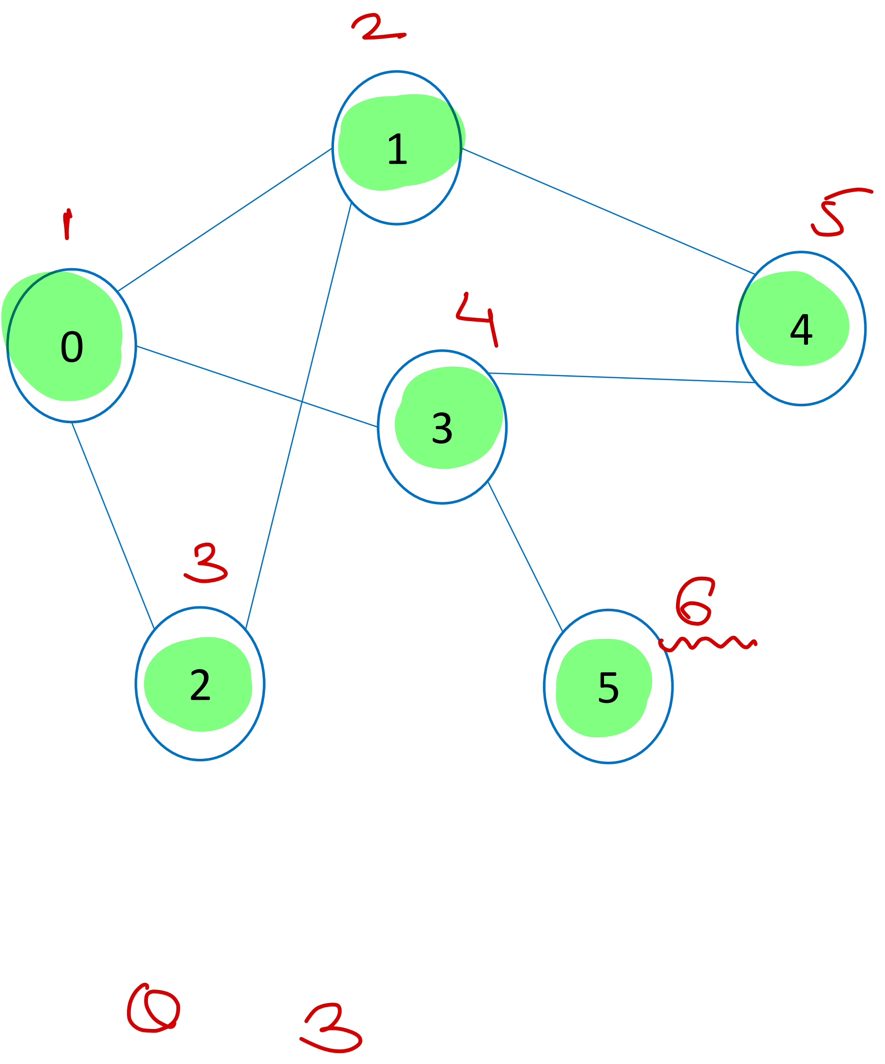
Data Structure & Algorithms

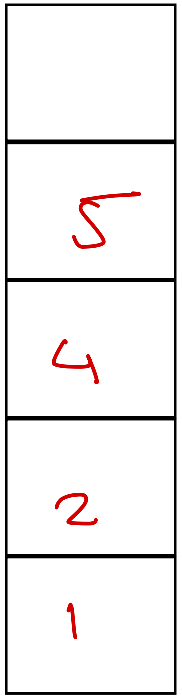
Sunbeam Infotech

Nilesh Ghule

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)

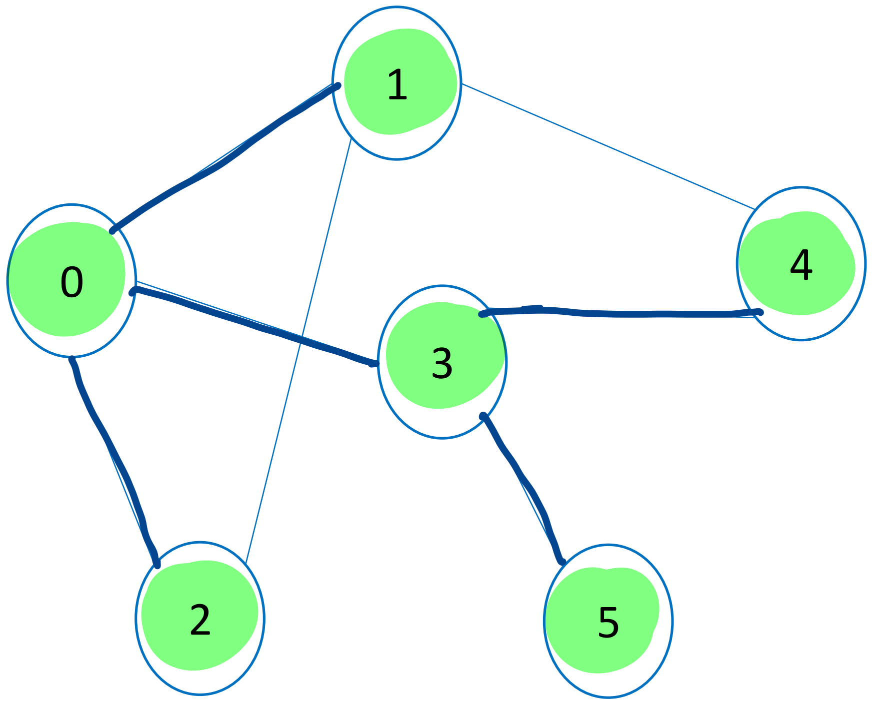


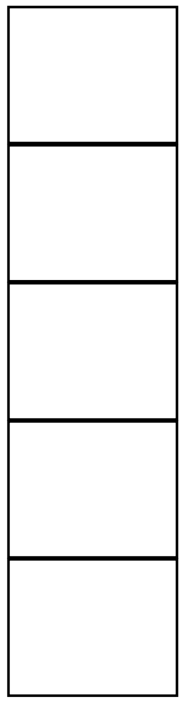


DFS Spanning Tree

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.

0 - 1
0 - 2
0 - 3
3 - 4
3 - 5

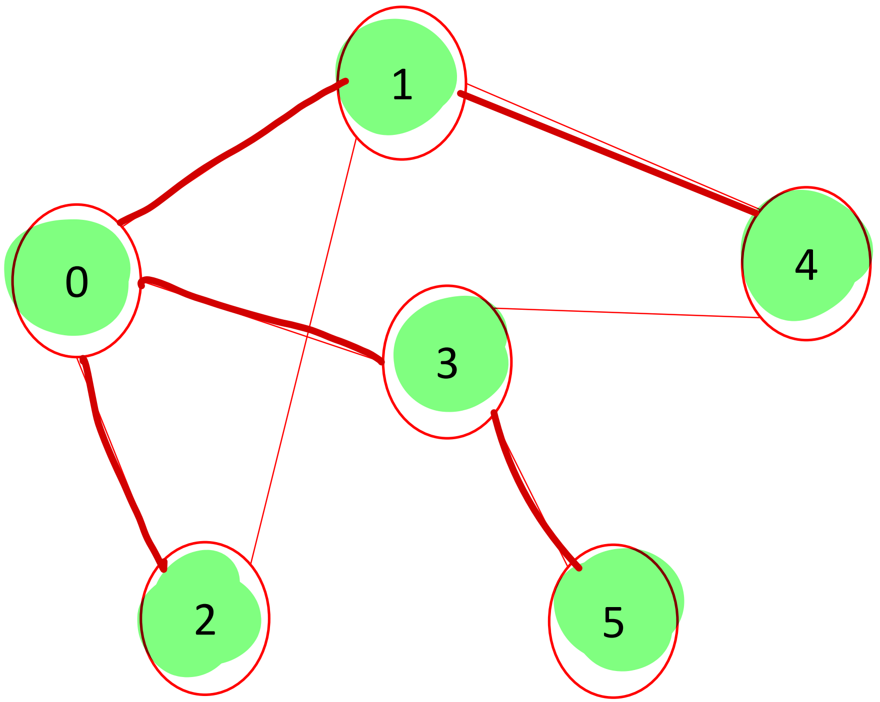




0 3 5 4 2 1

BFS Spanning Tree

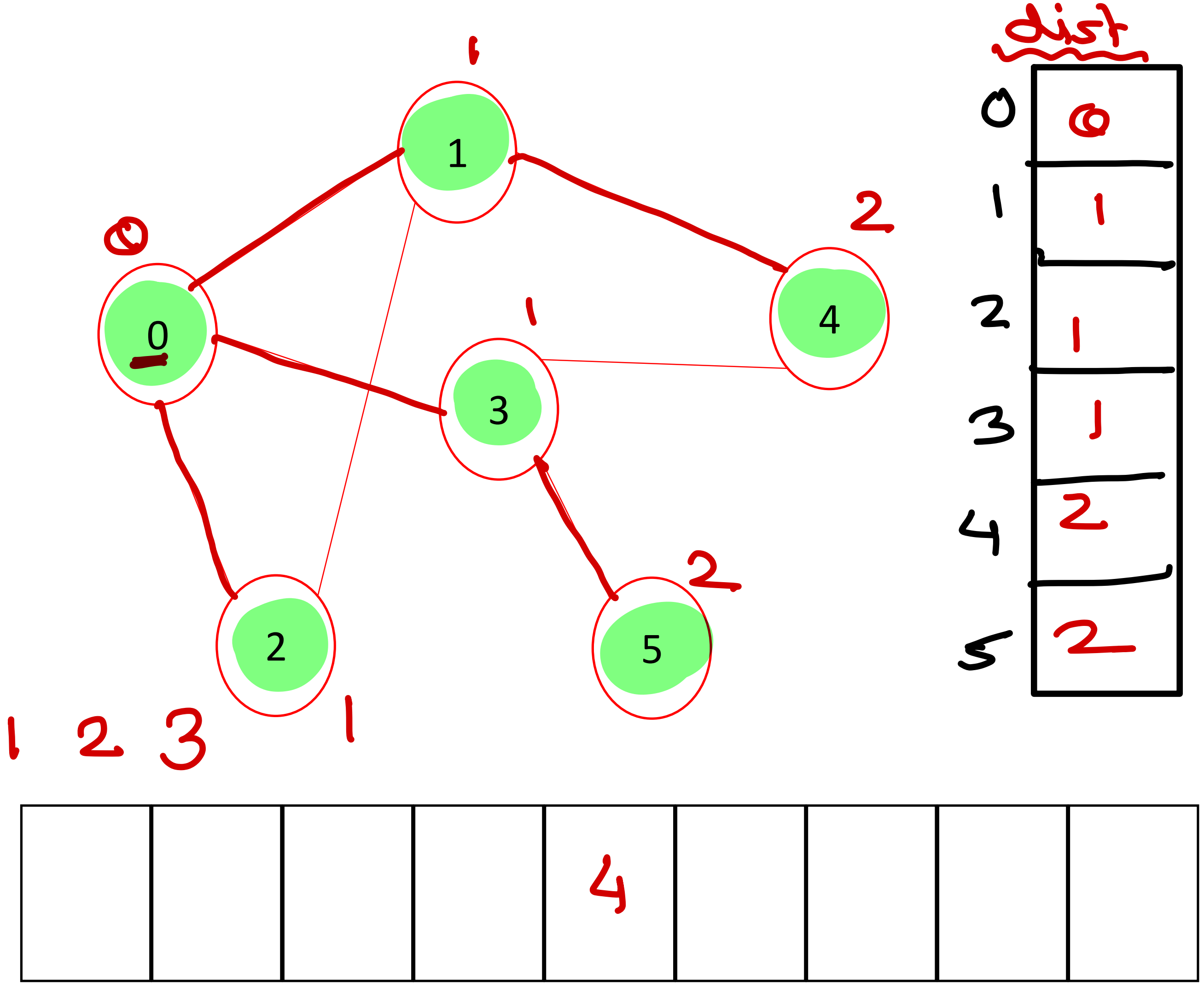
1. push starting vertex on queue & mark it.
2. pop the vertex.
3. push all its non-marked neighbors on the queue, mark them. Also print the vertex to neighboring vertex edges.
4. repeat steps 2-3 until queue is empty.



0 1 2 3 4 5

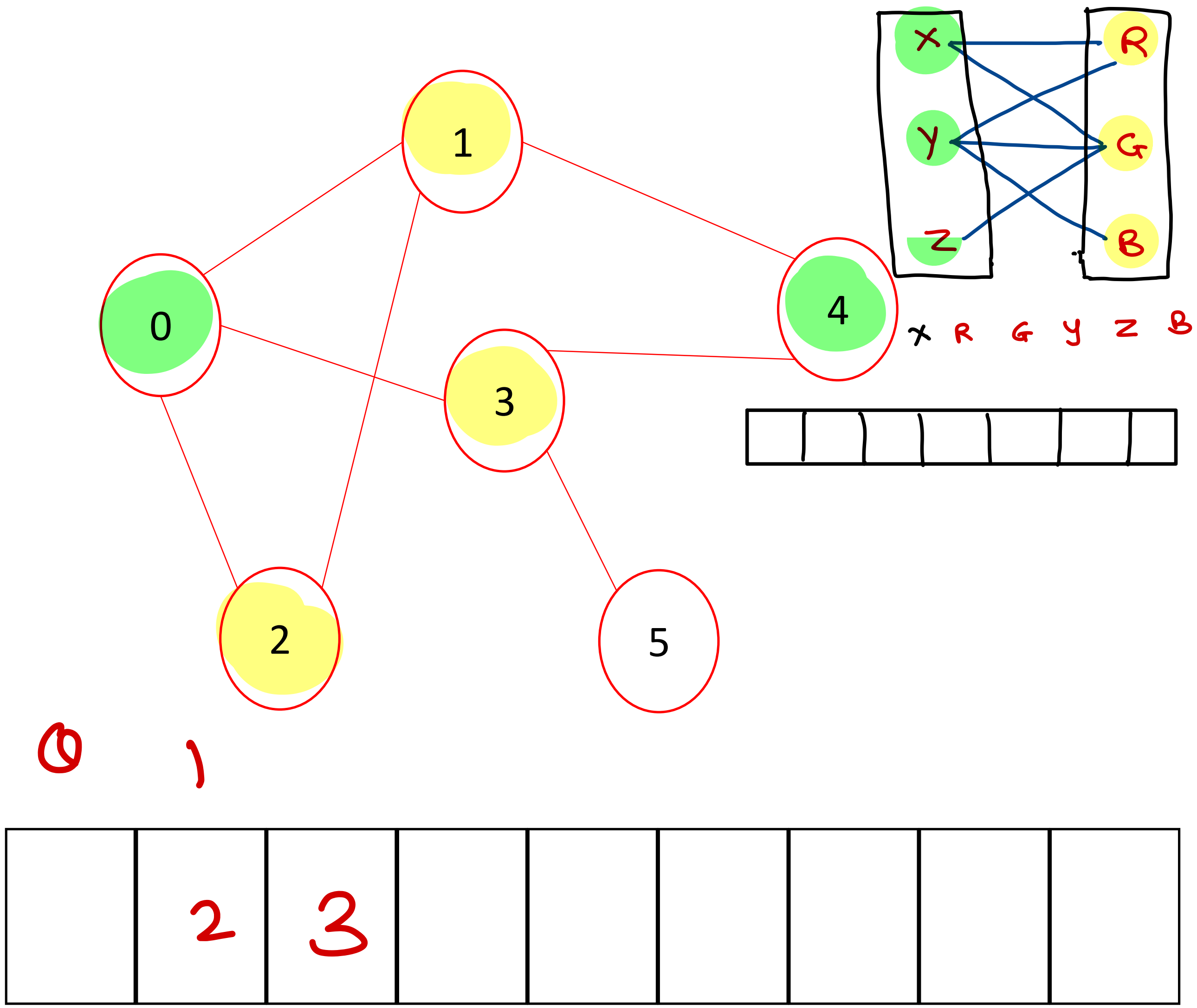
Single Source Path Length - for non-weighted graph.

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.



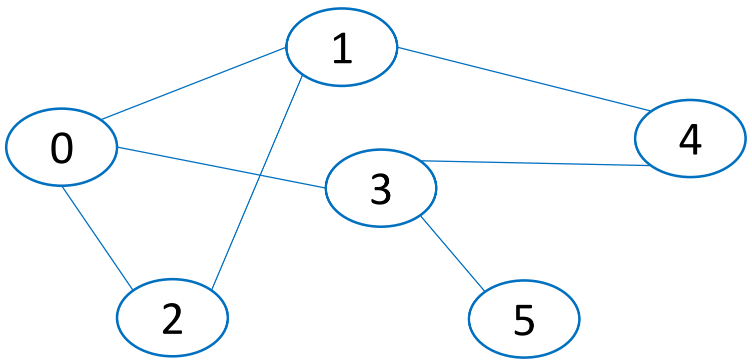
Check Bipartite-ness

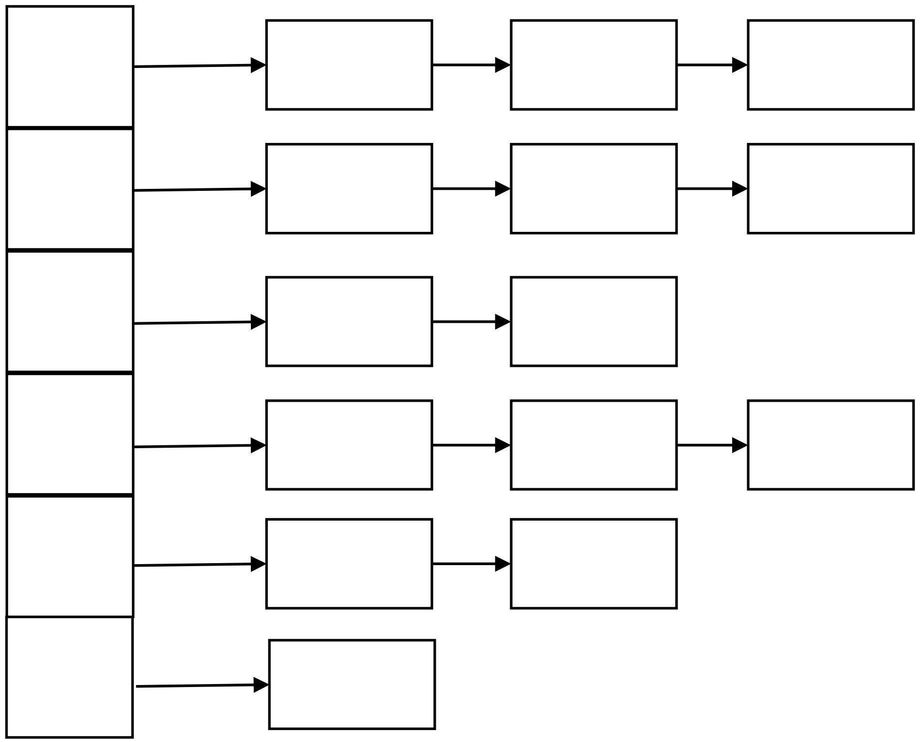
1. keep colors of all vertices in an array. Initially vertices have no color.
2. push start on queue & mark it. Assign it color1.
3. pop the vertex.
4. push all its non-marked neighbors on the queue, mark them.
5. For each such vertex if no color is assigned yet, assign opposite color of current vertex (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.



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).





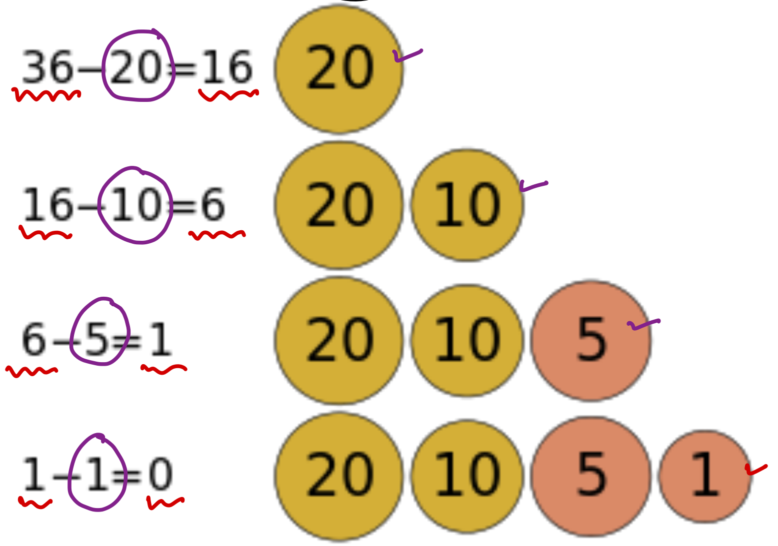
Problem solving technique: Greedy approach

- 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.

Graph Algos

MST
1. Kruskal algo
2. Prim's algo
3. Dijkstra's algo

50



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

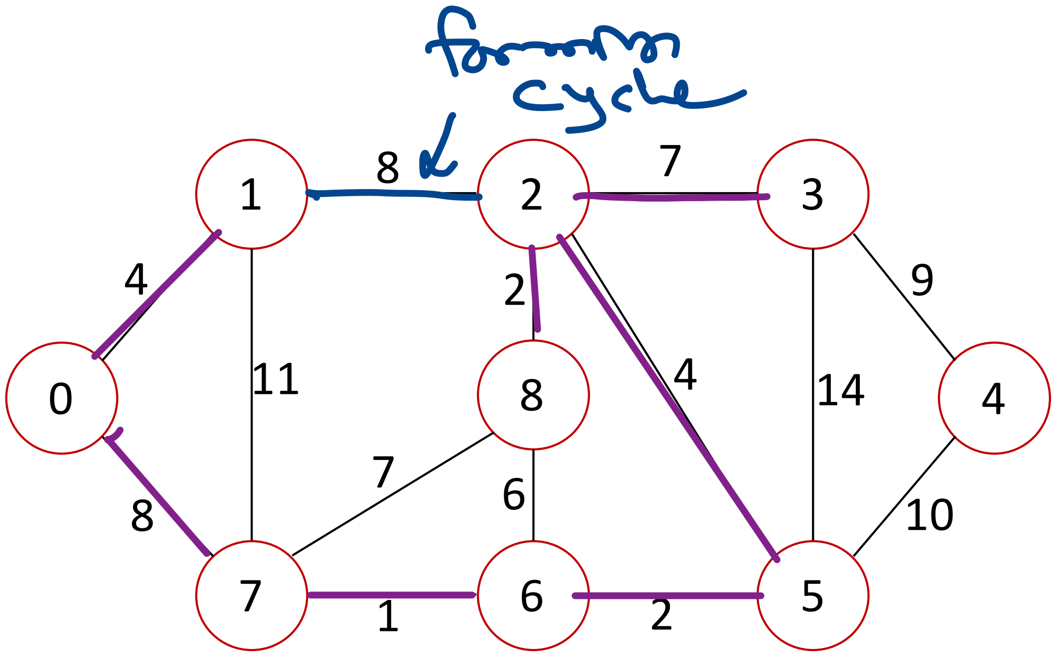
Union Find Algorithm - check if graph contains a cycle.

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

parent(src) = dest;

root root

p[ ] =



0	1	2	3	4	5	6	7	8
1	3	5			3	5	6	2

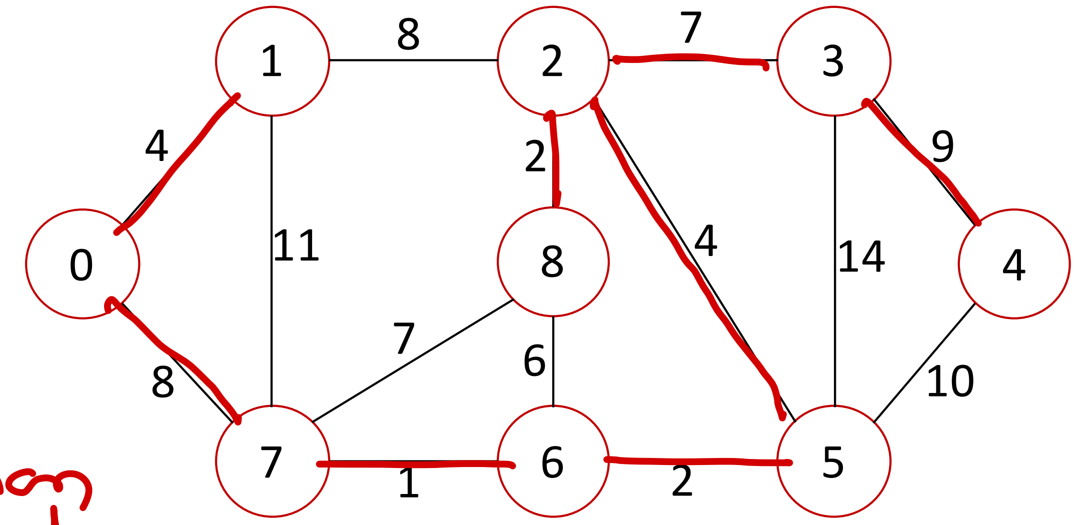
src	des	wt
7	6	1
8	2	2
6	5	2
0	1	4
2	5	4
~~8~~	~~6~~	~~6~~
2	3	7
~~7~~	~~8~~	~~7~~
0	7	8
1	2	8
3	4	9
5	4	10
1	7	11
3	5	14

Kruskal's MST min spanning tree

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.

greedy

union find



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)

gfg

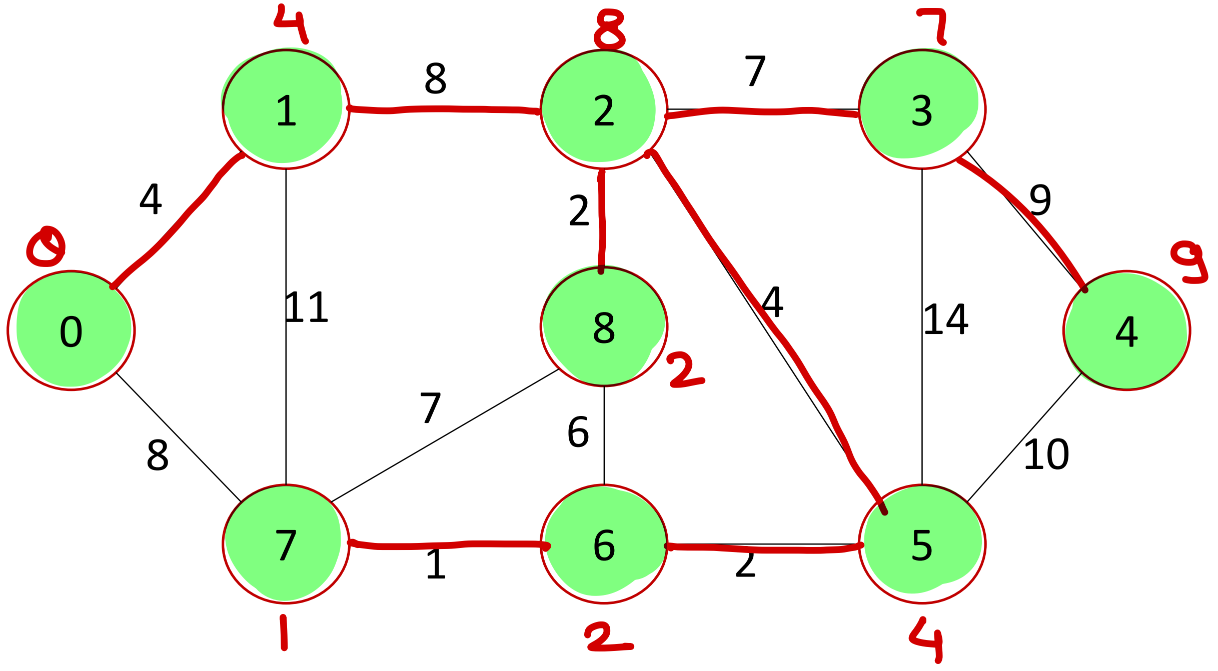
Kruskal's MST – Analysis

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).

Prim's MST - min spanning tree

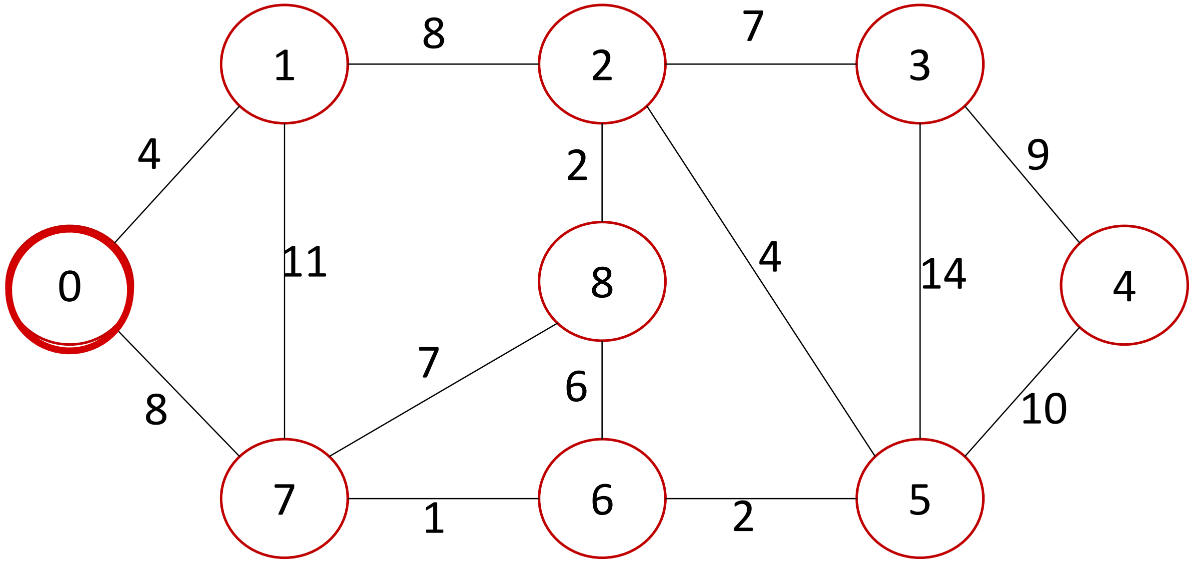
1. Create a set *mst* to keep track of vertices included in MST.
2. Also keep track of parent of each vertex. Initialize parent of each vertex -1.
3. Assign a key to all vertices in the input graph. Key for all vertices should be initialized to INF. The start vertex key should be 0.
4. While *mst* doesn't include all vertices
   i. Pick a vertex u which is not there in *mst* and has minimum key.
   ii. Include vertex u to *mst*.
   iii. Update key and parent of all adjacent vertices of u.
      a. For each adjacent vertex v, if weight of edge u-v is less than the current key of v, then update the key as weight of u-v.
      b. Record u as parent of v.



key(v) = weight(u,v)

Dijkstra's Algorithm – single src shortest path algo (SPT)

1. Create a set *spt* to keep track of vertices included in shortest path tree.
2. Track distance of all vertices in the input graph. Distance for all vertices should be initialized to INF. The start vertex distance should be 0.
3. While *spt* doesn't include all vertices
   i. Pick a vertex u which is not there in *spt* and has minimum distance.
   ii. Include vertex u to *spt*.
   iii. Update distances of all adjacent vertices of u. For each adjacent vertex v, if distance of u + weight of edge u-v is less than the current distance of v, then update its distance as distance of u + weight of edge u-v.



dist[v] = dist[u] + weight(u,v);

Dijkstra's SPT – Analysis

1. Create a set *spt* to keep track of vertices included in shortest path tree.
2. Track distance of all vertices in the input graph. Distance for all vertices should be initialized to INF. The start vertex distance should be 0.
3. While *spt* doesn't include all vertices
   1. Pick a vertex u which is not there in *spt* and has minimum distance.
   2. Include vertex u to *spt*.
   3. Update distances of all adjacent vertices of u. For each adjacent vertex v, if distance of u + weight of edge u-v is less than the current distance of v, then update its distance as distance of u + weight of edge u-v.

- Time complexity (adjacency matrix)
  - V vertices: O(V)
  - get min key vertex: O(V)
  - update adjacent: O(V)
- Time complexity (adjacency matrix)
  - $O(V^2)$
- Time complexity (adjacency list)
  - V vertices: O(V)
  - get min key vertex: O(log V)
  - update adjacent: O(E) – E edges
- Time complexity (adjacency list)
  - O(E log V)

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>