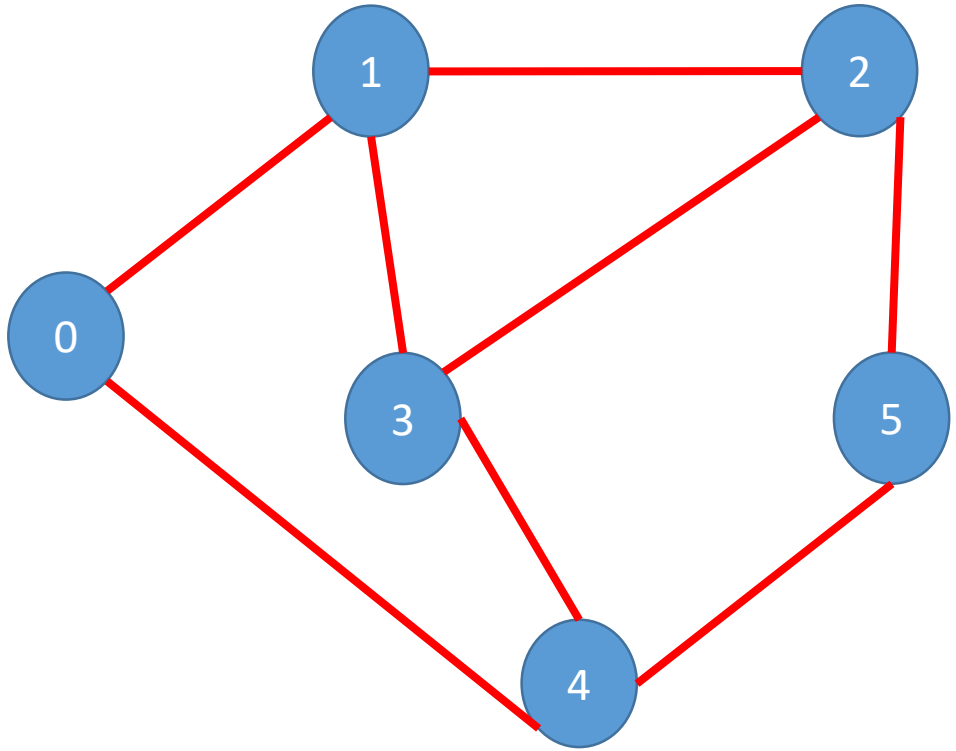
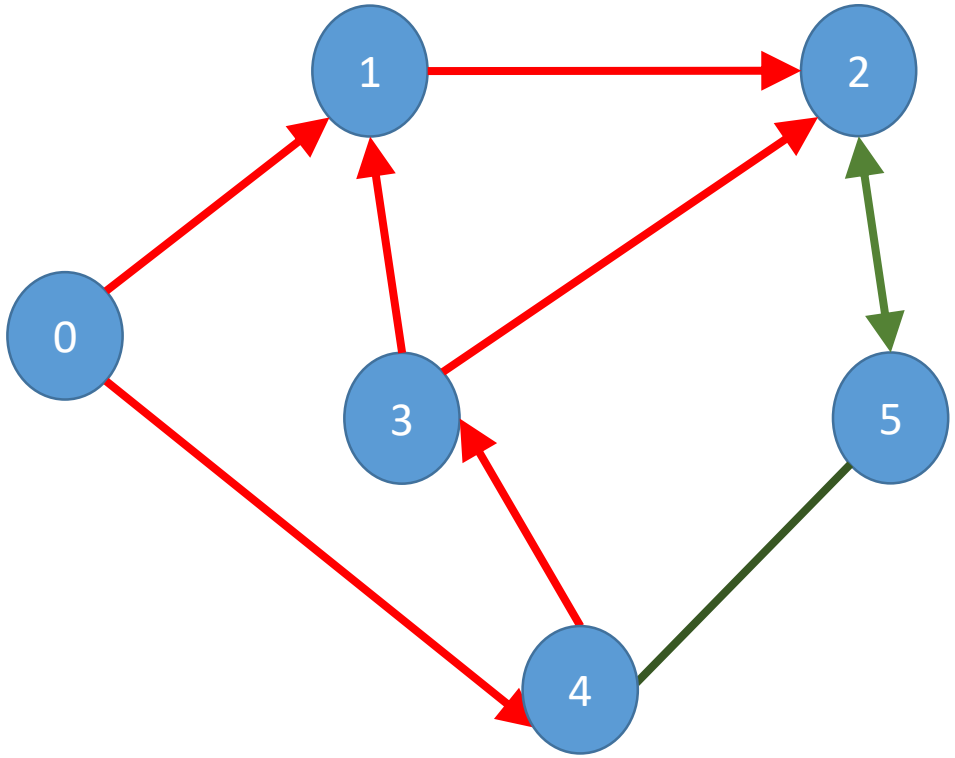
Graph: A Non-Linear Data Structure

Joy Mukherjee

Graph

- G = (V, E)
- V: Set of vertices
- E: Set of edges

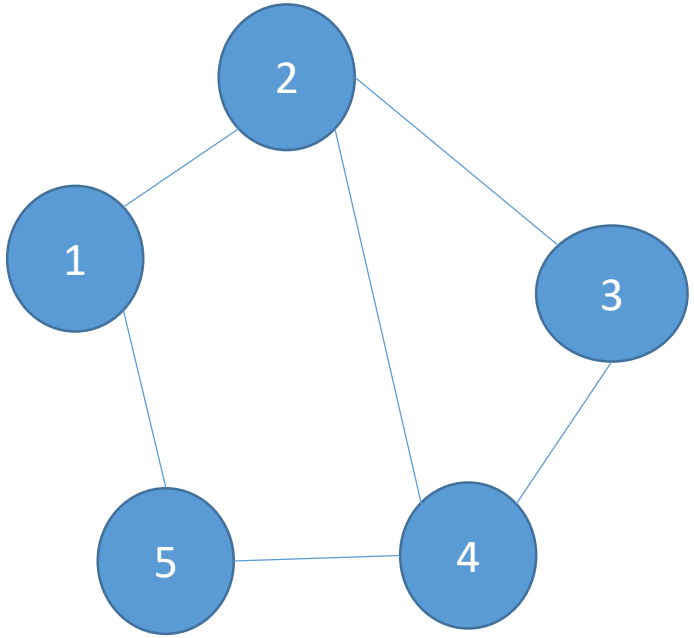


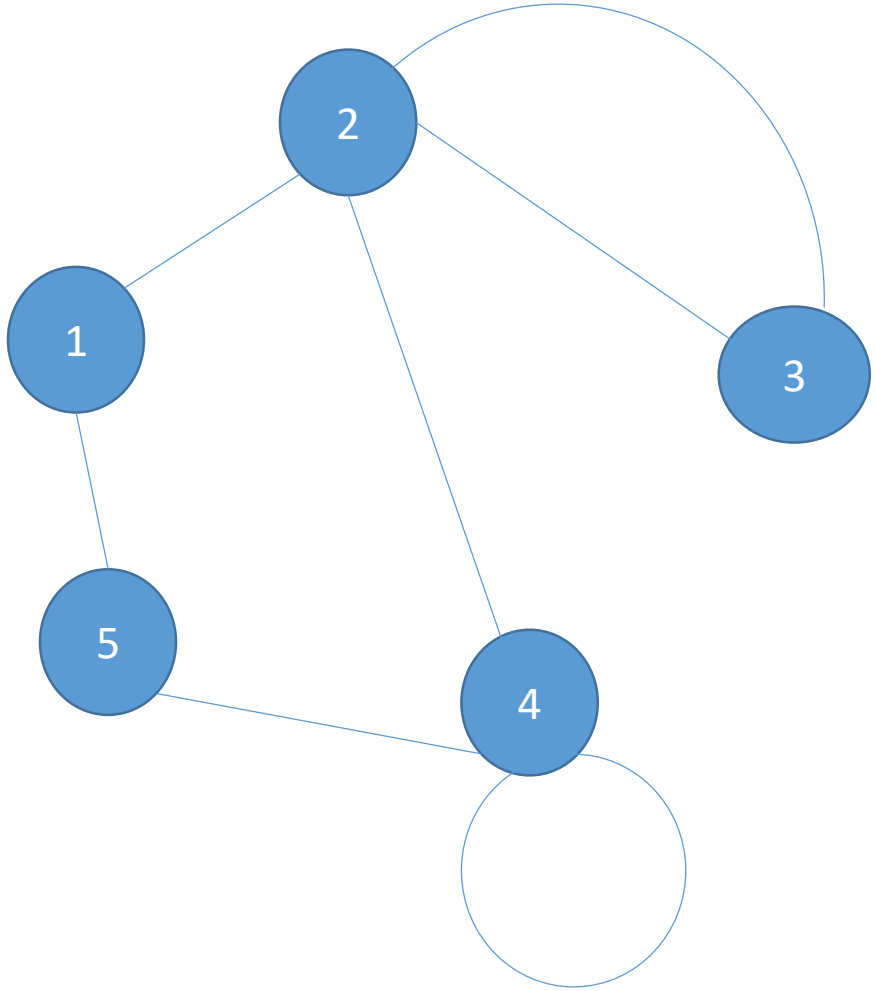


Loop, Parallel Edges and Simple Graph

- A loop is an edge whose endpoints are equal.
- Parallel edges are edges having the same pair of endpoints.
- A simple graph is a graph having no loops or parallel edges.

Simple vs. Non-Simple Graph





Degree, Order & Size

- Degree of a vertex v in graph G, ($d_G(v)$ or $d(v)$) = Number of non-loop edges containing v + 2 X Number of loops containing v.
  - The maximum degree is $\Delta(G)$
  - The minimum degree is $\delta(G)$
  - A graph is regular if $\Delta(G) = \delta(G)$
  - A graph is k-regular if $\Delta(G) = \delta(G) = k$
- Order of a graph G = $|V|$
- Size of a graph G = $|E|$

Handshaking Lemma: Undirected Graph

- Theorem: If G is an undirected graph, then $\sum_{v \in V(G)} d(v) = 2|E|$.
- Proof: Each edge has two endpoints.

  It contributes to the degree at each endpoint.

  A loop also contributes two to the degree of its endpoint.

Handshaking Lemma: Directed Graph

- Theorem: If G is a directed graph, then $\sum_{v \in V(G)} \text{indegree}(v) = \sum_{v \in V(G)} \text{outdegree}(v)$
- Proof: Each directed edge e = (u, v) has two endpoints which is directed from u to v.

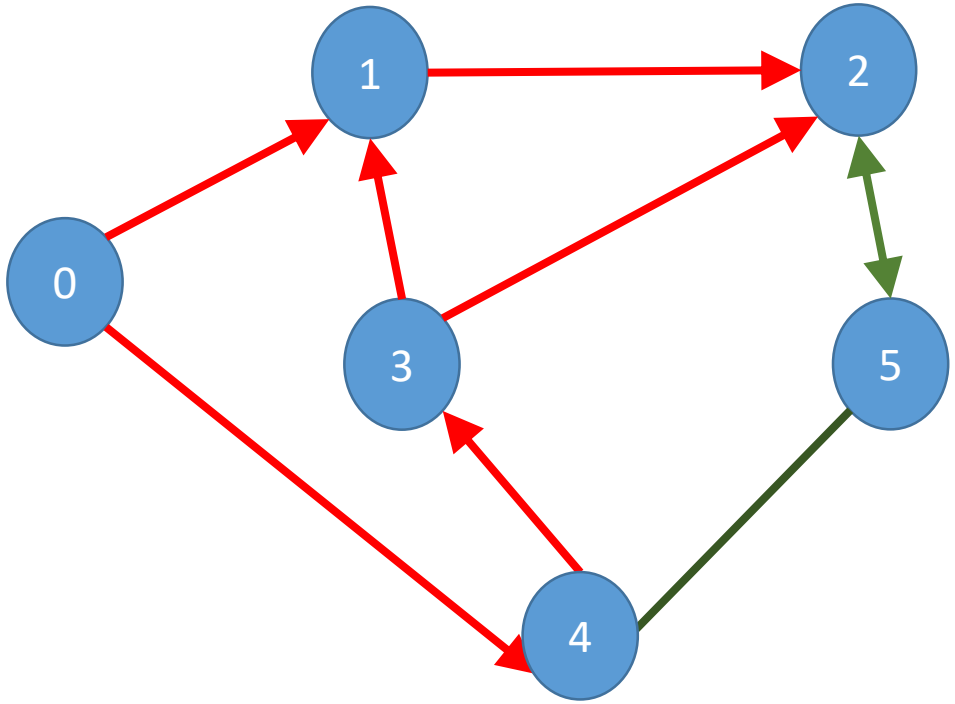
  The incoming edge towards v contributes to its indegree.

  The outgoing edge from u contributes to its outdegree.

  For a loop at a vertex u, the edge contributes to its indegree and outdegree.

Representation of Graph

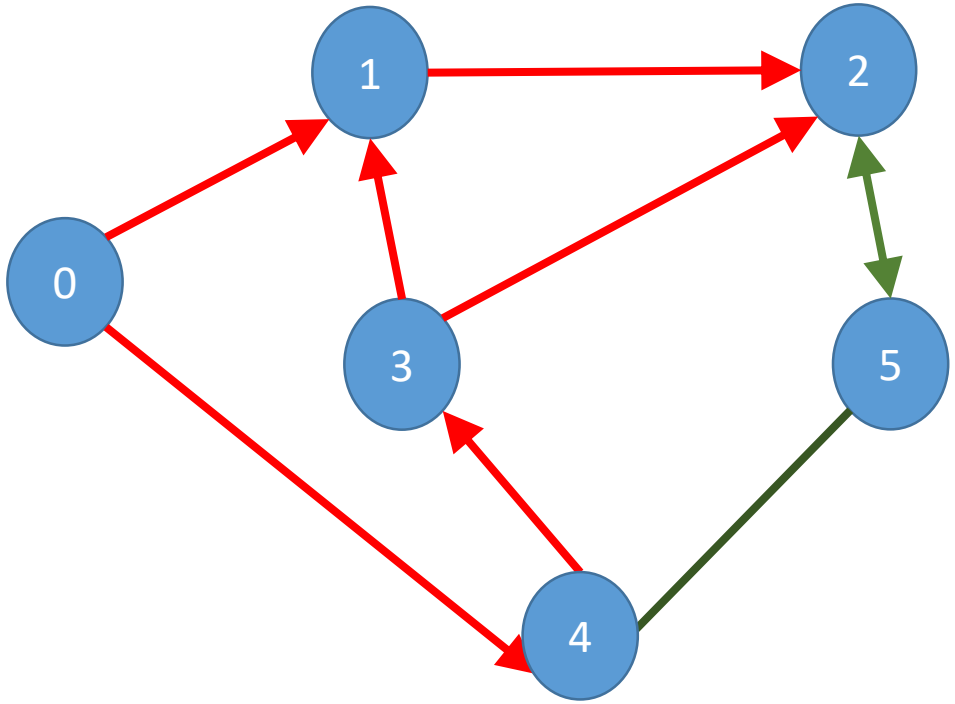
- Adjacency Matrix: Given a graph G=(V, E), we create a matrix M of size |V| X |V|, where M[i][j] is 1 if there is an edge from vertex i to vertex j; otherwise it is 0.
- O(|V|²).
- For undirected graph (no edge has a direction), M is a symmetric matrix.

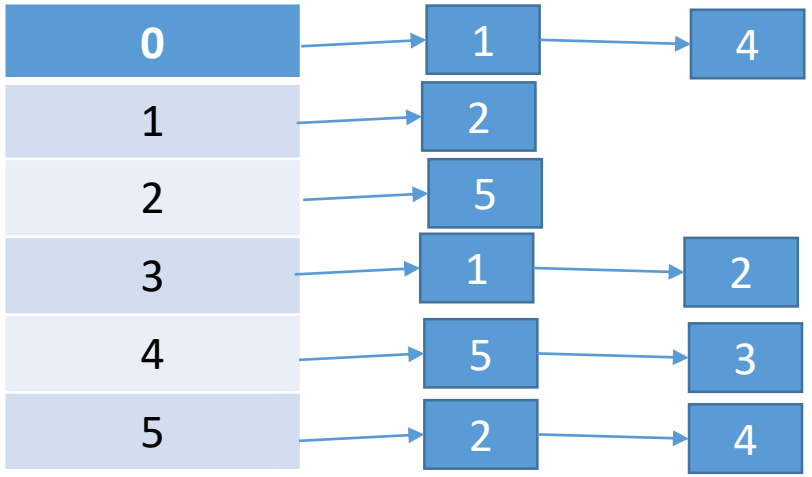


M	0	1	2	3	4	5
0	0	1	0	0	1	0
1	0	0	1	0	0	0
2	0	0	0	0	0	1
3	0	1	1	0	0	0
4	0	0	0	1	0	1
5	0	0	1	0	1	0

Representation of Graph

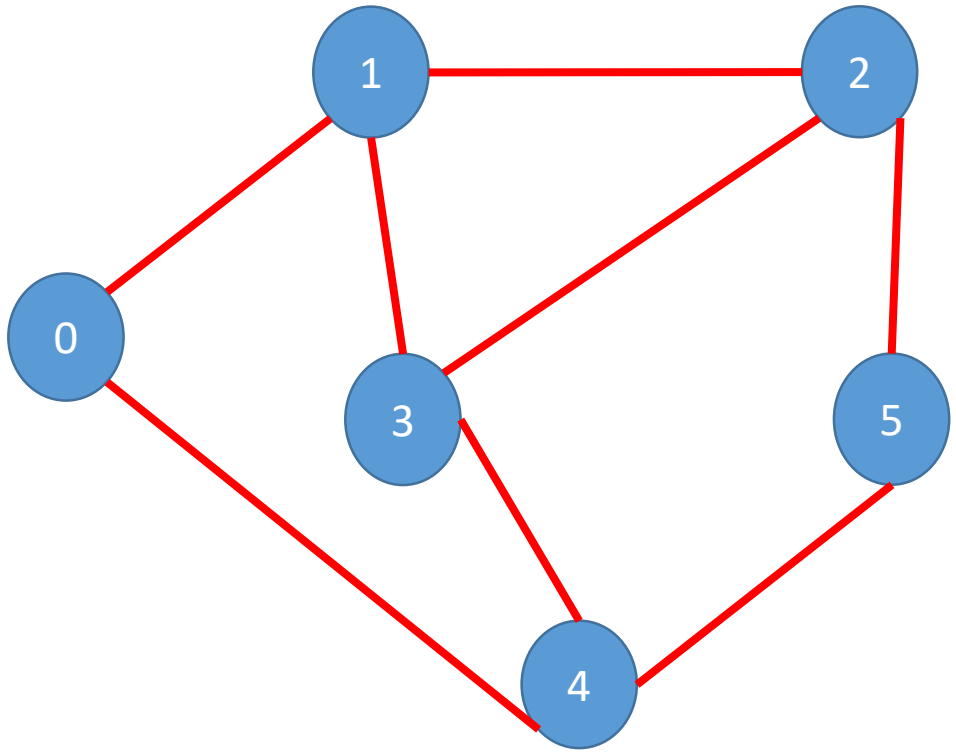
- Adjacency List: Array of Linked Lists, where the Array size is |V|, the linked list corresponding to a vertex v has all its neighbors. O(|V| + |E|)





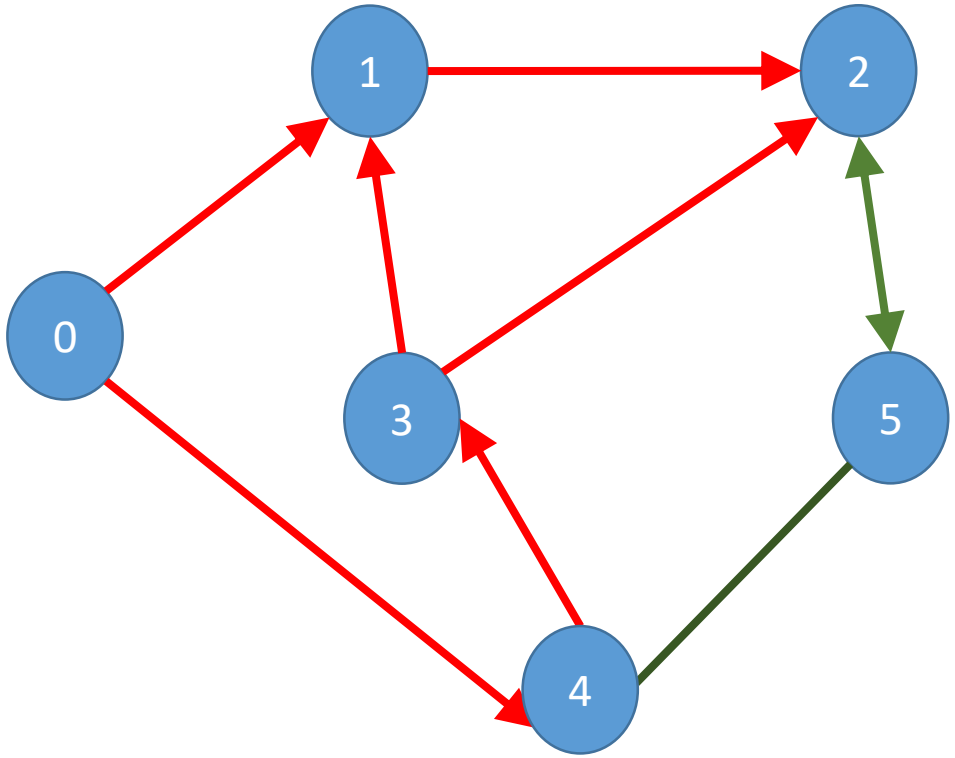
Undirected & Directed Graphs

Degree of a Vertex: Number of neighbors of the vertex



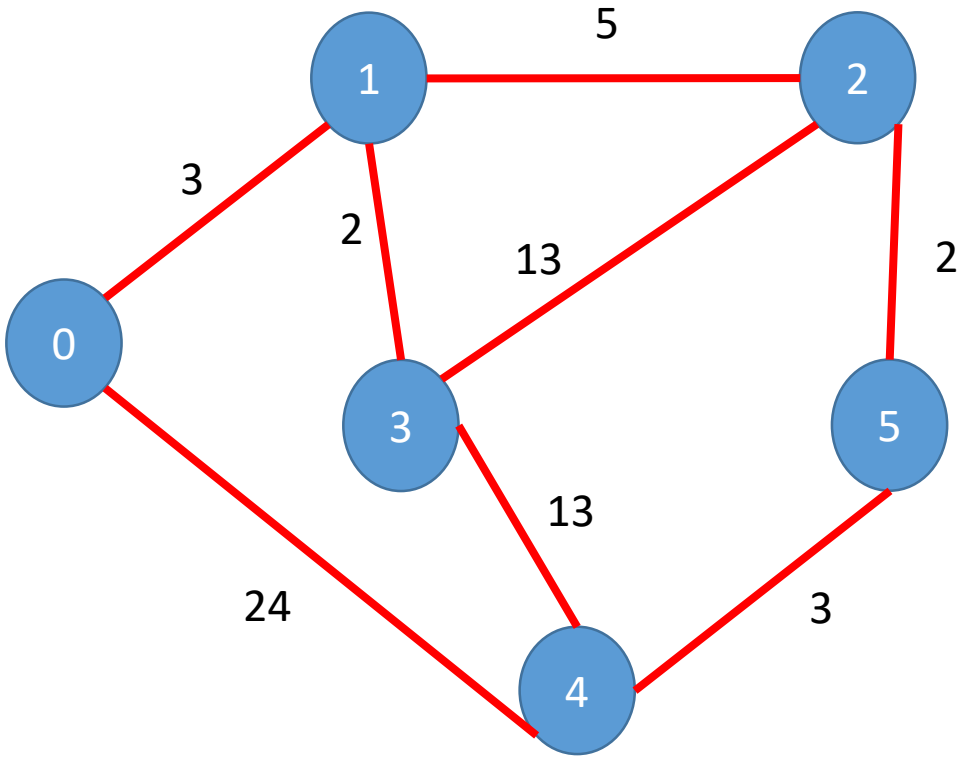
Indegree of a Vertex: Number of incoming edges

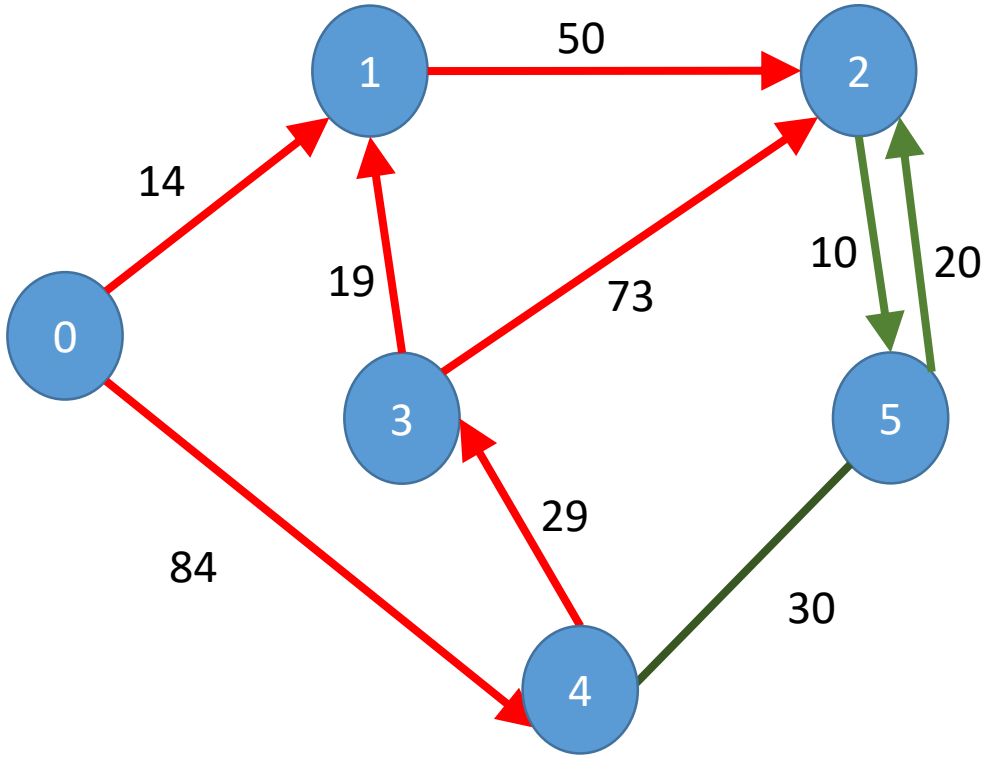
Outdegree of a Vertex: Number of outgoing edges



Weighted Graph

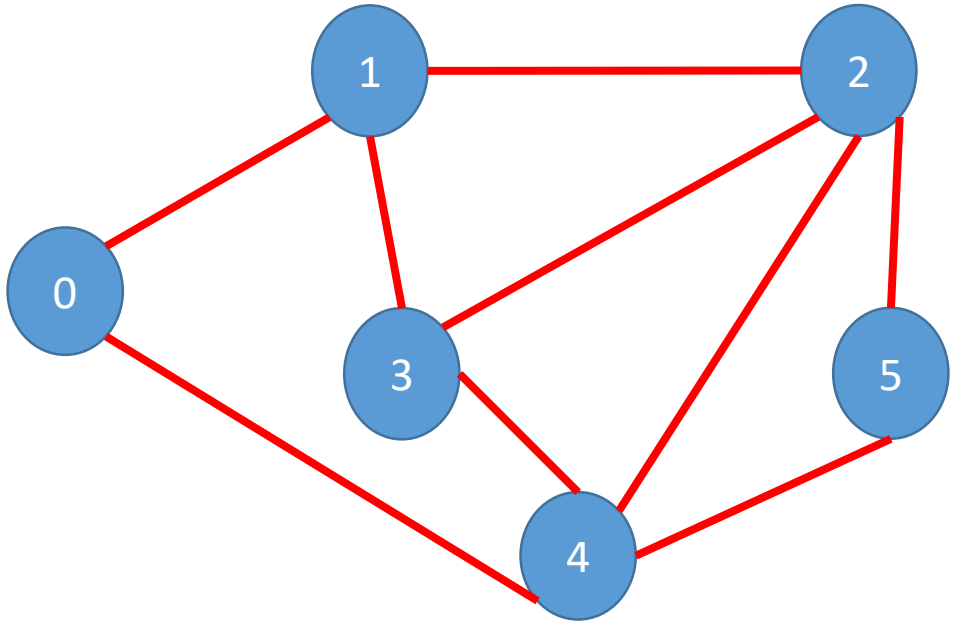
- G = (V, E, W)
- V: Set of vertices
- E: Set of edges
- W: E -> R





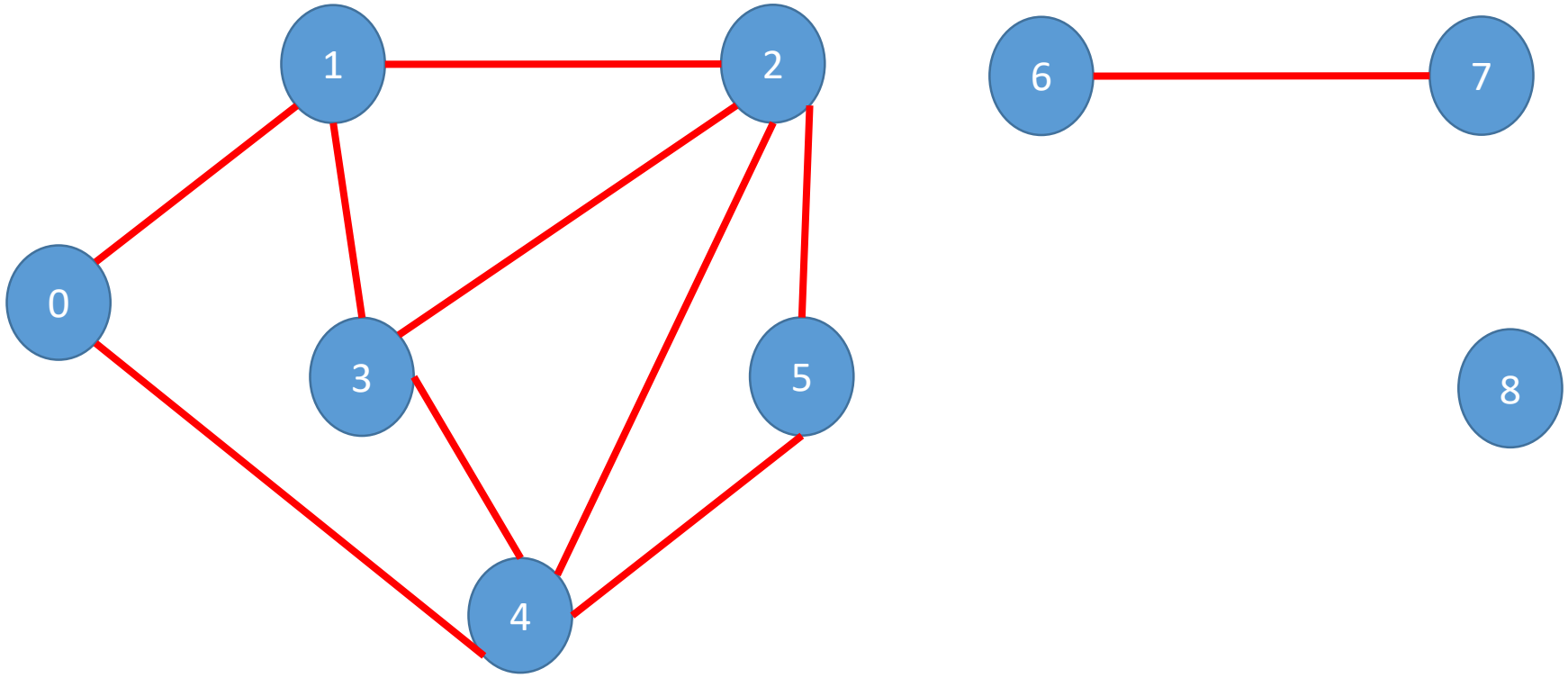
Walk, Trail, Path

- Walk: Sequence of vertices in the graph where adjacent vertices must have an edge between them. Example: 1 2 5 4 3 2 5 is a walk
- Trail: A walk with no repeated edge. Example: 1 2 5 4 3 2 4 is a trail. In trail, vertex can be repeated.
- Path: A trail/walk with no repeated vertex. Example: 1 2 5 4 3 is a path. If no vertex is repeated, then edges can't be repeated.



Connected Graph

- An undirected graph is called a connected graph if there is a path between every pair of vertices.



Graph Traversal Algorithms

- Given a vertex s (source vertex) in a graph G=(V, E), how do we systematically explore the other vertices V – {s} in G?

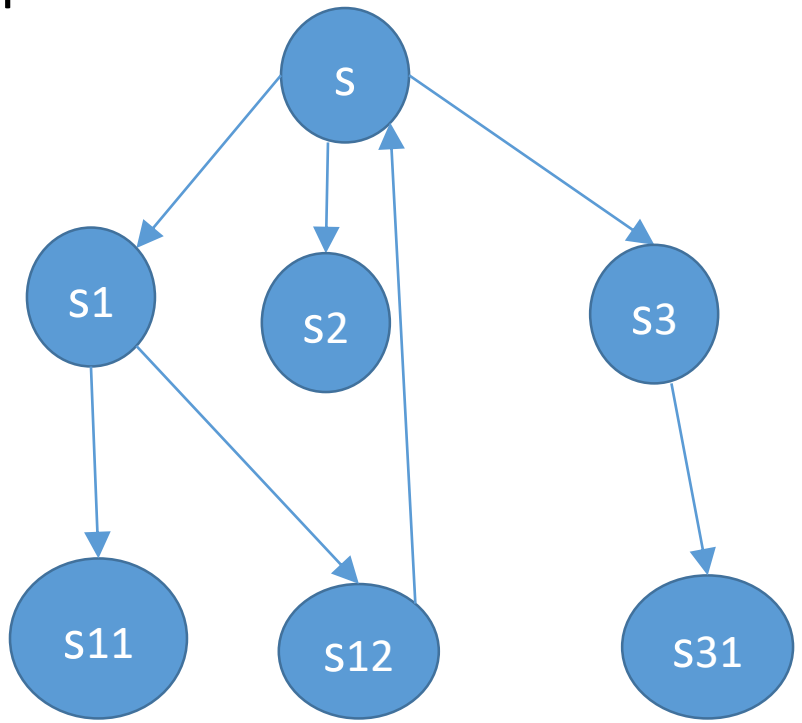
- Breadth-First Search

- Depth-First Search

Breadth First Traversal/Search (BFS)

- Input: G=(V, E)
- Choose a vertex s, and explore its neighbors first, subsequently exploring the neighbors of the neighbors.
- For an undirected unweighted graph G, BFS gives shortest path from s to all other reachable vertices of the graph G.

s, s1, s2, s3, s11, s12, s31



If there is a directed edge from a node to its ancestor in the BFS tree, then cycle is detected.

Data Structure

- For each node x, we maintain three variables
  1. Distance of x from the source s: dist
  2. Parent of x: parent
  3. Is x already visited during BFS: visited

- Linear Queue

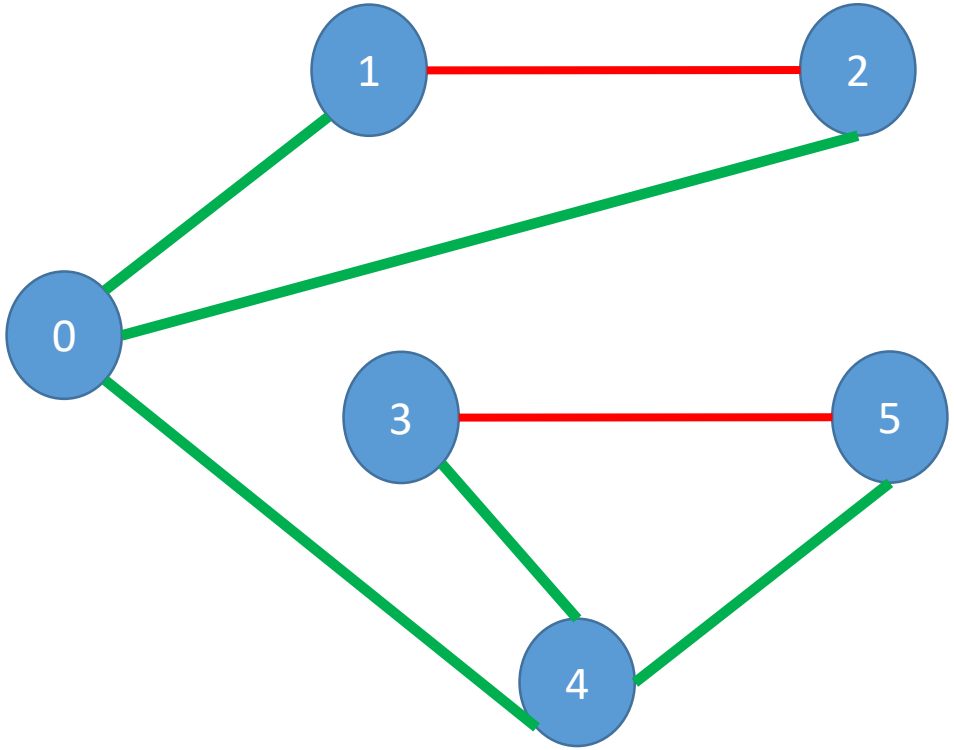
BFS Algorithm

BFS(G, s)

1. For all vertex x in V – {s} dist[x] = inf parent[x] = NULL visited[x] = 0
2. dist[s] = 0 parent[s] = NULL visited[s] = 1
3. Initialize the Queue Q
4. enqueue(Q, s)
5. While(Q is not empty) {
   a. u = dequeue(Q)
   b. For each v in Adj[u]
      If(visited[v] == 0)
         i. visited[v] = 1
         ii. dist[v] = dist[u] + 1
         iii. parent[v] = u
         iv. enqueue(Q, v)

   }

Breadth First Traversal/Search (BFS)



BFS gives us a BFS tree rooted at s, the green edges are called tree edges, and the red edges are called non-tree edges.

Implementation: Linear Queue

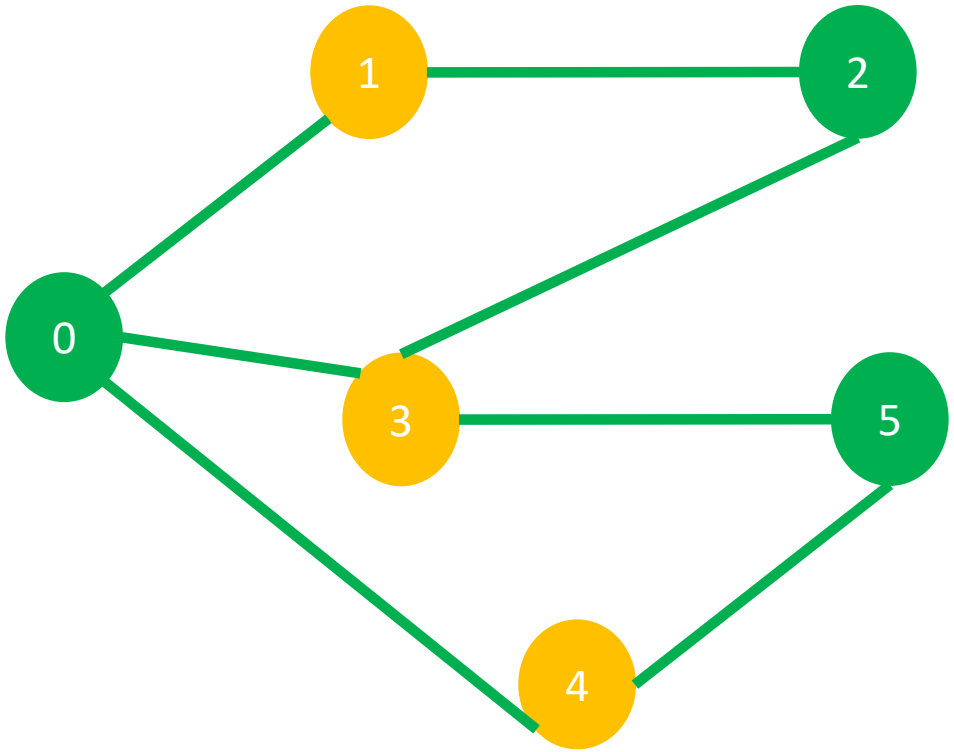
0 |2 1 4 | 3 5

Queue						
Dist	0	1	1	2	1	2
Parent	N	0	0	4	0	4
Visited	1	1	1	1	1	1

Application: BFS

- **Shortest Path from a given vertex in an unweighted graph**.
- **Cycle detection in an undirected graph**: During BFS, if we encounter a vertex v that is already visited and v is not a parent of the current vertex, then “a cycle is detected”.
- **Detecting whether a graph is bipartite or not**.
- A graph G= (V, E) is called a bipartite graph, if its vertices can be partitioned into two sets A and B such that A U B = V and $A \cap B = NULL$, and all the edges are between A and B.
- Bipartite graph does not have an odd cycle.
- Bipartite graph is 2-colourable.

Check for Bipartite Graph

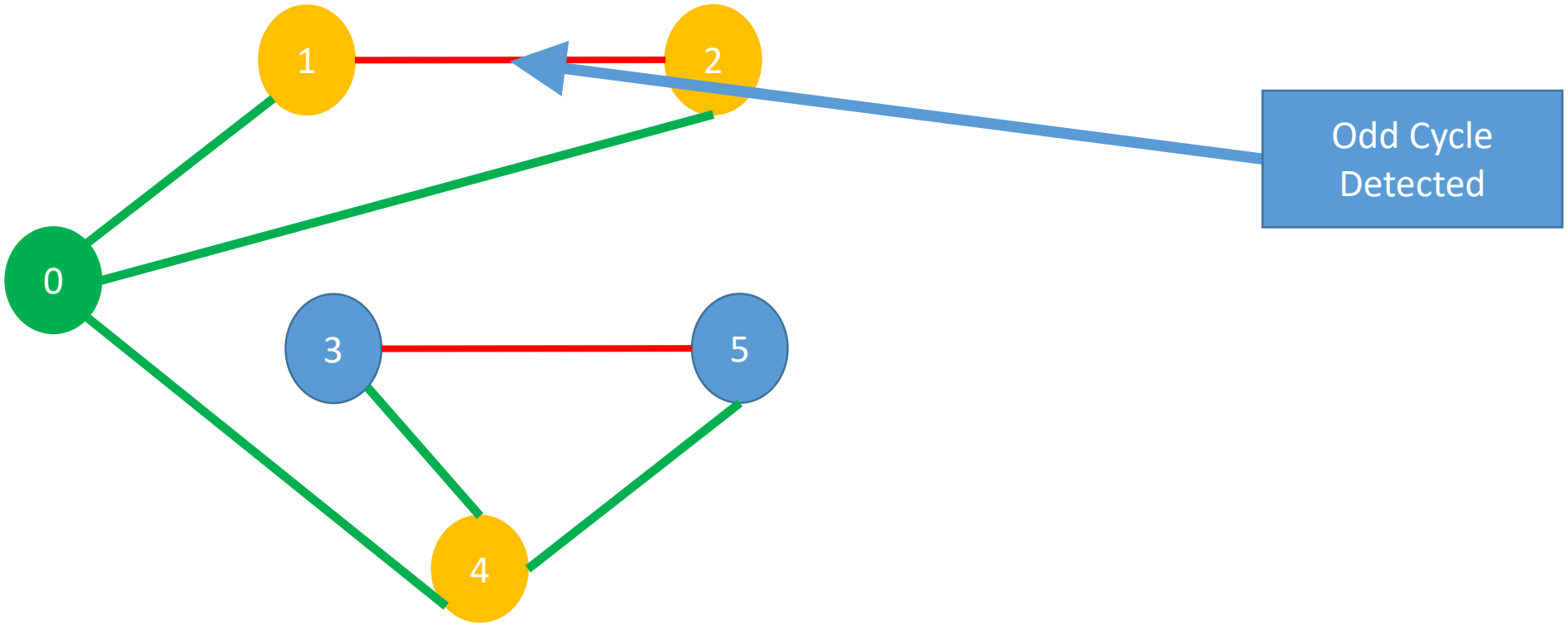


A = All vertices that are at even distance from the source
B = All vertices that are at odd distance from the source

A = {0, 2, 5}
B = {1, 3, 4}

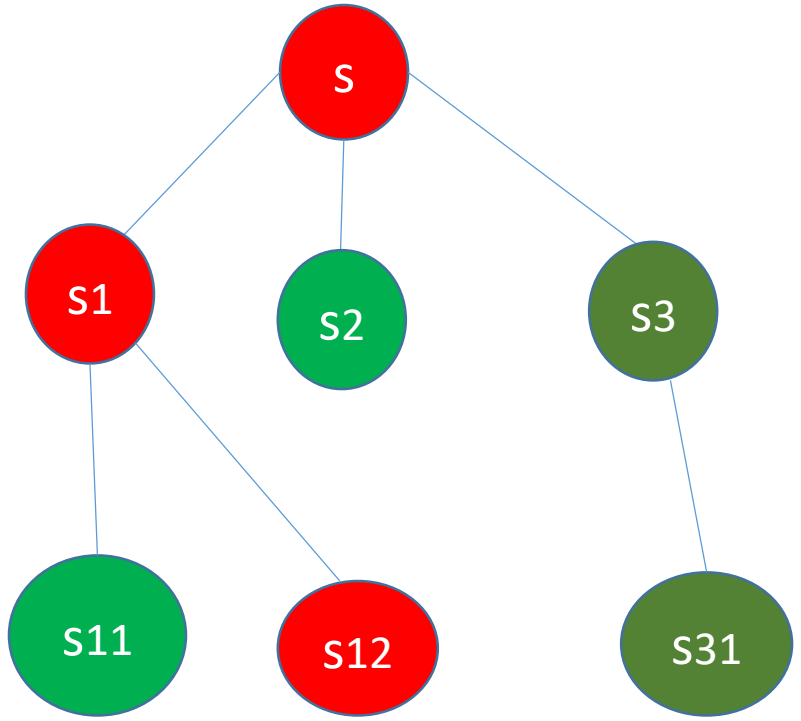
During colouring of neighbours of v, if a neighbour of v has already coloured with the same colour as of v, then the graph is not bipartite.

Check for Bipartite Graph



Depth First Traversal/Search (DFS)

- Given a graph G = (V, E), and a source vertex s, we explore s first, followed by any one neighbour v of s, and we do it recursively from v unless we encounter a dead end, where we start backtracking to explore the rest of the vertices not yet explored.



Data Structure

- For each node x, we maintain three variables
  1. Parent of x: parent
  2. Is x already visited during BFS: visited

- Stack

DFS Algorithm

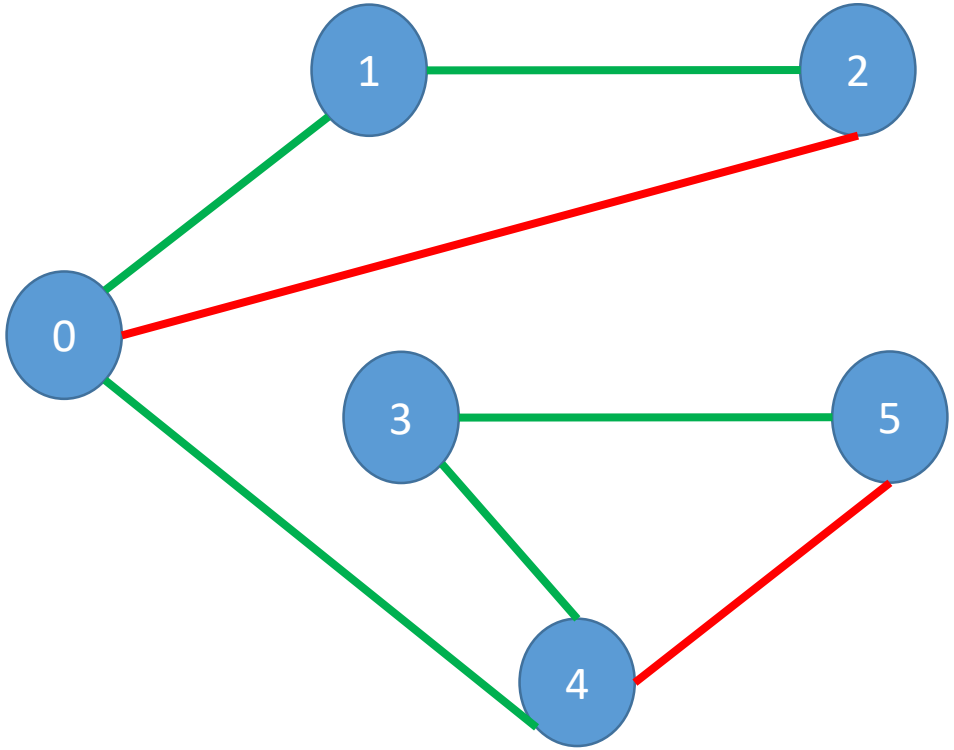
- DFS(G)

1. For all vertex x in V
   parent[x] = NULL
   visited[x] = 0
2. time = 0
3. For each vertex u in V
   If(visited[u] == 0)
   DFS_VISIT(G, u)

- DFS_VISIT(G, u)

1. visited[u] = 1
2. disc[u] = time = time+1
3. For each v in Adj[u]
   a. If(visited[v] == 0)
      i. parent[v] = u
      ii. DFS_VISIT(G, v)
4. finish[u] = time = time + 1

Depth First Traversal/Search (DFS)



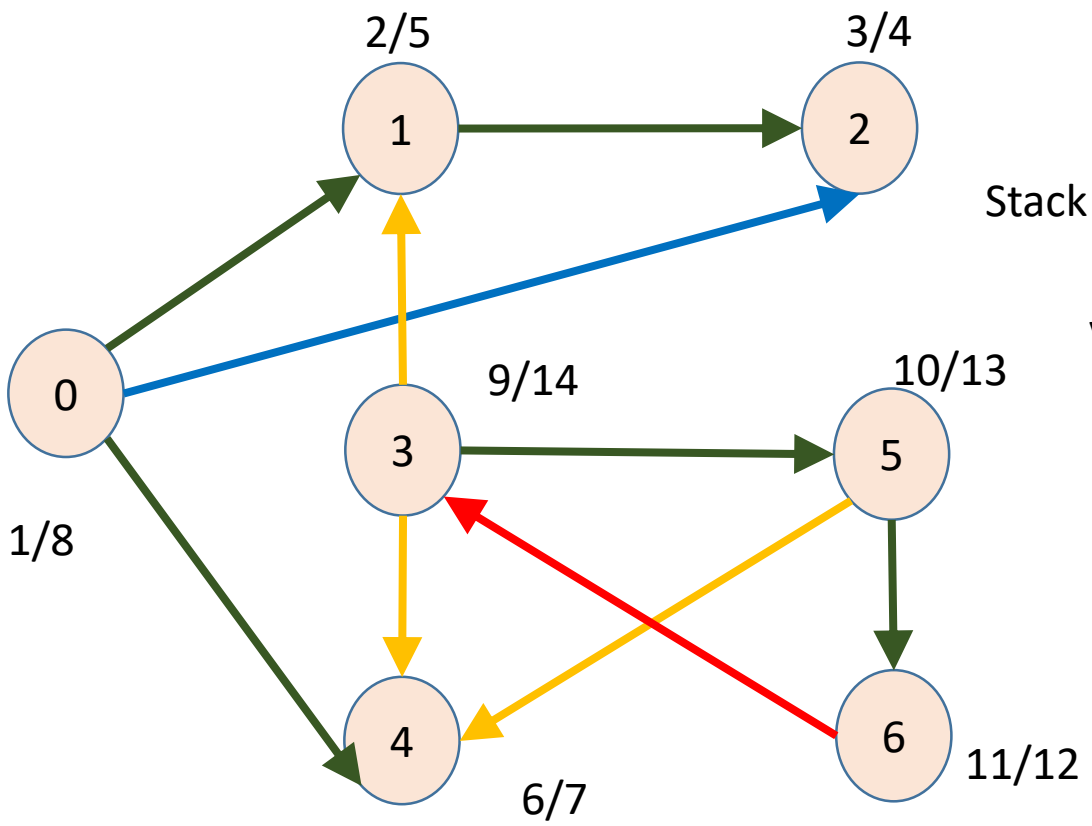
0 2 1 4 3 5

Implementation: Stack

Parent	Null	0	1	4	0	3
Visited	1	1	1	1	1	1
Start Time	1	8	9	3	2	4
Finish Time	12	11	10	6	7	5

Stack

Depth First Traversal/Search (DFS)



0 1 2 4 3 5 6

Stack

V

P

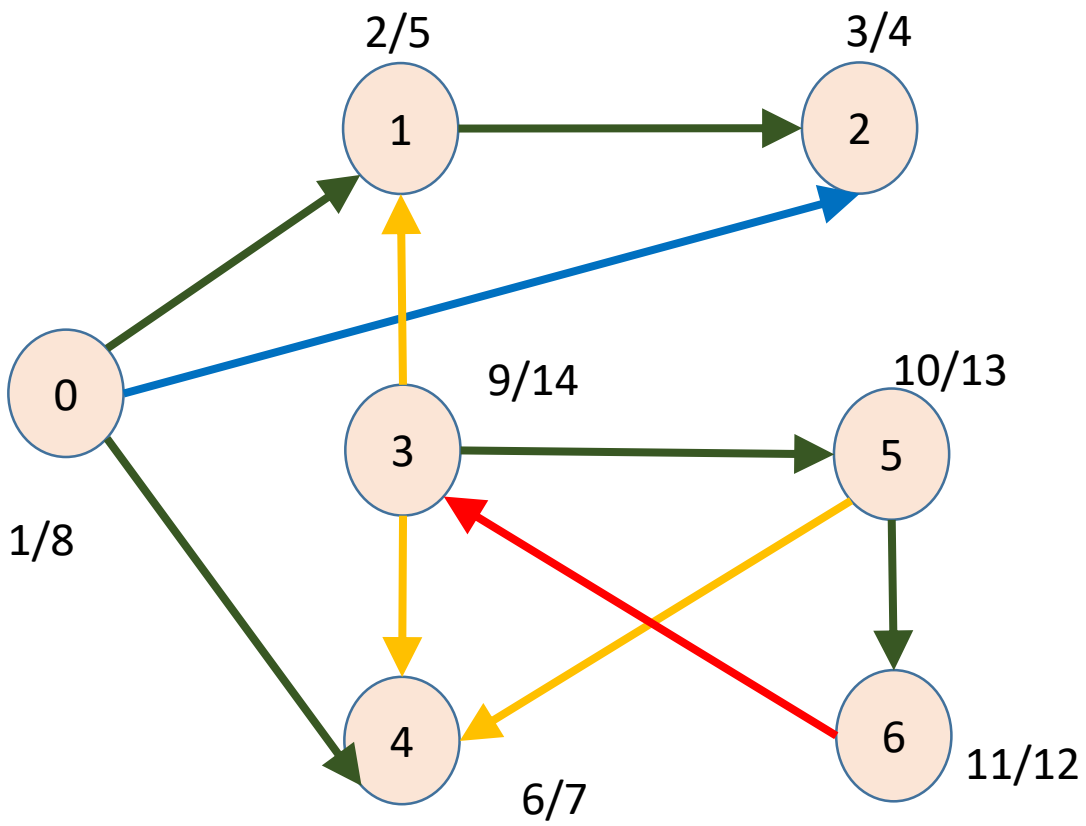
Discovery time/Finish Time

Discovery time: The node found for the first time

Finish time: The time at which the backtracking happens

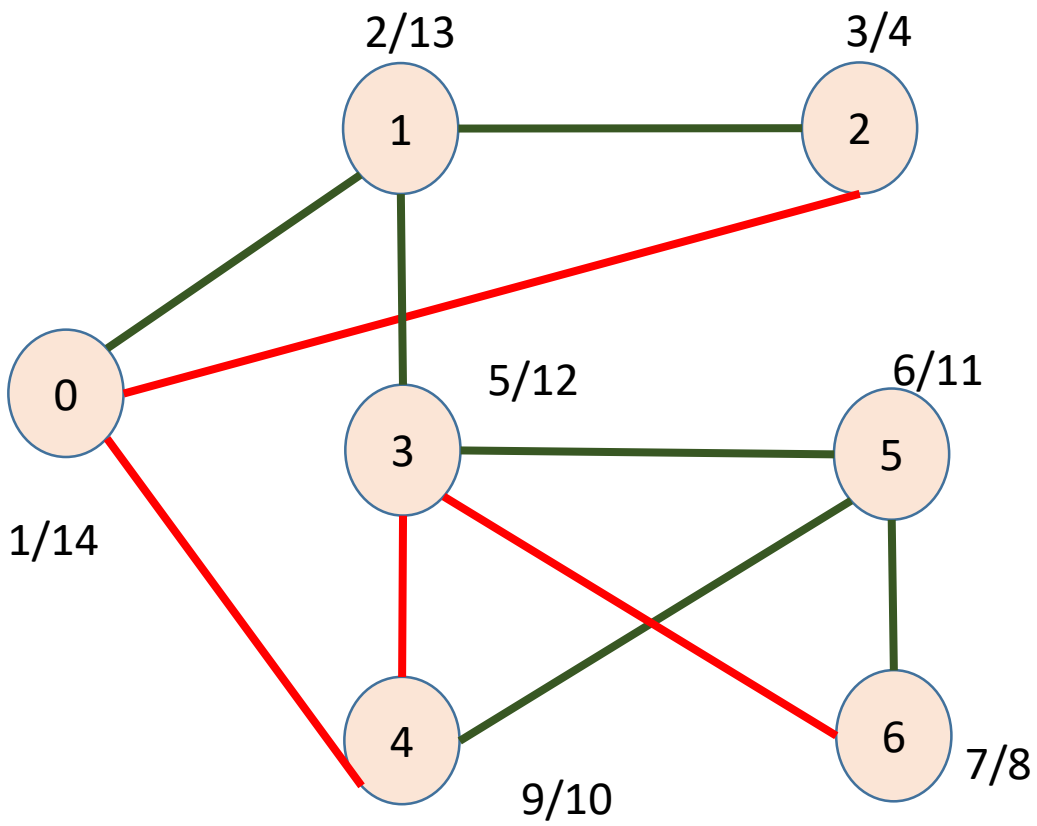
DFS gives us a DFS forest, the green edges are called tree edges, the red edges are called back edges, the blue edges are called forward edges, and the orange edges are called cross edges.

DFS on Directed Graph



- Tree Edge (u, v) :
  1. u is parent of v
  2. d[u] < d[v] < f[v] < f[u]
- Forward Edge(u, v):
  1. u is not the parent of v
  2. d[u] < d[v] < f[v] < f[u]
- Back Edge (u, v): [Cycle]
  1. d[v] < d[u] < f[u] < f[v]
- Cross Edge (u, v):
  1. (d[u], f[u]) is non-overlapping with (d[v], f[v])
  2. d[v] < f[v] < d[u] < f[u]

DFS on Undirected Graph



- Tree Edge (u, v) :
  1. u is parent of v
  2. d[u] < d[v] < f[v] < f[u]

- Back Edge (u, v): [Cycle]
  1. d[v] < d[u] < f[u] < f[v]

Application: DFS

- **Cycle detection in a graph**
  - **Undirected Graph:** If it is a tree, then no cycle is detected. Check for back edges. If exists, then a cycle is detected.
  - **Directed Graph:** If a back edge is found, then a cycle is detected.