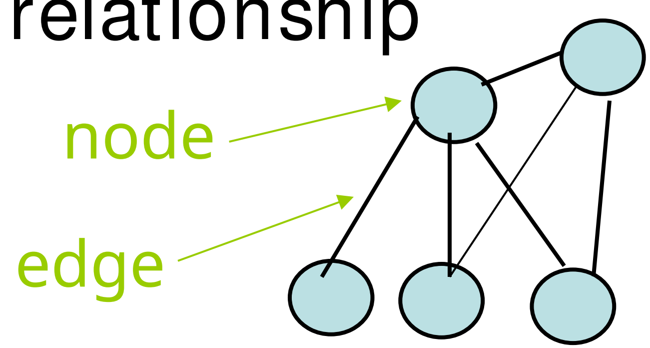
Graphs

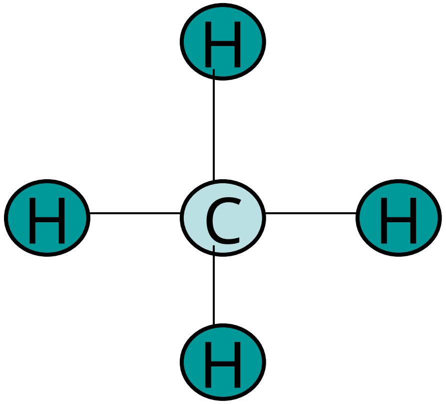
What is a graph?

- A graph is a finite set of nodes with edges between nodes
- Formally, a graph G is a structure (V,E) consisting 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

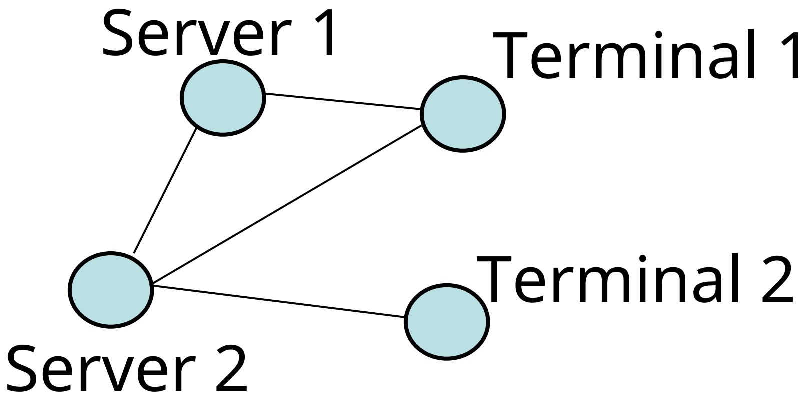


Examples of graphs

Molecular Structure



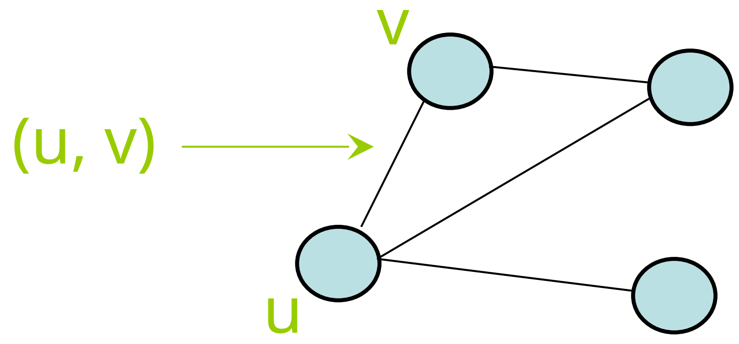
Computer Network



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)

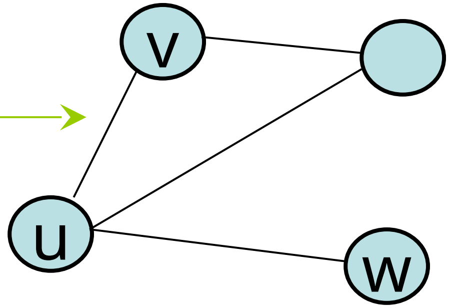


- The set of edges is denoted as E
- A graph G is defined as a pair (V, E)

Adjacent

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

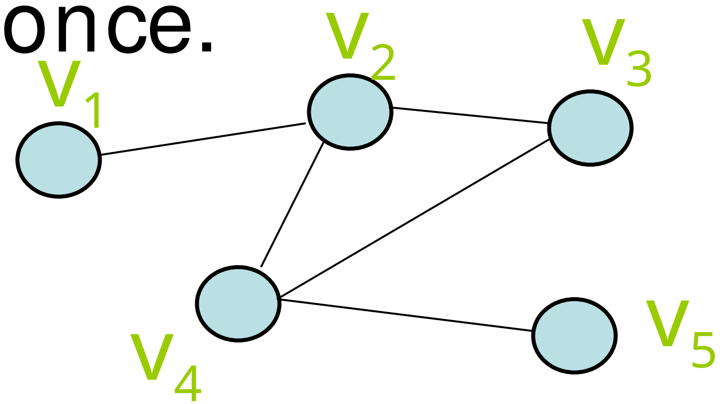
(u, v)



u and v are adjacent
v and w are not adjacent

Path and simple path

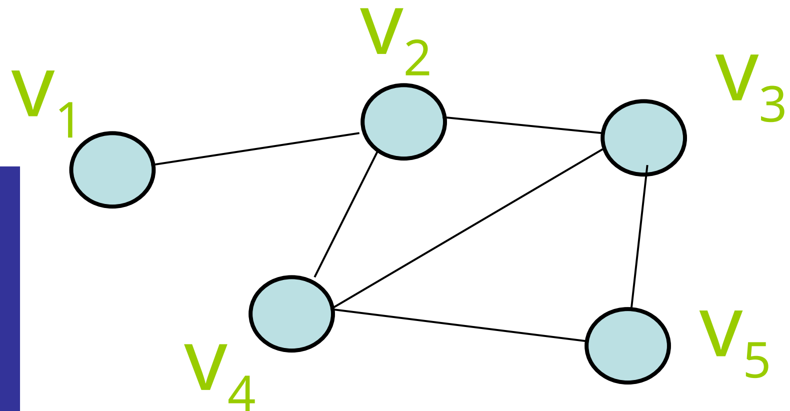
- A path from v_1 to v_k is a sequence of nodes $v_1, v_2, \ldots, v_k$ that are connected by edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$
- A path is called a simple path if every node appears at most once.



- 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

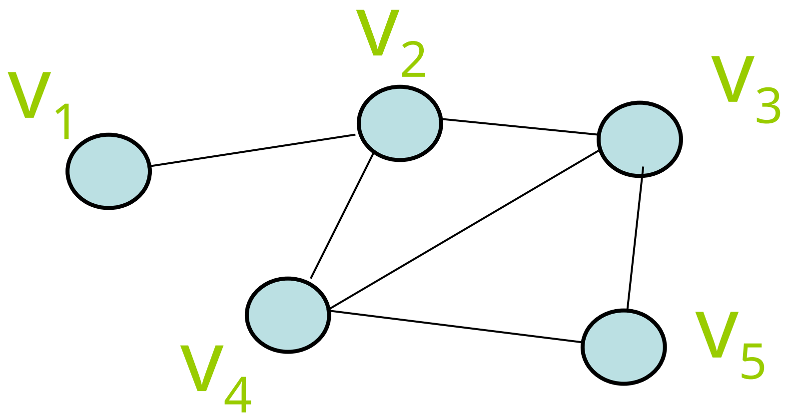
- A cycle is a path that begins and ends at the same node
- A simple cycle is a cycle if every node appears at most once, except for the first and the last nodes



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

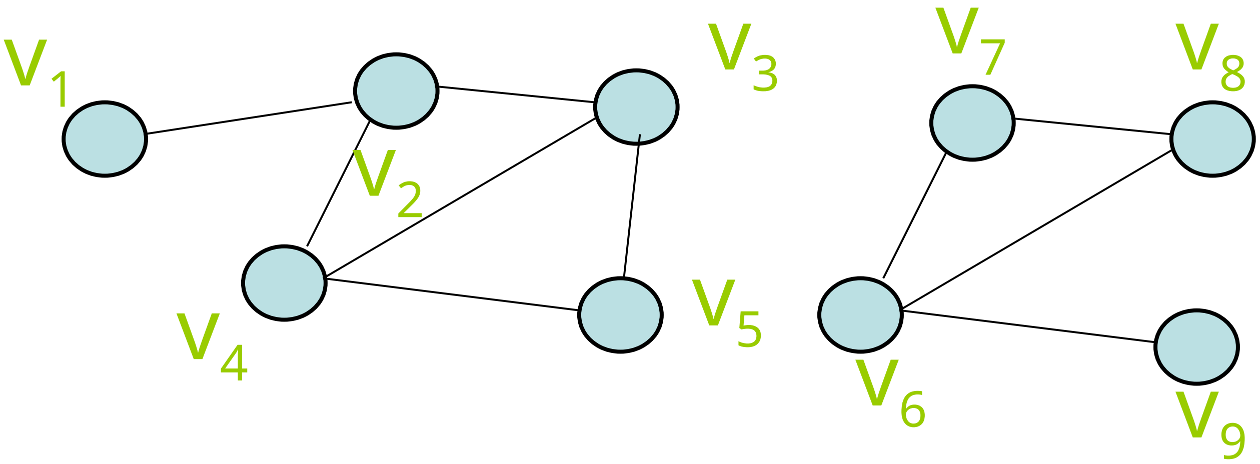
Connected graph

- 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

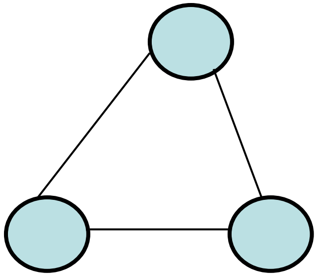
Example of disconnected graph



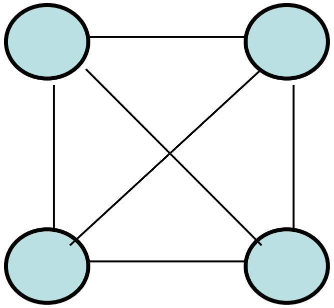
This is a disconnected graph because there does not exist path between some pair of nodes, says, v_1 and v_7

Complete graph

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



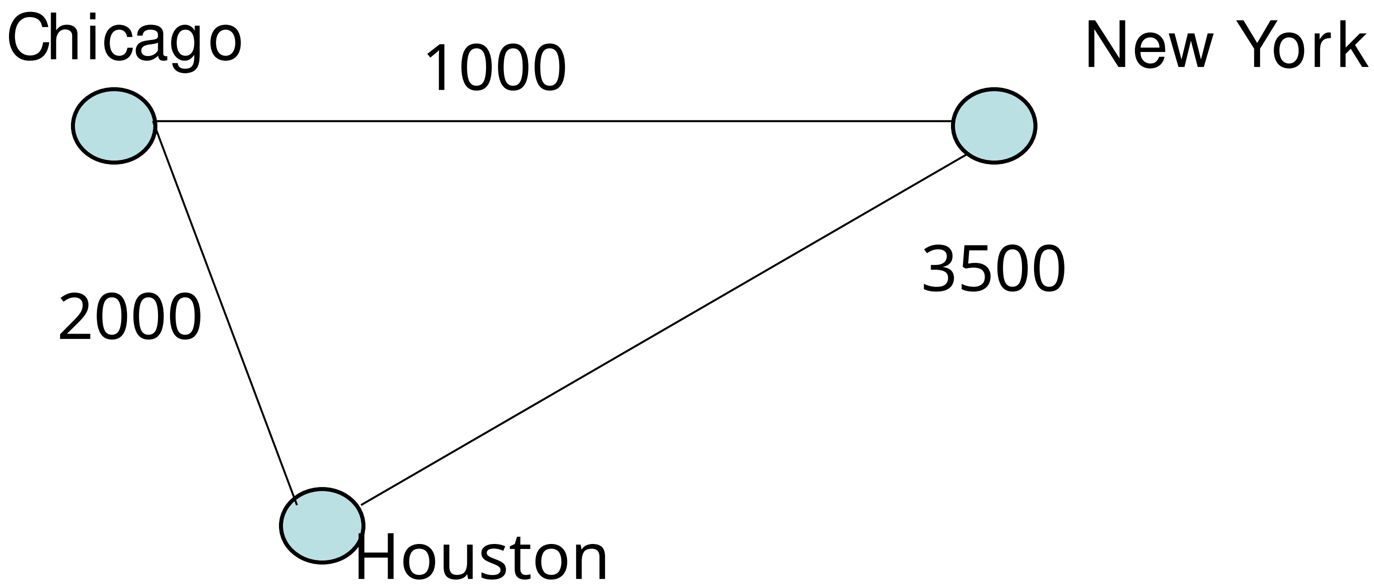
Complete graph with 3 nodes



Complete graph with 4 nodes

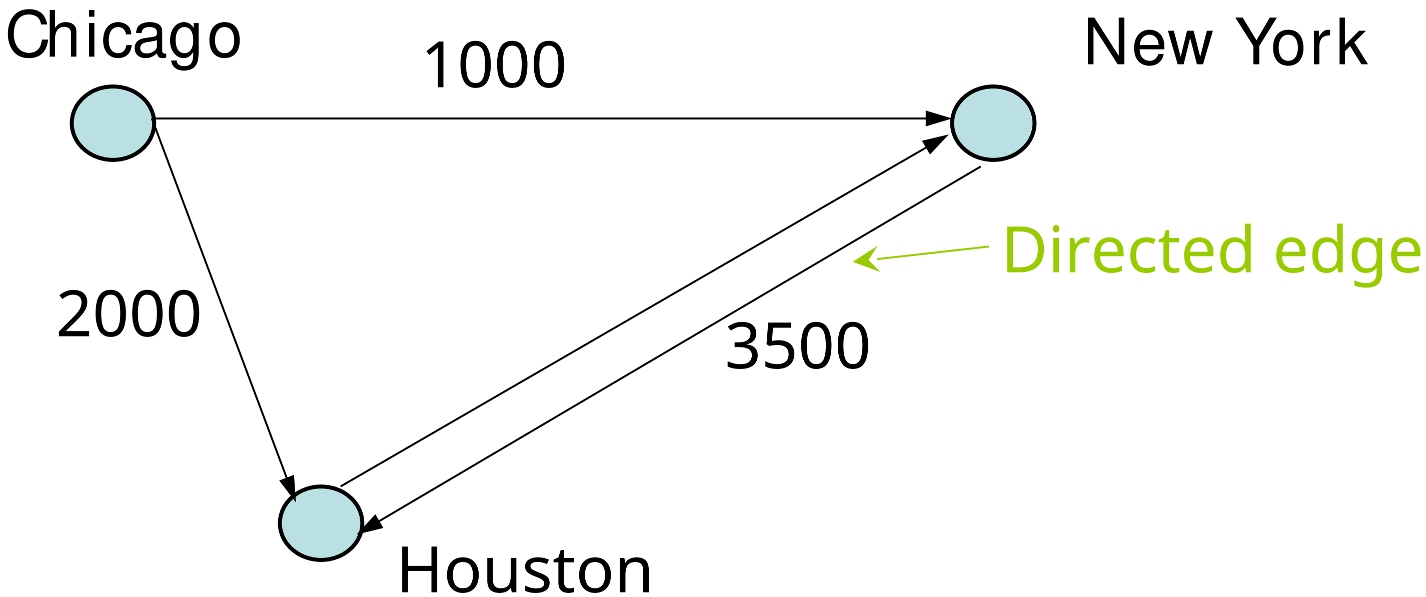
Weighted graph

- 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



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

Graph Representation

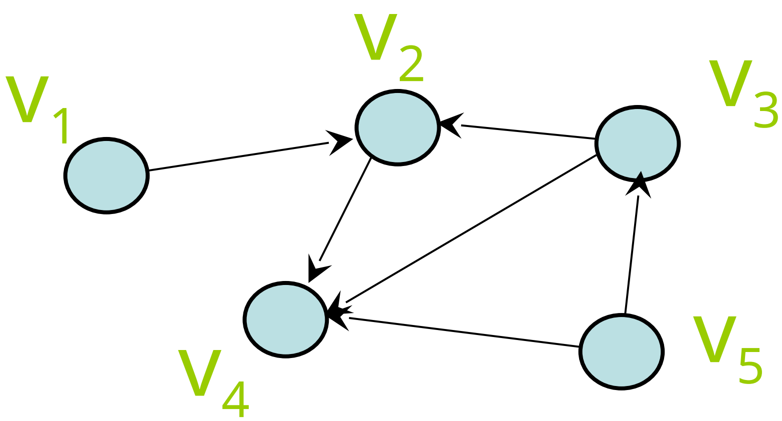
- For graphs to be computationally useful, they have to be conveniently represented in programs
- There are two computer representations of graphs:
  – Adjacency matrix representation
  – Adjacency lists representation

Adjacency Matrix Representation

- In this representation, each graph of n nodes is represented by an n x n matrix A, that is, a two-dimensional array A
- The nodes are (re)-labeled 1,2,…,n
- A[i][j] = 1 if (i,j) is an edge
- A[i][j] = 0 if (i,j) is not an edge

Adjacency matrix for directed graph

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

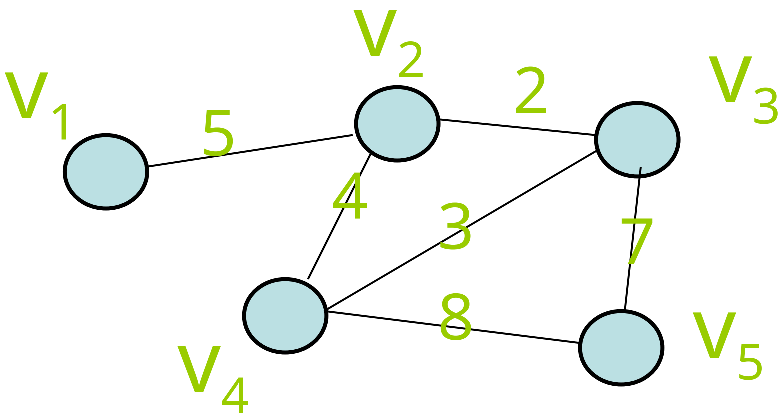


G

		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

$$\text{Matrix}[i][j] = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E \text{ or } (v_j, v_i) \in E \\ \infty & \text{otherwise} \end{cases}$$



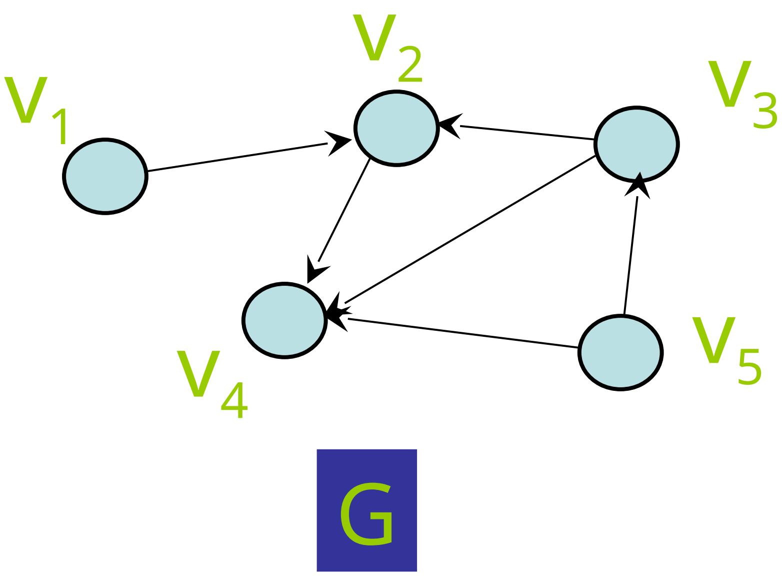
G

		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 Lists Representation

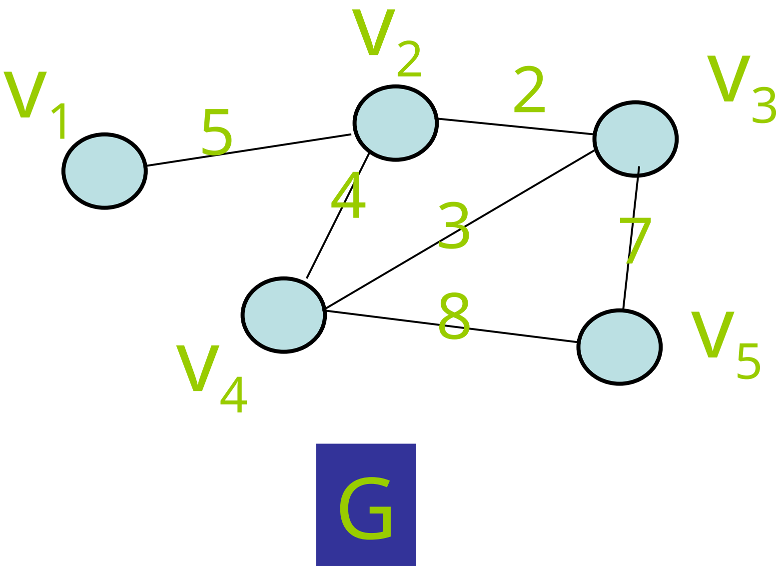
- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order

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	$\rightarrow$	$v_2(5)$					
2	v_2	$\rightarrow$	$v_1(5)$	$\rightarrow$	$v_3(2)$	$\rightarrow$	$v_4(4)$
3	v_3	$\rightarrow$	$v_2(2)$	$\rightarrow$	$v_4(3)$	$\rightarrow$	$v_5(7)$
4	v_4	$\rightarrow$	$v_2(4)$	$\rightarrow$	$v_3(3)$	$\rightarrow$	$v_5(8)$
5	v_5	$\rightarrow$	$v_3(7)$	$\rightarrow$	$v_4(8)$		

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

Problems related to Graph

- Graph Traversal
- Topological Sort
- Spanning Tree
- Minimum Spanning Tree
- Shortest Path

Graph Traversal Techniques

- The previous connectivity problem, as well as many other graph problems, can be solved using graph traversal techniques
- There are two standard graph traversal techniques:
  - *Depth-First Search* (DFS)
  - *Breadth-First Search* (BFS)

Two basic traversal algorithms

- Two basic graph traversal algorithms:
  - Depth-first-search (DFS)
    - After visit node v, DFS strategy proceeds along a path from v as deeply into the graph as possible before backing up
  - Breadth-first-search (BFS)
    - After visit node v, BFS strategy visits every node adjacent to v before visiting any other nodes

Depth-First Search

- DFS follows the following rules:
  1. Select an unvisited node x, visit it, and treat as the current node
  2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
  3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
  4. Repeat steps 3 and 4 until no more nodes can be visited.
  5. If there are still unvisited nodes, repeat from step 1.

Depth-first search (DFS)

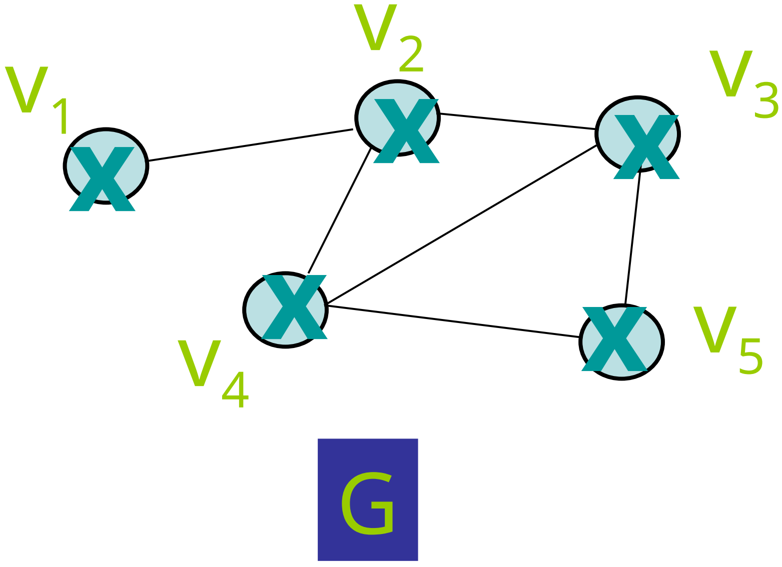
- DFS strategy looks similar to pre-order. From a given node v, it first visits itself. Then, recursively visit its unvisited neighbours one by one.
- DFS can be defined recursively as follows.

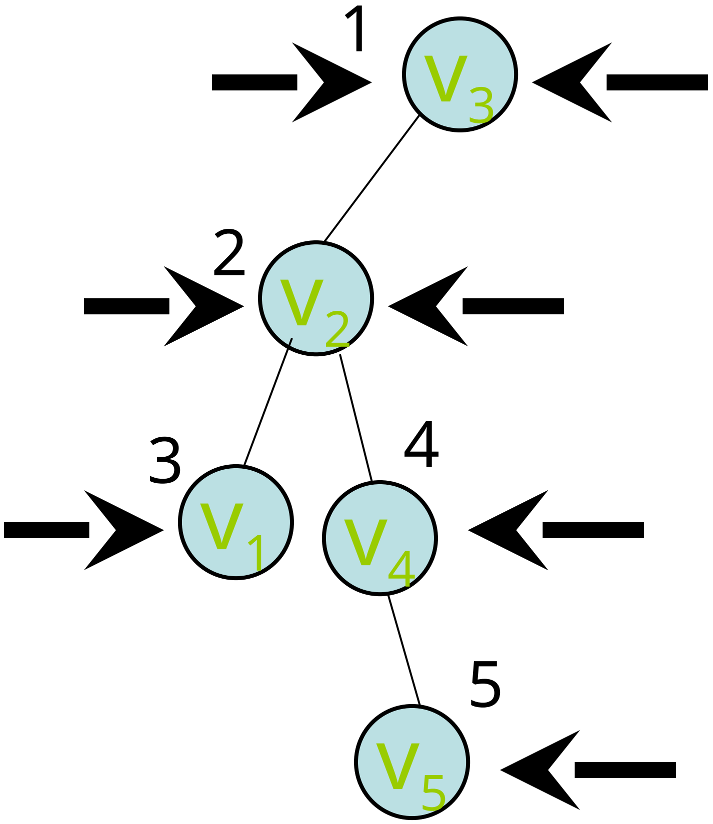
Algorithm dfs(v)

```
print v; // you can do other things!
mark v as visited;
for (each unvisited node u adjacent to v)
    dfs(u);
```

DFS example

- Start from v_3



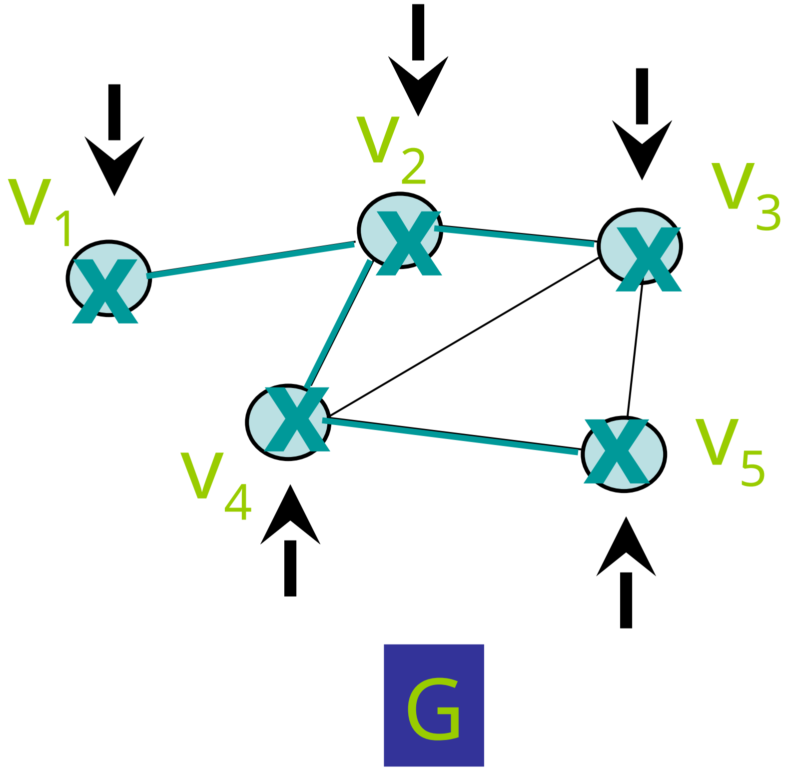


Non-recursive version of DFS algorithm

```
Algorithm dfs(v)
s.createStack();
s.push(v);
mark v as visited;
while (!s.isEmpty()) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x)
            s.pop(); // blacktrack
    else {
            select an unvisited node u adjacent to x;
            s.push(u);
            mark u as visited;
    }
}
```

Non-recursive DFS example

visit	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4, v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty



Breadth-First Search

- BFS follows the following rules:
  1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
  2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. 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.

Breadth-first search (BFS)

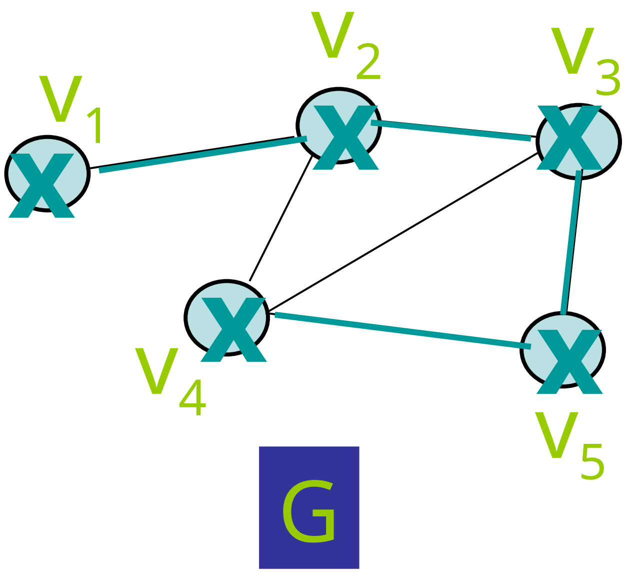
- BFS strategy looks similar to level-order. From a given node v, it first visits itself. Then, it visits every node adjacent to v before visiting any other nodes.
  - 1. Visit v
  - 2. Visit all v's neigbours
  - 3. Visit all v's neighbours' neighbours
  - …
- Similar to level-order, BFS is based on a queue.

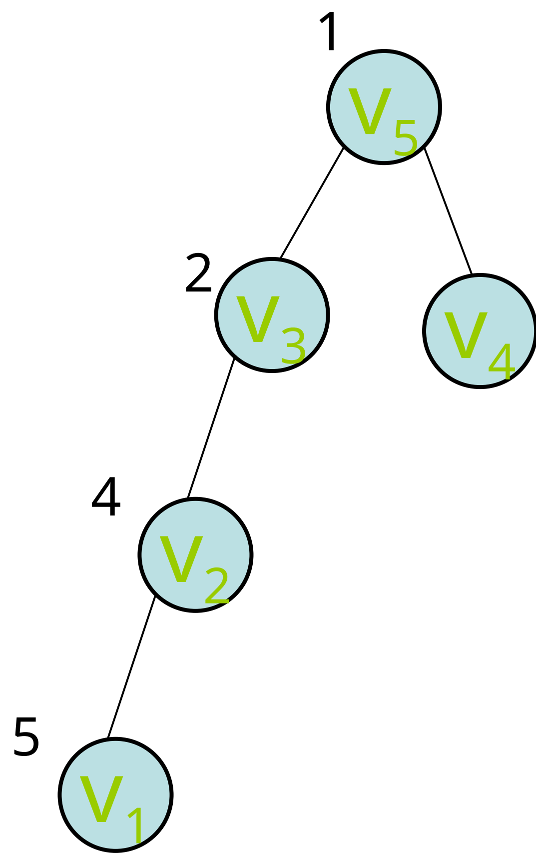
Algorithm for BFS

```
Algorithm bfs(v)
q.createQueue();
q.enqueue(v);
mark v as visited;
while(!q.isEmpty()) {
   w = q.dequeue();
   for (each unvisited node u adjacent to w) {
          q.enqueue(u);
          mark u as visited;
   }
}
```

BFS example

- Start from v_5

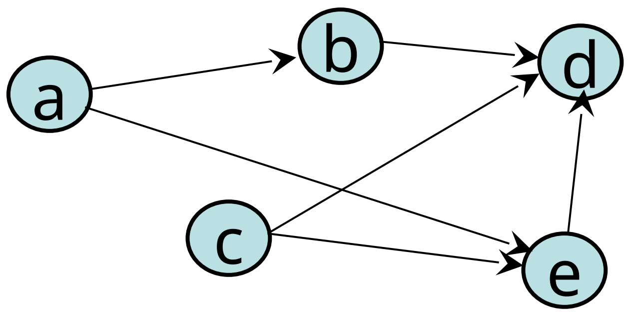




Visit	Queue (front to back)
v_5	v_5
	empty
v_3	v_3
v_4	v_3, v_4
	v_4
v_2	v_4, v_2
	v_2
	empty
v_1	v_1
	empty

Topological order

- Consider the prerequisite structure for courses:



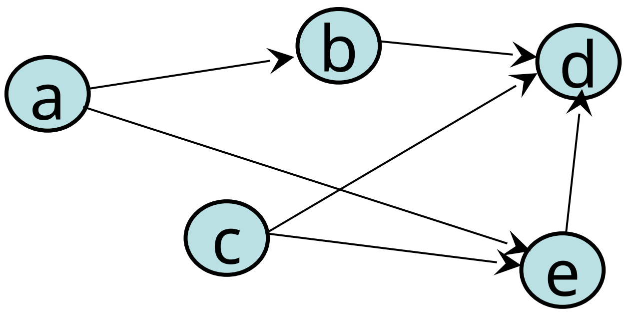
- Each node x represents a course x
- (x, y) represents that course x is a prerequisite to course y
- Note that this graph should be a directed graph without cycles (called a directed acyclic graph).
- A linear order to take all 5 courses while satisfying all prerequisites is called a topological order.
- E.g.
  - a, c, b, e, d
  - c, a, b, e, d

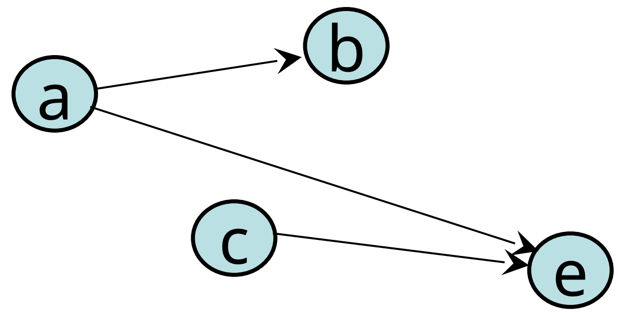
Topological sort

- Arranging all nodes in the graph in a topological order

```
Algorithm topSort
n = |V| ;
for i = 1 to n {
   select a node v that has no successor;
   aList.add(1, v);
   delete node v and its edges from the graph;
}
return aList;
```

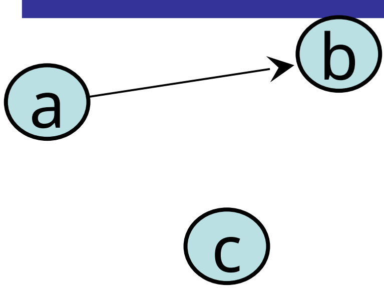
Example

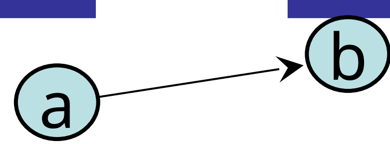


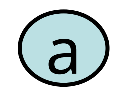


1. d has no successor! Choose d!

2. Both b and e have no successor! Choose e!







3. Both b and c have no successor! Choose c!

4. Only b has no successor! Choose b!

5. Choose a!
The topological order is **a,b,c,e,d**

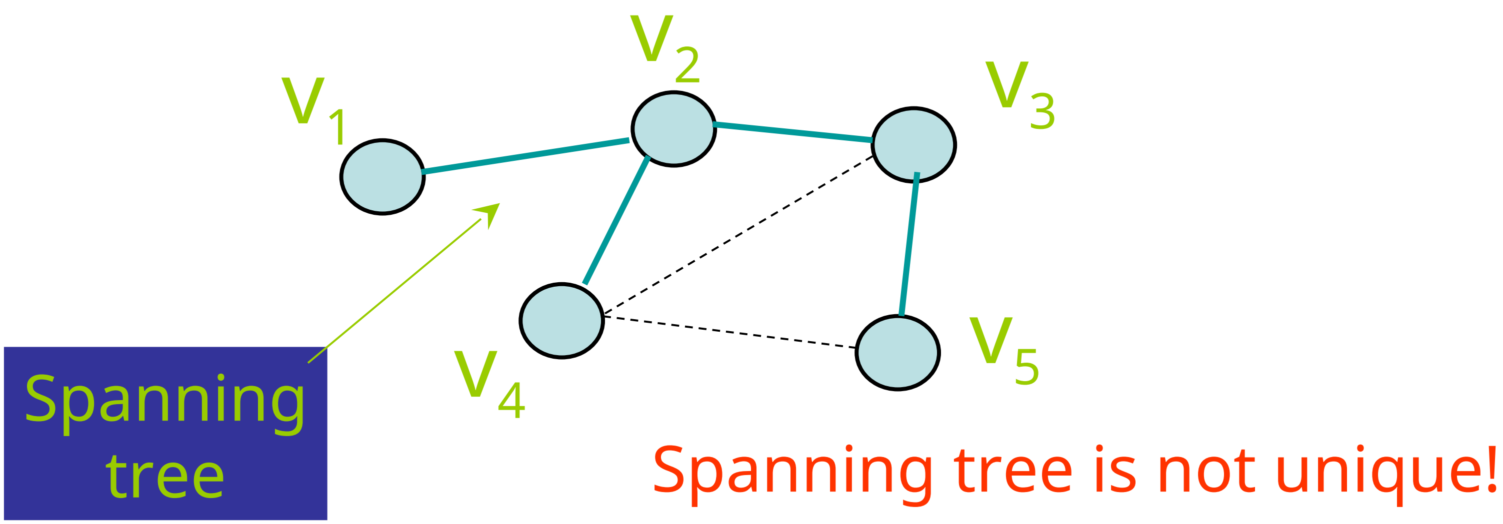
Topological sort algorithm 2

- This algorithm is based on DFS

```
Algorithm topSort2
s.createStack();
for (all nodes v in the graph) {
    if (v has no predecessors) {
            s.push(v);
            mark v as visited;
    }
}
while (!s.isEmpty()) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x) { // i.e. x has no unvisited successor
            aList.add(1, x);
            s.pop(); // blacktrack
    } else {
            select an unvisited node u adjacent to x;
            s.push(u);
            mark u as visited;
    }
}
return aList;
```

Spanning Tree

- Given a connected undirected graph G, a spanning tree of G is a subgraph of G that contains all of G's nodes and enough of its edges to form a tree.



DFS spanning tree

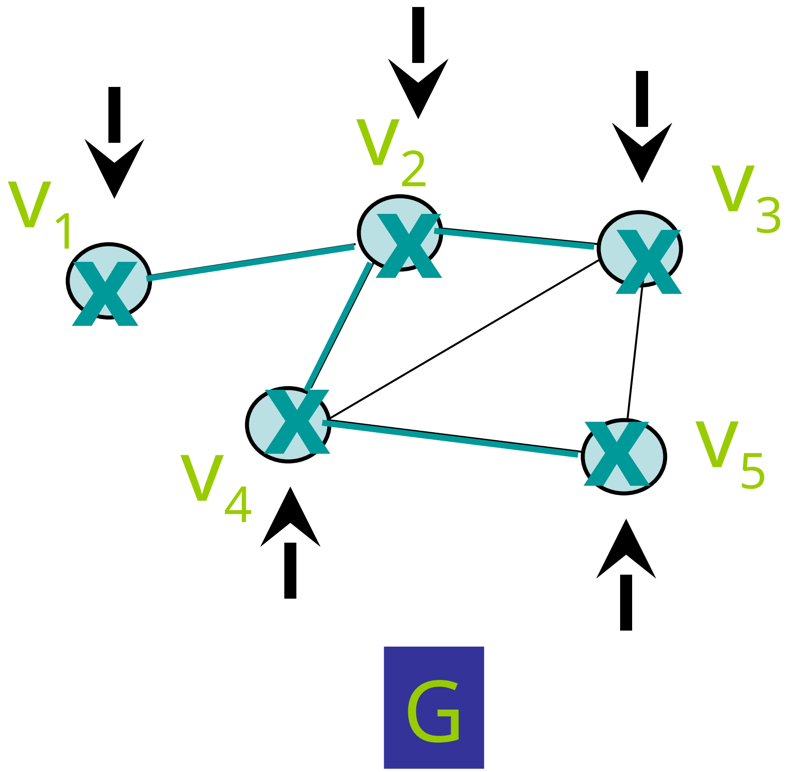
- Generate the spanning tree edge during the DFS traversal.

```
Algorithm dfsSpanningTree(v)
mark v as visited;
for (each unvisited node u adjacent to v) {
  mark the edge from u to v;
  dfsSpanningTree(u);
}
```

- Similar to DFS, the spanning tree edges can be generated based on BFS traversal.

Example of generating spanning tree based on DFS

	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4, v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty

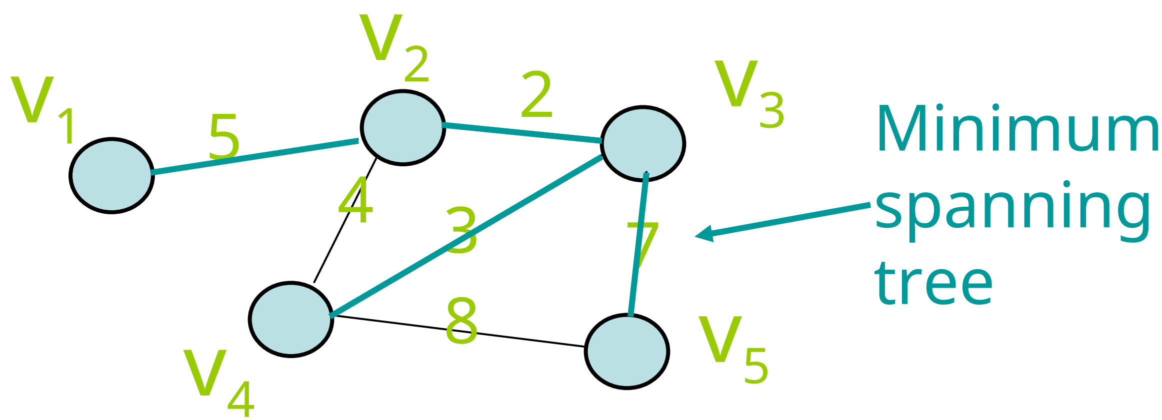


Minimum Spanning Tree

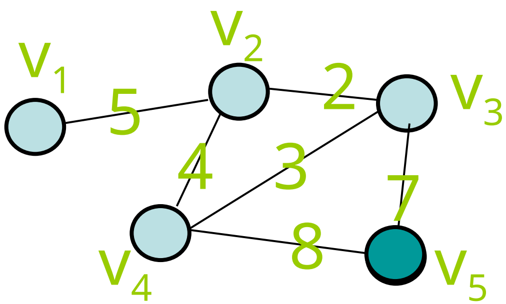
- Consider a connected undirected graph where
  - Each node x represents a country x
  - Each edge (x, y) has a number which measures the cost of placing telephone line between country x and country y
- Problem: connecting all countries while minimizing the total cost
- Solution: find a spanning tree with minimum total weight, that is, minimum spanning tree

Formal definition of minimum spanning tree

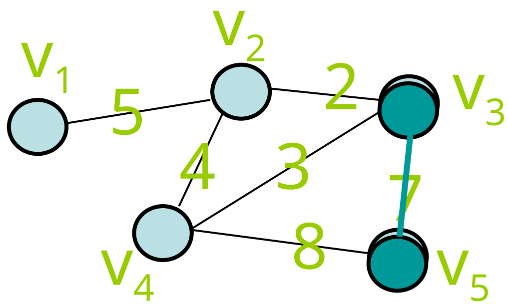
- Given a connected undirected graph G.
- Let T be a spanning tree of G.
- $\text{cost}(T) = \Sigma_{e \in T} \text{weight}(e)$
- The minimum spanning tree is a spanning tree T which minimizes cost(T)



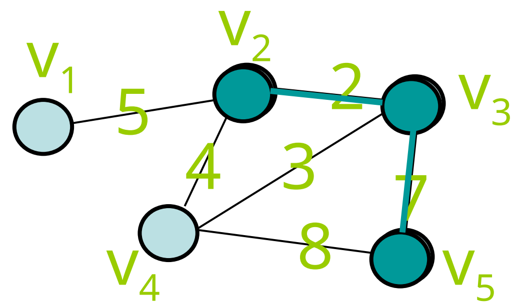
Prim's algorithm (I)



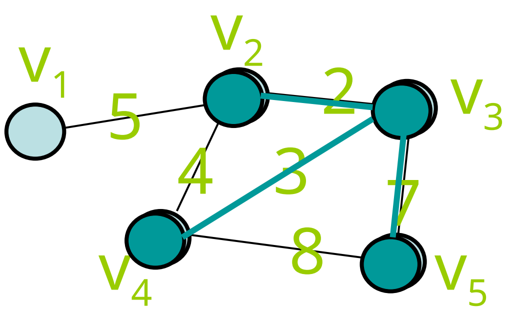
Start from v_5, find the minimum edge attach to v_5

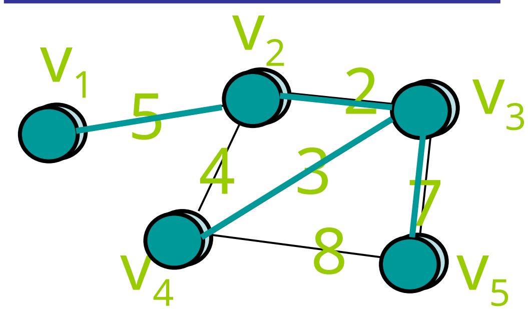


Find the minimum edge attach to v_3 and v_5



Find the minimum edge attach to v_2, v_3 and v_5





Find the minimum edge attach to v_2, v_3 , v_4 and v_5

Prim's algorithm (II)

Algorithm PrimAlgorithm(v)

- Mark node v as visited and include it in the minimum spanning tree;
- while (there are unvisited nodes) {
  - find the minimum edge (v, u) between a visited node v and an unvisited node u;
  - mark u as visited;
  - add both v and (v, u) to the minimum spanning tree;

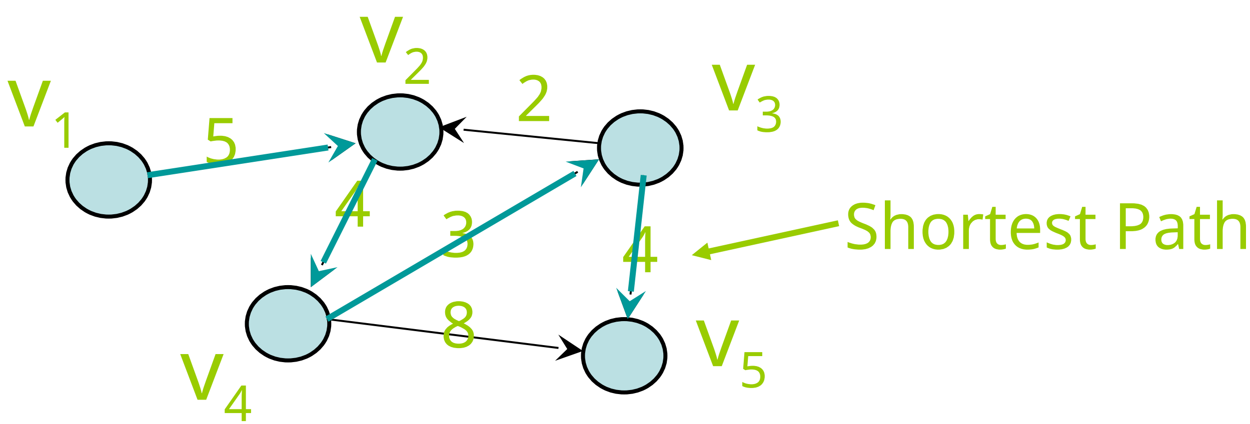
  }

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

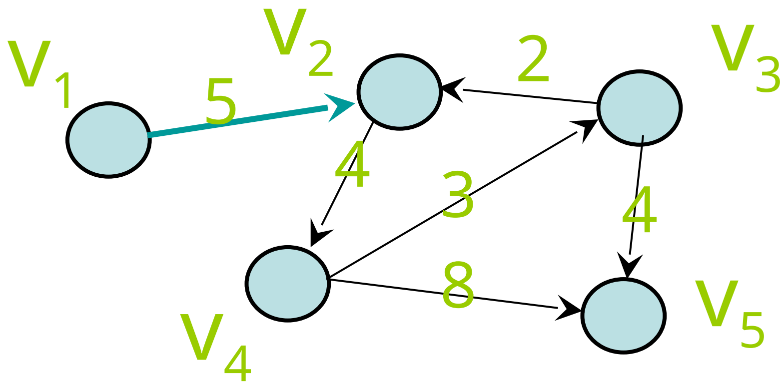
- Given a weighted directed graph G.
- Let P be a path of G from x to y.
- $cost(P) = \sum_{e \in P} weight(e)$
- The shortest path is a path P which minimizes cost(P)



Dijkstra's algorithm

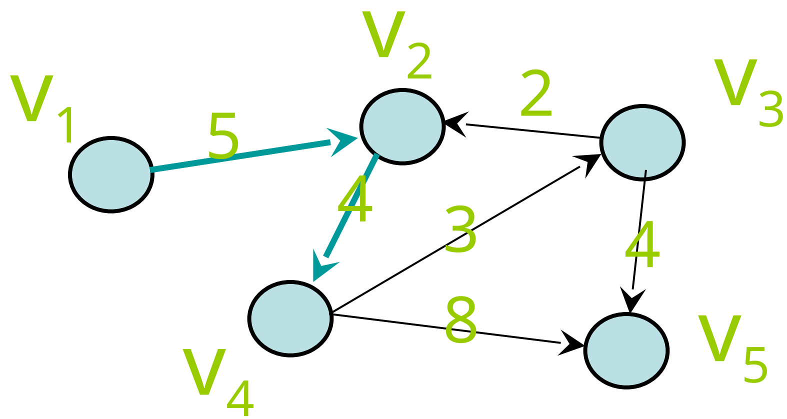
- Consider a graph G, each edge (u, v) has a weight $w(u, v) > 0$.
- Suppose we want to find the shortest path starting from v_1 to any node v_i
- Let VS be a subset of nodes in G
- Let $cost[v_i]$ be the weight of the shortest path from v_1 to v_i that passes through nodes in VS only.

Example for Dijkstra's algorithm



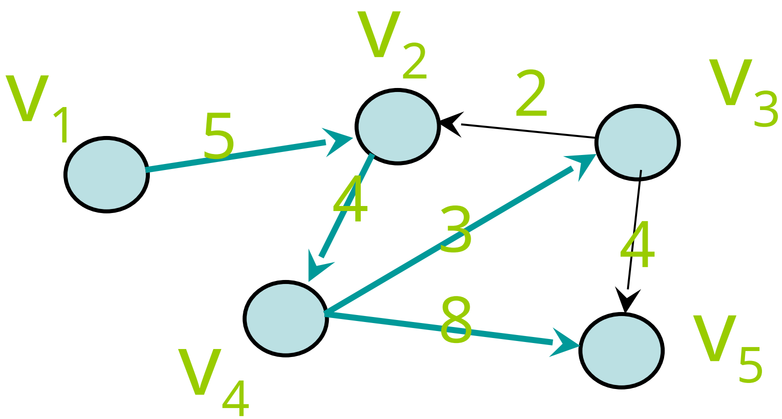
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞

Example for Dijkstra's algorithm



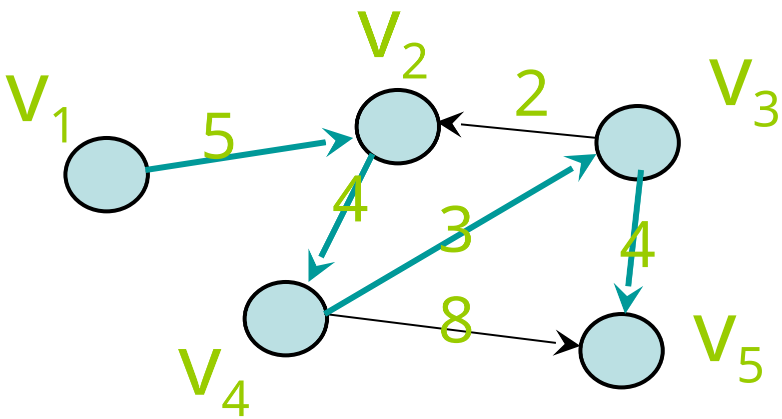
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞

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);
    }
}
```

Summary

- Graphs can be used to represent many real-life problems.
- There are numerous important graph algorithms.
- We have studied some basic concepts and algorithms.
  - Graph Traversal
  - Topological Sort
  - Spanning Tree
  - Minimum Spanning Tree
  - Shortest Path