DS-Graphs-Unit-4

18 May 2025 23:35

Graph: A graph is a pair (V, E), where V is a set of nodes, called vertices, and £ is a collection of pairs of vertices, called edges. • Vertices and edges are positions and store elements

Definitions that we use:

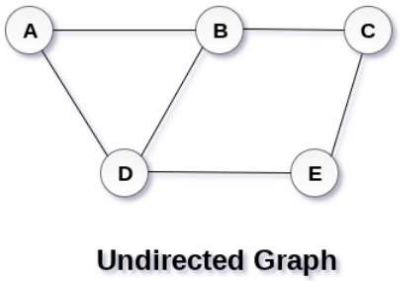
Directed edge:
Undirected edge
Directed graph:
Undirected graph:

Directed edge: ▪ ordered pair of vertices (u, v) ▪ first vertex u is the origin ▪ second vertex v is the destination ▪ Example: one-way road traffic
Undirected edge: ▪ unordered pair of vertices (u, v) ▪ Example: railway lines
Directed graph: ▪ all the edges are directed ▪ Example: route network
Undirected graph: ▪ all the edges are undirected ▪ Example: flight network

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.
A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure

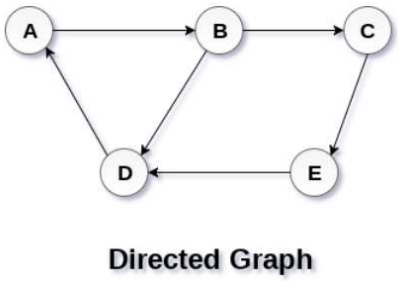
From <https://www.tpointtech.com/graph-data-structure>



Undirected Graph

in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

A directed graph is shown in the following figure.



Directed Graph

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

Graph Representation in Data Structure

- **Sequential representation** (or, Adjacency matrix representation)
- **Linked list representation** (or, Adjacency list representation)

In sequential representation, an adjacency matrix is used to store the graph. Whereas in linked list representation, there is a use of an adjacency list to store the graph.

Sequential Representation

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

An entry A_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between V_i and V_j.

If an Undirected Graph G consists of n vertices, then the adjacency matrix for that graph is n x n, and the matrix A = [aij] can be defined as -

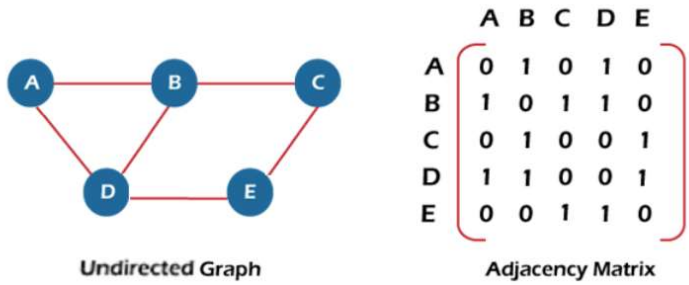
a_{ij} = 1 {if there is a path exists from V_i to V_j}

a_{ij} = 0 {Otherwise}

It means that, in an adjacency matrix, 0 represents that there is no association exists between the nodes, whereas 1 represents the existence of a path between two edges.

If there is no self-loop present in the graph, it means that the diagonal entries of the adjacency matrix will be 0.

Now, let's see the adjacency matrix representation of an undirected graph.



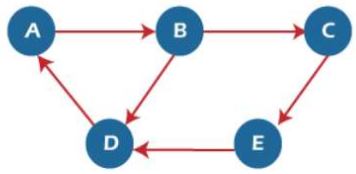
Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

Screen clipping taken: 21-05-2025 05:50

Consider the below-directed graph and try to construct the adjacency matrix of it.

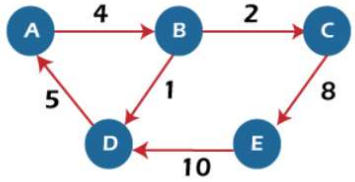


Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

Screen clipping taken: 21-05-2025 05:50



weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

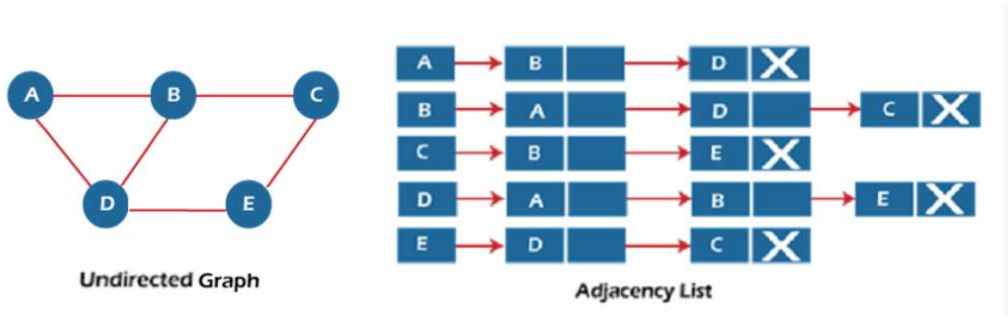
Adjacency Matrix

Screen clipping taken: 21-05-2025 05:50

Linked List Representation

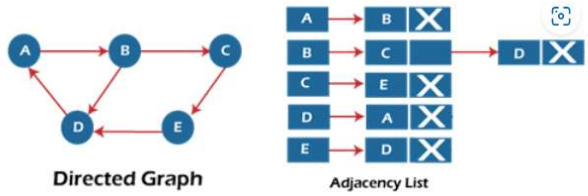
An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.



In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.
An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



Screen clipping taken: 21-05-2025 05:54

Graph Traversals:
• Depth First Search [DFS]
• Breadth First Search [BFS]

Depth First Search [DFS]
DFS algorithm works in a manner similar to preorder traversal of the trees. Like preorder traversal, internally this algorithm also uses stack

Initially all vertices are marked unvisited (false).
The DFS algorithm starts at a vertex u in the graph.
By starting at vertex u it considers the edges from u to other vertices.
If the edge leads to an already visited vertex, then backtrack to current vertex u.
If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex.
That means the new vertex becomes the current vertex.
Follow this process until we reach the dead-end.
At this point start backtracking.
The process terminates when backtracking leads back to the start vertex.
We can see that sometimes an edge leads to an already discovered vertex.
These edges are called back edges, and the other edges are called tree edges because deleting the back edges from the graph generates a tree.

The final generated tree is called the DFS tree and the order in which the vertices are processed is called DFS numbers of the vertices

Tree edge: encounter new vertex
Back edge: from descendent to ancestor
Forward edge: from ancestor to descendent
Cross edge: between a tree or subtrees
The process of returning from the “dead end” is called backtracking.

The time complexity of DFS is O(V + E), if we use adjacency lists for representing the graphs
if an adjacency matrix is used for a graph representation, then all edges adjacent to a vertex can’t be found efficiently, and this gives O(V 2) complexity

Applications of DFS
• Topological sorting
• Finding connected components
• Finding articulation points (cut vertices) of the graph
• Finding strongly connected components
• Solving puzzles such as mazes

==========

Breadth First Search [BFS]
The BFS algorithm works similar to level – order traversal of the trees.
Like level – order traversal, BFS also uses queues. In fact, level – order traversal got inspired from BFS.
BFS works level by level.
Initially, BFS starts at a given vertex, which is at level 0.
In the first stage it visits all vertices at level 1 (that means, vertices whose distance is 1 from the start vertex of the graph).
In the second stage, it visits all vertices at the second level. These new vertices are the ones which are adjacent to level 1 vertices.
BFS continues this process until all the levels of the graph are completed.
Generally queue data structure is used for storing the vertices of a level.
As similar to DFS, assume that initially all vertices are marked unvisited (false).
Vertices that have been processed and removed from the queue are marked visited (true).
We use a queue to represent the visited set as it will keep the vertices in the order of when they were first visited.

Time complexity of BFS is O(V + E), if we use adjacency lists for representing the graphs,
and O(V 2) for adjacency matrix representation.

Applications of BFS
• Finding all connected components in a graph
• Finding all nodes within one connected component
• Finding the shortest path between two nodes
• Testing a graph for bipartiteness

==========

Minimum Spanning Tree (MST)

- **Undirected graph:** An undirected graph is a graph in which all the edges do not point to any particular direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of V vertices and a set of E edges, each edge connecting two different vertices.
- **Connected graph:** A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
- **Directed graph:** Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges.
A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes).
Edges of the spanning tree may or may not have weights assigned to them.
All the possible spanning trees created from the given graph G would have the same number of vertices,
but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

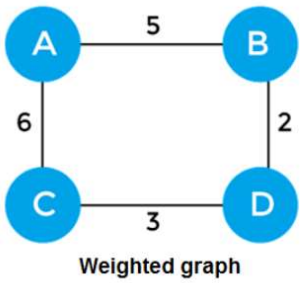
A complete undirected graph can have n^{n-2} number of spanning trees where n is the number of vertices in the graph. Suppose, if $n = 5$, the number of maximum possible spanning trees would be $5^{5-2} = 125$.

Minimum Spanning Tree

A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum.
The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.
In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

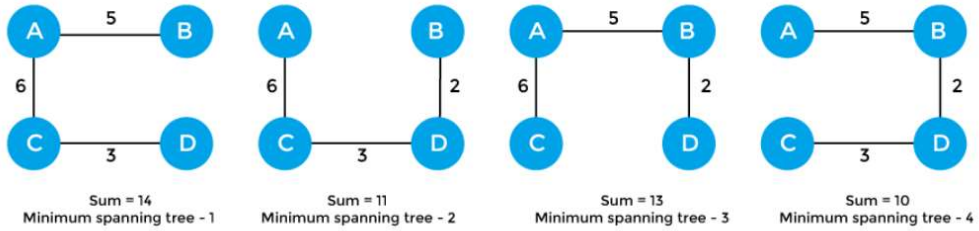
Example of Minimum Spanning Tree

Let's understand the minimum spanning tree with the help of an example.



Weighted graph

The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are -



Sum = 14
Minimum spanning tree - 1

Sum = 11
Minimum spanning tree - 2

Sum = 13
Minimum spanning tree - 3

Sum = 10
Minimum spanning tree - 4

So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is -

Screen clipping taken: 22-05-2025 07:59

Algorithms for Minimum Spanning Tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

- Prim's Algorithm
- Kruskal's Algorithm

Let's see a brief description of both of the algorithms listed above.

Prim's algorithm - It is a greedy algorithm that starts with an empty spanning tree. It is used to find the minimum spanning tree from the graph. This algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Kruskal's algorithm - This algorithm is also used to find the minimum spanning tree for a connected weighted graph. Kruskal's algorithm also follows greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.

How does the prim's algorithm work?

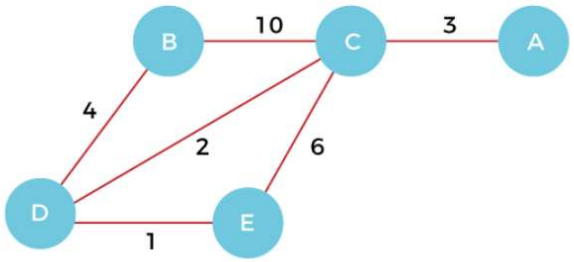
Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed

Example of prim's algorithm

Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

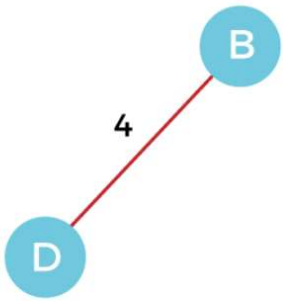
Suppose, a weighted graph is -



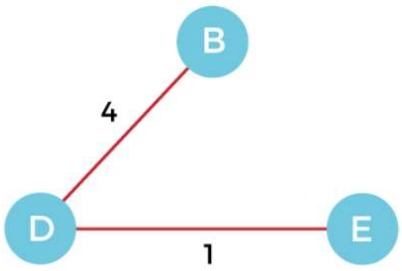
Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.

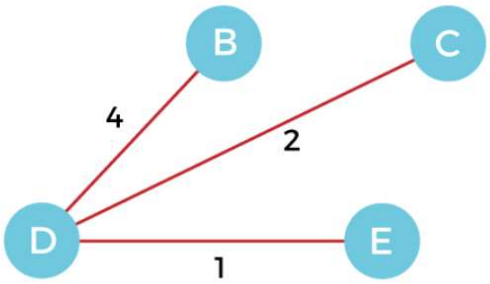
Screen clipping taken: 22-05-2025 08:08



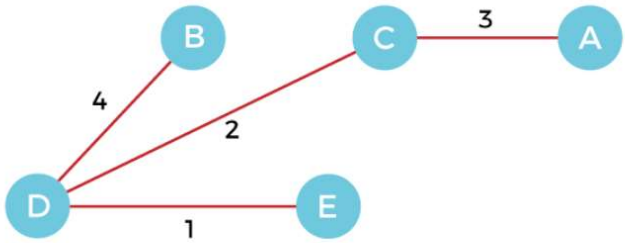
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

- **Time Complexity**

Data structure used for the minimum edge weight	Time Complexity
Adjacency matrix, linear searching	O(|V|2)
Adjacency list and binary heap	O(|E| log |V|)
Adjacency list and Fibonacci heap	O(|E|+ |V| log |V|)

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

How does Kruskal's algorithm work?

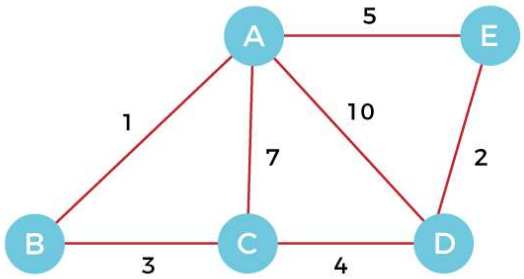
In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

Example of Kruskal's algorithm

Now, let's see the working of Kruskal's algorithm using an example. It will be easier to understand Kruskal's algorithm using an example.

Suppose a weighted graph is -



The weight of the edges of the above graph is given in the below table -

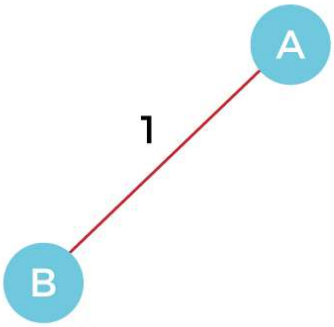
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

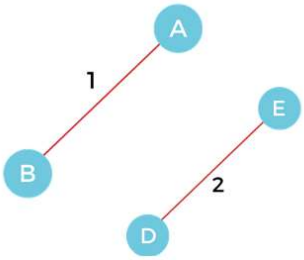
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Screen clipping taken: 22-05-2025 08:14

Step 1 - First, add the edge **AB** with weight **1** to the MST.

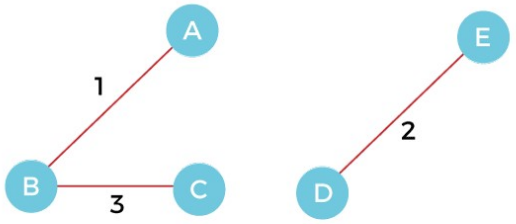


Step 2 - Add the edge **DE** with weight **2** to the MST as it is not creating the cycle.

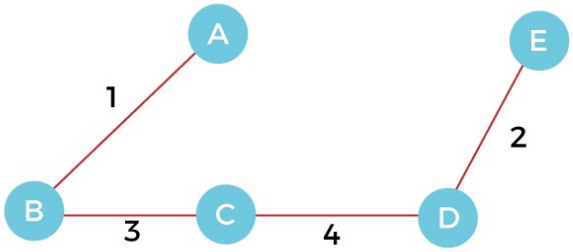


Screen clipping taken: 22-05-2025 08:15

Step 3 - Add the edge **BC** with weight **3** to the MST, as it is not creating any cycle or loop.



Step 4 - Now, pick the edge **CD** with weight **4** to the MST, as it is not forming the cycle.



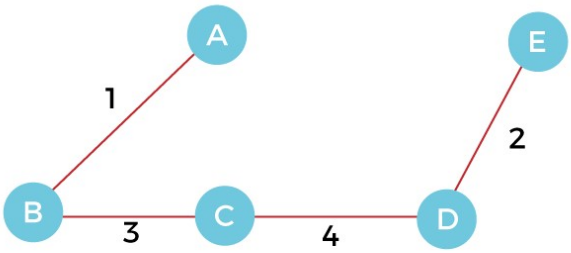
Step 5 - After that, pick the edge **AE** with weight **5.** Including this edge will create the cycle, so discard it.

Step 6 - Pick the edge **AC** with weight **7.** Including this edge will create the cycle, so discard it.

Step 7 - Pick the edge **AD** with weight **10.** Including this edge will also create the cycle, so discard it.

Screen clipping taken: 22-05-2025 08:15

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is -



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

Screen clipping taken: 22-05-2025 08:16