**Breadth First Search (BFS)**

BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

BFS (G, s) //Where G is the graph and s is the source node

let Q be queue.

Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

mark s as visited.

while ( Q is not empty)

//Removing that vertex from queue, whose neighbour will be visited now

v = Q.dequeue( )

//processing all the neighbours of v

for all neighbours w of v in Graph G

if w is not visited

Q.enqueue( w ) //Stores w in Q to further visit its neighbour

mark w as visited.

**Depth First Search (DFS):**

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:  
Pick a starting node and push all its adjacent nodes into a stack.  
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.  
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

DFS-iterative (G, s): //Where G is graph and s is source vertex

let S be stack

S.push( s ) //Inserting s in stack

mark s as visited.

while ( S is not empty):

//Pop a vertex from stack to visit next

v = S.top( )

S.pop( )

//Push all the neighbours of v in stack that are not visited

for all neighbours w of v in Graph G:

if w is not visited :

S.push( w )

mark w as visited

DFS-recursive(G, s):

mark s as visited

for all neighbours w of s in Graph G:

if w is not visited:

DFS-recursive(G, w)

**Spanning Tree:**

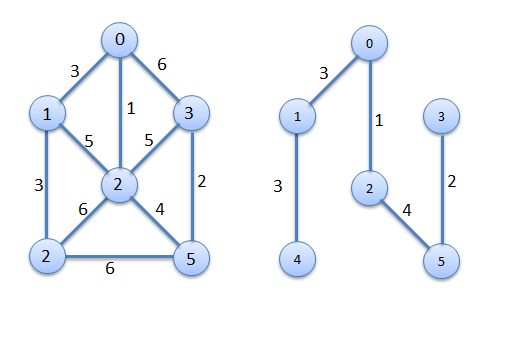
**A spanning tree is a tree that connects all the vertices of a graph with the minimum possible number of edges. Thus, a spanning tree is always connected. Also, a spanning tree never contains a cycle. A spanning tree is always defined for a graph, and it is always a subset of that graph.**

**Prim's Algorithm**

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph.

It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

This algorithm is directly based on the MST( minimum spanning tree) property.



1. MST-PRIM(G, w, r)
3. 1. for each u ­ V [G]
5. 2. do key[u] ← ∞
7. 3. π[u] ← NIL
9. 4. key[r] ← 0
11. 5. Q ← V [G]
13. 6. while Q ≠ Ø
15. 7. do u ← EXTRACT-MIN(Q)
17. 8. for each v ­ Adj[u]
19. 9. do if v ­ Q and w(u, v) < key[v]
21. 10. then π[v] ← u
23. 11. key[v] ← w(u, v)

A picture containing text, clock

Description automatically generatedA picture containing clock

Description automatically generatedA picture containing clock

Description automatically generatedA picture containing clock

Description automatically generatedA picture containing clock

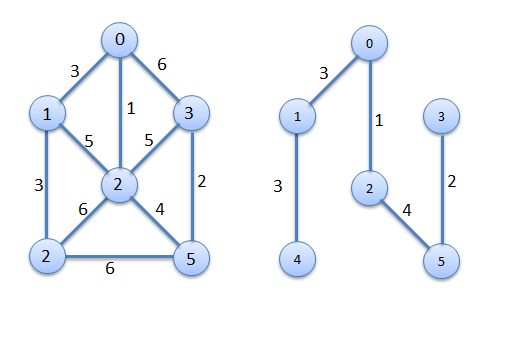
Description automatically generated

**Kruskal's Algorithm**

Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph.

It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

This algorithm is directly based on the MST( minimum spanning tree) property.



1. MST-KRUSKAL(G, w)
2. 1. A ← Ø
3. 2. for each vertex v ­ V[G]
4. 3. do MAKE-SET(v)
5. 4. sort the edges of E into nondecreasing order by weight w
6. 5. for each edge (u, v) ­ E, taken in nondecreasing order by weight
7. 6. do if FIND-SET(u) ≠ FIND-SET(v)
8. 7. then A ← A ­ {(u, v)}
9. 8. UNION(u, v)
10. 9. return A

Chart, bubble chart

Description automatically generated A picture containing clock

Description automatically generatedA picture containing clock

Description automatically generatedA picture containing clock

Description automatically generatedA picture containing clock

Description automatically generatedA picture containing clock

Description automatically generated

Min cost = 1+2+3+3+4=13s