Data Structures and Algorithms

Lecture 09 – Terminologies of Graph (Basics of Graph)

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

Problem Solving (1)——Social Networking

How can you determine if a friendship can be established between you and a stranger through several intermediaries?

Graph

What is a graph ? Why do we need it?

Graph Terminologies

Types of graphs, Graph Representation

Minimum Spanning Tree (*Krushal* and *Prim* Algo)

Traversal of Graph (*BFS* and *DFS*)

Topological Sorting

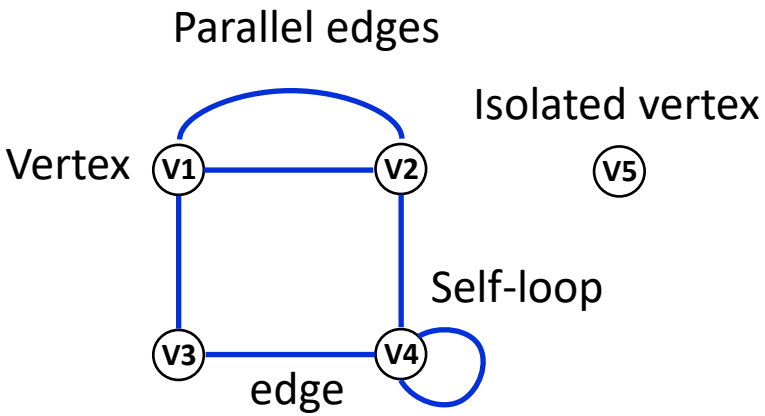
Single source shortest path (*BFS*, *Dijkstra* and *Ford* Algo)

All pairs shortest path (*Floyd Warshall* Algo)

What is Graph ?

Graph consists of a finite set of *V*ertices(or nodes) and a set of *E*dges which connect a pair of nodes

- A graph $G = (V, E)$ consists of two sets:
- V: non-empty set of vertices ($V \neq \emptyset$), vertex is the basic units of a Graph, can represent *people, web pages*, *cities* etc.
- E: set of edges, each edge connecting two vertices in V, represent *relationships* or *paths*.



Terminology of Graphs

Unweighted Graph: A graph which does not have a weight associated with any edge or vertex

Edge/vertex-Weighted Graph: A graph which has a weight associated with any edge (or vertex), could be positive and negative (*cost, latency, etc.*)

Undirected edge: Edge with no direction, denoted by parentheses, e.g., (u, v) is same as (v, u)

Undirected Graph: In case the edges of the graph do not have a direction associated with them

Directed edge: Edge with a direction, denoted by angle brackets, e.g., $\langle u, v \rangle$ means from u to v

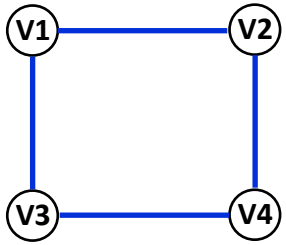
Directed Graph: If the edges in a graph have a direction associated with them

Cyclic Graph: A graph which has at least one loop

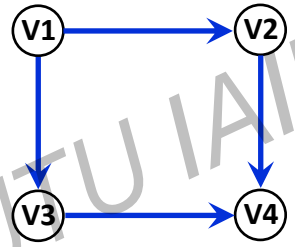
Acyclic Graph: A graph with no loop

Types of Graphs

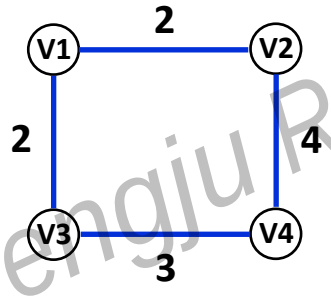
Unweighted Undirected



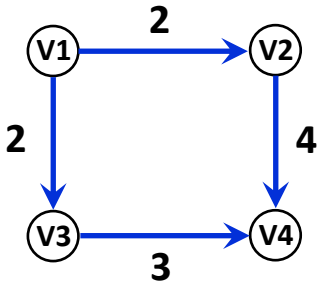
Unweighted directed



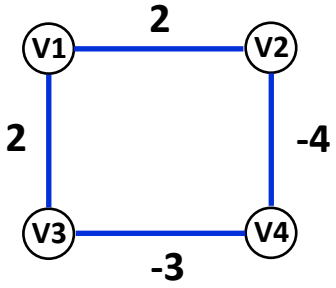
Positive weighted Undirected



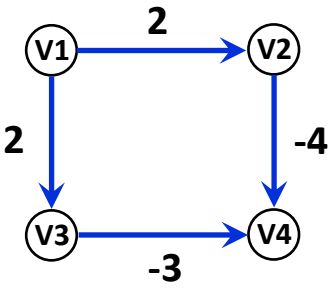
Positive weighted directed



Negative weighted Undirected



Negative weighted directed



Common Types of Graphs

Tree: A connected acyclic graph, |E|=|V|−1

Forest : An acyclic graph

Complete graph: Every two distinct vertices are adjacent. An undirected complete graph has $n(n-1)/2$ edges.

Bipartite graph: Vertices can be partitioned into two sets such that every edge connects vertices from different sets (no edge inside a set)

Sparse graph: Number of edges is much less than V^2, typically $E = O(V)$

Dense graph: Number of edges is close to V^2, $E = O(V^2)$

DAG: Directed Acyclic Graph

Degree of a Vertex

Undirected graph: The degree $deg(v)$ of vertex v = number of edges incident to v.

Directed graph:

- **In-degree** $indeg(v)$ = number of edges with v as the head.
- **Out-degree** $outdeg(v)$ = number of edges with v as the tail.
- **Degree** $deg(v) = indeg(v) + outdeg(v)$.

Handshaking Lemma: For any undirected graph, the sum of all vertex degrees = 2|E| (each edge contributes 2 to the sum). For a directed graph, sum of all in-degrees = sum of all out-degrees = |E|.

Paths, Cycles, and Connectivity

Path: a sequence of vertices where each adjacent pair is connected by an edge.

Simple path: a path in which *no vertices*(and thus no edges) are repeated. *e.g <V1, V2, V4, V6>*

Cycle: A path with the same start and end vertex and length ≥ 1.

Walk: a sequence of vertices where each adjacent pair is connected by an edge

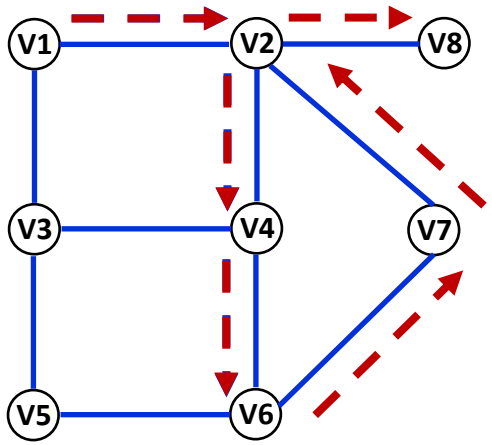
Closed Walk: starts and ends at same vertex

Trail: a walk in which *no edges* are repeated

e.g <V1, V2, V4, V6, V7, V2, V8>

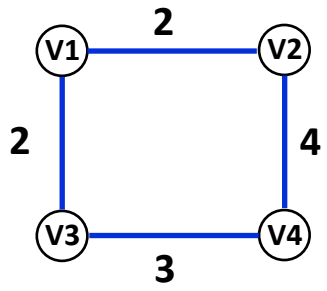
Strongly Connected Graphs: there exist at least one path from any vertex to any other vertex

Connected component: A maximal connected subgraph of an undirected graph.



Graph Representation

Vertex and Edge Array: *G={V, E}*



Vertex Array

V1
V2
V3
V4

Edge Array

V1	V2	2
V1	V3	2
V2	V4	4
V3	V4	3

<start, end> weight>

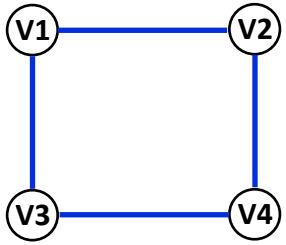
Space Complexity: $O(V+E)$

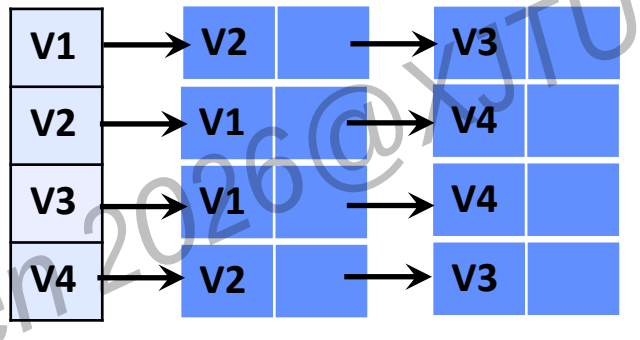
Time Complexity of find all nodes adjacent to a given node? $O(E)$

check if given nodes are connected? $O(E)$

Graph Representation

Adjacency List: an unordered list used to represent a graph. Each list describes the set of neighbors of a vertex in the graph (*Sparse preferred*) $E \ll V^2$





Space Complexity: $O(E+V)$

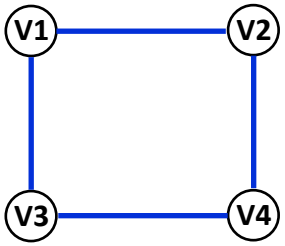
Time Complexity of find all nodes adjacent to a given node? $O(V)$

check if given nodes are connected? $O(V)$

Disadvantages: Checking edge existence may require traversing the list, $O(degree)$**, and each edge stored twice for Undirected graph.**

Graph Representation

Adjacency Matrix: a square matrix, and the elements of the matrix indicate whether pairs of vertices are adjacent or not in the Graph (*Dense preferred*)



	V1	V2	V3	V4
V1	∞	1	1	∞
V2	1	∞	∞	1
V3	1	∞	∞	1
V4	∞	1	1	∞

$$A_{ij} = \begin{cases} 1 & if \ \exists edge \ from \ i \ to \ j \\ 0 & otherwise \end{cases}$$

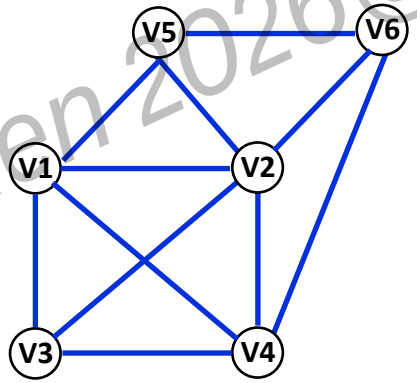
Space Complexity: $O(V^2)$

Time Complexity of find all nodes adjacent to a given node? $O(V) + O(V)$

check if given nodes are connected? $O(1)$

Disadvantages: Space $O(n^2)$,<u>wasteful for sparse graphs</u>, adding/removing vertices is expensive.

Quiz1 : How to count the number of Triangles in an Undirected Graph ?



Powers of the Adjacency Matrix

Theorem: A is the adjacency matrix of a graph G, Then, for any positive integer k, the element $(A^k)_{ij}$ in row i, column j of A^k equals the number of **different walks** of length exactly k from vertex i to vertex j.

Proof: By induction on k.

Base case $k = 1$: By definition of the adjacency matrix, $(A)_{ij}$ is exactly the number of walks of length 1 from i to j (0 or 1). So the statement holds.

Induction hypothesis: Assume that for some $k \geq 1$, $(A^k)_{ij}$ equals the number of walks of length k from i to j.

Inductive step: Consider $A^{k+1} = A^k \cdot A$ .Then

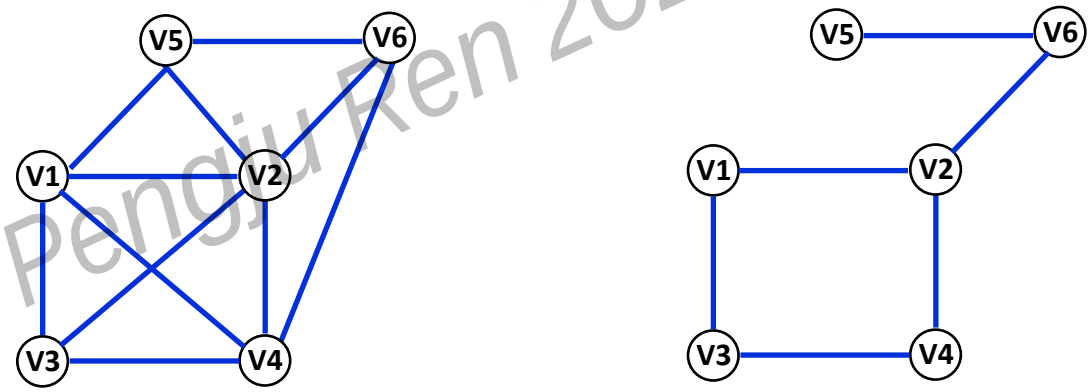
$$(A^{k+1})_{ij} = \sum_{t=1}^{n} (A^k)_{it} \cdot A_{tj}$$

Quiz2: **How to count the number of Triangles in a *directed Graph* ?**

Subgraph

Given a graph $G = (V, E)$,if there exists a graph $H = (V', E')$ such that, then H is called a **subgraph** of G.

$$V' \subseteq V$$
$$E' \subseteq E$$

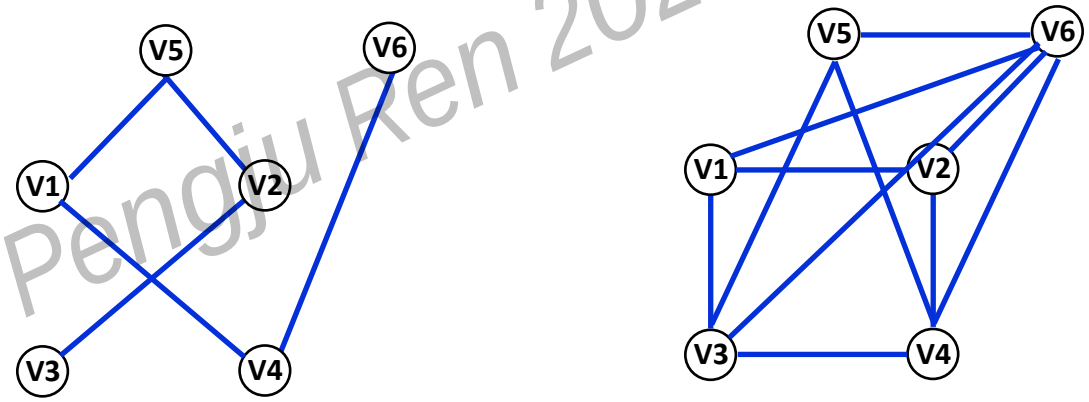


Complement Graph

For a simple undirected graph $G = (V, E)$, its **complement** $\bar{G} = (V, \bar{E})$ satisfies:

$$V(\bar{G}) = V(G)$$

$$\bar{E} = \{\{u, v\}, | \, u \neq v \, \{u, v\} \notin E\}$$



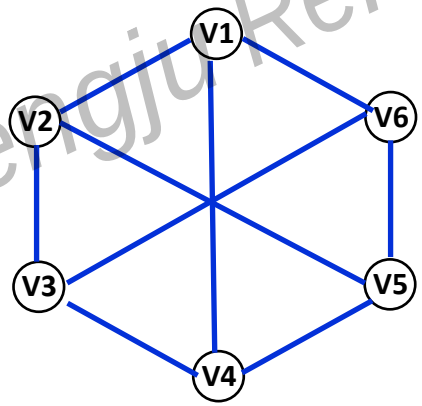
Isomorphism

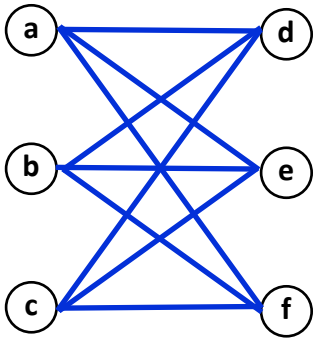
Two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are **isomorphic** (denoted $G \cong H$) if there exists a bijection $\varphi: V_G \to V_H$ such that for every pair of vertices $u, v \in V_G$:

$$\{u, v\} \in E_G \iff \{\varphi(u)\varphi(v)\} \in E_H$$

For directed graphs, the condition becomes:

$$\langle u, v\rangle \in E_G \iff \langle \varphi(u)\varphi(v)\rangle \in E_H$$

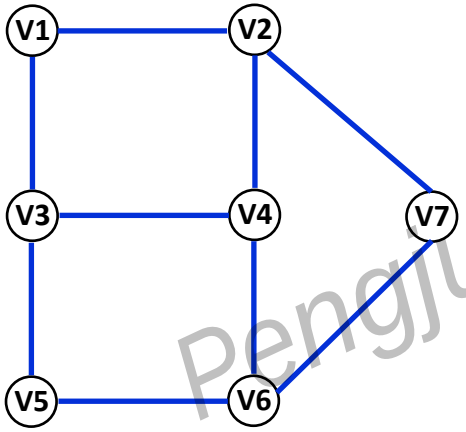


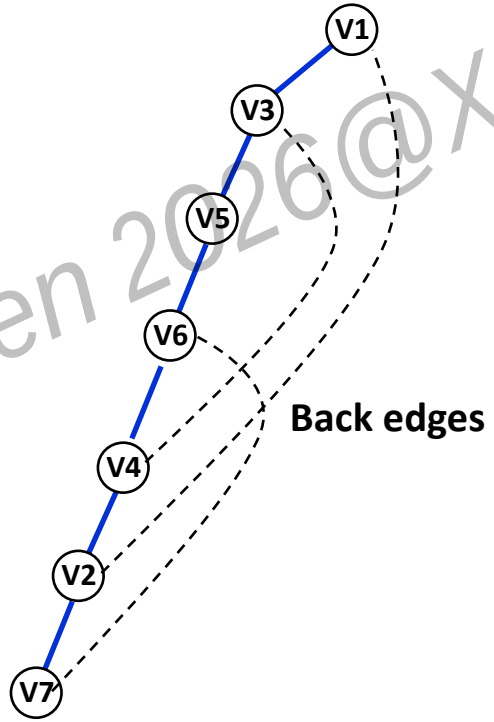


Intuitively: The two graphs have the same structure; only the vertex labels differ.

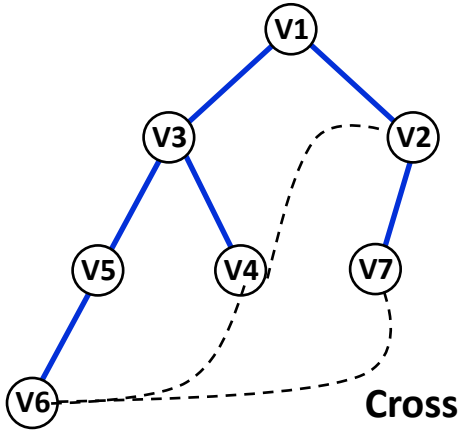
Graph Traversal

Graph traversal means visiting all vertices in a systematic way without repetition, two basic methods: BFS and DFS





DFS Spanning Tree

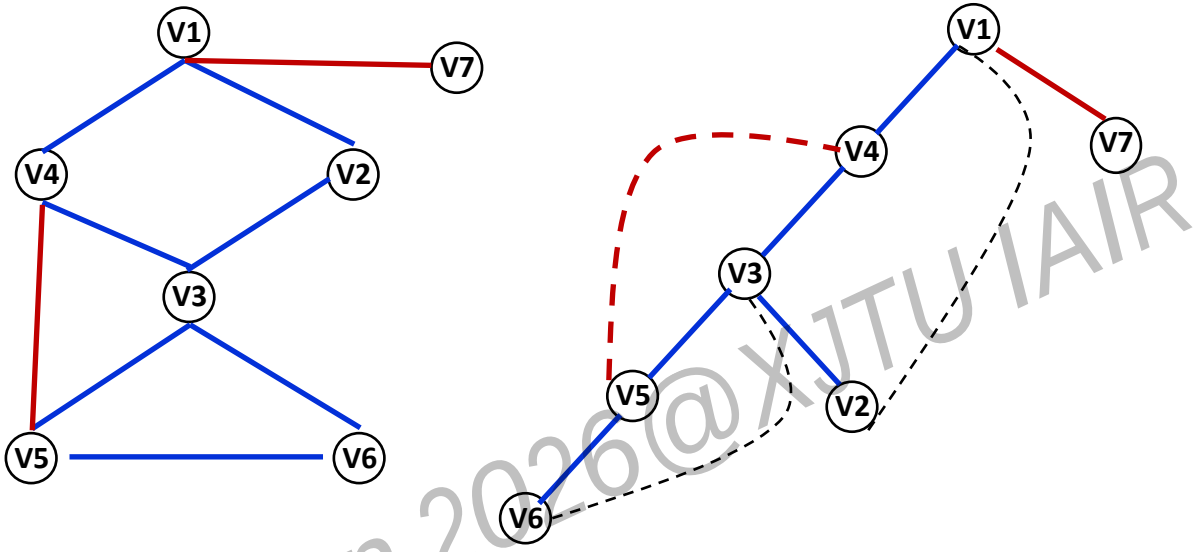


BFS Spanning Tree

Recap: Problem Solving (1))——Social Networking

How can you determine if a friendship can be established between you and a stranger through several intermediaries?

DFS and Articulation Point



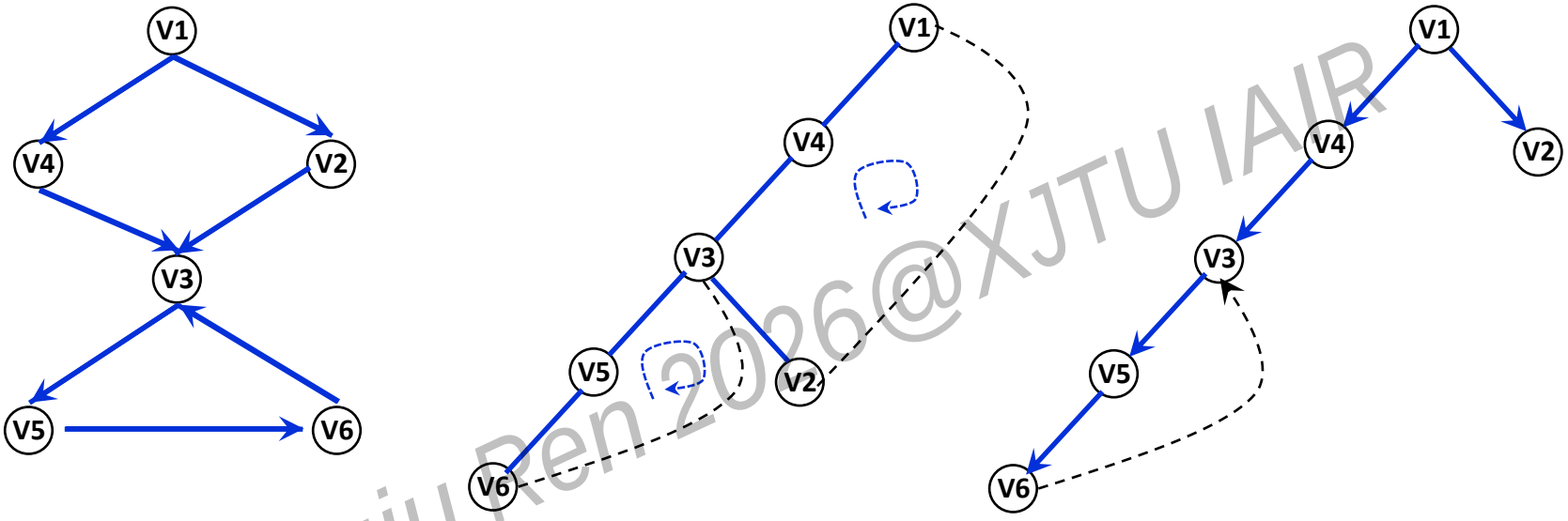
	1	2	3	4	5	6
dfn	1	6	3	2	4	5
low	1	1	1	1	3	3
low	1	1	1	1	2	2

For no-root node u, edge (u,v): if $L[v] \geq d[u]$ then u is *Articulation Point*

e.g. (V3-V5), L[V5]=3, d[V3]=3, so V3 is *Articulation Point*

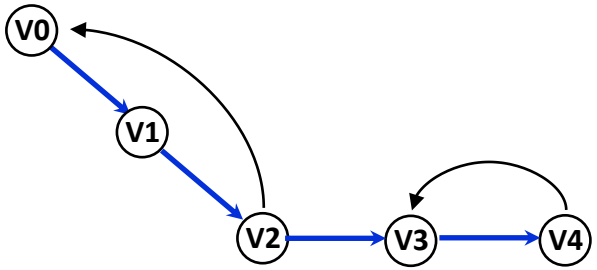
For root: if its #children ≥ 2, the root is *Articulation Point*

DFS and Cycle Detection



- **Undirected graph: During DFS, if an edge connecting the current vertex to an already visited vertex that is not the direct parent of the current vertex, then a cycle exists.**
- **Directed graph: During DFS, use three-color marking (white = unvisited, gray = in the recursion stack, black = processed). If you explore an edge to a vertex that is gray, which indicates the presence of a cycle.**

DFS and Strong Connected Components(SCC)



	0	1	2	3	4
dfn	1	2	3	4	5
low	1	2	1	4	4

Push visited vertices onto a stack. Whenever *low[u] == dfn[u]*, pop vertices from the stack up to and including *u*; these vertices form an SCC.

E.g. *SCC 1: {3, 4}, SCC 2: {0, 1, 2}*

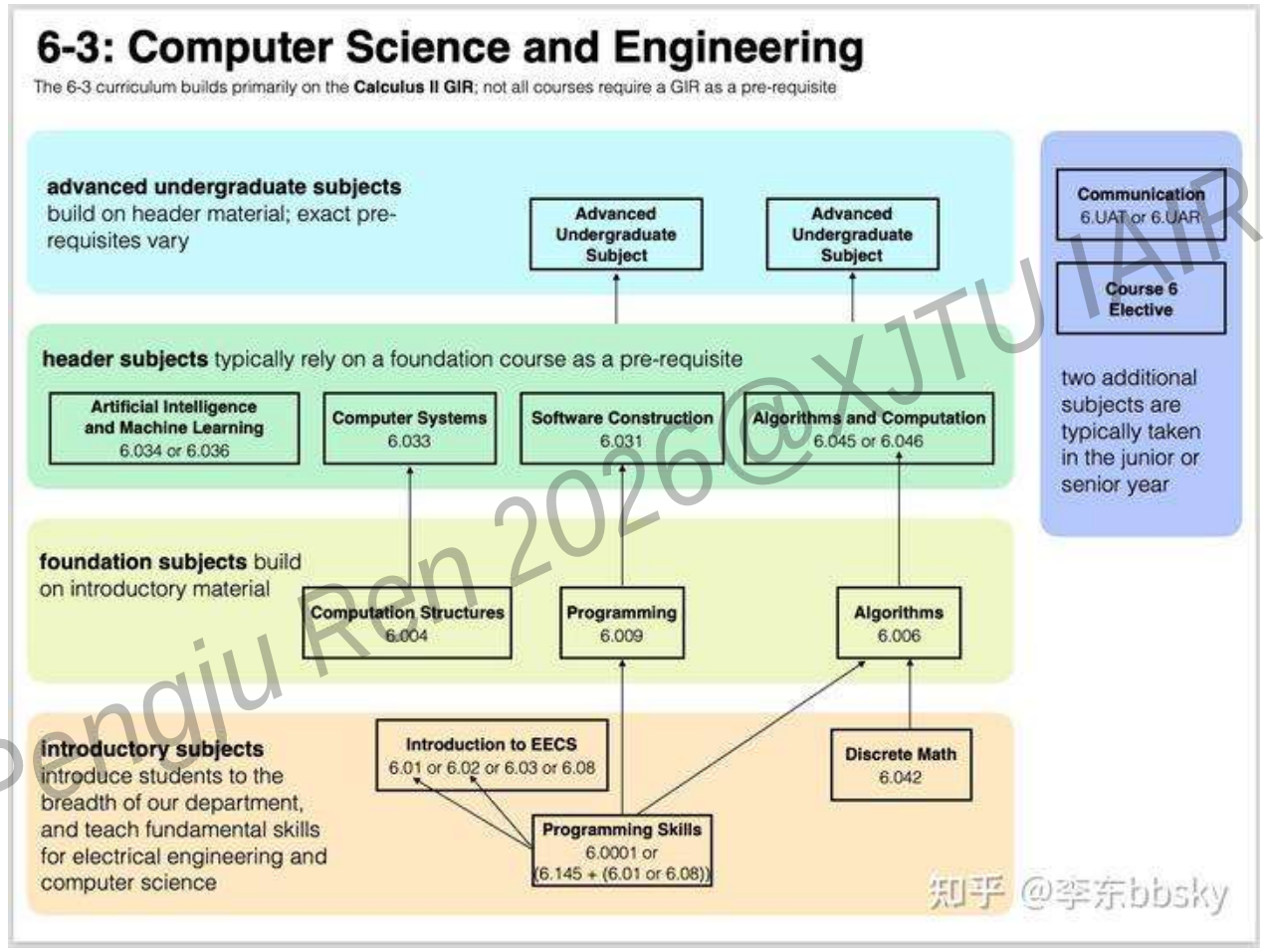
BFS and Applications

Single-Source Shortest Paths in Unweighted Graphs： **The moment a vertex is discovered for the first time, we have already found the shortest path from the source to that vertex.**
Applications: friend distance in social networks, minimum number of steps in a maze.

Bipartite Graph Detection (Coloring Method)： **During BFS, assign two colors (0 and 1) alternately to the vertices of each layer. If an edge is found whose two endpoints share the same color, the graph is not bipartite.**

Finding the Diameter (Diameter of a Tree)： **the diameter of a tree (or a general graph) —is the longest shortest path. two runs of BFS suffice to obtain the diameter: start from an arbitrary vertex *a*, run BFS to find the farthest vertex *b*; then run BFS from *b* to find the farthest vertex *c*. The distance between *b* and *c* is the diameter.**

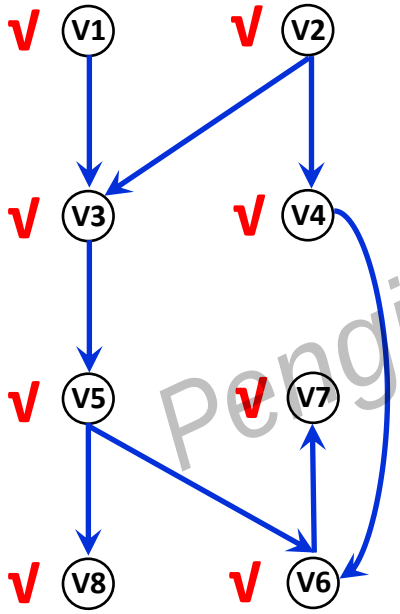
Problem Solving (2) —— Prerequisite Courses



Course selection for MIT's Computer Science programs requires comprehensive consideration of prerequisite courses. Please provide students with a reasonable course plan.

Topological Sort

Topological Sort: Sorts given actions in such a way that if there is a dependency of one action on another, then the dependent action always comes later than its parent action

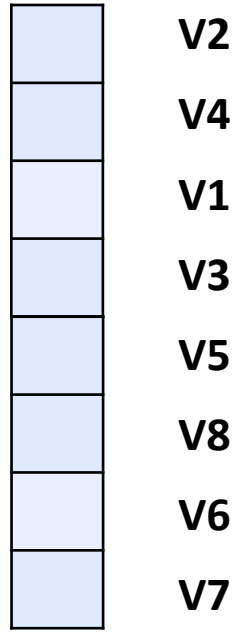



```
if a vertex depends on currentVertex:
   go to that vertex then come back to
currentVertex
else
   push currentVertex to Stack
```

V1,V2,V3,V4,V5,V6,V7,V8

V2,V1,V3,V4,V5,V6,V8,V7

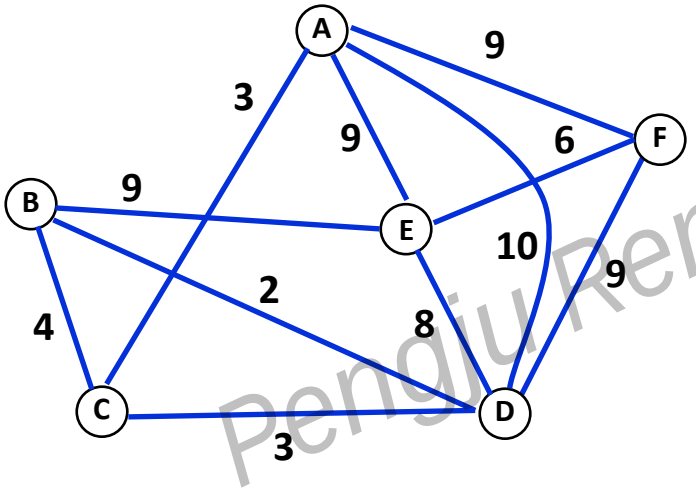
V2,V4,V1,V3,V5,V8,V6,V7

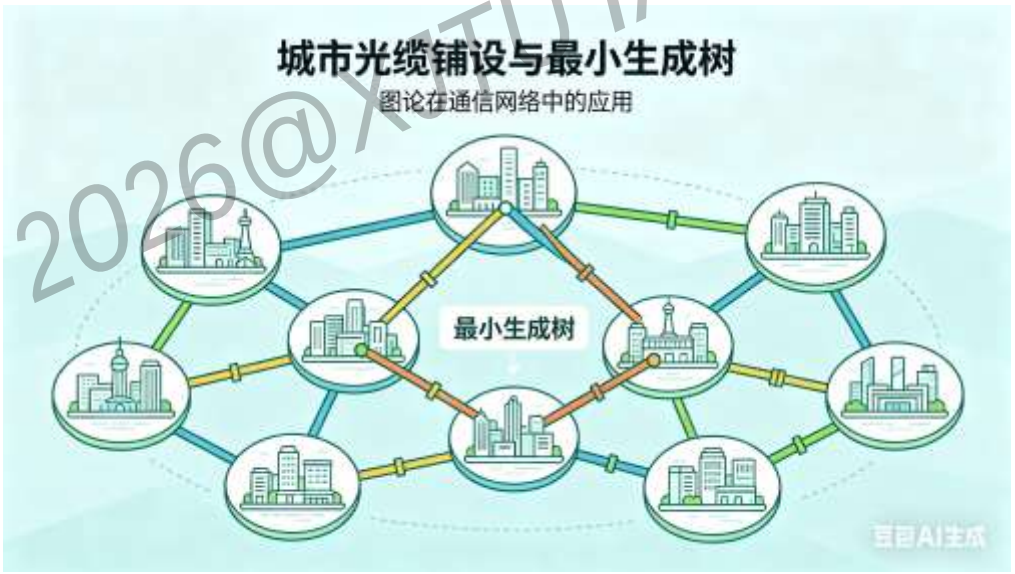


Stack

Problem Solving (3)——MST

A city plans to lay fiber-optic cables to connect all neighborhoods. The costs of laying cables between different neighborhoods vary (due to distance, terrain, etc.). How can you connect all neighborhoods at the lowest total cost?

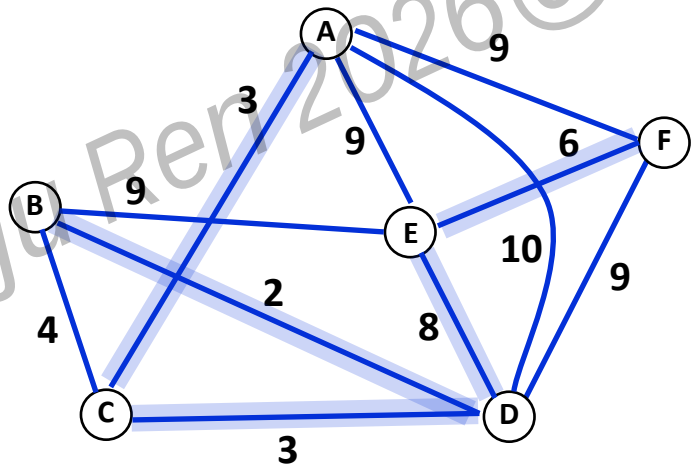




Given a connected, undirected graph $G = (V, E)$ with a real-valued weight function $w(e)$ defined on the edges, a ***Minimum Spanning Tree(MST)*** is a subset of edges $T \subseteq E$ that connects all the vertices in V, contains no cycles, and has the minimum possible total edge weight $\sum_{e \in T} w(e)$.

MST Solution : Kruskal

1. **Sort all edges by weight increasingly.**
2. **Process each edge: if it connects two vertices that are currently in different connected components (i.e., adding it does not form a cycle using *Find and Union*), add it to the MST and merge the two components.**
3. **Stop when the MST has $|V|-1$ edges.**



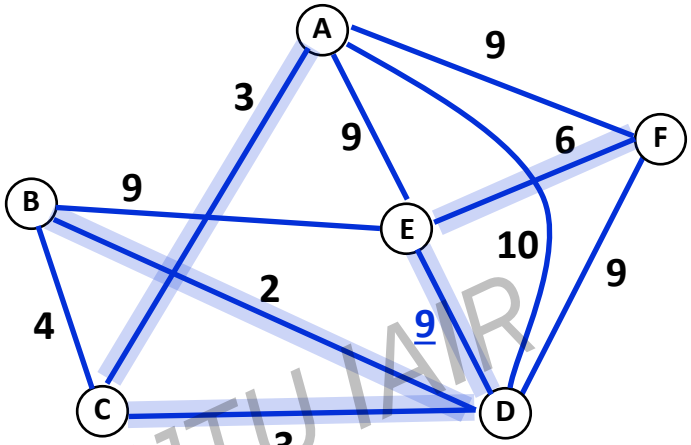
$(B,D),(C,D),(A,C),(\cancel{B,C}),(E,D),(E,F)$

MST Solution : Kruskal

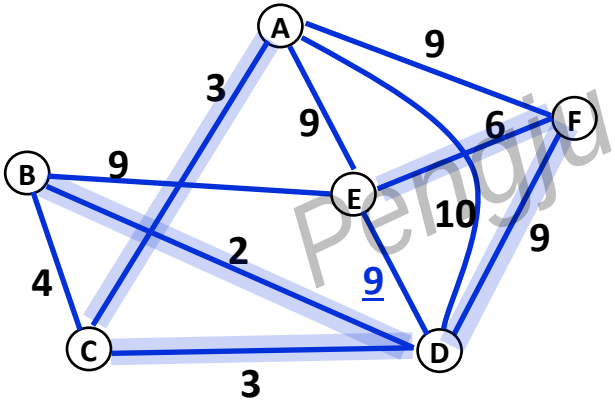
Time Complexity: $O(E \log E)$

Space Complexity: $O(E + V)$

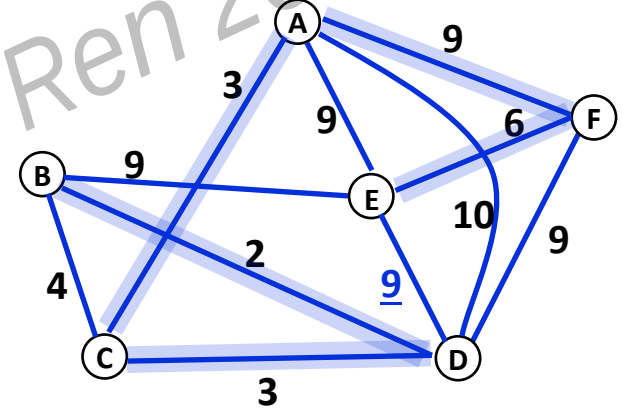
Kruskal is *Greedy* and *Optimal*



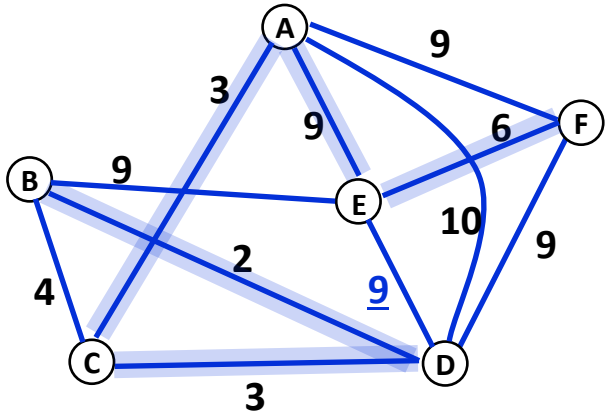
K



T



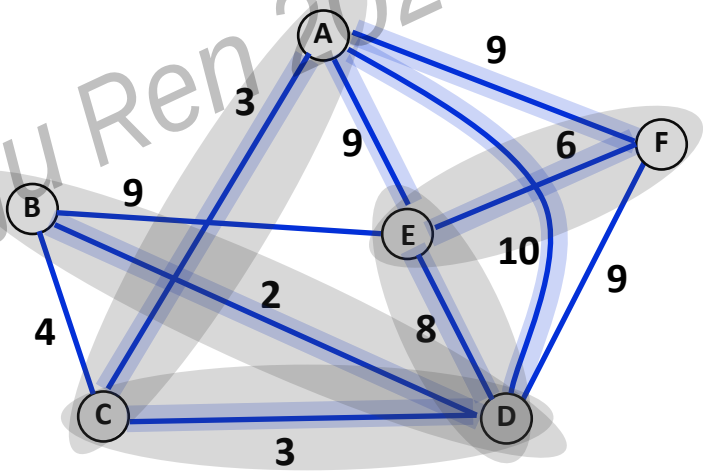
T'



T"

MST Solution: Prim's Algorithm

- Start from an arbitrary vertex, maintain a set S of vertices already in the tree. Each step, choose the lightest edge connecting S to $V \setminus S$, add its other endpoint to S.
- Repeat until all vertices are in S. Similar to Dijkstra's algorithm, but the update rule is "*maintain the minimum edge weight from each vertex outside S to S*."

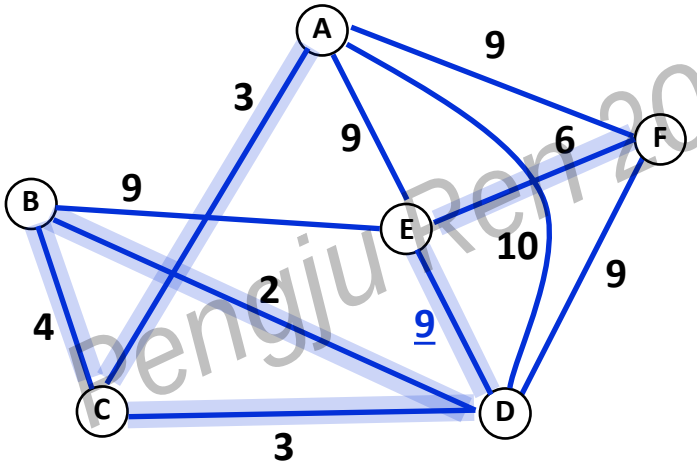


MST Solution : Prim's Algorithm

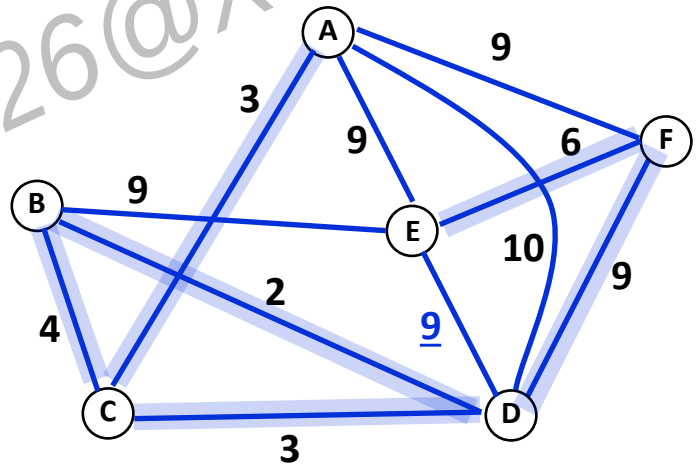
Time Complexity: $O(E \log V)$

Space Complexity: $O(E + V)$

Prim is *Greedy* and *Optimal*



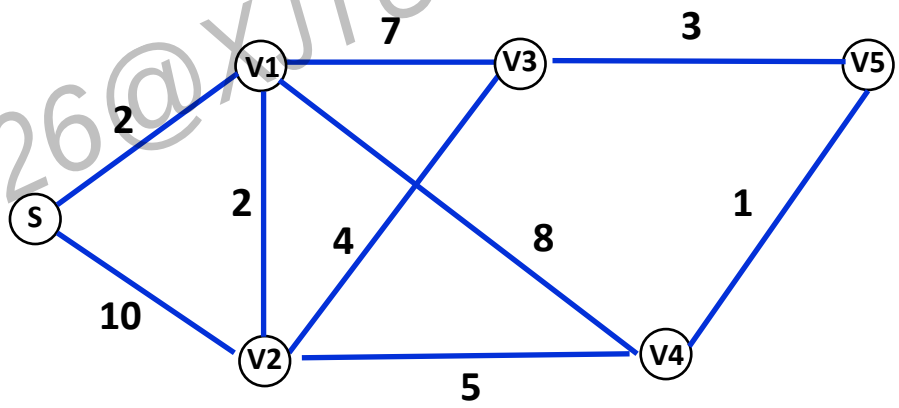
P



T

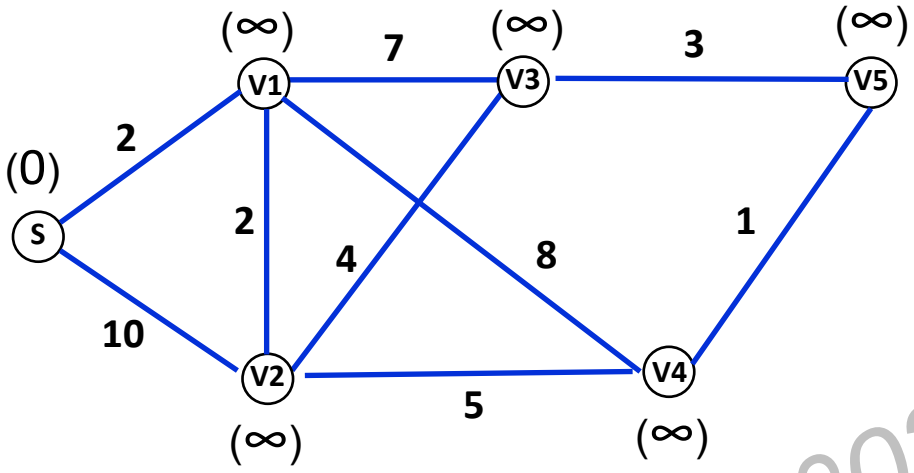
Problem Solving (4)——SSSP

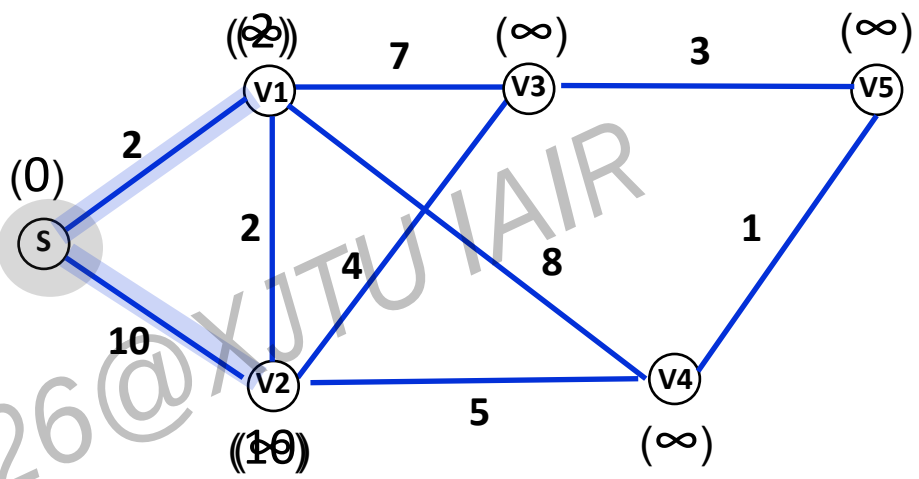
The Single-Source Shortest Paths （SSSP） problem involves finding the shortest paths from a given source vertex to all other vertices in a weighted graph.

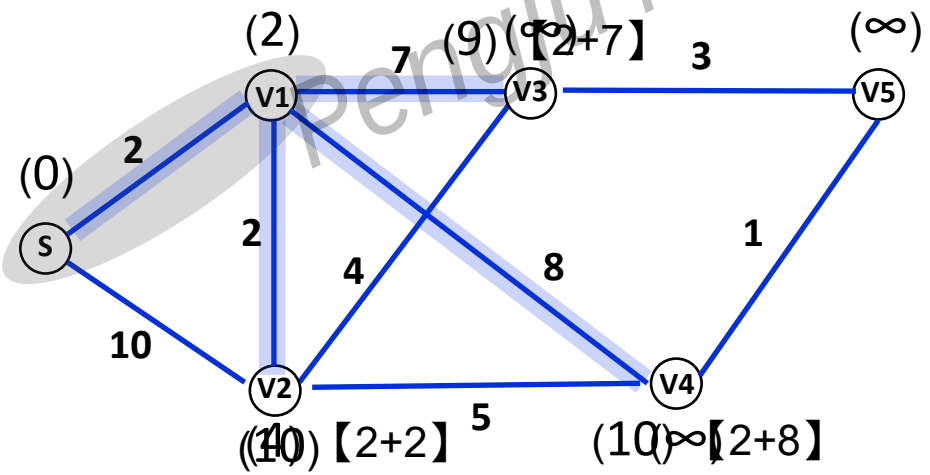


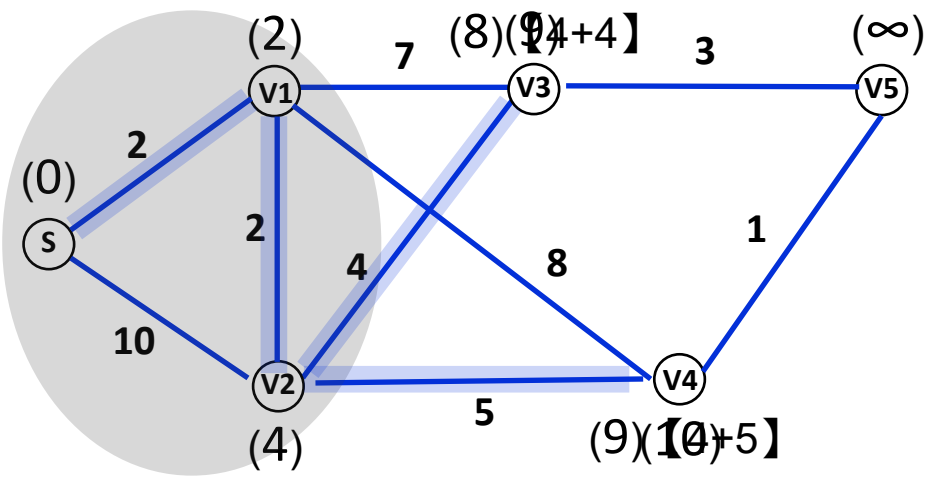
Given a directed/undirected graph $G = (V, E)$ with *non-negative edge weights* and a source vertex $s \in V$, compute the shortest path distance $d(s, v)$ and a corresponding shortest path for every $v \in V$.

SSSP Solution：Dijkstra

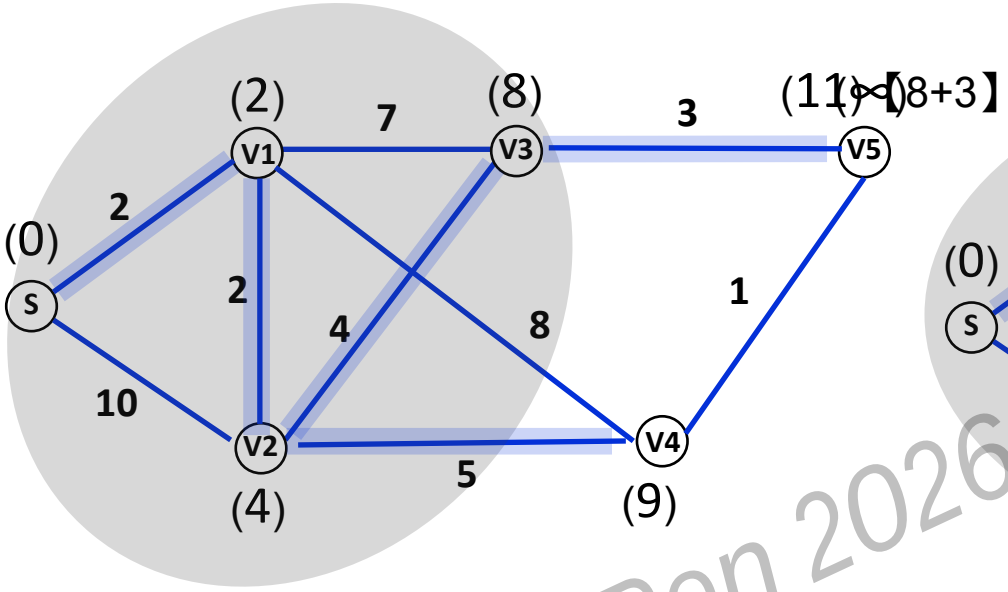


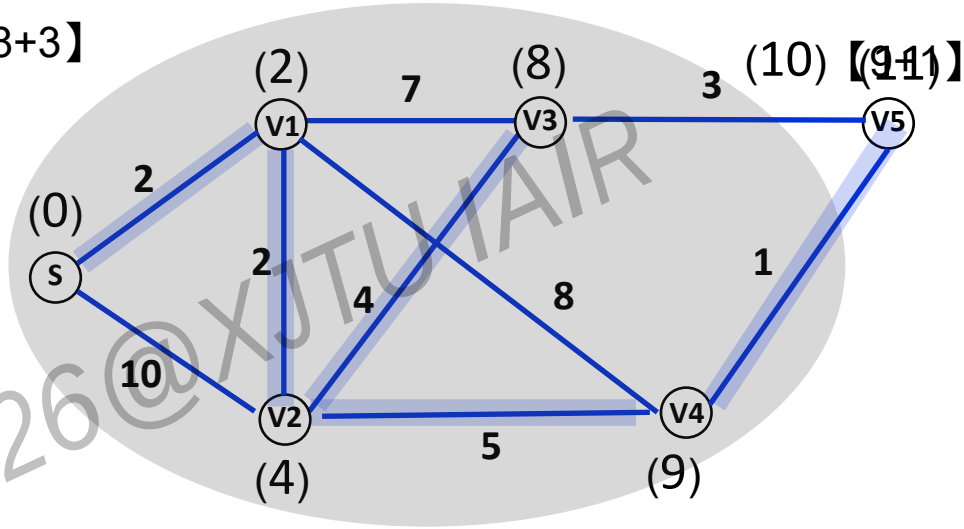


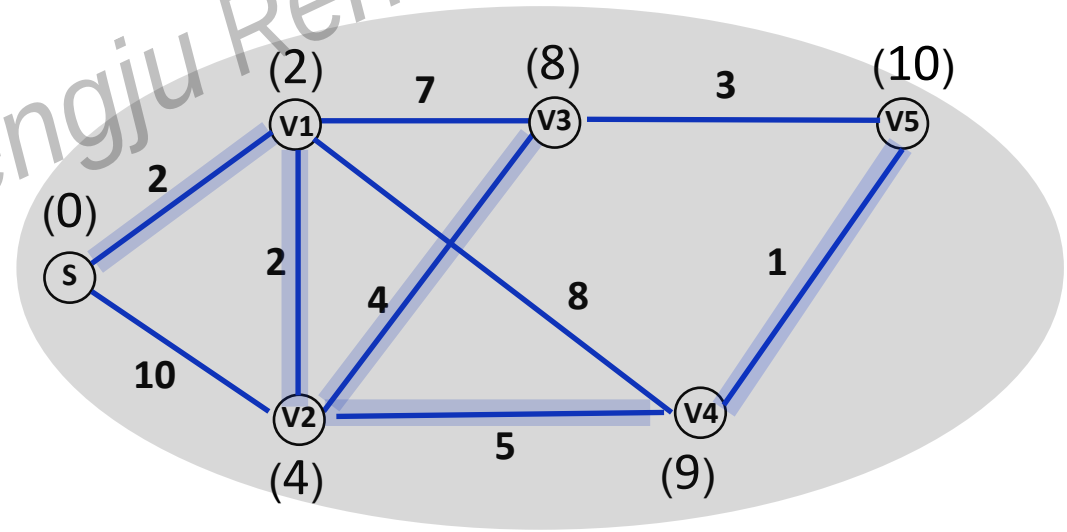




SSSP Solution：Dijkstra







SSSP Solution：Dijkstra

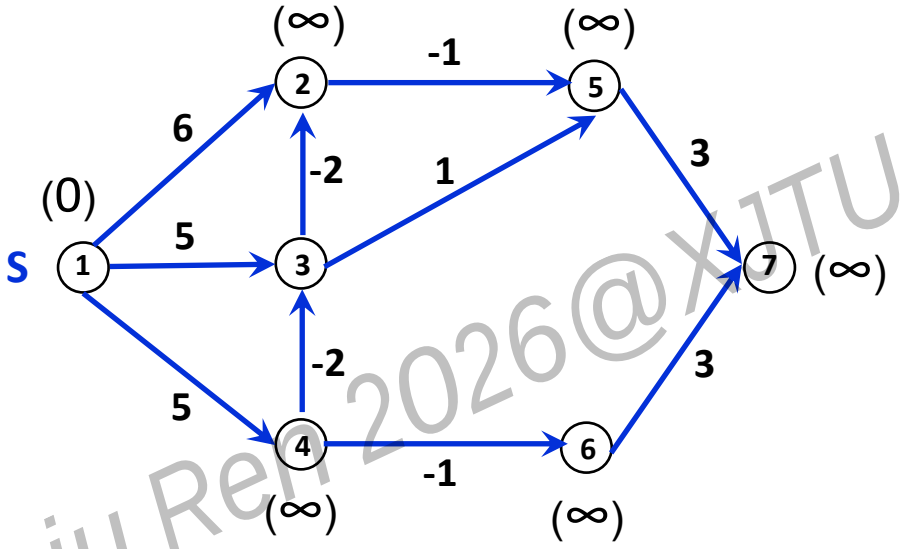
```
def dijkstra(adj, s):
    # adj: 邻接表, adj[u] = [(v, w), ...]
    n = len(adj)
    dist = [float('inf')] * n
    dist[s] = 0
    prev = [-1] * n
    pq = [(0, s)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:  # 过期条目跳过
            continue
        for v, w in adj[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                prev[v] = u
                heapq.heappush(pq, (dist[v], v))
    return dist, prev
```


Time Complexity: $O(E \log V)$

Space Complexity: $O(E + V)$

Dijkstra is *Greedy* and *Optimal* (there is NO negative edge)

SSSP Solution : Bellman-Ford



$$For\ every\ edge < u, v, w >:\ \ if\ dist[u] + w < dist[v]\ then\ \ dist[v] = dist[u] + w$$

$$|V| = 7, \qquad \#\ Relax\ iteration = 7 - 1 = 6$$

$$edge\ list = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle, \langle 2,5\rangle, \langle 3,2\rangle, \langle 3,5\rangle, \langle 4,3\rangle, \langle 4,6\rangle, \langle 5,7\rangle, \langle 6,7\rangle\}$$

SSSP Solution：Bellman-Ford

```
def bellman_ford(n, edges, s):
    dist = [float('inf')] * n
    dist[s] = 0
    prev = [-1] * n
    # 松弛 V-1 次
    for _ in range(n-1):
        updated = False
        for u, v, w in edges:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                prev[v] = u
                updated = True
        if not updated:
            break
    # 检测负环
    for u, v, w in edges:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            return None, None  # 存在负环
    return dist, prev
```

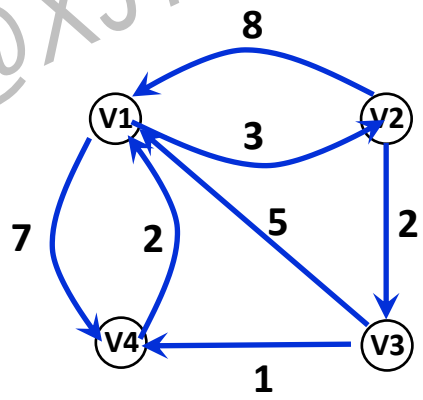

Time Complexity: $O(EV)$

Space Complexity: $O(E+V)$

Bellman-Ford is *Dynamic Programming* and *Optimal* (negative edge is OK)

Problem Solving (5)——APSP

Finding a path between every vertex to all other vertices in a graph such that the distance between them (source and destination) is minimum (*All Pairs Shortest Path, APSP*)



Given a weighted (directed or undirected) graph $G = (V, E)$, the **APSP** problem is to find the shortest path distance (and the path itself) between **every pair of vertices** $u, v \in V$.

ASAP Solution：Floyd-Warshall

```
def floyd_warshall(graph):
    """
    graph: 邻接矩阵, graph[i][j] = weight (float('inf') 表示无边)
    返回距离矩阵和路径后继矩阵
    """
    n = len(graph)
    dist = [row[:] for row in graph]
    # 初始化 next 矩阵用于路径重构
    nxt = [[None]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if i != j and dist[i][j] != float('inf'):
                nxt[i][j] = j
    # Floyd-Warshall 核心
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
                    nxt[i][j] = nxt[i][k]  # 记录路径
    # 检测负环：若某顶点到自身距离 < 0，则有负环
    for i in range(n):
        if dist[i][i] < 0:
            raise ValueError("图中存在负环")
    return dist, nxt
```

ASAP Solution : Floyd-Warshall

Time Complexity: $O(V^3)$

Space Complexity: $O(V^2)$

Floyd-Warshall is *Dynamic Programming* and *Optimal* (negative edge is OK)

$$d_{ij}^{(k)} = \min\left(d_{ij}^{k-1},\ d_{ik}^{k-1} + d_{kj}^{k-1}\right) \quad k = 0, \ldots\ldots,\ n$$

- When $k = n$, the set of allowed intermediate vertices is the full vertex set, so $d_{ij}^{(n)}$ is precisely the length of the true global shortest path from i to j
- If there exists a vertex i such that $d_{ii}^{(n)} < 0$,then the graph contains a negative cycle

Shortest path algorithms of Graph

	BFS	Dijkstra	Bellman Ford	Floyd Warshall
Time Complexity	$O(V+E)$	$O(E \log V)$	$O(EV)$	$O(V^3)$
Space Complexity	$O(V+E)$	$O(V+E)$	$O(V+E)$	$O(V^2)$
Implementation	Easy	**Moderate**	**Moderate**	**Moderate**
Unweighted Graph (same as hop-count)	**√**	**√**	**√**	**√**
Weighted Graph	X	**√**	**√**	**√**
Negative Edge	X	X	**√**	**√**
Negative Cycle Det	X	X	**√**	**√**
Limitation	**SSSP** **Not work for Weighted Graph**	**SSSP**	**SSSP**	**APSP** **Poor support for dynamic graph with minor modifications**

Summary

- **Graph**: A graph consists of vertices and edges, and can be directed or undirected. It is represented by adjacency matrices or lists and serves as a basic mathematical model of networks.
- **Minimum spanning tree**: A subgraph that connects all vertices with minimal total edge weight and no cycles, commonly built by Prim's or Kruskal's greedy algorithms.
- **Graph traversal**: Depth-first search goes deep along branches and backtracks; breadth-first search expands layer by layer. Both are fundamental to search and path problems.
- **Shortest path**: Finding a path with minimum total edge weight between two vertices in a weighted graph. *Dijkstra* handles non-negative weights, while *Bellman–Ford* copes with negative edges, *Floyd* can solve all pairs shortest path problem.