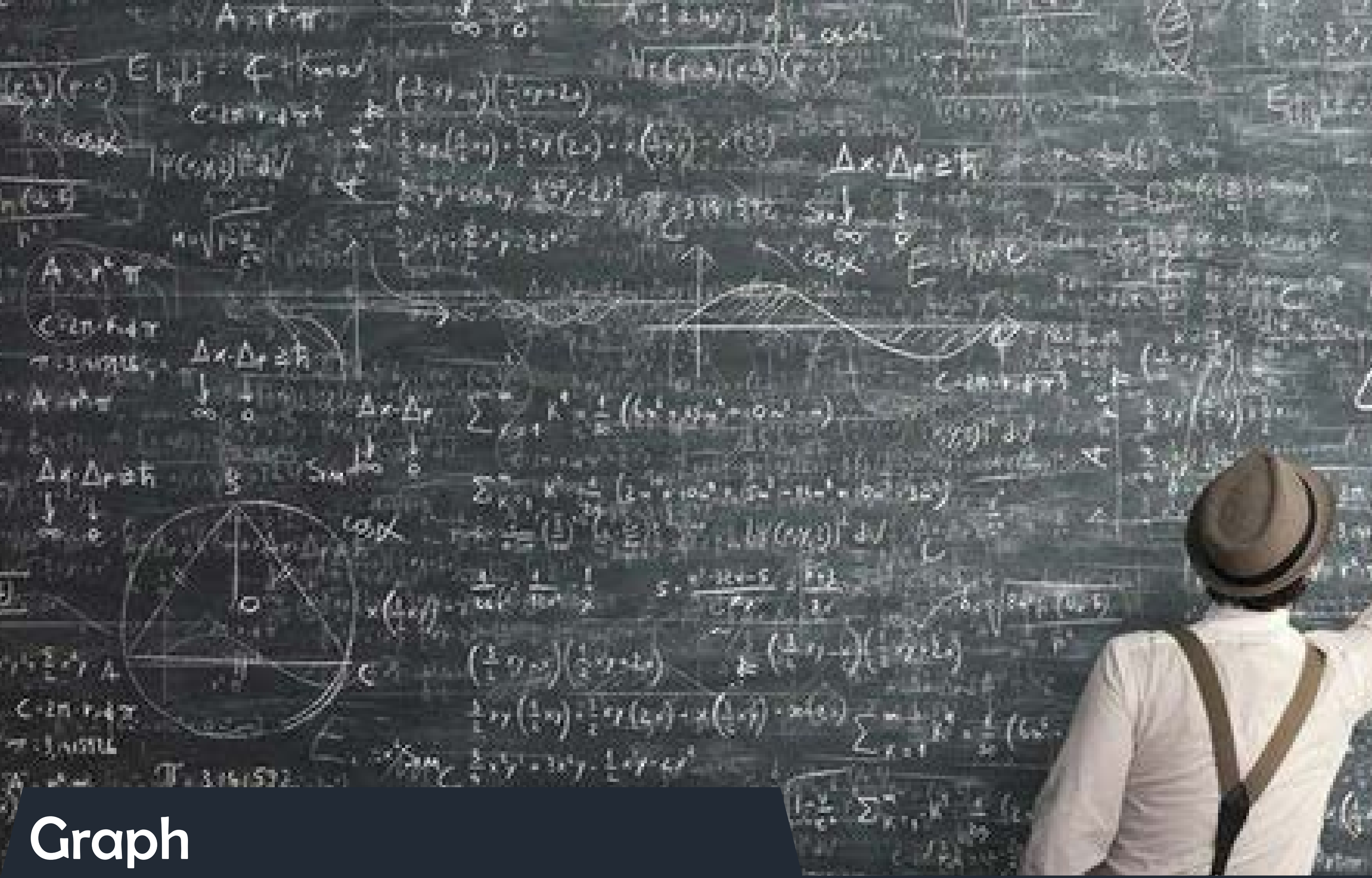
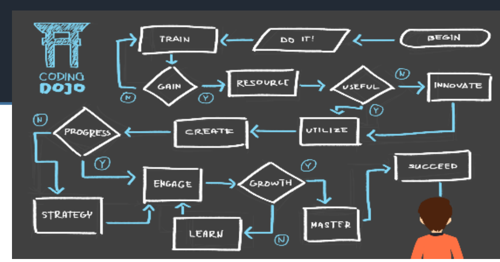


Graph

Ja-Hee Kim

Agenda

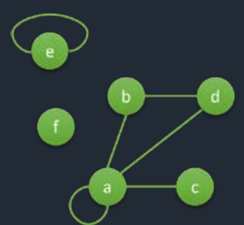


Terminology

- Graph
- Digraph

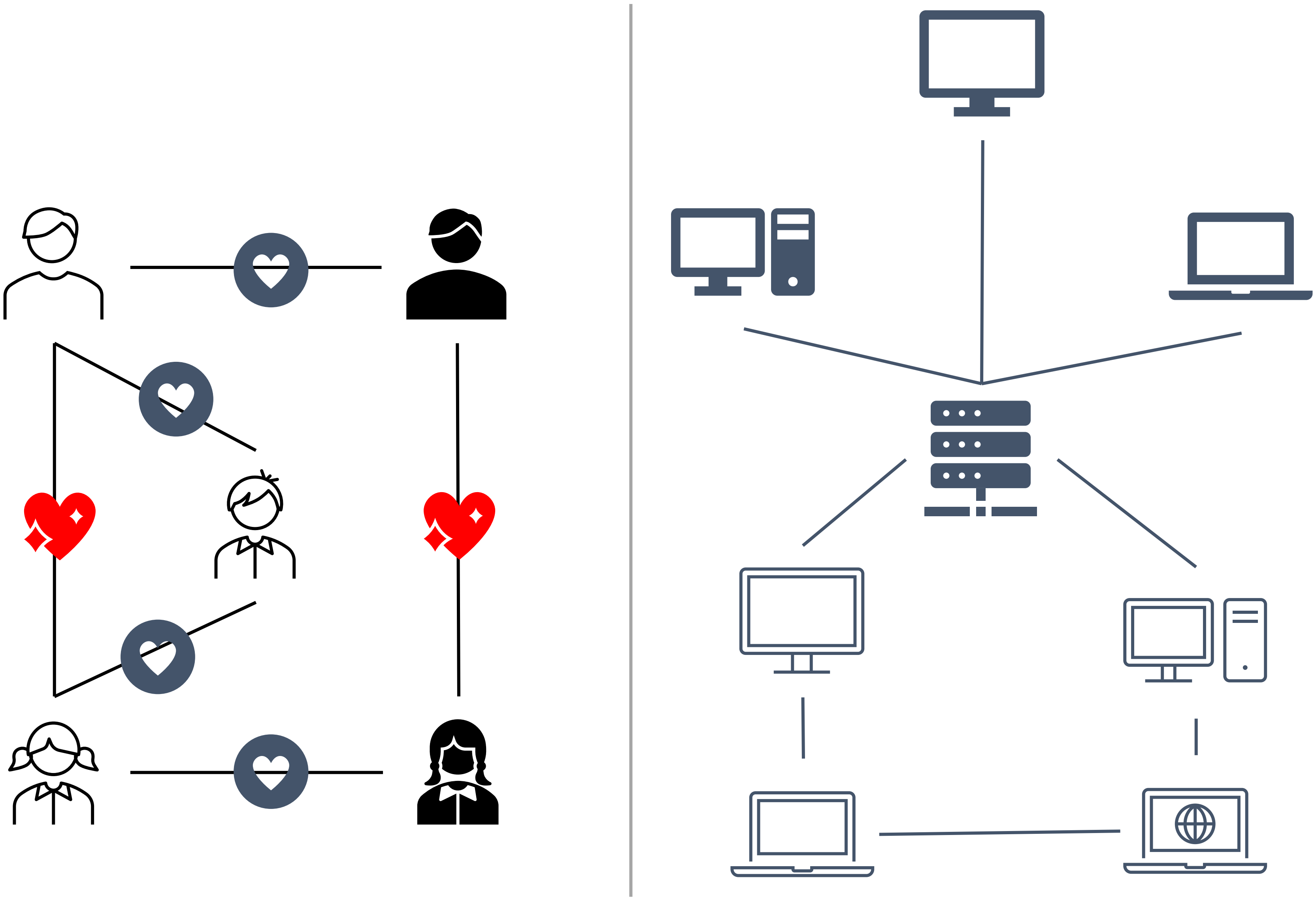
Algorithms

Group activity



Introduction to Graph

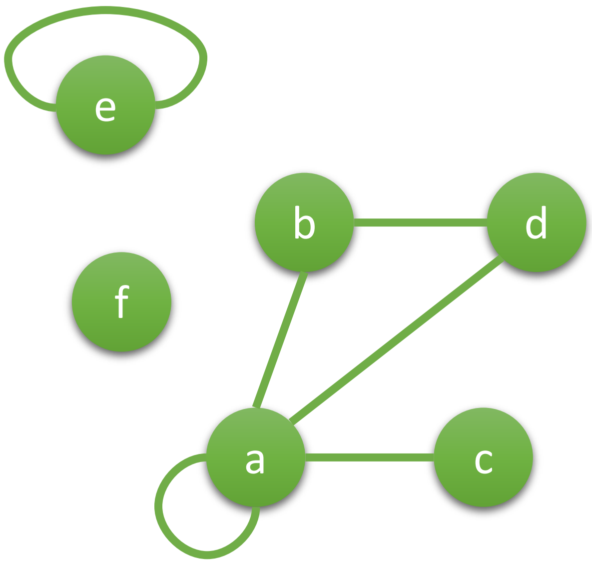
Example of Graph



Graph

- An **undirected graph**, **or graph** is a couple $G = (V, E)$ consists of
  - V: a nonempty set of vertices
  - E: a set of **unordered pairs** of distinct elements of V called edges.
  - For each $e \in E$, $e = \{u, v\}$ where $u, v \in V$.

- Example
  - V={a, b, c, d, e, f}
  - E = {{a, a}, {a, b}, {a, c}, {a, d}, {b, d}, {e, e}}

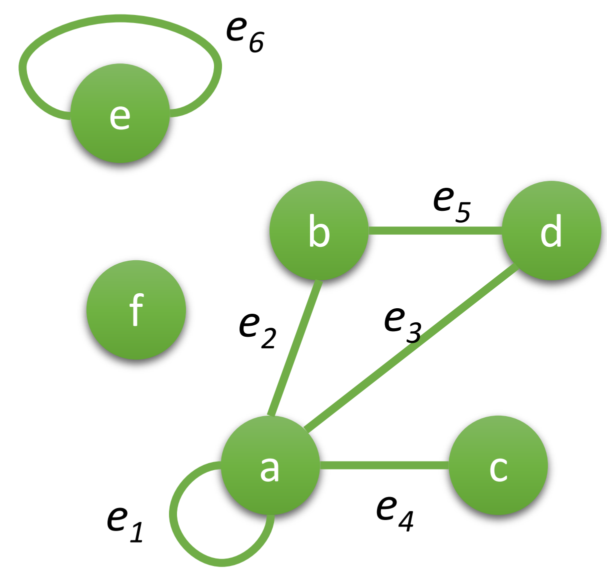


Terminology

- Ends of edge
- Adjacent, Incident
- Loop, link, simple graph
- Degree, pendant, isolated
- Walk, trail, path, cycle, circuit
- Connectivity
- Tree
- Identical, isomorphic
- Complete graph
- Subgraph
- Weighted graph

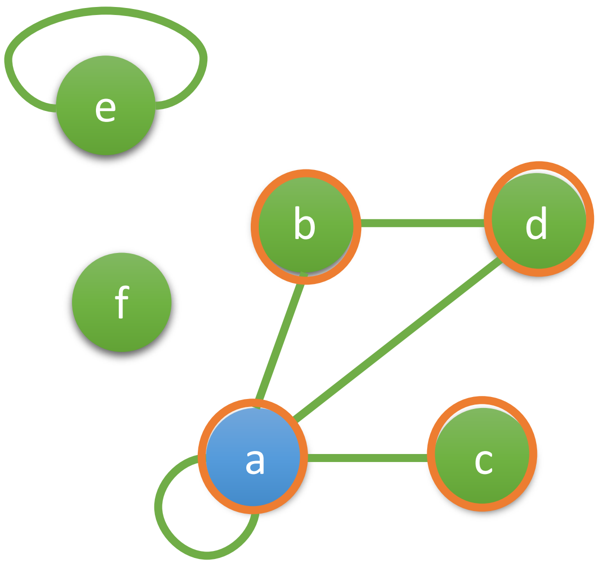
Ends of edge

- Condition
  - e is an edge {u, v}
  - u, v: vertices
- Definition
  - Join: e is said to join u and v
  - End: u and v are called the ends of e.
- Example
  - e_1 joins a and a
  - The ends of e_3 are a and d.



adjacent and incident

- Two vertices u and v in an undirected graph *G* are called **adjacent** (or **neighbors**) in *G* if *{u, v}* is an edge in *G*.
- If e = {u, v}, the edge e is called **incident with** the vertices u and v. The edge e is also said to **connect** u and v.

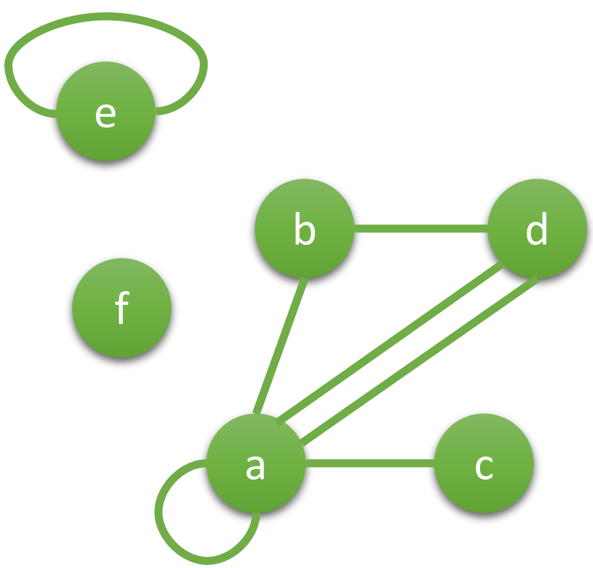


- Ex: adjacent of *a*
- Edge *{a,d}* incidents with *a* and *d*
- Edge *{a,d}* connects *a* and *d*
- *a* and *d* are the end points or ends of edge *{a,d}*

Loop

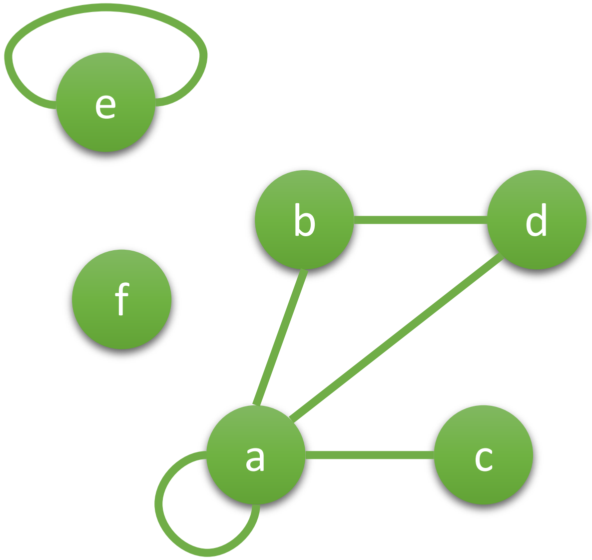
- If there is an edge incidenting to itself, this is a **loop**.
  - $e=(u,u) \in E$
- An edge with distinct eds is called as a **link**.
- Question: How many does this graph have a loop?

- A **simple graph** a graph with no loop and no multiple edges with the same ends.



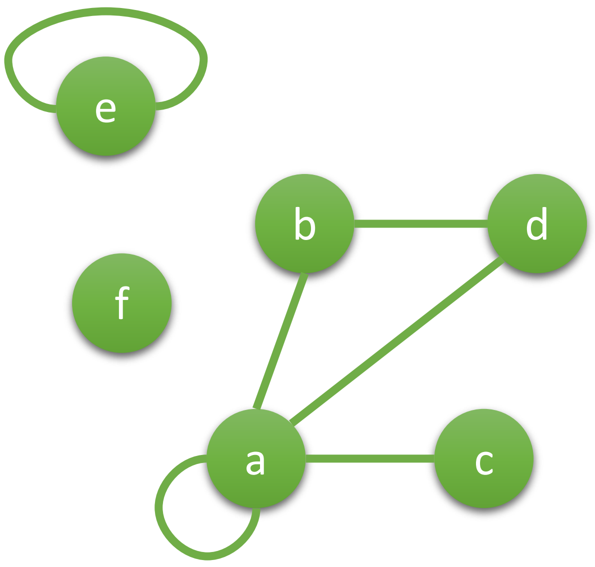
degree for an undirected graph

- The **degree** of a vertex in an undirected graph is the number of edges incident with it
  - a loop at a vertex contributes twice to the degree of that vertex
- **counting the lines** that touch it
- denoted ***deg(v)***
- Question
  - deg(a):
  - deg(b):
  - deg(c):
  - deg(f):



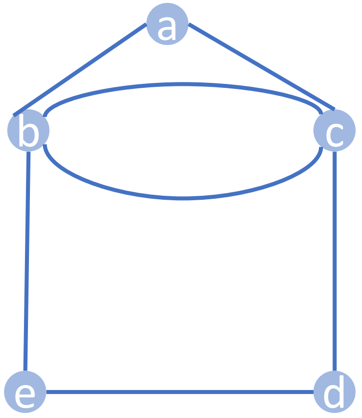
pendant and isolated

- pendant:
  - A vertex of degree 1 is called **pendant**. It is adjacent to exactly one other vertex.
- isolated:
  - A vertex of degree 0 is called **isolated**, since it is not adjacent to any vertex.
- Question
  - Which is a pendant? c
  - Which is isolated? f



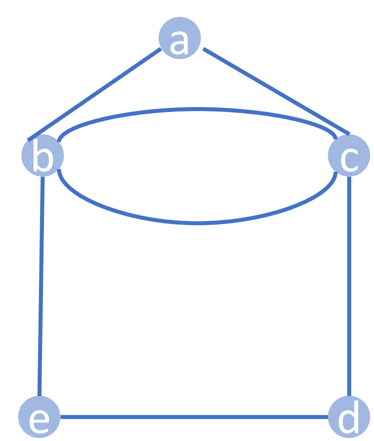
Walk

- A walk is a finite or infinite sequence of edges which joins a sequence of vertices.[2]
- Let G = (V, E) be a graph. A finite walk is a sequence of edges $(e_1, e_2, \ldots, e_{n-1})$
- This walk is closed if $v_1 = v_n$, and open else.
- A trail is a walk in which all edges are distinct.



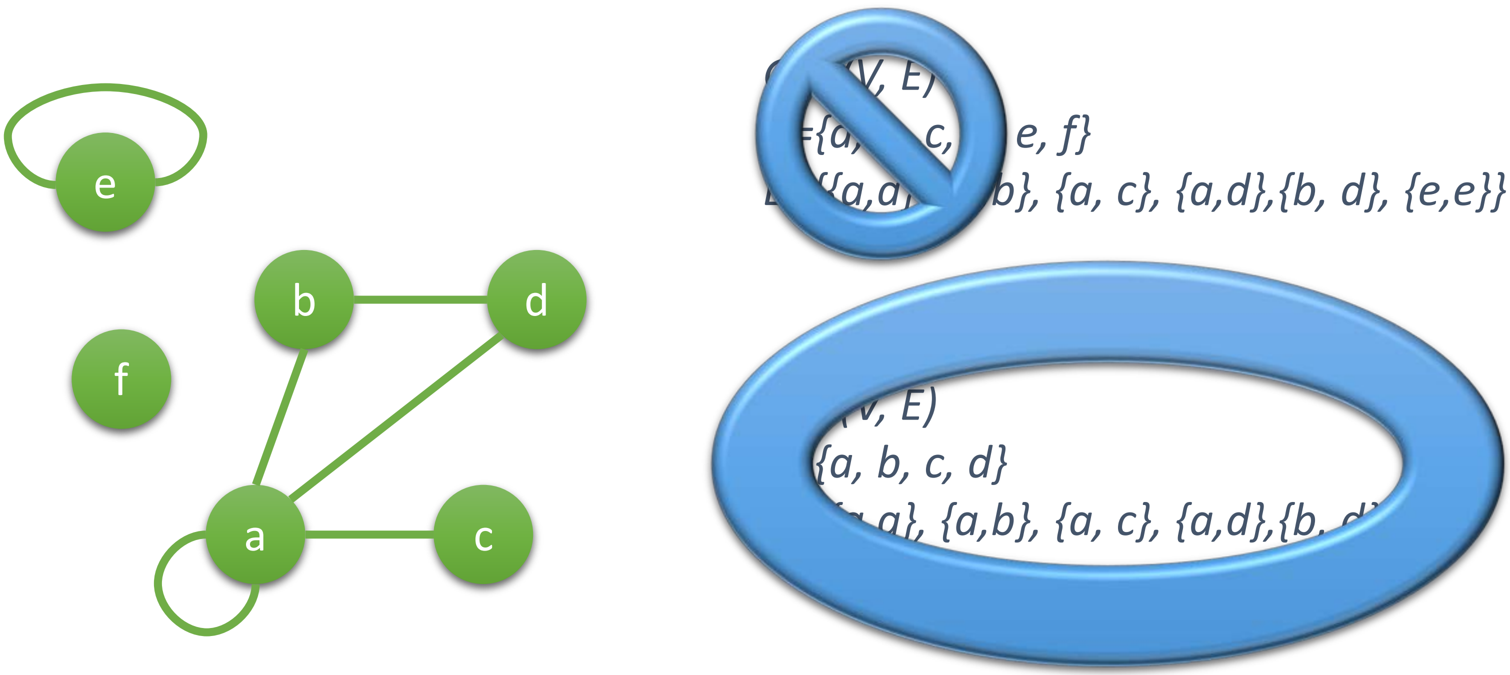
Path

- A **path** of length n from u to v, where n is a positive integer, in an **undirected graph** is a sequence of edges $e_1, e_2, \ldots, e_n$ of the graph such that $e_1 = \{x_0, x_1\}$, $e_2 = \{x_1, x_2\}, \ldots, e_n = \{x_{n-1}, x_n\}$, where $x_0 = u$ and $x_n = v$. The path is a **circuit(cycle)** if it begins and ends at the same vertex, that is, if $\boldsymbol{u = v}$.
- A path or cycle is **simple** if it does not contain the same vertex more than once.



Connectivity

- An undirected graph is called **connected** if there is a path between every pair of distinct vertices in the graph.

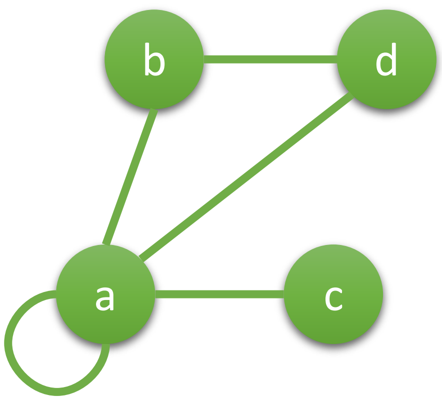


Tree

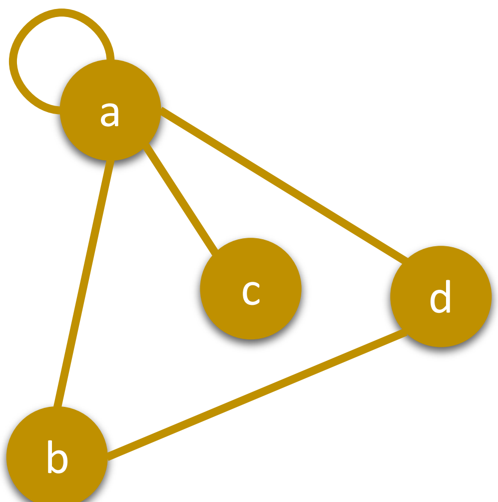
- a **tree** is an undirected graph in which
  - any two vertices are connected by exactly one path
  - or equivalently **a connected acyclic** undirected graph.
- A **forest** is an undirected graph in which
  - any two vertices are connected by at most one path
  - equivalently an **acyclic** undirected graph, or equivalently a disjoint union of trees.

Identical vs isomorphism

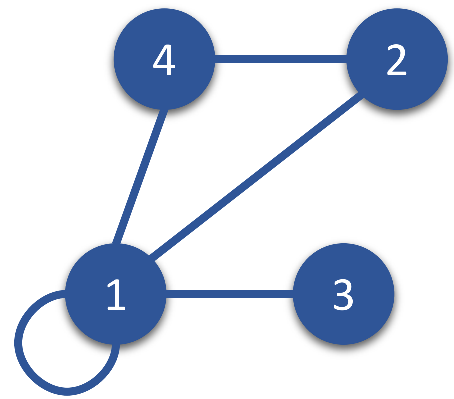
- Two graphs G and H are **identical** if V(G) = V(H) and E(G)=E(H)
- If there is a mapping V(G)→V(H) and E(G)→E(H), we say the mapping is an **isomorphism** between G and H



G1



G2

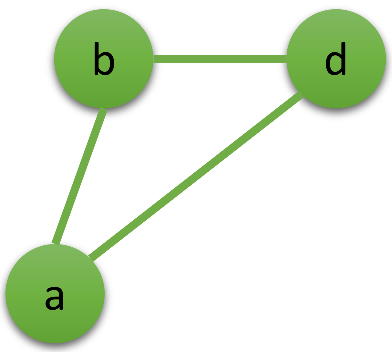


G3

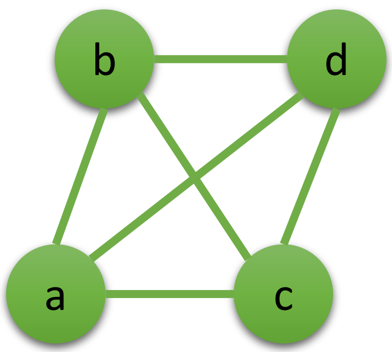
complete graph

- The **complete graph** on *n* vertices, denoted by K_n, is the simple graph that contains exactly one edge between each pair of distinct vertices.

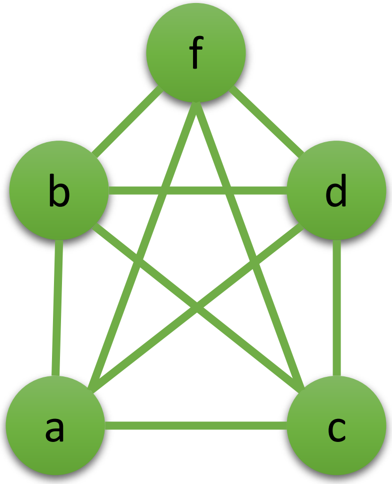
- An **empty graph** is one with no edge.



K_3



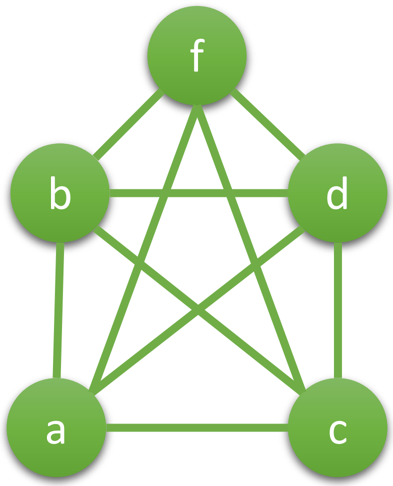
K_4

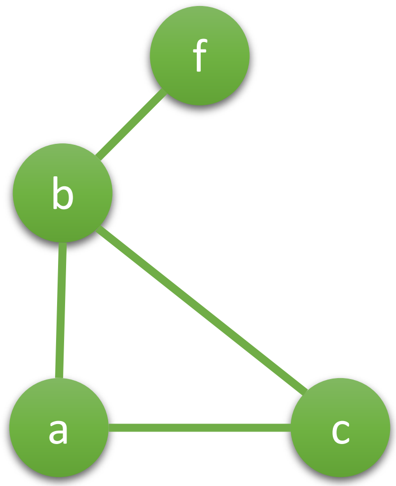


K_5

subgraph

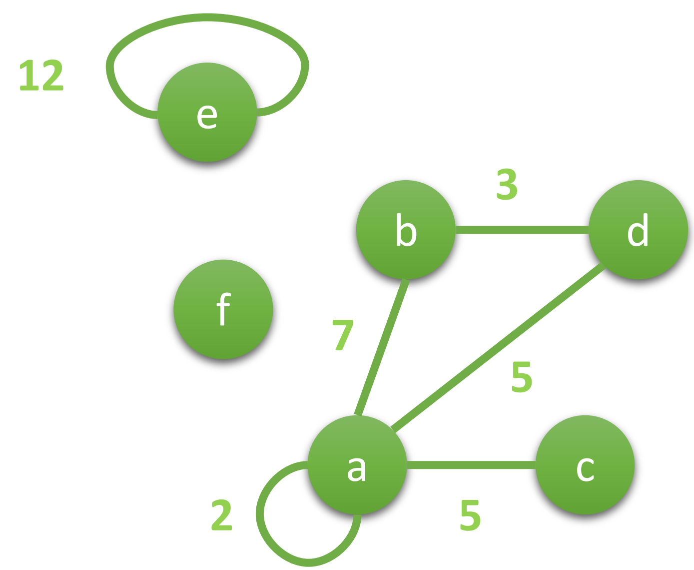
- A **subgraph** of a graph $G = (V, E)$ is a graph $H = (W, F)$ where $W \subseteq V$ and $F \subseteq E$. Of course, H is a valid graph, so we cannot remove any endpoints of remaining edges when creating H.

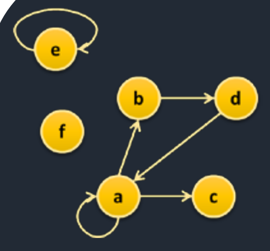




weighted graph

- A **weighted graph** is a graph in which a number (the weight) is assigned to each edge.
  - weights might represent for example costs, lengths or capacities, depending on the problem at hand.
  - *G = (V, E, W)*
    - *W:E →Z, where Z is a real number.*





Digraph

Motivation

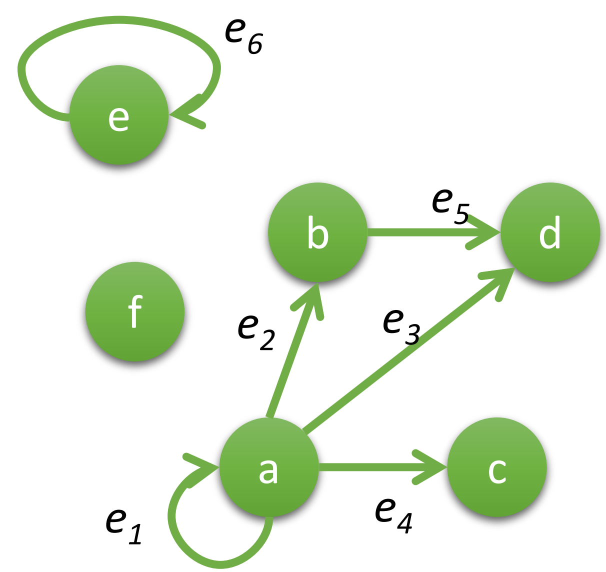
- An edge sometimes needs a direction.

Terminologies

- directed graph, digraph
- subdigraph
- underlying graph
- directed walk, trail, path
- degree
- connectiviy
- symmetric digraph

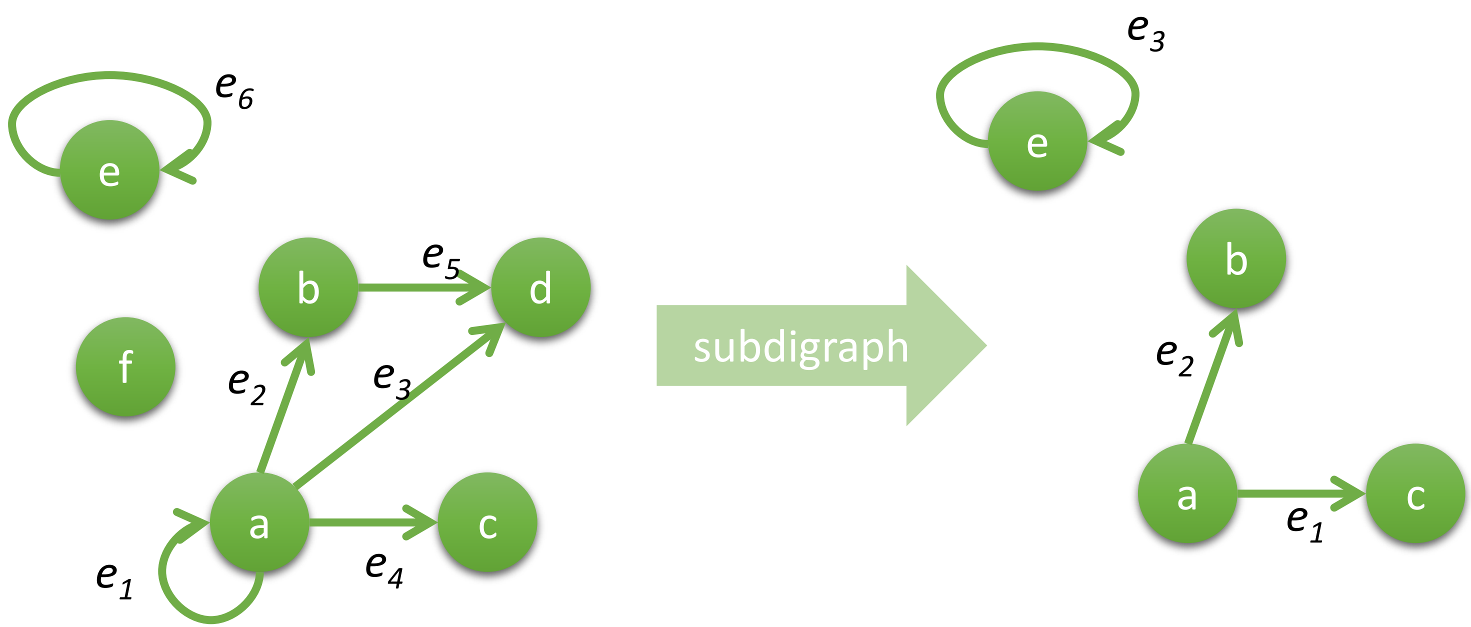
Directed graph

- **Directed graph (Digraph)** D is an ordered pair $(V(D), A(D))$
  - $V(D)$: a nonempty set of vertices
  - $A(D)$: a set of **arc**s. Each arc of A is an ordered pair of vertices of D
- If a is an arc and u and v are vertices of a: (u, v)
  - u: **tail** of a
  - v: **head** of a



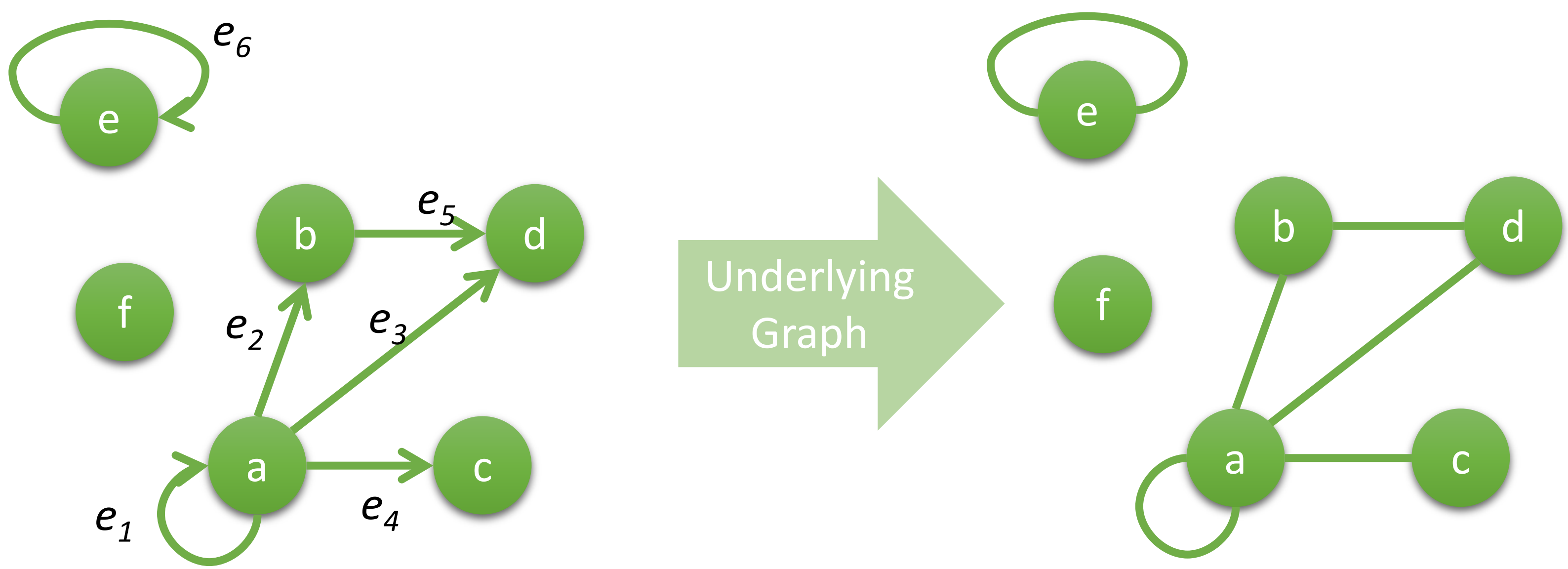
subdigraph

- A digraph D' is a **subdigraph** of D if $V(D') \subseteq V(D)$, $A(D') \subseteq A(D)$ Of course, D' is a valid graph, so we cannot remove any endpoints of remaining edges when creating D'.



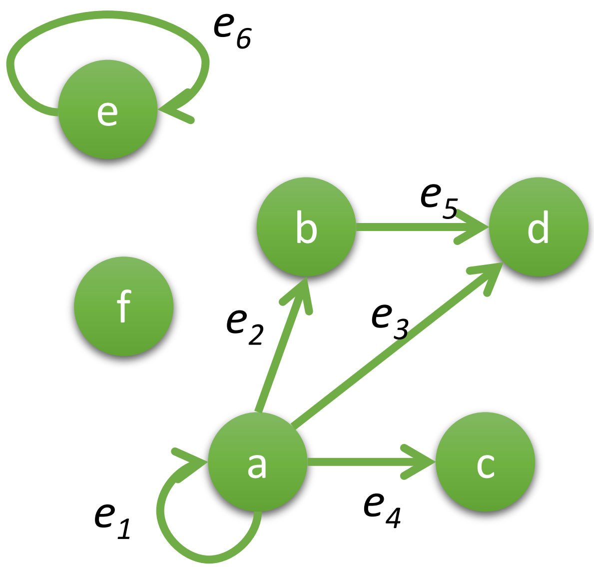
Underlying graph

- The **underlying graph** *G* of a digraph *D*
  - With each digraph *D* we can associate a graph *G* on the same vertex set;
  - corresponding to each arc of *D* there is an edge of *G* with the same ends.
  - *D* is an **orientation** of *G*



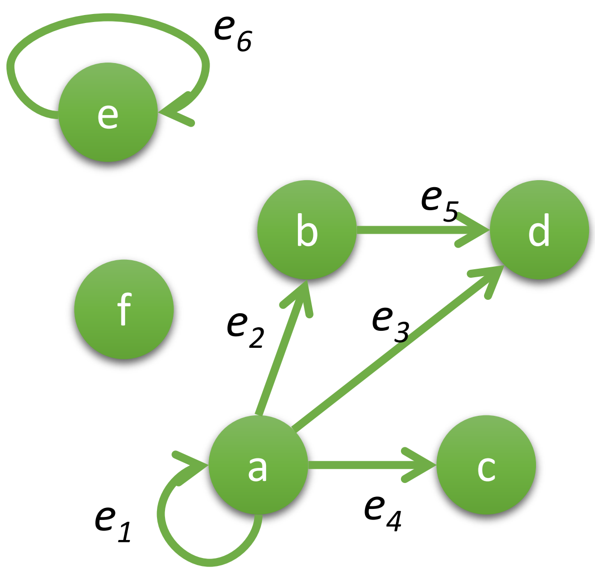
Directed walk/trail/path

- **Directed walk**: a sequence of its vertices
- **Directed tail**: directed walk that is a trail
- **Directed path, directed cycle**
- If there is a directed *(u,v)*-path in *D*, vertex *v* is said to be **reachable** from vertex *u*



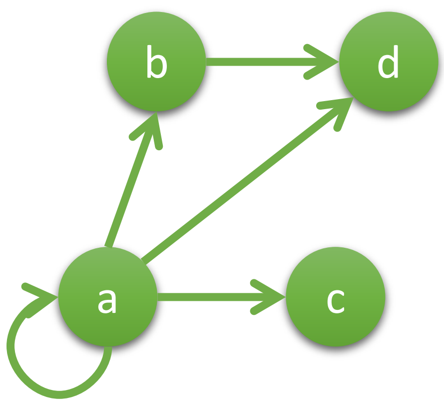
degree

- **Indegree**:
  - Denoted as $deg^{-}(v)$
  - the number of arcs with head v
- **Outdegree**
  - Denoted as $deg^{+}(v)$
  - The number of arc with tail v
- Example
  - $deg^{-}(a)$:
  - $deg^{+}(a)$:



Connectivity

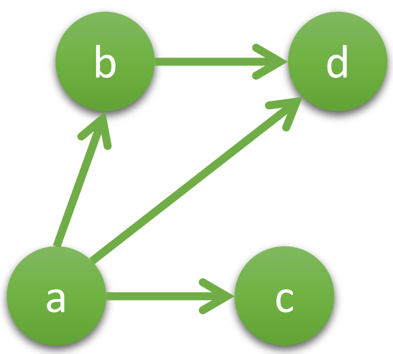
- A directed graph is called **strongly connected** if there is a path in each direction between each pair of vertices of the graph.
- A directed graph is called **weakly connected** if there is a path in each direction between each pair of vertices of its underlying graph.



Strongly connected or weakly connected?

Symmetric digraph

- **Symmetric directed graphs** are directed graphs where all edges are bidirected
  - = undirected graph
- **Simple directed graphs** are directed graphs that have no loops and no multiple arrows with same source and target nodes.



A symmetric digraph or a simple directed graph?

Rooted tree

- A rooted tree is a directed acyclic graph.
- Two kinds of tree
  - In tree: edges point away from the root
  - Out tree: edges point towards the root
- A rooted forest is a disjoint union of rooted trees.

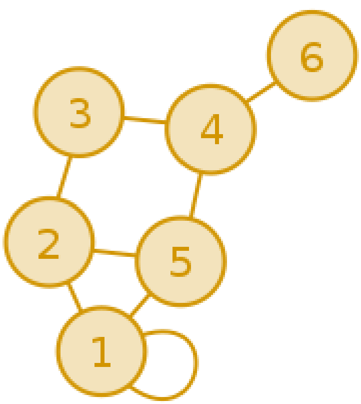
Implementation

Implementations

- Adjacency matrix
- Adjacency list
- Incidence matrix

Adjacency matrix

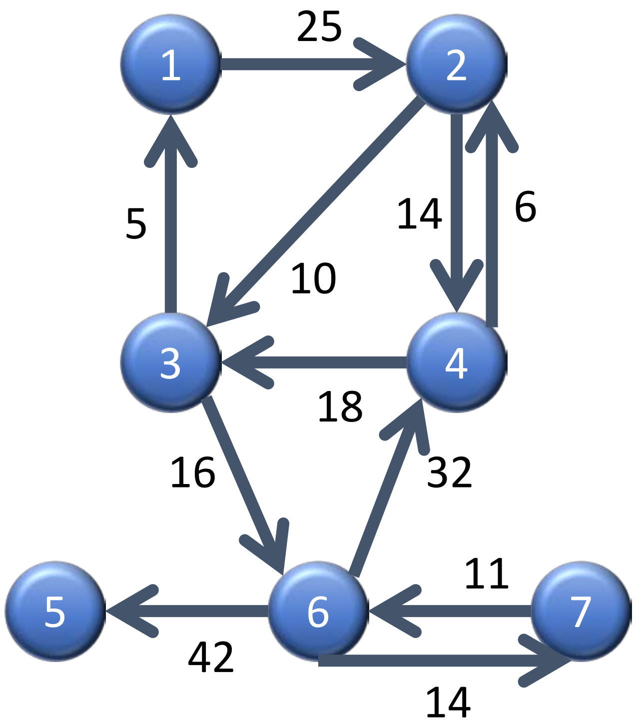
- Let $G = (V, E)$ be a graph with $|V| = n$. Suppose that the vertices of G are listed in arbitrary order as $v_1, v_2, \ldots, v_n$. The **adjacency matrix** A (or A_G) of G, with respect to this listing of the vertices, is the $n \times n$ zero-one matrix with 1 as its $(i, j)^{th}$ entry when v_i and v_j are adjacent, and 0 otherwise. In other words, for an adjacency matrix $A = [a_{ij}]$,
- $a_{ij} = 1$ if $\{v_i, v_j\}$ is an edge of G,
- $a_{ij} = 0$ otherwise.



$$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

https://graphonline.ru/en/create_graph_by_matrix

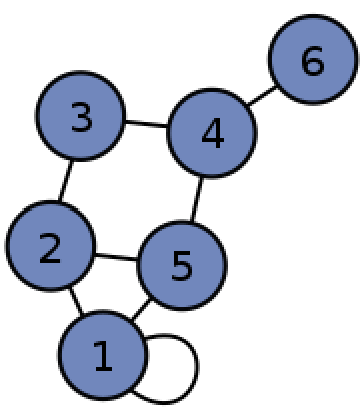
Adjacent matrix for a weighted digraph

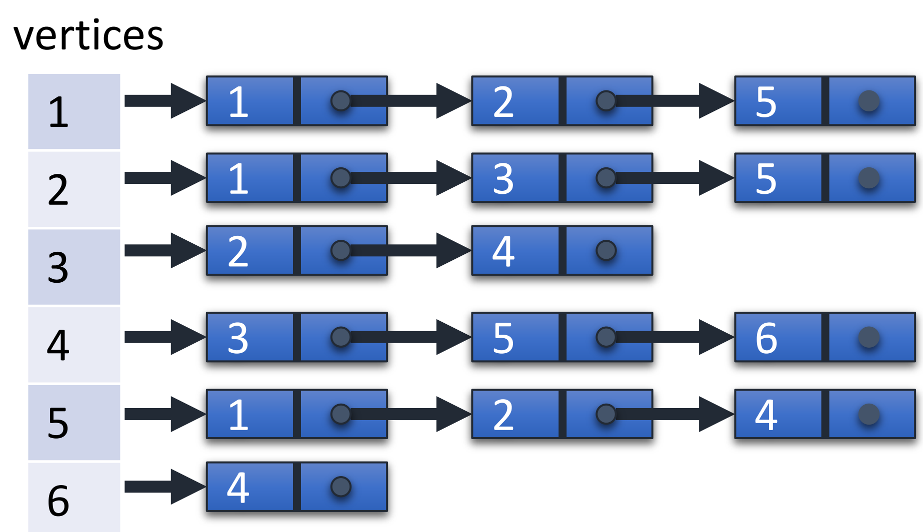


$$
\begin{pmatrix}
0 & 25 & \infty & \infty & \infty & \infty & \infty \\
\infty & 0 & 10 & 14 & \infty & \infty & \infty \\
5 & \infty & 0 & \infty & \infty & 16 & \infty \\
\infty & 6 & 18 & 0 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty \\
\infty & \infty & \infty & 32 & 42 & 0 & 14 \\
\infty & \infty & \infty & \infty & \infty & 11 & 0
\end{pmatrix}
$$

Adjacent list

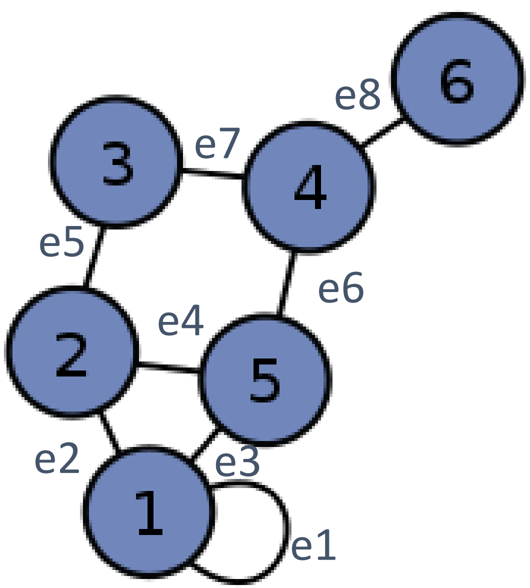
- A collection of unordered lists used to represent a finite graph.
- Each list describes the set of neighbors of a vertex in the graph





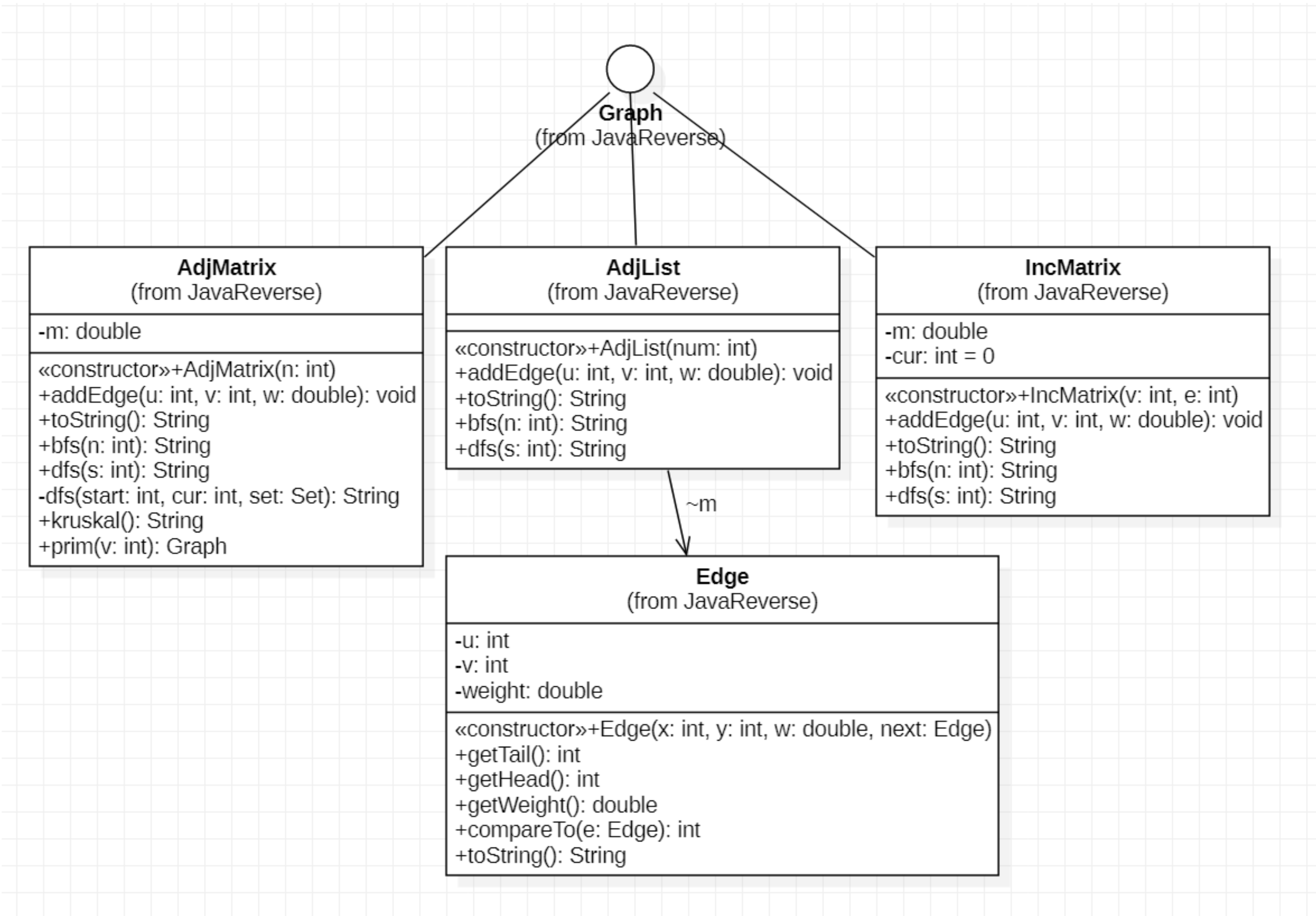
Incidence matrix

- listing of the vertices and edges is the $n \times m$ zero-one matrix with 1 as its $(i, j)^{th}$ entry when edge is incident with v_i, and 0 otherwise. In other words, for an incidence matrix $M = [m_{ij}]$,
  - $m_{ij} = 1$ if edge e_j is incident with v_i
  - $m_{ij} = 0$ otherwise.



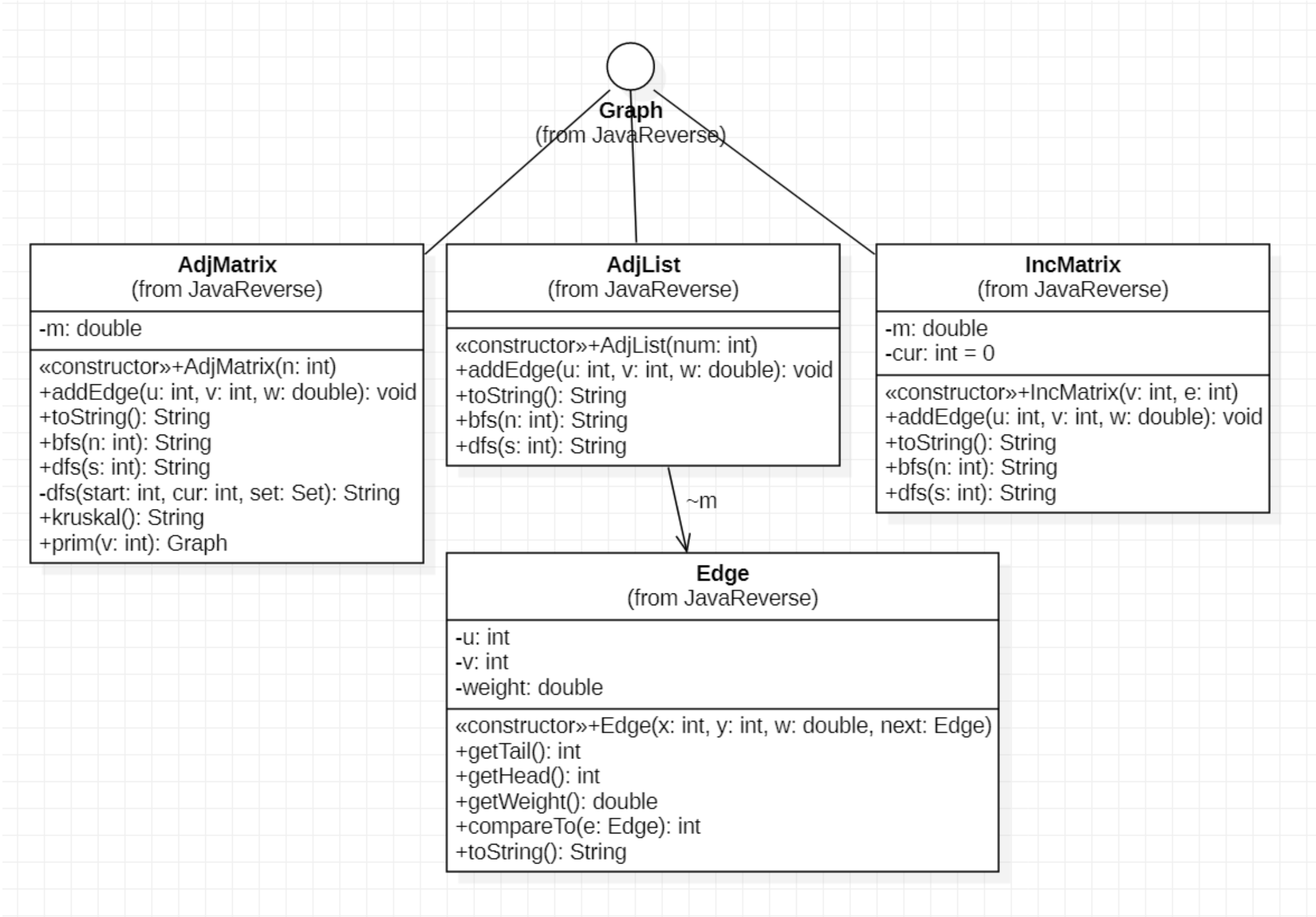
$$\begin{array}{cc} \text{vertex} & \begin{array}{cccccccc} e1 & e2 & e3 & e4 & e5 & e6 & e7 & e8\text{: edge} \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Class diagram



Graph.java

```java
public interface Graph {
    public void addEdge(int u, int v, double w);
}
```



AdjMatrix

```
public class AdjMatrix implements Graph {
     private double m[][];
     public AdjMatrix(int n){
          m = new double[n][n];
          for(int i = 0; i < n ; i++)
               for(int j = 0 ; j < n ; j++)
                    if(i==j) m[i][j] = 0;
                    else m[i][j] = Double.POSITIVE_INFINITY;
     }
     public void addEdge(int u, int v, double w) {
          if( w <= 0 ) return;
          if(u<m.length && v < m.length) {
                    m[u][v] = w;
          }
     }
     public String toString() {
          String s = "";
          for(int i = 0; i < m.length ; i++){
               for(int j = 0 ; j < m.length ; j++) s += m[i][j] + " ";
          s+="\n";
          }
          return s;
     }
}
```

AdjList

```
public class AdjList implements Graph{
    Edge[] m;

    public AdjList(int num ) {
        m=new Edge[num];
    }
    public void addEdge(int u, int v, double w) {
        m[u] = new Edge(u, v, w, m[u]);
    }

    public String toString() {
        String result = "";
        for(int i = 0 ; i < m.length ; i++) {
            result += "\n[" + i + "]; ";
            Edge p = m[i];
            while(p != null) {
                result += p.getHead() + "(" + p.getWeight() +") ";
                p = p.n;
            }
        }
        return result;
    }
}
```

Edge for AdjList

```
public class Edge implements Comparable<Edge>{
    private int u, v;
    private double weight;
    Edge n;
    public Edge(int x, int y, double w, Edge next) {
        u=x;v=y;weight =w; n=next;
    }
    public int getTail() {
        return u;
    }
    public int getHead() {
        return v;
    }
    public double getWeight() {
        return weight;
    }
    public int compareTo(Edge e) {
        if(weight < e.weight)   return -1;
        else if(weight > e.weight) return 1;
        else return 0;
    }
    public String toString() {
        return "("+u+","+v+"):"+weight;
    }
}
```

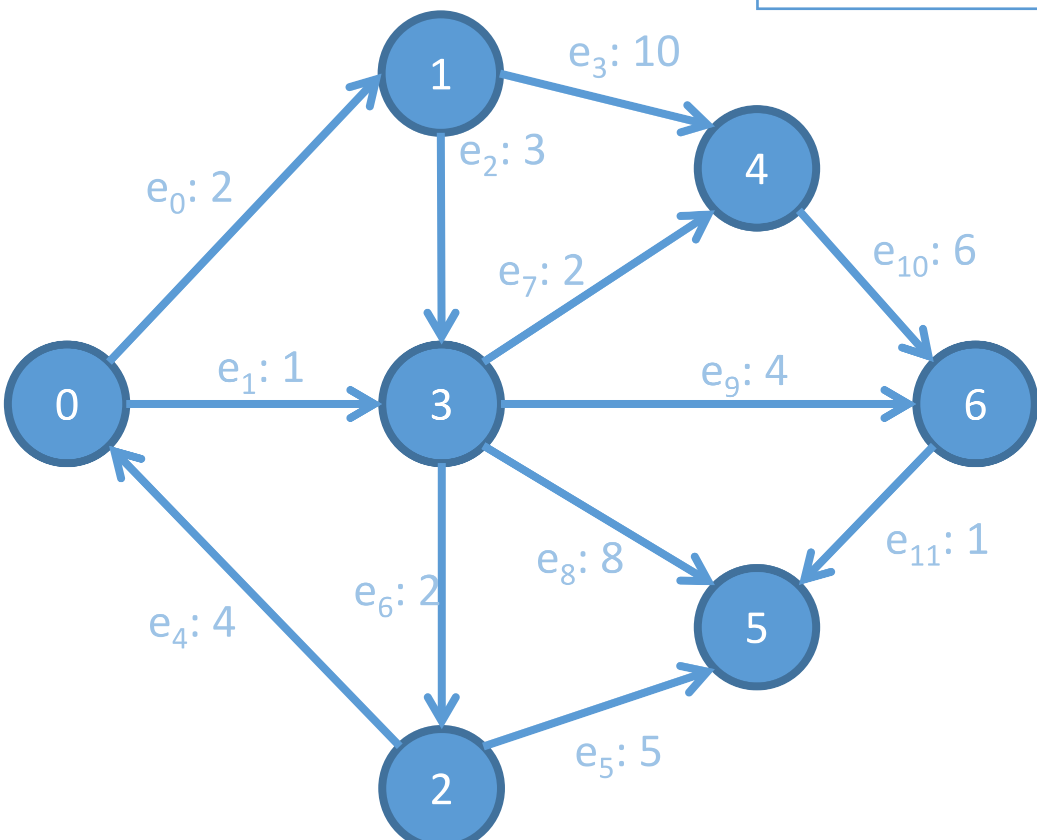
IncMatrix

```java
public class IncMatrix implements Graph {
    private double m[][];
    private int cur=0;
    public IncMatrix(int v, int e){
        m = new double[v][e];
    }
    //vertex u, v, weight w
    public void addEdge(int u, int v, double w){
        m[u][cur] = -w;
        m[v][cur] = w;
        cur++;
    }
    public String toString() {
        String s="";
        for(int i = 0 ; i < m.length ; i++) {
            for(int j = 0 ; j < cur ; j++)
                s+=m[i][j]+"\t";
            s+="\n";
        }
        return s;
    }
}
```

```java
Graph g = new IncMatrix(7, 12);
g.addEdge(0, 1, 2);g.addEdge(0, 3, 1);
g.addEdge(1, 3, 3);g.addEdge(1, 4, 10);
g.addEdge(2, 0, 4);g.addEdge(2, 5, 5);
g.addEdge(3, 2, 2);g.addEdge(3, 4, 2);
g.addEdge(3, 5, 8);g.addEdge(3, 6, 4);
g.addEdge(4, 6, 6);g.addEdge(6, 5, 1);

System.out.println(g);
```

- vertex: 7, edge: 12



Traversal

Graph traversal

- A graph traversal begins at a any vertex (origin vertex) and visits only the vertices that it can reach.

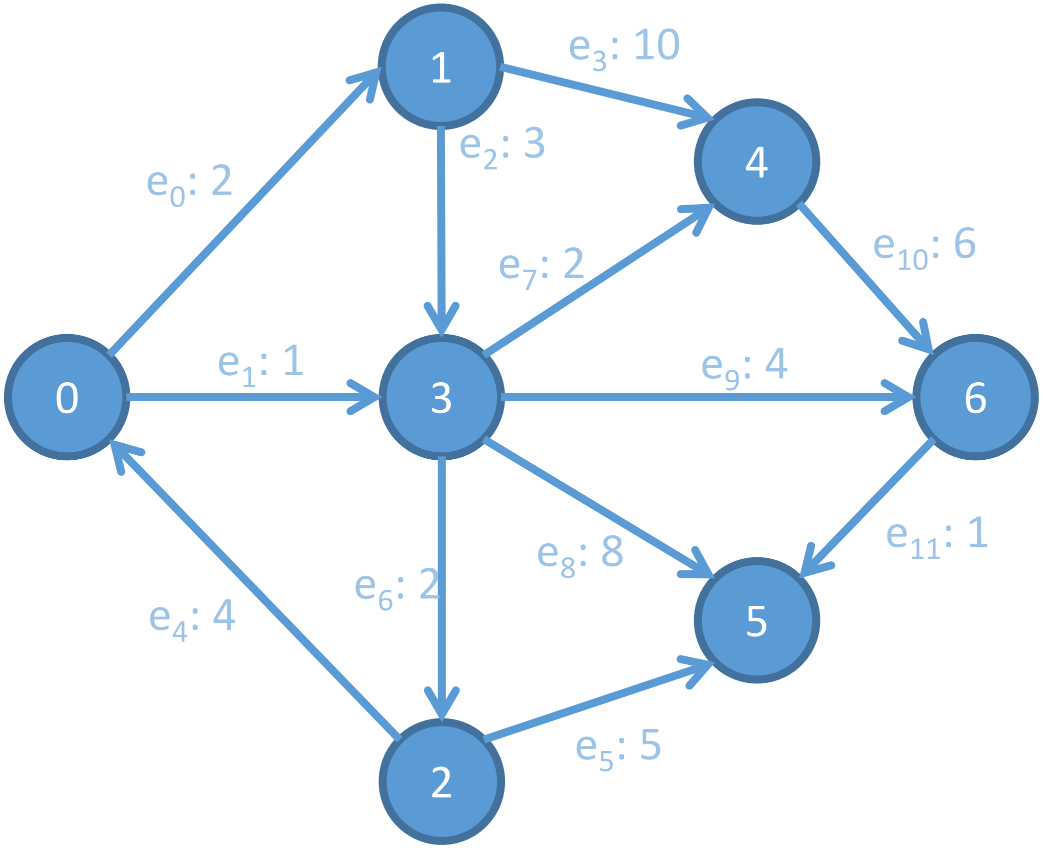
  - Depth First Traversal
  - Breadth First Traversal

Depth First Traversal

- A depth-first traversal visits a vertex, then a neighbor of the vertex, a neighbor of the neighbor, and so on, advancing as far as possible from the original vertex.
- Algorithm for given origin
  1. Define traversalOrder
  2. Put origin to traversalOrder and marked as visited
  3. Check all the neighbors of origin

     Dfs(the neighbor)

Example of DFT



1. Define traversalOrder
2. Put origin to traversalOrder and marked as visited
3. Check all the neighbors of origin

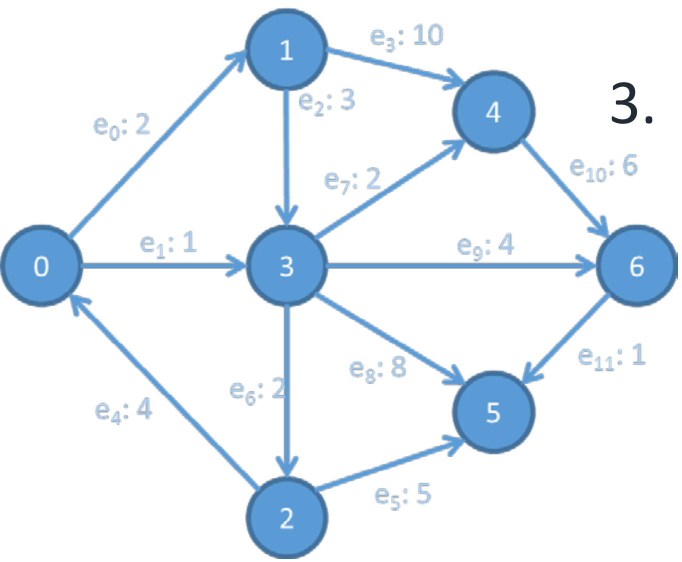
   Dfs(the neighbor)

TraversalOrder:

Breadth First Traversal

- A breadth first traversal visits a vertex and then each of the vertex's neighbors before advancing.
- Algorithm
    1. Make queues for traversalOrder, vertexQ
    2. Enqueue the origin vertex to traversalOrder and vertexQ
    3. While (!vertexQ.isEmpty)
        1. frontVertex = vertexQ.dequeue
        2. While(frontVertex has a neighbor)
            1. nextNeighbor = nextNeighbor of frontVertex
            2. Mark nextNeighbor as visited
            3. If(nextNeighbor is not visited)
                1. Mark nextNeighbor as visited
                2. TraversalOrder.enqueue(nextVertex)
                3. vertexQ.enqueue(nextVertex)

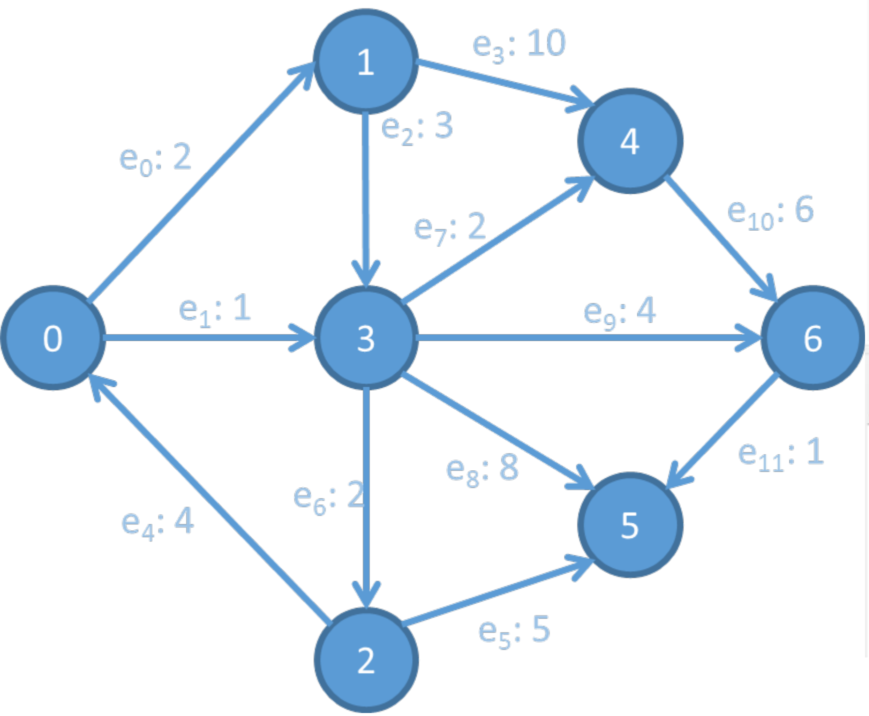
Example of BFT



1. Make queues for traversalOrder, vertexQ
2. Enqueue the origin vertex to traversalOrder and vertexQ
3. While (!vertexQ.isEmpty)
    1. frontVertex = vertexQ.dequeue
    2. While(frontVertex has a neighbor)
        1. nextNeighbor = nextNeighbor of frontVertex
        2. Mark nextNeighbor as visited
        3. If(nextNeighbor is not visited)
            1. Mark nextNeighbor as visited
            2. TraversalOrder.enqueue(nextVertex)

traversalOrder	
vertexQ	

Implementation of traversal




```java
public class GraphTest {
    public static void main(String[] args) {
        Graph g = new AdjMatrix(7);
        g.addEdge(0, 3, 1);     g.addEdge(0, 1, 2);
        g.addEdge(1, 4, 10);    g.addEdge(1, 3, 3);
        g.addEdge(2, 5, 5);     g.addEdge(2, 0, 4);
        g.addEdge(3, 6, 4);     g.addEdge(3, 5, 8);
        g.addEdge(3, 4, 2);     g.addEdge(3, 2, 2);
        g.addEdge(4, 6, 6);     g.addEdge(6, 5, 1);

        System.out.println(g);

        System.out.println("BFT: " + g.bft(0));
        System.out.println("DFT: " + g.dft(0));
    }
}
```

Console Problems Debug Shell

```
<terminated> GraphTest (1) [Java Application] C:₩Program Files₩Java₩jdk-15.0.1₩bin₩javaw.exe (2021. 3. 22. 오후 11:04:31 – 오후 1
0.0	2.0	Infinity	1.0	Infinity	Infinity	Infinity
Infinity	0.0	Infinity	3.0	10.0	Infinity	Infinity
4.0	Infinity	0.0	Infinity	Infinity	5.0	Infinity
Infinity	Infinity	2.0	0.0	2.0	8.0	4.0
Infinity	Infinity	Infinity	Infinity	0.0	Infinity	6.0
Infinity	Infinity	Infinity	Infinity	Infinity	0.0	Infinity
Infinity	Infinity	Infinity	Infinity	Infinity	1.0	0.0

BFT: 0 1 3 4 2 5 6
DFT: 0 1 3 2 5 4 6
```

AdjMatrix BFT

```
public String bft(int n) {
    String s = "";
    boolean[] visit = new boolean[m.length];
    for(int i = 0; i< m.length ; i++)
        visit[i] = false;
    Queue<Integer> q = new ArrayDeque<Integer>();
    q.add(n);
    visit[n] = true;
    Integer temp;
    while(q.size()>0){
        temp = q.remove();
        s+= temp + " ";
        for(int i = 0 ; i < m.length ; i++) {
            if(m[temp][i] < Double.POSITIVE_INFINITY)
                if(visit[i]==false) {
                    q.add(i);
                    visit[i] = true;
                }
        }
    }
    return s;
}
```

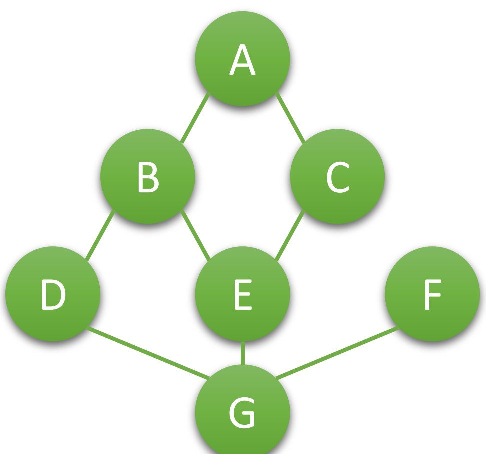
AdjMatrix DFT

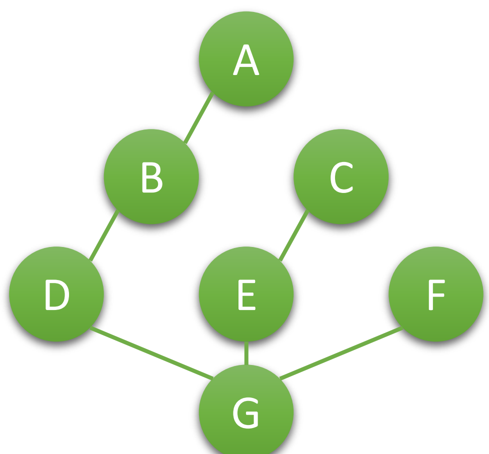
```
    public String dft(int s) {
        Set set = new Set(m.length);
        return dft(s, s, set);
    }
    private String dft(int start, int cur, Set set) {
        String result = cur +" ";
        set.union(start, cur);
        for(int i = 0 ; i < m.length ; i++) {
            if(m[cur][i] < Double.POSITIVE_INFINITY && !set.isSameSet(start, i))
                result += dft(start, i, set);
        }
        return result;
    }
}
```

Spanning Tree

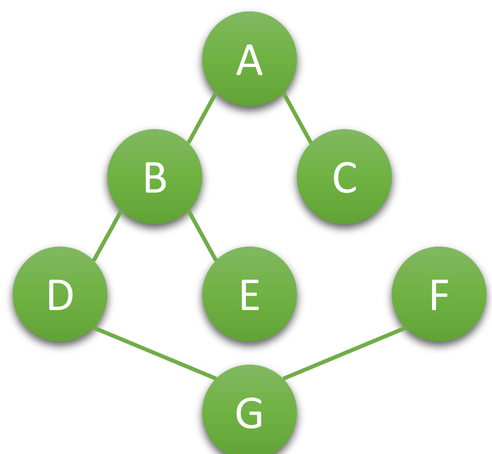
Spanning Tree

- Spanning tree T of an undirected graph G
  - a **subgraph** which includes all of the vertices of G, with **minimum possible number of edges**.
  - In general, a graph may have **several spanning trees**, but a graph that is not connected will not contain a spanning tree (but Spanning forests).





DF Spanning Tree



BF Spanning Tree

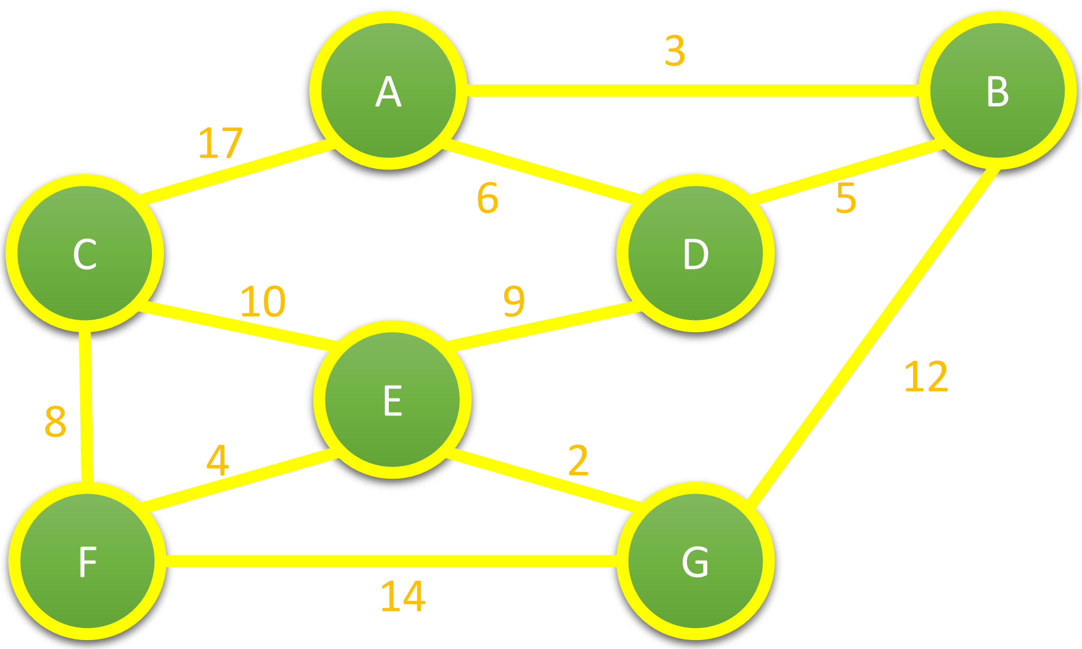
Minimum Spanning Tree

- aka
  - **MST**
  - **minimum weight spanning tree**
  - Minimum cost spanning tree
- A spanning tree with the minimum possible total edge weight.

- Algorithm
  - Kruskal’s algorithm
  - Prim’s algorithm

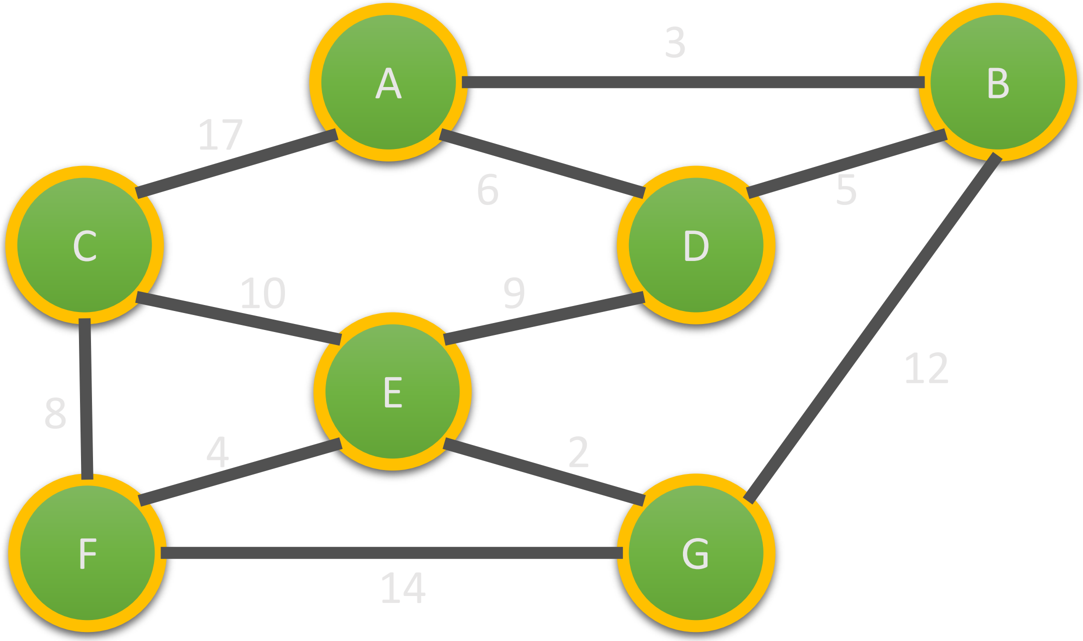
Kruskal's algorithm

```
KRUSKAL(G):
A = { }
Sort edges (u, v)ordered by weight(u, v), increasing:
foreach (u, v)
      if A = A ∪ {(u, v)} does not make a cycle
            UNION(u, v)
      if |A| = n-1
            break;
return A
```



Kruskal's algorithm

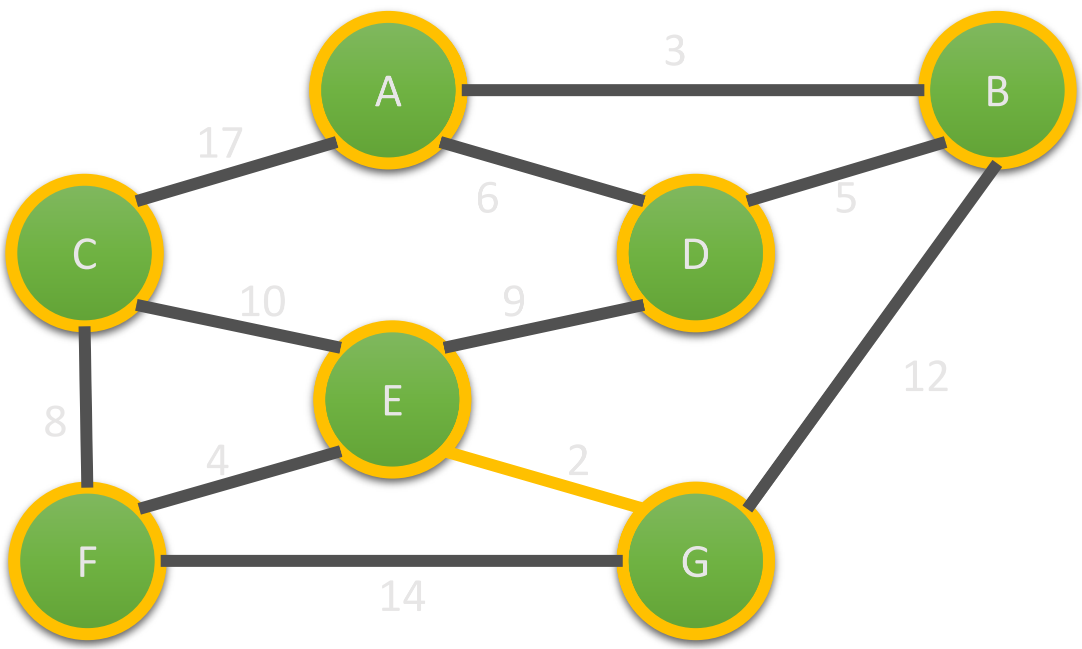
```
A = { }
Sort edges (u, v)ordered by weight(u, v), increasing:
```



Weight	Edge
2	{E,G}
3	{A,B}
4	{E,F}
5	{B,D}
6	{A,D}
8	{C,F}
9	{D,E}
10	{C,E}
12	{B,G}
14	{F,G}
17	{A,C}

Kruskal's algorithm

foreach (u, v)
    if A = A ∪ {(u, v)} does not make a cycle
        UNION(u, v)
    if |A| = n-1
        break;



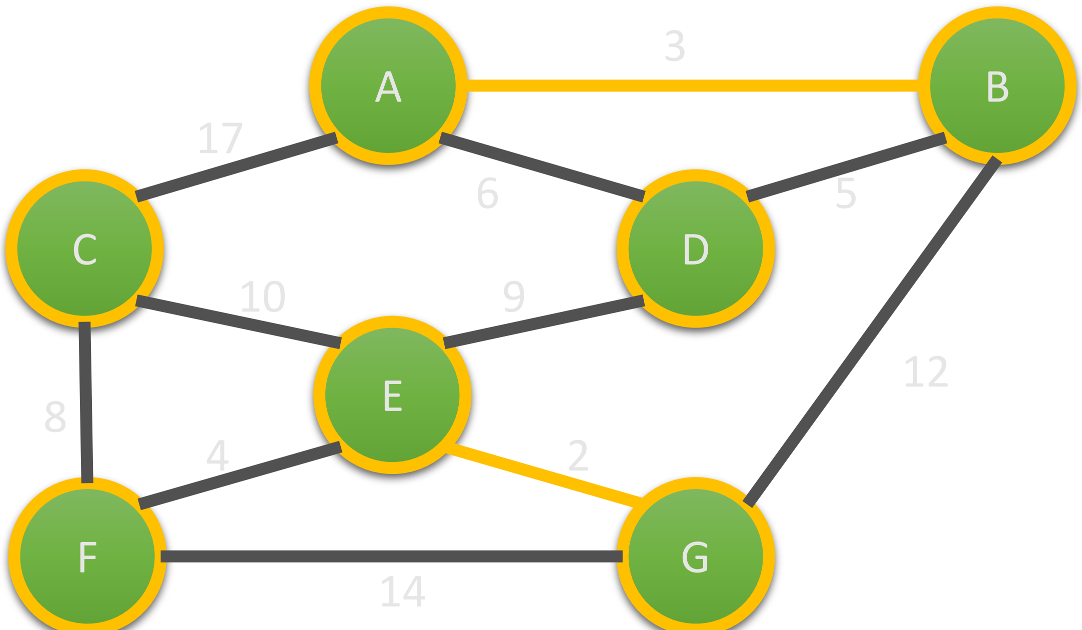
A = {(E,G)}

Weight	Edge
2	{E,G}
3	{A,B}
4	{E,F}
5	{B,D}
6	{A,D}
8	{C,F}
9	{D,E}
10	{C,E}
12	{B,G}
14	{F,G}
17	{A,C}

2

Kruskal's algorithm

foreach (u, v)
    if A = A ∪ {(u, v)} does not make a cycle
        UNION(u, v)
    if |A| = n-1
        break;

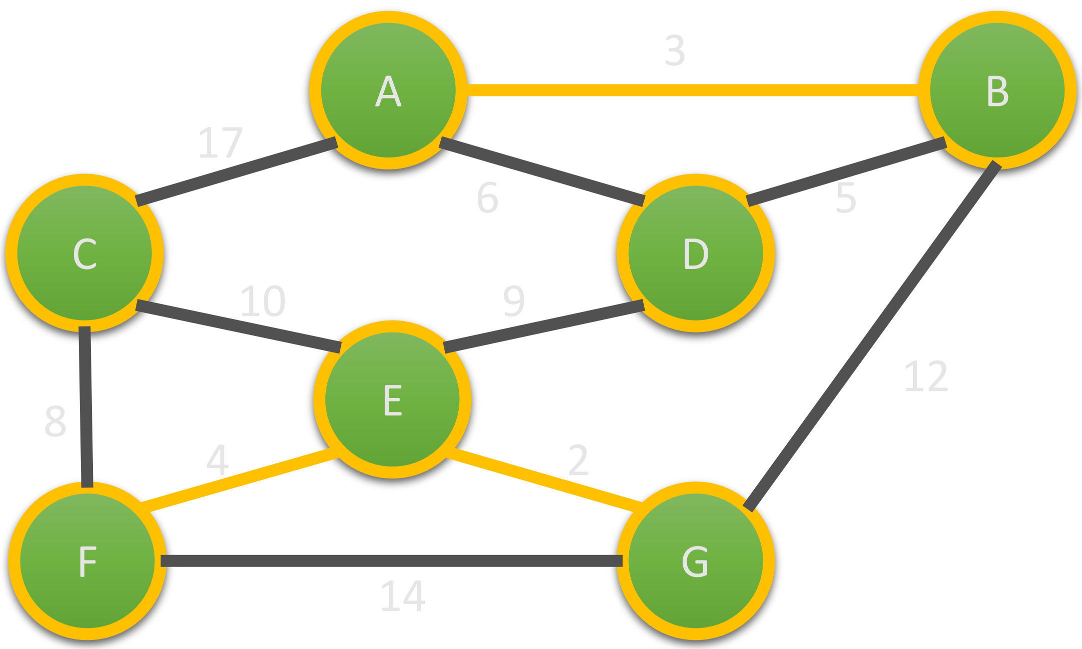


A = {(E,G), {A,B}}

Weight	Edge
2	{E,G}
3	{A,B}
4	{E,F}
5	{B,D}
6	{A,D}
8	{C,F}
9	{D,E}
10	{C,E}
12	{B,G}
14	{F,G}
17	{A,C}

5

Kruskal's algorithm

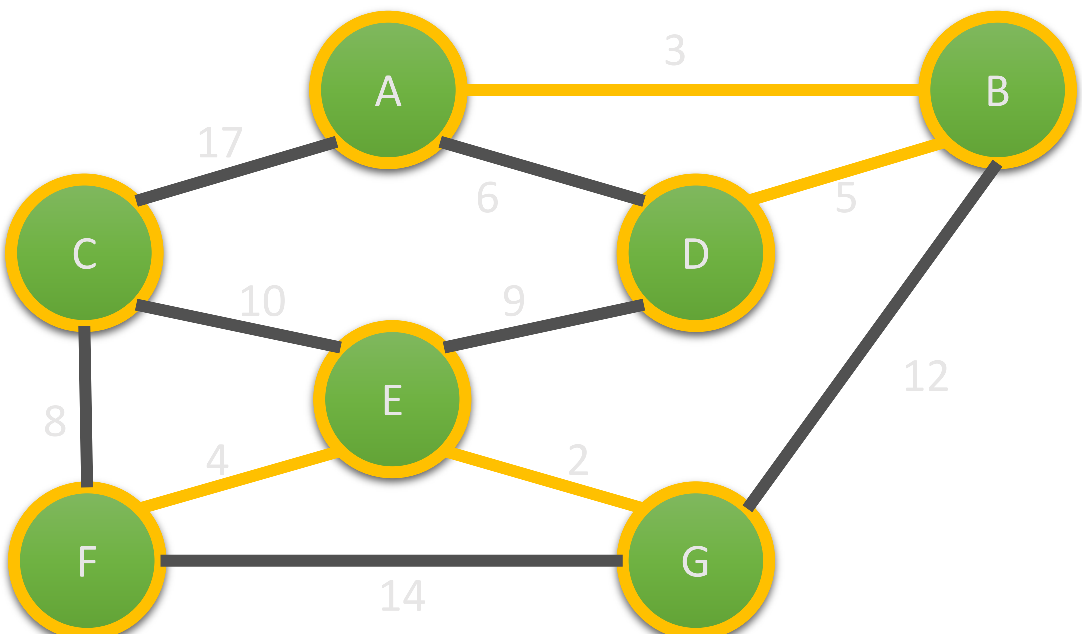


A = {(E,G), {A,B}, {E,F}}

Weight	Edge
2	{E,G}
3	{A,B}
4	{E,F}
5	{B,D}
6	{A,D}
8	{C,F}
9	{D,E}
10	{C,E}
12	{B,G}
14	{F,G}
17	{A,C}

9

Kruskal's algorithm

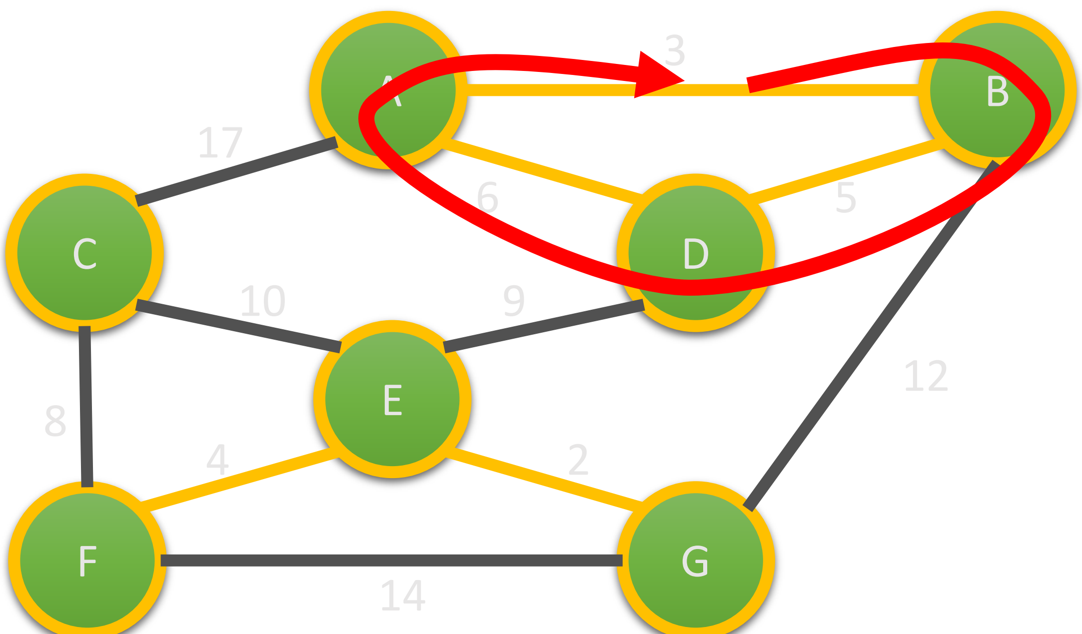


Weight	Edge
2	{E,G}
3	{A,B}
4	{E,F}
5	{B,D}
6	{A,D}
8	{C,F}
9	{D,E}
10	{C,E}
12	{B,G}
14	{F,G}
17	{A,C}

14

A = {(E,G), {A,B}, {E,F}, {B,D}}

Kruskal's algorithm

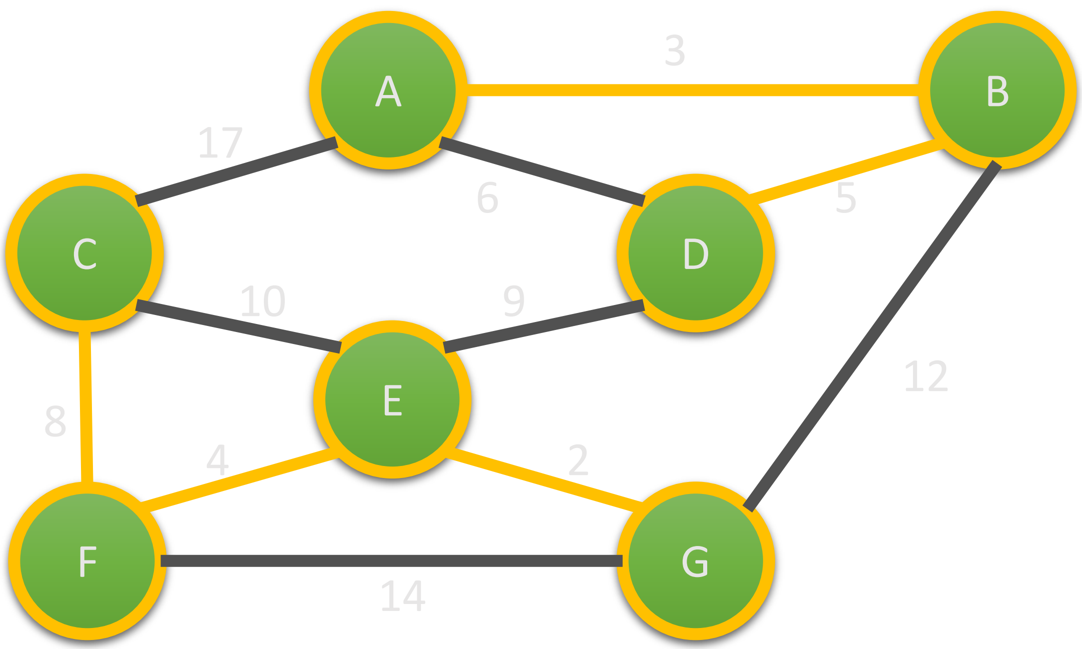


Weight	Edge
2	{E,G}
3	{A,B}
4	{E,F}
5	{B,D}
6	~~{A,D}~~
8	{C,F}
9	{D,E}
10	{C,E}
12	{B,G}
14	{F,G}
17	{A,C}

14

A = {(E,G), {A,B}, {E,F}, {B,D}}

Kruskal's algorithm

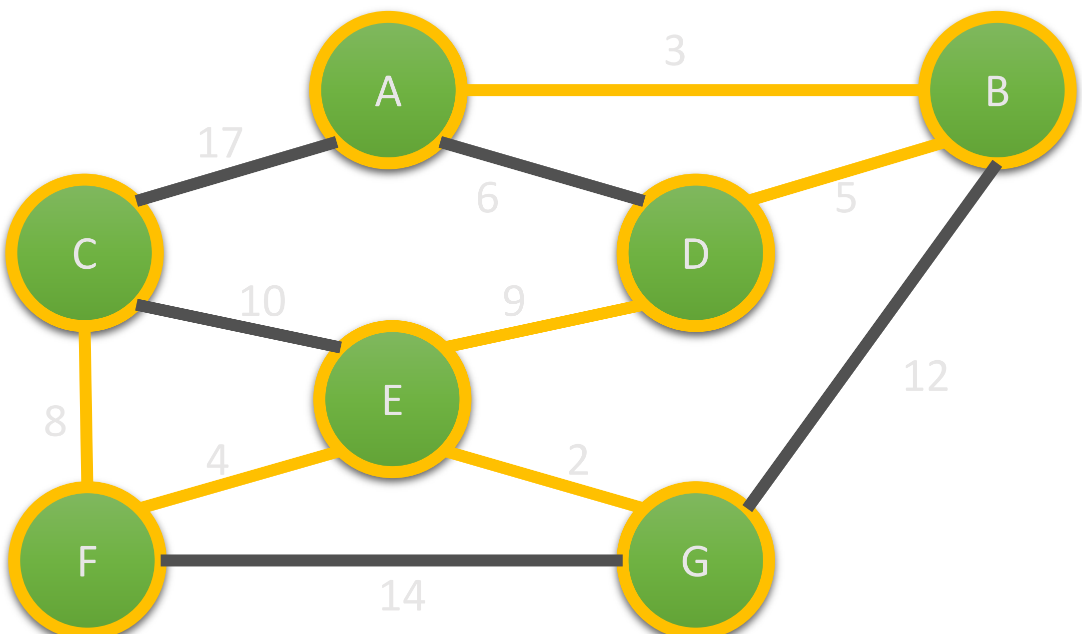


A = {(E,G), {A,B}, {E,F}, {B,D}, {C,F}}

Weight	Edge
2	{E,G}
3	{A,B}
4	{E,F}
5	{B,D}
6	~~{A,D}~~
8	{C,F}
9	{D,E}
10	{C,E}
12	{B,G}
14	{F,G}
17	{A,C}

22

Kruskal's alorithm



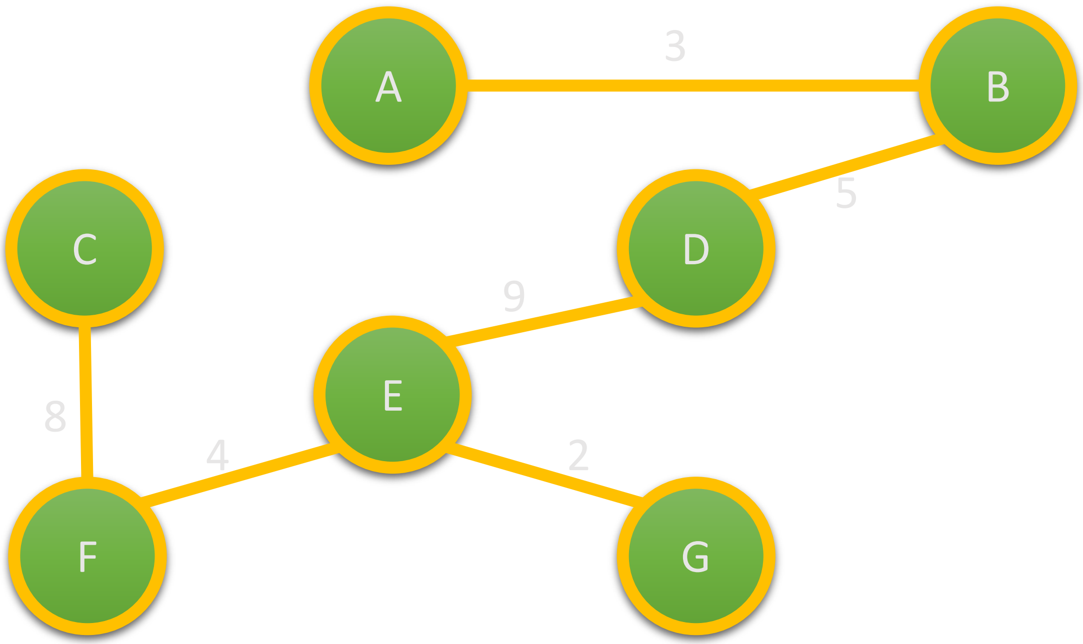
A = {(E,G), {A,B}, {E,F}, {B,D}, {C,F},{D,E}}

Weight	Edge
2	{E,G}
3	{A,B}
4	{E,F}
5	{B,D}
6	~~{A,D}~~
8	{C,F}
9	{D,E}
10	{C,E}
12	{B,G}
14	{F,G}
17	{A,C}

31

Kruskal's alorithm

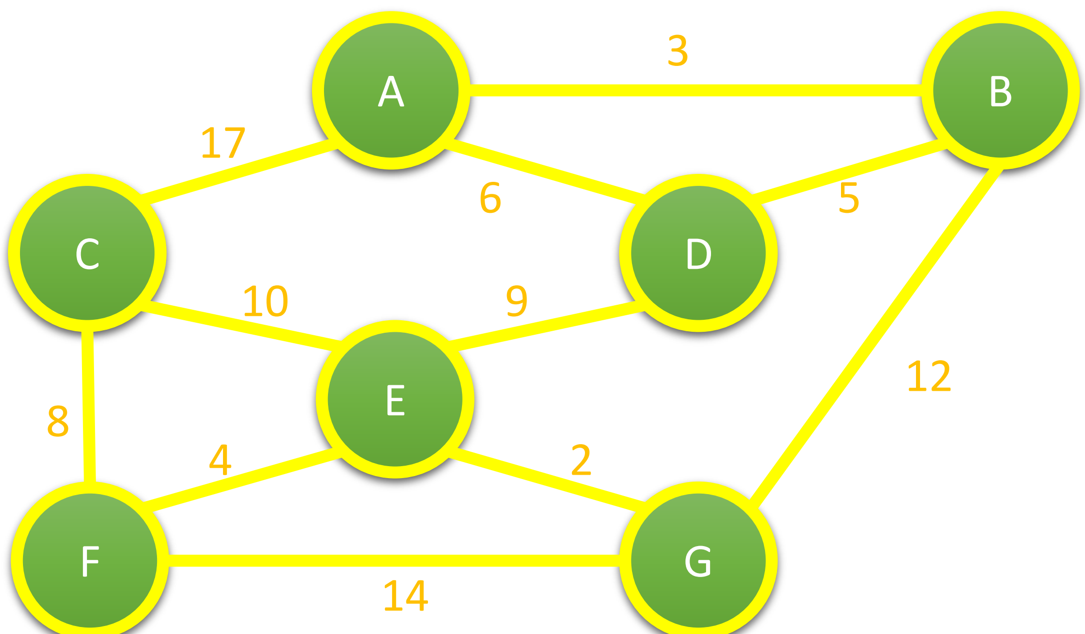
- Total weight: 31
- Return A = {(E,G), {A,B}, {E,F}, {B,D}, {C,F},{D,E}}



Prim's algorithm

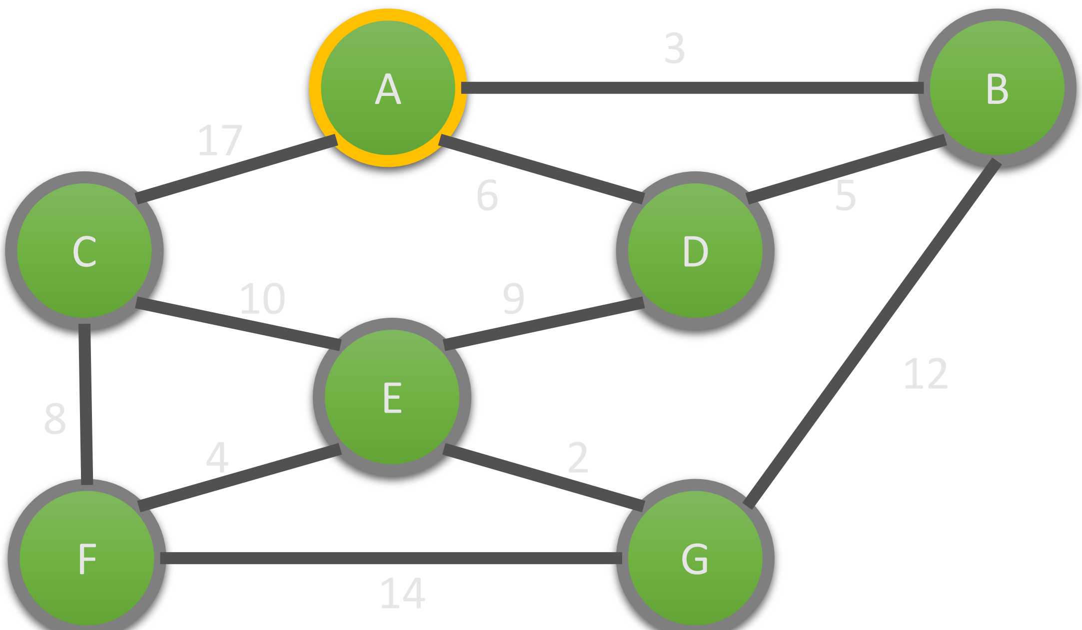
```
Prim(G):
```

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).



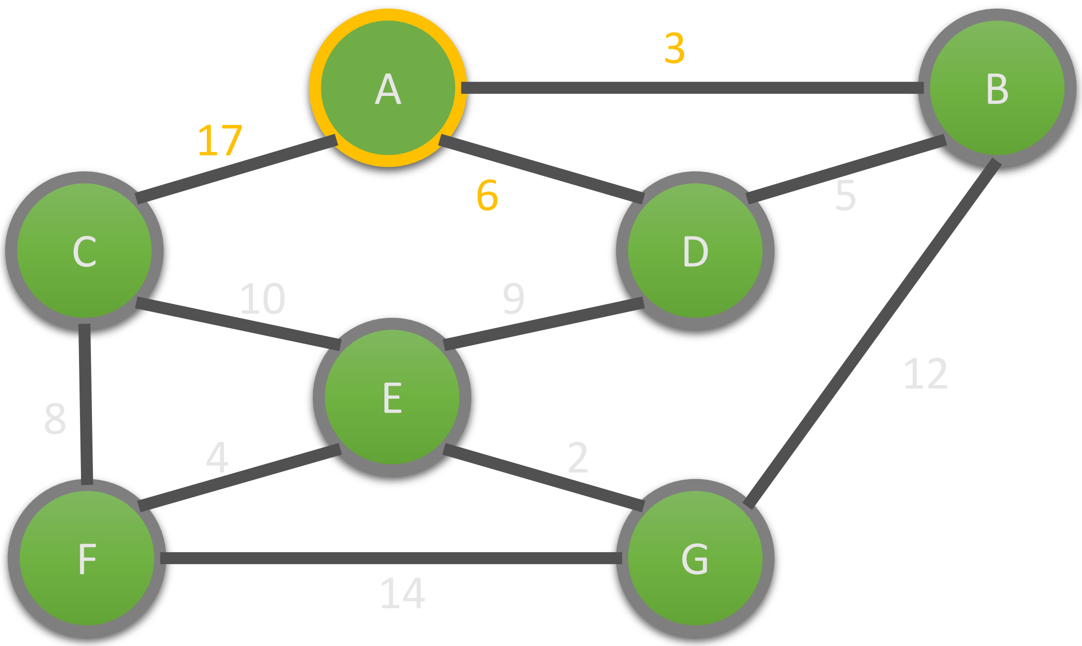
Prim's algorithm

Initialize a tree with a single vertex, chosen arbitrarily from the graph.



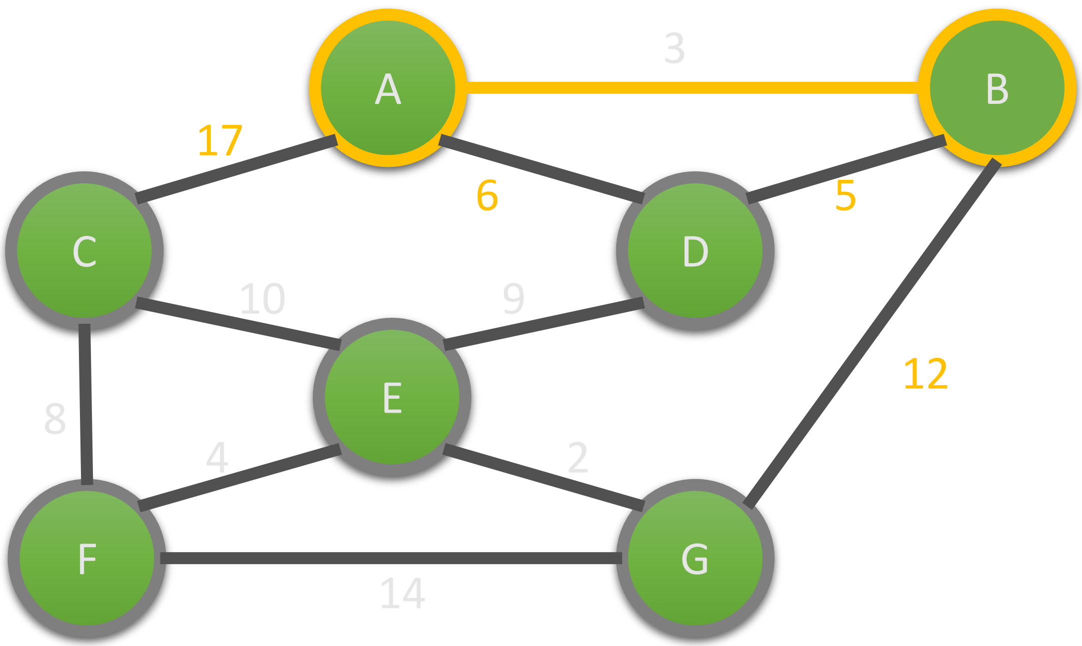
Prim's algorithm

Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.



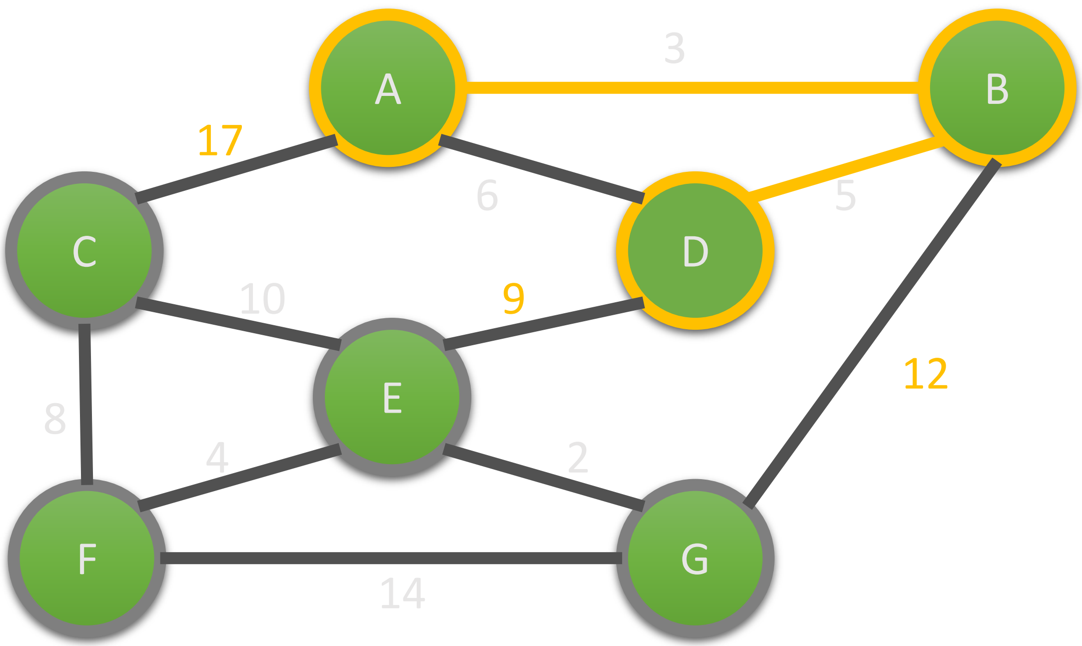
Prim's algorithm

Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.



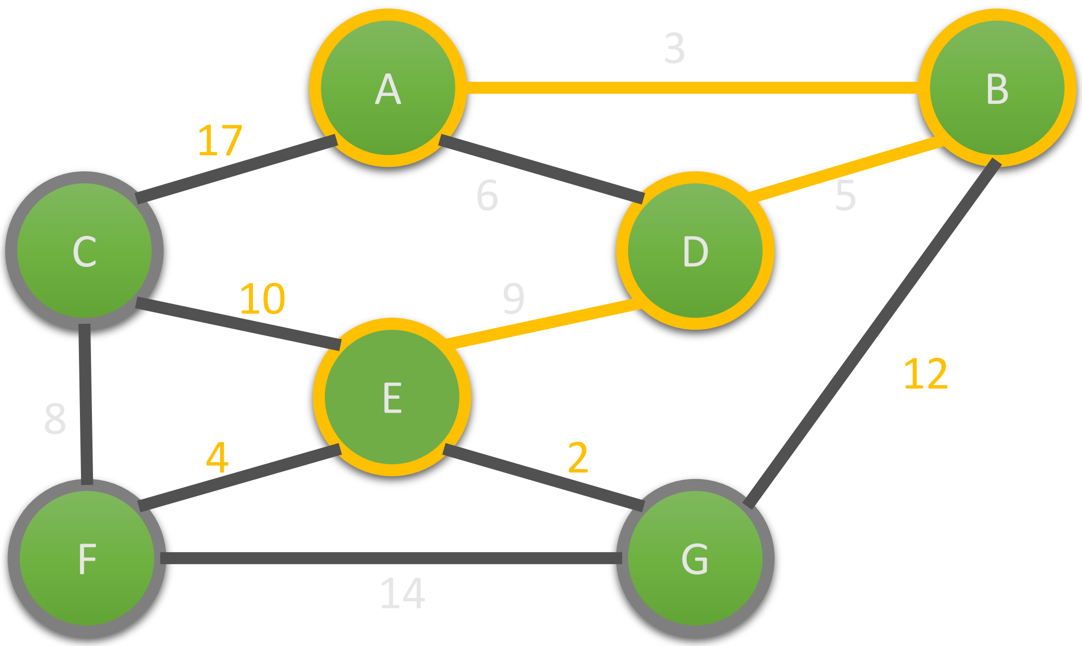
Prim's algorithm

Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.



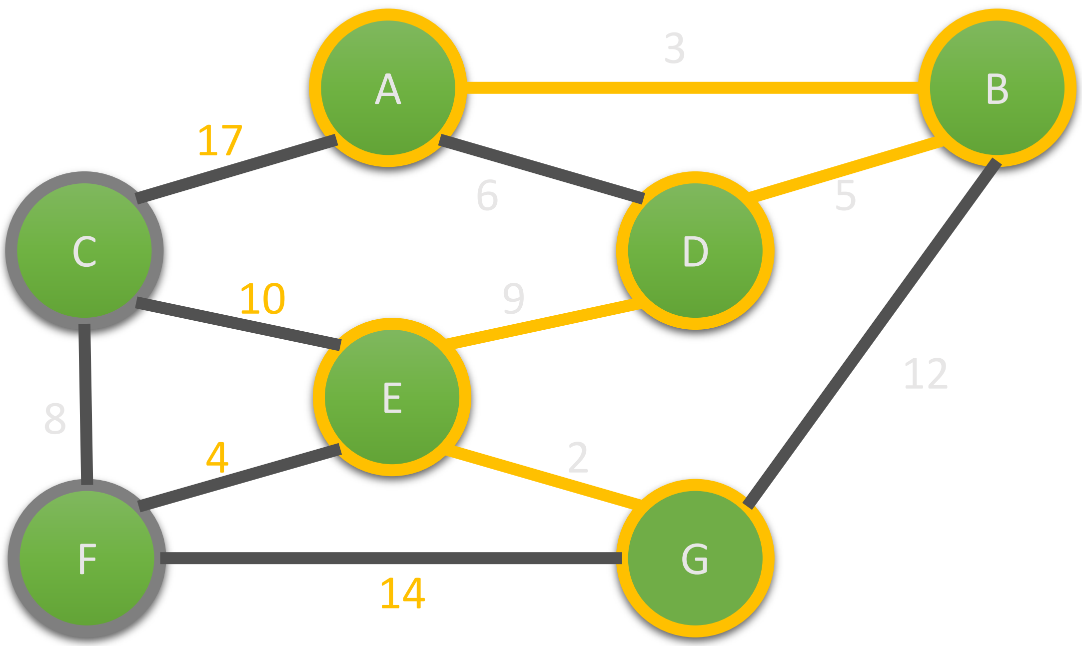
Prim's algorithm

Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.



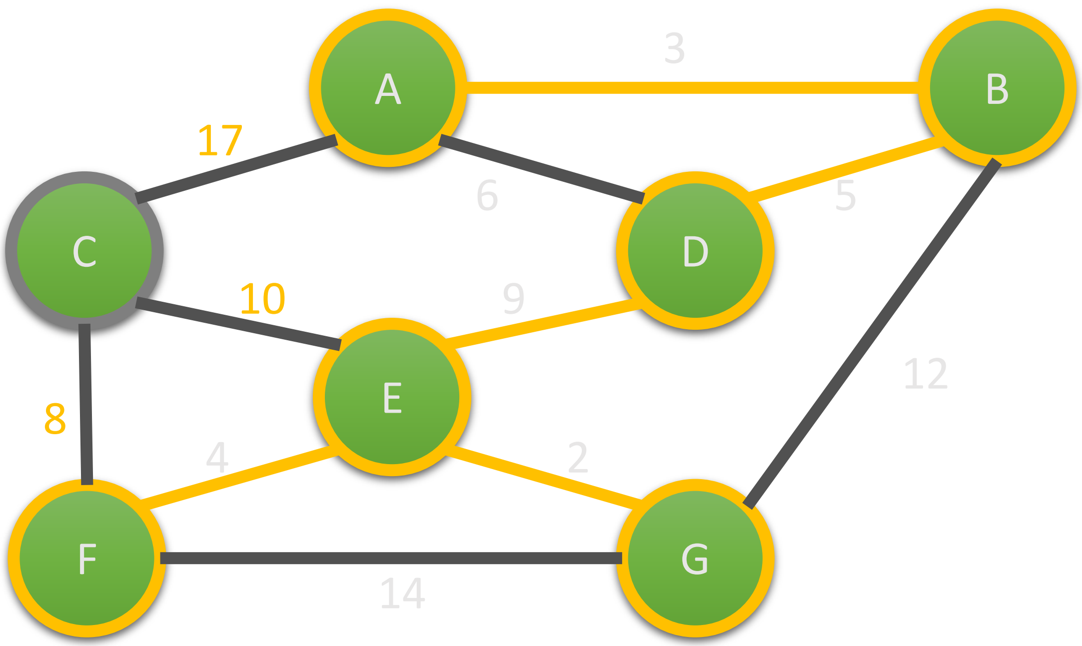
Prim's algorithm

Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.



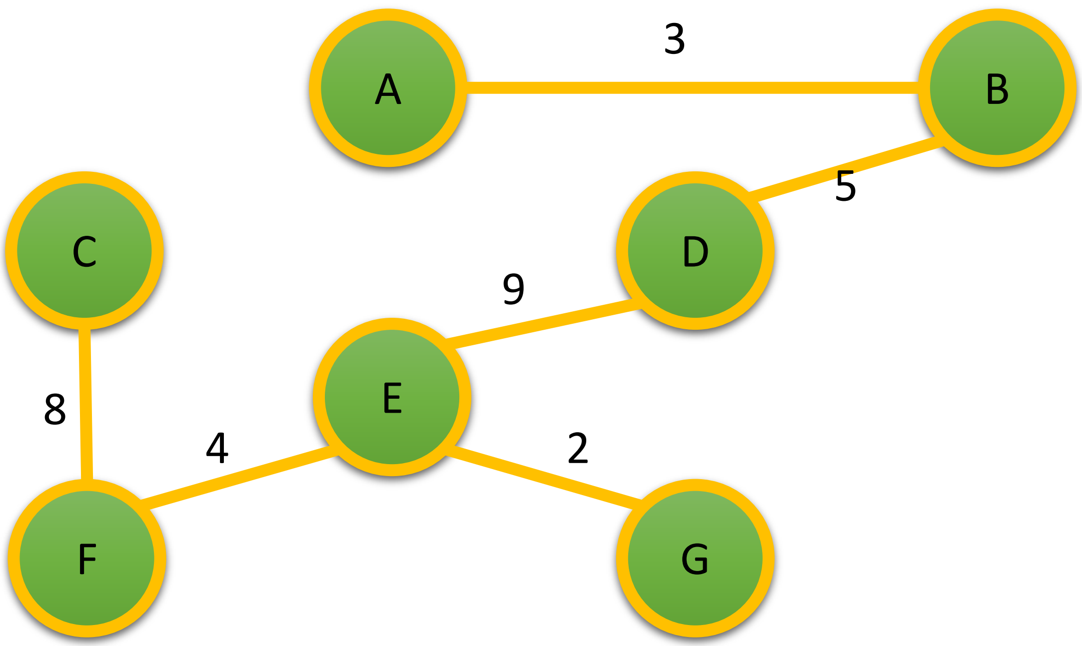
Prim's algorithm

Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.



Prim's algorithm

- Total weight: 31 (3+5+9+2+4+8)



Shortest path

Shortest path problem

- Finding the shortest possible route from Seoul to Pusan.
  - Shortest path

$$\delta(u,v) = \begin{cases} min \sum_{k=1}^{k} w(v_{i-1}, v_i) & if\ there\ is\ a\ path \\ \infty & no\ path \end{cases}$$

  - Vertex: intersection
  - Edge: road segment between intersections
  - Weight: distance

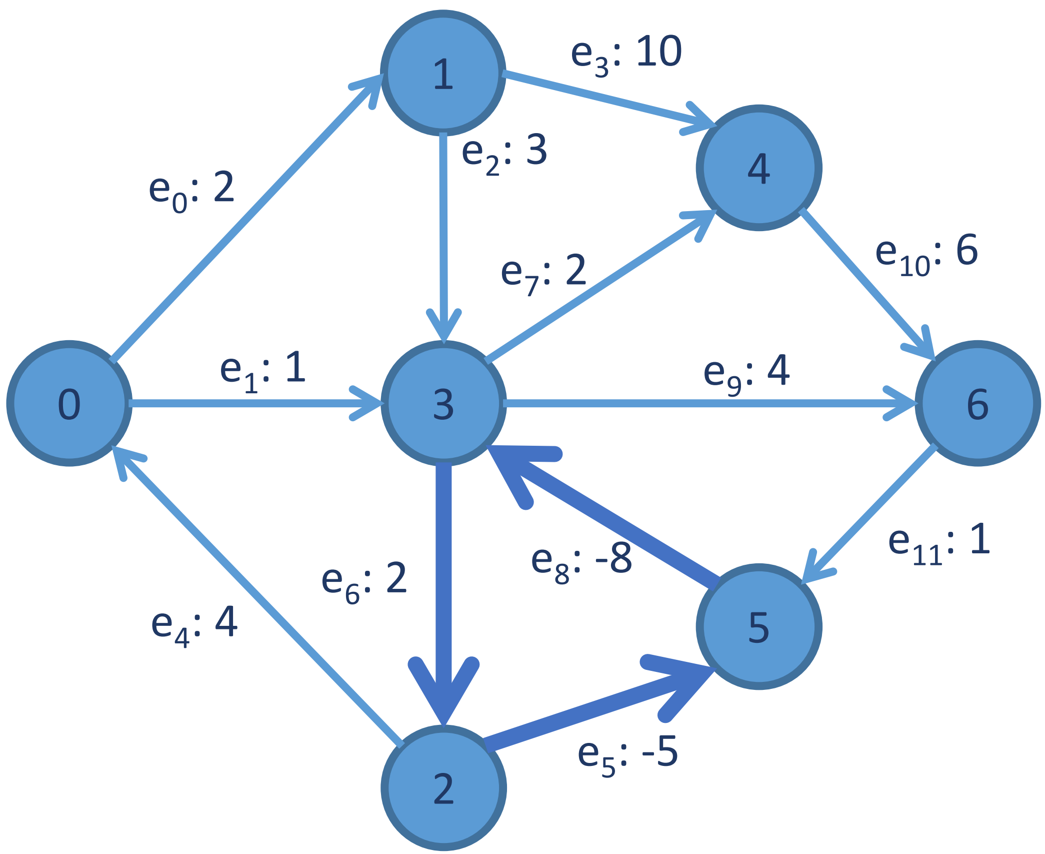
variation

- Single source shortest path
  - No negative weight edges: Dijkstra’s algorithm
  - Negative weight edges: The Bellman-Ford algorithm
  - Directed acyclic graph
- All pairs shortest paths
  - Floyd-Warshall algorithm

- Single destination shortest path algorithm
  - Reverse of single source shortest path
- Single pair shortest path problem
  - A* search algorithm

Negative edge

- What happens if there is a negative cycle?



Dijkstra algorithm

- Condition
  - Dijkstra algorithm works only for **connected graphs**.
  - Dijkstra algorithm works only for those graphs that do **not** contain any **negative weight edge**.
  - Dijkstra algorithm works for directed as well as undirected graphs.

Dijkstra algorithm

- dist[S] ← 0 // The distance to source vertex is set to 0
- Π[S] ← **NIL** // The predecessor of source vertex is set as NIL
- **for** all v ∈ V - {S} // For all other vertices

  do dist[v] ← ∞ // All other distances are set to ∞

  Π[v] ← **NIL** // The predecessor of all other vertices is set as NIL
- S ← ∅ // The set of vertices that have been visited 'S' is initially empty
- Q ← V // The queue 'Q' initially contains all the vertices
- **while** Q ≠ ∅ // While loop executes till the queue is not empty

  do u ← mindistance (Q, dist) // A vertex from Q with the least distance is selected

  S ← S ∪ {u} // Vertex 'u' is added to 'S' list of vertices that have been visited

  for all v ∈ neighbors[u] // For all the neighboring vertices of vertex 'u'

  do if dist[v] > dist[u] + w(u,v) // if any new shortest path is discovered

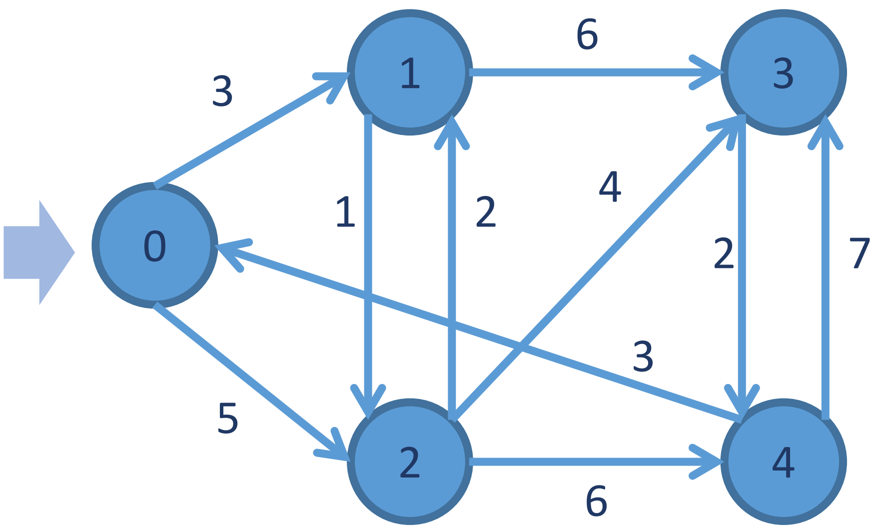
  then dist[v] ← dist[u] + w(u,v) // The new value of the shortest path is selected
- **return** dist

Dijkstra algorithm

- dist[S] ← 0
- Π[S] ← **NIL**
- **for** all v ∈ V - {S} // For all other vertices

  do dist[v] ← ∞

  Π[v] ← **NIL**
- S ← Ø
- Q ← V



Dijkstra algorithm

- **while** Q ≠ Ø

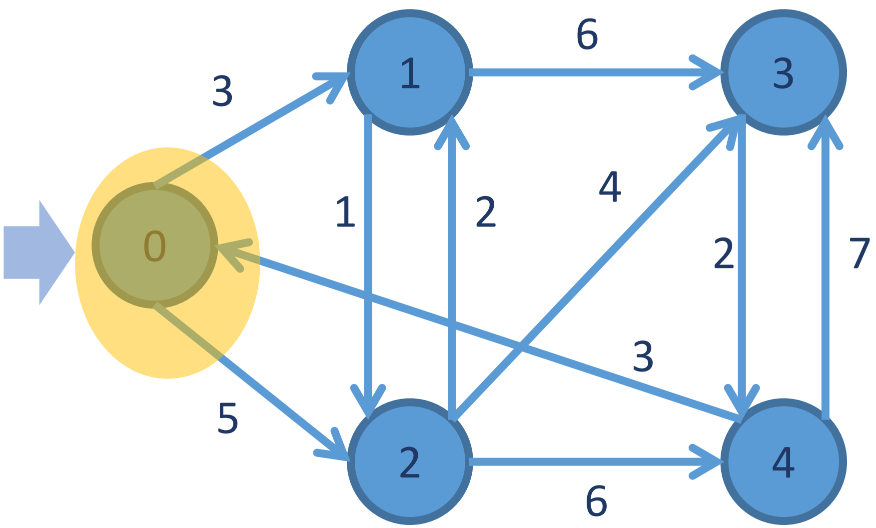
  do u ← minDistance (Q, dist)

  S ← S ∪ {u}

  for all v ∈ neighbors[u]

  do if dist[v] > dist[u] + w(u,v)

  then dist[v] ← dist[u] + w(u,v)



Dijkstra algorithm

- **while** Q ≠ Ø

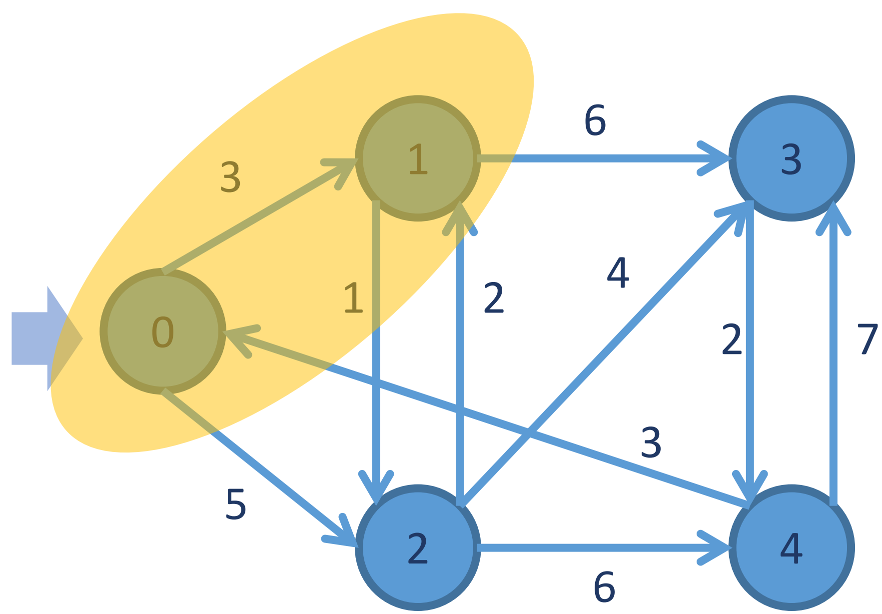
  do u ← minDistance (Q, dist)

  S ← S ∪ {u}

  for all v ∈ neighbors[u]

    do if dist[v] > dist[u] + w(u,v)

      then dist[v] ← dist[u] + w(u,v)



Dijkstra algorithm

- **while** Q ≠ ∅

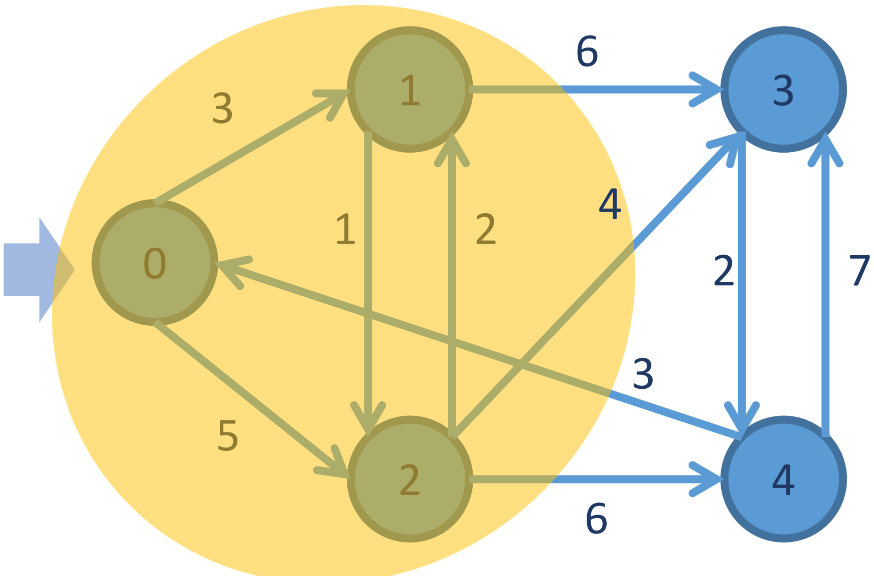
  do u ← minDistance (Q, dist)

  S ← S ∪ {u}

  for all v ∈ neighbors[u]

  do if dist[v] > dist[u] + w(u,v)

  then dist[v] ← dist[u] + w(u,v)



Dijkstra algorithm

- **while** Q ≠ Ø

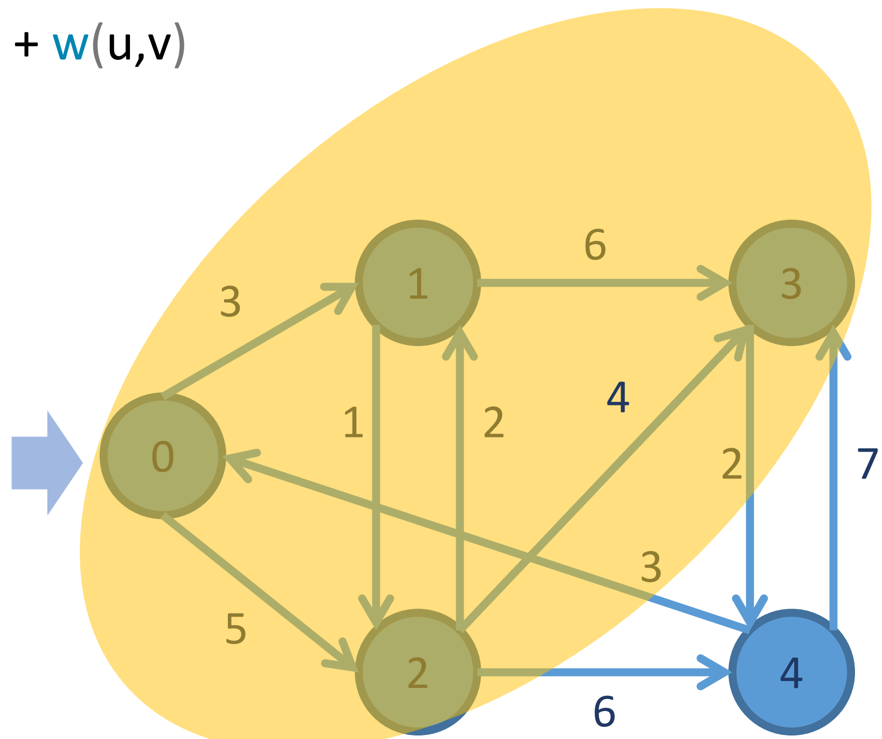
  do u ← minDistance (Q, dist)

  S ← S ∪ {u}

  for all v ∈ neighbors[u]

    do if dist[v] > dist[u] + w(u,v)

      then dist[v] ← dist[u] + w(u,v)



Dijkstra algorithm

- **while** Q ≠ Ø

  do u ← minDistance (Q, dist)

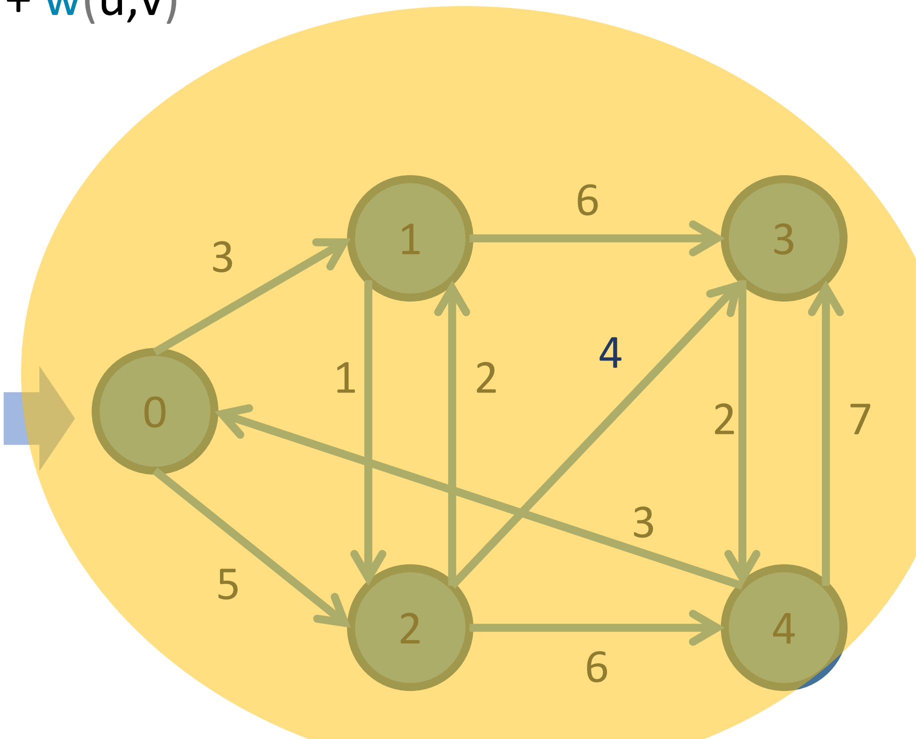
  S ← S ∪ {u}

  for all v ∈ neighbors[u]

  do if dist[v] > dist[u] + w(u,v

  then dist[v] ← dist[u] + w(u,v)

```
Return dist
```

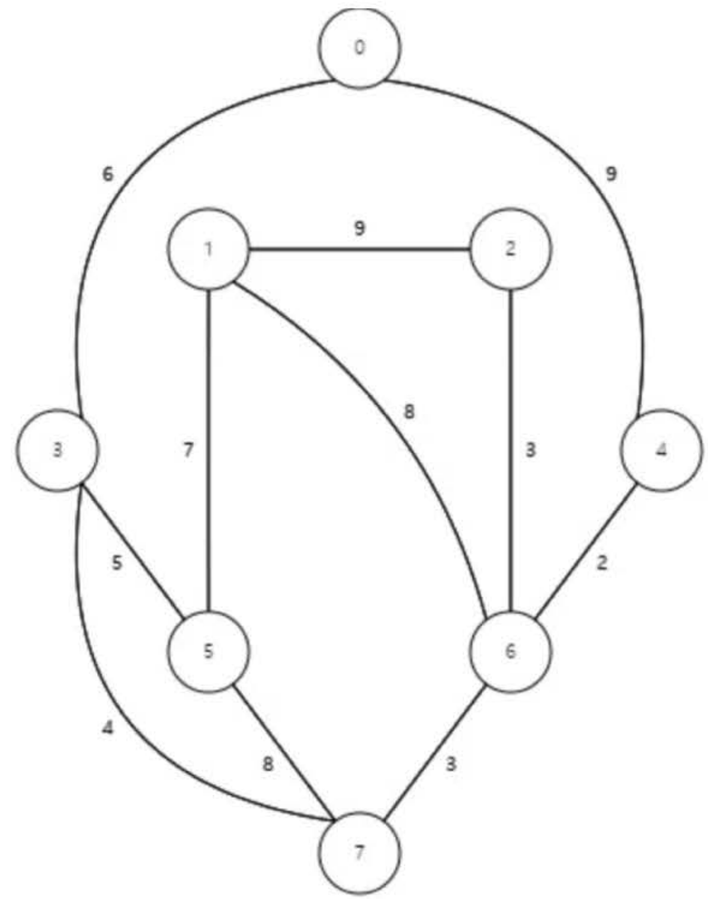


Dijkstra algorithm

- Visualization

  https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

Vertex	Known	Cost	Path
0			
1			
2			
3			
4			
5			
6			
7			



Time complexity of Dijkstra algorithm

- Time taken for selecting i with the smallest dist is O(V).
- For each neighbor of i, time taken for updating dist[j] is O(1) and there will be maximum V neighbors.
- Time taken for each iteration of the loop is O(V) and one vertex is deleted from Q.
- Thus, total time complexity becomes $O(V^2)$.
- With adjacency list representation, all vertices of the graph can be traversed using BFS in O(V+E) time.
- In min heap, operations like extract-min and decrease-key value takes O(logV) time.

O(E+V) x O(logV) ➔ O(ElogV)

It can be reduced to O(E+VlogV) using Fibonacci heap.

Bellman-Ford algorithm

- Condition
  - Edge weights may be negative
  - If there is a negative cycle, no solution exists.

Bellman-Ford algorithm

```
Initialize single source (G, s)
For I = 1 to |G,V|-1
        for each edge(u,v) in G,E
                if v.d > u.d + w(u,v)
                        v.d = u.d+w(u,v)
                        v.source = u
For each edge
        if v.d > u.d +w(u,v)
                return false
Return true
```

Bellman-Ford algorithm

Initialize single source (G, s)

For I = 1 to |G,V|-1

    for each edge(u,v) in G,E

        if v.d > u.d + w(u,v)

            v.d = u.d+w(u,v)

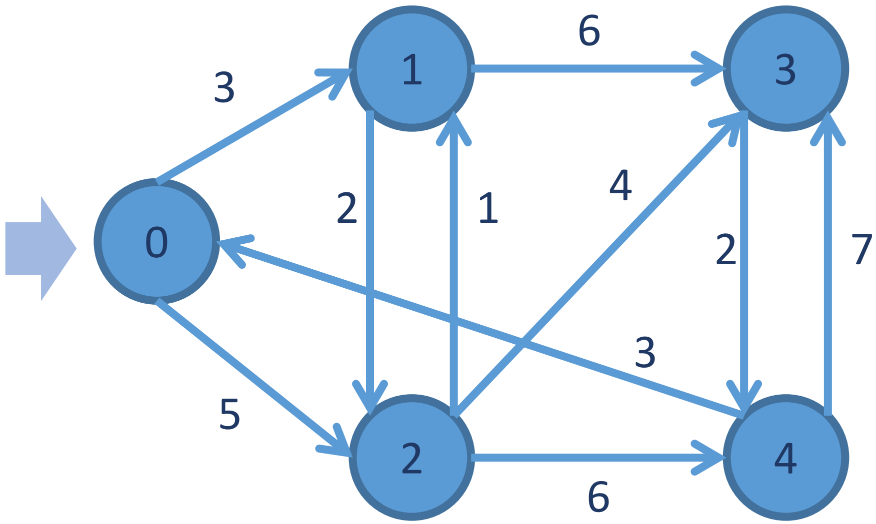
            v.source = u

For each edge

    if v.d > u.d +w(u,v)

        return false

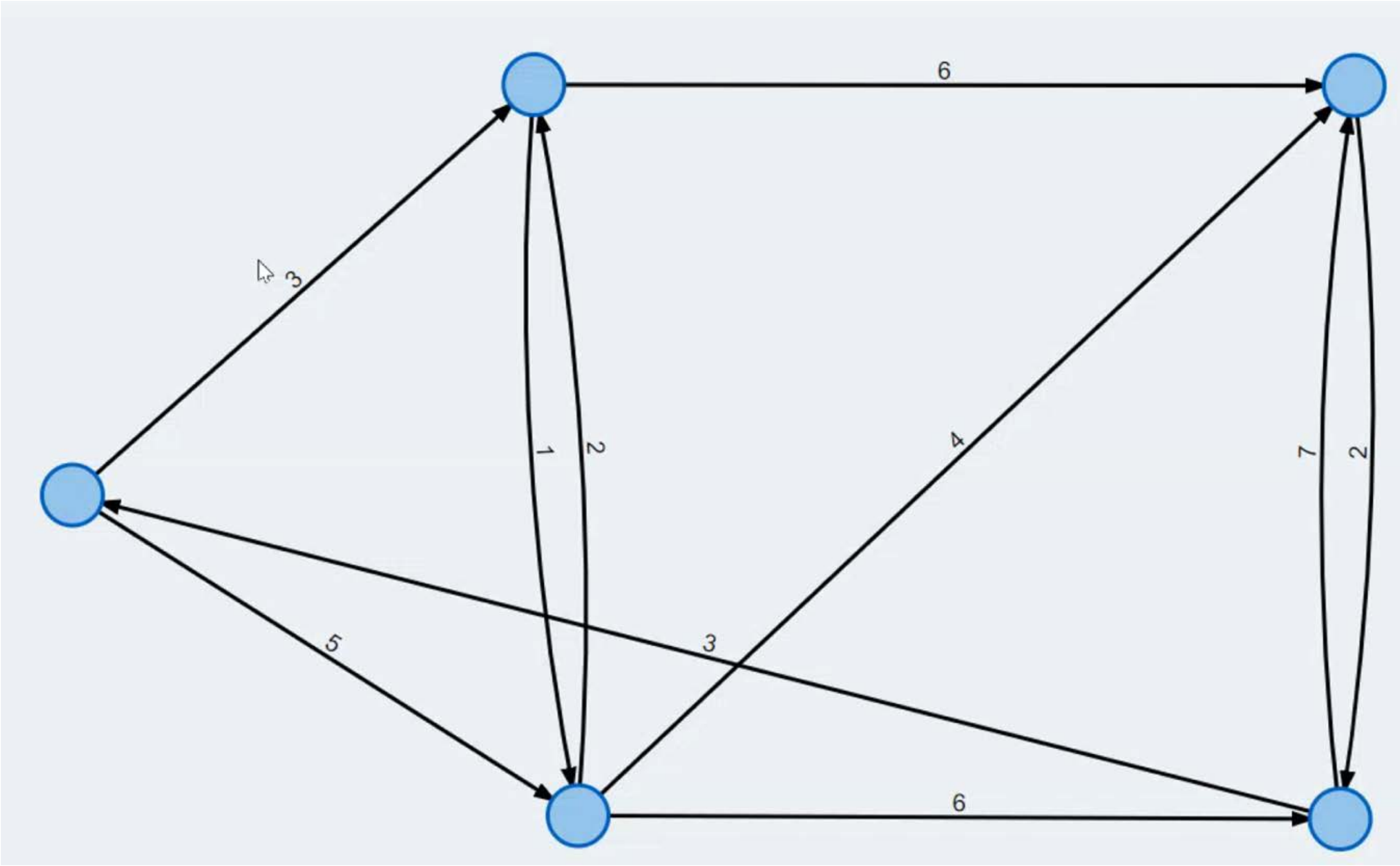
Return true



https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html

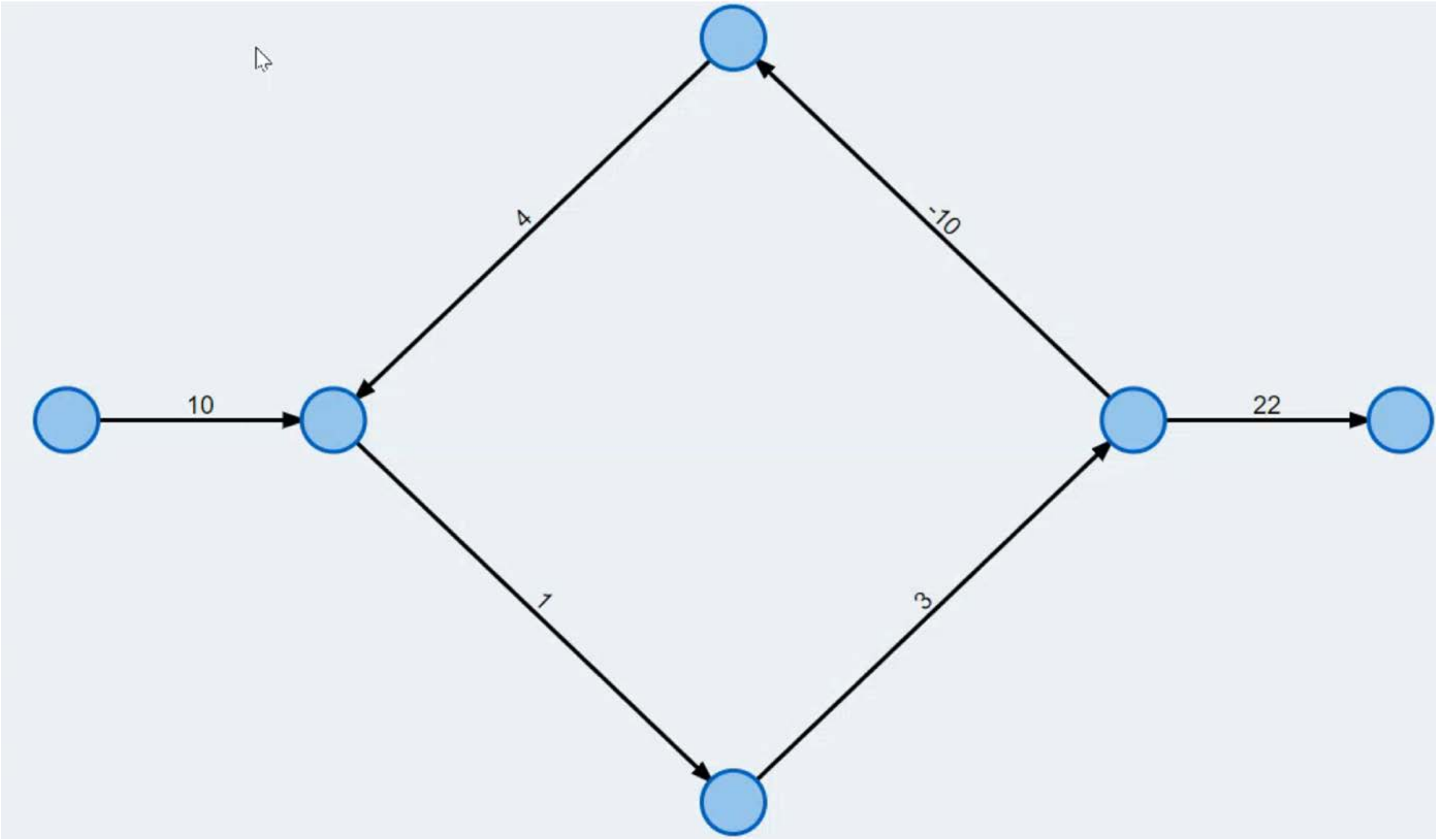
Bellman-Ford algorithm

When a graph has only positive cycles.



Bellman-Ford algorithm

When a graph has a negative cycle.



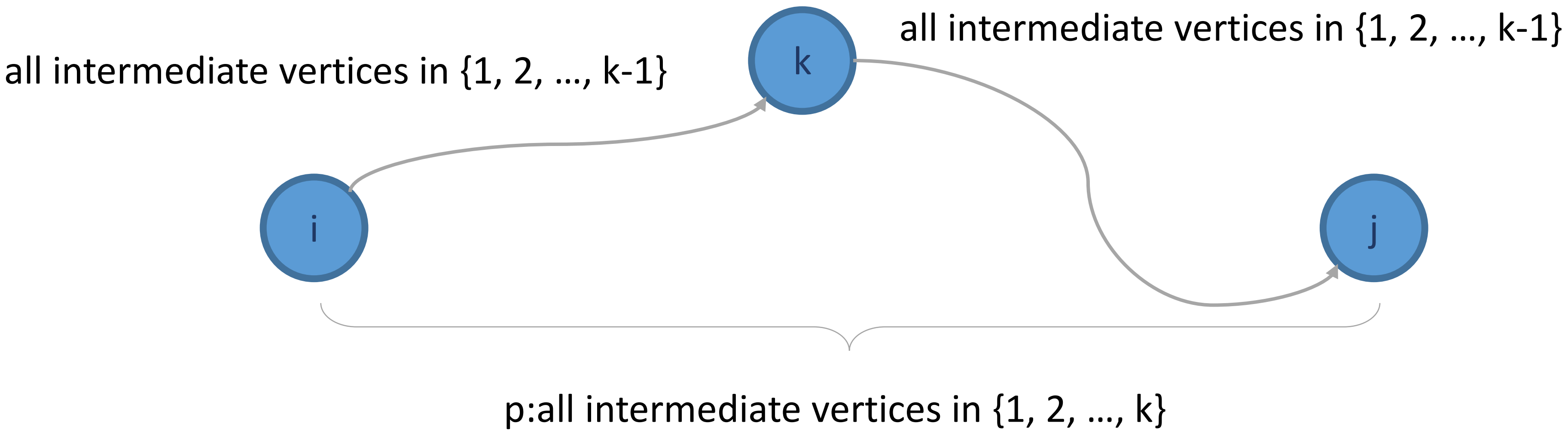
Bellman-Ford algorithm

- Time complexity: $\Theta(|V||E|) = O(n^3)$
  - $|V|$ or n is number of vertices
  - $|E|$ is number of edges.
  - Worst case: a complete graph
    - the value of $|E|$ becomes $\Theta(|V|^2)$.
    - $\Theta(|V||E|) = \Theta(|V|^3)$.

Floyd Warshall algorithm

- Condition
  - For dense graph
  - Find the shortest path between every pair of vertices
- $d_{ij}^{(k)}$: the weight of a shortest path from vertex *i* to *j*

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & if\ k = 0 \\ \min\{d_{ij}^{(k-1)}, d_{ij}^{(k-1)} + d_{ij}^{(k-1)}\} & if\ k \geq 1 \end{cases}$$



Floyd Warshall algorithm

- N = W.rows
- D(0)= W
- For k = 1 to n
  - Let D(k) = ($d_{ij}^{(k)}$) be a new nXn matrix
  - For i=1 to n
    - For j = 1 to n
      - $d_{ij}^{(k)}$= $\min\{d_{ij}^{(k-1)}, d_{ij}^{(k-1)} + d_{ij}^{(k-1)}\}$
- Return $D^{(n)}$

https://www.cs.usfca.edu/~galles/visualization/Floyd.html

Floyd Warshall algorithm

- Time complexity
  - $O(|V|^3)$
  - for each source vertex
  - for each destination vertex
  - for each intermediate vertex

Thanks