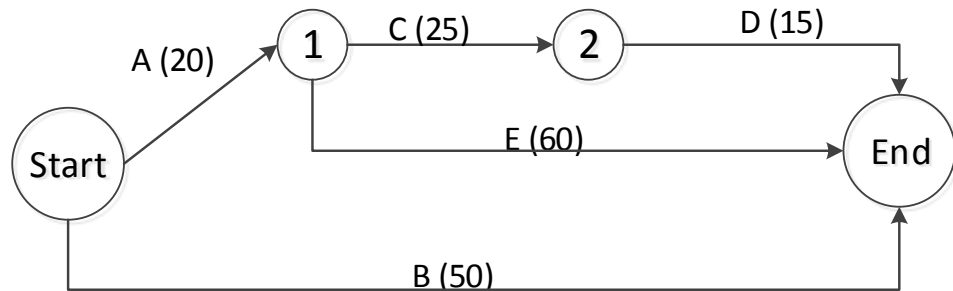
PERT/CPM Graph

- has only one vertex source where the project begins
- has only one final vertex, where the project ends
- has no cycles
- vertices may represent the activities and the edges their precedence relationships - **Activity on node (AON)**
- edges may represent the activities and vertices represent milestones in time - the starting or the completion of activities - **Activity on arc (AOA)**
- An activity can only start when all its previous have finalized

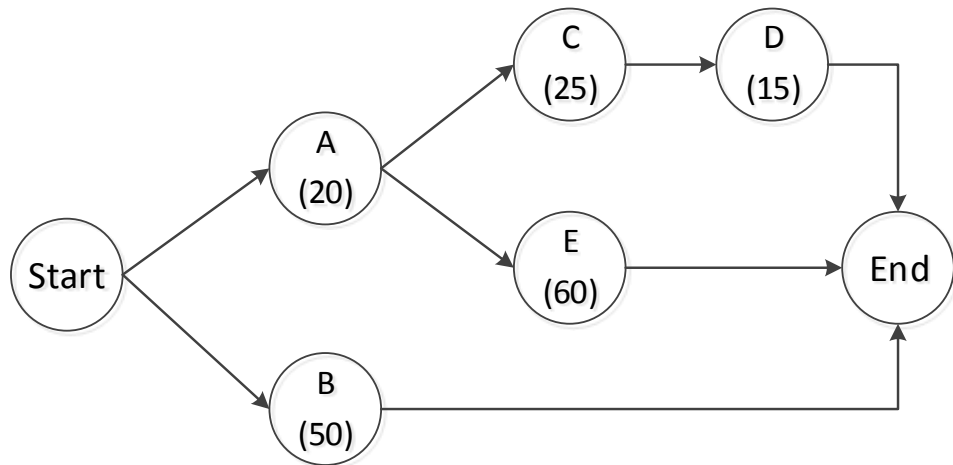
Project network representation

Activity	A	B	C	D	E
Immediate Predecessor	—	—	A	C	A
Duration (days)	20	50	25	15	60

Activity on arc (AOA)



Activity on node (AON)



The Critical Path and Activity Slack

The **length of a path** from the start node to the finish node is the **sum of the (estimated) durations of the activities** on the path

The (estimated) **project duration** or project completion time equals the **length of the longest path** through the project network - this **longest path** is called the **critical path**

If more than one path tie for the longest, they are **all critical paths**

Activity Slack is the amount of time that noncritical activities can be delayed without increasing the project completion time

The **critical path** will always consist of **activities with zero slack**

Earliest start and earliest finish time of an activity

The PERT/CPM scheduling procedure begins by addressing the question:

- When can the individual activities start and finish (at the earliest) if no delays occur?

The starting and finishing times of each activity if no delays occur anywhere in the project are called the **earliest start time** and the **earliest finish time** of the activity

- ES = earliest start time for a particular activity
- EF = earliest finish time for a particular activity

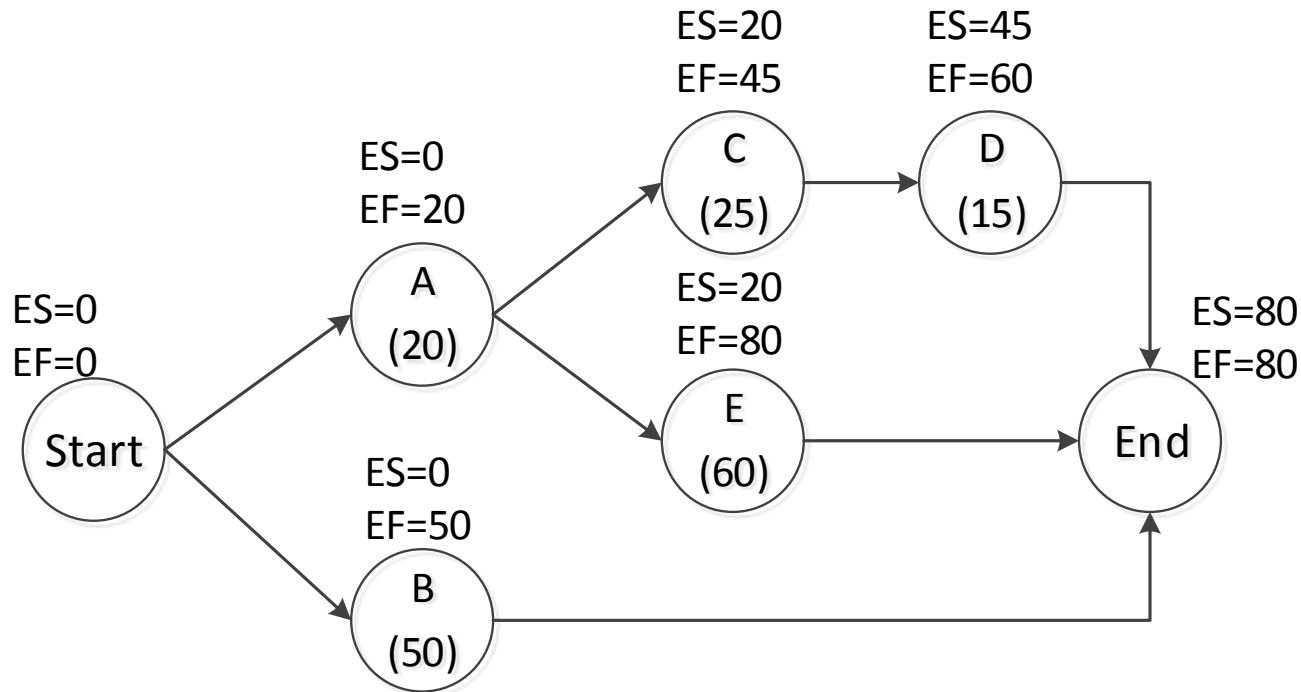
where

$$EF = ES + (\text{estimated}) \text{ duration of the activity}$$

Earliest start rule

- The earliest start time of an activity is equal to the *largest* of the earliest finish times of its immediate predecessors

$$ES_j = \max(EF_i) \quad i = \text{all of the immediate predecessors of } j$$



ES and EF are calculated in a **forward pass** (from the start node to the end node)

Questions addressed by PERT/CPM

PERT/CPM help to address questions such as:

1. What is the total time required to complete the project (the expected total time for PERT)? **80 days**
2. What are the start and the completion times for individual activities? **A(ES=0;EF=20) B(ES=0;EF=50) C(ES=20;EF=45) D(ES=45;EF=60) E(ES=20;EF=80)**
3. **How much delay can be tolerated for non-critical activities without incurring a delay in the estimated project completion time?**
4. Which critical activities must be completed as scheduled to meet the estimated project completion time?

Latest start and latest finish time of an activity

The next question the PERT/CPM procedure focuses on is:

- How much later can an activity start or finish without delaying project completion

The **latest start time** for an activity is the latest possible time that it can start without delaying the completion of the project, assuming no subsequent delays in the project

The **latest finish time** has the corresponding definition with respect to finishing the activity

- LS = latest start time for a particular activity
- LF = latest finish time for a particular activity

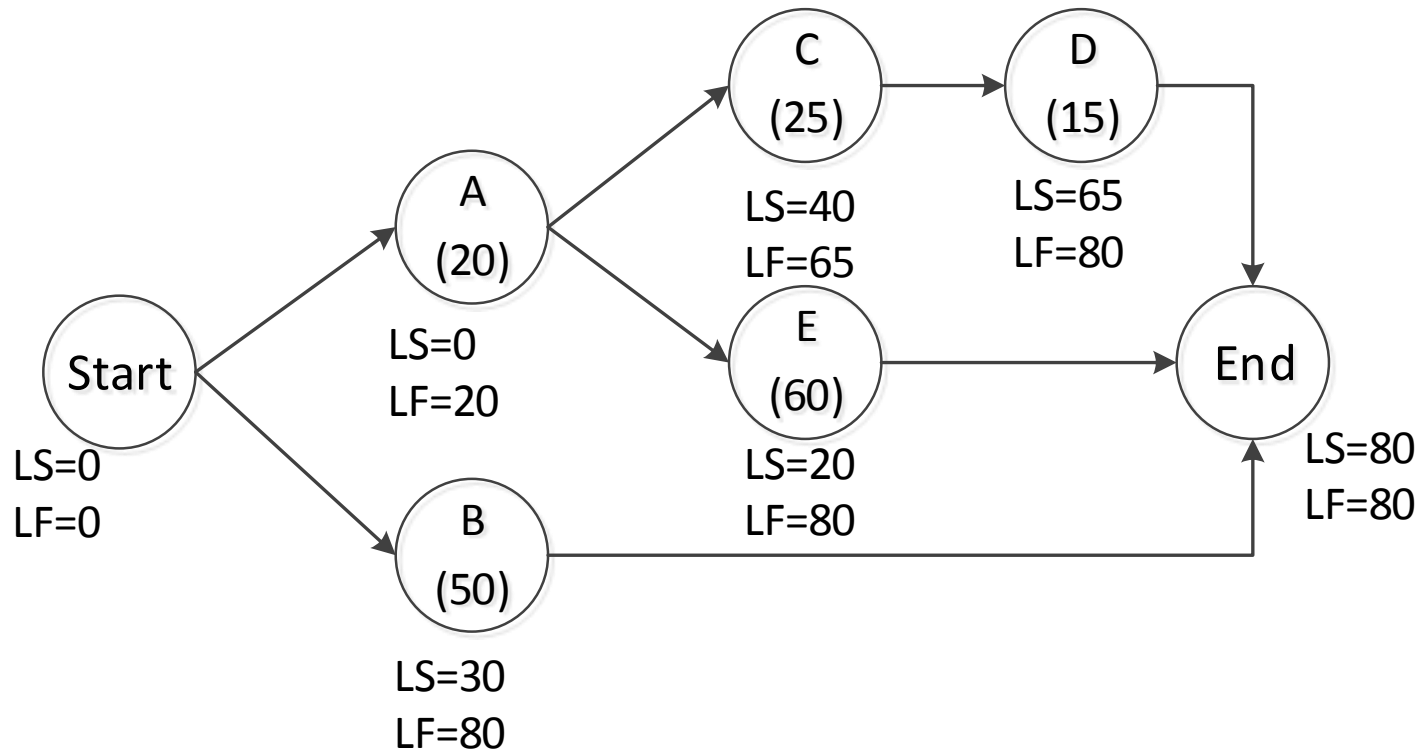
where

$$LS = LF - (\text{estimated}) \text{ duration of the activity}$$

Latest finish rule

- The latest finish time of an activity is equal to the *smallest* of the latest start times of its immediate successors. In symbols

$$LF_j = \min(LS_i) \quad i = \text{all of the immediate successors of } j$$



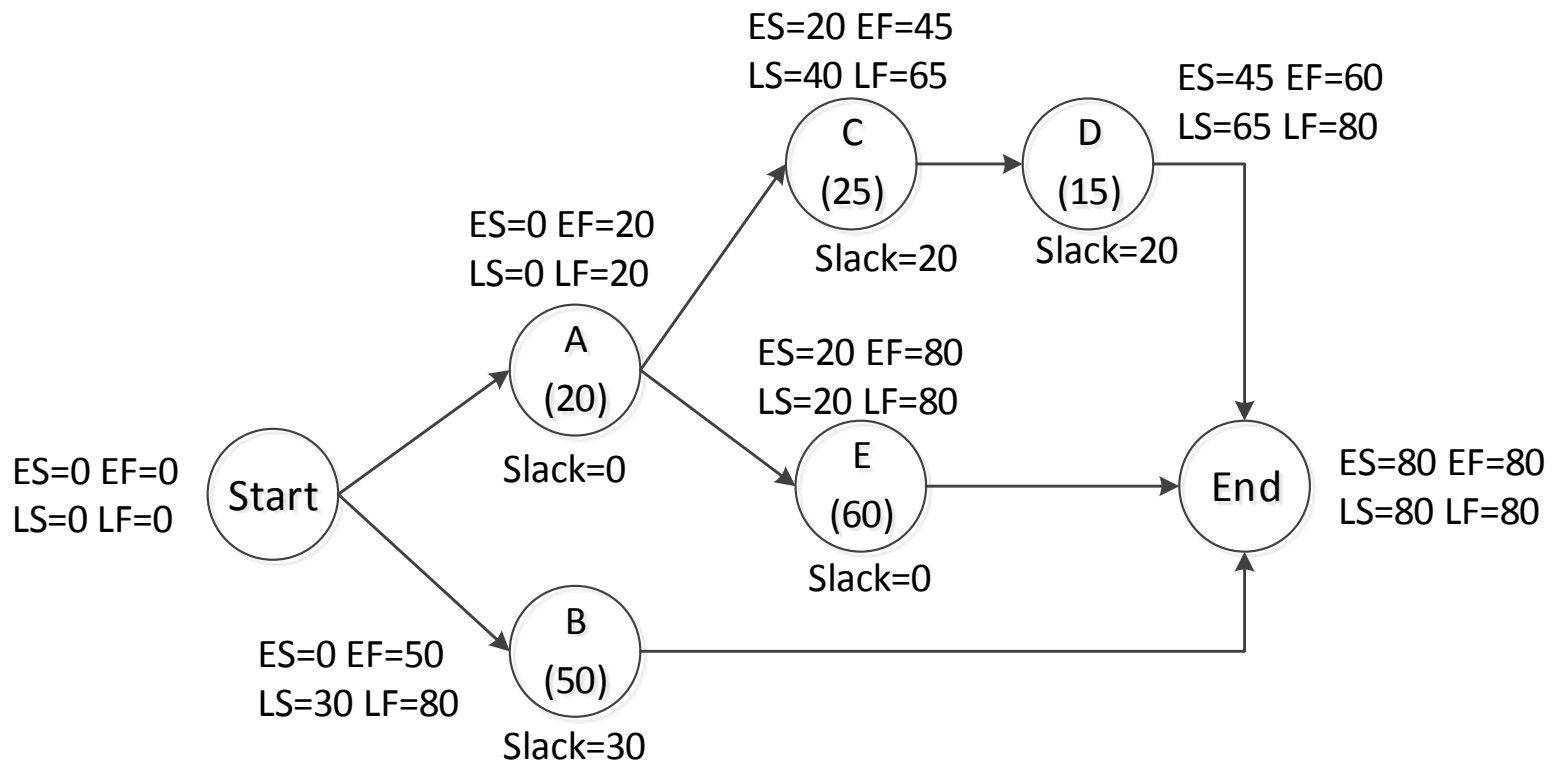
LS and LF are calculated in a **backward pass** (from the end node to the start node)

Slack

The **slack** for an activity is the difference between its latest finish time and its earliest finish time

$$\text{Slack} = \text{LF} - \text{EF}$$

Since $\text{LF} - \text{EF} = \text{LS} - \text{ES}$, either difference can be used to calculate slack



Questions addressed by PERT/CPM

PERT/CPM help to address questions such as:

1. What is the total time required to complete the project (the expected total time for PERT)? 80 days
2. What are the start and the completion times for individual activities?
A(ES=0;EF=20) B(ES=0;EF=50) C(ES=20;EF=45) D(ES=45;EF=60)
E(ES=20;EF=80)
3. How much delay can be tolerated for non-critical activities without incurring a delay in the estimated project completion time?
B(slack=30) C(slack=20) D(slack=20)
4. Which critical activities must be completed as scheduled to meet the estimated project completion time?
A(slack=0) E(slack=0)