Graph

A graph is a non-linear data structure that consists of a set of vertices (also called nodes) and a set of edges connecting those vertices. Graphs are used to represent relationships between objects, such as connections between computers in a network, relationships between web pages on the internet, or dependencies between software components.

A graph can be directed or undirected, depending on whether the edges have a direction or not. In a directed graph, the edges are represented by arrows indicating the direction of the connection. In an undirected graph, the edges are represented by lines connecting the vertices without indicating a direction.

Fig: Undirected Graph

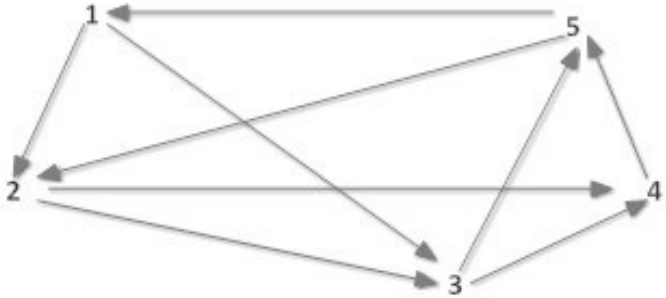


Fig: Directed Graph

Graph

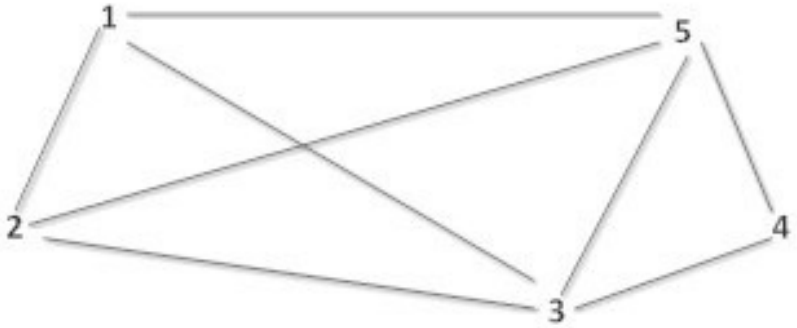


Figure: Simple Graph

A graph **G** = (**V**, **E**) consists of two sets **V** and **E** where **V** is a finite and non-empty set of vertices and **E** is a set of pair of vertices and these pair of vertices called edges. Graph can not contain duplicate value vertex.

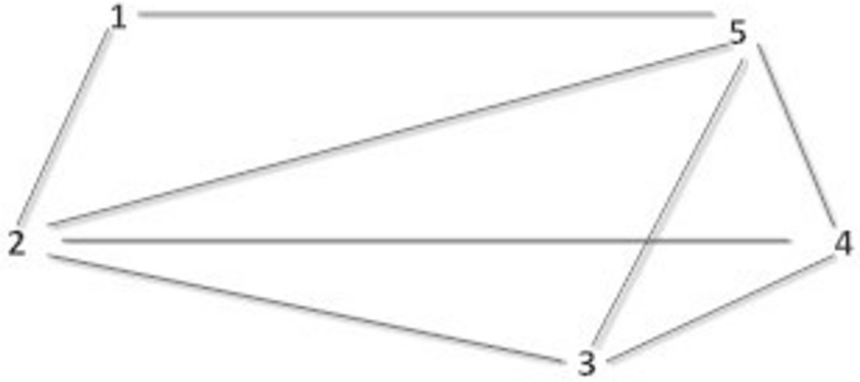
The set of vertices for the above graph is:

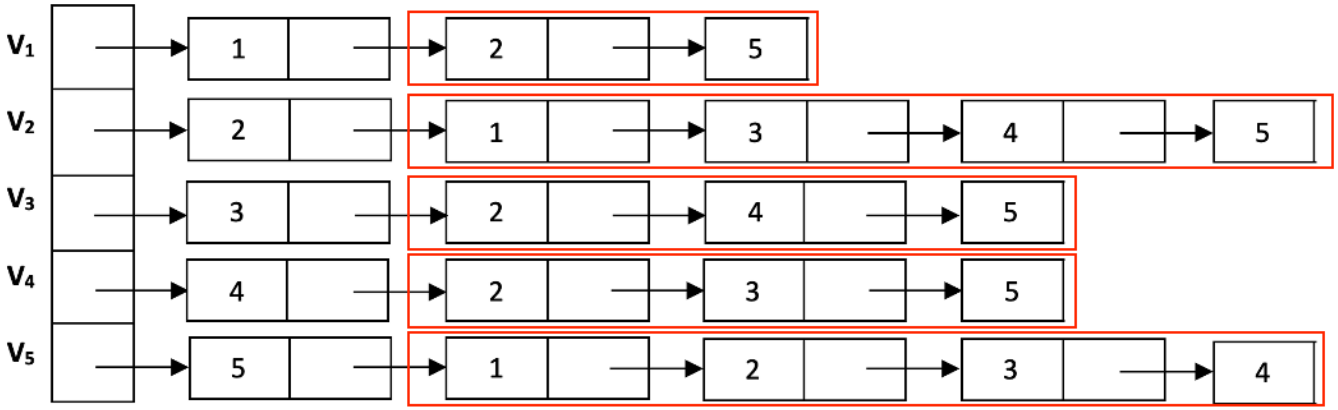
V(G) = {1, 2, 3, 4, 5}

and the set of edges of the above graph is:

E(G) = {(1, 2),(1, 3), (1, 5), (2, 3), (2, 5), (3, 4), (3, 5), (4, 5)}

Adjacency List Representation



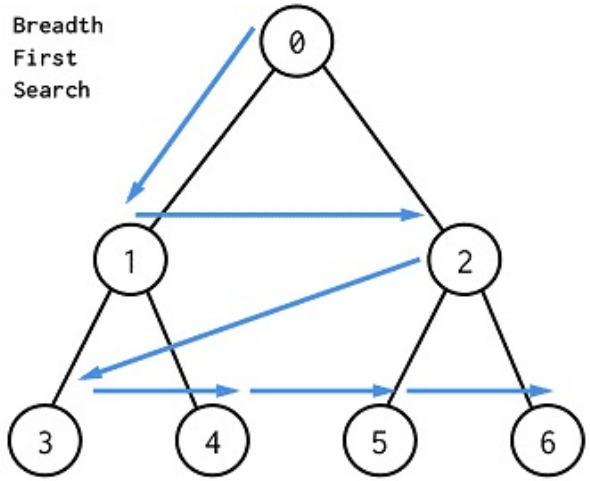


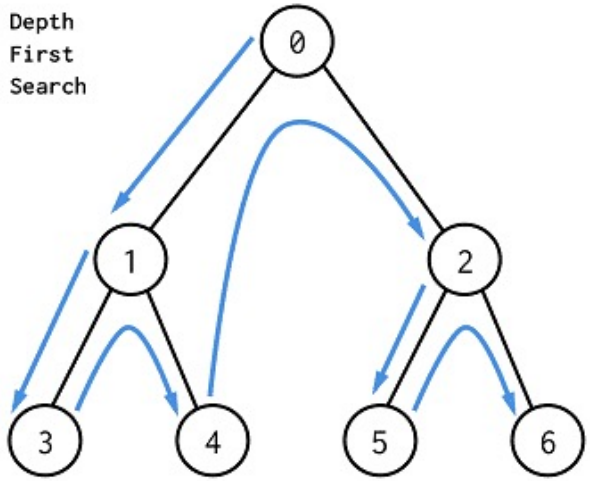
BFS & DFS

BFS (Breadth-First Search) and DFS (Depth-First Search) are two graph traversal algorithms used to systematically explore and visit all the vertices of a graph.

BFS explores the graph in a breadth-ward motion, visiting all the neighbors of a vertex before moving on to their neighbors.

DFS explores the graph in a depth-ward motion, going as deep as possible along each branch before backtracking.





BFS vs DFS

incapp

BFS(Breadth First Search)	DFS(Depth First Search)
Traverses a graph level-wise.	Traverses a graph depth-wise.
Uses queue data structure to store the nodes to be visited.	Uses stack data structure to store the nodes to be visited.
BFS takes lesser time if the destination node is closer to the source.	DFS takes more time than BFS if the destination node is closer to the source node but is not explored first.
BFS Less memory efficient than DFS as it has to store nodes of each layer before moving to the next layer.	DFS is memory efficient as it only needs to store the nodes on the path from the source node to the current node.
BFS always finds the minimal path from the source node to the destination node.	DFS might not find the shortest path to a given node when there are multiple possible paths from the source node to the destination node.

Algo to Create Graph

Here **edges** is an array to represent graph, **v** will store the number of vertices in graph, **e** will store the number of edges in graph, and then the algorithm is:

1. Read v.
2. Read e.
3. Create **edges** array of LinkedList type
4. Repeat for int i = 0 to e-1
    a. Read source and destination of edge
    b. Set edges[source]->add(destination)
    c. Set edges[destination]->add(source)
5. End

Algo for BFS Traversal

Here **edges** is an array to represent graph, **v** will store the number of vertices in graph, **e** will store the number of edges in graph, **visited** is an array having visited status of the vertices of graph, **q** is the queue where vertices of graph will be enqueued, **sv** will store the source vertex, **current** will store the dequeued from queue and then the algorithm is:

1. Read sv.
2. Enqueue sv to q
3. Set visited[sv] = true
4. Repeat while queue q is not empty
    a. Set current = element dequeued from queue q.
    b. Print, current.
    c. Repeat for-each (neighbor : edges[current])
        - Check, if visited[neighbor] = false, then:
            - Enqueue vertex neighbor into queue q.
            - Set visited[neighbor] = true.
5. End

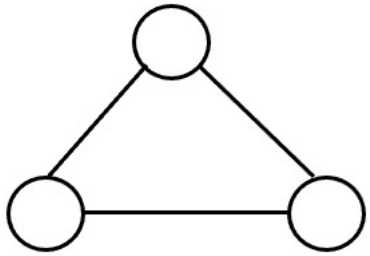
Algo for DFS Traversal

Here **edges** is an array to represent graph, **v** will store the number of vertices in graph, **e** will store the number of edges in graph, **visited** is an array having visited status of the vertices of graph, **stack** is the stack where vertices of graph will be pushed, **sv** will store the source vertex, **current** will store the popped from stack and then the algorithm is:

1. Read sv.
2. Push sv to stack
3. Set visited[sv] = true
4. Repeat while stack is not empty
    a. Set current = element popped from stack.
    b. Print, current.
    c. Repeat for-each (neighbor : edges[current])
        - Check, if visited[neighbor] = false, then:
            - Push vertex neighbor into stack.
            - Set visited[neighbor] = true.
5. End

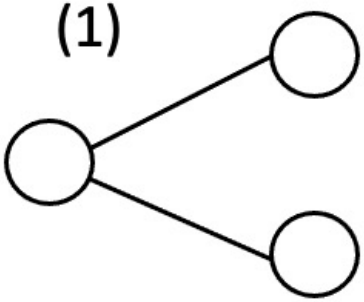
Spanning Tree

A tree that connects all the vertices of a graph without loop is known as **spanning tree**.

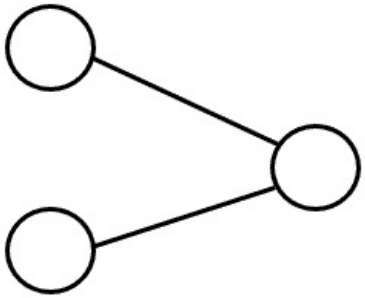


And all the possible spanning trees of above graph are given below:

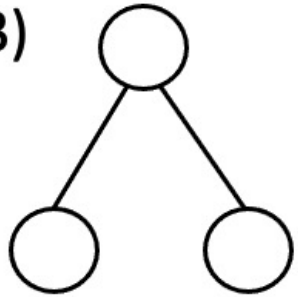
(1)



(2)



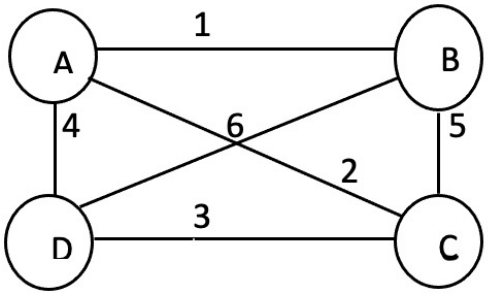
(3)



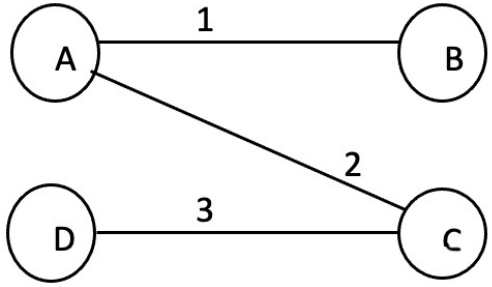
Note: A graph can have maximum V^{v-2} spanning trees where V is the no of vertices of the graph.

Minimum Cost Spanning Tree

A spanning tree, in which tree weight is minimum of all possible spanning tree is known as **minimum cost spanning** or **minimum spanning .**



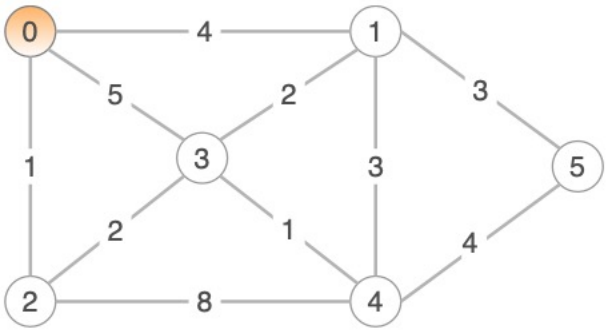
The minimum cost spanning tree of above graph is given below:



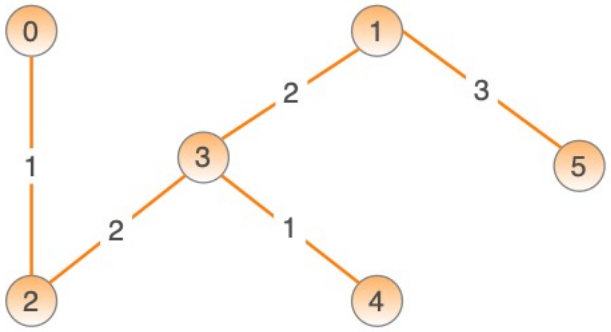
Above tree is a minimum cost spanning tree because it connects all vertices of graph without loop and tree weight is minimum of all possible spanning trees of graph.

Prim’s Algorithm

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. The algorithm starts with a single vertex and gradually grows a tree by adding the cheapest edge that connects the tree to a new vertex.

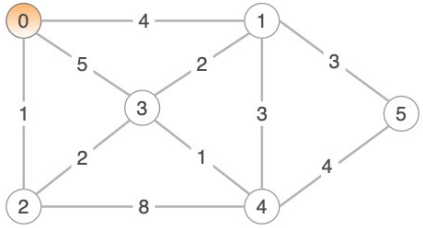


Given Graph

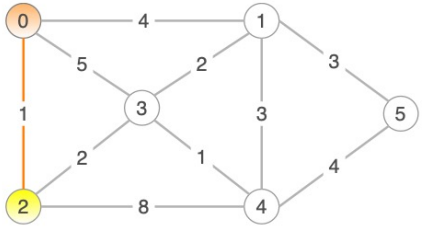


Minimum Spanning Tree

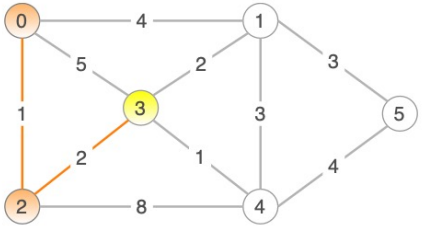
Prim's Algorithm working



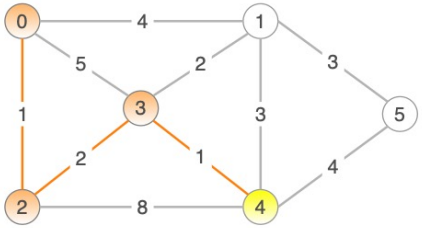
Step 1. Select any random node (0) and add it to the spanning tree.



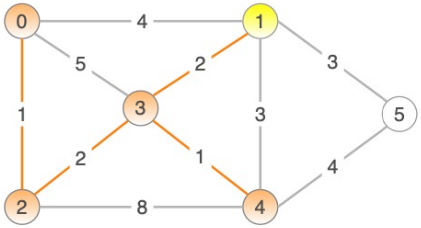
Step 2. Select an edge with the smallest cost from node (0) and add it to the spanning tree.



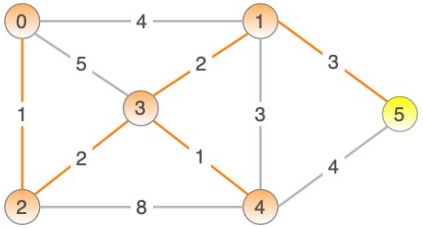
Step 3. Select an edge with the smallest cost from node(s) (0, 2) and add it to the spanning tree.



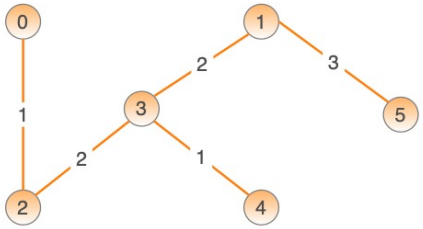
Step 4. Select an edge with the smallest cost from node(s) (0, 2, 3) and add it to the spanning tree.



Step 5. Select an edge with the smallest cost from node(s) (0, 2, 3, 4) and add it to the spanning tree.

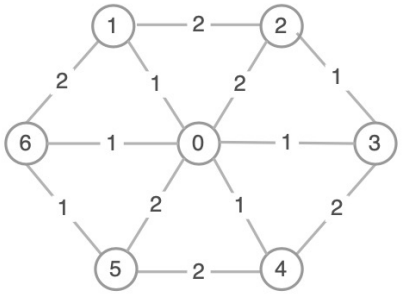


Step 6. Select an edge with the smallest cost from node(s) (0, 2, 3, 4, 1) and add it to the spanning tree.

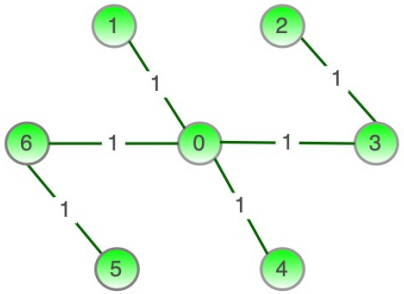


e) Minimum spanning tree with edges (0-1),(3-4),(1-3),(2-3) and (1-5).
Total cost: 9

Prim's Algorithm Assignment



Graph 2



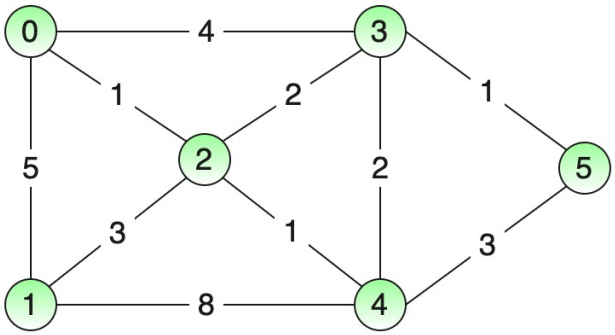
Minimum spanning tree with edges (0-1), (0-3), (0-4),(0-6), (2-3) and (5-6). Total cost: 6

Prim's Algorithm

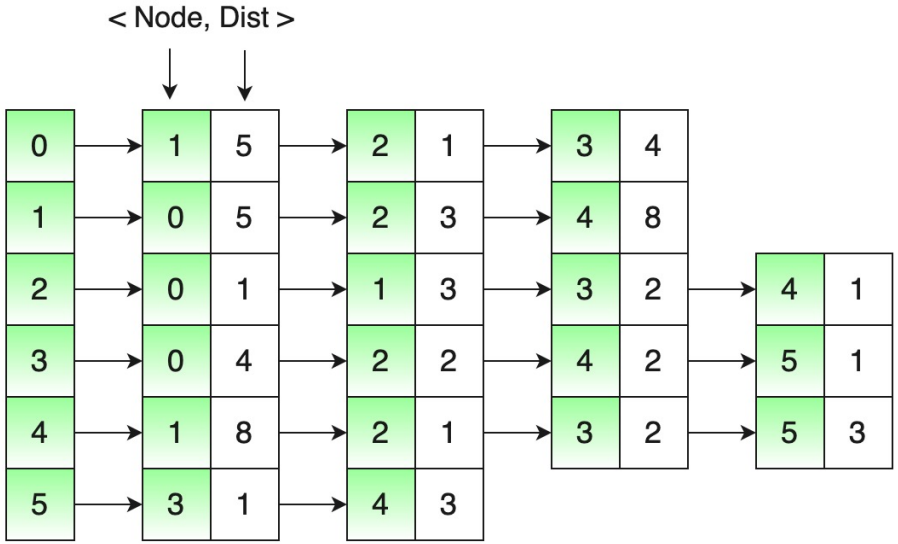
Algo goes here

Dijkstra's Algorithm

- Dijkstra's algorithm finds the shortest path in a **weighted graph** from a single source.
- The weighted graph for Dijkstra's algorithm contains only positive edge weights.
- It uses a priority queue to select a node (vertex) nearest to the source that has not been edge relaxed.

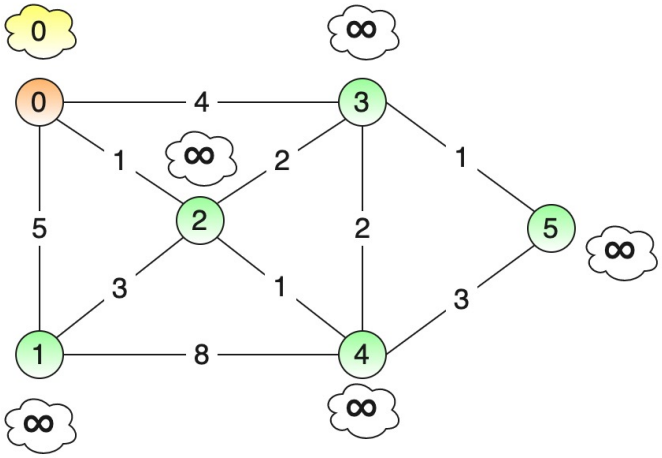


Directed graph with bi-directional edges and positive edge weights

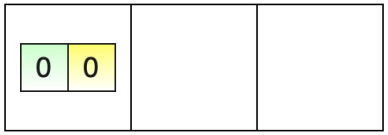


Adjacency list representation for storing the graph

Dijkstra's Algorithm

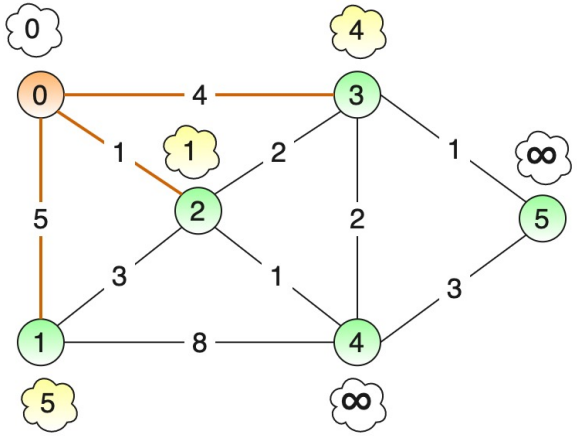


Source Node	Destination Node	Distance from single source (Node 0)
0	0	0
0	1	∞
0	2	∞
0	3	∞
0	4	∞
0	5	∞

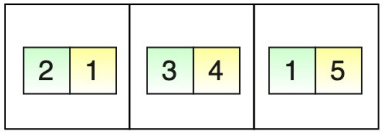


Priority queue

Dijkstra's Algorithm

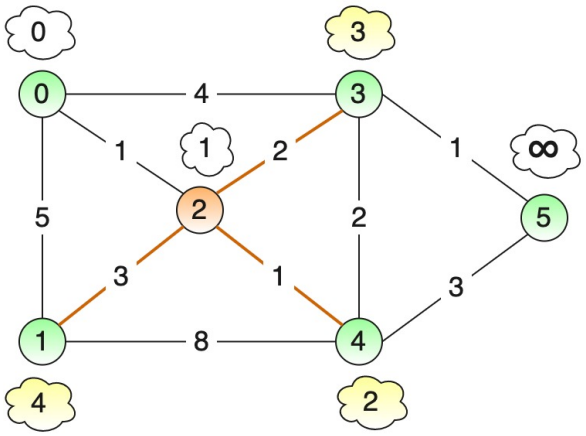


Current source	Destination	Distance form original source (0)
0	0	0
0	1	5
0	2	1
0	3	4
0	4	∞
0	5	∞

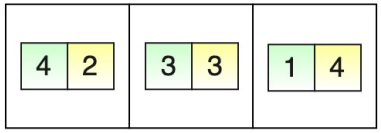


Priority Queue

Dijkstra's Algorithm

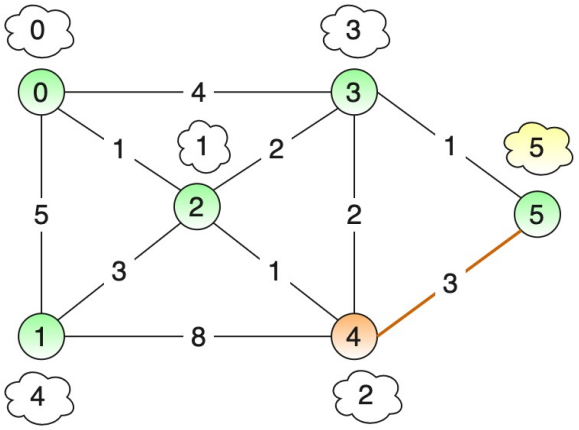


Current source	Destination	Distance from original source (0)
2	0	0
2	1	4
2	2	1
2	3	3
2	4	2
2	5	∞

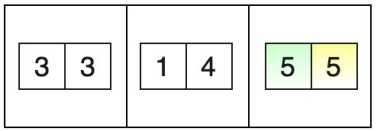


Priority Queue

Dijkstra's Algorithm

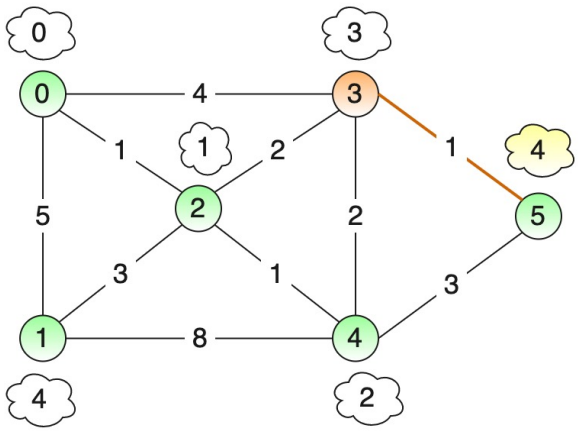


Current source	Destination	Distance from original source (0)
4	0	0
4	1	4
4	2	1
4	3	3
4	4	2
4	5	5

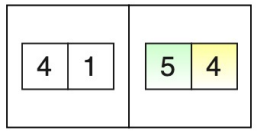


Priority Queue

Dijkstra's Algorithm

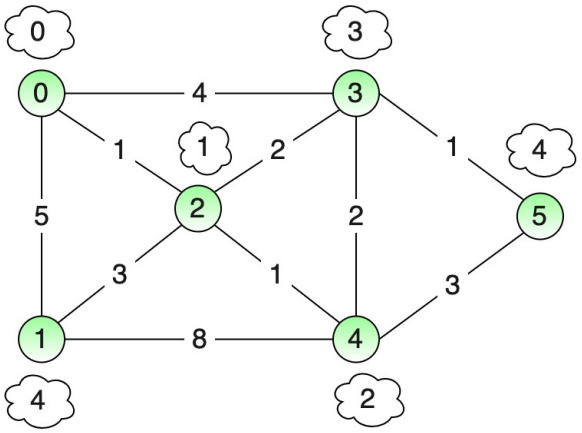


Current source	Destination	Distance from original source (0)
3	0	0
3	1	4
3	2	1
3	3	3
3	4	2
3	5	4

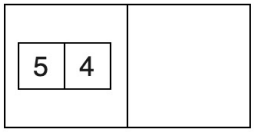


Priority Queue

Dijkstra's Algorithm

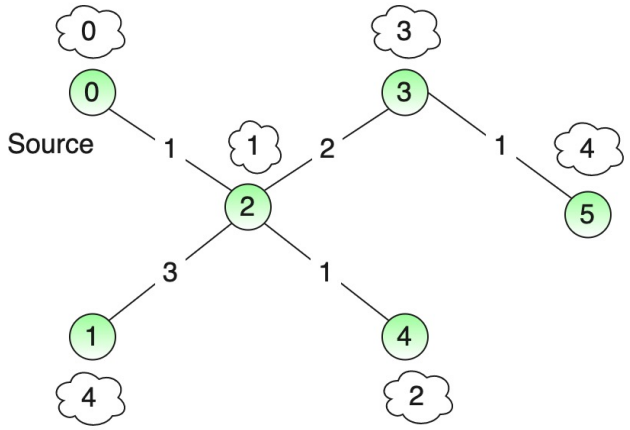


Current source	Destination	Distance from original source (0)
1	0	0
1	1	4
1	2	1
1	3	3
1	4	2
1	5	4

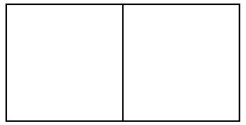


Priority Queue

Dijkstra's Algorithm



Original source	Destination	Distance from original source (0)
0	0	0
0	1	4
0	2	1
0	3	3
0	4	2
0	5	4



Priority Queue

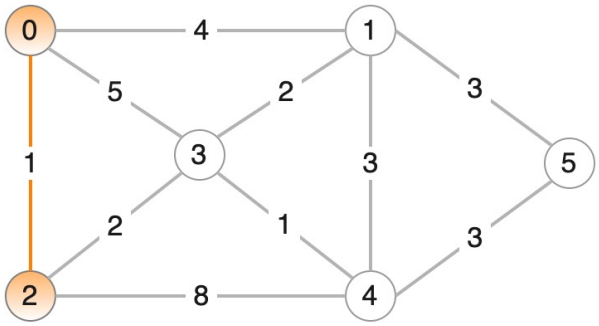
Dijkstra’s Algorithm

Algo goes here

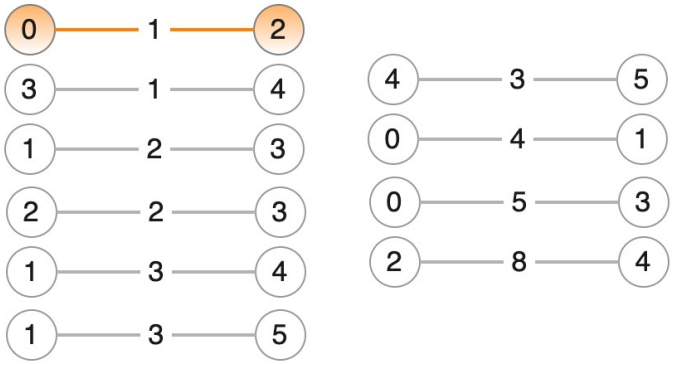
Kruskal's Algorithm

Kruskal's algorithm creates a minimum spanning tree from a **weighted undirected graph** by adding edges in increasing order of weights.
Kruskal's algorithm is **greedy** in nature as the edges are chosen in the increasing order of their weights.
The algorithm makes sure that the addition of new edges to the spanning tree does not create a cycle within it.
A Union-Find technique used for merging two disjoint subsets.

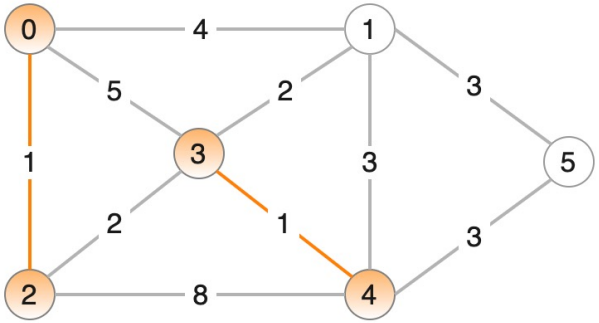
Kruskal's Algorithm

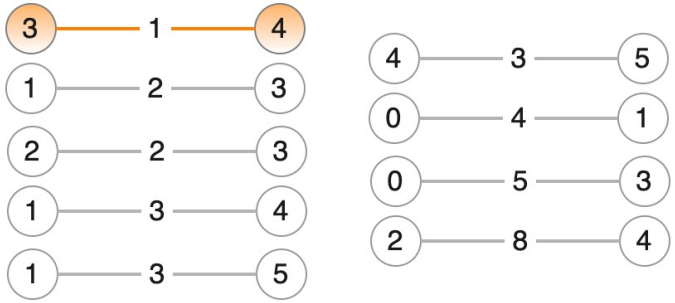


Graph 1



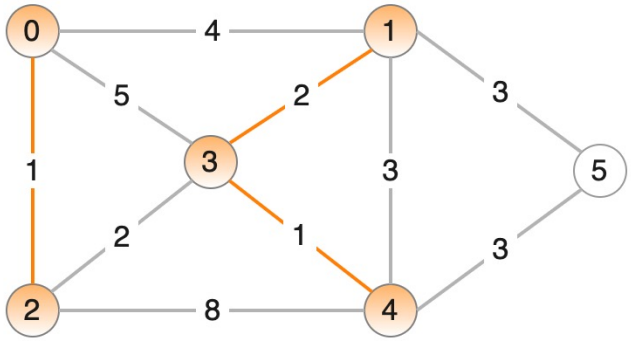
a) Sort the edges based on the cost. Pick the least expensive edge (0-2) and add it to the spanning tree.

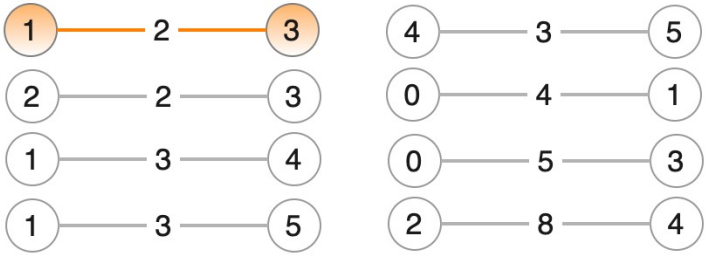




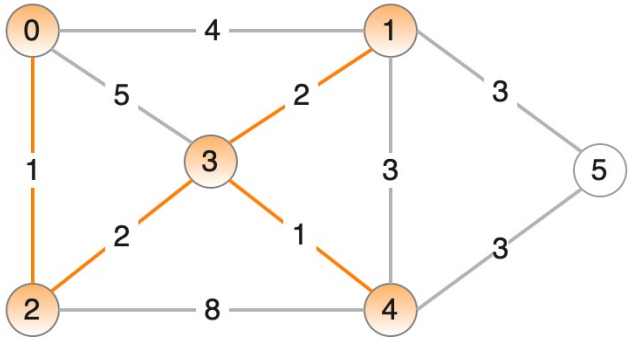
b) Pick the least expensive edge (3-4) from the available edges that do not form a cycle and add it to the spanning tree.

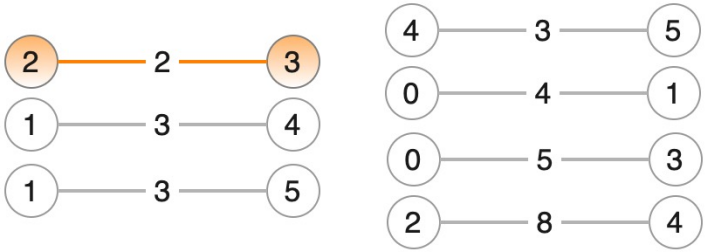
Kruskal's Algorithm





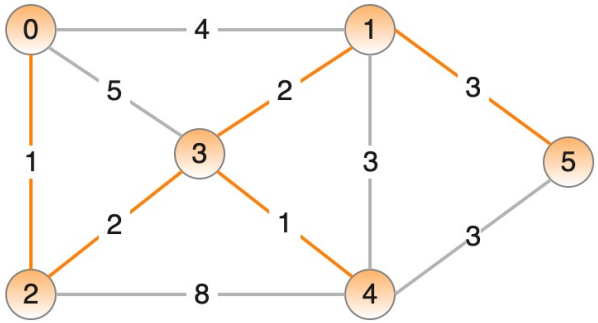
c) Similarly, pick the least expensive edge (1-3) and add it to the spanning tree.

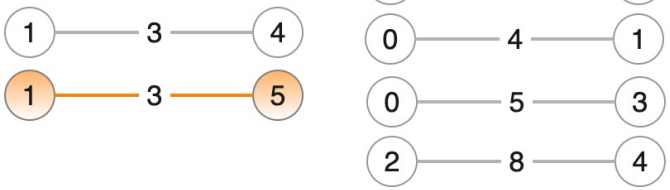




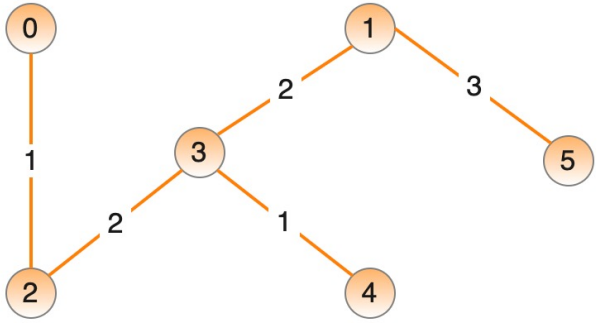
d) Similarly, pick the least expensive edge (2-3) and add it to the spanning tree.

Kruskal's Algorithm





d) Pick the least expensive edge (1-5) as it does not create a cycle and add it to the spanning tree.



e) Minimum spanning tree with edges (0-1),(3-4),(1-3),(2-3) and (1-5). Total cost: 9

Kruskal's Algorithm

Algo goes here

Prim's vs Kruskal's

incapp

	Kruskal	Prim
Multiple MSTs	Offers a good control over the resulting MST	Controlling the MST might be a little harder
Implementation	Easier to implement	Harder to implement
Requirements	Disjoint set	Priority queue
Time Complexity	$O(E \cdot log(V))$	$O(E + V \cdot log(V))$