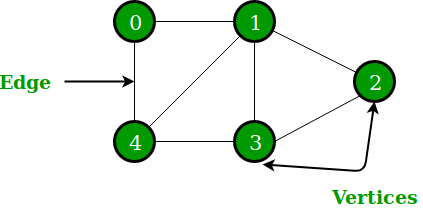
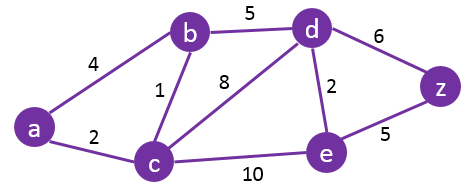
# Graphs:

**Topics covered:** Traversal techniques, topological, single Source shortest path, all path shortest path, Min spanning tree.

Graph: Pair of sets (v, e), vertices and edge.



## Why do we need Graph?



## Terminology

**Vertex**- Node;

**Edge**: Path to connect vertex;

**Un-weighted**: No weight for the edge; Weighted: Weight available;

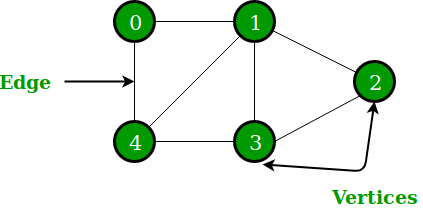
**Undirected**: No direction;

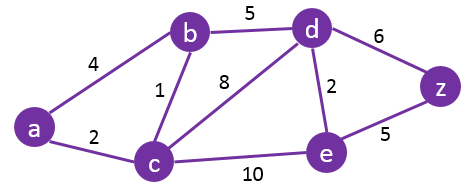
**Directed**: Directed graph;

**Cyclic**: At least one loop.

**Acyclic**: No loop.

**Tree**: Special type of graph, Directed acyclic graph (DAG).



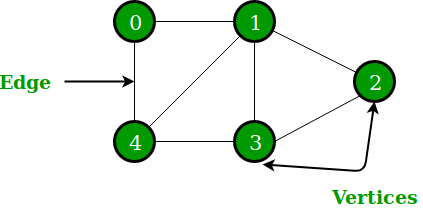


# Types of graph:

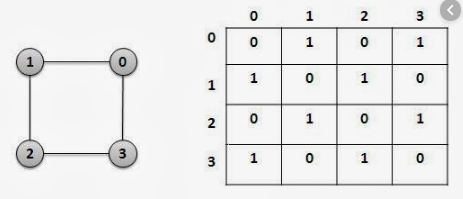
Directed and undirected.

Further into weighted and unweighted.

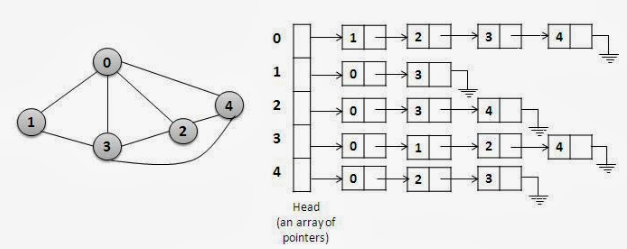
Weighted further classified to positive and negative.



How to represent a graph: Using Adjacency matrix. Mentioning the weight/Connectivity of the graph.(2D array) row and column is same and row number= no of vertex.



Adjacency list- 1D of list. Create multiple list to store the edges.



**Representation in real life:**

If graph is complete or near complete: Adjacency matric.

If the edges are few then adjacency list.

# Graph traversal:

Breadth first search and depth first search.

**BFS**: It starts from some arbitrary node f graph and explores the neighbour nodes (At same level) first before moving on to the next level neighbour.

BFS (G)

While all the vertices are not explored do:

Enqueue(Any vertices)

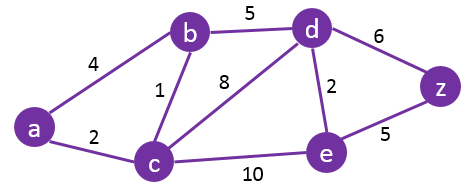
While Q is not empty

P = Dequeue()

If p is unvisited

Print p and map p as visited

Enqueue (all adjacent un-visited vortexed of p)



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

**Working**: First level, second level and third level- In a proper order.

**Note**: In a disconnected graph: BFS and DFS is not possible.

**Time complexity: O (V+E); Space complexity: O (V+E)**

**BFS (G)**

While all the vertices are not explored do: ------------------------------------------🡪 O (V)

Enqueue(Any vertices) :------------------------------------------🡪 O (1)

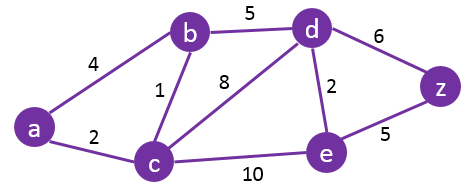
While Q is not empty: ------------------------------------------🡪 O (V)

P = Dequeue ():------------------------------------------🡪 O (1)

If p is unvisited: ------------------------------------------🡪 O (1)

Print p and map p as visited: ------------🡪 O (1)

Enqueue (all adjacent un-visited vortexed of p) ----🡪 O (Adj V)



**Un-weighted**: No weight for the edge; Weighted: Weight available;

**Undirected**: No direction;

**Directed**: Directed graph;

**Cyclic**: At least one loop.

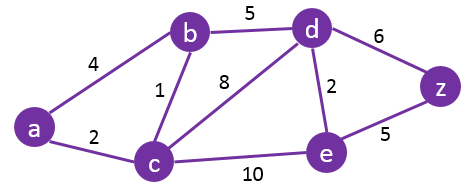
**Acyclic**: No loop.

**Advantage**: BFS has no issues with weight. It can be used in all type of graphs.

# DFS:

Starts selecting some arbitrary node and explores as far as possible along with the edges before back tracking.

All level, then second level and third level.



DFS()

While all the vertices are not explored do: ------------------------------------------🡪 O (V)

push (any vertices) ------------------------------------------🡪 O (1)

While stack is not empty----------------------🡪 O (V)

p= pop () ------------------------------------------🡪 O (1)

If p is unvisited -----------------------------------------------🡪 O (1)

Print p and mark p as visited --------------------🡪 O (1)

Push (all visited adjacent vertices as p) -------🡪 O (Adj V)

Time complexity: O (V+E) Space: O(V+E)

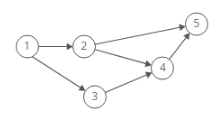
**Advantage**: BFS has no issues with weight. It can be used in all type of graphs.

# Difference b/w BFS and DFS:

|  |  |  |
| --- | --- | --- |
| **Type** | **BFS** | **DFS** |
| **Technique** | Level order – Breath first | Depth first |
| **Internal usage** | Queue | Stack |
| **Time** | O (V + E) | O (V + E) |
| **Space** | O (V + E) | O (V + E) |
| **Advantage** | When target is close to vertex | When target is further |
| **Disadvantage** | Disconnected graph | Disconnected graph |

# Topological Sort:

Parent action should be given the priority over the dependency.



|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Explanation:

Push stack ()

If (Vertex is dependent)

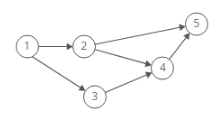
Go to its parent vertex

Then come back to the current dependent

Else push current vertex into the stack.

Pop stack ()

# Topological Sort:



|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

**Algorithm:**

Topological (G)

For all the nodes:

If vertex is not visited

Recursive (V)

Pop stack

Recursive (V)

For each neighbour of current node

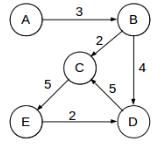
If neighbour is not visited

Recursive (visit neighbour)

Mark current node as visited and push node in stack

# Single source shortest path:

It is finding best path from one node of a graph to another in minimum/shortest distance



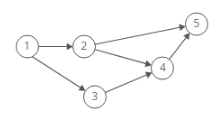
Finding the best path for each Vertex from a point.

BFS, **DFS**, Dijkstra, Bellman Ford.

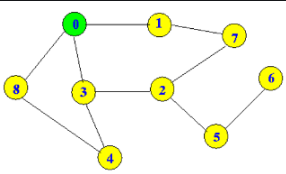
**BFS (G)**

Graph traversal technique. It starts at an arbitrary node and explores the neighbour nodes before moving to the next level neighbour nodes.

Only change from the BFS is that every node should have a parent node parameter at each node to know path of the track.



**BFS- Algorithm:**



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

While all the vertices are not explored do: ------------------------------------------🡪 O (V)

Initialize a queue ------------------------------------------🡪 O (1)

Create parent reference in each node ------------------------🡪 O (1)

Enqueue (Source vertex):------------------------------------------🡪 O (1)

While Q is not empty: ------------------------------------------🡪 O (V)

CurrentVertex = Dequeue ():------------------------------------------🡪 O (1)

For each adjacent vertices ------------------------------------------🡪 O (Adj V)

If adjacent vertex is visited;

Don’t do anything

Else:

CurrentVertex is unvisited: ------------------------------------------🡪 O (1)

Enqueue all adjacent vertex and update parent as source vertex and mark current vertex as visited----🡪 O (Adj V)

**Time O (E); Space: O (E)**

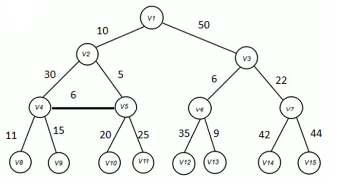
Time is efficient because we are checking for connected graph.

# Why disconnected and weighted graph do not work in BFS and DFS:

Only Unweighted graph can be used using BFS.

**Why we can’t use in Weighted Graph:**

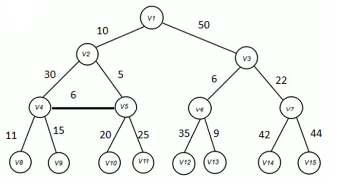
The logic of BFS do not support better route. It just follows a traversal.



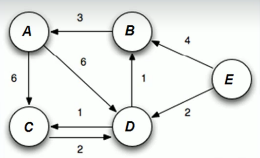
Dis-advantage: If weighted we cannot use the BFS, it is not the optimum.

**DFS:**

All the characteristic were a disadvantage.



# Dijkstra:



Dijkstra (G)

Set the distance from all the vertexes as infinite and root as 0 ----🡪 O (V)

Save all the vertices in min-heap ----🡪 O (V)

Do until min-heap is not empty ----🡪 O (V)

Current vertex=extract from Min-heap ----🡪 O (1)

for each neighbour of current vertex ----🡪 O (V)

If current vertex distance + current edge < neighbours distance🡪 O (V)

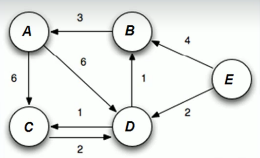
Update neighbours distance and parent ----🡪 O (1)

Time complexity – O (V^2); Space complexity - O(E)

**Dis-advantage**: Can’t work in negative cycle, because it’s not possible, we can do the same cycle again and can never find a shortest path.

Bellman ford algorithm:

It is used find SSSP. Graph contains a negative cycle. A cycle whose edges sum to negative value that is reachable from the source, then there is no cheapest path. Any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case bellman can detect negative cycles and report their existence.



**Weight**: Stores the weight of all the edges.

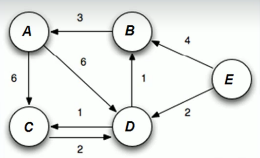
A -> C = 6 A -> D = 6 C-> D = 2 D -> C = 1 D->B = 1

B->A = 3 E -> B = 4 E-> D = 2

**Distance Matrix:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Vertex | Distance | Iter -1 Distance | Parent | Iter -2 Distance | Parent | Iter – 3 Distance | Parent | Final | Parent |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

**Algorithm:**



BellmanFord (D)

Set all the vertex distance to Infinity and source as 0

For 1 to V-1: -------🡪 O (V)

For each edge (u, v) -------🡪 O (E)

If D (V) > d (u) + w (u, v) -------🡪 O (1)

D (V) = d (u) + w (u, v) -------🡪 O (1)

Update parent of V -------🡪 O (1)

For each edge (u, v)

D (V) != d (u) + w (u, v)

Then report negative cycle existence.

**Time Complexity = O (EV); Space Complexity = O (V)**

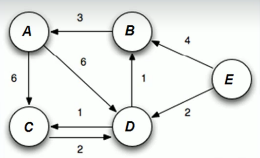
# How Negative Cycle Works In Bellman-Ford:

**Why it runs for V-1:**

**How it works internally:**

In every Iteration it checks if nodes has achieved better distance.

Then, it tries in current step it achieves better distance and improve distance with other vertices.



**Worst Case:**

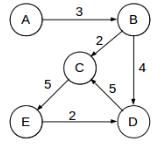


**What happens in Vth Iteration?**

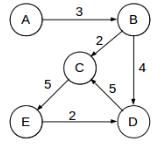
All pair shortest path: Best path for all the vertices.

# All Pair shortest Path Algorithm:

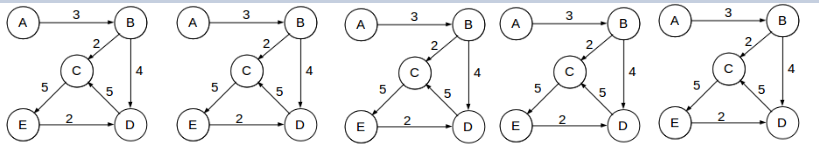
In this algorithm, we will have every vertex as a source vertex and calculate the minimum distance.

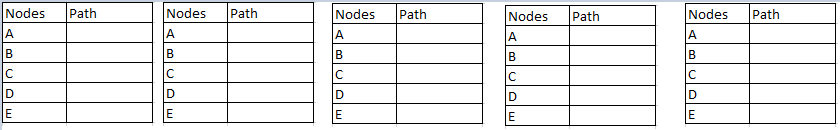


Difference between Single source shortest Path algorithm and APSP:

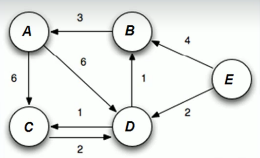


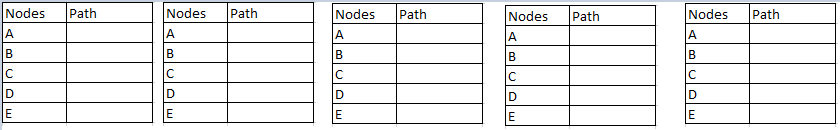
What can be used to solve APSP Problem?





# Dry run of all pair shortest pair Algorithm:

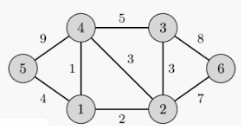




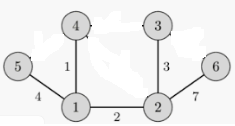
# Minimum Spanning tree:

Sub-set of edges of a connected, weighted and undirected graph.

**Task**: Connect vertices together, without any cycles, min total edge weight.



**Total Weight**: 39



**Minimum Weight:** 17

**Confusion with Minimum Spanning tree and SSSP Explanation**

# Disjoint set:

It keeps track of the set of nodes which is portioned into a number of disjoint and non-overlapping sets.

Each set as a representative, helps in identifying the set.

## Why we need to learn Dis-joint set?

Used for several algorithm and it helps in solving minimum spanning tree.

## Standard operation of Disjoint set?

Make set, Union, Find set.

**Make set**: Create a set ---🡪 O (N)

**Union (S1, S2):** -🡪 O (N)

If they are in same set,

Then return.

Else:

If S1 is bigger, merge S2 with S1.

Else: merge S2 into S1.

Return merged set.

**Find set(X):** -🡪 O (1)

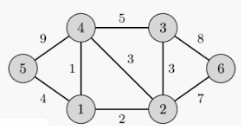
Return representative of set in which element x is available.

# Krusal Algorithm:

Greedy algorithm. Finds min spanning tree for a connected weight graph.

By adding increasing cost arcs at each step.

Avoids cycle in every step.



MST-Krusal (G)

For each vertices. Makeset (X) --🡪 O (V)

Sort each edge in nondecreasing order by weight. --🡪 O (E Log V)

For each edge (u, v) do: --🡪 O (E)

If findset u != findset (v) --🡪 O (1)

Union (u, v) --🡪 O (V)

sort = cost +edge(u, v) --🡪 O (1)

**Time Complexity = O(V + E Log E + E.V) =O (E log V) Space Complexity = O(E)**

# Prims Algorithm:

Greedy algorithm.

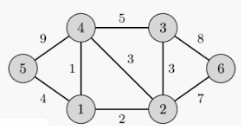
Take any vertex and mark its weight as 0 and rest as Infinite.

For every adjacent unvisited vertex, if current weight is more adjacent vertex edge, update the current weight with adjacent vertex’s edge.

Mark current vertex as visited.

Do above steps in increasing order of weight.

Print all the vertex with weight.



**Prims Algorithm:**

MST-Prim

Create a priority Queue ---🡪 O (1)

Insert all the vertex into the Q, Starting value is 0 and others will be infinity. ---🡪 O (V)

While Q is empty ---🡪 O (V)

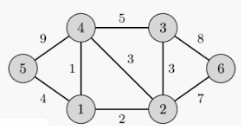
Currntvertex = dequeuer(Q) ---🡪 O (1)

For every adjacent unvisited vertex of currentVertex ---🡪 O (V)

If current weight > adjacent weight, current weight= adjacent weight ---🡪 O (Log V)

Mark current vertex as visited ---🡪 O (1)

Print all vertex with weight ---🡪 O (V)



**Time Complexity = O (V2 Log V) = O (E Log V)**

**Space Complexity = O (V)**