Graph Data Structure & Algorithms

Sunbeam Infotech

Agenda

- Graph terminologies & types
- Graph implementation – Adjacency Matrix & Adjacency List
- Breadth First & Depth First Search/Traversal
- BFS & DFS Spanning Tree
- Check connected graph
- Check bi-partite graph
- Single source path length algorithm
- Greedy approach
  - Kruskal MST algorithm
  - Prim's MST algorithm
  - Dijkstra's Shortest Path algorithm
- Dynamic programming
  - Optimizing recursion
  - Dynamic programming
  - Bellman Ford Algorithm
  - Floyd-Warshall Algorithm
- A* Search Algorithm

Graph

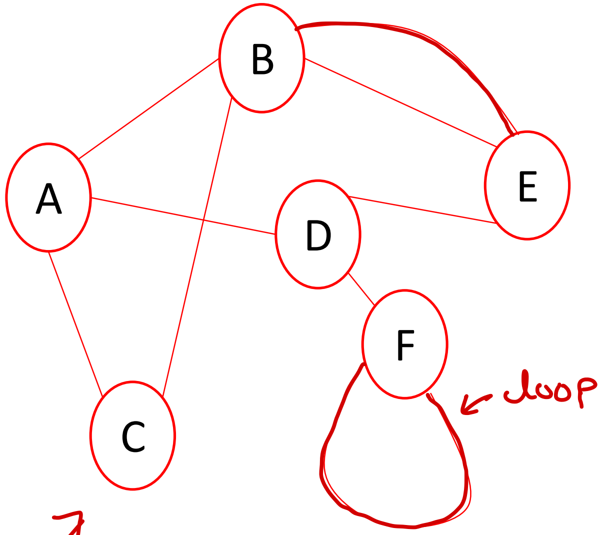
$G = \{V, E\}$ $V = \{A, B, C, D, E, F\}$

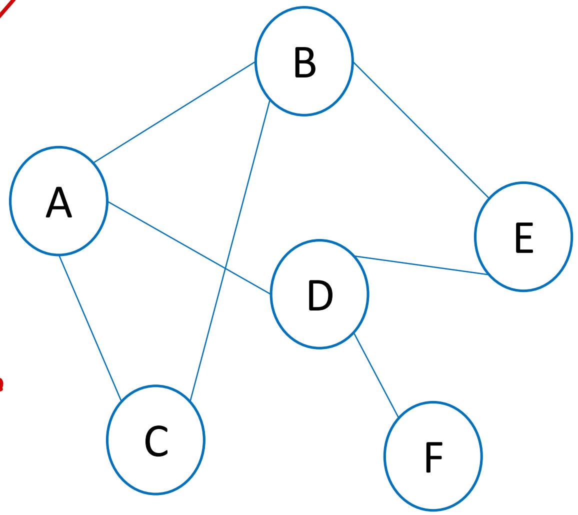
$E = \{(A,B), (A,C), (A,D), (B,C), (B,E), (D,E), (D,F), (B,E) (F,F)\}$

- Graph is a non-linear data structure.
- Graph is defined as set of vertices and edges. Vertices (also called as nodes) hold data, while edges connect vertices and represent relations between them.
  - G = { V, E }
- Vertices hold the data and Edges represents relation between vertices.
- When there is an edge from vertex P to vertex Q, P is said to be adjacent to Q.

$P \rightarrow Q$

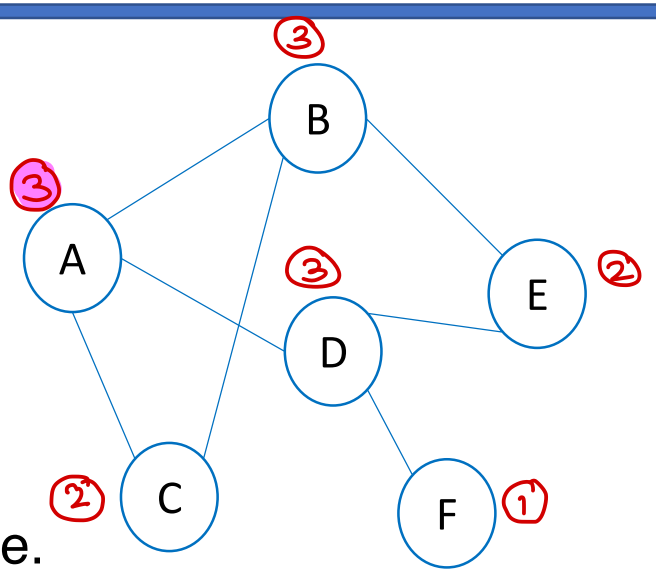
- Multi-graph
  - Contains multiple edges in adjacent vertices or loops (edge connecting a vertex to it-self).
- Simple graph
  - Doesn't contain multiple edges in adjacent vertices or loops.



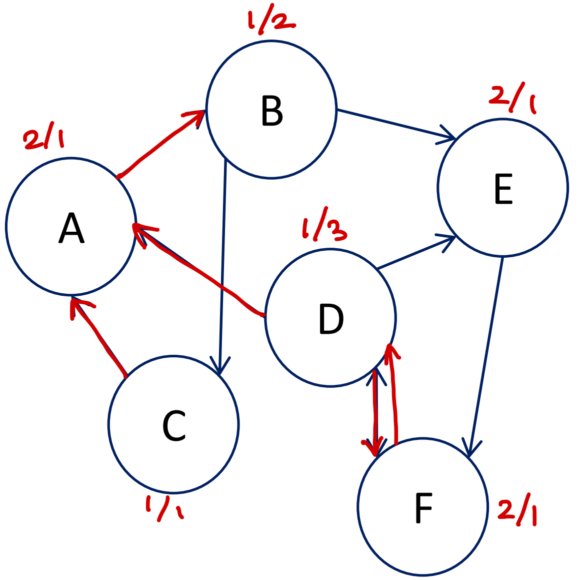


Graph

- Graph edges may or may not have directions.



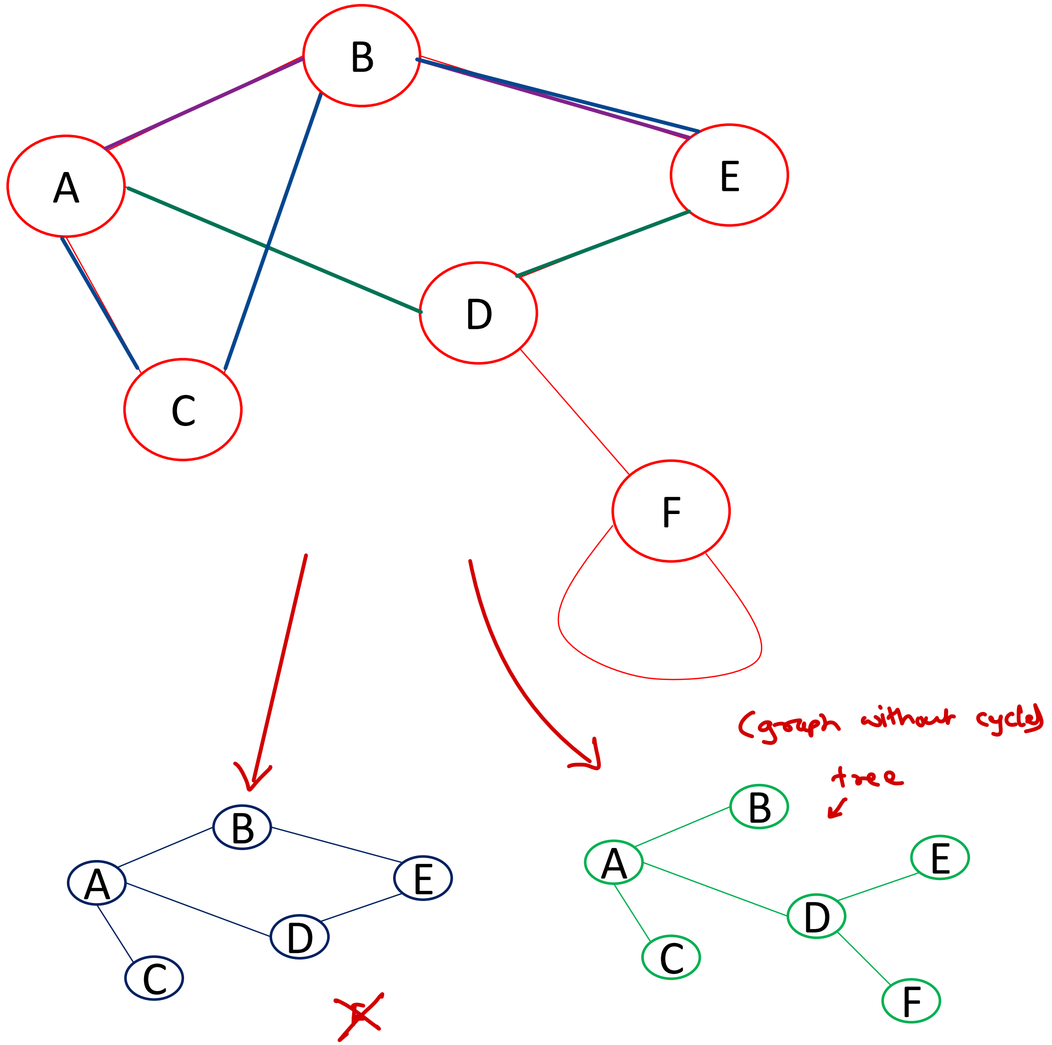
- Undirected Graph: G = { V, E }
  - V = { A, B, C, D, E, F}
  - E = { (A,B), (A,C), (A,D), (B,C), (B,E), (D,E), (D,F) }
  - If P is adjacent to Q, then Q is also adjacent to P.
  - Degree of node: Number of nodes adjacent to the node.
  - Degree of graph: Maximum degree of any node in graph.



- Directed Graph: G = { V, E }
  - V = { A, B, C, D, E, F}
  - E = {<A,B>, <B,C>, <B,E>, <C,A>, <D,A>, <D,E>, <D,F>, <E,F>, <F,D>}
  - If P is adjacent to Q, then Q is may or may not be adjacent to P.
  - Out-degree: Number of edges originated from the node
  - In-degree: Number of edges terminated on the node

Graph

- Path: Set of edges between two vertices. There can be multiple paths between two vertices.
    - A – D – E
    - A – B – E
    - A – C – B – E
- Cycle: Path whose start and end vertex is same.
    - A – B – C – A
    - A – B – E – D – A
- Loop: Edge connecting vertex to itself. It is smallest cycle.
    - F – F
- Sub-Graph: A graph having few vertices and few edges in the given graph, is said to be sub-graph of given graph.



Graph

- Weighted graph
  - Graph edges have weight associated with them.
  - Weight represent some value e.g. distance, resistance.
- Directed Weighted graph (Network)
  - Graph edges have directions as well as weights.
- Applications of graph
  - Electronic circuits
  - Social media
  - Communication network
  - Road network
  - Flight/Train/Bus services
  - Bio-logical & Chemical experiments
  - Deep learning (Neural network, Tensor flow)
  - Graph databases (Neo4j)

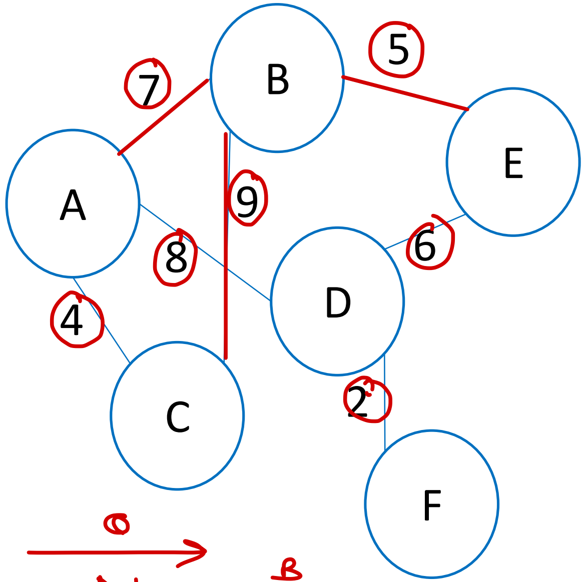
DAG → scheduling

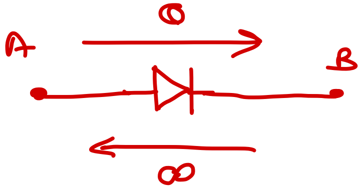
↓

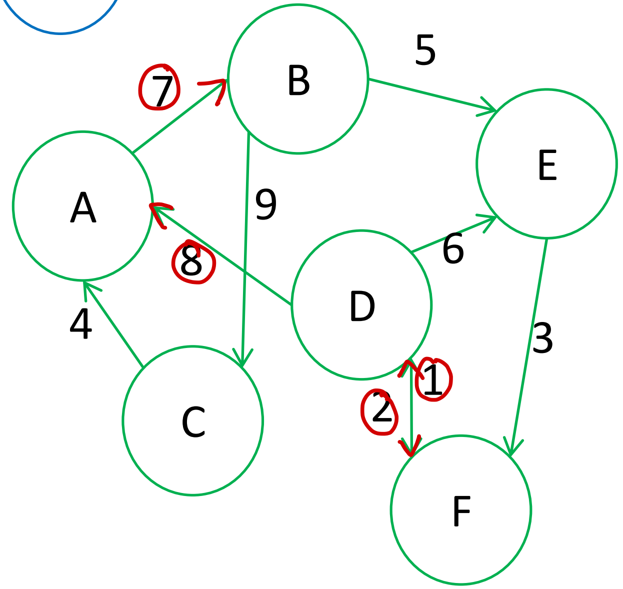
YARN

Spark

Tez

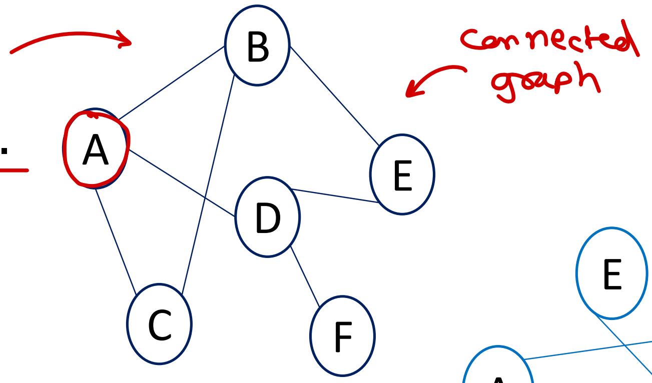


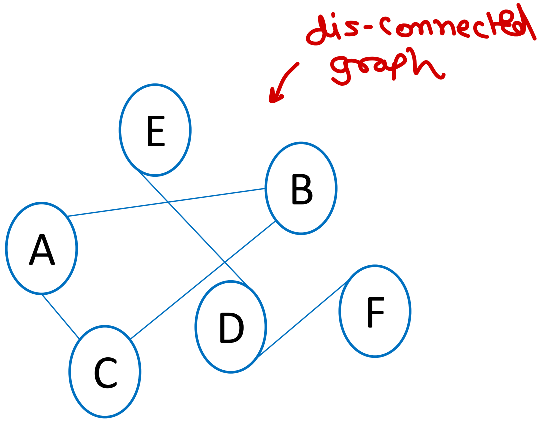




Graph

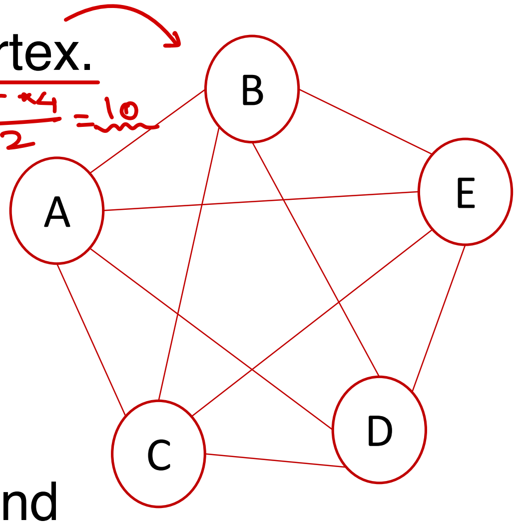
- Connected graph
  - From each vertex some path exists for every other vertex.
  - Can traverse the entire graph starting from any vertex.



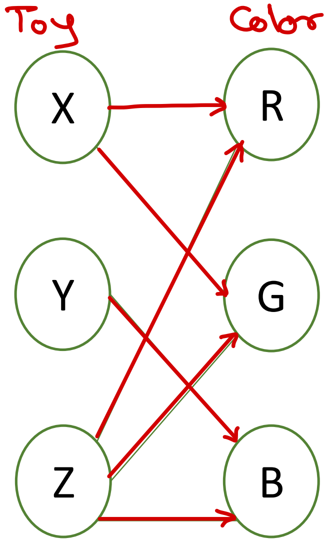


- Complete graph
  - Each vertex of a graph is adjacent to every other vertex.
  - Un-directed graph: Number of edges = n (n-1) / 2 $= \frac{5*4}{2} = 10$
  - Directed graph: Number of edges = n (n-1)

    $5*4=20$

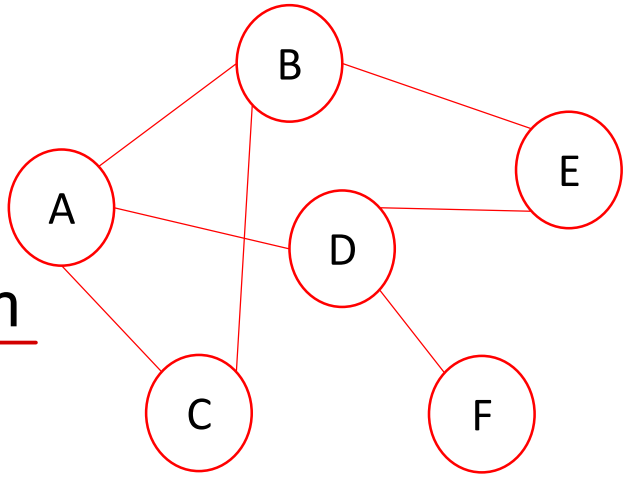


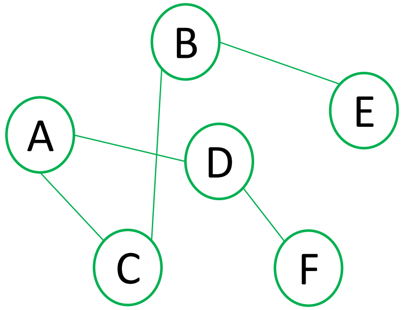
- Bi-partite graph
  - Vertices can be divided in two disjoint sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected to each other.

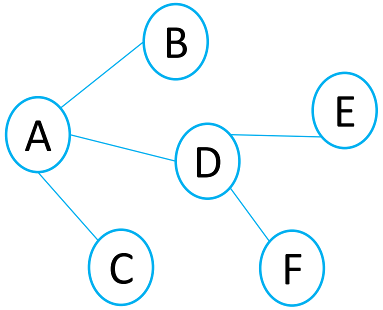


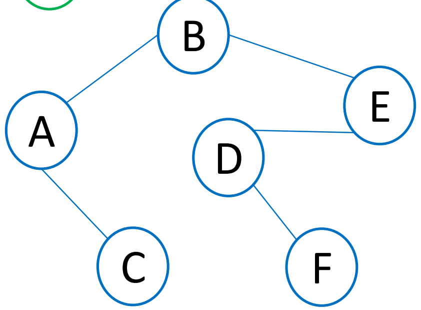
Spanning Tree

- Tree is a graph without cycles.
- Spanning tree is connected sub-graph of the given graph that contains all the vertices and sub-set of edges (V-1).
- Spanning tree can be created by removing few edges from the graph which are causing cycles to form.
- One graph can have multiple different spanning trees.
- In weighted graph, spanning tree can be made who has minimum weight (sum of weights of edges). Such spanning tree is called as Minimum Spanning Tree.
- Spanning tree can be made by various algorithms.
  - BFS Spanning tree
  - DFS Spanning tree
  - Prim's MST
  - Kruskal's MST



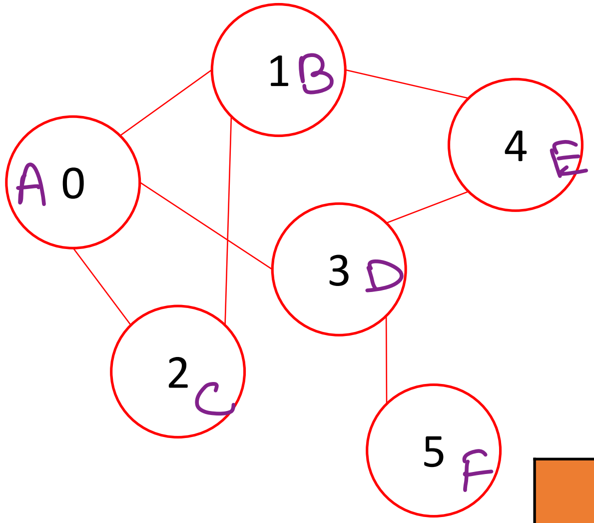






Graph Implementation – Adjacency Matrix

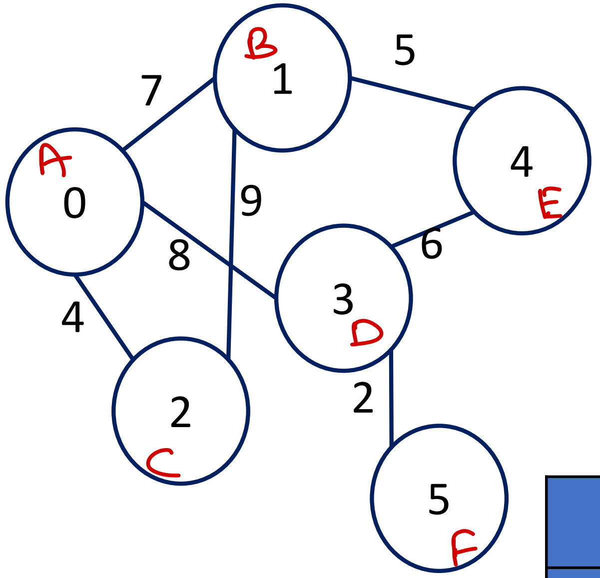
- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For *non-weighted graph*, 1 indicate edge and 0 indicate no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is $O(V^2)$.



		0	1	2	3	4	5
		A	B	C	D	E	F
0	A	0	1	1	1	0	0
1	B	1	0	1	0	1	0
2	C	1	1	0	0	0	0
3	D	1	0	0	0	1	1
4	E	0	1	0	1	0	0
5	F	0	0	0	1	0	0

Graph Implementation – Adjacency Matrix

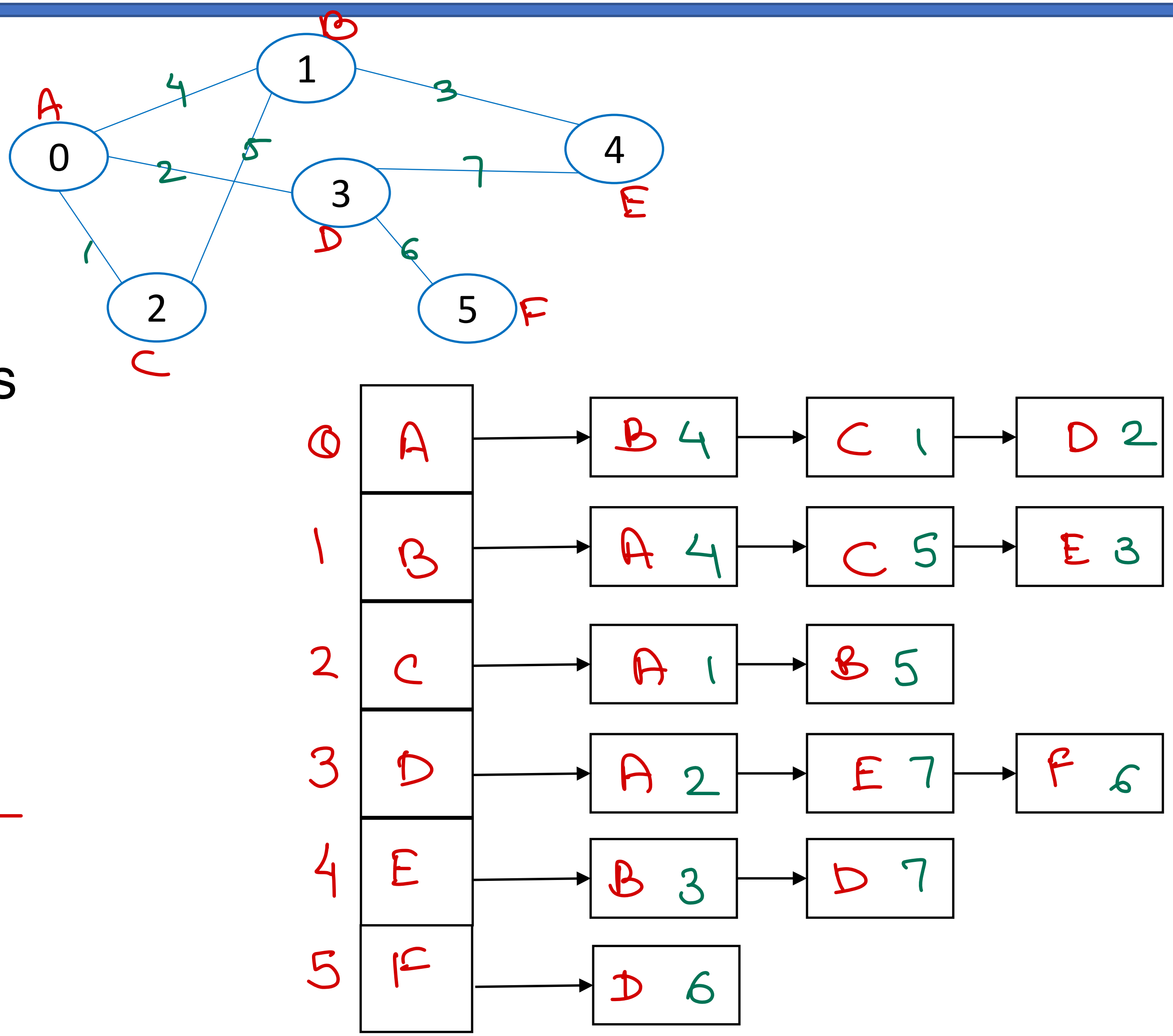
- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For *weighted graph*, weight value indicate the edge and infinity sign ∞ represent no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is $O(V^2)$.



	A	B	C	D	E	F
A	∞	7	4	8	∞	∞
B	7	∞	9	∞	5	∞
C	4	9	∞	∞	∞	∞
D	8	∞	∞	∞	6	2
E	∞	5	∞	6	∞	∞
F	∞	∞	∞	2	∞	∞

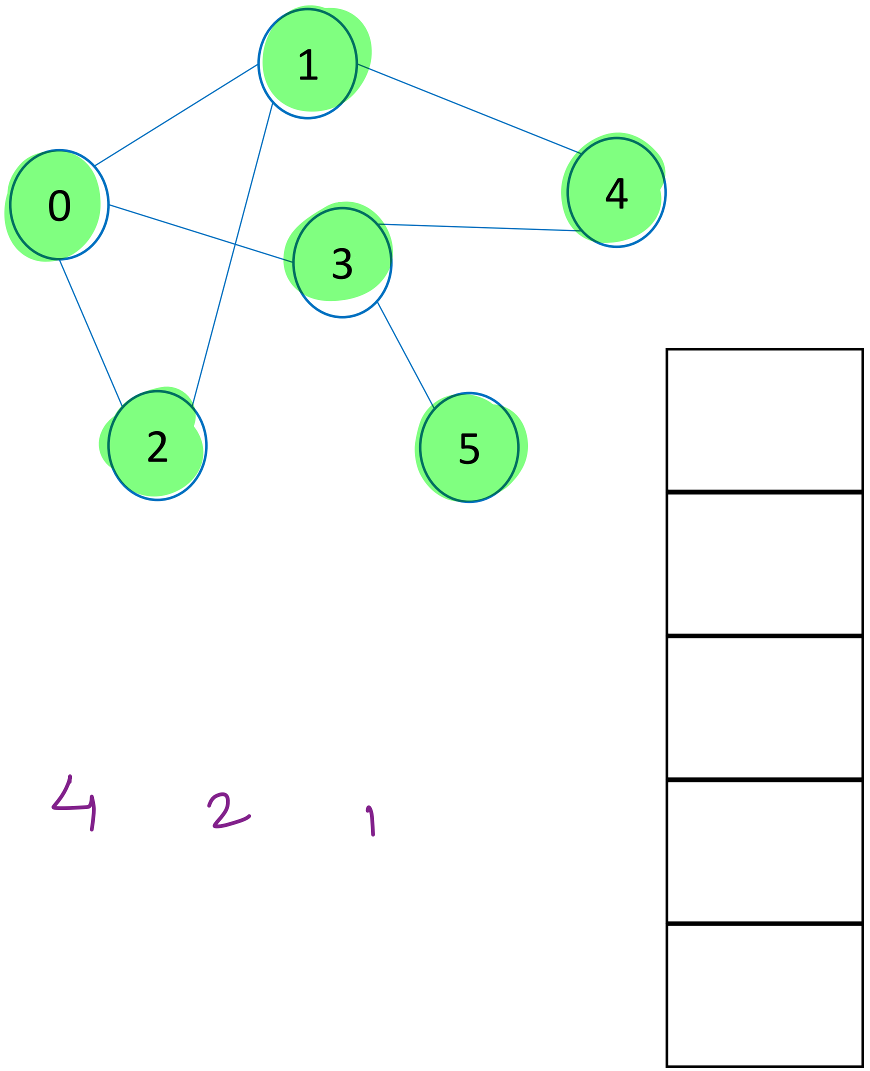
Graph Implementation – Adjacency List

- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs only, neighbour vertices are stored.
- For weighted graph, neighbour vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V*E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).



Graph Traversal – DFS Algorithm

1. Choose a vertex as start vertex.
2. Push start vertex on stack & mark it.
3. Pop vertex from stack.
4. Visit (Print) the vertex.
5. Put all non-~~visited~~ marked neighbours of the vertex on the stack and mark them.
6. Repeat 3-5 until stack is empty.



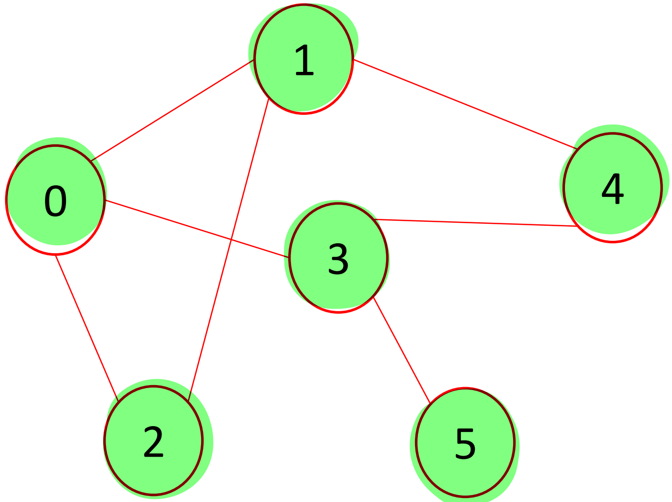
O(V * V)

0 3 5 4 2 1

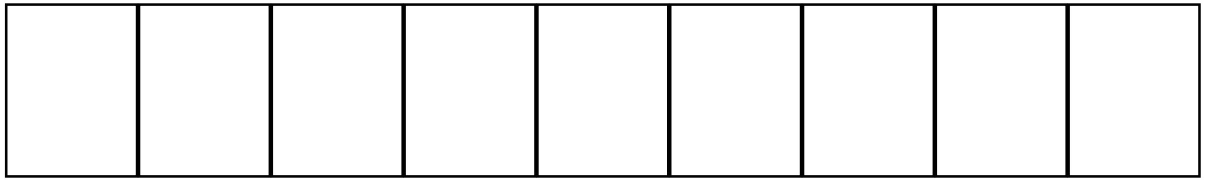
Graph Traversal – BFS Algorithm

1. Choose a vertex as start vertex.
2. Push start vertex on queue & mark it.
3. Pop vertex from queue.
4. Visit (Print) the vertex.
5. Put all non-~~visited~~ marked neighbours of the vertex on the queue and mark them.
6. Repeat 3-5 until queue is empty.

- BFS is also referred as level-wise search algorithm.



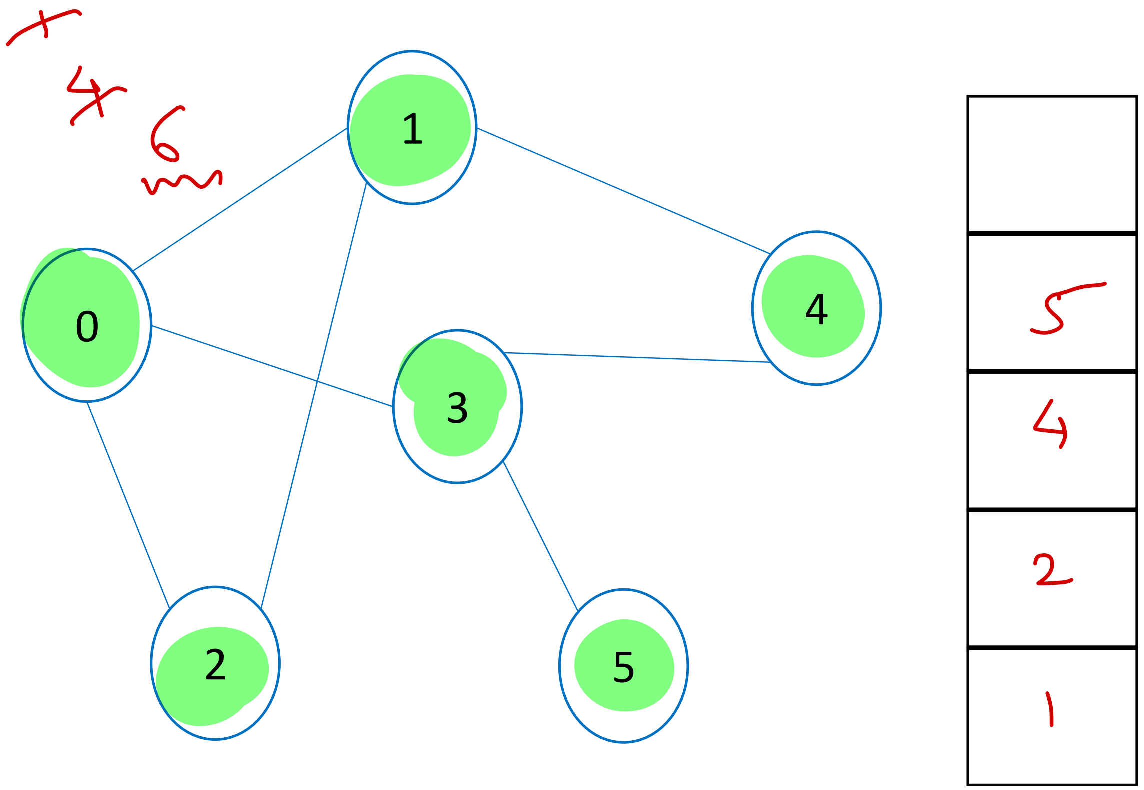
5 3 0 4 1 2



$O(V^2)$

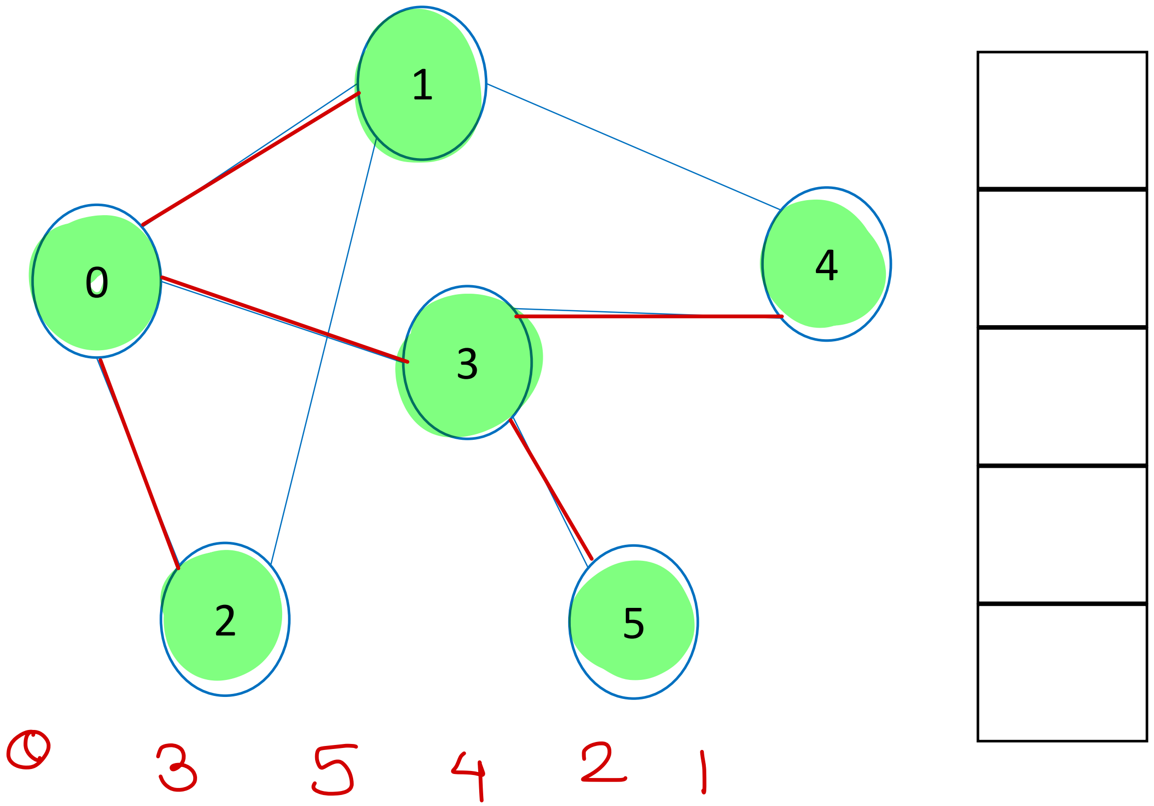
Check Connected-ness

1. push starting vertex on stack & mark it.
2. begin counting marked vertices from 1.
3. pop a vertex from stack.
4. push all its non-marked neighbors on the stack, mark them and increment count.
5. if count is same as number of vertices, graph is connected (return).
6. repeat steps 3-5 until stack is empty.
7. graph is not connected (return)



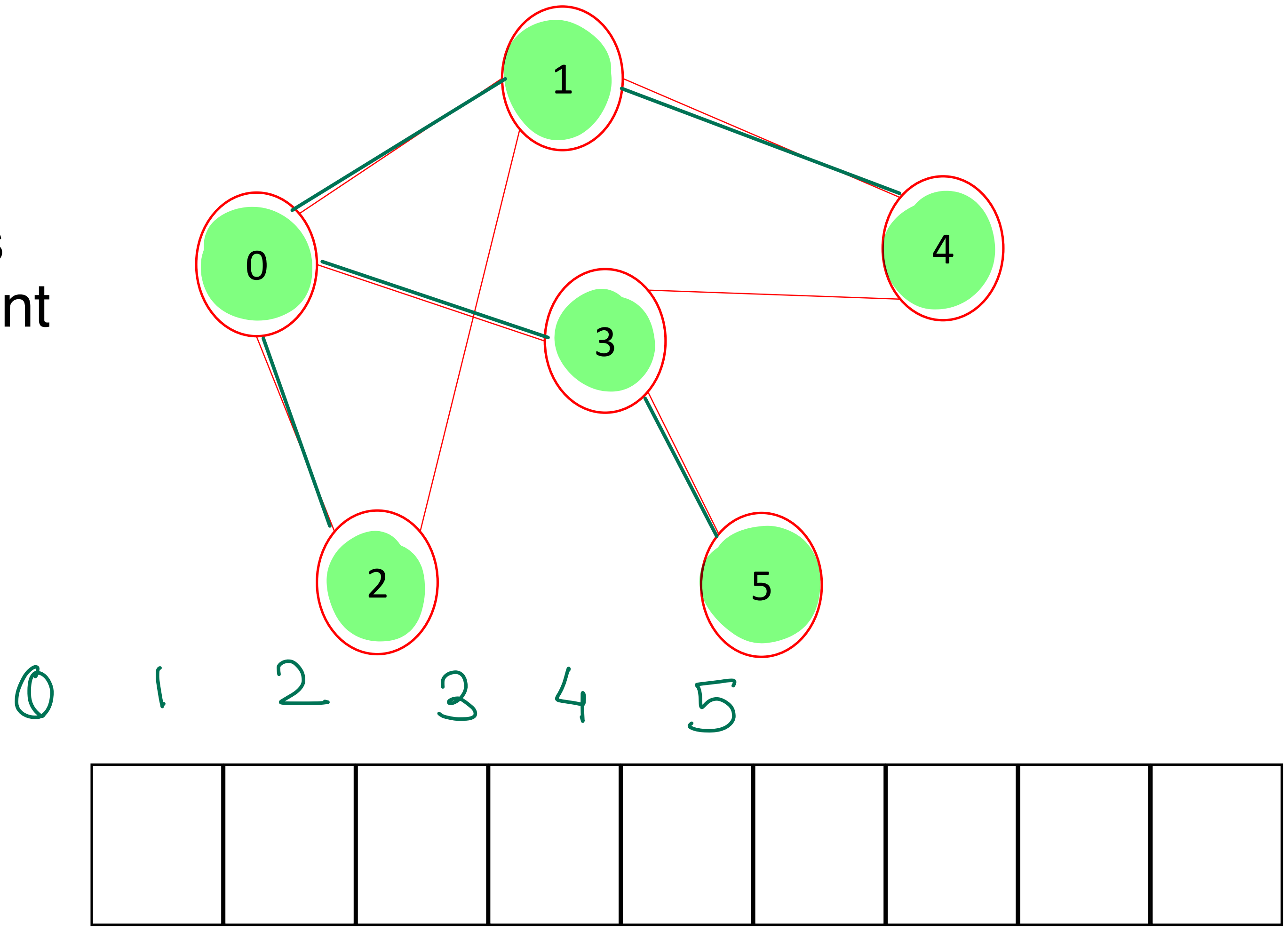
DFS Spanning Tree

1. push starting vertex on stack & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the stack, mark them. Also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until stack is empty.



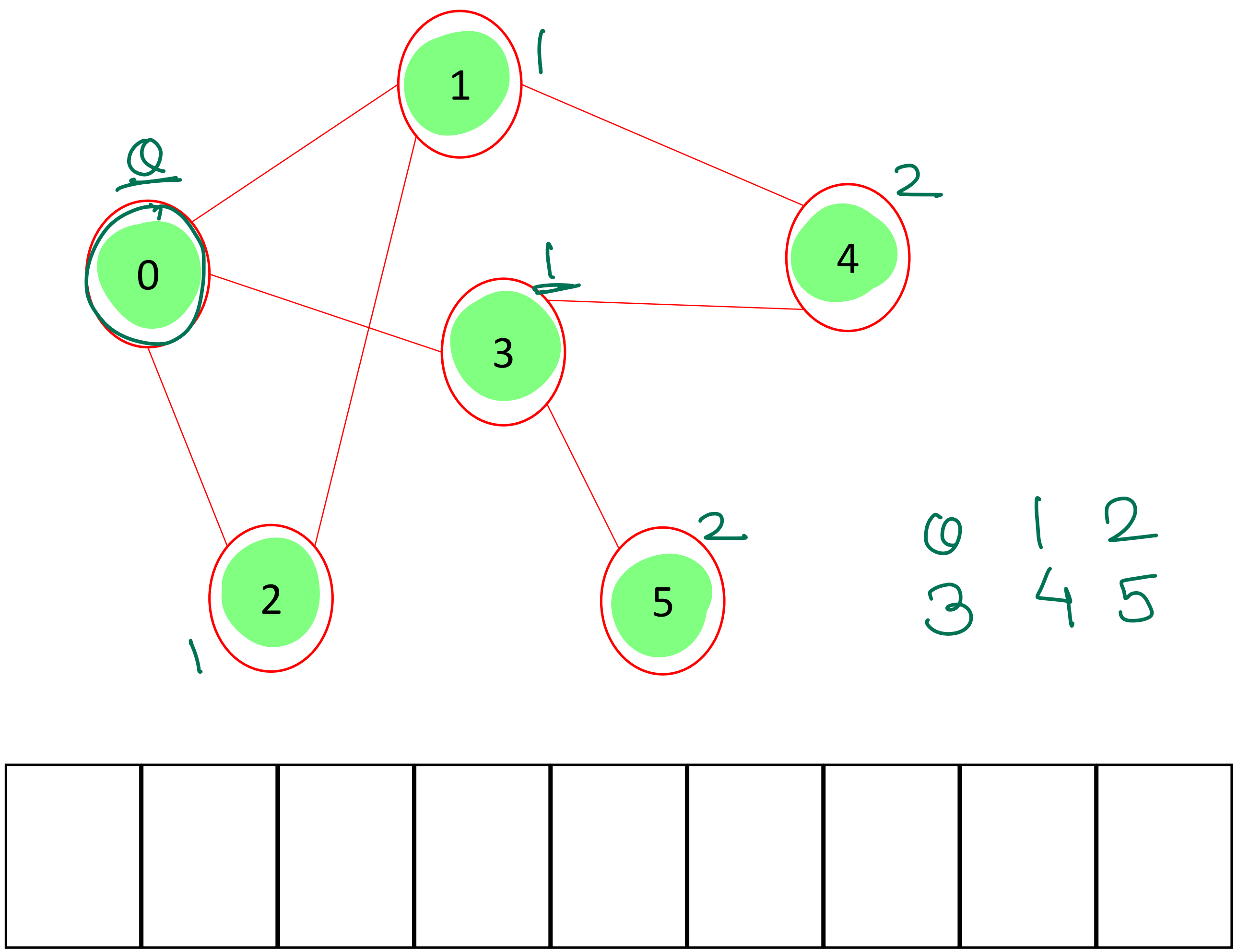
BFS Spanning Tree

1. push starting vertex on queue & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the queue, mark them. Also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until queue is empty.



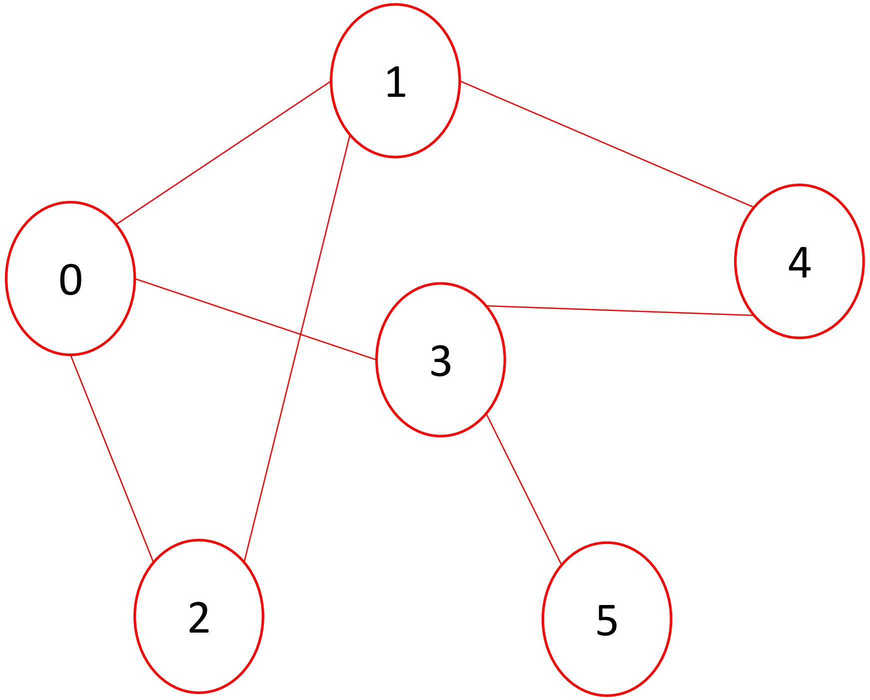
Single Source Path Length

1. Create path length array to keep distance of vertex from start vertex.
2. Consider dist of start vertex as 0.
3. push start vertex on queue & mark it.
4. pop the vertex.
5. push all its non-marked neighbors on the queue, mark them.
6. For each such vertex calculate its distance as dist[neighbor] = dist[current] + 1
7. repeat steps 3-6 until queue is empty.
8. Print path length array.



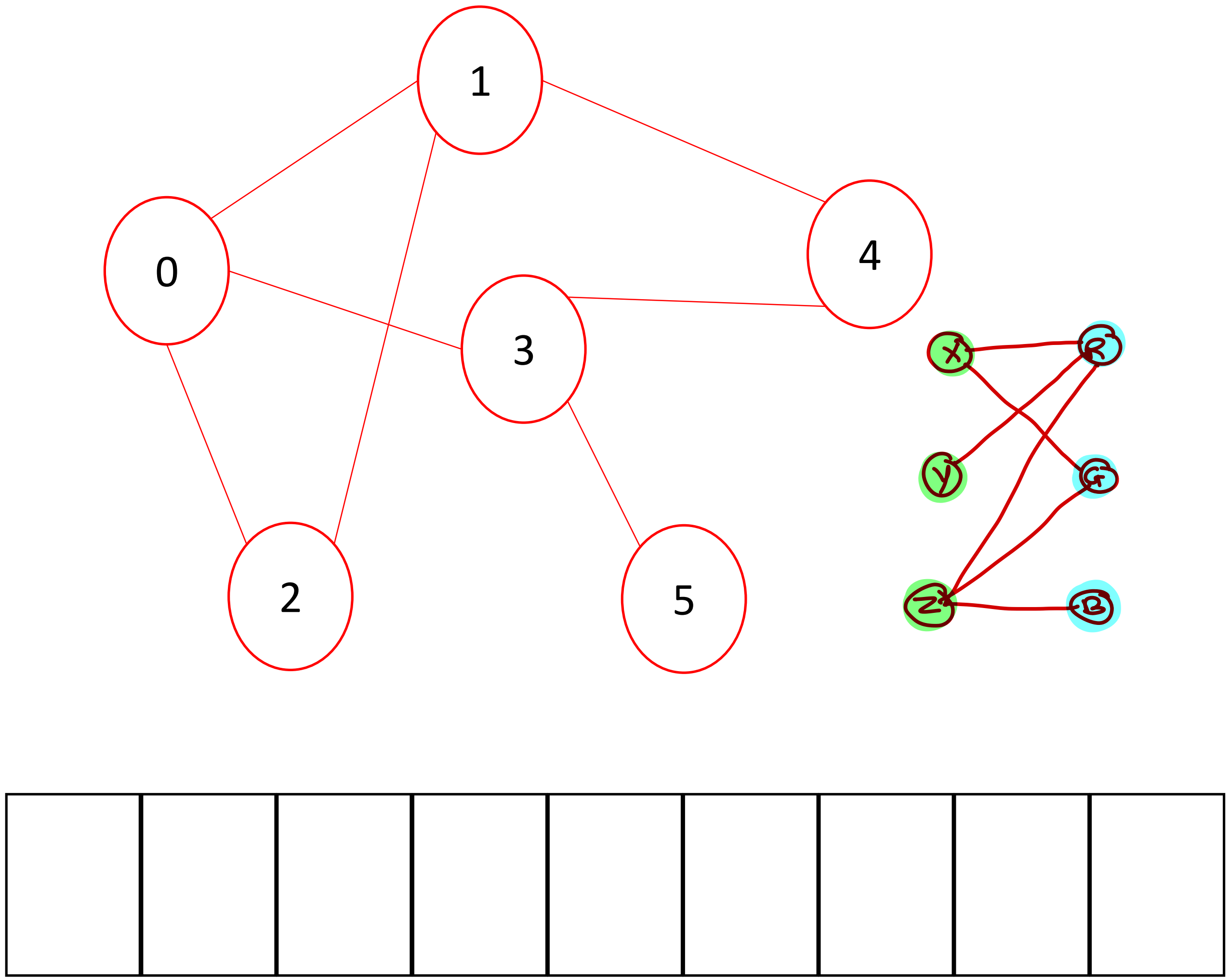
Check Bipartite-ness

1. keep colors of all vertices in an array. Initially vertices have no color.
2. push start on queue & mark it. Assign it color1.
3. pop the vertex.
4. push all its non-marked neighbors on the queue, mark them.
5. For each such vertex if no color is assigned yet, assign opposite color of current vertex (c1-c2, c2-c1).
6. If vertex is already colored with same of current vertex, graph is not bipartite (return).
7. repeat steps 3-6 until queue is empty.



Check Bipartite-ness

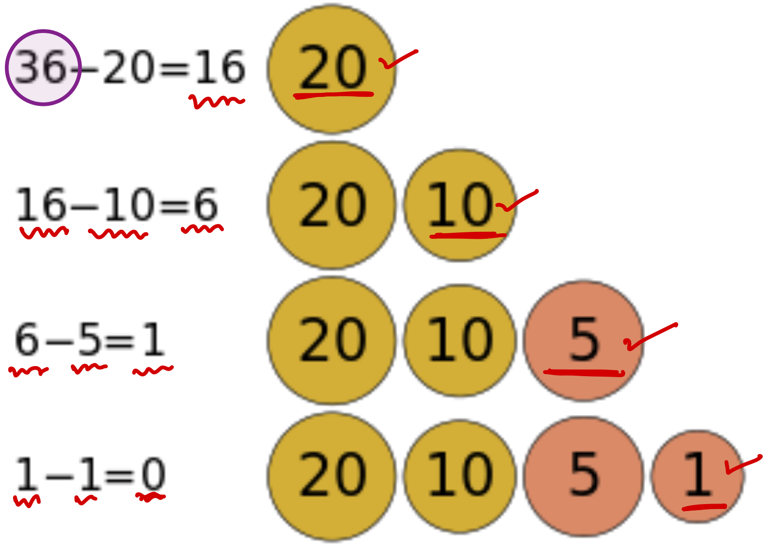
1. keep colors of all vertices in an array. Initially vertices have no color.
2. push start on queue & mark it. Assign it color1.
3. pop the vertex.
4. push all its non-marked neighbors on the queue, mark them.
5. For each such vertex if no color is assigned yet, assign opposite color of current vertex (c1-c2, c2-c1).
6. If vertex is already colored with same of current vertex, graph is not bipartite (return).
7. repeat steps 3-6 until queue is empty.



Problem solving technique: Greedy approach

- A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.
- We can make choice that seems best at the moment and then solve the sub-problems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the sub-problem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.
- A greedy algorithm never reconsiders its choices.
- A greedy strategy may not always produce an optimal solution.

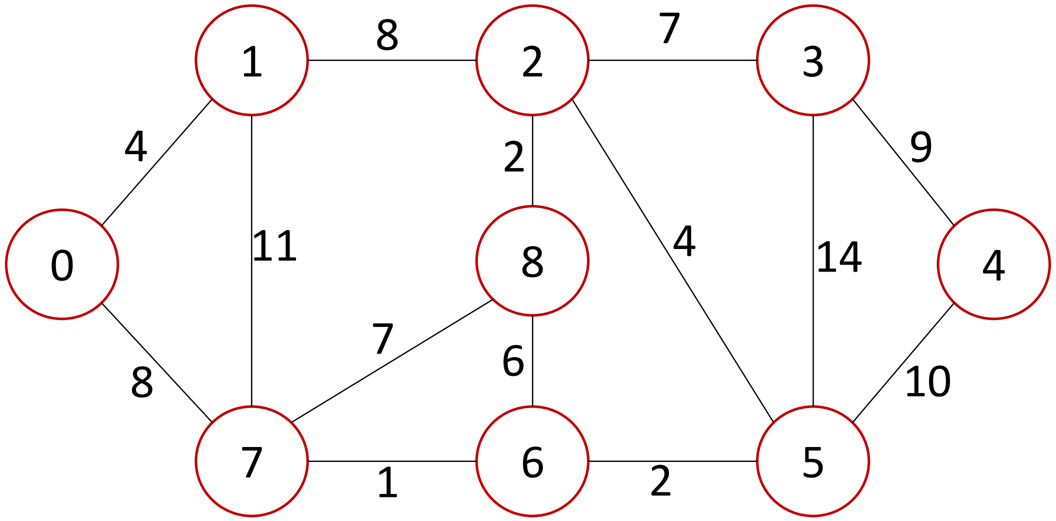
20 10 5 1

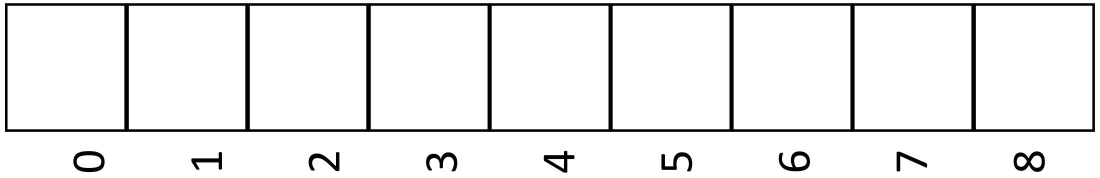


- Greedy algorithm decides minimum number of coins to give while making change.

Union Find Algorithm

1. Consider all vertices as disjoint sets (parent = -1).
2. For each edge in the graph
    1. Find set of first vertex.
    2. Find set of second vertex.
    3. If both are in same set, cycle is detected.
    4. Otherwise, merge both the sets i.e. add root of first set under second set

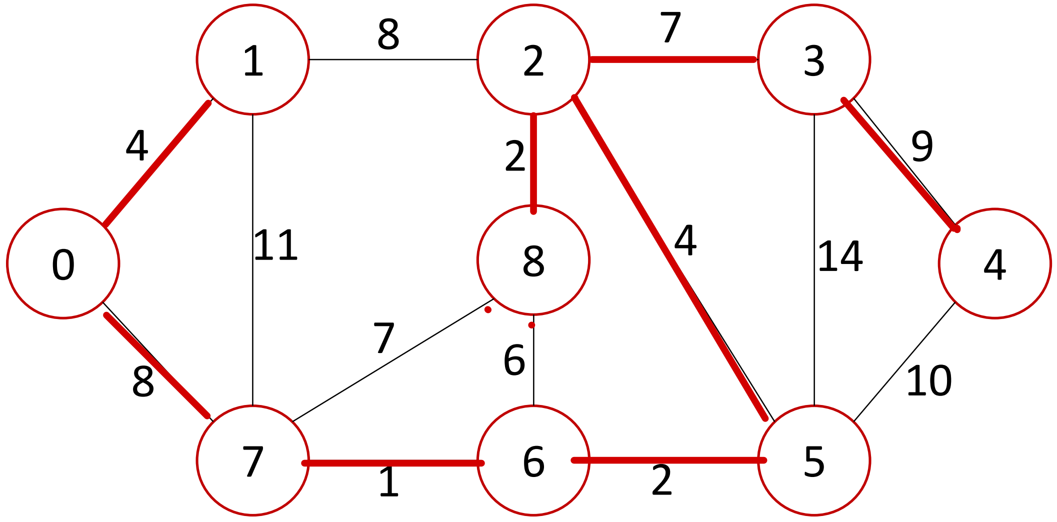




src	des	wt
7	6	1
8	2	2
6	5	2
0	1	4
2	5	4
8	6	6
2	3	7
7	8	7
0	7	8
1	2	8
3	4	9
5	4	10
1	7	11
3	5	14

Kruskal's MST

1. Sort all the edges in ascending order of their weight. O(n log n)
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.



src	des	wt
7	6	1
8	2	2
6	5	2
0	1	4
2	5	4
8	6	6
2	3	7
7	8	7
0	7	8
1	2	8
3	4	9
5	4	10
1	7	11
3	5	14

Union Find Algorithm – Analysis

1. Consider all vertices as disjoint sets (parent = -1).
2. For each edge in the graph
   1. Find set of first vertex.
   2. Find set of second vertex.
   3. If both are in same set, cycle is detected.
   4. Otherwise, merge both the sets i.e. add root of first set under second set

- Time complexity
  - Skewed tree implementation
  - O(V)

- Improved time complexity
  - Rank based tree implementation
  - O(log V)

Kruskal's MST – Analysis

1. Sort all the edges in ascending order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.

- Time complexity
  - Sort edges: O(E log E)
  - Pick edges (E edges): O(E)
  - Union Find: O(log V)

- Time complexity
  - O(E log E + E log V)

  - E can max V^2.
  - So max time complexity: O(E log V).

SUNBEAM

Thank you!

Nilesh Ghule <Nilesh@sunbeaminfo.com>