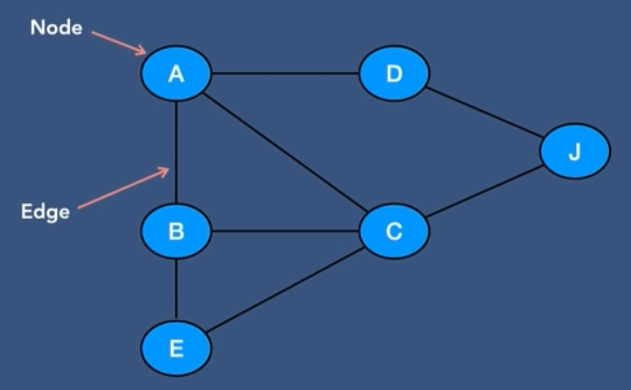
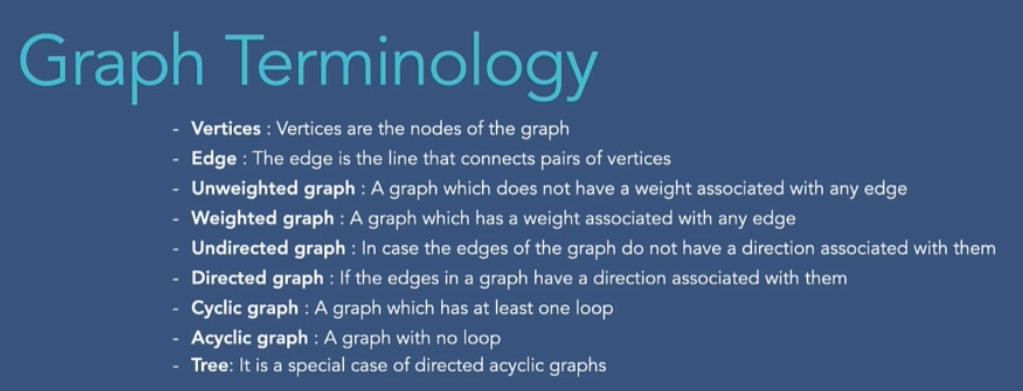
**Graph:** Graph is a finite set of vertices (nodes) and set of edges which connect a pair of vertices. Graphs are used for many real-life problems. It represents a network. Network may represent path of a city, telephone network, electric circuit. It is also used in social networks for connecting different persons (vertices).





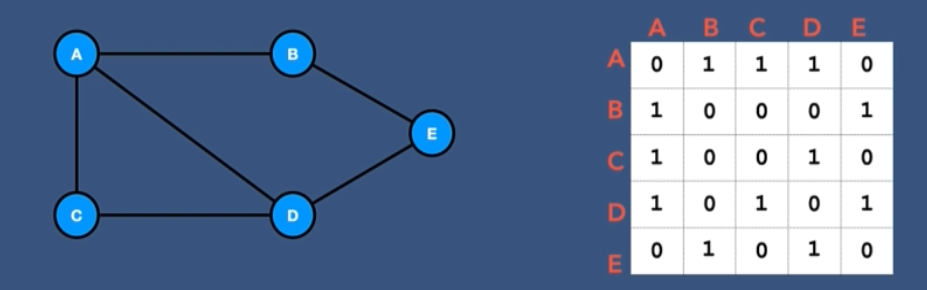
**Tree vs Graph:**

Tree is a special case of directed acyclic graph. Main difference is that tree don’t have cycles while graph can have cycle. Another difference is in tree we have a root node but in graph there is no root node

**Graph Representation:**

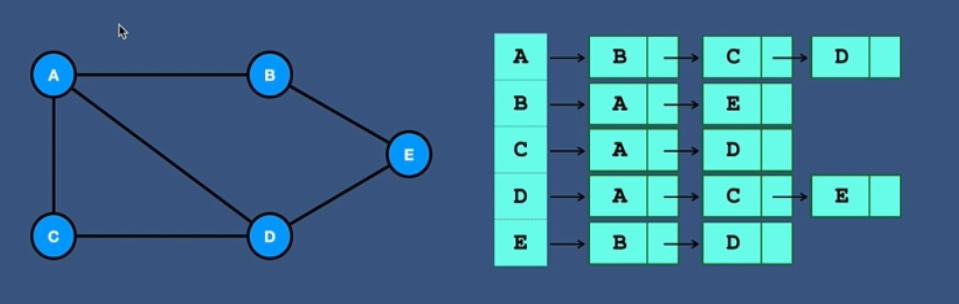
1. **Adjacency matrix**: This is a 2D array which indicates connection between pairs.

* Uses O(n^2) memory
* It is fast to look up and check for presence or absence of a specific edge  
  between any two nodes O (1)
* It is slow to iterate over all edges
* It is slow to add/delete a node; a complex operation O(n^2)
* It is fast to add a new edge O (1)



1. **Adjacency list**: It is a collection of unordered lists that represent a graph. Each list describes the set of neighbors of a vertex in the graph.

* Memory usage depends more on the number of edges (and less on the number of nodes), which might save a lot of memory if the adjacency matrix is sparse
* Finding the presence or absence of specific edge between any two nodes is slightly slower than with the matrix O(k); where k is the number of neighbors nodes
* It is fast to iterate over all edges because you can access any node neighbors directly
* It is fast to add/delete a node; easier than the matrix representation
* It fast to add a new edge O (1)

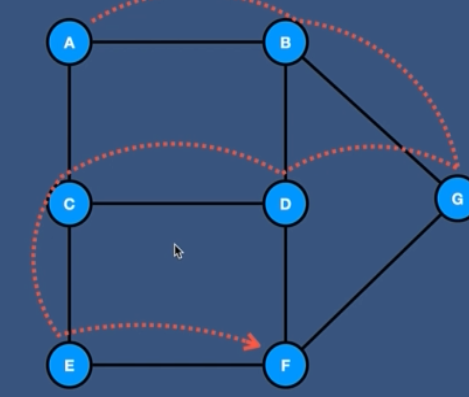
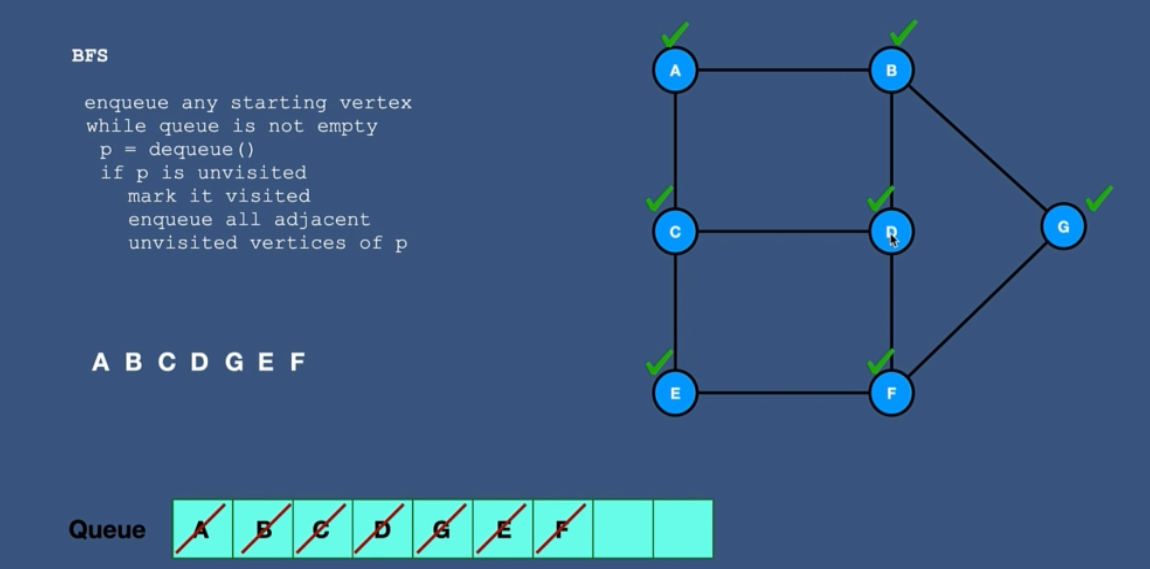


**Which one to use?**

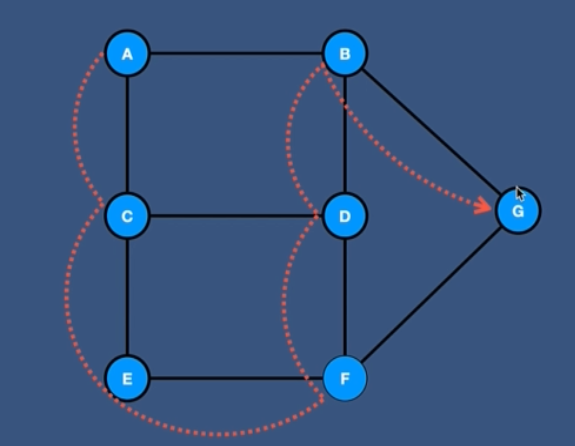
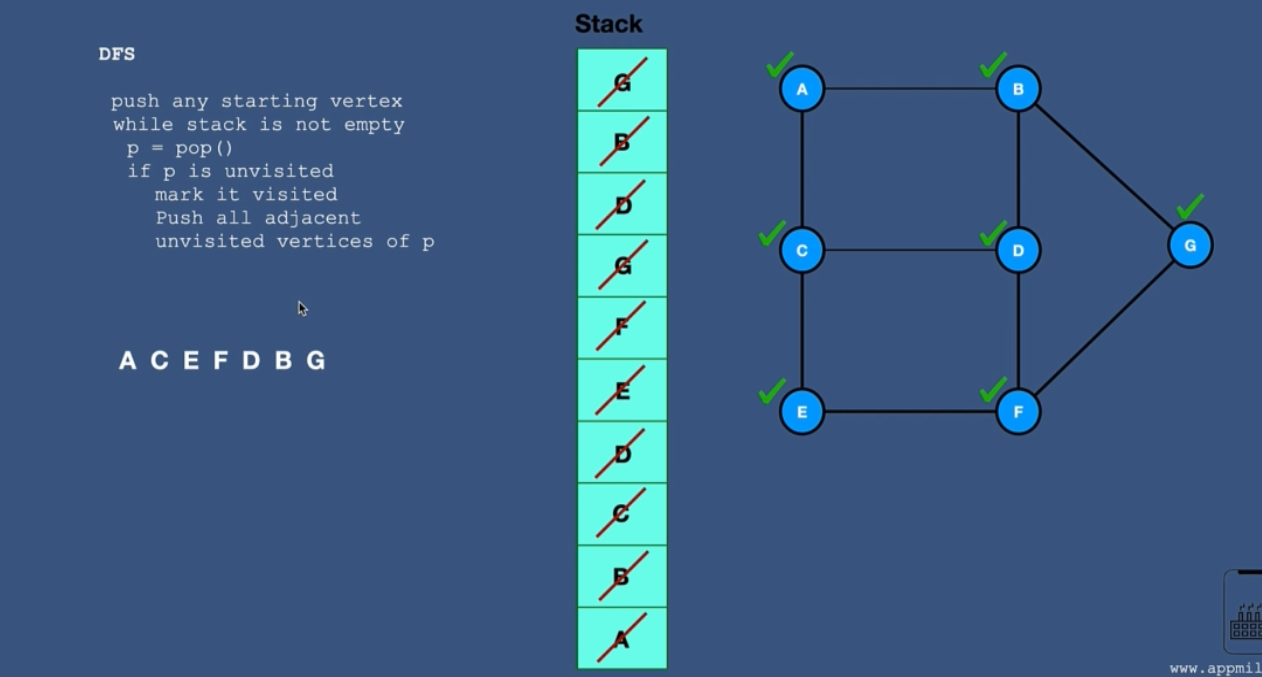
* For complete or almost complete graph we should use adjacency matrix as it will use the matrix efficiently.
* If the number of edges is low, we should use Adjacency list.

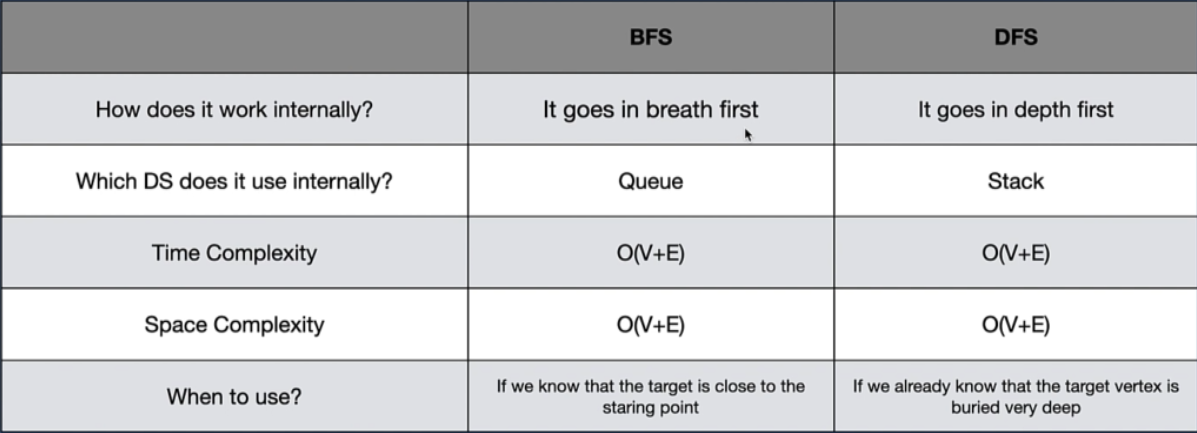
**Graph Traversal:** It is the process of visiting each graph.

1. **Breadth First Search (BFS):** This is an algorithm for traversing graph data structure. It starts at some arbitrary node of graph and explore the neighbor nodes first. Then it moves to next level neighbors.

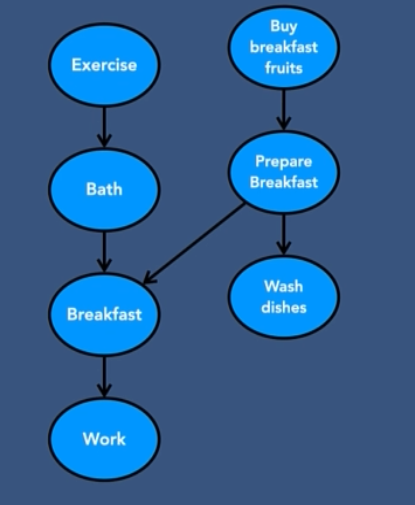
 

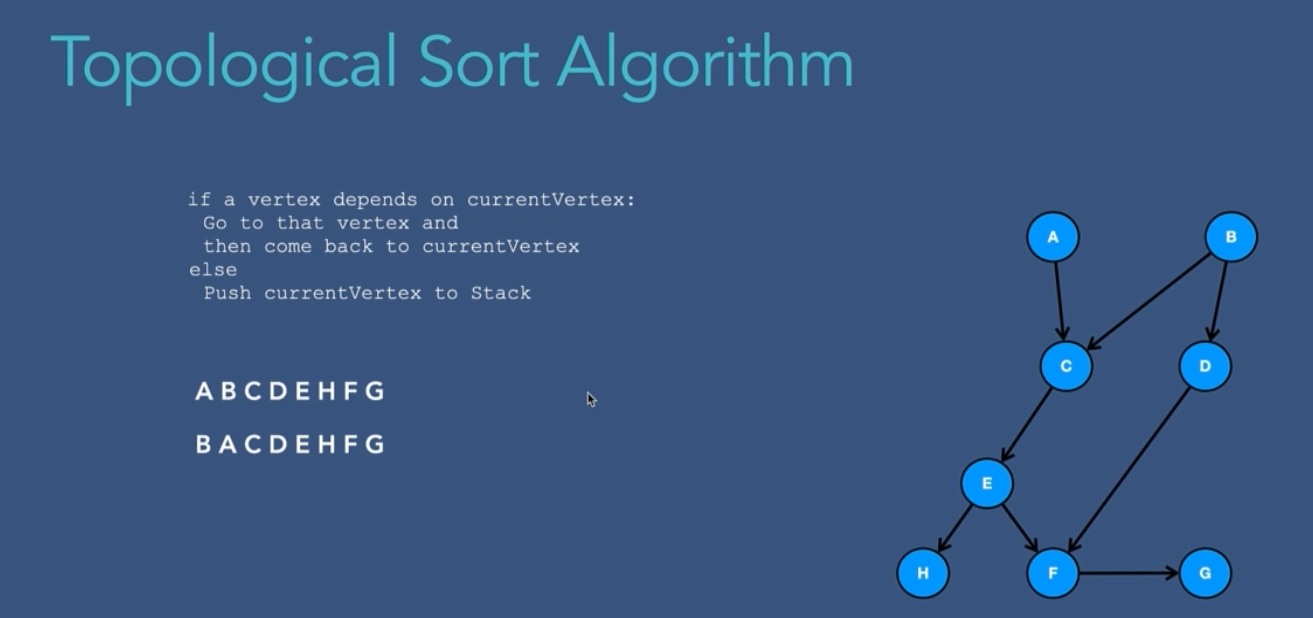
1. **Depth First Search (DFS):** This is an algorithm for traversing graph data structure. It starts at some arbitrary node of graph and explores as far as possible along each edge before backtracking.



**Topological sort:** If there is dependency of one action (for example, Bath will always come after exercise), then the dependent action (Bath) will always come after the parent (Exercise) action

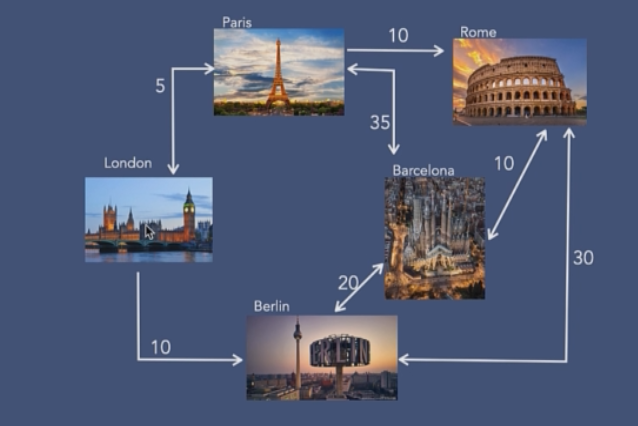
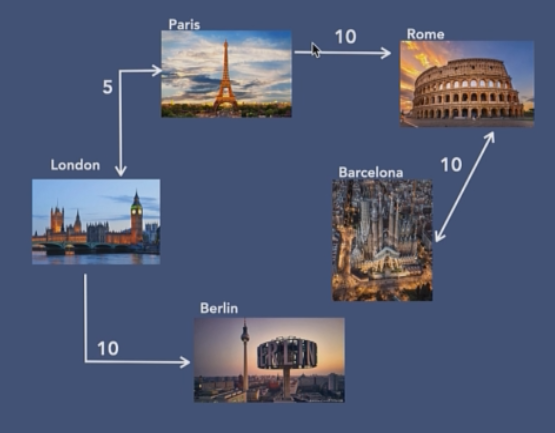
****

****

**Time and space complexity:** O (E + V)

**Single Source Shortest Path problem (SSSP):** This problem is about finding a path between a given vertex (source) and to all other vertices in the graph such that the total distance between them (source to destination) is minimum.

For example, let’s say there are five offices in five different cities. The travel cost between each cites are known. We need to find the cheapest way from head office to branches.

First image shows all possible routes. Second image shows the shortest possible routes.

There are basically three main algorithms for solving this problem

1. **BFS:** For BFS we can find shortest path for **un-weighted graph** only. Being unweighted adjacency is always shortest path to any adjacent node.

BFS first visits all nodes at distance 1 from the starting node. Then all nodes at distance 2. Then all nodes at distance 3, etc. So, BFS finds the shortest path (in terms of number of edges) to a goal node, because the shortest path, definitionally, is the least-distance path. It cannot find a longer path because that would imply that it was visiting nodes out of distance order.

The time complexity of this algorithm is O(E), it is the number of edges.

**Why BFS doesn’t work for weighted graph?** Because it does not take into accounts the weight of the edges in the graph. It simply explores each nearest levels without considering weights or order. So, we will not get accurate results.

**Why DFS doesn’t work with SSSPP?** DFS tends to go furthest levels from source. So it can never find the shortest path.

1. **Dijkstra algorithm:** As BFS doesn’t work with weighted graph, we can use Dijkstra algorithm to find shortest path when edges have **non-negative weight**.

* **Initialization**: This algorithm set the startNode distance to zero and all other nodes distance to infinity initially.
* **Visit the neighbors:** Consider all the neighbors of the current vertex. Calculate the ***tentative distance = weight of the edge from current vertex to neighbour vertex + current vertex distance value***
* **Update distances:** if the tentative distance is smaller then the current distance of the neighbor, update the neighbor’s distance value with the tentative distance.
* **Mark the current vertex as visited:** Once all the neighbors are updated, mark the current vertex as visited to ensure it is not visited again.
* **Select the next vertex:** Now from the unvisited vertices, select the vertex with smallest distance value as the next current vertex and repeat step 2,3,4.
* **Repeat steps 2 to 5:** continue this until all vertices are marked visited or until the destination is reached.
* **Shortest path extraction:** finally. we can find the shortest distance by tracking the smallest distance neighbors.

**Limitations:** Dijkstra’s algorithm doesn’t work under two conditions:

1. Graph has a cycle in it
2. The total distance of the cycle is negative.

If this happens, then Dijkstra’s algorithm fails. Since everytime it will try to update the distance of that cycle, the distance will keep getting smaller as it is subtracting a negative number. So, it will have an infinite loop.

1. **Bellman-Ford algorithm:**