Data Structure & Algorithms

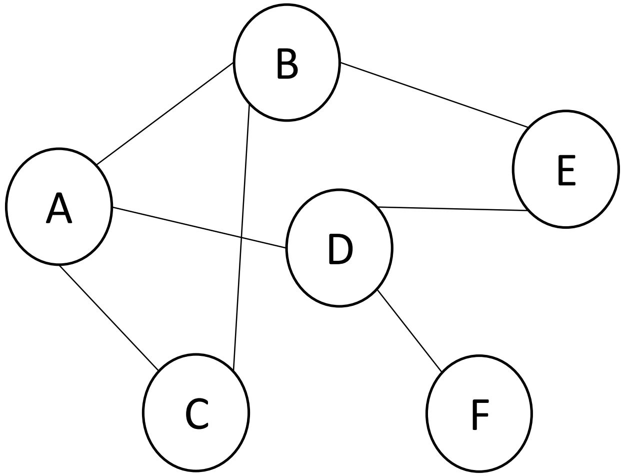
Sunbeam Infotech

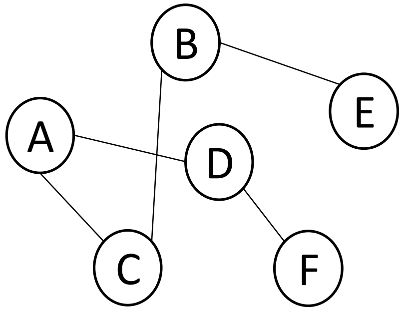
Agenda

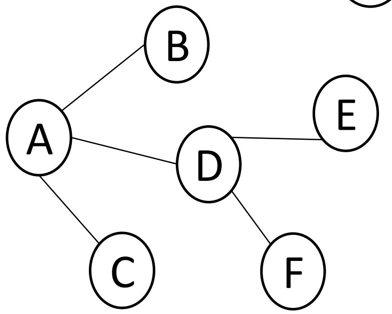
- Graph implementation
  - Adjacency list ✓
- BFS & DFS Traversal ✓
- DFS based algorithms
  - Check connected ness ✓
  - DFS Spanning tree ✓
- BFS based algorithms
  - BFS Spanning tree ✓
  - Single source path length ✓
  - Check bi-partite ness ✓
- Greedy approach ✓
- Prim’s algorithm ✓
- Dijkstra’s algorithm ✓

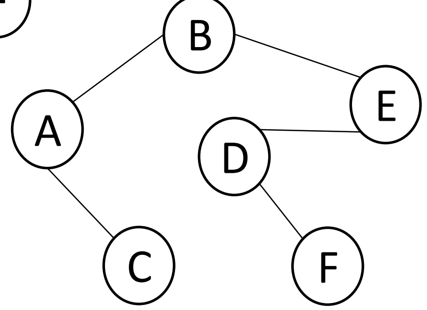
Spanning Tree

- 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.
- 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



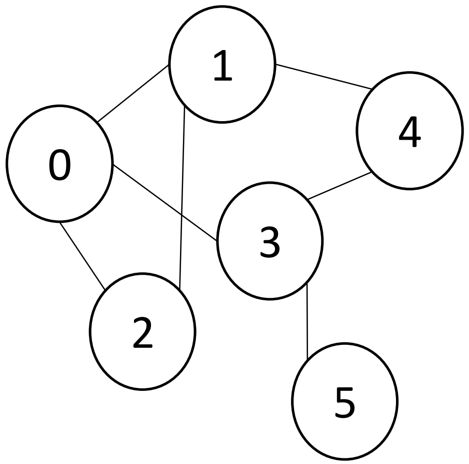


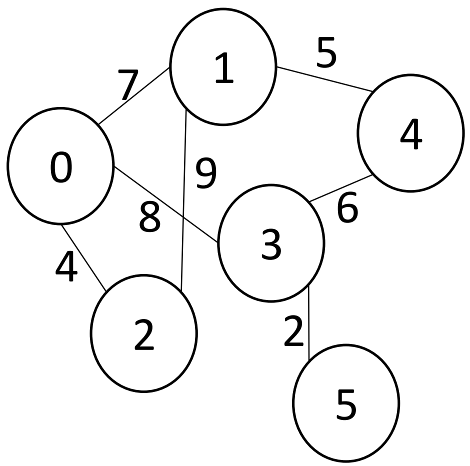




Graph Implementation – Adjacency Matrix

- 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 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 $O(V^2)$.



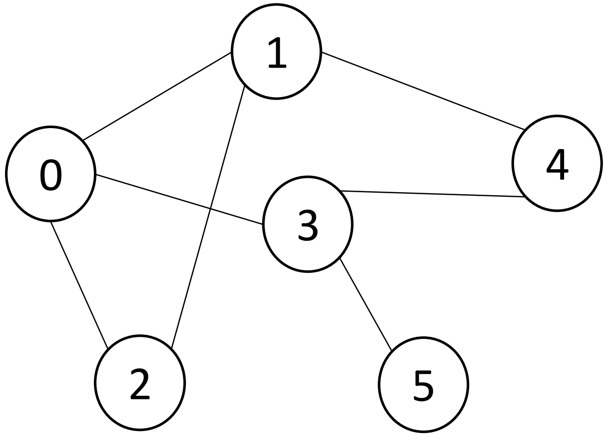


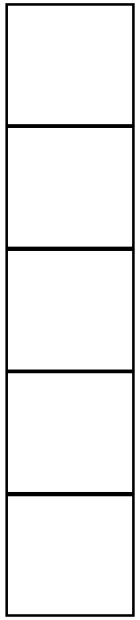
	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	1	1	0	0	0	0
D	1	0	0	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

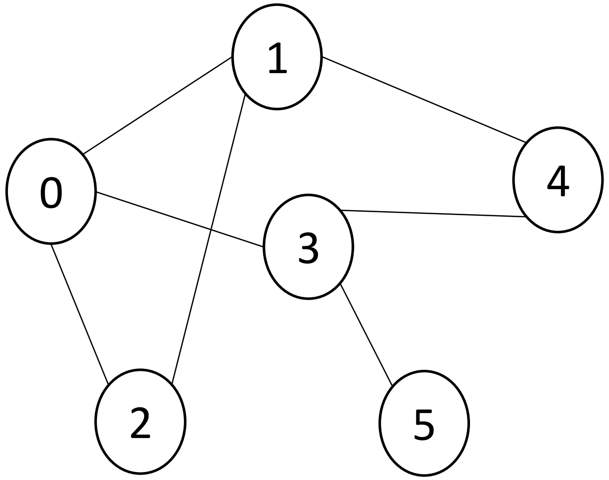
	A	B	C	D	E	F
A	∞	7	4	8	∞	∞
B	7	∞	9	∞	5	∞
C	4	9	∞	∞	∞	∞
D	8	∞	∞	∞	6	2
E	∞	5	∞	6	∞	∞
F	∞	∞	∞	2	∞	∞

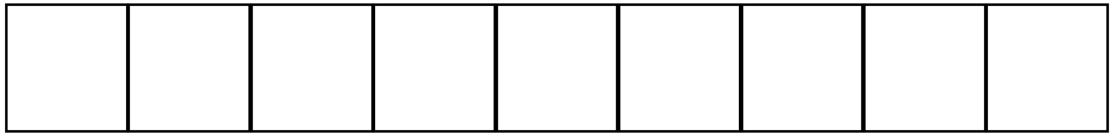
Graph Traversal – BFS & DFS

- DFS algorithm.
    1. Choose a vertex as start vertex.
    2. Push start vertex on stack & mark it.
    3. Pop vertex from stack.
    4. Print the vertex.
    5. Put all non-visited neighbours of the vertex on the stack and mark them.
    6. Repeat 3-5 until stack is empty.
- BFS algorithm.
    1. Choose a vertex as start vertex.
    2. Push start vertex on queue & mark it
    3. Pop vertex from queue.
    4. Print the vertex.
    5. Put all non-visited neighbours of the vertex on the queue and mark them.
    6. Repeat 3-5 until queue is empty.





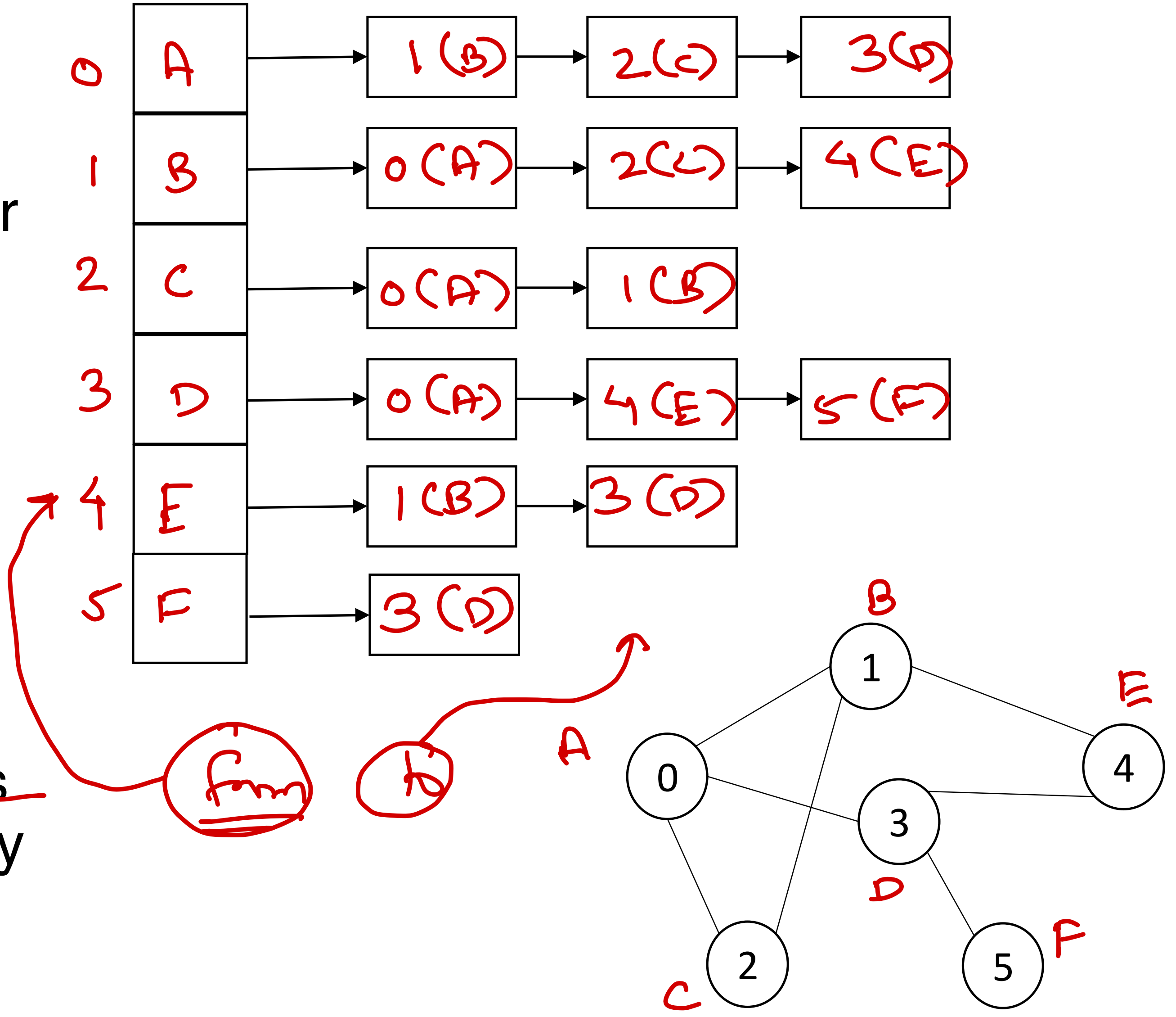




Graph Implementation – Adjacency List

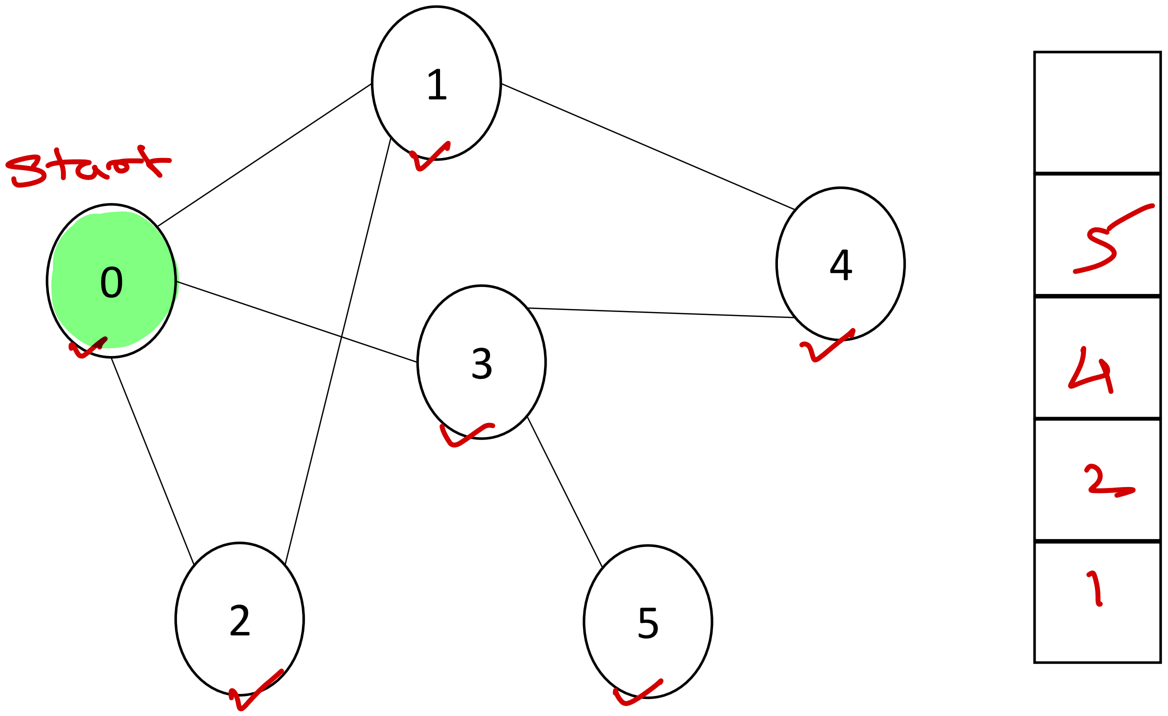
V = 6 E = 7 | 14

- 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).



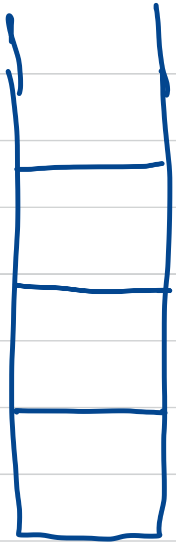
Check Connected-ness → dfs also. vert_cnt = 6

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



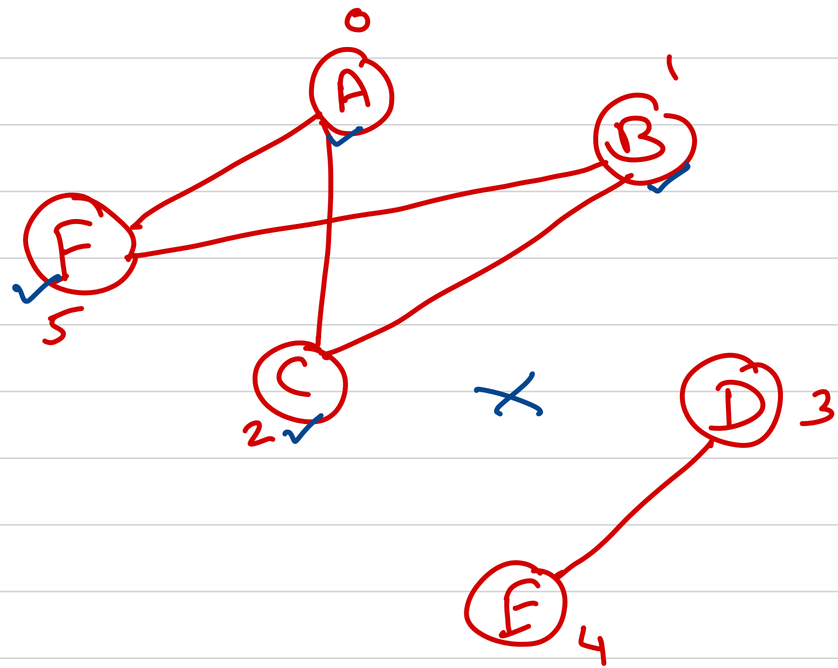
0 3

cnt = 6



cnt = 4

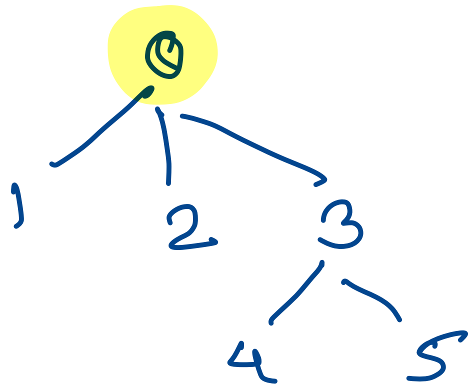
Vertcnt = 6

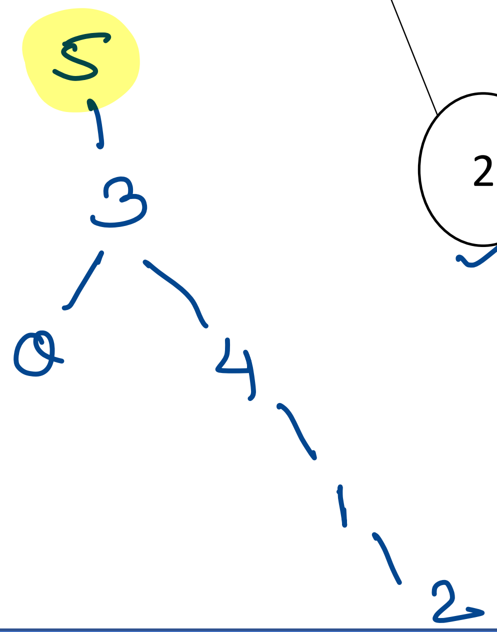


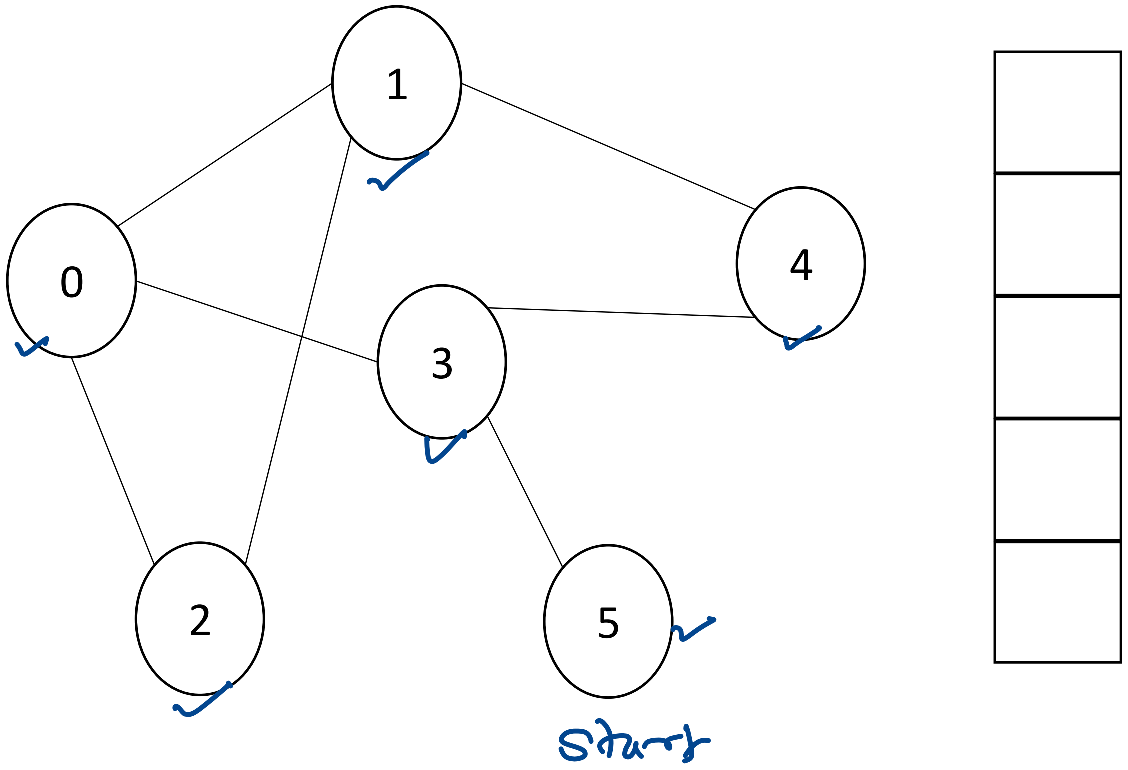
0 5 1 2

DFS Spanning Tree

1. push start on stack & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the stack, mark them.
4. print current vertex to that neighbor vertex edge.
5. repeat steps 2-4 until stack is empty.

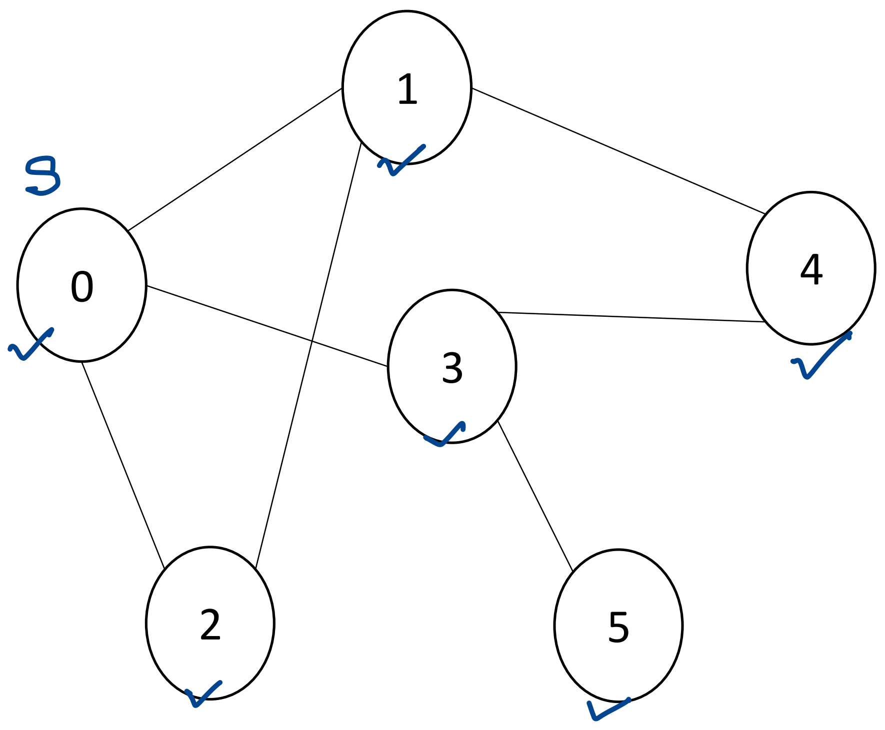




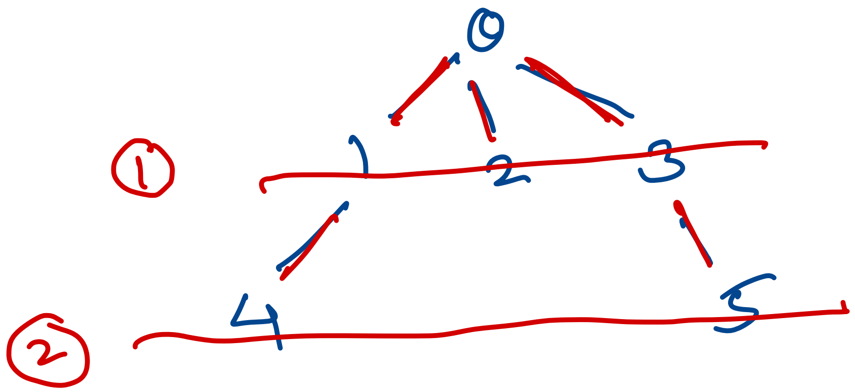


BFS Spanning Tree

1. push start on queue & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the queue, mark them.
4. print current vertex to that neighbor vertex edge.
5. repeat steps 2-4 until queue is empty.

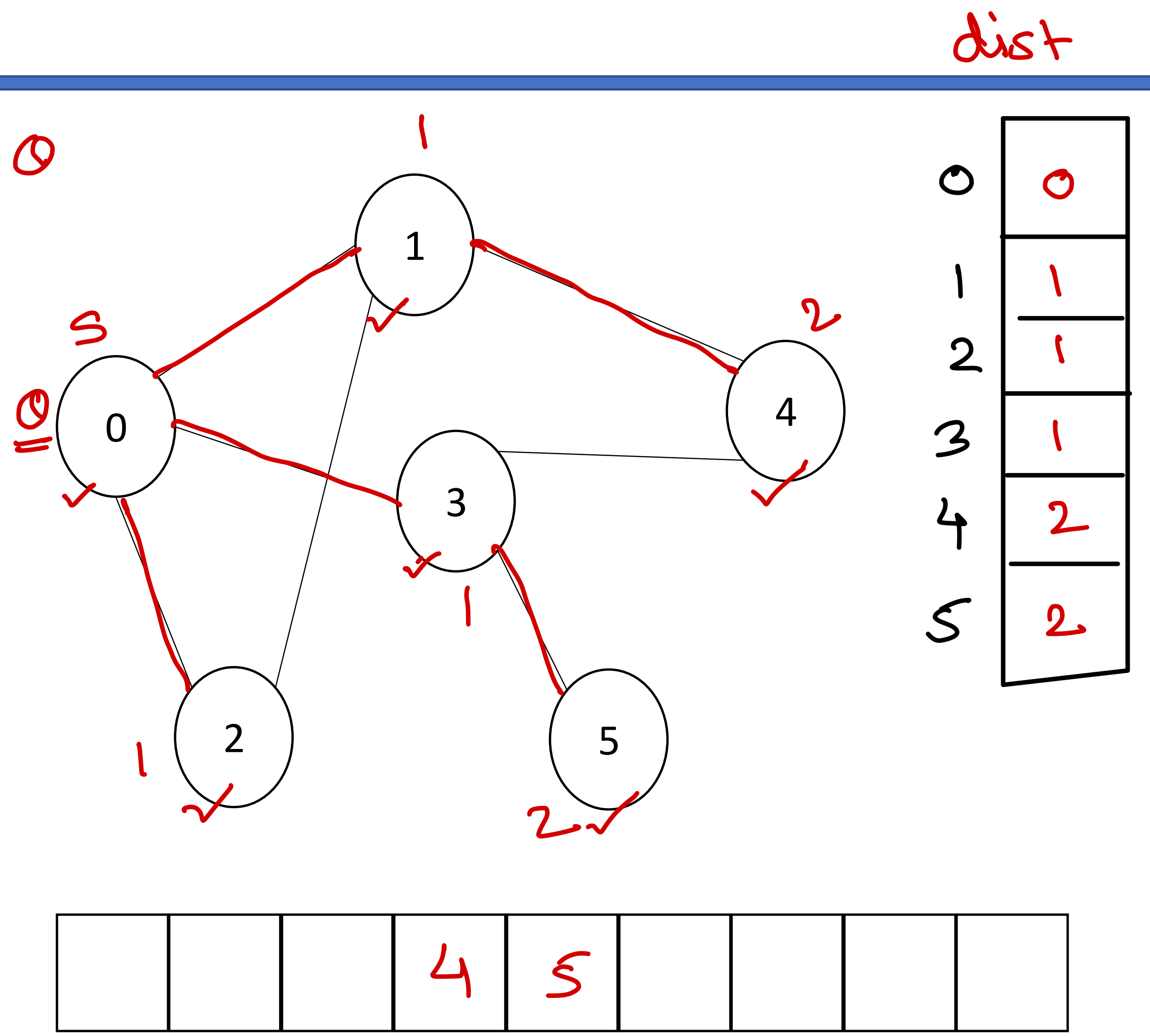


			4	5				



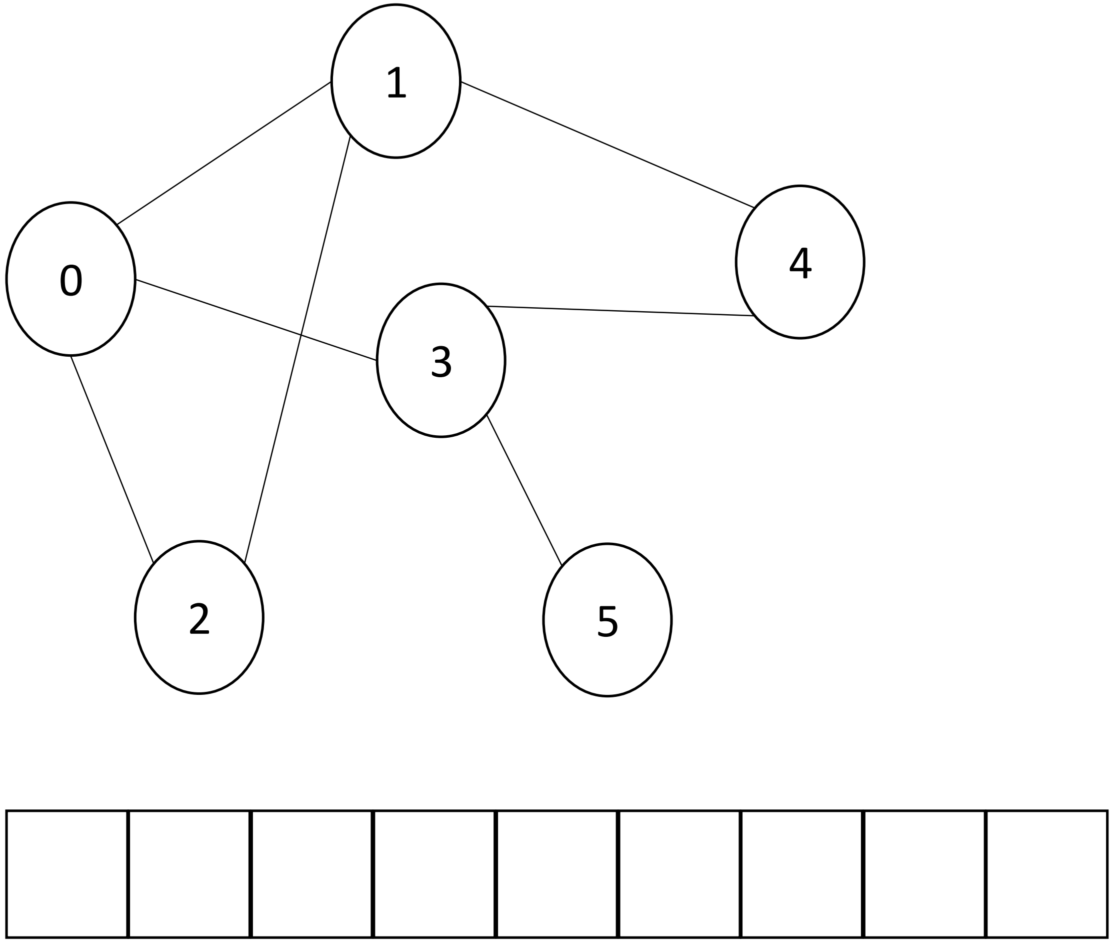
Single Source Path Length

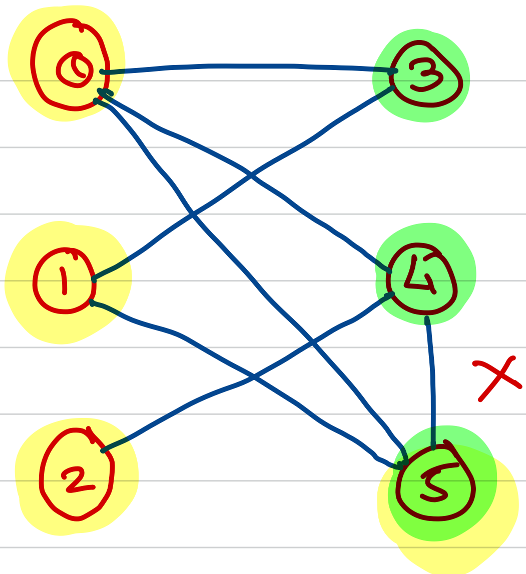
1. Create path length array to keep distance of vertex from start vertex.
2. push start on queue & mark it.
3. pop the vertex.
4. push all its non-marked neighbors on the queue, mark them.
5. For each such vertex calculate distance as dist[neighbor] = dist[current] + 1
6. print current vertex to that neighbor vertex edge.
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. =0
2. push start on queue & mark it. Assign it color1(1)
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 (-1) 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.



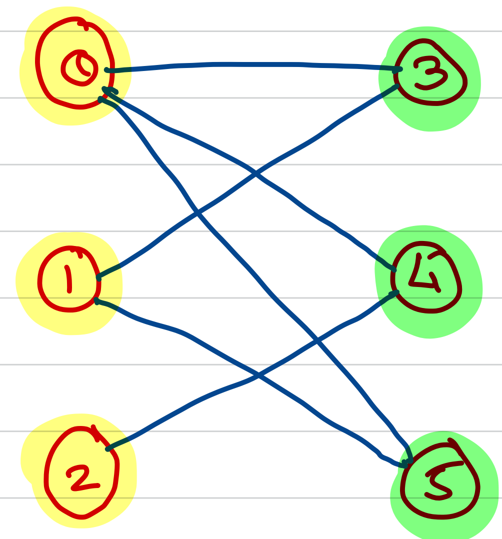


yellow = 1

green = -1

no color = 0

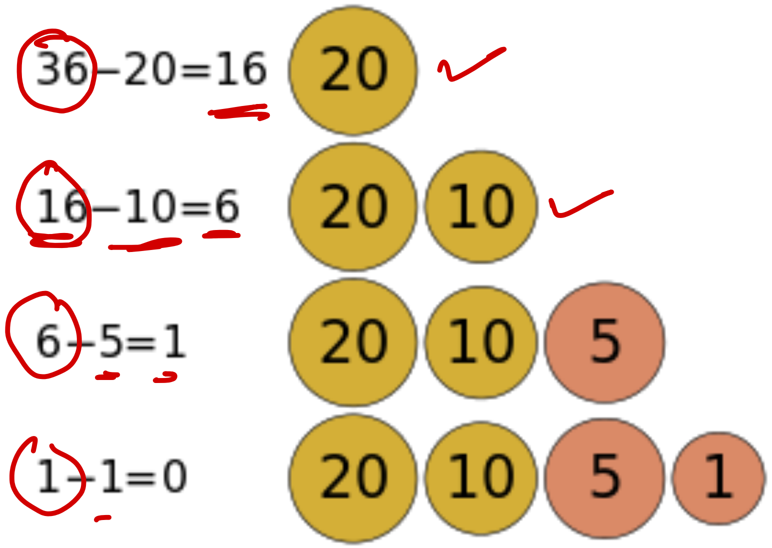
4



5 1 2

Greedy approach

- A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage[1] with the intent of finding a global optimum.
- We can make whatever choice 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.
- In other words, a greedy algorithm never reconsiders its choices.
- A greedy strategy may not always produce an optimal solution.



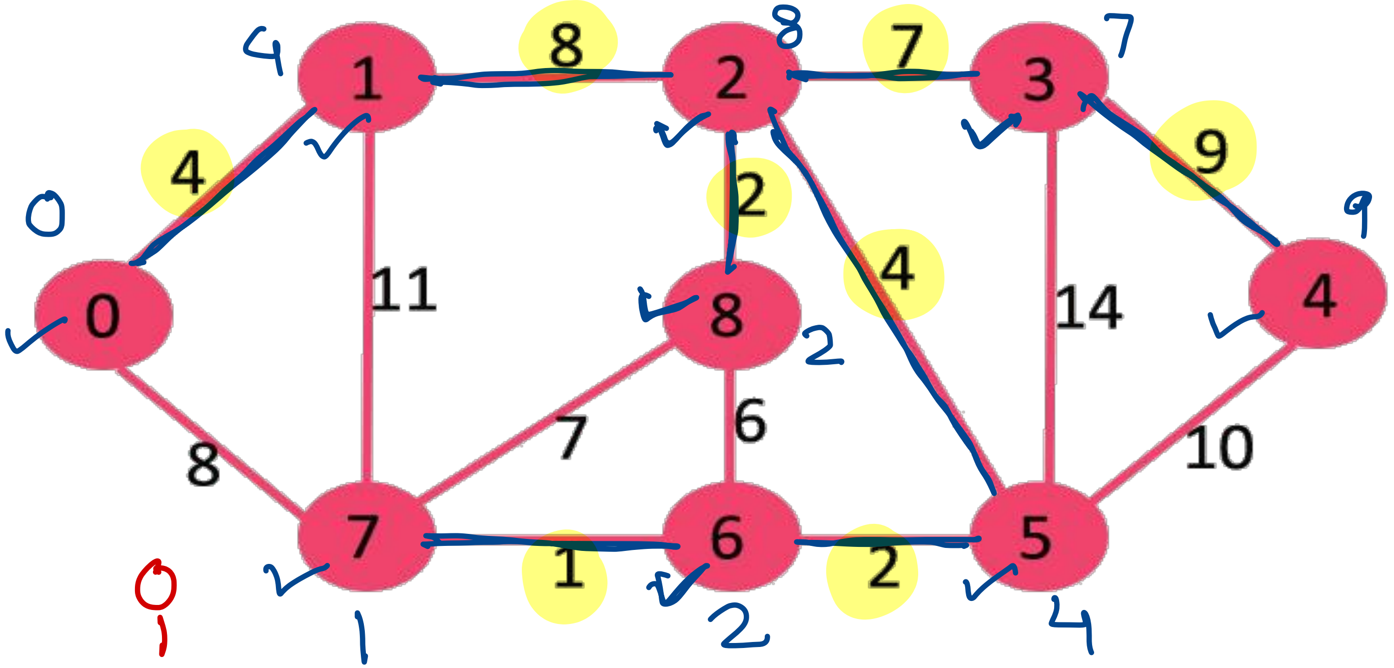
- Greedy algorithm decides minimum number of coins to give while making change.

Prim's MST (Minimum Weight Spanning Tree)

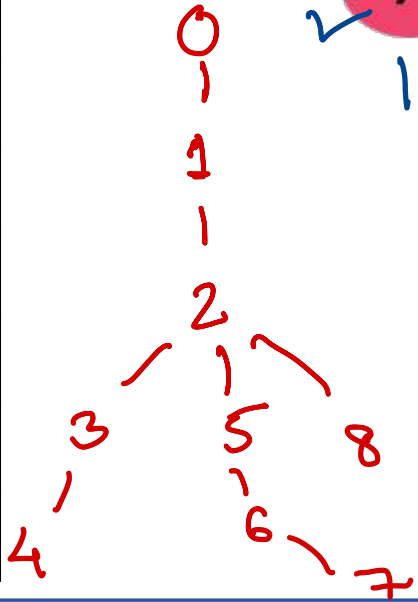
geeks for geeks.

1. Start with empty MST set and all vertex keys as infinity.
2. Consider starting vertex key as 0.
3. Get new minimum key vertex and add it into MST set.
4. Update keys of all neighbor vertices to the weights of edges, if its current key is greater than weight of connecting edge.
5. Repeat 3-4 until all vertices are added into MST set.

0	
1	
2	
3	
4	
5	
6	
7	
8	



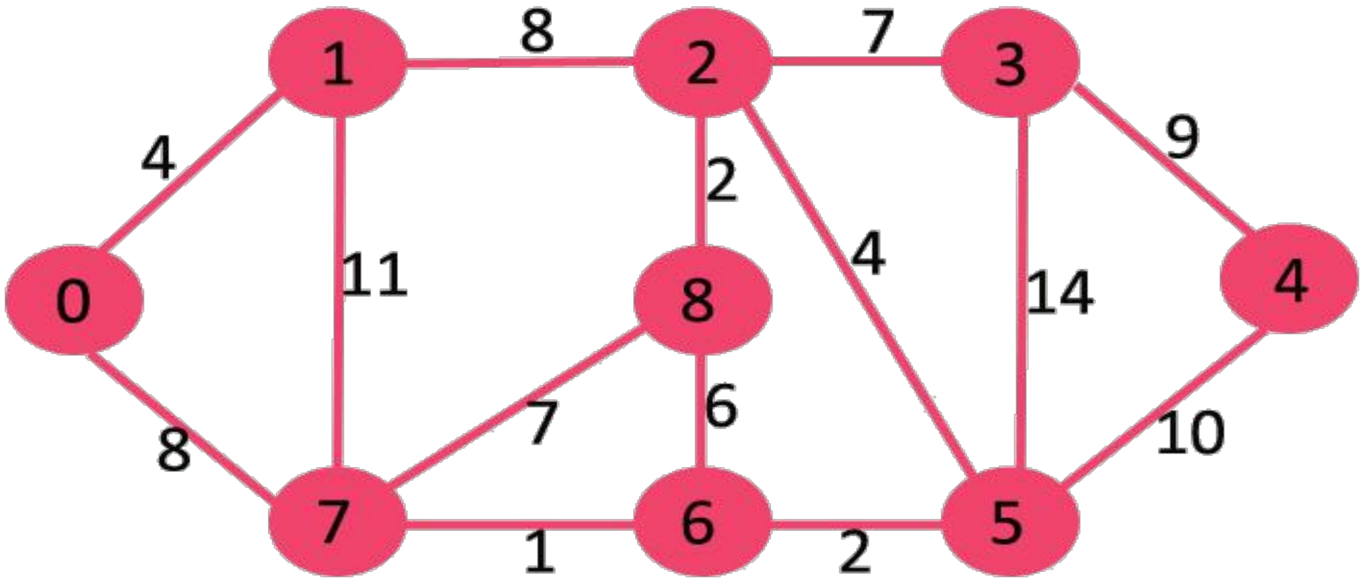
weight of MST = 37



Dijkstra’s Algorithm (Single Source Shortest Path)

1. Start with empty Shortest Path Tree set and all vertex distance as infinity.
2. Consider starting vertex distance as 0.
3. Get new minimum distance vertex and add it into SPT set.
4. Update distances of all neighbor vertices to the sum of current vertex distance & weight of connecting edge, if its current distance is greater than this sum.
5. Repeat 3-4 until all vertices are added into SPT set.

0	
1	
2	
3	
4	
5	
6	
7	
8	



SUNBEAM

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>