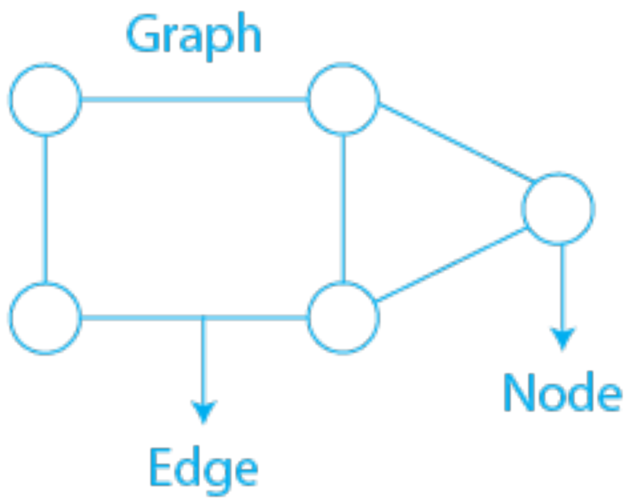
Graphs

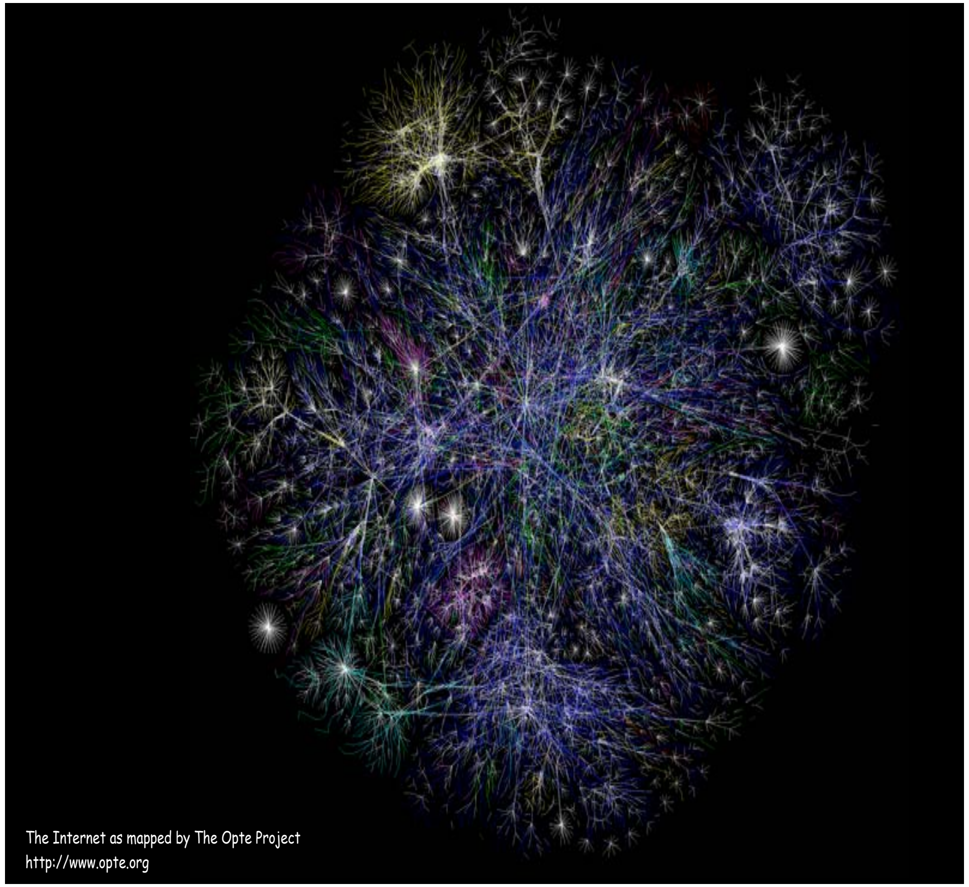
Graph Data Structure

- A graph is a nonlinear data structure composed of objects (called nodes or vertices) connected to each other by edges



Graph Applications

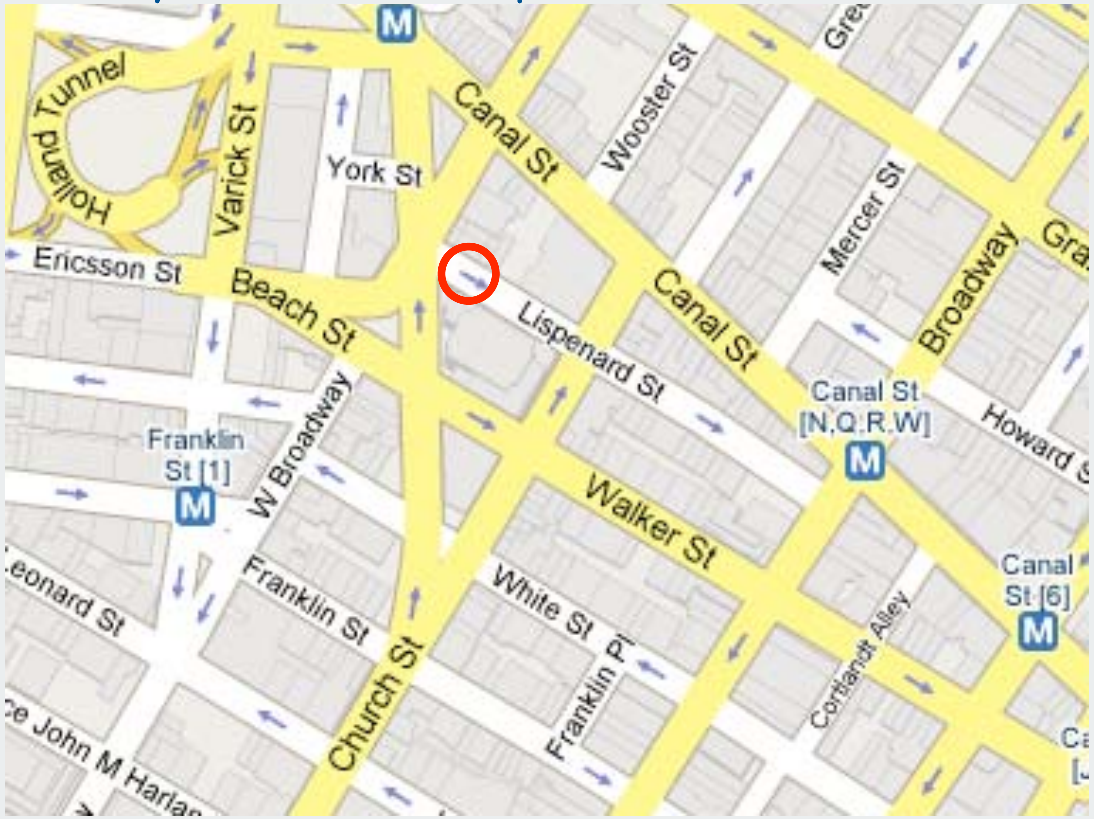
Internet Graph



Graph Applications

Map Graph

one-way streets in a map



Graph Applications

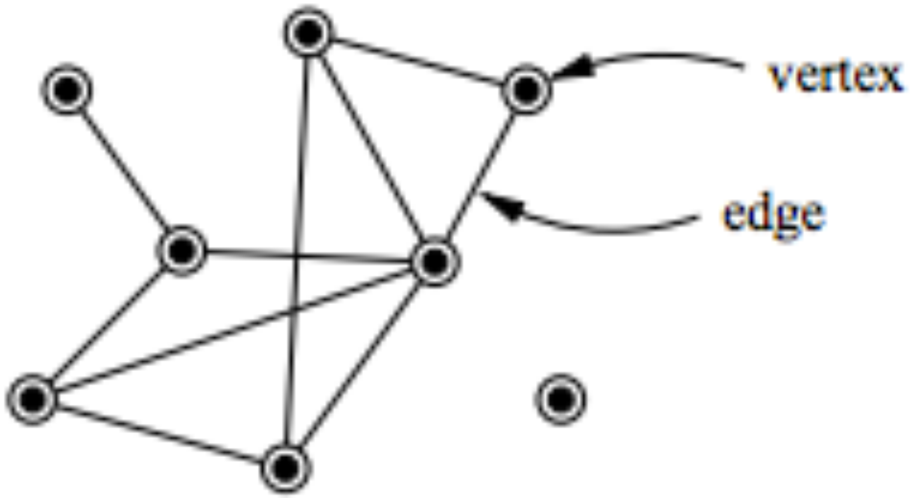
Social Network Graph

Graph Applications

graph	vertices	edges
communication	telephones, computers	fiber optic cables
circuits	gates, registers, processors	wires
mechanical	joints	rods, beams, springs
hydraulic	reservoirs, pumping stations	pipelines
financial	stocks, currency	transactions
transportation	street intersections, airports	highways, airway routes
scheduling	tasks	precedence constraints
software systems	functions	function calls
internet	web pages	hyperlinks
games	board positions	legal moves
social relationship	people, actors	friendships, movie casts
neural networks	neurons	synapses
protein networks	proteins	protein-protein interactions
chemical compounds	molecules	bonds

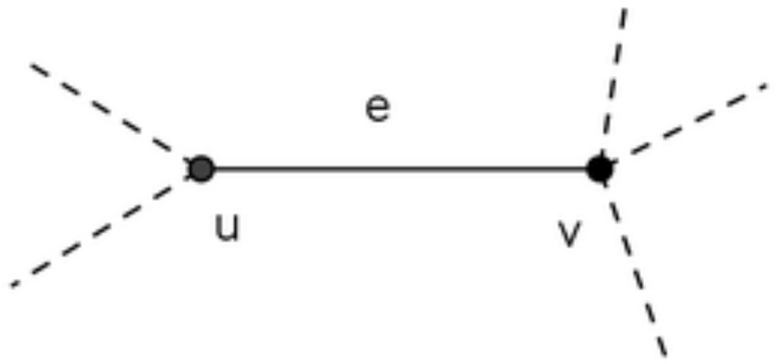
Graphs

- A graph is a set of vertices (singular : vertex) and a collection of edges that each connect a pair of vertices.
- G =(V,E)



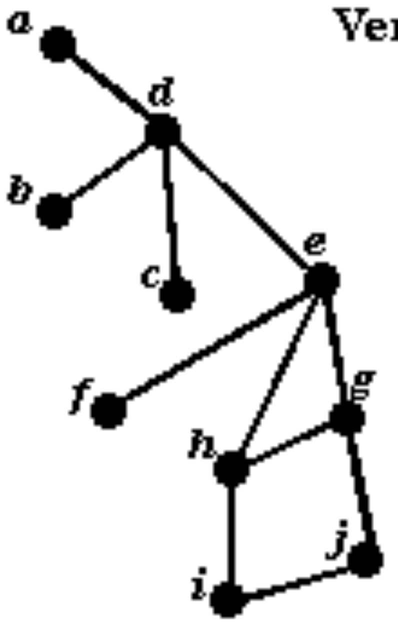
Graph Terminology

- When there is an edge connecting two vertices, we say that the vertices are adjacent to one another. The edge is incident to both vertices:
  - G=(V,E)
  - e = {u,v} is incident to u and v or joins u and v



Graph Terminology

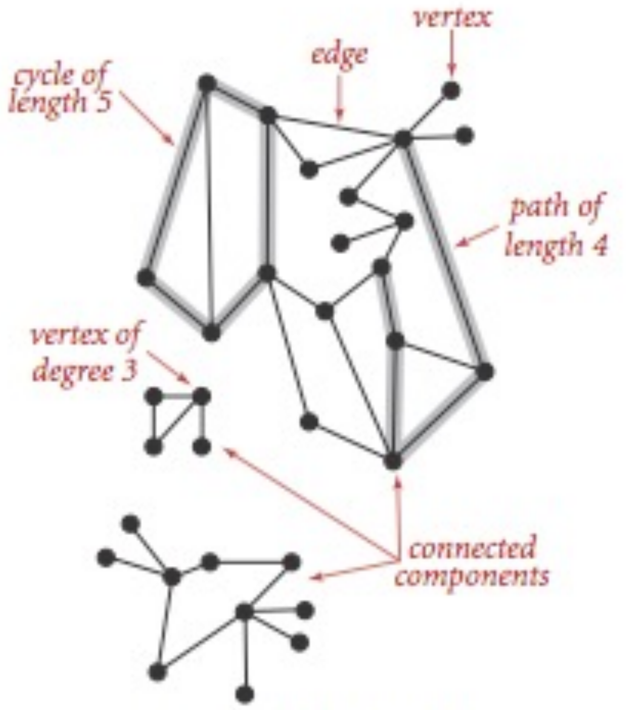
- The degree of a vertex is the number of edges incident to it.



Vertex	Degree
a	1
b	1
c	1
d	4
e	4
f	1
g	3
h	3
i	2
j	2

Graph Terminology

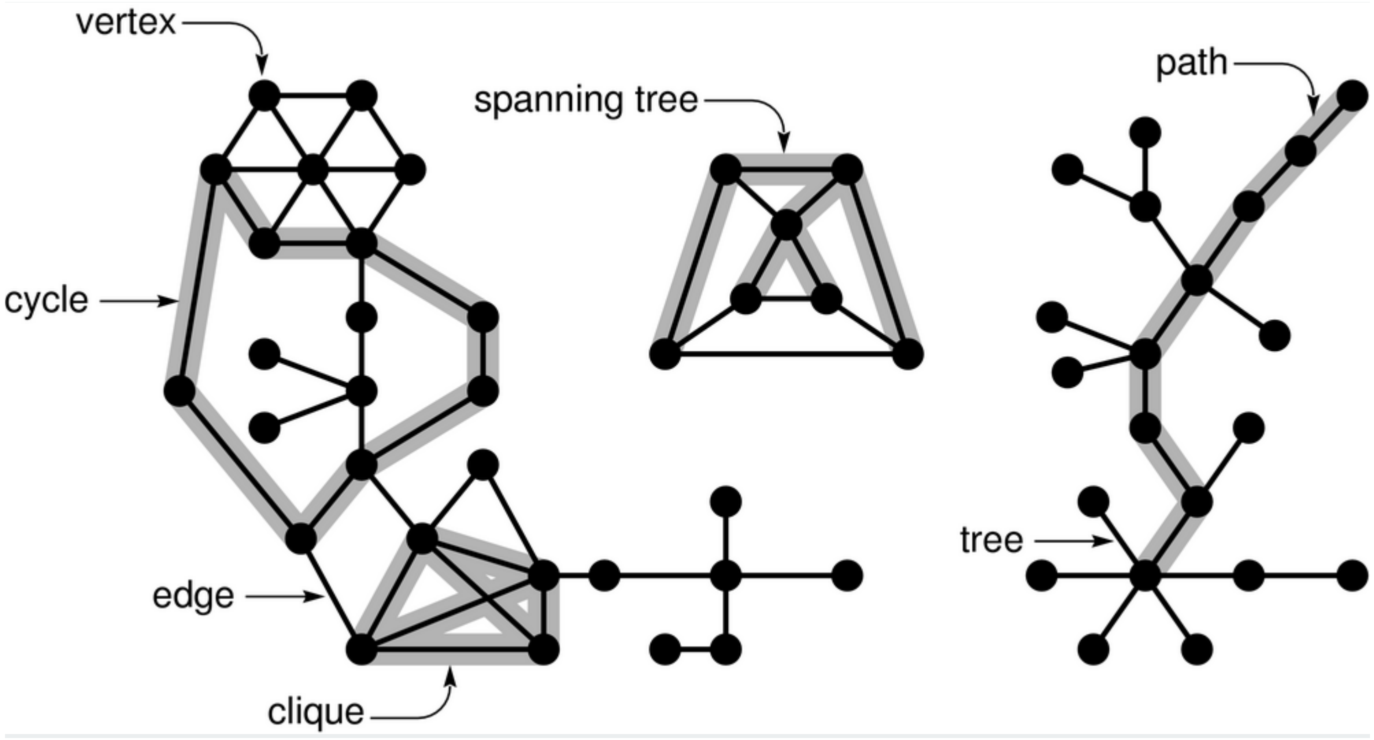
- A path is a sequence of vertices connected by edges
- The length of a path is its number of edges



Anatomy of a graph

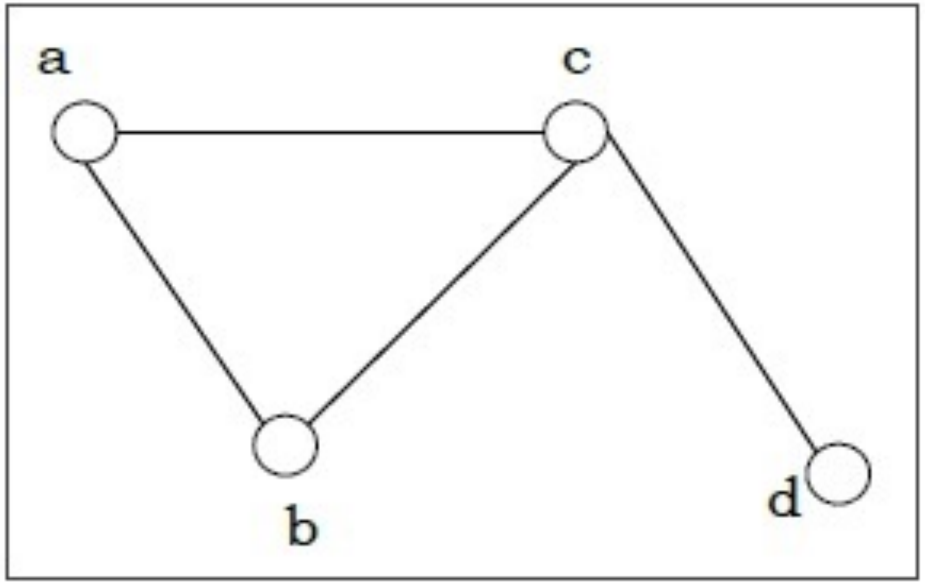
Graph Terminology

- An undirected graph is a tree if it is connected and does not contain a cycle



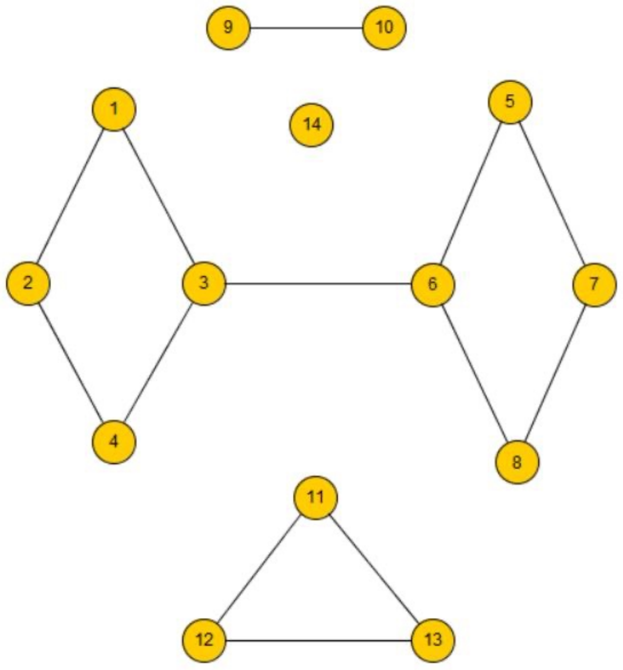
Graph Terminology

- A graph is connected if any two vertices of the graph are connected by a path.



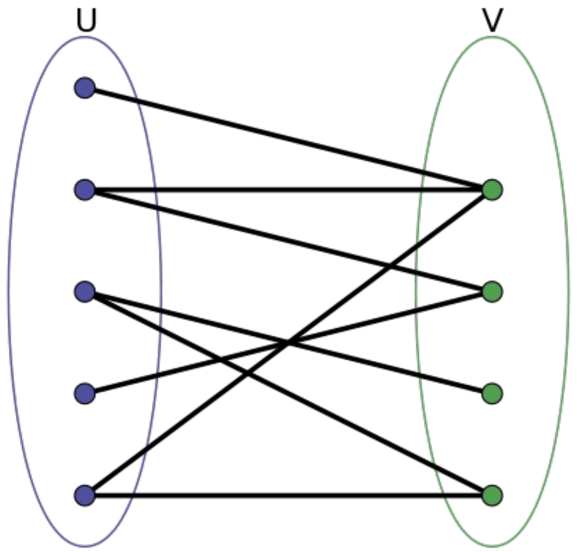
Graph Terminology

- A graph is disconnected, at least two vertices of the graph are not connected by a path.



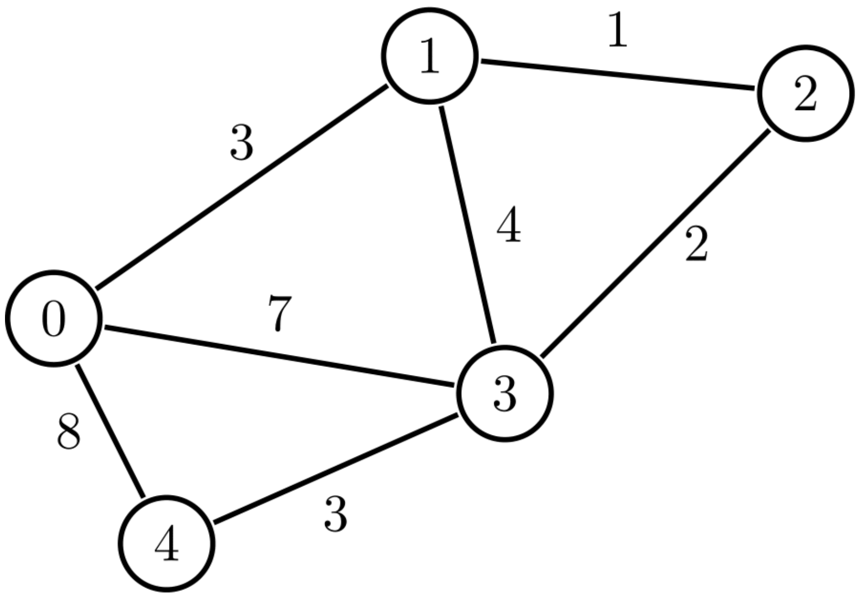
Graph Terminology

- A bipartite graph, is a special kind of graph with the following properties:
  - It consists of two sets of vertices U and V.
  - The vertices of set U join only with the vertices of set V.
  - The vertices within the same set do not join.



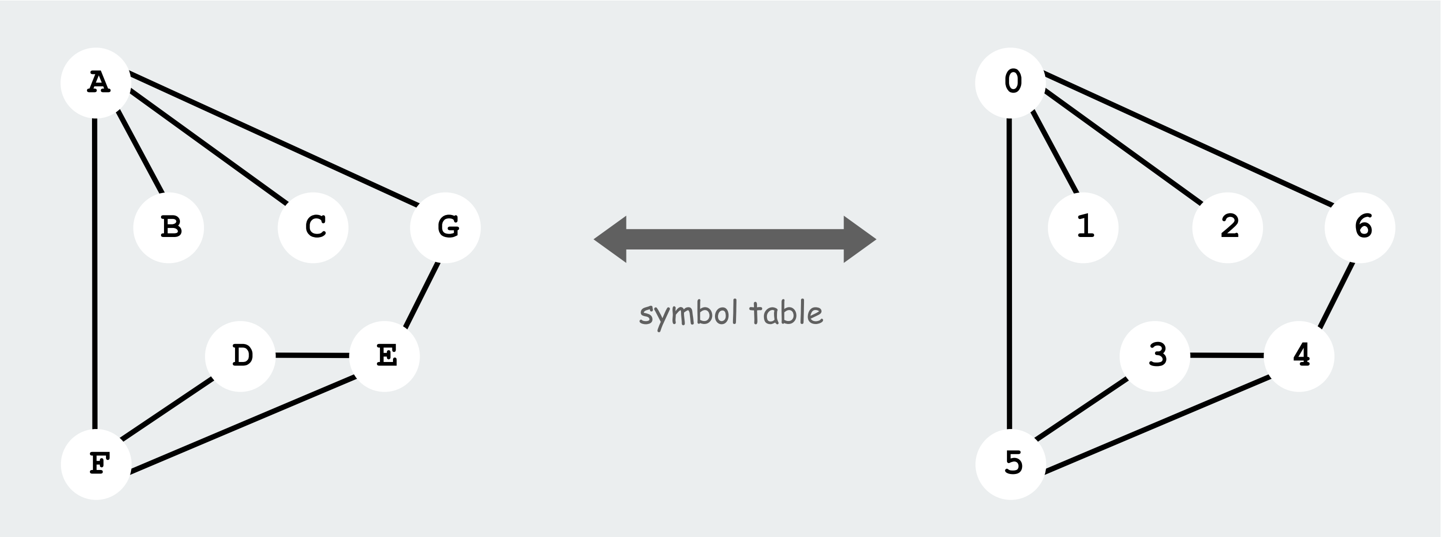
Graph Terminology

- An edge-weighted graph is a graph where we associate weights or costs with each edge



Undirected Graph Representation

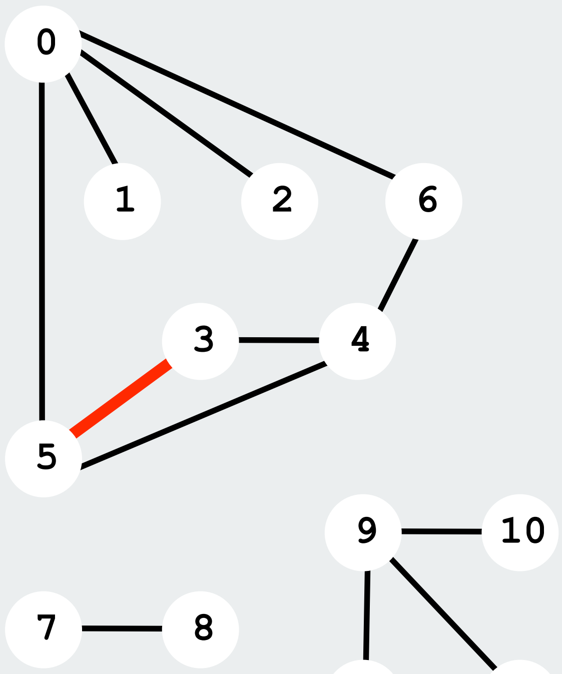
- Computer : use integers between 0 and V-1
- Real world: convert between names and integers with symbol table



Undirected Graph - Adjacency Matrix

Maintain a two-dimensional $V \times V$ boolean array.

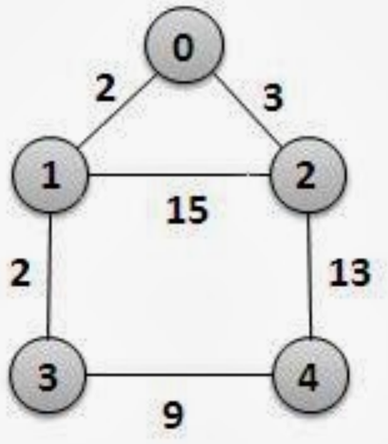
For each edge `v-w` in graph: `adj[v][w] = adj[w][v] = true.`



two entries for each edge

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Undirected Weighted Graph - Adjacency Matrix

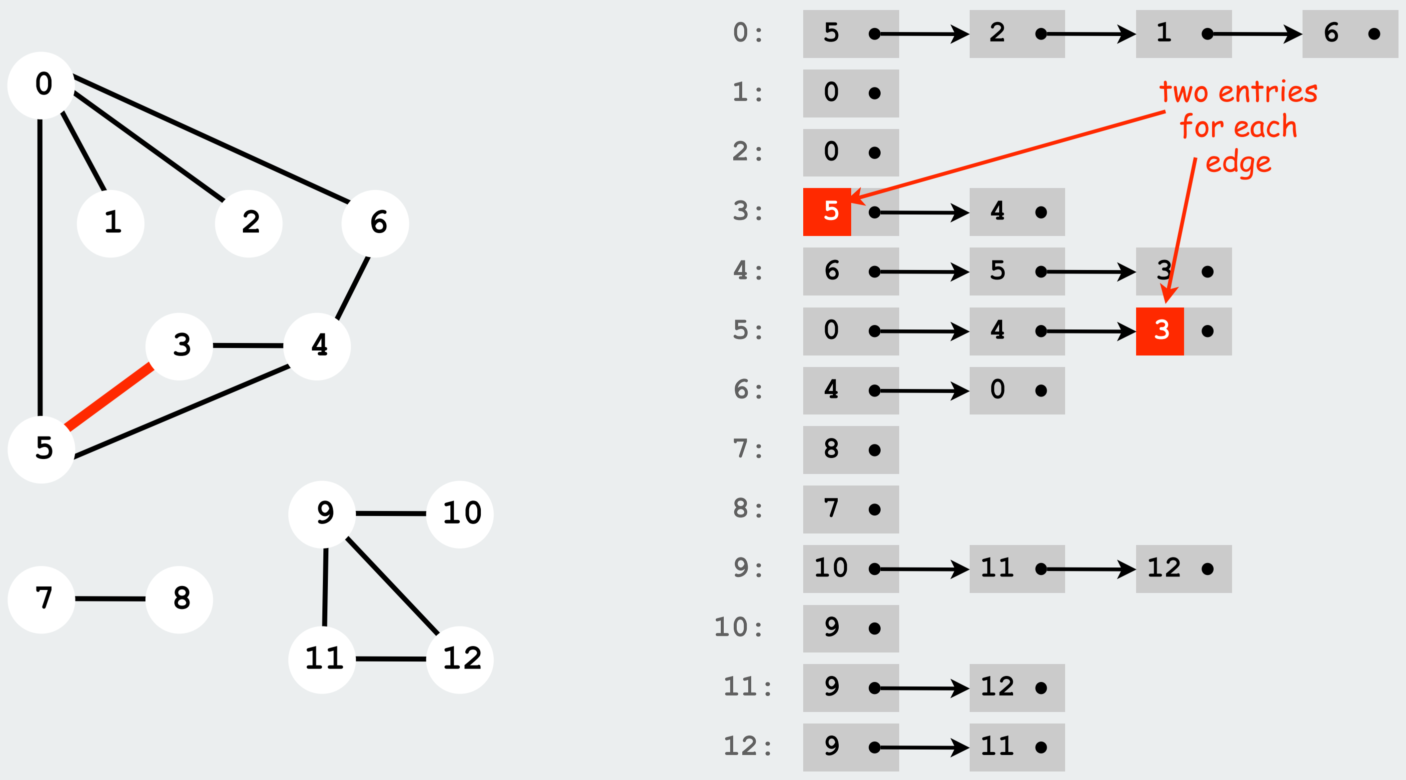


	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

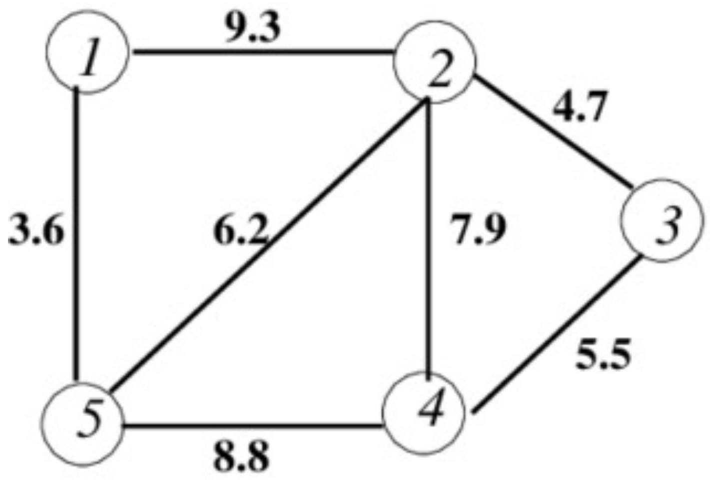
Adjacency Matrix Representation of Weighted Graph

Undirected Graph - Adjacency List

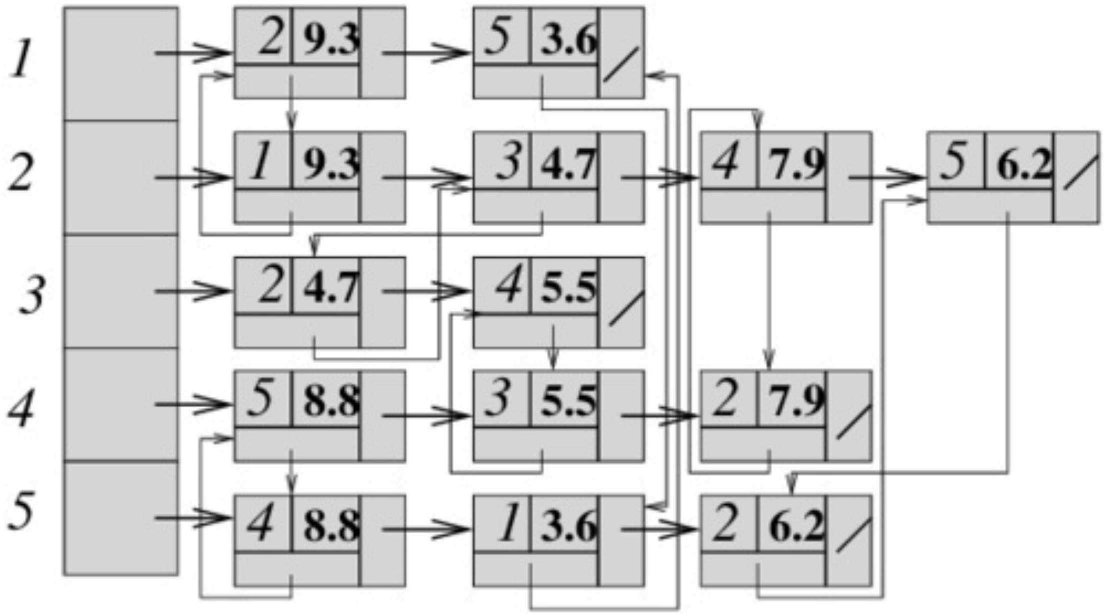
Maintain vertex-indexed array of lists (implementation omitted)



Undirected Weighted Graph - Adjacency List



(a)



(b)

Adjacency List in C

```
struct node
{
        int vertex;
        struct node* next;
};
struct Graph
{
        int numVertices;
        struct node** adjLists;
};
```

Adjacency List in C

```
struct node* createNode(int v)
{
        struct node* newNode = malloc(sizeof(struct node));
        newNode->vertex = v;
        newNode->next = NULL;
        return newNode;
}
```

Adjacency List in C

```
struct Graph* createGraph(int vertices)
{
        int i;
        struct Graph* graph = malloc(sizeof(struct Graph));
        graph->numVertices = vertices;
        graph->adjLists = malloc(vertices * sizeof(struct node*));
        for (i = 0; i < vertices; i++)
                graph->adjLists[i] = NULL;
        return graph;
}
```

Adjacency List in C

```
void addEdge(struct Graph* graph, int src, int dest)
{
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
```

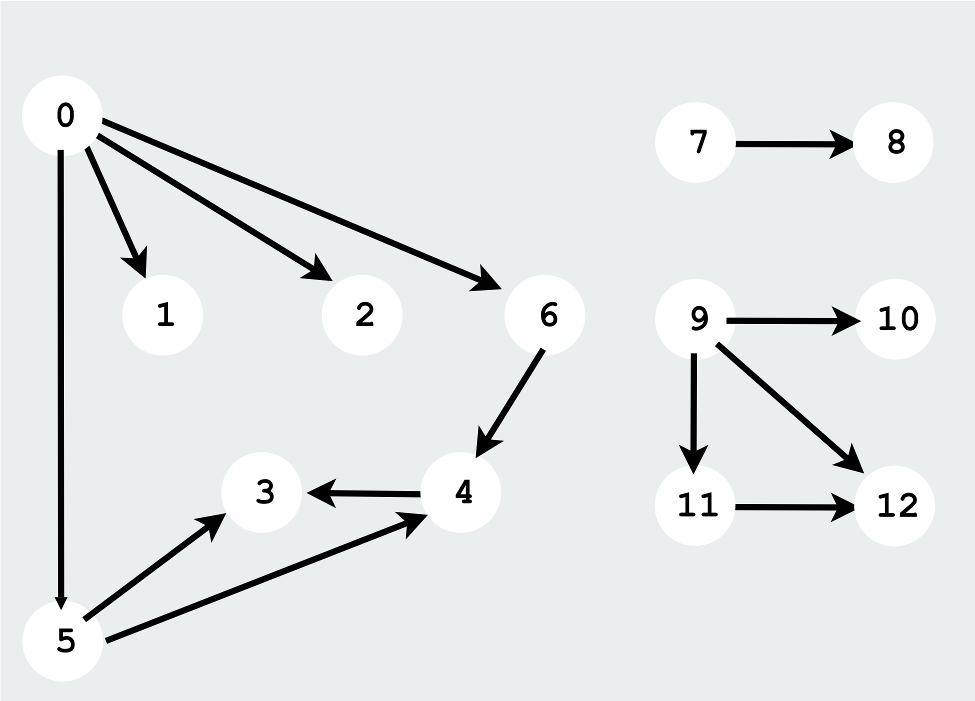
Graph Representations

Representation	Space	Edge between v and w?	Edge from v to anywhere?	Enumerate all edges
Adjacency matrix	$O(V^2)$	$O(1)$	$O(V)$	$O(V^2)$
Adjacency list	$O(E + V)$	$O(E)$	$O(1)$	$O(E + V)$

E : number of edges
V : number of vertices

Directed Graphs

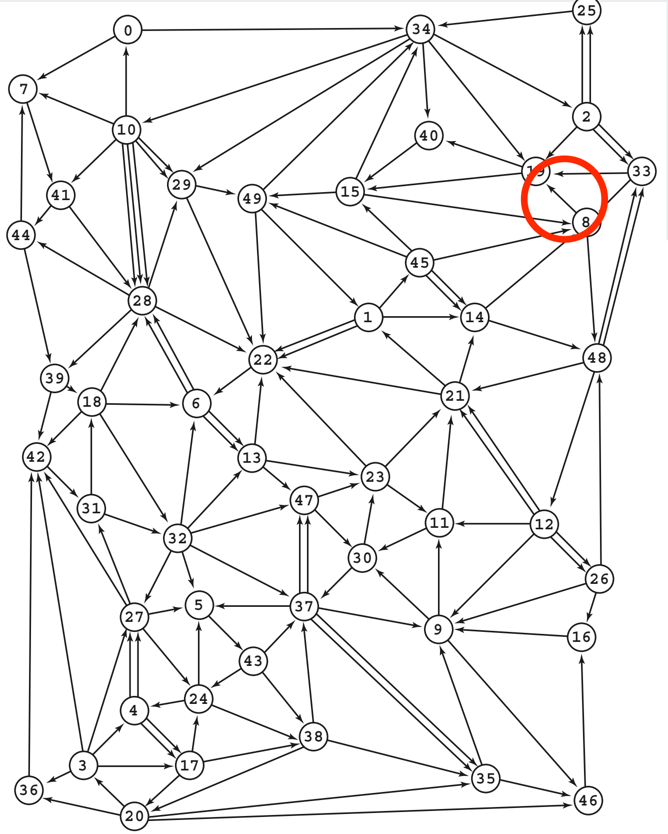
- A directed graph is a set of vertices and a collection of directed edges.
- Edges are one way



Directed Graph Applications

one-way streets in a map

hyperlinks connecting web pages



Directed Graph Applications

digraph	vertex	edge
financial	stock, currency	transaction
transportation	street intersection, airport	highway, airway route
scheduling	task	precedence constraint
WordNet	synset	hypernym
Web	web page	hyperlink
game	board position	legal move
telephone	person	placed call
food web	species	predator-prey relation
infectious disease	person	infection
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Directed Graph Representation

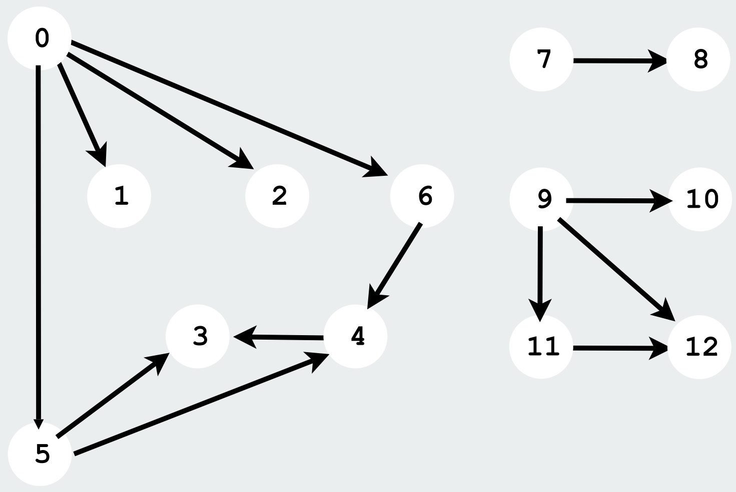
Edges: four easy options

- list of vertex pairs
- vertex-indexed adjacency arrays (adjacency matrix)
- vertex-indexed adjacency lists
- vertex-indexed adjacency SETs

Same as undirected graph

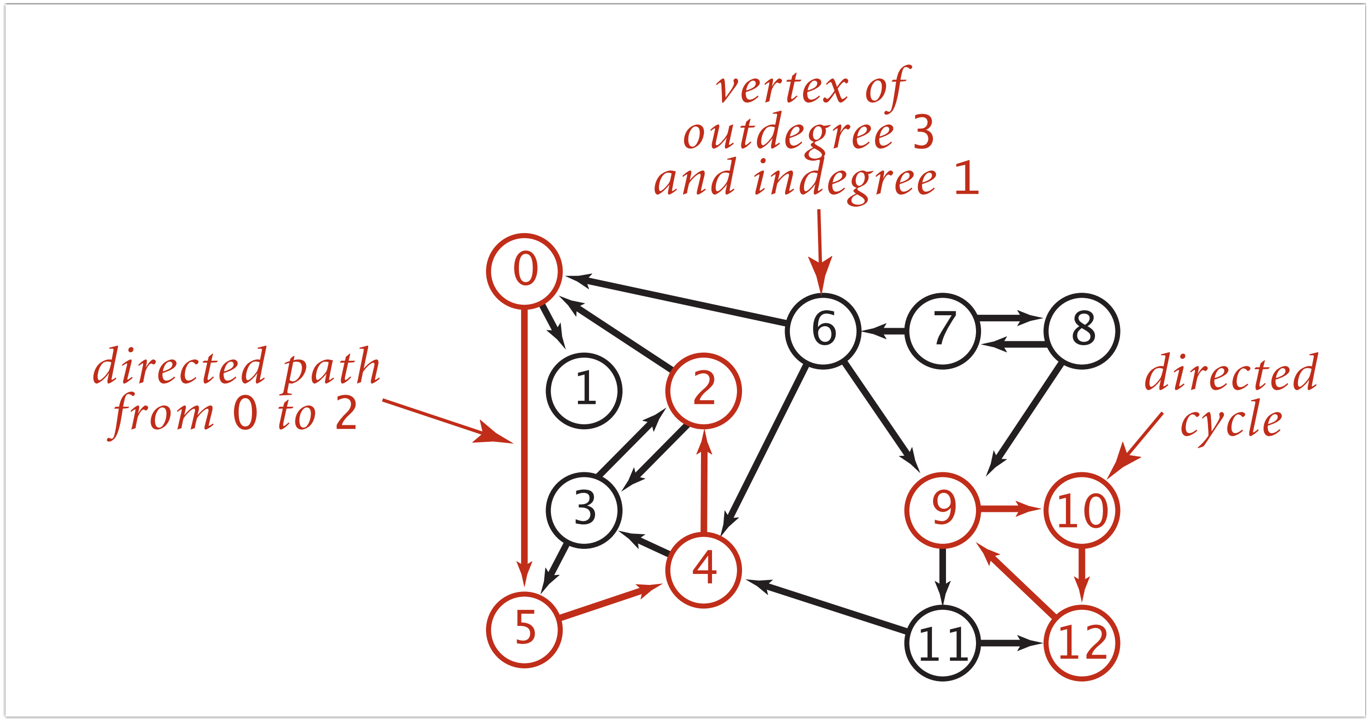
BUT

orientation of edges is significant.



Directed Graphs

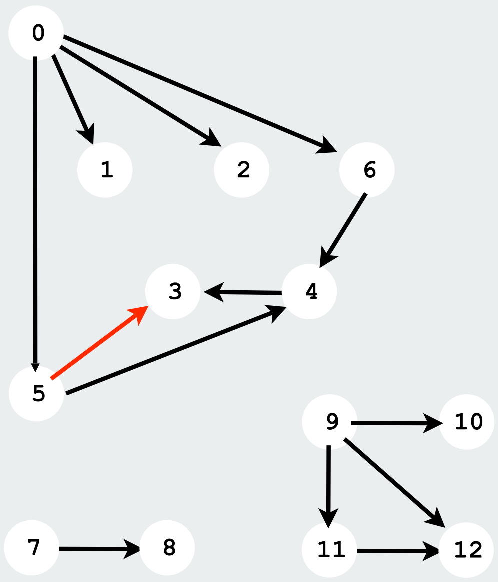
- For a vertex :
  - the number of head ends adjacent to a vertex is called the indegree of the vertex
  - the number of tail ends adjacent to a vertex is its outdegree



Adjacency Matrix - Digraph Representation

Maintain a two-dimensional $V \times V$ boolean array.

For each edge `v→w` in graph: `adj[v][w] = true`.



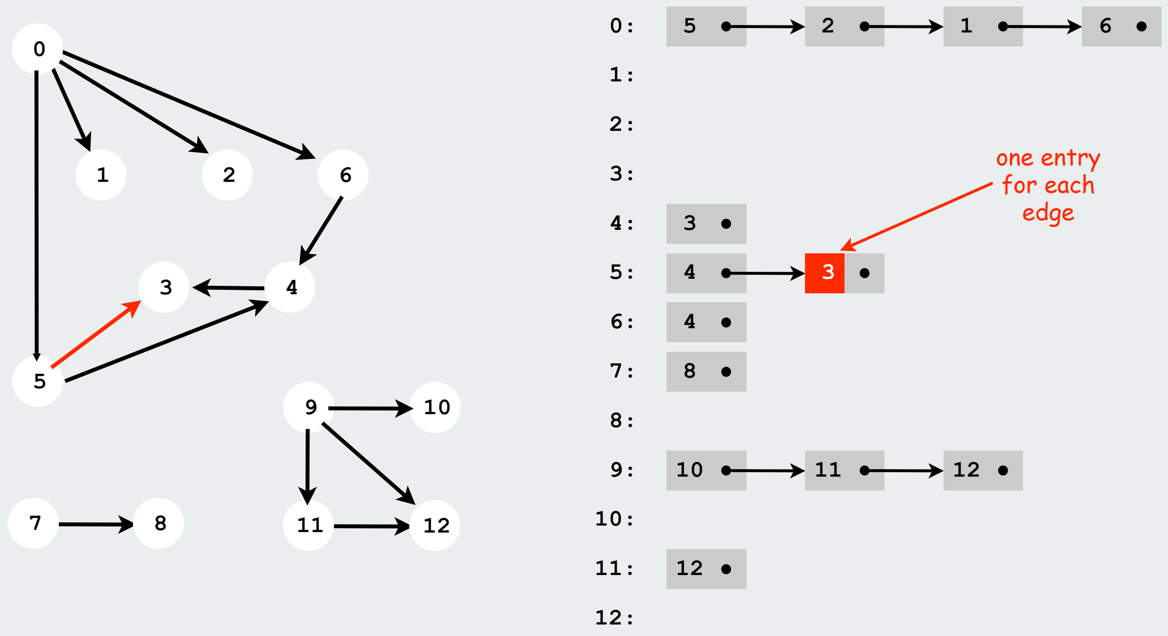
one entry for each edge

to

from	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0	0	0	0	0
5	0	0	0	1	1	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0	0	0	0

Adjacency List - Digraph Representation

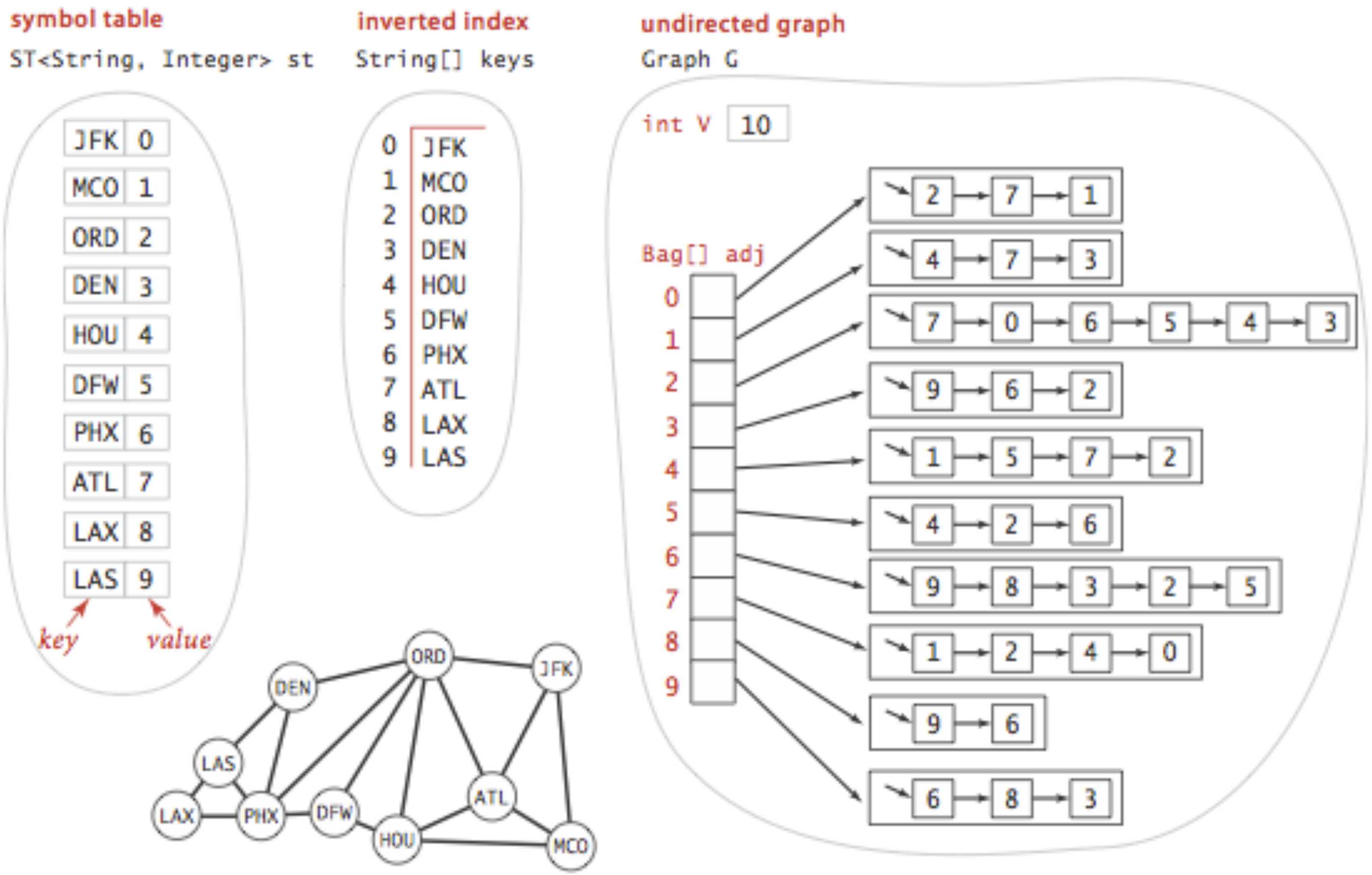
Maintain vertex-indexed array of lists.



Symbol graphs

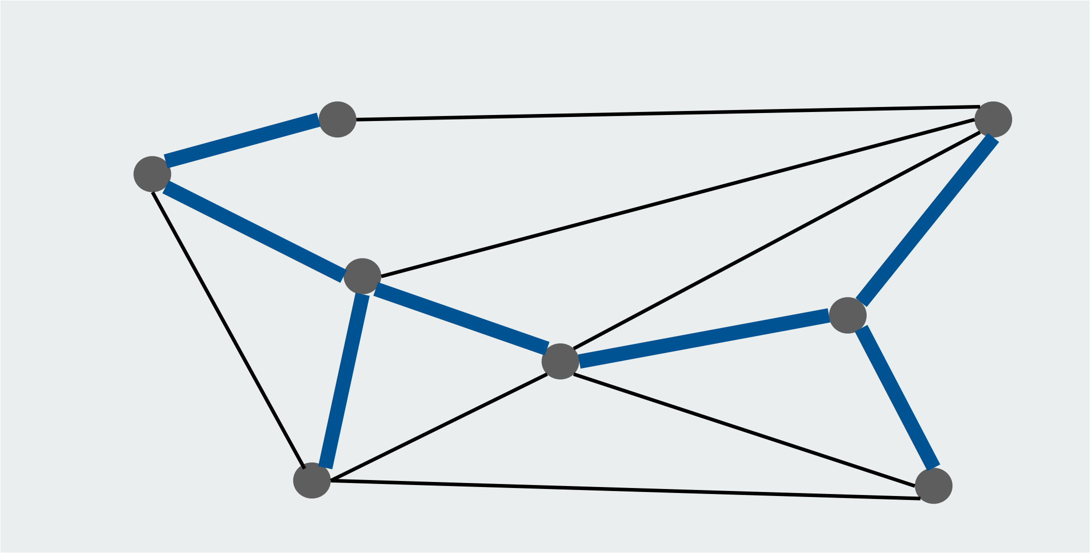
- Typical applications involve processing graphs using strings, not integer indices, to define and refer to vertices.
- Define an input format with the following properties:
  - Vertex names are strings.
  - A specified delimiter separates vertex names (to allow for the possibility of spaces in names).
  - Each line represents a set of edges, connecting the first vertex name on the line to each of the other vertices named on the line.

Symbol graphs



Minimum Spanning Tree

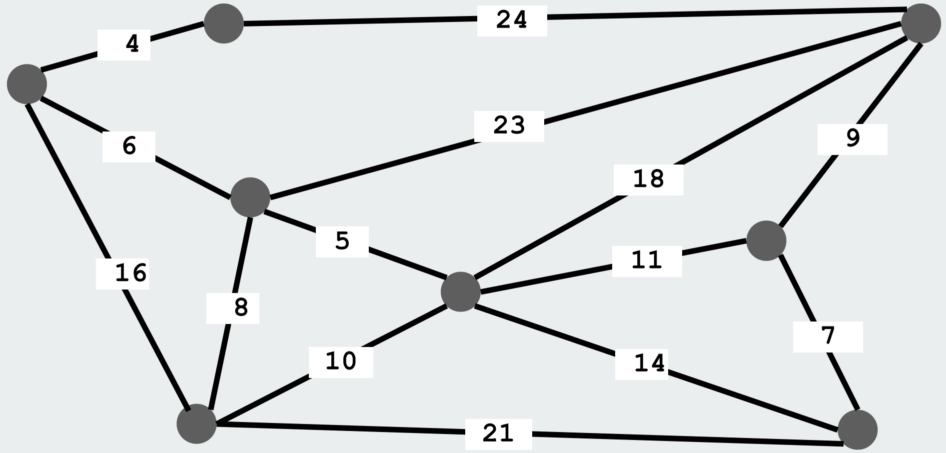
- Tree : an undirected and an acyclic graph
- Spanning Tree : A tree, which contains all the vertices of the graph
- Minimum Spanning Tree : Spanning tree with the minimum sum of weights



Minimum Spanning Tree

Given. Undirected graph G with positive edge weights (connected).

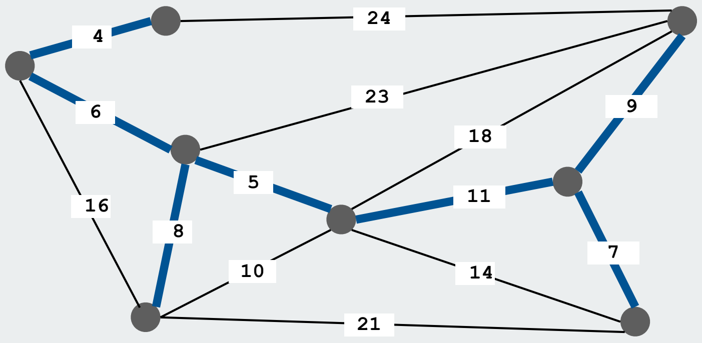
Goal. Find a min weight set of edges that connects all of the vertices.



G

Minimum Spanning Tree

Given. Undirected graph G with positive edge weights (connected).
Goal. Find a min weight set of edges that connects all of the vertices.



weight(T) = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

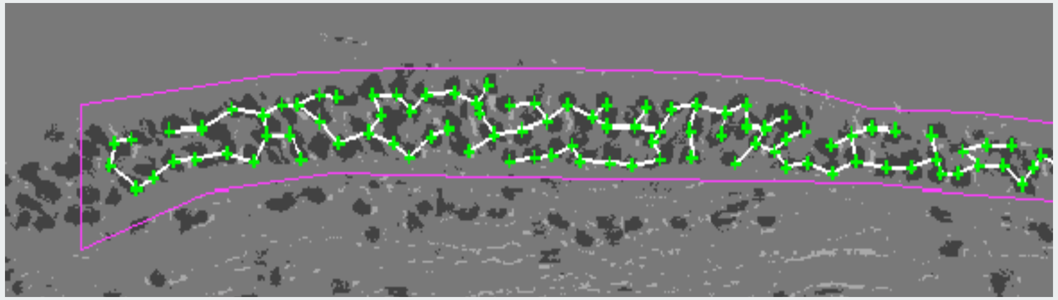
Brute force: Try all possible spanning trees

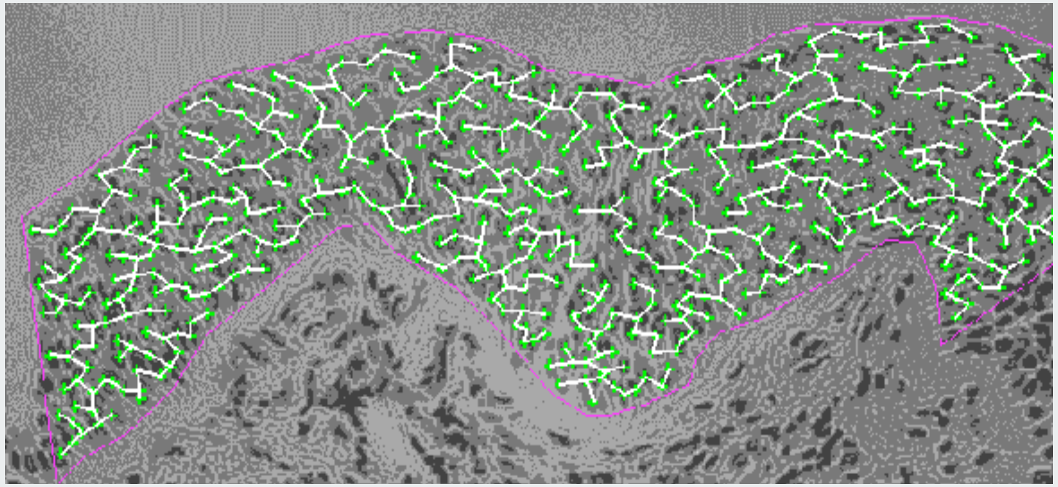
- problem 1: not so easy to implement
- problem 2: far too many of them

Ex: [Cayley, 1889]: V^{V-2} spanning trees on the complete graph on V vertices.

Application – Medical Image Processing

MST describes arrangement of nuclei in the epithelium for cancer research





http://www.bccrc.ca/ci/ta01_archlevel.html

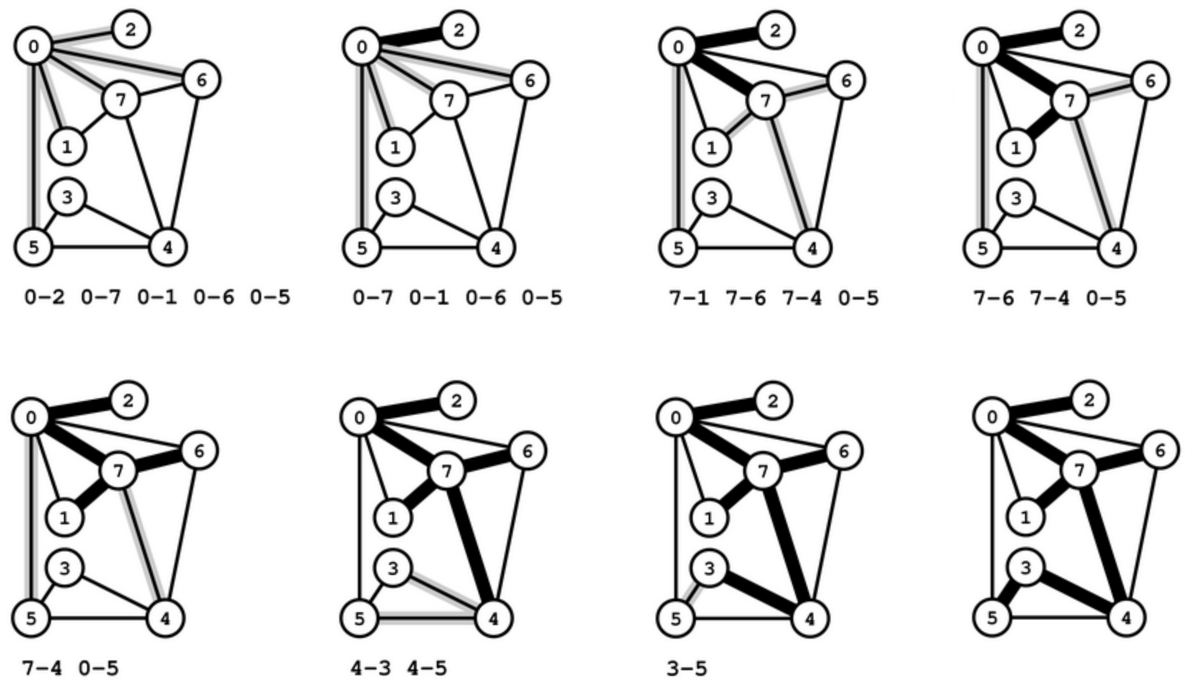
Prim’s Algorithm

- Greedy Strategy :
  - Select the best local option from all available choices without regard for global structures
  - A locally optimal choice is globally optimal

- Start from one node
- Grows the MST one edge at a time until all nodes are included
  - Always choose the edge which contributes the minimum amount possible

Prim’s Algorithm Example

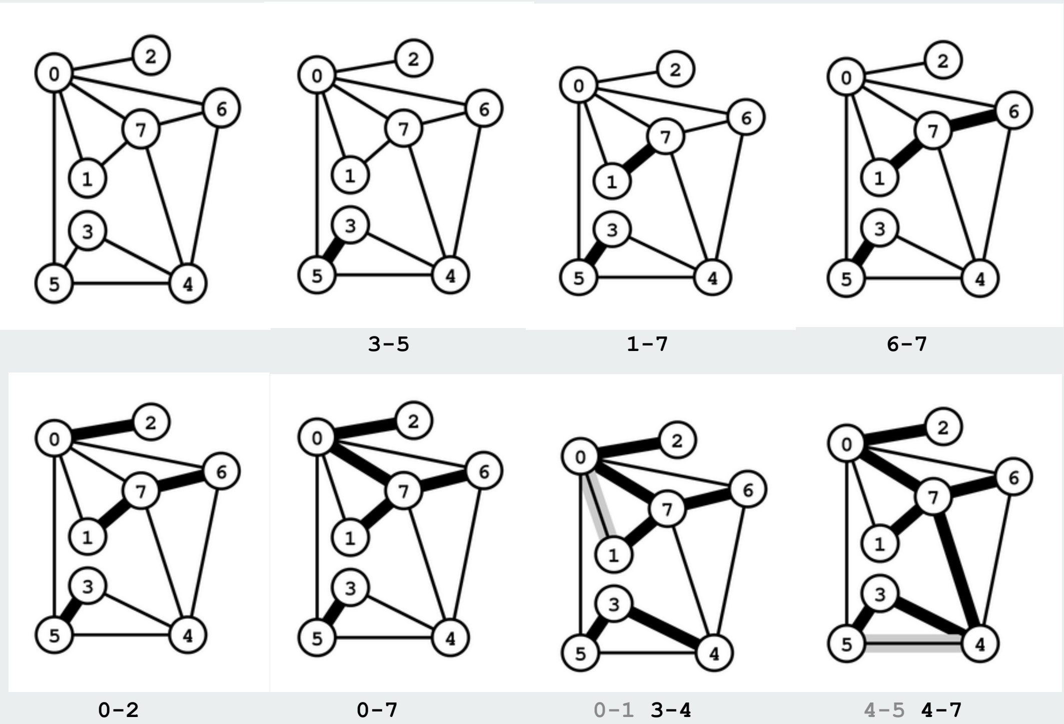
Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]
Start with vertex 0 and greedily grow tree T. At each step, add cheapest edge that has exactly one endpoint in T.



0-1	0.32
0-2	0.29
0-5	0.60
0-6	0.51
0-7	0.31
1-7	0.21
3-4	0.34
3-5	0.18
4-5	0.40
4-6	0.51
4-7	0.46
6-7	0.25

Kruskal Algorithm

Kruskal's algorithm. [Kruskal, 1956] Consider edges in ascending order of cost. Add the next edge to T unless doing so would create a cycle.



3-5	0.18
1-7	0.21
6-7	0.25
0-2	0.29
0-7	0.31
0-1	0.32
3-4	0.34
4-5	0.40
4-7	0.46
0-6	0.51
4-6	0.51
0-5	0.60

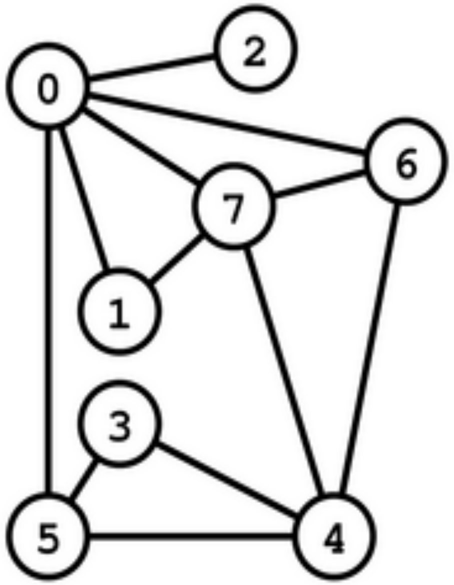
Kruskal Algorithm

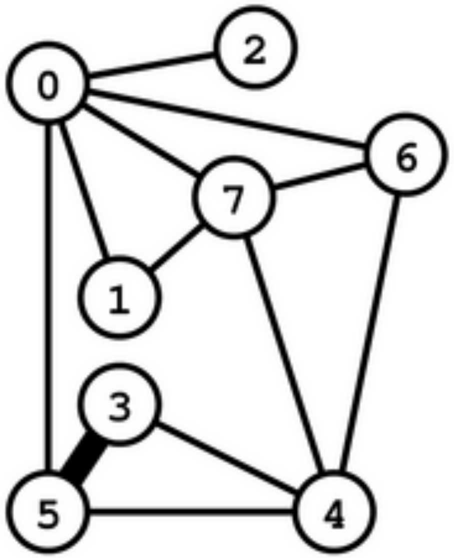
- G = (V,E)   n :number or vertices
- MST has exactly n-1 edges
- Arrange E in the order of increasingly costs

```
for(i=1; i<=n-1; i++)
{
        select the next smallest cost edge
        if the edge connects two different connected components
                add the edge to MST
}
```

Kruskal Algorithm

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

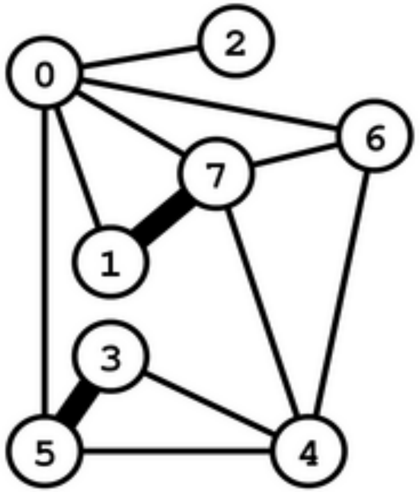





```
3-5 0.18
1-7 0.21
6-7 0.25
0-2 0.29
0-7 0.31
0-1 0.32
3-4 0.34
4-5 0.40
4-7 0.46
0-6 0.51
4-6 0.51
0-5 0.60
```

Kruskal Algorithm

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



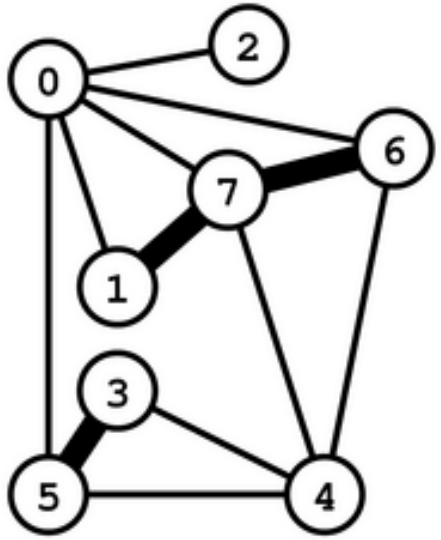
1-7

3-5 0.18
1-7 0.21
6-7 0.25
0-2 0.29
0-7 0.31
0-1 0.32
3-4 0.34
4-5 0.40
4-7 0.46
0-6 0.51
4-6 0.51
0-5 0.60

Kruskal Algorithm

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

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



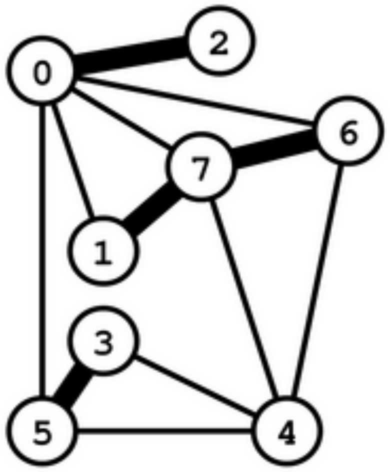
6-7

3-5 0.18
1-7 0.21
6-7 0.25
0-2 0.29
0-7 0.31
0-1 0.32
3-4 0.34
4-5 0.40
4-7 0.46
0-6 0.51
4-6 0.51
0-5 0.60

Kruskal Algorithm

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

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



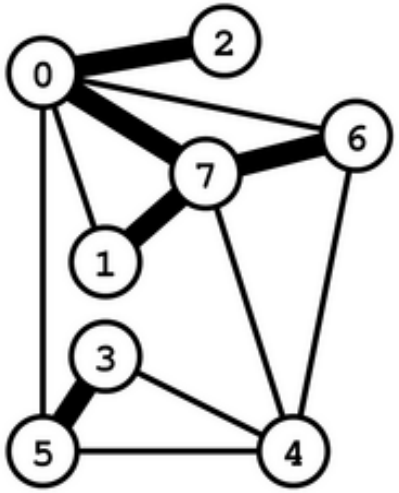
0–2

3–5 0.18
1–7 0.21
6–7 0.25
0–2 0.29
0–7 0.31
0–1 0.32
3–4 0.34
4–5 0.40
4–7 0.46
0–6 0.51
4–6 0.51
0–5 0.60

Kruskal Algorithm

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

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



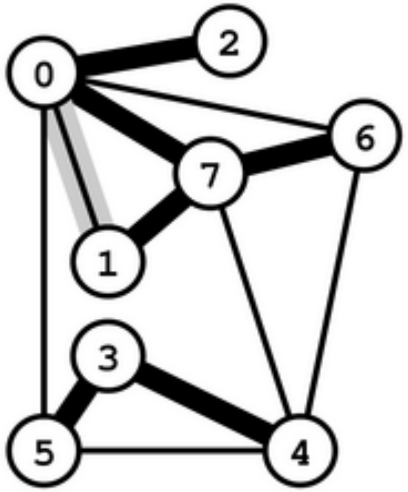
0-7

3-5 0.18
1-7 0.21
6-7 0.25
0-2 0.29
0-7 0.31
0-1 0.32
3-4 0.34
4-5 0.40
4-7 0.46
0-6 0.51
4-6 0.51
0-5 0.60

Kruskal Algorithm

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

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



0-1 3-4

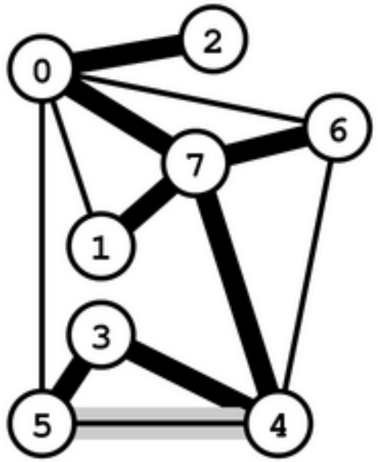
3-5 0.18
1-7 0.21
6-7 0.25
0-2 0.29
0-7 0.31
0-1 0.32
3-4 0.34
4-5 0.40
4-7 0.46
0-6 0.51
4-6 0.51
0-5 0.60

Kruskal Algorithm

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

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

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



4-5 **4-7**

3-5 0.18
1-7 0.21
6-7 0.25
0-2 0.29
0-7 0.31
0-1 0.32
3-4 0.34
4-5 0.40
4-7 0.46
0-6 0.51
4-6 0.51
0-5 0.60

Kruskal Algorithm

```
typedef struct Edge
{
	int u,v;
	int weight;
} Edge;
```

Kruskal Algorithm

```
void  kruskal(Edge gr[], mst[], int nV);
{
        int labelNo=1; i, j;
        for(i=0; i< NV ; i++)
                label[i] = 0;
        i=j=0;
        while(i< NV-1 ; i++)
        {
                uu = gr[j].u ;
                vv = gr[j].v;
                if(label[uu] + label[vv] == 0)
                {
                        mst[i].u = uu;
                        mst[i].v = vv;
                        mst[i].weight = gr[j].weight;
                        label[uu]=label[vv] = labelNo++;
                        i++;
                }
                if(label[uu] != label[vv])
                {
                        mst[i].u = uu;
                        mst[i].v = vv;
                        mst[i].weight = gr[j].weight;
                        i++;
                        if(!label[uu])
                                label[uu] = label[vv];
                        else if(!label[vv])
                                label[vv] = label[uu];
                        else
                                union(label,nV,uu,vv) ;
                }
                j++;
}
```

Kruskal Algorithm

```
void union(int label[],int nV,int uu,int vv)
{
        int i;
        for(i=0; i< nV; i++)
        {
              if(label[i] == uu)
                  label[i] = vv;
        }
}
```

MST Algorithms Running Time

- Prim's Algorithm :

Running time.

- **V - 1 iterations since each iteration adds 1 vertex.**

Each iteration consists of:

- **Choose next vertex to add to S by minimum `dist[w]` value.**
  - **O(V) time.**
- **For each neighbor w of v, if w is closer to v than to a vertex in S, update `dist[w]`.**
  - **O(V) time.**

$O(V^2)$ overall.

- Kruskal Algorithm :

Kruskal analysis. O(E log V) time.

- **`Sort()`: O(E log E) = O(E log V).**