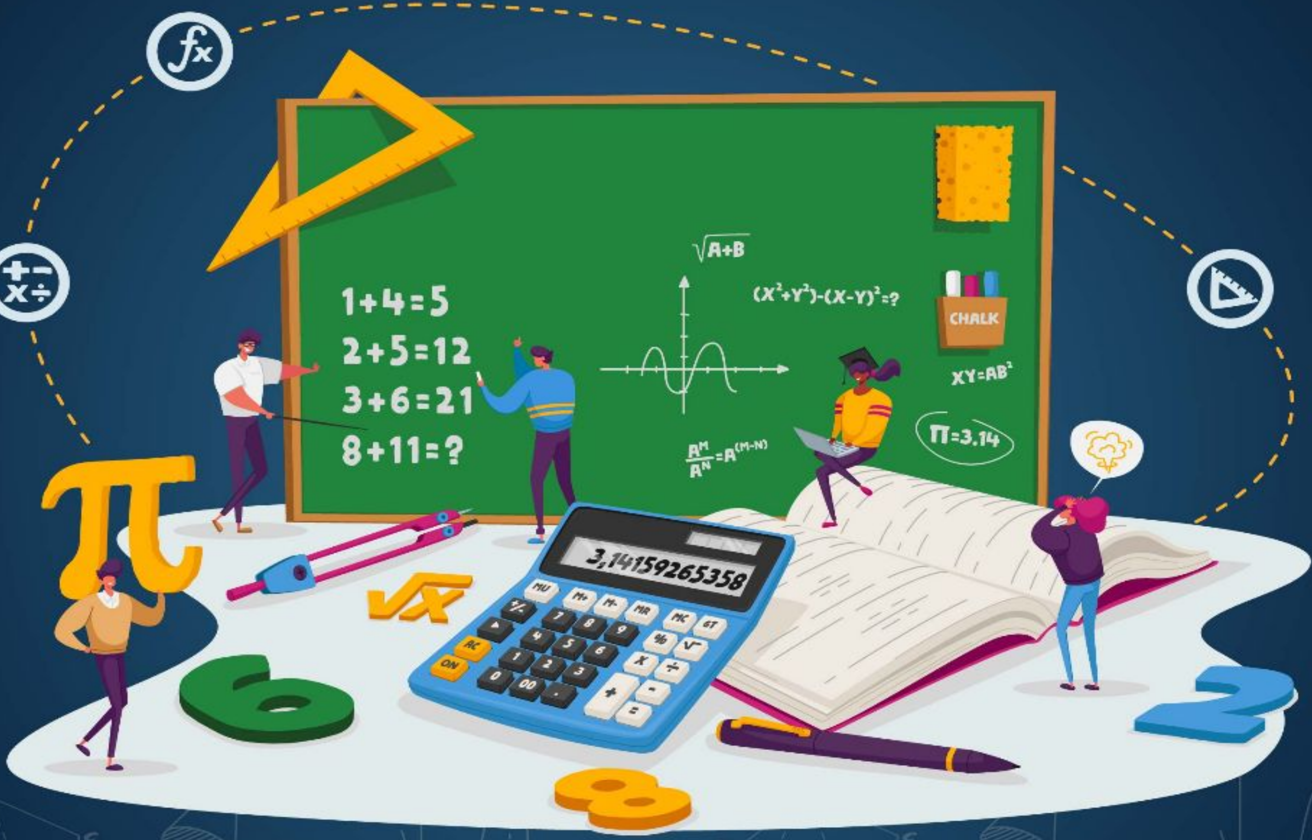
IIT MADRAS **BS DEGREE**



MATHEMATICS

Graph theory with applications

Srikanth Venkatesan

About the author

Srikanth Venkatesan is an accomplished and highly regarded instructor with extensive expertise in Mathematics and Statistics. Driven by a deep passion for these subjects, Srikanth has dedicated his career to advancing knowledge and fostering a deeper understanding within these fields.

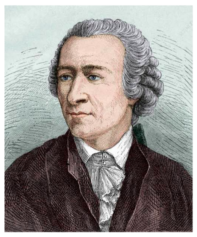
Srikanth holds a Master's degree in Mathematics from the prestigious IIT Madras, where he specialized in the subject. Alongside his academic pursuits, Srikanth actively shares his wealth of knowledge with others. He has been actively engaged in mentoring and guiding IITM BS students in Mathematics and Statistics, imparting valuable insights and fostering their growth.

With a dynamic and engaging teaching style, Srikanth has successfully delivered captivating lectures to diverse audiences. His ability to inspire others to explore and excel in the realm of Mathematics and Statistics is widely recognized.

Drawing from his extensive experience and expertise, Srikanth brings a profound depth of knowledge to this textbook. Readers will undoubtedly benefit from his keen understanding and unique insights. Srikanth's unwavering commitment to excellence and his unwavering passion for Mathematics continue to drive his significant contributions to the field, both as an esteemed author and as a respected instructor.

Contents

1. Graphs and general graph problems



"Logic is the foundation of the certainty of all the knowledge we acquire"

— Leonhard Euler

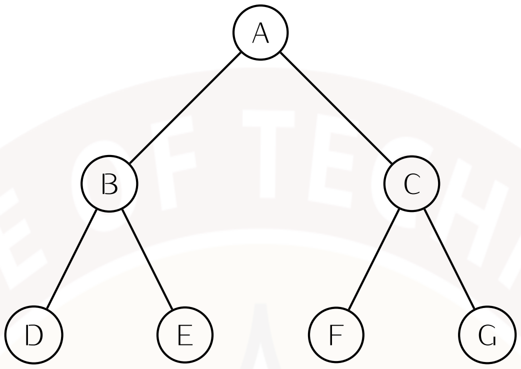
1.1 Introduction

Many real-world situations can conveniently be described by means of a diagram consisting of a set of points together with lines joining certain pairs of these points. For example, the points could represent people, with lines joining pairs of friends; or the points might be communication centres, with lines representing communication links. Notice that in Such diagrams one is mainly interested in whether or not two given points are joined by a line; the manner in which they are joined is immaterial. A mathematical abstraction of situations of this type gives rise to the concept of a graph.

1.2 Graph

Definition 1.2.1. In graph theory, a *graph* is a mathematical structure used to model pairwise relationships between objects. A graph consists of a set of vertices (also known as nodes or points) and a set of edges (also known as arcs or lines) connecting pairs of vertices.

Example 1.2.1. .

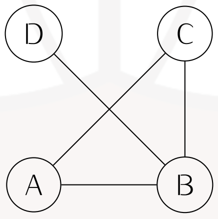


In this graph, the set of vertices(V) is A, B, C, D, E, F, G, and the set of edges is $(A, B), (A, C), (B, D), (B, E), (C, F), (C, G)$. Each edge connects a pair of vertices, such as the edge between A and B, which is denoted by (A, B).
This graph is an undirected graph because all edges are bidirectional; for example, the edge (B, D) connects B to D and also connects D to B.

1.3 Types of graphs

1.3.1 Simple Graph

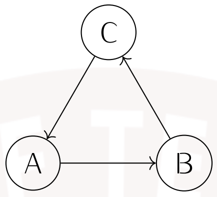
A *simple graph* is a graph with no loops or multiple edges between the same two vertices.



In our course, we consider only simple graphs.

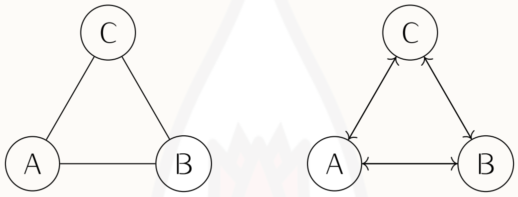
1.3.2 Directed Graph

A directed graph is a graph in which edges have a direction, i.e., they are ordered pairs of vertices. In a directed graph, if $(A, B) \in E$, then it is not necessary that $(B, A) \in E$.



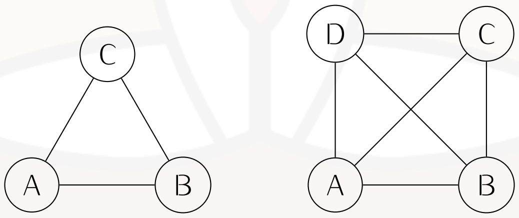
1.3.3 Undirected Graph

An undirected graph is a graph in which edges have no direction or all edges are bidirectional. In an undirected graph, if $(A, B) \in E$, then it implies that $(B, A) \in E$.



1.3.4 Complete Graph

A complete graph is a simple graph in which every pair of distinct vertices is connected by an edge.

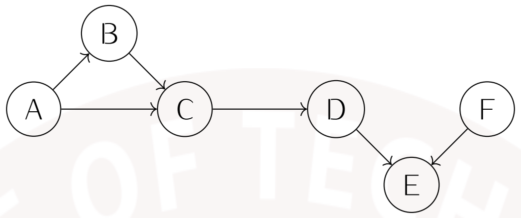


1.4 Paths and Reachability in Graphs

1.4.1 Paths

A path is a sequence of vertices $v_1, v_2, \ldots, v_k$ connected by edges (v_i, v_{i+1}) for $i = 1, 2, \ldots, k$.

Example 1.4.1. .



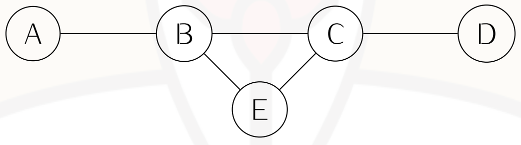
In the above graph, there are two paths from vertex A to vertex D: A → B → C → D, and A → C → D. Similarly, there is only one path from vertex B to vertex E: B → C → D → E and there are no paths to vertex *F*.

1.4.2 Reachability

In a directed graph, a vertex *u* is reachable from a vertex *v* if there is a path from *v* to *u*. In the graph shown in Example 1.4.1., vertex D is reachable from vertex A, but vertex A is not reachable from vertex D. Similarly, vertex E is reachable from vertex B, but vertex B is not reachable from vertex E.

Paths and reachability in undirected graphs

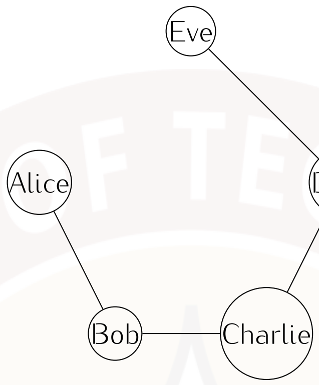
Consider an undirected graph given below:



In the above graph,

- there are two paths from vertex A to vertex D: A → B → C → D, and A → B → E → C → D.
- every vertex is reachable from every other vertex, since the graph is connected.

Example 1.4.2. Consider the problem of determining whether two people in a social network are connected by a path of mutual friends. We can represent the social network as a graph, where each vertex represents a person and each edge represents a connection between two people.



In this example, we have a social network with five people (Alice, Bob, Charlie, Dave, and Eve). We can use reachability to determine whether two people in the network are connected by a path of mutual friends. For example, suppose we want to determine whether Alice and Eve are connected. We can start at Alice and perform a depth-first search of the graph to find all the people she is connected to, and then repeat the process for each of those people until we either find Eve or exhaust all possible paths.

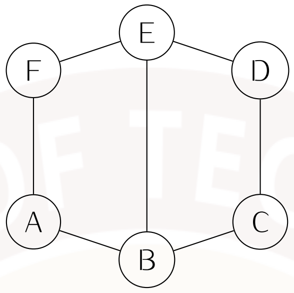
In this case, we can see that Alice is connected to Bob, who is connected to Charlie, who is connected to Dave, who is connected to Eve. Therefore, Alice and Eve are connected by a path of mutual friends in the social network.

1.5 More on graphs

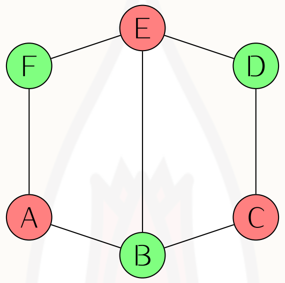
1.5.1 Graph coloring

In graph theory, *graph coloring* is the assignment of colors to the vertices of a graph such that no two adjacent vertices have the same color. Mathematically, graph colouring is a function $c : V \longrightarrow C$ such that $(u, v) \in E \implies c(u) \neq c(v)$, where V is the set of vertices, E is the set of edges in the graph, and C is the set of colours. The smallest number of colors required to color a graph is called its *chromatic number*. The chromatic number of a graph is an important graph invariant, and finding the chromatic number of a graph is a well-studied problem.

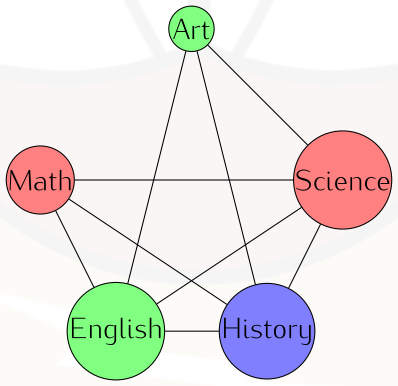
Example 1.5.1. Find the minimum number of colours required to colour the following graph.



The minimum number of colours required to colour the above graph is 2



Example 1.5.2. Consider the problem of scheduling classes at a university. Each class has a set of time slots in which it can be scheduled, and each time slot has a maximum number of classes that can be held during that time slot. We can represent this problem as a graph, where each vertex represents a class and each edge represents a conflict between two classes that cannot be scheduled in the same time slot.



In this example, we have five classes (Math, English, History, Science, and Art) that need to be scheduled. Each vertex represents a class, and each edge represents a conflict between two classes that cannot be scheduled in the same time slot. We can use vertex coloring to find a valid schedule, subject to the constraint that each time slot can hold a maximum of two classes.

One possible valid schedule is:

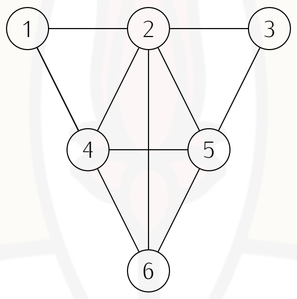
Time Slot	Classes
1	Math, History
2	English, Science
3	Art

In this schedule, we have assigned three time slots (colors) to the five classes (vertices), such that no two adjacent vertices (classes with conflicts) have the same color (time slot).

1.5.2 Vertex cover

First, we define the vertex cover problem as follows: Given a graph $G = (V, E)$, a vertex cover is a set of vertices $V' \subseteq V$ such that every edge in E is incident to at least one vertex in V'. The vertex cover problem is to find a minimum-size vertex cover for G.

Here's an example of a vertex cover for a graph:



In this graph, a vertex cover of size 3 is $\{2, 4, 5\}$, since these vertices cover all the edges. It's easy to see that no vertex cover can be smaller than size 3.

We can represent the vertex cover problem mathematically using integer programming. Let x_i be a binary variable that is 1 if vertex i is in the vertex cover and 0 otherwise. Then we can write the following integer program:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{|V|} x_i \\
\text{subject to} \quad & x_i + x_j \geq 1 \quad \text{for all } (i, j) \in E \\
& x_i \in 0, 1 \quad \text{for all } i \in V
\end{aligned}$$

The first constraint ensures that every edge is covered by at least one vertex in the cover, and the second constraint ensures that x_i is a binary variable.

This integer program can be solved using a variety of methods, including branch-and-bound, cutting plane algorithms, and integer programming solvers.

Example 1.5.3. One application of the vertex cover problem is in wireless sensor networks. In a wireless sensor network, a set of sensor nodes are deployed to monitor a physical environment, such as a building or a bridge. These nodes communicate wirelessly with each other and with a central base station, which collects and processes the data.

One challenge in wireless sensor networks is to conserve energy, since the nodes are typically battery-powered and have limited energy resources. One way to conserve energy is to activate only a subset of the nodes at any given time, while the rest of the nodes remain in a sleep mode.

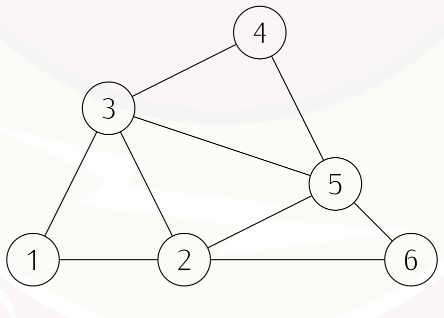
The vertex cover problem can be used to find an energy-efficient set of nodes to activate. Specifically, we can represent the network as a graph, where the nodes are vertices and the communication links are edges. Then, a vertex cover corresponds to a set of nodes to activate, such that every edge is covered by at least one active node.

By finding a minimum-size vertex cover for the graph, we can identify the smallest set of nodes that need to be activated to ensure that the network is connected and all data is transmitted. This reduces the overall energy consumption of the network and prolongs the lifetime of the sensor nodes.

1.5.3 Independent set

First, we define the independent set problem as follows:
Given a graph $G = (V, E)$, an independent set is a set of vertices $S \subseteq V$ such that no two vertices in S are adjacent. The independent set problem is to find a maximum-size independent set for G.

Here's an example of an independent set for a graph:



In this graph, one possible independent set is $\{1, 4, 6\}$, since no two of these vertices are adjacent. This is a maximum-size independent set, since there are no more than three vertices that are not adjacent to each other.

Example 1.5.4. The maximum independent set problem can be used to model a variety of real-world situations, including scheduling problems, task allocation, and social network analysis. One example of a real-life application of the maximum independent set problem is in computer network security.

Computer networks are vulnerable to attacks from hackers and other malicious actors who attempt to gain unauthorized access to sensitive information. One way to defend against these attacks is to identify and isolate compromised devices, such as computers or servers, before they can be used to launch further attacks. This process is known as "isolation and containment."

The maximum independent set problem can be used to identify a maximum-size set of devices to isolate and contain, such that no two devices in the set are directly connected. In this context, a device is represented as a vertex in a graph, and a connection between two devices is represented as an edge.

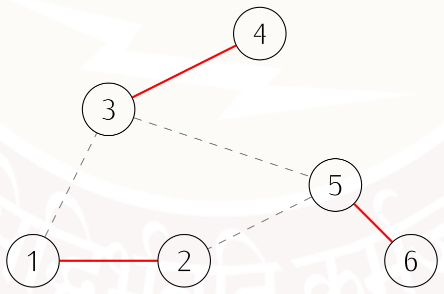
By finding a maximum independent set for the graph, we can identify the largest possible set of compromised devices that can be isolated and contained. This helps to minimize the impact of a security breach and prevent further attacks.

Mathematically, the maximum independent set problem can be used to identify a maximum-size set of compromised devices to isolate and contain. Given a graph $G = (V, E)$ representing the network, an independent set is a set of vertices $S \subseteq V$ such that no two vertices in S are adjacent. By finding a maximum independent set for the graph, we can identify the largest possible set of compromised devices that can be isolated and contained, thus minimizing the impact of a security breach and preventing further attacks.

1.5.4 Matching

A matching in a graph $G = (V, E)$ is a set of edges $M \subseteq E$ such that no two edges in M share an endpoint. In other words, a matching is a set of edges that do not "overlap" with each other.

The matching problem is to find a maximum-size matching for G, that is, a matching that contains as many edges as possible. Here is an example of a maximum-size matching in the same graph: Here is an example of a matching in a graph:



In this graph, the set of edges $\{(1,2),(3,4),(5,6)\}$ forms a maximum-size matching, since no other matching can contain more than three edges.

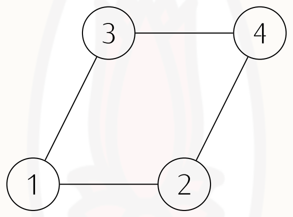
In addition to finding maximum-size matchings, the matching problem has many real-world applications, such as scheduling problems, pairing problems, and network flow problems.

1.6 Representing graphs

1.6.1 Adjacency matrix

The adjacency matrix of a graph $G = (V, E)$ is a matrix A of size $|V| \times |V|$ such that $A_{i,j} = 1$ if there is an edge $(i, j) \in E$, and $A_{i,j} = 0$ otherwise. In other words, the adjacency matrix is a binary matrix that encodes the connectivity of the graph.

Here is an example of an undirected graph, along with its adjacency matrix:

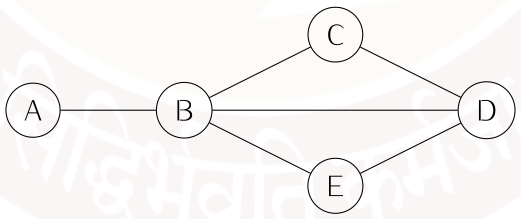


The adjacency matrix for this graph is:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

1.6.2 Adjacency list

An adjacency list is a way to represent a graph as a collection of lists, where each list corresponds to a vertex in the graph. Each list contains the vertices that are adjacent to the corresponding vertex in the graph. In other words, each vertex's adjacency list contains the vertices that it is connected to by an edge.
Here's an example of an undirected graph and its adjacency list:

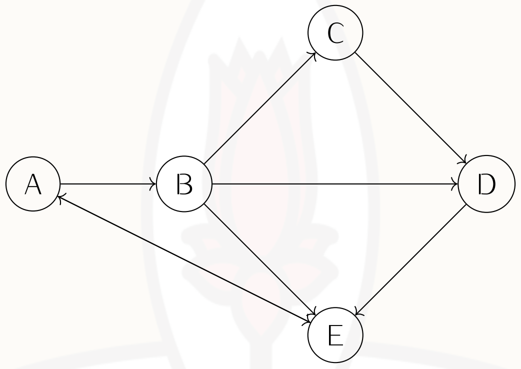


The list of neighbours for each vertex is:

A	{B}
B	{A,C,D,E}
C	{B,D}
D	{B,C,E}
E	{B,D}

In this example, each vertex is represented by a letter (A, B, C, D, E). The edges between vertices are represented by lines connecting the vertices. The adjacency list for this graph shows that vertex A is adjacent to vertex B, vertex B is adjacent to vertices A, C, D, and E, and so on.

Here's another example of a directed graph and its adjacency list:



The list of neighbours for each vertex is:

A	{B,E}
B	{C,D,E}
C	{D}
D	{E}
E	{A}

In this example, the graph is directed, which means that the edges have a direction. The adjacency list for this graph shows that vertex A has edges pointing to vertices B and E, vertex B has edges pointing to vertices C, D, and E, and so on.

Overall, adjacency lists are a useful way to represent graphs because they are compact, easy to implement, and efficient for certain types of graph algorithms.

1.7 Breadth-First Search

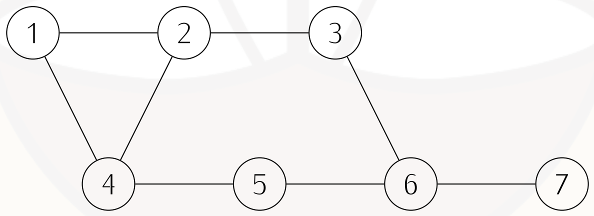
Breadth-first search (BFS) is a graph traversal algorithm that starts at a given vertex and explores the neighboring vertices in a breadth-first manner, meaning that it visits all the vertices at a given distance from the starting vertex before moving on to vertices at a greater distance. BFS can be used to find the shortest path between two vertices in an unweighted graph.

1.7.1 Algorithm

Here's how the BFS algorithm works:

1. Start at a given vertex and mark it as visited.
2. Add the vertex to a queue.
3. While the queue is not empty, remove the first vertex from the queue and explore its neighbors.
4. For each unvisited neighbor, mark it as visited, add it to the queue, and record its distance from the starting vertex.
5. Repeat steps 3 and 4 until the queue is empty.

Example 1.7.1. Let's illustrate how BFS works on the following undirected graph:



Let's start BFS from vertex 1. We mark vertex 1 as visited and add it to the queue.

To explore queue					
1					

The queue now contains only vertex 1. We then remove vertex 1 from the queue and explore its neighbors. Vertex 2 and vertex 4 are neighbors of vertex 1, so we mark them as visited, add them to the queue. The queue now contains vertices 2 and 4.

To explore queue					
2	4				

Next, we remove vertex 2 from the queue and explore its neighbors. Vertex 1 and vertex 3 are neighbors of vertex 2, but vertex 1 has already been visited, so we don't add it to the queue. We mark vertex 3 as visited, add it to the queue. The queue now contains vertex 4 and vertex 3.

To explore queue					
4	3				

We then remove vertex 4 from the queue and explore its neighbors. Vertex 1 and vertex 5 are neighbors of vertex 4, but vertex 1 has already been visited and vertex 5 has not, so we mark vertex 5 as visited, add it to the queue. The queue now contains vertices 3 and 5.

To explore queue					
3	5				

Next, we remove vertex 3 from the queue and explore its neighbors. Vertex 2 and vertex 6 are neighbors of vertex 3, but vertex 2 has already been visited and vertex 6 has not, so we mark vertex 6 as visited, add it to the queue. The queue now contains vertices 5 and 6.

To explore queue					
5	6				

We then remove vertex 5 from the queue and explore its neighbors. Vertex 4 and vertex 6 are neighbors of vertex 5, but vertex 4 has already been visited and vertex 6 has already been added to the queue, so we don't add any new vertices to the queue.

To explore queue					
6					

Finally, we remove vertex 6 from the queue and explore its neighbors. Vertex 3 and vertex 7 are neighbors of vertex 6, but vertex 3 has already been visited and vertex 7 has not, so we mark vertex 7 as visited, add it to the queue. The queue now contains only vertex 7.

To explore queue					
7					

We finally remove vertex 7. Since the queue is now empty, we have explored all the vertices that are reachable from vertex 1. The order in which the vertices were added to the queue during the BFS traversal is:

$$1, 2, 4, 3, 5, 6, 7$$

The distances (in terms of number of edges) of each vertex from vertex 1 are:

$$\begin{aligned}
\text{distance}(1) &= 0\\
\text{distance}(2) &= 1\\
\text{distance}(3) &= 2\\
\text{distance}(4) &= 1\\
\text{distance}(5) &= 2\\
\text{distance}(6) &= 3\\
\text{distance}(7) &= 4
\end{aligned}$$

This BFS traversal can be used to find the shortest path from vertex 1 to any other vertex in the graph. For example, the shortest path from vertex 1 to vertex 7 is:

$$1 \longrightarrow 4 \longrightarrow 5 \longrightarrow 6 \longrightarrow 7$$

1.8 Depth First Search (DFS)

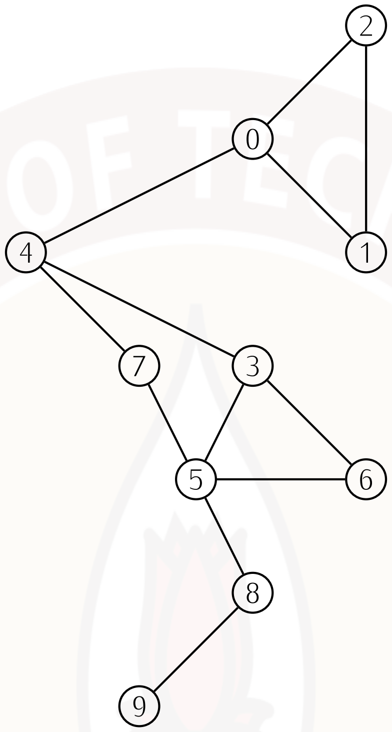
DFS is another algorithm for exploring graphs. Unlike BFS, DFS traverses the graph by exploring as far as possible along each branch before backtracking.

1.8.1 Algorithm

The DFS algorithm can be described as follows:

1. Start from a vertex i, visit an unexplored neighbour j.
2. Suspend the exploration of i and explore j instead.
3. Continue till you reach a vertex with no unexplored neighbours.
4. Backtrack to nearest suspended vertex that still has an unexplored neighbour.
5. Suspended vertices are stored in a stack
    (a) Last in, first out
    (b) Most recently suspended is checked first

Example 1.8.1. Let's apply the DFS algorithm to the following undirected graph:



Suppose we start at vertex 4. We mark vertex 4 as visited. Now, suspend 4 and then explore its unvisited neighbour Vertex 0.

Stack of suspended vertices							
4							

Now, suspend vertex 0 and add vertex 0 to the stack. Explore its unvisited neighbor Vertex 1.

Stack of suspended vertices							
4	0						

Then, suspend vertex 1 and add vertex 1 to the stack. Explore its unvisited neighbor Vertex 2.

Stack of suspended vertices							
4	0	1					

Now, there are no unvisited neighbors to vertex 2 so backtrack to vertex 1. Similarly, there are no unvisited neighbors to vertex 1 so backtrack to vertex 0.

Stack of suspended vertices							
4	0						

If we follow the same process, then the following shows the vertices that add to the stack and get removed from the stack.

-

We mark vertex 2 as visited and then recursively apply steps 1–3 to its unvisited neighbors. Vertex 3 is the only unvisited neighbor of vertex 2, so we apply steps 1–3 to vertex 3.

We mark vertex 3 as visited and then recursively apply steps 1–3 to its unvisited neighbors. Vertex 4 is the only unvisited neighbor of vertex 3, so we apply steps 1–3 to vertex 4.

We mark vertex 4 as visited and then recursively apply steps 1–3 to its unvisited neighbors. There are no unvisited neighbors of vertex 4, so we backtrack to vertex 3.

We have now explored all the unvisited neighbors of vertex 3, so we backtrack to vertex 2.

We have already explored all the unvisited neighbors of vertex 2, so we backtrack to vertex 1.

We have already explored all the unvisited neighbors of vertex 1, so we are done. The order in which vertices were visited during the DFS traversal is:

$$1, 2, 3, 4, 5, 6$$

Note that DFS has no notion of distance or path length like BFS

1.8.2 Applications

DFS has a variety of applications in computer science and beyond. Some common uses include:

- Topological sorting: DFS can be used to sort the nodes of a directed acyclic graph (DAG) such that for every directed edge (u, v), node u comes before node v in the sorted list.
- Strongly connected components: DFS can be used to find the strongly connected components of a directed graph. A strongly connected component is a subset of the nodes of the graph such that there is a path from any node in the subset to any other node in the subset.
- Maze solving: DFS can be used to solve mazes or other types of puzzles that can be represented as graphs.
- Network traversal: DFS can be used to traverse networks of various kinds, such as file systems or social networks.

1.8.3 Pseudocode

Here is the pseudocode for the DFS algorithm:

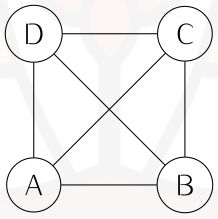
```
dfs(graph G, vertex v, set visited):
add v to visited set
for each neighbor w of v:
if w is not in visited:
dfs(G, w, visited)
```

This pseudocode assumes that the graph is represented as an adjacency list and that the visited nodes are stored in a set.

1.9 Degree of a vertex in an undirected graph

The degree of a vertex in an undirected graph is the number of edges incident to that vertex. In other words, it's the number of edges that are connected to that vertex.

Example 1.9.1. *Complete graph*
A complete graph is an undirected graph in which every pair of distinct vertices is connected by a unique edge. Let's consider a complete graph with 4 vertices:



Its corresponding adjacency matrix is

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

To compute the degree of each vertex, we simply count the number of 1's in each row (or column) of the adjacency matrix.
For example, the degree of vertex 1 is the sum of the entries in the first row of the matrix, which is 3.
Similarly, the degree of vertex 2 is the sum of the entries in the second row of the matrix, which is also 3, and so on.
Therefore, the degree sequence of the complete graph is (3, 3, 3, 3), which means that all vertices have the same degree.

Example 1.9.2. *Bipartite graph*
A bipartite graph is an undirected graph whose vertices can be divided into two disjoint sets such that no two vertices within the same set are adjacent. Let's consider an adjacency matrix of a bipartite graph with 4 vertices in each set:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

To compute the degree of each vertex, we simply count the number of 1's in each row (or column) of the adjacency matrix.
For example, the degree of vertex 1 is the sum of the entries in the first row of the matrix, which is 2.
Similarly, the degree of vertex 5 is the sum of the entries in the fifth row of the matrix, which is 3, and the degree of vertex 6 is the sum of the entries in the sixth row of the matrix also 3.
Therefore, the degree sequence of the bipartite graph is (3, 3, 3, 3, 2, 2, 2, 2).

1.10 Indegrees and Outdegrees of a directed graph

In a directed graph, each edge is associated with a direction. In other words, the edge connects one vertex (the source) to another vertex (the target). Let's consider an adjacency matrix of a directed graph with 4 vertices:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

To compute the degree of each vertex in a directed graph, we need to consider both incoming edges and outgoing edges. The in-degree of a vertex is the number of edges that are directed towards that vertex, and the out-degree of a vertex is the number of edges that are directed away from that vertex.

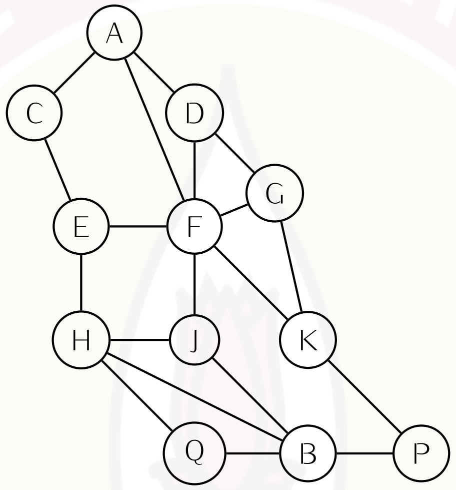
For example, the in-degree of vertex 1 is the sum of the entries in the first column of the matrix, which is 1. The out-degree of vertex 1 is the sum of the entries in the first row of the matrix, which is also 1. Similarly, the in-degree of vertex 3 is the sum of the entries in the third column of the matrix, which is 1, and the out-degree of vertex 3 is the sum of the entries in the third row of the matrix, which is 1 as well.

Therefore, the in-degree sequence of the directed graph is (1, 1, 1, 0), and the out-degree sequence is (1, 2, 1, 0). Note that the sum of the in-degree sequence and the sum of the out-degree sequence are equal since every edge contributes to both an in-degree and an out-degree.

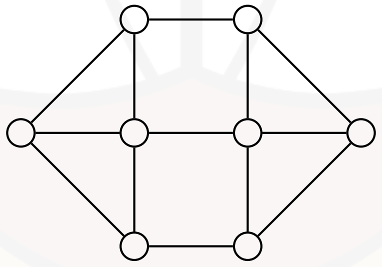
In conclusion, we can compute the degree of vertices in an undirected graph by counting the number of edges incident to each vertex, which can be done by looking at the adjacency matrix. In a directed graph, we need to consider both incoming edges and outgoing edges to compute the in-degree and out-degree of each vertex.

1.11 Problems

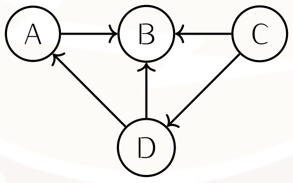
Question 1. The person 'A' is a job applicant while another person 'B' is a hiring manager at a company. Although both of them are members of the website 'Linked-In', 'A' is not directly connected to 'B' via the website. The networks of 'A' and 'B' are shown in the graph below. Find out the shortest path of connecting 'A' and 'B'.



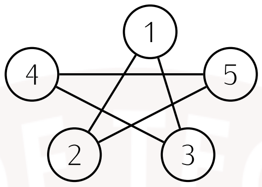
Question 2. Calculate the maximum number of colors that may be required to color the graph below so that no two adjacent regions have the same color.



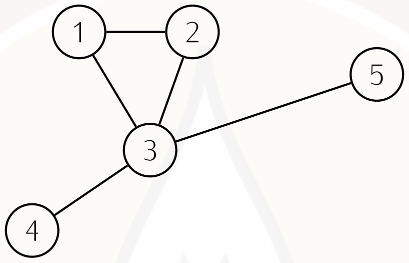
Question 3. Find the adjacency matrix for the following graph.



Question 4. Find the adjacency matrix of the following graph.

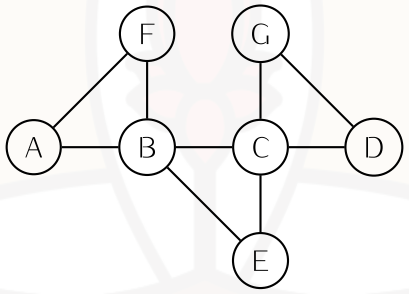


Question 5. Find a vertex cover for the graph given below.



NOTE: Multiple vertex cover sets may be possible for the same graph.

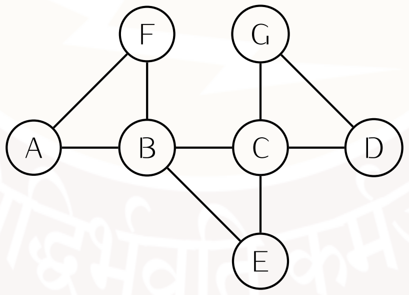
Question 6. Draw the BFS tree for the following graph starting from the vertex 'E'.



Question 7. The BFS algorithm follows the structure of a '*queue*'. The '*queue*' is a –

Question 8. The 'Depth First Search' or DFS algorithm follows the structure of a '*stack*'. The '*stack*' is a–

Question 9. Draw the DFS tree for the following graph starting from the vertex 'E'.

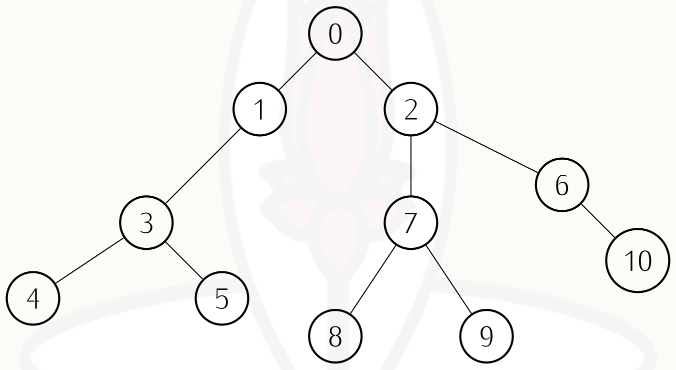


Question 10. Consider an undirected planar graph with 169 vertices. Which of the following is correct?

1. Number of edges will be less than 14196
2. Number of edges will be less than 14281
3. Number of edges will be less than 169
4. Number of edges will be less than 28382

Question 11. If G is a graph with n vertices and does not contain a triangle, then the maximum number of edges in G can be

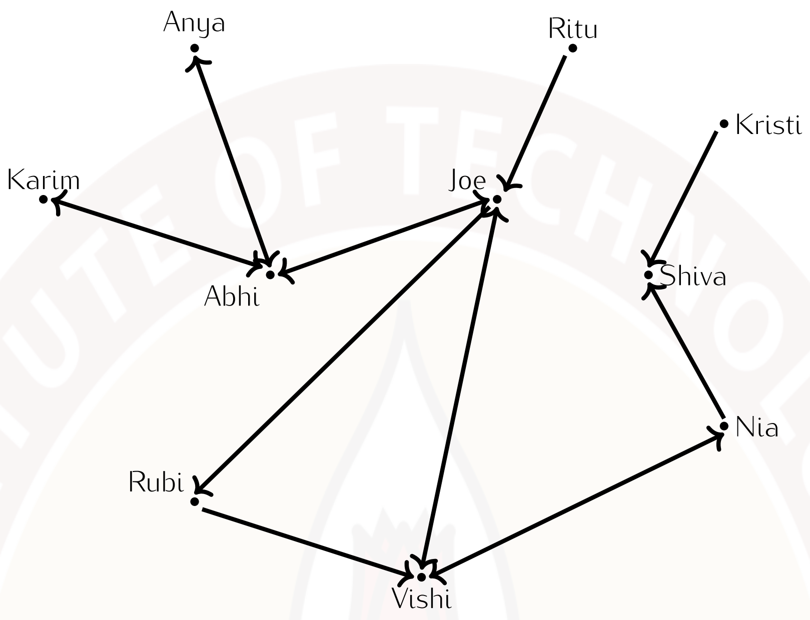
Question 12. Suppose we obtain the following DFS tree rooted at node 0 for an undirected graph with vertices $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.



Which of the following cannot be an edge in the original graph?

1. $(1, 4)$
2. $(0, 4)$
3. $(7, 10)$
4. $(2, 9)$

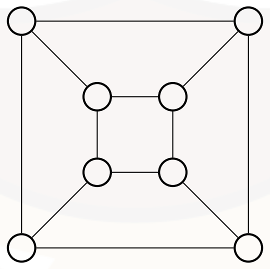
Use the following information for questions 13-14:
Ten friends in a college decided to have a night party at the home of one of them. Unfortunately at D-day, the government closes many of the routes of the city. The graph below shows the location of their homes and the open routes as $G = (V, E)$, where V is the set of nodes and E is the edges representing the open routes.



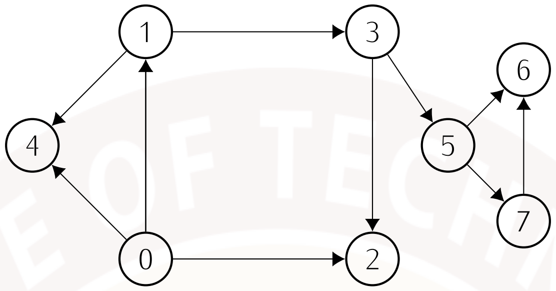
Question 13. The possible place for the party is _

Question 14. If Joe wants to have the party at his home, then at most how many members can join the party?

Question 15. Suppose G be a graph (shown in the below figure). Let V be the set of vertices of G, V_i be the maximum independent set and V_c be the minimum vertex cover. Find the cardinalities of the sets V_i and V_c.



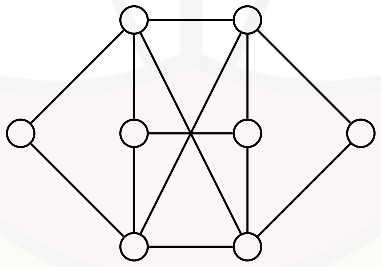
Question 16. Consider the graph given below.



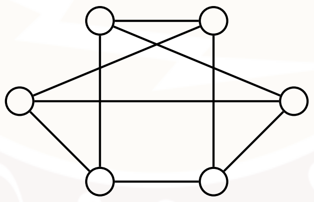
Suppose we perform BFS/DFS so that when we visit a vertex, we explore its unvisited neighbours in a random order. Which of the following options are correct?

1. If we perform Breadth First Search at node 0, then one of the possible order in which the nodes will be visited is 01423567.
2. If we perform Depth First Search at node 0, then one of the possible order in which the nodes will be visited is 04123576
3. If we perform Breadth First Search at node 0, then one of the possible order in which the nodes will be visited is 01423576.
4. If we perform Depth First Search at node 0, then one of the possible order in which the nodes will be visited is 04132567.

Question 17. The cardinality of the maximum independent set of the graph given below is



Question 18. The minimum coloring of the below graph is



Question 19. Let G be an undirected graph with 8 vertices and all vertices have degree 4. How many edges are there in the graph G?

2. DAGs, Topological sorting and Longest path

"What we know is a drop, what we don't know is an ocean."

— Isaac Newton

let us look at a general problem, where we have some tasks to do and there are some dependencies between the tasks. As an example, suppose there is a startup, which is trying to move into some new office space. So, the startup needs to set up this office before it can move in.
This new office space is completely unfinished so we need to do a number of things like we need to lay the floor tiles, we need to plaster and paint the walls, we also need to lay pipes, conduits in order to take wires from here to there. So, there are wires of two types there are electrical wires and there are networking cables for computers, there are also telephone cables and so on, and these cannot go in the same conduit because they interfere. So, we will have separate conduits for electrical wires and separate conduits for telecom equipment, then you have to put in the wiring and after you finish the wiring of the electrical things, you have to put in the fittings, you have to put in the lights, the fans, the switches, and so on.
Now, these are all activities which need to be done, but clearly they cannot be done in arbitrary order. So for instance, we have these constraints:

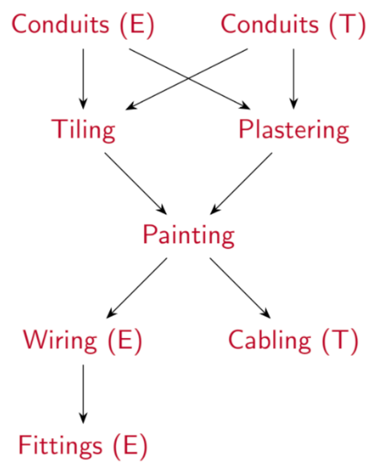
(i) Lay conduits before tiles and plastering.

(ii) Lay tiles, plaster wall before painting.

(iii) Finish painting before any cabling/wiring work.

(iv) Electrical wiring before installing fittings.

We take this scenario and try to draw the graph such that the vertices corresponding to the tasks and directed edge from u to v if task u has to be completed before task v. Below graph shows the corresponding graph.



Typical questions that can be asked are

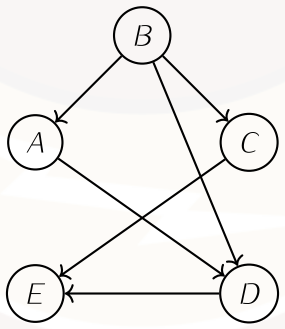
1. Schedule the tasks respecting the dependencies.
2. How long will it take to complete all the tasks?

In this chapter we will try to answer these questions using the concepts of DAGs, Topological sorting and Longest path.

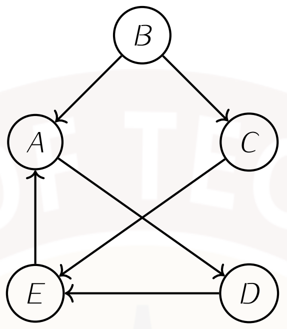
2.1 Directed acyclic graph

Definition 2.1.1. A directed graph $G = (V, E)$ without directed cycles is called as directed acyclic graph, in short it is also called as 'DAG'.

Example 2.1.1. DAG with five vertices $(A, B, C, D,$ and $E)$ and six directed edges:



Example 2.1.2. A graph with five vertices $(A, B, C, D,$ and $E)$ and six directed edges:



Question 20. The graph given in Example 2.1.2. is not a DAG. Why???

2.1.1 Real life examples

There are many real life situations where these dependencies arise. Few examples are listed below.

1. Pre-requisites between courses for completing a degree
2. Recipe for cooking
3. Construction projects

2.1.2 Facts about DAGs

1. Directed acyclic graphs does not have cycles.
2. Every DAG has a vertex with indegree 0.
3. If we remove a vertex u which has indegree 0 and its corresponding edges, then the resulting graph will again be a DAG.
4. Topological sorting and longest path can be solved on DAGs

2.1.3 Problems to be solved on DAGs

We are going to find the topological sorting and Longest path in a DAG which will give us the task dependencies and how long will it take to complete all the tasks, respectively.

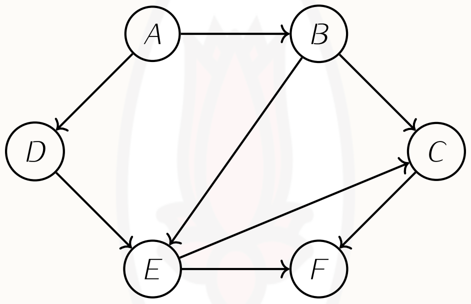
2.2 Topological sorting

Let $G = (V, E)$ be a directed graph with no cycles. Topological sorting is basically ordering all the vertices in a sequence such that for all $(i, j) \in E$, vertex i appears before vertex j in the sequence.

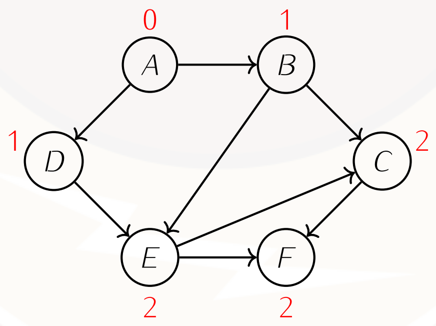
Topological sorting algorithm

1. Write the indegrees of all the vertices in the DAG.
2. Find a vertex 'u' from the DAG that has indegree 0.
3. Add the vertex 'u' to topological sequence.
4. Remove vertex 'u' from the DAG and update the indegrees of each of the remaining vertices.
5. Repeat the process from step 2 till the last vertex is added to the topological sequence.

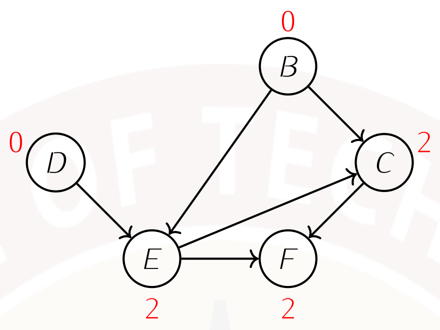
Example 2.2.1. Find the topological sequence of the below-given graph.



Solution:
The given graph is a DAG...think why?
First we will write the indegrees of all the vertices in the graph.

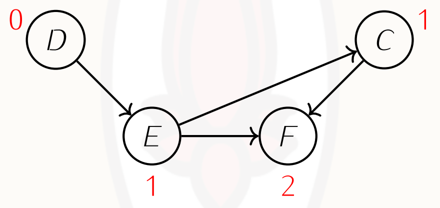


Now, vertex *A* has indegree 0. So we will add vertex *A* to the topological sequence. Later, remove the vertex *A* from the DAG and update the indegrees of each of the remaining vertices. The resulting graph and the topological sequence will be as follows,



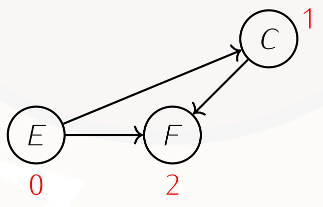
Topological sequence(TP): A

Again, vertex B and vertex D have indegree 0. So we can add any of the vertexes to the topological sequence if the order is not mentioned. Without loss of generality, let us remove the vertex B from the DAG and update the indegrees of each of the remaining vertices. The resulting graph and the topological sequence will be as follows,



Topological sequence(TP): A, B

Next, vertex D has indegree 0. So we can add it to the topological sequence. Removing vertex D from the DAG and updating the indegrees of the remaining vertices, we get:



Topological sequence(TP): A, B, D

If we repeat the process until all the vertices are removed from the DAG, then the final topological sequence will be A, B, D, E, C, F.
Another possible topological sequence will be A, D, B, E, C, F...think how is this possible?

2.3 Longest path in a DAG

The Longest Path in a Directed Acyclic Graph (DAG) problem is to find the longest path in a DAG from a given source vertex to a destination vertex. When the DAG has no edge weights, we can simply count the number of edges in the longest path.

The algorithm for finding the longest path in a DAG with no edge weights is similar to the topological sort algorithm. The difference is that instead of storing the minimum distance found so far from the source vertex to each vertex, we store the maximum length of the path found so far.
Here is the pseudocode for the algorithm:
1. Topologically sort the vertices of the DAG.
2. Initialize the length of the path to(i) for all the vertices to 0.
3. For each vertex u in the topological order:
a. For each edge (u, v) in the graph:
i. If the length of the path to(v) through u is greater than the current maximum length of the path to(v), update the length of the path to(v).
4. The maximum length of the path to the destination vertex is the longest path in the DAG.

Example:

Consider the following DAG:

2.4 Transitive Closure

In graph theory, the transitive closure of a directed graph is a graph that represents the reachability between all pairs of vertices. It determines whether there is a directed path from one vertex to another in the graph. The transitive closure is often denoted as $TC(G)$, where G is the original graph.

2.4.1 Definition

Let $G = (V, E)$ be a directed graph with vertex set V and edge set E. The transitive closure $TC(G)$ is defined as follows:

$$TC(G) = (V, E')$$

where E' contains an edge (u, v) if and only if there is a directed path from vertex u to vertex v in G. In other words, E' contains all the edges necessary to ensure transitivity.

Example 2.4.1. Consider the following directed graph G:

$$\begin{tikzcd}
A \arrow[r] & B \arrow[d] & \\
& C \arrow[ul] & D \arrow[ul]
\end{tikzcd}$$

To find the transitive closure $TC(G)$, we need to determine all pairs of vertices that are reachable from each other. We can do this by iteratively checking if there is a path of length 1, 2, 3, and so on, between the vertices.

- For length 1 paths, we have the edges (A, B), (B, C),(C, A), and (D, B).
- For length 2 paths, we have the edge (A, C), which can be obtained by combining the edges (A, B) and (B, C) and we have the edge (C, B), which can be obtained by combining the edges (C, A) and (A, B).

Thus, the transitive closure $TC(G)$ of graph G is:

$$\begin{tikzcd}
A \arrow[r] \arrow[dr] & B \arrow[d, bend left] & \\
& C \arrow[ul, bend left] \arrow[u] & D \arrow[ul]
\end{tikzcd}$$

The transitive closure $TC(G)$ includes the additional edges (A, C) and (C, B), which represent the transitive relationships between the vertices.

2.4.2 Properties

The transitive closure $TC(G)$ of a directed graph G possesses the following properties:

- **Transitivity**: If there is a path from vertex u to vertex v, and a path from vertex v to vertex w, then there is an edge from vertex u to vertex w in $TC(G)$.
- **Reachability**: For any pair of vertices (u, v), there is an edge from vertex u to vertex v in $TC(G)$ if and only if there is a directed path from vertex u to vertex v in the original graph G.

These properties ensure that the transitive closure accurately represents the reachability relationships between vertices.

2.4.3 Applications

The concept of transitive closure finds applications in various fields, including:

- **Social networks**: In social network analysis, transitive closure can be used to determine the indirect connections between individuals based on their direct connections.
- **Database systems**: Transitive closure can be utilized in database systems to compute the transitive relationships between records in a relational database, enabling efficient querying and analysis.
- **Software engineering**: Transitive closure is employed in program analysis and software verification to determine the flow of data and control dependencies between program statements.

The ability to efficiently compute the transitive closure of a directed graph has significant implications in these domains.

2.4.4 Conclusion

The transitive closure of a directed graph is a graph that represents the reachability relationships between all pairs of vertices. It can be computed by iteratively checking for paths of increasing lengths between vertices. The transitive closure exhibits properties such as reflexivity and transitivity, ensuring that it accurately captures the reachability relationships in the original graph. Its applications span diverse areas such as social networks, database systems, and software engineering, contributing to the analysis and understanding of complex systems.

2.5 Matrix Multiplication

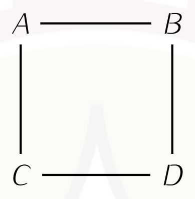
In graph theory, matrix multiplication is a powerful tool used to analyze graphs. Matrices can represent various properties of a graph, such as adjacency, reachability, and paths. In this section, we will explore the process of matrix multiplication and its applications in graph theory.

2.5.1 Matrix representation of a Graph

Let's start by considering an undirected graph G with n vertices. We can represent G using an $n \times n$ matrix called the *adjacency matrix*. The adjacency matrix A of G is defined as follows:

$$A_{ij} = \begin{cases} 1, & \text{if there is an edge between vertices } i \text{ and } j, \\ 0, & \text{otherwise.} \end{cases}$$

For example, consider the following graph G:



The adjacency matrix A for this graph is:

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

2.5.2 Matrix Multiplication

Matrix multiplication allows us to combine two matrices to obtain a third matrix that represents a specific property of interest. In graph theory, we often use matrix multiplication to compute the *reachability matrix*.

Given two matrices A and B, their product $C = AB$ is calculated using the following formula:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

In other words, the element C_{ij} in the resulting matrix C is obtained by taking the dot product of the i-th row of matrix A and the j-th column of matrix B.

2.5.3 Computing Reachability Matrix

The reachability matrix of a graph G determines whether there is a path between any two vertices. By raising the adjacency matrix to the power of k, we can obtain the reachability matrix A^k, where $A^k[i][j] = 1$ if there exists a path of length k between vertices i and j, and $A^k[i][j] = 0$ otherwise.

For example, consider the graph G and its adjacency matrix A from before. To compute the reachability matrix A^2, we can perform matrix multiplication as follows:

$$A^2 = A \times A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

2.5.4 Computing Transitive closure

By performing matrix multiplication with a modified adjacency matrix, we can obtain the transitive closure A', where $A'[i][j] = 1$ if there is a path from vertex i to vertex j or else $A'[i][j] = 0$.

To compute the Matrix for the Transitive closure of a graph, we modify the adjacency matrix A as follows:

$$A' = A + A^2 + A^3 + \ldots + A^n$$

where n is the number of vertices in the graph.

For example, let's compute the transitive closure matrix for the graph G with adjacency matrix A:

$$A' = A + A^2 + A^3 + A^4$$

$$A' = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

2.5.5 Applications

Matrix multiplication has various applications in graph theory, including:

- **Finding paths**: As shown earlier, the reachability matrix obtained through matrix multiplication can be used to determine the existence and length of paths between vertices in a graph.
- **Connectivity**: By raising the adjacency matrix to power and examining the resulting matrix, we can determine if a graph is strongly connected (where there is a path between any two vertices) or weakly connected (where there may be unreachable vertices).

- **Graph operations**: Matrix multiplication can be used to perform various graph operations, such as computing the union, intersection, and difference of graphs.
- **Graph algorithms**: Matrix multiplication is a fundamental component of graph algorithms, such as the Floyd-Warshall algorithm for all-pairs shortest paths.

Matrix multiplication provides a powerful framework for analyzing and manipulating graphs, enabling the development of efficient algorithms and solutions for a wide range of graph-related problems.

2.5.6 Conclusion

Through matrix multiplication, we can compute the reachability matrix to determine the existence of paths between vertices, and the path matrix to find the shortest paths in a graph. These computations have applications in connectivity analysis, graph operations, and graph algorithms. Understanding and utilizing matrix multiplication in the context of graph theory enhances our ability to analyze and solve complex graph-related problems.

In this section, we have explored the matrix representation of graphs using the adjacency matrix. We have also discussed the process of matrix multiplication and its applications in graph theory, specifically in computing the reachability matrix and the path matrix. These matrices provide valuable insights into the connectivity and shortest paths within a graph.

2.6 Problems

1. Suppose $R = \{(1,3),(3,4),(4,5)\}$ is a relation on the set $\{1,3,4,5,7\}$. Which of the following represents the transitive closure of R?

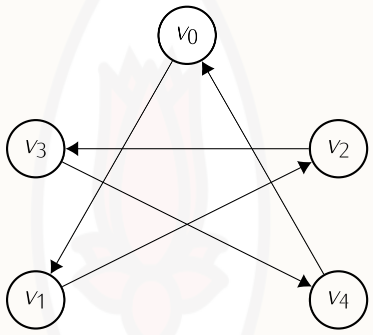
   (a) $\{(1,3),(3,4),(1,4),(4,5),(3,5),(5,1)\}$

   (b) $\{(1,3),(3,4),(4,5),(3,5),(1,5),(4,3)\}$

   (c) $\{(1,3),(3,4),(4,5),(3,5),(1,5),(1,4)\}$

   (d) $\{(1,3),(3,1),(3,4),(4,3),(4,5),(5,4)\}$

2. Consider the graph given below



If A is the adjacency matrix of G, then which of the following represents A^2?

(a)

	v_0	v_1	v_2	v_3	v_4
v_0	0	1	0	0	0
v_1	0	0	1	0	0
v_2	0	0	0	1	0
v_3	0	0	0	0	1
v_4	1	0	0	0	0

(b)

	v_0	v_1	v_2	v_3	v_4
v_0	0	0	1	0	0
v_1	0	0	0	1	0
v_2	0	0	0	0	1
v_3	1	0	0	0	0
v_4	0	1	0	0	0

(c)

	v_0	v_1	v_2	v_3	v_4
v_0	0	1	0	0	1
v_1	1	0	1	0	0
v_2	0	1	0	1	0
v_3	0	0	1	0	1
v_4	1	0	0	1	0

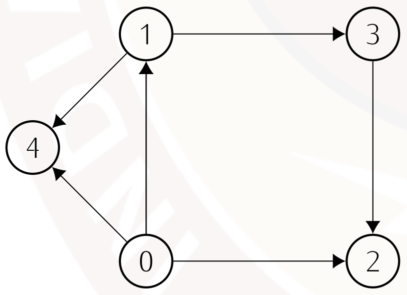
(d)

	v_0	v_1	v_2	v_3	v_4
v_0	0	0	1	1	0
v_1	0	0	0	1	1
v_2	1	0	0	0	1
v_3	1	1	0	0	0
v_4	0	1	1	0	0

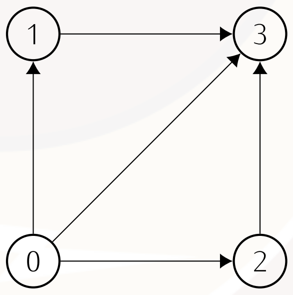
3. Suppose $G = (V, E)$ is a directed graph, where $V = \{1, 2, 3, 4, 6, 7, 12\}$. There is an edge from a to b, that is, $(a, b) \in E$ if and only if $a|b$ (a divides b). Which of the following can be a topological sorting of the graph G?
   (a) $1, 2, 4, 3, 6, 12, 7$
   (b) $1, 7, 2, 4, 3, 6, 12$
   (c) $7, 1, 2, 4, 3, 6, 12$
   (d) $1, 2, 3, 4, 7, 6, 12$

4. Which of the following graphs represent its own transitive closure?

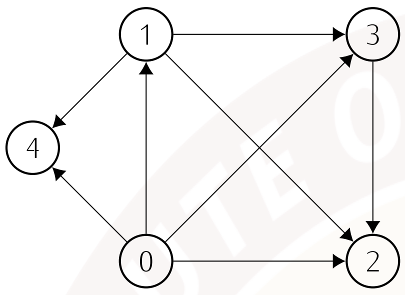
(a)



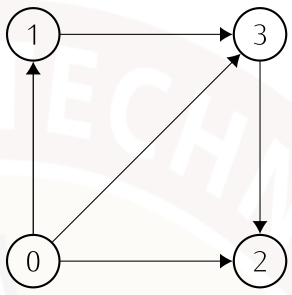
(b)



(c)



(d)



5. Which of the following options is (are) correct?

   (a) If G is a graph with n vertices then length of a path in G is bounded by $n-2$.

   (b) If G is a directed graph, then the sum of the in-degrees of all the vertices is equal to the sum of out-degrees of all the vertices.

   (c) $A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$ can represent the adjacency matrix of an undirected graph G.

   (d) If G is an undirected graph with exactly two vertices of odd degree, then there is a path between those two vertices.

6. Which of the following options is (are) correct? [Ans: b,c]

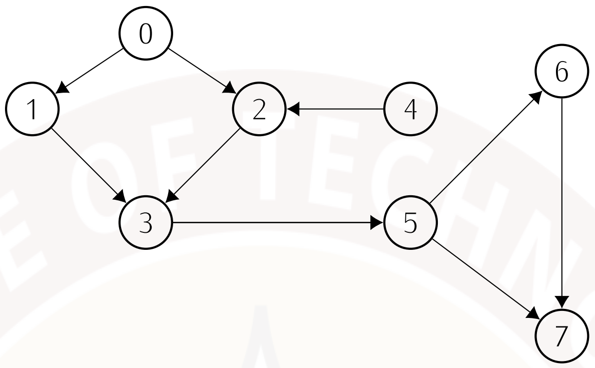
   (a) If $A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$, then $A^2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$.

   (b) If $A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$, then $A^n = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ for all $n > 2$.

   (c) If A is a 3×3 matrix and $I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, then $AI = IA = A$.

   (d) If A is a 3×3 matrix and $I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, then $AI \neq A$.

7. The longest path of the below DAG contains x edges. Find x.


0
1
2
3
4
5
6
7

3. Weighted graphs and shortest path algorithms

"If only I had the Theorems! Then I should find the proofs easily enough"

— Bernhard Riemann

3.1 Weighted Graph

Definition 3.1.1. A *weighted graph* is a graph in which each edge has a weight or cost associated with it.

These weights can represent various information such as distance, time, cost, etc. In a weighted graph, the vertices are represented by nodes, and the edges are represented by lines connecting the nodes. The weight of an edge is usually represented by a number or label near the edge.

Example 3.1.1. An example of a weighted graph is shown below:

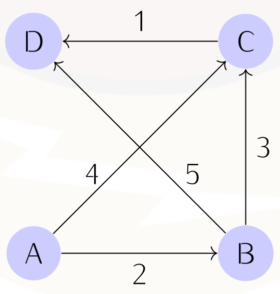


Figure 3.1: Weighted graph G

In the above graph, the weight of the edge between vertex A and B is 2, the weight of the edge between vertex B and C is 3, the weight of the edge between vertex C and D is 1, the weight of the edge between vertex A and C is 4, and the weight of the edge between vertex B and D is 5.

3.2 Dijkstra's Algorithm

Dijkstra's algorithm is a popular shortest path algorithm that can be used to find the shortest path between two nodes in a weighted graph. It was developed by Edsger W. Dijkstra in 1956 and is commonly used in network routing protocols, pathfinding in games, and other applications where finding the shortest path is important.

3.2.1 Algorithm

The algorithm works by maintaining a set of nodes whose shortest distance from the source node is already known. Initially, this set only contains the source node with a distance of 0. At each iteration, the algorithm selects the node with the smallest distance from the source that is not yet included in the set of visited nodes. This node is added to the set, and its neighbors are examined to see if their shortest distance can be updated based on the distance of the newly visited node.

The algorithm continues iterating until all nodes have been visited or the target node has been added to the set. At the end of the algorithm, the shortest distance from the source to each node in the graph will be known.

3.2.2 Pseudocode

Here's the pseudocode for Dijkstra's algorithm: The algorithm takes a graph and a source node as input. It initializes an array dist to hold the shortest distance from the source node to each node in the graph. The distance from the source node to itself is 0, and the distance from the source node to all other nodes is initialized to infinity. A set Q is used to keep track of unvisited nodes.

The while loop continues until all nodes in Q have been visited. At each iteration, the node u with the smallest distance from the source that is still in Q is selected. This node is removed from Q, and its neighbors are examined. If the distance to a neighbor v through u is less than the current distance to v, the distance to v is updated to the new value.

Finally, the function returns the array dist, which now contains the shortest distance from the source node to each node in the graph.

3.2.3 Example

Let's walk through an example to see how Dijkstra's algorithm works. Consider the following graph:

	A	B	C	D
A	0	4	1	∞
B	4	0	2	5
C	1	2	0	3
D	∞	5	3	0

Let's say we want to find the shortest path from node A to node D using Dijkstra's algorithm. We start by initializing the distance from A to all other nodes to infinity, except for A itself which has a distance of 0.

	A	B	C	D
Distance	0	∞	∞	∞

Next, we add A to the set of visited nodes and examine its neighbors. The distance from A to C is 1, which is less than the current distance to C (infinity), so we update the distance to C to 1. The distance to B is 4, which is also less than the current distance to B (infinity), so we update the distance to B to 4.

	A	B	C	D
Distance	0	4	1	∞
Visited	A			

Next, we select the node with the smallest distance that is not yet in the set of visited nodes. In this case, that is C. We add C to the set of visited nodes and examine its neighbors. The distance from C to A is 1, which is not less than the current distance to A (0), so we do not update the distance to A. The distance from C to B is 2, which is less than the current distance to B (4), so we update the distance to B to 2. The distance from C to D is 3, which is less than the current distance to D (infinity), so we update the distance to D to 3.

	A	B	C	D
Distance	0	4	1	3
Visited	A	C		

Next, we select the node with the smallest distance that is not yet in the set of visited nodes. In this case, that is B. We add B to the set of visited nodes and examine its neighbors. The distance from B to A is 4, which is not less than the current distance to A (0), so we do not update the distance to A. The distance from

B to C is 2, which is not less than the current distance to C (1), so we do not update the distance to C. The distance from B to D is 5, which is less than the current distance to D (3), so we update the distance to D to 5.

	A	B	C	D
Distance	0	4	1	3
Visited	A	C	B	

Finally, we select the node with the smallest distance that is not yet in the set of visited nodes. In this case, that is D. We add D to the set of visited nodes and since D is the destination node, we can stop the algorithm. The shortest path from A to D has a distance of 3 and the path is A -> C -> D.

Dijkstra's algorithm guarantees that once a node is added to the set of visited nodes, its distance is the shortest possible distance from the starting node. Therefore, we can be sure that when we select the next node to add to the set of visited nodes, its distance is the shortest possible distance from the starting node among all nodes that are not yet in the set of visited nodes.

Dijkstra's algorithm has a time complexity of $O(V^2)$, where V is the number of nodes in the graph. However, with a priority queue data structure to select the node with the smallest distance, the time complexity can be improved to $O(ElogV)$, where E is the number of edges in the graph.

3.3 Bellmann-Ford Algorithm

The Bellman-Ford algorithm is a widely used algorithm for finding the shortest paths between nodes in a weighted graph. It is capable of handling graphs with negative weight edges, unlike Dijkstra's algorithm, which can only handle non-negative weights. The algorithm was developed by Richard Bellman and Lester Ford Jr. in 1958.

3.3.1 Algorithm Description

The Bellman-Ford algorithm iteratively relaxes the edges of the graph in order to find the shortest paths from a source node to all other nodes. The algorithm keeps track of the minimum distance found so far for each node. Initially, the distance to the source node is set to 0, and all other distances are set to infinity.

The algorithm performs a total of $|V| - 1$ iterations, where $|V|$ is the number of vertices in the graph. In each iteration, it examines all edges in the graph and relaxes them if a shorter path is found. Relaxing an edge means updating the distance of the destination node if a shorter path is discovered.

3.3.2 Algorithm:

Let $D(j)$ be the minimum distance known so far to vertex 'j'
(1) Initialize:
• $D(0) = 0$ if $j = 0$; otherwise $D(j) = \infty$
(2) Repeat (n - 1) times
• For every vertex j = 0, 1, 2, ... (n-1) for each edge $(j, k) \in E$.
• $D(k) = minD(k), D(j) + W(j, k)$

After $|V| - 1$ iterations, the algorithm checks for negative weight cycles. If a shorter path is found in the ($|V|$)-th iteration, it indicates the presence of a negative weight cycle. This is because, in a graph with $|V|$ vertices, the shortest path between any two nodes can have at most $|V| - 1$ edges. If a negative weight cycle exists, the algorithm reports it.

Example 3.3.1. Consider the following graph with 5 vertices and 7 edges:

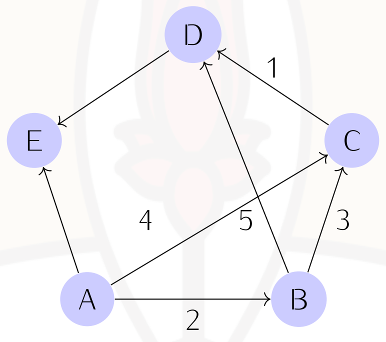


Figure 3.2: Weighted graph G

Let's apply the Bellman-Ford algorithm to find the shortest paths from node A to all other nodes.

Iteration 1:

Step 1: Initialize distances:

dist[A] = 0

dist[B] = ∞

dist[C] = ∞

dist[D] = ∞

dist[E] = ∞

Step 2: Relax edges:

dist[B] = dist[A] + (-1) = 0 + (-1) = -1

dist[C] = dist[A] + 4 = 0 + 4 = 4

Step 3: No updates for D and E

Iteration 2:

Step 1: Initialize distances:

dist[A] = 0

dist[B] = -1

dist[C] = 4

dist[D] = ∞

dist[E] = ∞

Step 2: Relax edges:

dist[C] = dist[B] + 3 = -1 + 3 = 2

dist[D] = dist[B] + 2 = -1 + 2 = 1

Step 3: No updates for E

Iteration 3:

Step 1: Initialize distances:

dist[A] = 0

dist[B] = -1

dist[C] = 2

dist[D] = 1

dist[E] = ∞

Step 2: Relax edges:

dist[D] = dist[C] + 5 = 2 + 5 = 7

Step 3: No updates for E

Iteration 4:

Step 1: Initialize distances:

dist[A] = 0

dist[B] = -1

dist[C] = 2

dist[D] = 1

dist[E] = ∞

Step 2: Relax edges:

dist[B] = dist[D] + 1 = 1 + 1 = 2

dist[E] = dist[B] + 2 = 2 + 2 = 4

Step 3: No updates for C

Iteration 5:

Step 1: Initialize distances:

dist[A] = 0

dist[B] = -1

dist[C] = 2

dist[D] = 1

dist[E] = 4

Step 2: Relax edges:

Step 3: No updates for any node

After the 5th iteration, we obtained the final distances:

$$\text{dist}[A] = 0$$
$$\text{dist}[B] = -1$$
$$\text{dist}[C] = 2$$
$$\text{dist}[D] = 1$$
$$\text{dist}[E] = 4$$

These distances represent the shortest paths from node A to all other nodes in the graph.

3.3.3 Applications

The Bellman-Ford algorithm has several applications in various fields. Some of the notable applications include:

- **Routing Protocols:** The algorithm is used in network routing protocols to find the shortest paths between routers in a network. It helps determine the optimal routes for transmitting data packets.
- **Network Analysis:** Bellman-Ford is used for analyzing network topologies and identifying critical nodes or bottleneck points. It helps in understanding the connectivity and efficiency of networks.
- **Distance Vector Routing:** The algorithm is a fundamental component of distance vector routing protocols, such as the Routing Information Protocol (RIP). These protocols use Bellman-Ford to exchange routing information and update routing tables.
- **Negative Cycle Detection:** The Bellman-Ford algorithm can be used to detect negative weight cycles in a graph. This property is utilized in applications such as arbitrage detection in financial markets.
- **Path Planning:** The algorithm is employed in robotics and autonomous systems for path planning. It helps find the shortest paths for robots or vehicles to navigate through obstacles.
- **Graph Analysis:** Bellman-Ford is used for analyzing graph structures and extracting important information such as centrality measures, connectivity, and reachability.

These are just a few examples of the wide range of applications where the Bellman-Ford algorithm is utilized.

3.4 Spanning Trees

A spanning tree of a connected, undirected graph is a tree that spans all the vertices of the graph and is a subgraph that is connected and acyclic. A graph can have multiple spanning trees, but the edges of a spanning tree are a subset of the edges of the original graph.

3.5 Prim's Algorithm

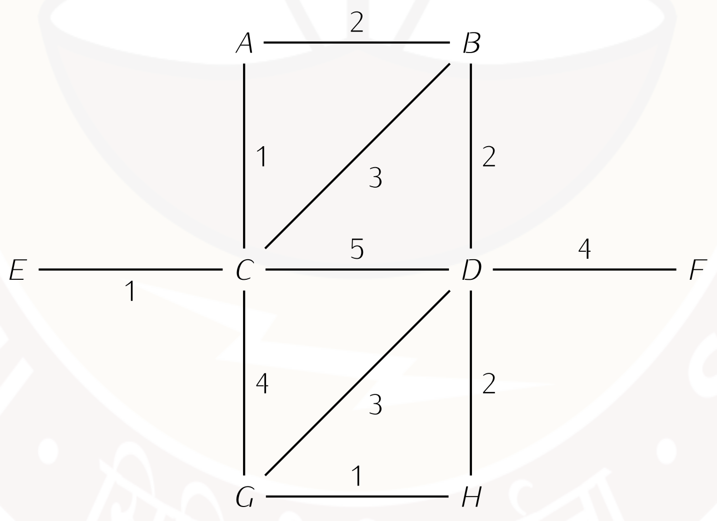
Prim's algorithm is a greedy algorithm used to find the minimum cost spanning tree (MCST) of a weighted undirected graph. The MCST is a subset of the edges that connects all the vertices in the graph while minimizing the total weight of the tree. Prim's algorithm starts with an arbitrary vertex and iteratively adds the shortest edge that connects a vertex in the MCST to a vertex outside the MCST until all vertices are included.

3.5.1 Algorithm

The steps of Prim's algorithm can be summarized as follows:

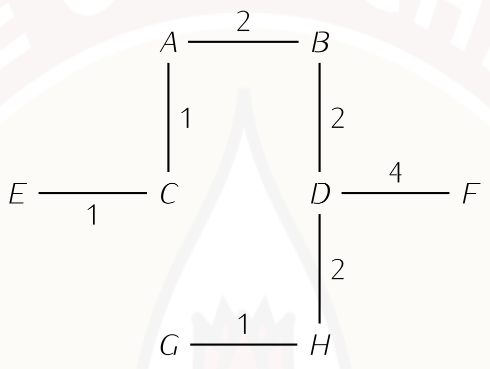
```
1: Let $MST$ be an empty set of edges
2: Select an arbitrary vertex $v$ from $G$
3: Mark $v$ as visited
4: while There are unvisited vertices do
5:     Let $(u, w)$ be the minimum-weight edge with $u$ in $MST$ and $w$ outside MST
6:     Add $(u, w)$ to $MST$
7:     Mark $w$ as visited
8: end while
9: return $MST$
```

Example 3.5.1. Let's consider the following weighted undirected graph G with 8 vertices and 11 edges:



We will apply Prim's algorithm to find the minimum cost spanning tree (MCST) of G. Starting with vertex A, we add the edge (A, C) with weight 1 to MST. Then, we mark vertex C as visited. Next, we add the edge (C, E) with weight 1 to MST and mark vertex E as visited. We continue this process, adding the minimum-weight edges and marking the newly visited vertices until all vertices are visited.

The final minimum cost spanning tree (MCST) obtained using Prim's algorithm for the given graph G is:



The MCST contains 7 edges with a total weight of 13.

3.6 Kruskal's Algorithm

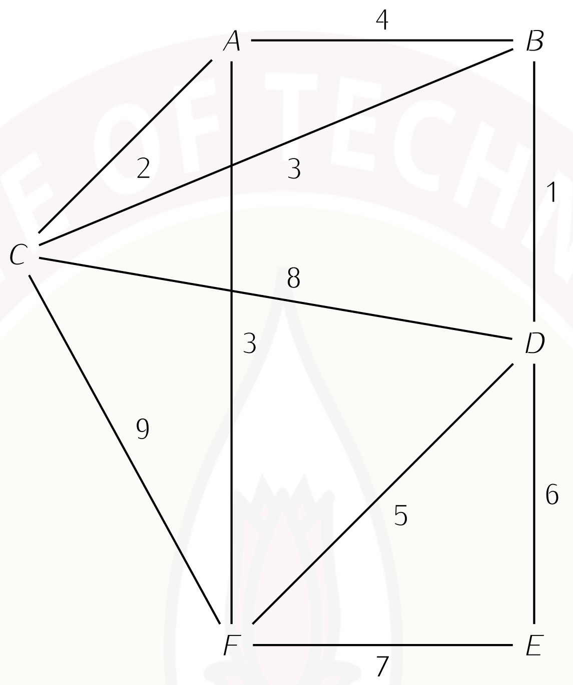
Kruskal's algorithm is a greedy algorithm used to find the minimum cost spanning tree (MCST) of a weighted undirected graph. The MCST is a subset of the edges that connects all the vertices in the graph while minimizing the total weight of the tree. Kruskal's algorithm starts with an empty set of edges and iteratively adds the shortest edge that does not form a cycle in the MCST until all vertices are included.

3.6.1 Algorithm

The steps of Kruskal's algorithm can be summarized as follows:

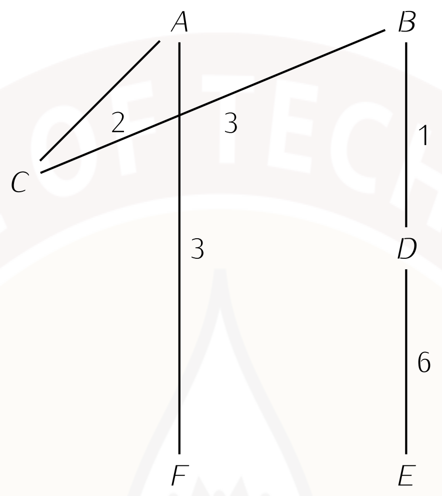
```
1: Let $TE$ be an empty set of edges
2: Sort the edges of $G$ in ascending order of their weights
3: for Each edge $(u, v)$ in the sorted order do
4:     if Adding $(u, v)$ to $TE$ does not form a cycle then
5:         Add $(u, v)$ to $TE$
6:     end if
7: end for
8: return $TE$
```

Example 3.6.1. Let's consider the following weighted undirected graph G with 6 vertices and 10 edges:



We will apply Kruskal's algorithm to find the minimum cost spanning tree (MCST) of G.

Sorting the edges in ascending order of their weights, we start with the edge (B, D) with weight 1. Adding this edge to TE does not form a cycle, so we include it. Next, we consider the edge (A, C) with weight 2. Again, adding this edge to TE does not form a cycle, so we include it. Continuing this process, we add the remaining edges (A, F), (B, C), and (D, E) to TE. Finally, we obtain the minimum cost spanning tree (MCST) as follows:



The MCST contains 5 edges with a total weight of 15 units.

3.7 Problems

1. Which of the following statements is (are) correct?

   (a) If a graph has a negative cycle, shortest paths are not defined.

   (b) A negative cycle is one which has only negative edge weights.

   (c) In a weighted graph, the adjacency matrix records the weight where ever there is an edge and 0 if there is no edge.

   (d) Shortest path in a weighted graph need not be minimum in terms of the number of edges.

Use the following information for questions 2, 3, and 4.
The Ministry of Earth Sciences (MoES) gave flood warnings to the cities $A, B, C, D, E, F,$ and G which are very near the foothills of the Himalayas. At time 0 minutes, city B was completely flooded. In the graph given below, vertices represent cities and edges represent how cities are connected.
Note: The weight of each edge indicates the time in minutes by which water reaches different cities.

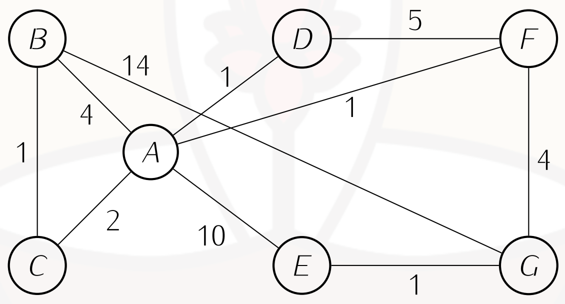


Figure 3.3:

2. At what time, will city G start flooding?

3. What is the shortest path to reach city D from the source of the flood (B)?

4. At what time will all the cities be flooded?

Use the following information for questions 5 and 6.
While using Bellman-Ford Algorithm for the graph shown below, let $D(v)$ be the shortest distance of vertex v from the source vertex after 7 iterations.

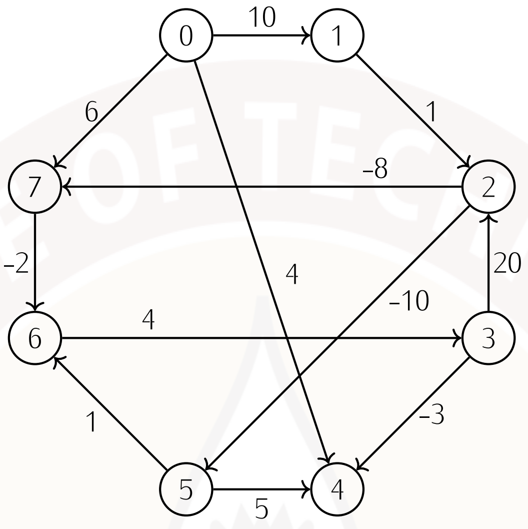


Figure 3.4:

5. Suppose the source vertex is 0 and we perform Bellmann-Ford Algorithm. Which of the following options is (are) correct?

   (a) D(5) = D(6) = 1
   (b) D(3) = 5
   (c) D(4) = 2
   (d) D(4) = 5

6. If the source vertex is changed from vertex 0 to vertex 4, then which of the following options is (are) correct?

   (a) Bellman-Ford algorithm stabilizes after the first iteration.
   (b) Bellman-Ford algorithm will not be applicable.
   (c) $D(v)$ is finite for some vertex v other than the source vertex.
   (d) None of the above.

7. For what values of x can we use the Bellman-Ford algorithm to find the shortest path from a source vertex 0 to every other vertex in the graph given below?

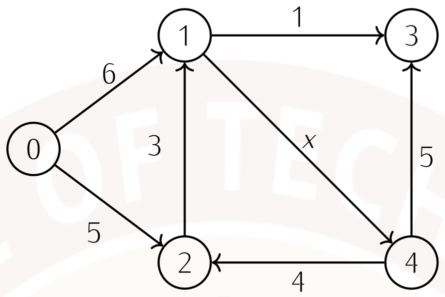


Figure 3.5:

(a) (-6,∞)

(b) (-14,∞)

(c) (-7,∞)

(d) (-21,∞)

8. Which of the following is (are) matched correctly?

   (a) For transitive closure - Warshall algorithm.

   (b) For all pair shortest path - Floyd-Warshall algorithm.

   (c) For single source shortest distance for non-negative edge weights - Dijkstra's algorithm.

   (d) For single source shortest distance for negative or non-negative edge weights - Bellman-Ford algorithm.

Use the below graph for questions 9, 10, 11, and 12.
(Hint: Use Floyd-Warshall Algorithm.)

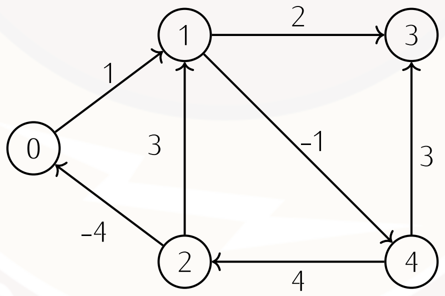


Figure 3.6:

9. Which of the following matrices represents SP^0?

SP^0	0	1	2	3	4
0	∞	1	∞	∞	∞
1	∞	∞	∞	2	-1
2	-4	3	∞	∞	∞
3	∞	∞	-4	∞	∞
4	∞	∞	4	3	∞

(a)

SP^0	0	1	2	3	4
0	∞	1	∞	∞	∞
1	∞	∞	∞	2	-1
2	-4	3	∞	∞	∞
3	∞	∞	∞	∞	∞
4	∞	∞	4	3	∞

(b)

SP^0	0	1	2	3	4
0	∞	∞	-4	∞	∞
1	1	∞	3	∞	∞
2	∞	∞	∞	∞	4
3	∞	2	∞	∞	3
4	∞	-1	∞	∞	∞

(c)

SP^0	0	1	2	3	4
0	∞	1	∞	∞	∞
1	∞	∞	∞	2	-1
2	-4	3	∞	∞	∞
3	∞	∞	∞	∞	∞
4	∞	∞	3	4	∞

(d)

10. Which of the following matrices represents SP^3?

SP^3	0	1	2	3	4
0	∞	1	∞	3	0
1	∞	∞	∞	2	-1
2	-4	-3	∞	-1	-4
3	∞	∞	∞	∞	∞
4	0	1	4	3	0

(a)

SP^3	0	1	2	3	4
0	∞	1	∞	∞	∞
1	∞	∞	∞	2	-1
2	-4	-3	∞	-1	-4
3	∞	∞	∞	∞	∞
4	∞	1	4	3	∞

(b)

SP^3	0	1	2	3	4
0	∞	1	∞	∞	∞
1	∞	∞	∞	2	-1
2	-4	-3	∞	-1	-4
3	∞	∞	∞	∞	∞
4	0	1	4	3	∞

(c)

SP^3	0	1	2	3	4
0	∞	1	∞	∞	∞
1	∞	∞	∞	2	-1
2	-4	-3	∞	∞	-4
3	∞	∞	∞	∞	∞
4	∞	1	4	3	0

(d)

11. Which of the following matrices represents SP^5?

SP^5	0	1	2	4	3
0	0	1	4	3	0
1	-1	0	3	2	-1
2	-4	-3	0	-1	-4
3	∞	∞	-4	∞	∞
4	0	1	4	3	0

(a)

SP^5	0	1	2	3	4
0	0	1	4	3	0
1	-1	0	3	2	-1
2	-4	-3	0	-1	-4
3	∞	∞	-4	∞	∞
4	0	1	4	3	0

(b)

SP^5	0	1	2	3	4
0	0	1	4	3	0
1	-1	0	3	2	-1
2	-4	-3	0	-1	-4
3	∞	∞	∞	∞	∞
4	0	1	4	3	0

(c)

SP^5	0	1	2	3	4
0	0	1	4	3	0
1	∞	∞	∞	2	-1
2	-4	3	∞	∞	∞
3	∞	∞	∞	∞	∞
4	∞	∞	3	4	∞

(d)

12. If $SP^i = SP^j$, for some $i, j \in \{0, 1, 2...5\}$, then which of the following may be the values of i and j?

    (a) $i = 4$ and $j = 3$
    (b) $i = 3$ and $j = 4$
    (c) $i = 2$ and $j = 3$
    (d) $i = 3$ and $j = 2$

13. Which of the following is (are) correct with respect to the spanning tree G?

    (a) G is connected.
    (b) G is acyclic.
    (c) G has n edges.
    (d) Among the different spanning trees, one with minimum cost is the minimum cost spanning tree.

14. Suppose the graph given below is a spanning tree of a graph G.

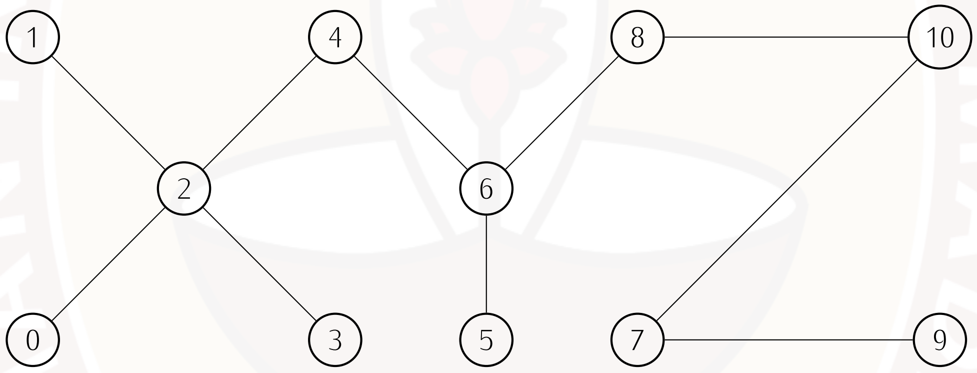
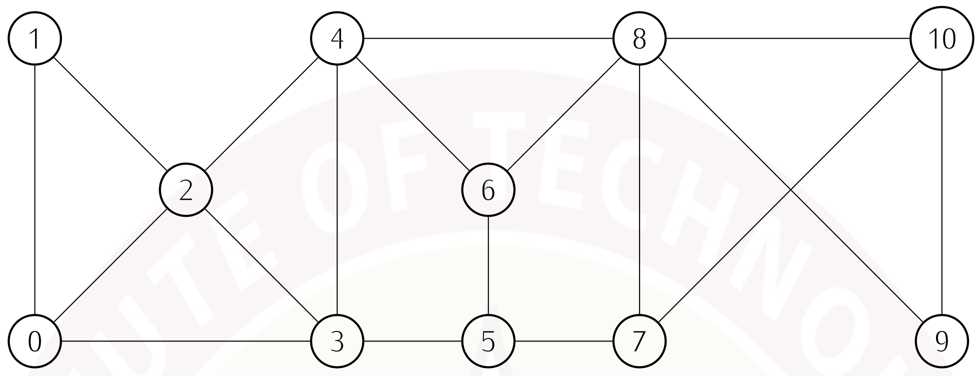
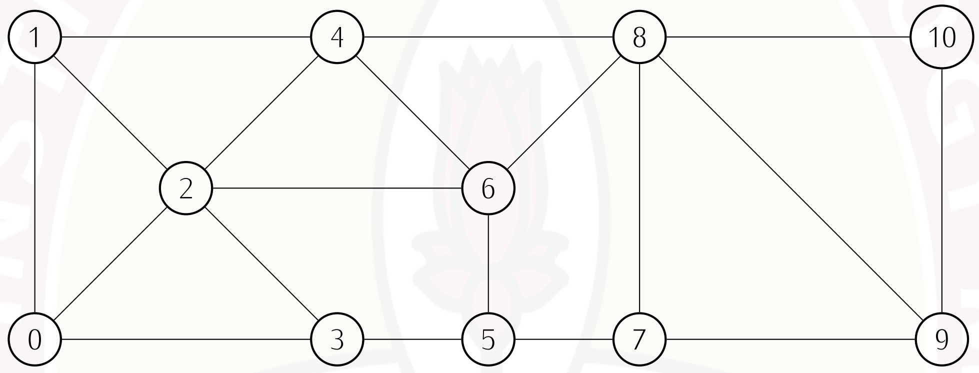


Figure 3.7:

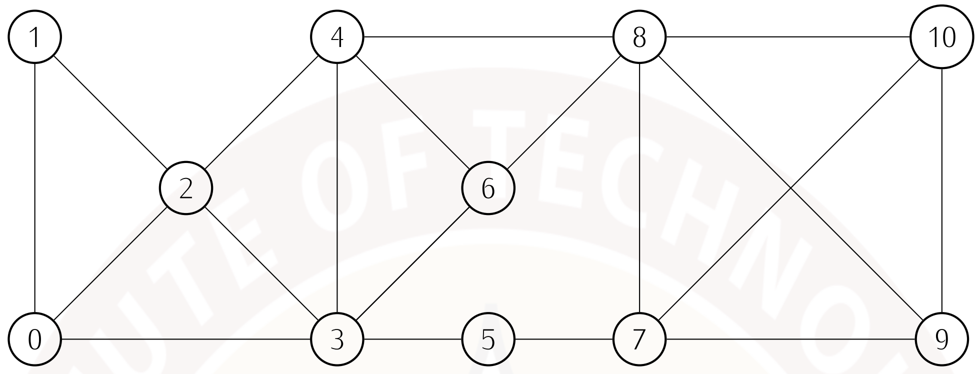
Which of the following graphs may represent G?



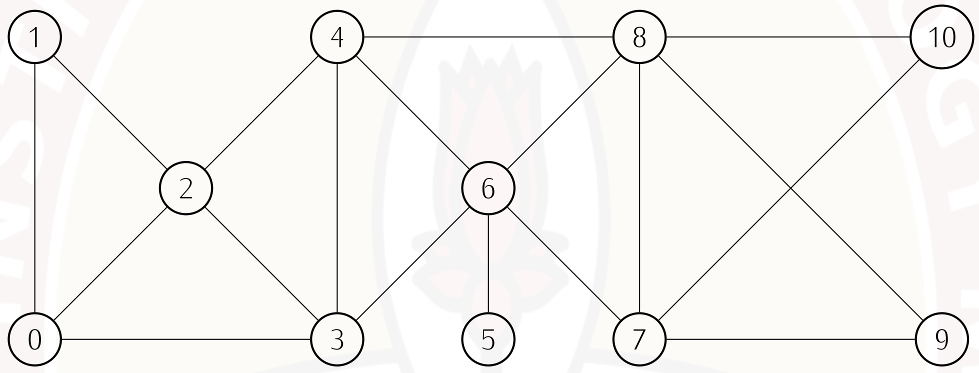
(a)



(b)



(c)



(d)

Use the following information for questions 15, 16, and 17:
An undirected weighted graph G is given below.

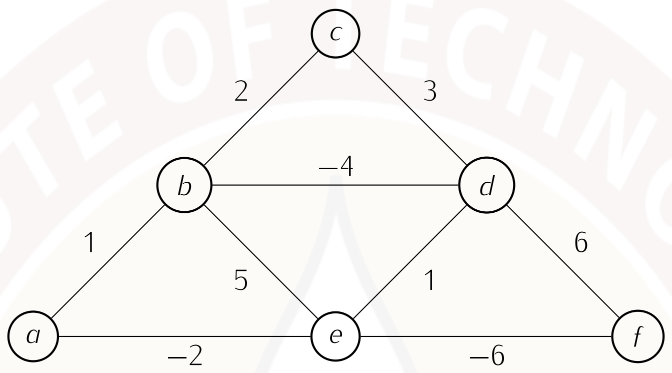
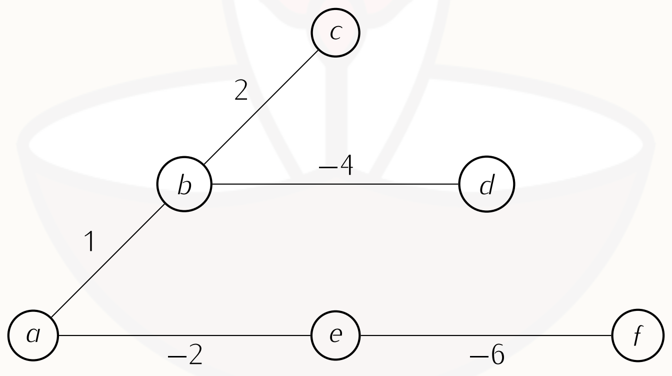
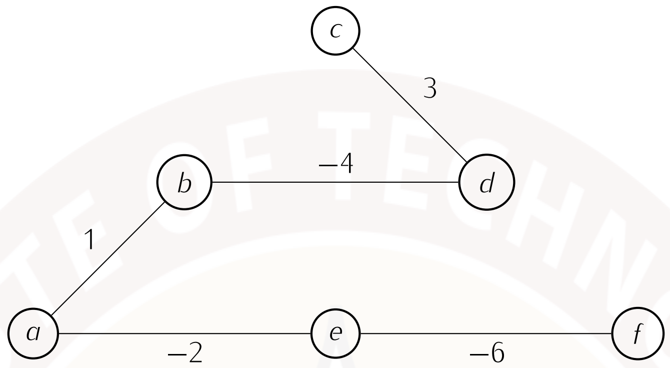


Figure 3.8:

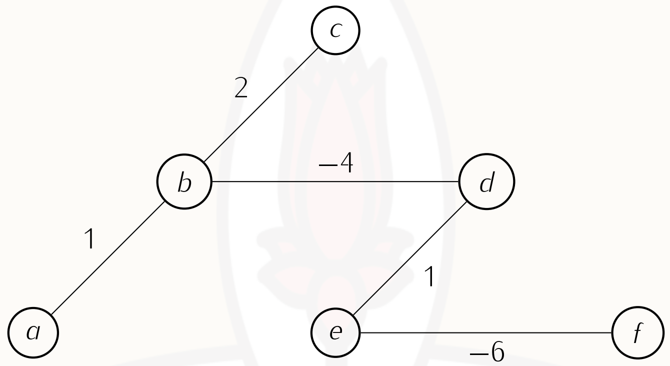
15. If we perform Prim's algorithm on G, then which of the following options may represent a minimum cost spanning tree?



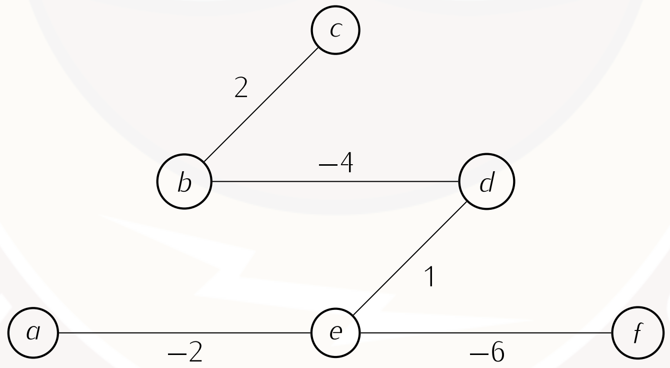
(a)



(b)



(c)



(d)

16. Which of the following is the order in which edges are added to the minimum cost spanning tree?

(a) $(e,f),(a,e),(e,d),(b,d),(b,c)$

(b) $(e,f),(b,d),(a,e),(a,b),(b,c)$

(c) $(e,f),(a,e),(b,d),(e,d),(b,c)$

(d) $(e,f),(e,d),(b,d),(a,b),(b,c)$

17. What is the weight of the minimum cost spanning tree?

Use the following information for questions 18, 19, and 20:
An undirected weighted graph G is given below.

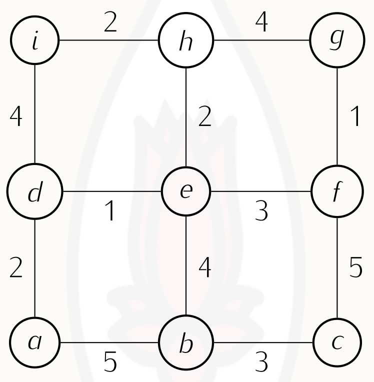
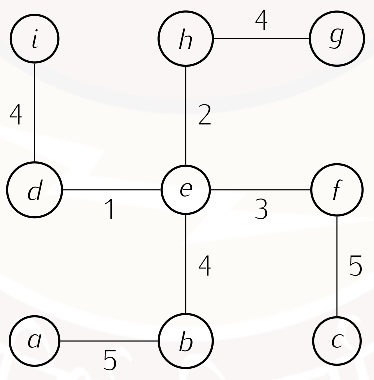
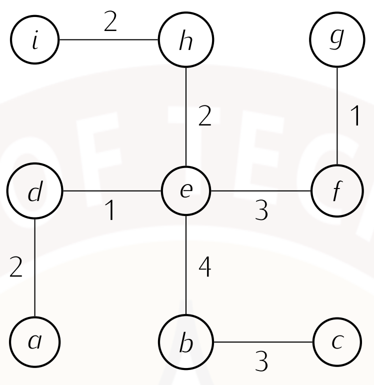


Figure 3.9:

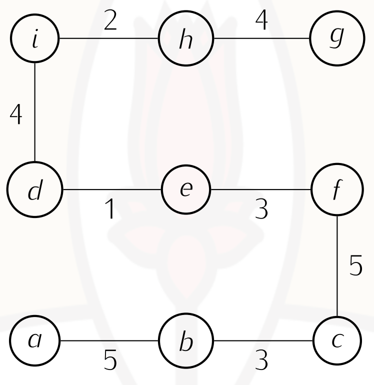
18. If we perform Kruskal's algorithm on G, then which of the following options may represent the minimum cost spanning tree?



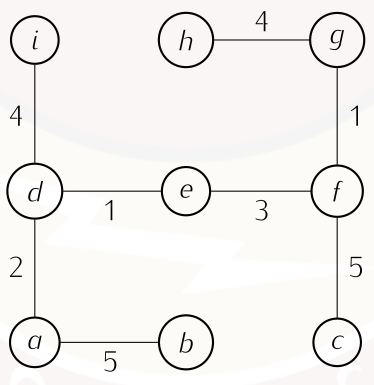
(a)


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

(b)


i
h
g
d
e
f
a
b
c
2
4
4
1
3
5
5
3

(c)


i
h
g
d
e
f
a
b
c
4
4
1
1
3
2
5
5

(d)

19. Which of the following options is (are) correct w.r.t the graph G?

    (a) The minimum cost spanning tree of the graph G is unique.

    (b) There can be more than one minimum cost spanning tree of the graph G because there are some edges with equal weights.

    (c) The weight of the minimum cost spanning tree of the graph G is 18.

    (d) The order in which the edges are added to the minimum cost spanning tree is not unique.

20. Find the number of edges that are removed from the graph G to obtain the minimum cost spanning tree.

21. Suppose Prim's algorithm and Kruskal's algorithm is performed on a graph G separately to find the minimum cost spanning tree. Which of the following options will always be the same for both algorithms?

    (a) The weight of the minimum cost spanning tree of the graph G.

    (b) The order in which the edges are added to the minimum cost spanning tree.

    (c) Number of edges in the minimum cost spanning tree.

    (d) The minimum weight edge in the minimum cost spanning tree.

4. Answers

4.1 Chapter 1

1. $A \longrightarrow F \longrightarrow J \longrightarrow B$
2. 4
3. $\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$
4. $\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$
5. Vertex cover $V_c = \{2, 3\}$.
6.
7. FIFO (First-In-First-Out) structure.
8. LIFO (Last-In-First-Out) structure.
9.
10. Number of edges will be less than 14196.
11. $\frac{n^2}{4}$
12. $(7, 10)$
13. Shiva's house.

14. 8
15. $|V_i| = 4, |V_c| = 4.$
16. Options 1,3, and 4.
17. 4
18. 2
19. 16

4.2 Chapter 2

1. (c)
2. (b)
3. (a),(b),(d)
4. (b),(c)
5. (b),(d)
6. (b),(c)
7. 5

4.3 Chapter 3

1. Options 1,3, and 4.
2. 8 minutes.
3. $B \longrightarrow C \longrightarrow A \longrightarrow D.$
4. 9 minutes.
5. (a),(b),(c)
6. (a)
7. $(-7, \infty)$
8. All options are correct.
9. (b)

10. (a)
11. (c)
12. (a),(b)
13. (a),(b),(d)
14. (d)
15. (a),(d)
16. (a)
17. –9
18. (b)
19. (a),(c),(d)
20. 4
21. (a),(c),(d)