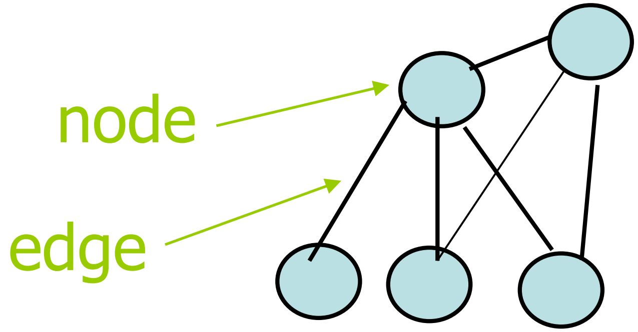
Graph

UNIT II

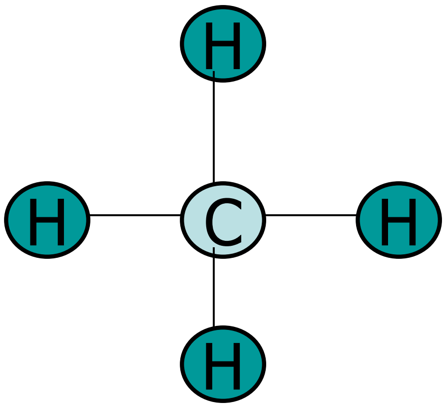
What is a graph?

- Graph is a non-linear data structure. Graphs represent the relationships among data items.
- A graph G consists of
  - a set V of nodes (vertices)
  - a set E of edges: each edge connects two nodes
- Each node represents an item
- Each edge represents the relationship between two items

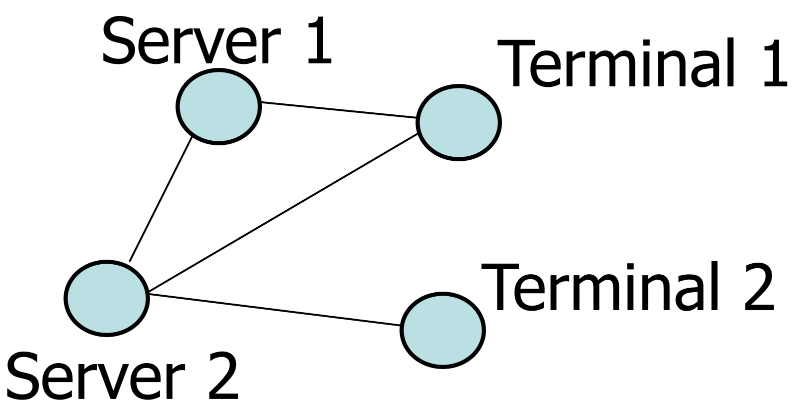


Examples of graphs

Molecular Structure



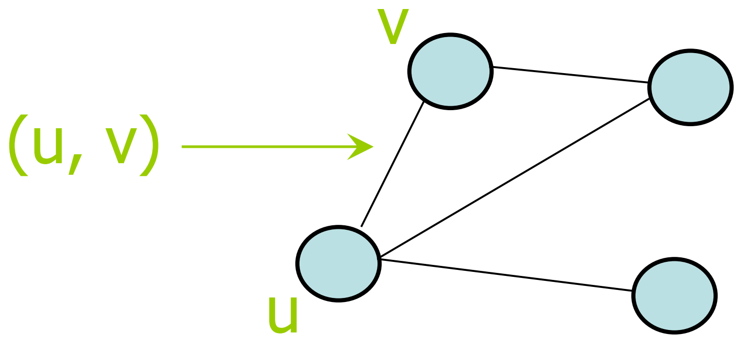
Computer Network



Other examples: electrical and communication networks, airline routes, flow chart, graphs for planning projects

Formal Definition of graph

- The set of nodes is denoted as V
- For any nodes u and v, if u and v are connected by an edge, such edge is denoted as (u, v)

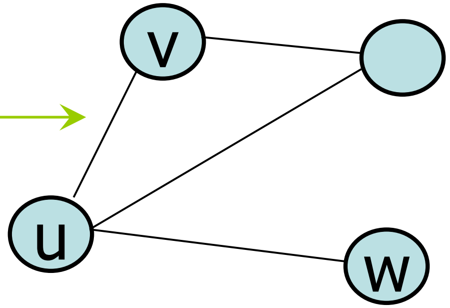


- The set of edges is denoted as E
- A graph G is defined as a pair (V, E)

Adjacent

- Two nodes u and v are said to be adjacent if (u, v) $\in$ E

(u, v)

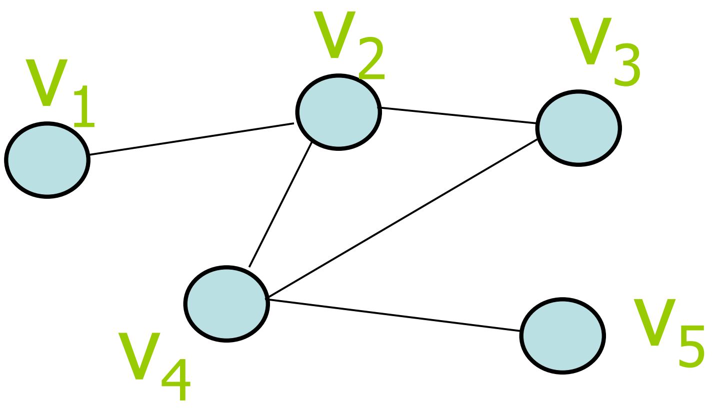


u and v are adjacent
v and w are not adjacent

Path and simple path

- A path from v_1 to v_k is a sequence of nodes $v_1, v_2, \ldots, v_k$ that are connected by edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$
- A path is called a simple path if every node appears at most once.

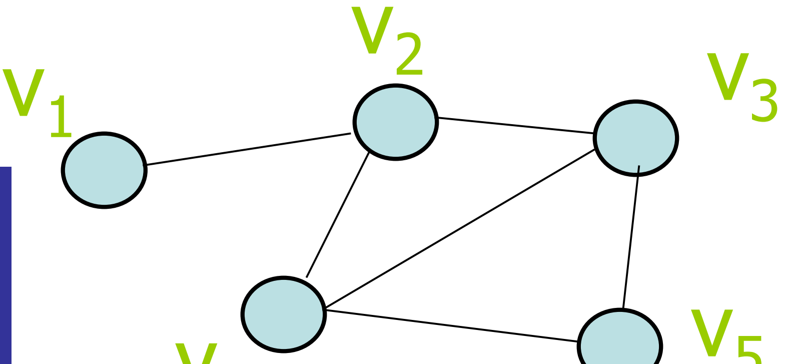
- v_2, v_3, v_4, v_2, v_1 is a path
- v_2, v_3, v_4, v_5 is a path, also it is a simple path



Cycle and simple cycle

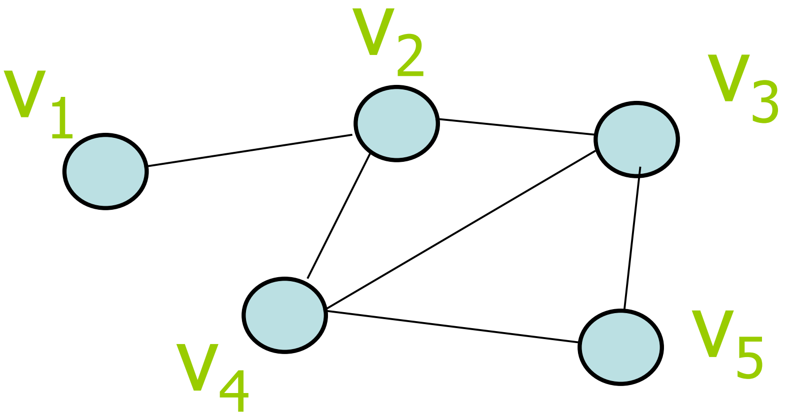
- A cycle is a path that begins and ends at the same node
- A simple cycle is a cycle if every node appears at most once, except for the first and the last nodes

- $v_2, v_3, v_4, v_5, v_3, v_2$ is a cycle
- v_2, v_3, v_4, v_2 is a cycle, it is also a simple cycle



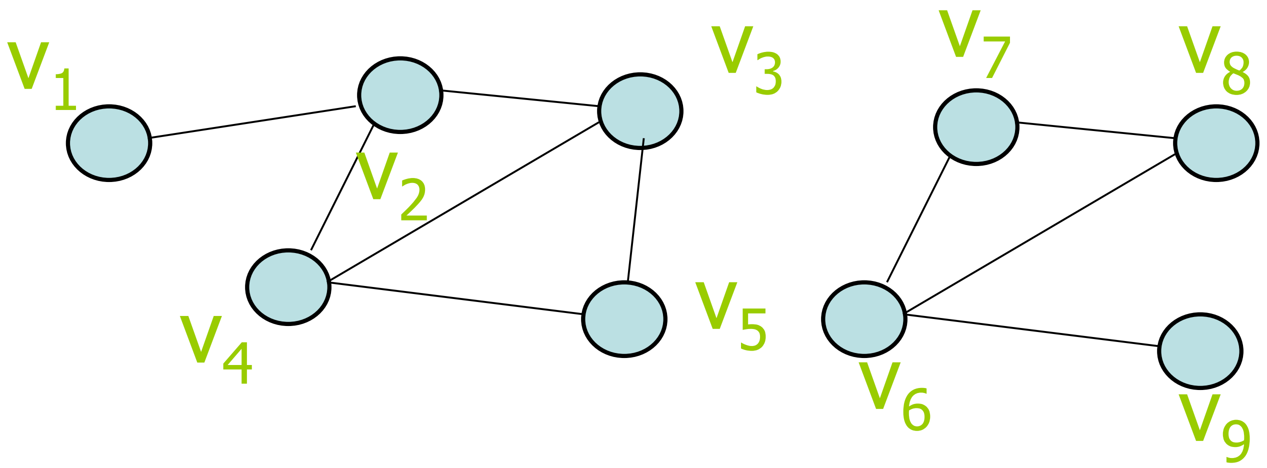
Connected graph

- A graph G is connected if there exists path between every pair of distinct nodes; otherwise, it is disconnected



This is a connected graph because there exists path between every pair of nodes

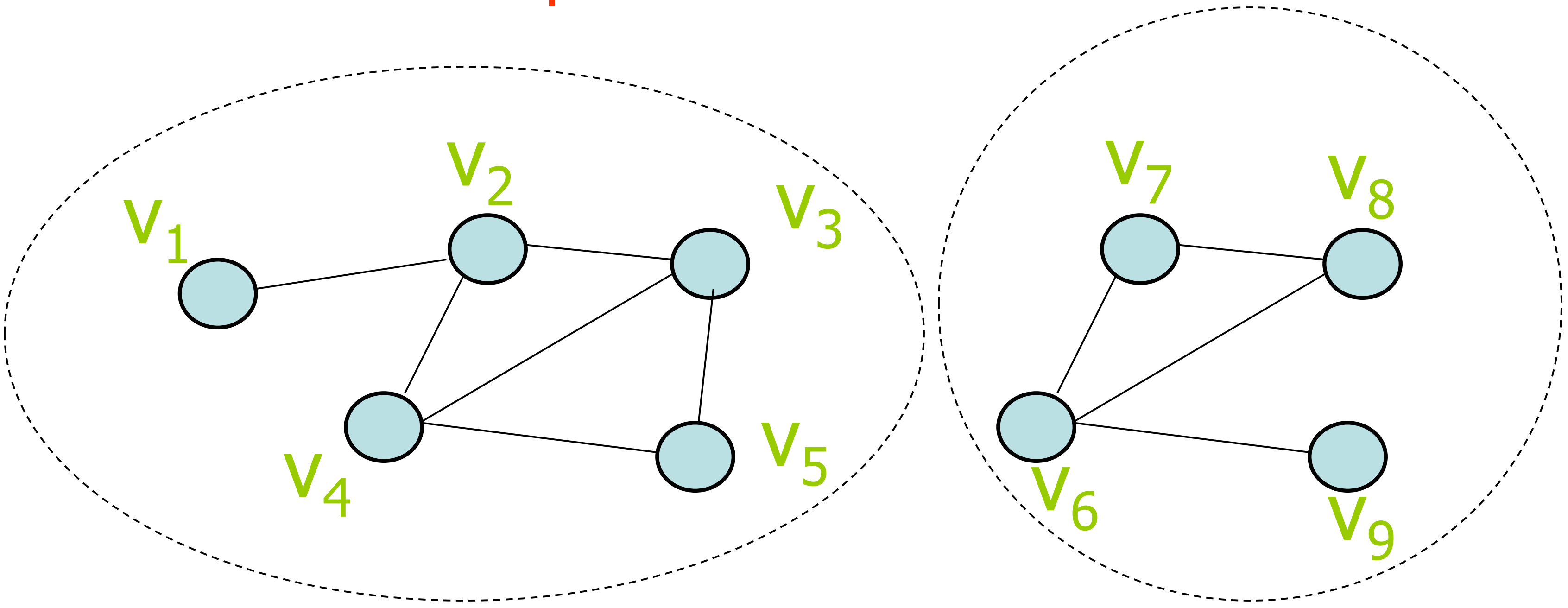
Example of disconnected graph



This is a disconnected graph because there does not exist path between some pair of nodes, says, v_1 and v_7

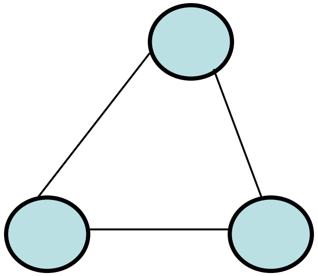
Connected component

- If a graph is disconnected, it can be partitioned into a number of graphs such that each of them is connected. Each such graph is called a connected component.

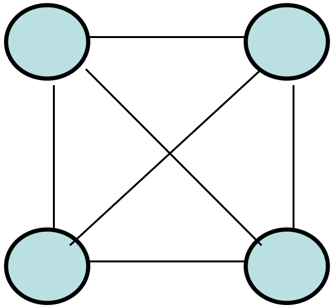


Complete graph

- A graph is complete if each pair of distinct nodes has an edge



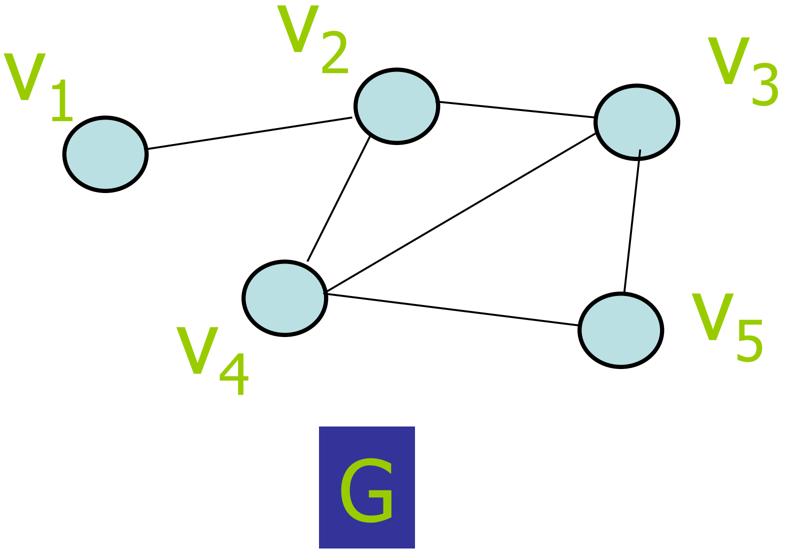
Complete graph with 3 nodes

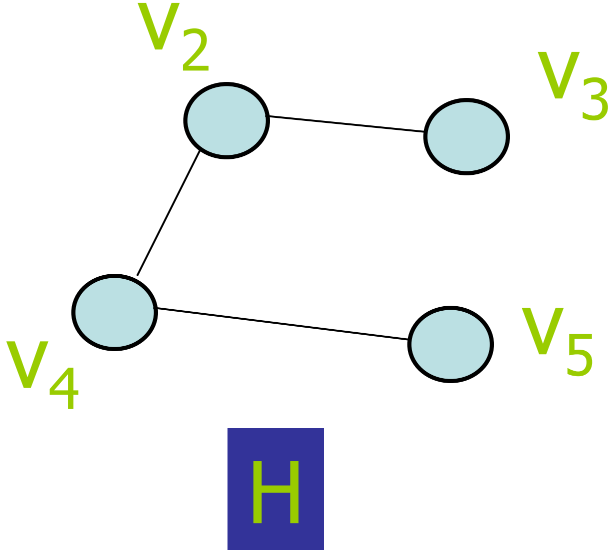


Complete graph with 4 nodes

Subgraph

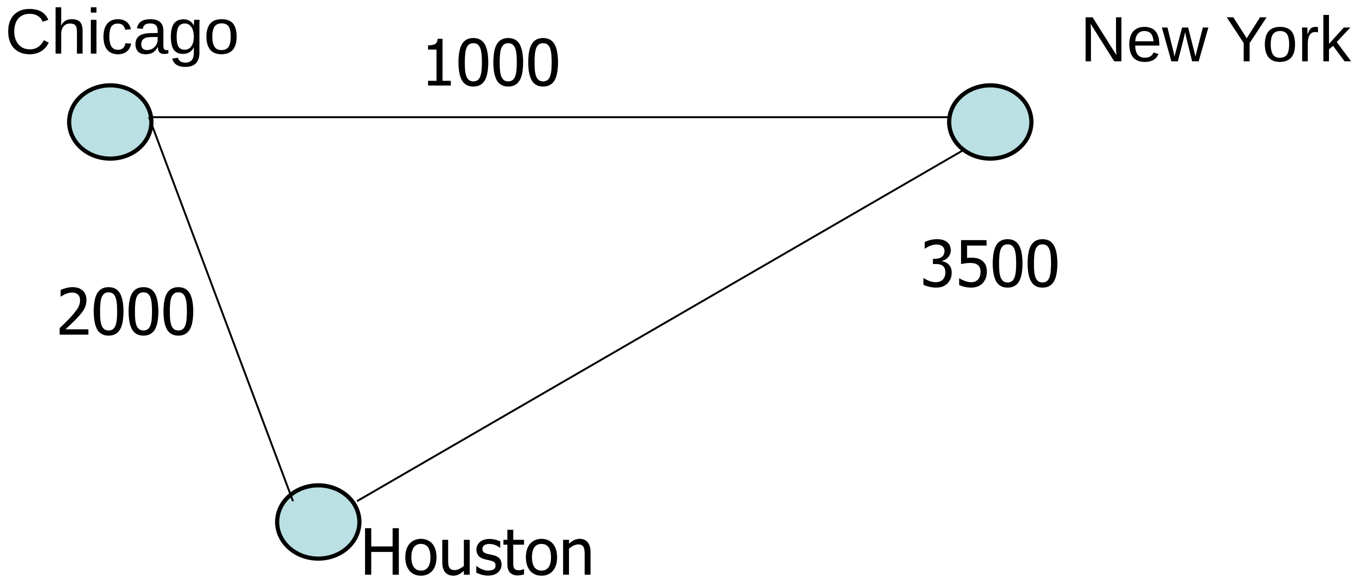
- A subgraph of a graph G =(V, E) is a graph H = (U, F) such that U $\subseteq$ V and F $\subseteq$ E.





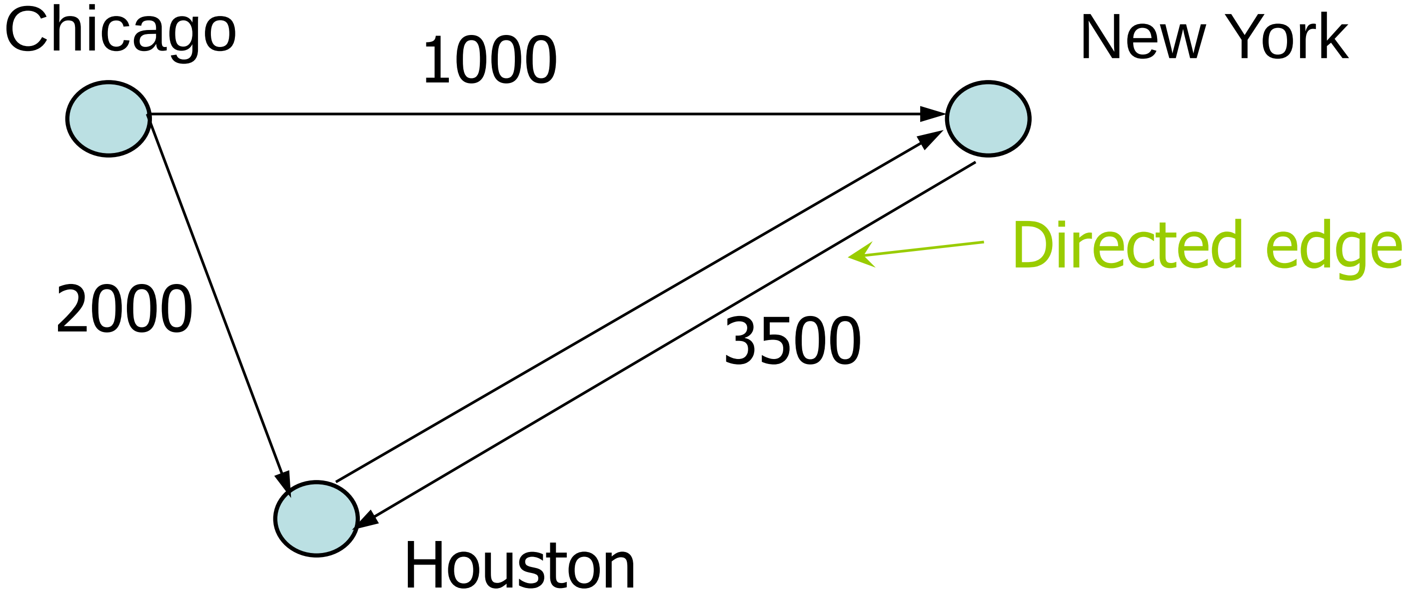
Weighted graph

- If each edge in G is assigned a weight, it is called a weighted graph.

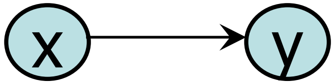


Directed graph (digraph)

- All previous graphs are undirected graph.
- If each edge in E has a direction, it is called a directed edge.
- A directed graph is a graph where every edges is a directed edge.



More on directed graph



- If (x, y) is a directed edge, we say
  - y is adjacent to x
  - y is successor of x
  - x is predecessor of y
- In a directed graph, directed path, directed cycle can be defined similarly

Property of graph

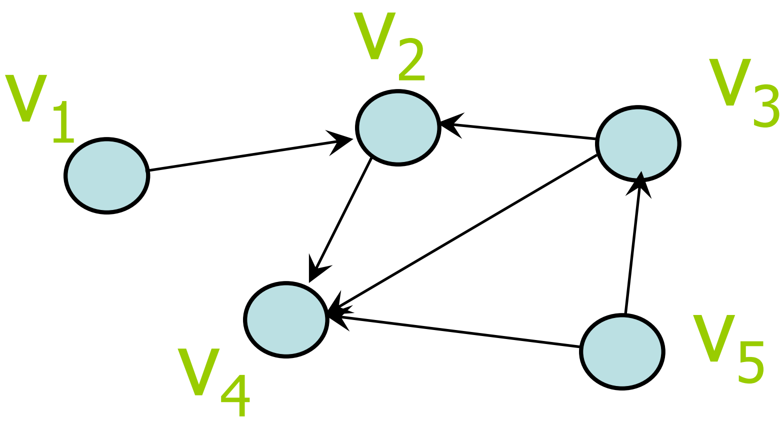
- An undirected graph that is connected and has no cycle is a tree.
- A tree with n nodes have exactly n-1 edges.
- A connected undirected graph with n nodes must have at least n-1 edges.

Implementing Graph

- Adjacency matrix
  - Represent a graph using a two-dimensional array
- Adjacency list
  - Represent a graph using n linked lists where n is the number of vertices

Adjacency matrix for directed graph

$$\text{Matrix}[i][j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{if } (v_i, v_j) \notin E \end{cases}$$

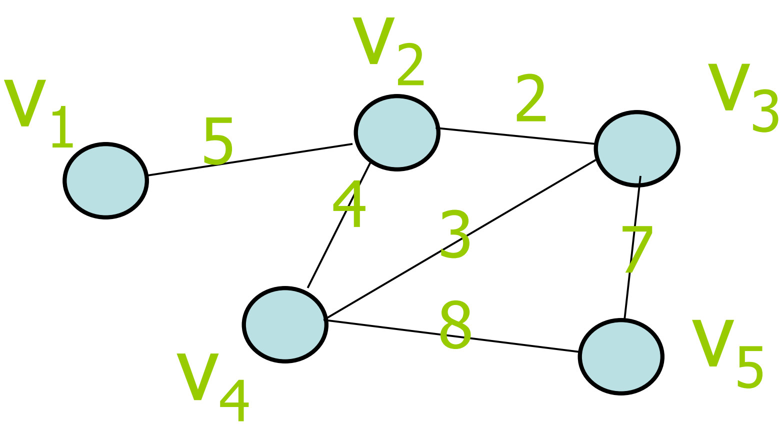


G

		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	0	1	0	0	0
2	v_2	0	0	0	1	0
3	v_3	0	1	0	1	0
4	v_4	0	0	0	0	0
5	v_5	0	0	1	1	0

Adjacency matrix for weighted undirected graph

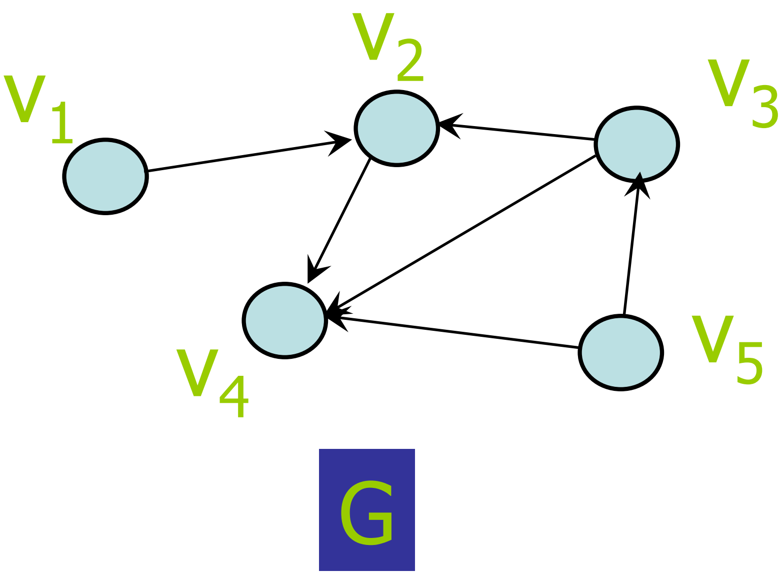
Matrix[i][j] = $w(v_i, v_j)$ if $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$
∞ otherwise



G

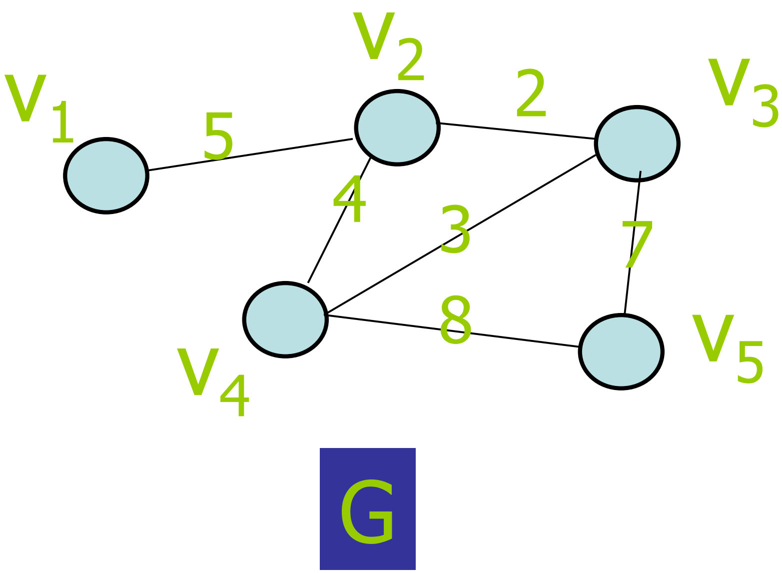
		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	∞	5	∞	∞	∞
2	v_2	5	∞	2	4	∞
3	v_3	0	2	∞	3	7
4	v_4	∞	4	3	∞	8
5	v_5	∞	∞	7	8	∞

Adjacency list for directed graph



1	v_1	$\rightarrow v_2$
2	v_2	$\rightarrow v_4$
3	v_3	$\rightarrow v_2 \rightarrow v_4$
4	v_4	
5	v_5	$\rightarrow v_3 \rightarrow v_4$

Adjacency list for weighted undirected graph



1 v_1 → $v_2(5)$

2 v_2 → $v_1(5)$ → $v_3(2)$ → $v_4(4)$

3 v_3 → $v_2(2)$ → $v_4(3)$ → $v_5(7)$

4 v_4 → $v_2(4)$ → $v_3(3)$ → $v_5(8)$

5 v_5 → $v_3(7)$ → $v_4(8)$

Pros and Cons

- Adjacency matrix
  - Allows us to determine whether there is an edge from node i to node j in O(1) time
- Adjacency list
  - Allows us to find all nodes adjacent to a given node j efficiently
  - If the graph is sparse, adjacency list requires less space

Directed Graph using **Adjacency Matrix**

```
#include <stdio.h>
#define MAX 100   // maximum number of vertices allowed

int main() {
    int n, e;                  // n = number of vertices, e = number of edges
    int adj[MAX][MAX];         // adjacency matrix
    int i, j, src, dest;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    // Step 1: Initialize adjacency matrix with 0 (no edges yet)
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            adj[i][j] = 0;
        }
    }

    printf("Enter number of edges: ");
    scanf("%d", &e);
```

Directed Graph using **Adjacency Matrix**

```
// Step 2: Input edges
    printf("Enter edges (src dest):\n");
    for (i = 0; i < e; i++) {
        scanf("%d%d", &src, &dest);
        adj[src][dest] = 1;   // directed edge from src to dest
    }

    // Step 3: Print adjacency matrix
    printf("\nAdjacency Matrix:\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            printf("%d ", adj[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Directed Graph using **Adjacency List**

```
#include <stdio.h>
#include <stdlib.h>

// Node structure for adjacency list
struct Node {
    int vertex;             // stores the destination vertex
    struct Node* next;      // pointer to the next node
};

int main() {
    int n, e, i, src, dest;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    // ----------------- Adjacency Matrix -----------------
    int adjMat[n][n];   // adjacency matrix representation

    // Step 1: Initialize matrix with 0 (no edges yet)
    for (i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            adjMat[i][j] = 0;
```

Directed Graph using **Adjacency List**

```
// ----------------- Adjacency List -----------------
    struct Node* adjList[n];   // array of pointers (one per vertex)
    for (i = 0; i < n; i++)
        adjList[i] = NULL;     // initially no edges

    printf("Enter number of edges: ");
    scanf("%d", &e);

    // Step 2: Input edges
    printf("Enter edges (src dest):\n");
    for (i = 0; i < e; i++) {
        scanf("%d%d", &src, &dest);
        // Matrix representation update
        adjMat[src][dest] = 1;
        // List representation update
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->vertex = dest;
        newNode->next = adjList[src];   // insert at beginning
        adjList[src] = newNode;
    }
```

Directed Graph using **Adjacency List**

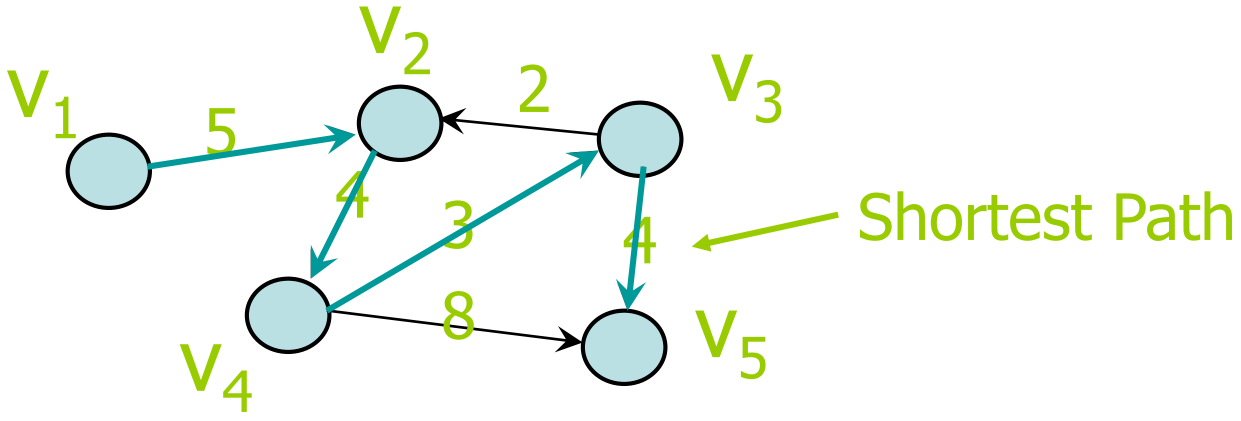
```
// Step 3: Print adjacency matrix
   printf("\nAdjacency Matrix:\n");
   for (i = 0; i < n; i++) {
       for (int j = 0; j < n; j++)
           printf("%d ", adjMat[i][j]);
       printf("\n");
   }

   // Step 4: Print adjacency list
   printf("\nAdjacency List:\n");
   for (i = 0; i < n; i++) {
       printf("Vertex %d: ", i);
       struct Node* temp = adjList[i];
       while (temp) {
           printf("%d -> ", temp->vertex);
           temp = temp->next;
       }
       printf("NULL\n");
   }

   return 0;
}
```

Shortest path

- Consider a weighted directed graph
  - Each node x represents a city x
  - Each edge (x, y) has a number which represent the cost of traveling from city x to city y
- Problem: find the minimum cost to travel from city x to city y
- Solution: find the shortest path from x to y

Formal definition of shortest path

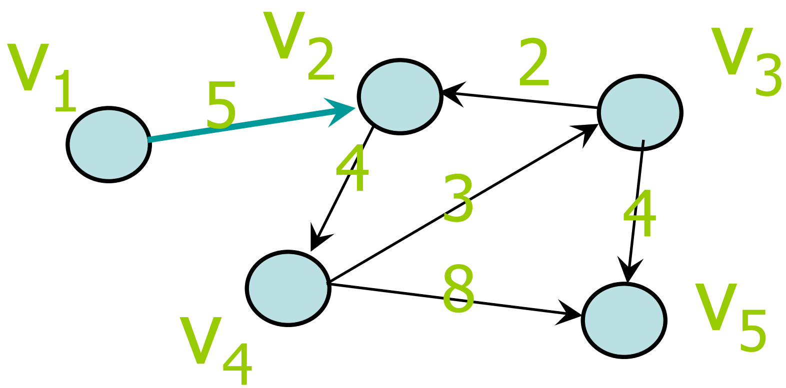
- Given a weighted directed graph G.
- Let P be a path of G from x to y.
- $\text{cost}(P) = \sum_{e \in P} \text{weight}(e)$
- The shortest path is a path P which minimizes cost(P)



Dijkstra's algorithm

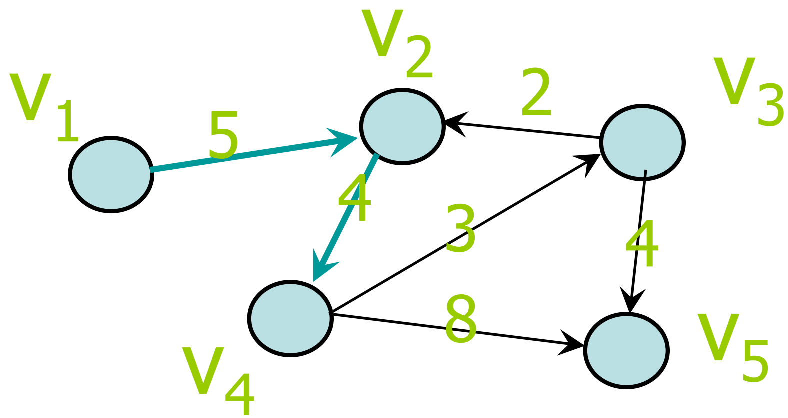
- Consider a graph G, each edge (u, v) has a weight w(u, v) > 0.
- Suppose we want to find the shortest path starting from v_1 to any node v_i
- Let VS be a subset of nodes in G
- Let cost[v_i] be the weight of the shortest path from v_1 to v_i that passes through nodes in VS only.

Example for Dijkstra's algorithm



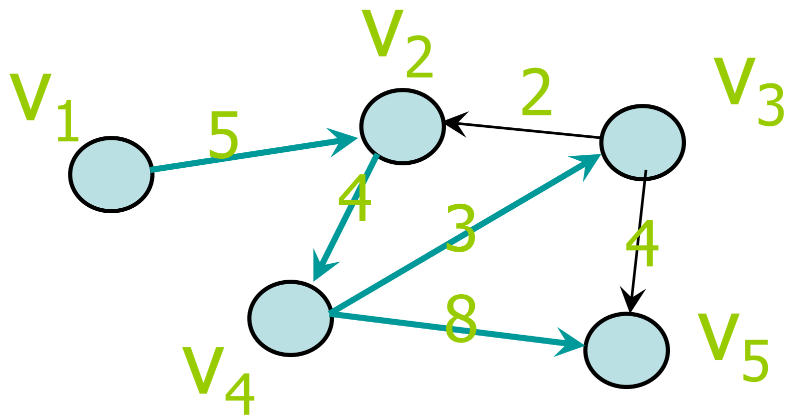
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞

Example for Dijkstra's algorithm



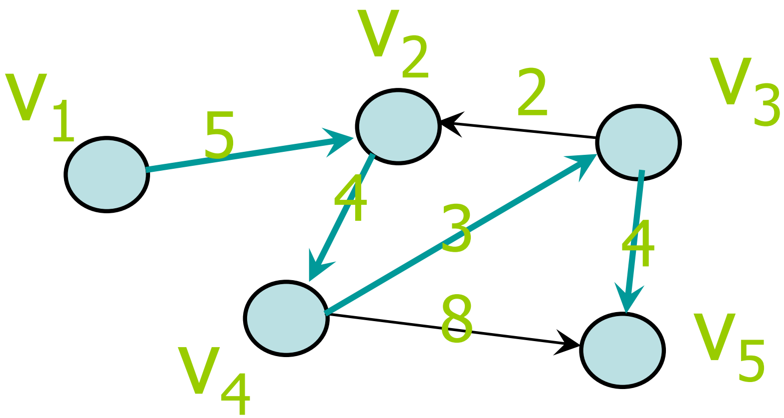
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞

Example for Dijkstra's algorithm



	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17

Example for Dijkstra's algorithm



	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17
4	v_3	[v_1, v_2, v_4, v_3]	0	5	12	9	16
5	v_5	[v_1, v_2, v_4, $v_{3,}$ v_5]	0	5	12	9	16

Dijkstra's algorithm

Algorithm shortestPath()

n = number of nodes in the graph;

for i = 1 to n

  cost[v_i] = w(v_1, v_i);

VS = { v_1 };

for step = 2 to n {

  find the smallest cost[v_i] s.t. v_i is not in VS;

  include v_i to VS;

  for (all nodes v_j not in VS) {

    if (cost[v_j] > cost[v_i] + w(v_i, v_j))

      cost[v_j] = cost[v_i] + w(v_i, v_j);

  }

}