Graph Algorithms

A Graph G=(V,E) where V is the set of Vertices and E is the set of edges (pair wise connections between vertices)

Path: Sequence of vertices connected by edges.

Graph problems :

- Given two vertices v and w is there a path between them?
- What is the shortest path between two vertices
- All pairs shortest path
- Is the Graph connected?
- Does the graph have a cycle?
- Is there a cyclic path that touches every edge in the graph exactly once (**Eulerian tour**)
- Is there a cyclic path that touches every vertex exactly once (**Hamiltonian tour**)

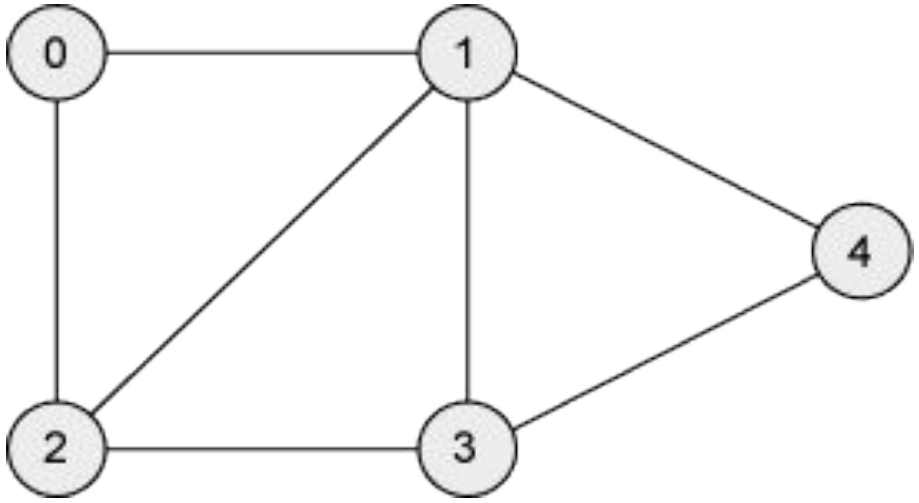
A Graph G=(V,E) where V is the set of Vertices and E is the set of edges (pair wise connections between vertices)

Graph problems (Cont…):

- MST (Minimum Spanning Tree): Spanning Tree with minimum total weight
- Biconnectivity: Is there a vertex whose removal disconnects the graph?
- Planarity: Can the Graph be drawn the plane with no crossing edges?
- Graph Isomorphism : Do two adjacency lists represent the same graph?

Depth First Search

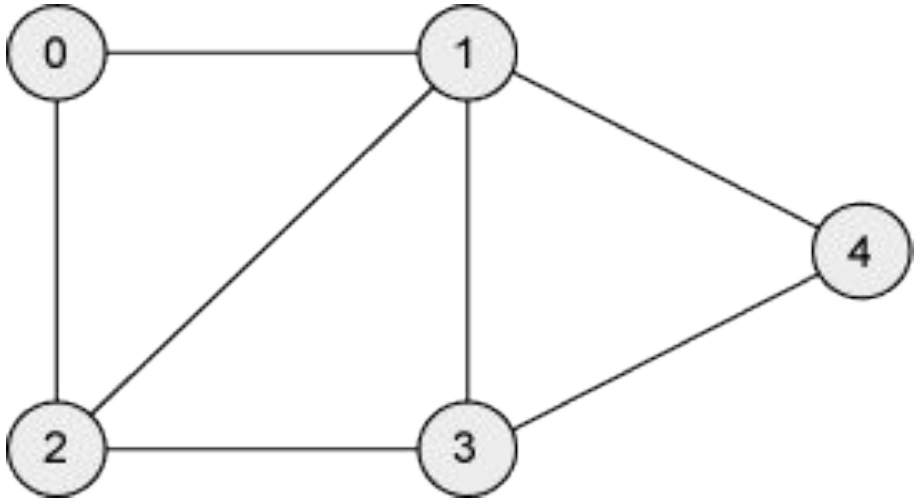
```
DFS(graph, start_vertex):
    Initialize an empty set or list to keep track of visited vertices: visited = {}
    Call the recursive helper function: DFS_Util(graph, start_vertex, visited)

DFS_Util(graph, vertex, visited):
    Mark the current vertex as visited: visited.add(vertex)
    Perform an action on the current vertex (e.g., print vertex or process vertex's
value)

    For each neighbor in graph.adjacent(vertex):  // Assuming
'graph.adjacent(vertex)' returns a list of adjacent vertices to 'vertex'
        If neighbor is not in visited:
            Recursively call DFS_Util on the neighbor: DFS_Util(graph, neighbor,
visited)
```




```
DFS(graph, start_vertex):
    Initialize an empty set or list to keep track of visited vertices: visited = {}
    Call the recursive helper function: DFS_Util(graph, start_vertex, visited)

DFS_Util(graph, vertex, visited):
    Mark the current vertex as visited: visited.add(vertex)
    Perform an action on the current vertex (e.g., print vertex or process vertex's value)

    For each neighbor in graph.adjacent(vertex):  // Assuming 'graph.adjacent(vertex)' returns a list of adjacent vertices to 'vertex'
        If neighbor is not in visited:
            Recursively call DFS_Util on the neighbor: DFS_Util(graph, neighbor, visited)
```

Breadth First Search (BFS)

```
BFS(graph, start_vertex):
    Create a queue, Q, and enqueue the start_vertex
    Initialize an empty set for visited vertices and add start_vertex to it
    While Q is not empty:
        vertex = Q.dequeue()  // Remove and return the front element of the queue
        Perform an action on vertex (e.g., print vertex or process vertex's value)

        For each neighbor in graph.adjacent(vertex):  // Assuming
'graph.adjacent(vertex)' returns a list of adjacent vertices to 'vertex'
            If neighbor is not in visited:
                Mark neighbor as visited (add to visited set)
                Enqueue neighbor to Q
```




```
BFS(graph, start_vertex):
    Create a queue, Q, and enqueue the start_vertex
    Initialize an empty set for visited vertices and add start_vertex to it
    While Q is not empty:
        vertex = Q.dequeue()  // Remove and return the front element of the queue
        Perform an action on vertex (e.g., print vertex or process vertex's value)

        For each neighbor in graph.adjacent(vertex):  // Assuming 'graph.adjacent(vertex)' returns a list of adjacent vertices to 'vertex'
            If neighbor is not in visited:
                Mark neighbor as visited (add to visited set)
                Enqueue neighbor to Q
```

Connected Components

Two vertices are connected if there is a path between them.
Connected component : maximal set of connected vertices

"Is connected to" is an equivalence relation for undirected graph
- Reflexive
- Symmetric
- Transitive

The equivalence classes are the connected components of the Graph.

Connected Components

```
Find_Connected_Components(graph):
    Initialize an empty list, connected_components, to store the components
    Initialize an empty set, visited, to keep track of visited vertices

    For each vertex v in graph:
        If v is not in visited:
            Create an empty list, component, to store the current component's vertices
            DFS(graph, v, visited, component)
            Add component to connected_components

    Return connected_components

DFS(graph, vertex, visited, component):
    Add vertex to visited set
    Add vertex to the current component list

    For each neighbor in graph.adjacent(vertex):  // Assuming 'graph.adjacent(vertex)' returns a list of adjacent vertices to 'vertex'
        If neighbor is not in visited:
            DFS(graph, neighbor, visited, component)
```

Directed Graph : Edges have directions

BFS and DFS work the same way as in Undirected Graphs

Topological Sorting : Given a set of tasks with precedence constraints, how to schedule them?
Topological Sort works on DAG (Directed Acyclic Graphs)

```
topologicalSort(graph):
    Initialize an empty stack to store the topological sort
    Mark all vertices as not visited

    For each vertex u in graph:
        if u is not visited:
            topologicalSortUtil(u, visited, stack)

    While stack is not empty:
        Pop a vertex from stack and print it (This is the topological sorted order)

topologicalSortUtil(vertex, visited, stack):
    Mark the current node as visited

    For each neighbor v of vertex:
        if v is not visited:
            topologicalSortUtil(v, visited, stack)

    Push the current node to stack (All vertices reachable from vertex are already in stack)
```

Strongly Connected Components

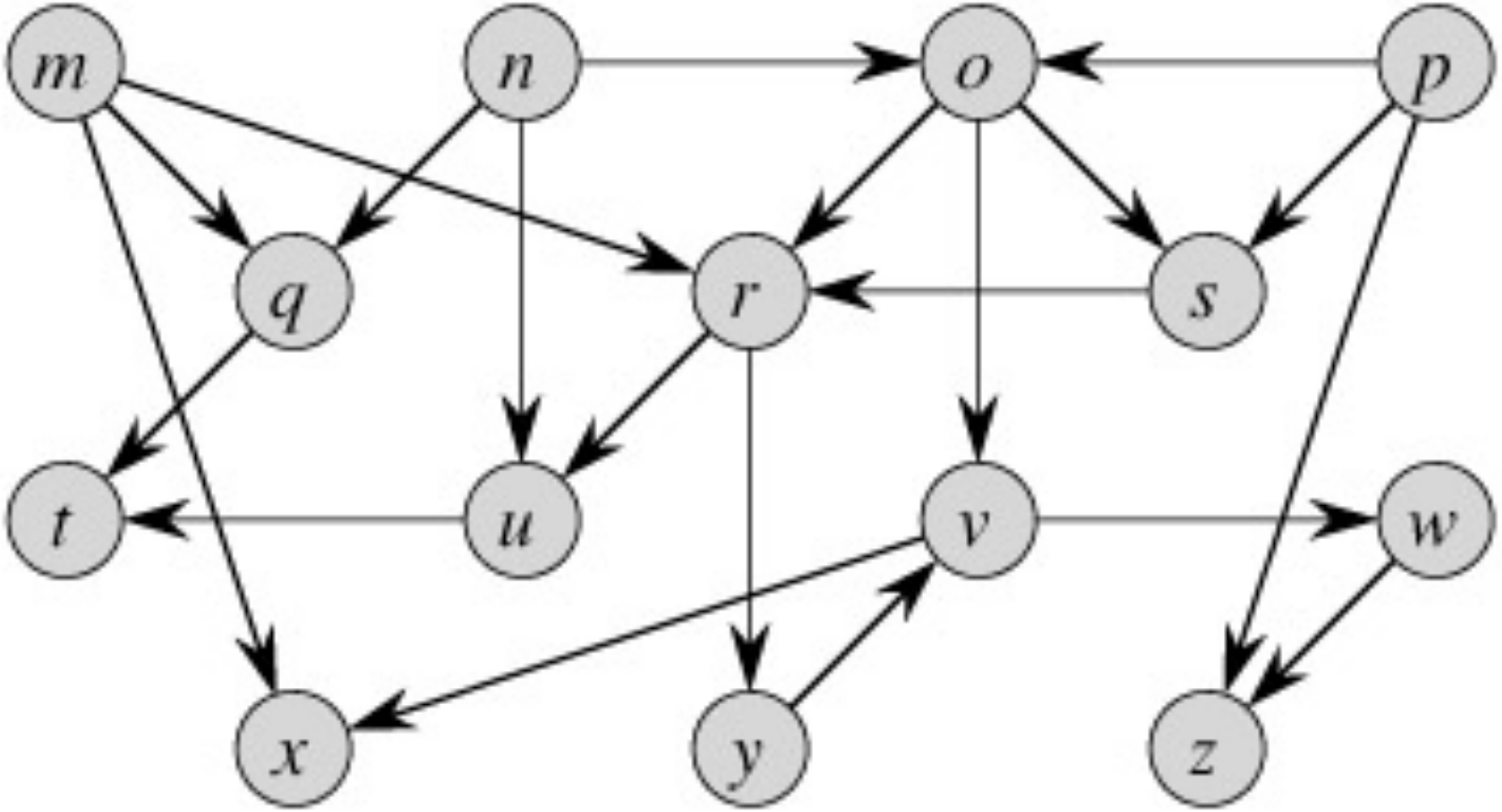
Vertices v and w are strongly connected if there is a directed path from v to w, and there is a directed path from w to v.

Is strongly connected to is an equivalence relation => Partitions the graph into strongly connected components

Kosaraju –Sharir Algorithm

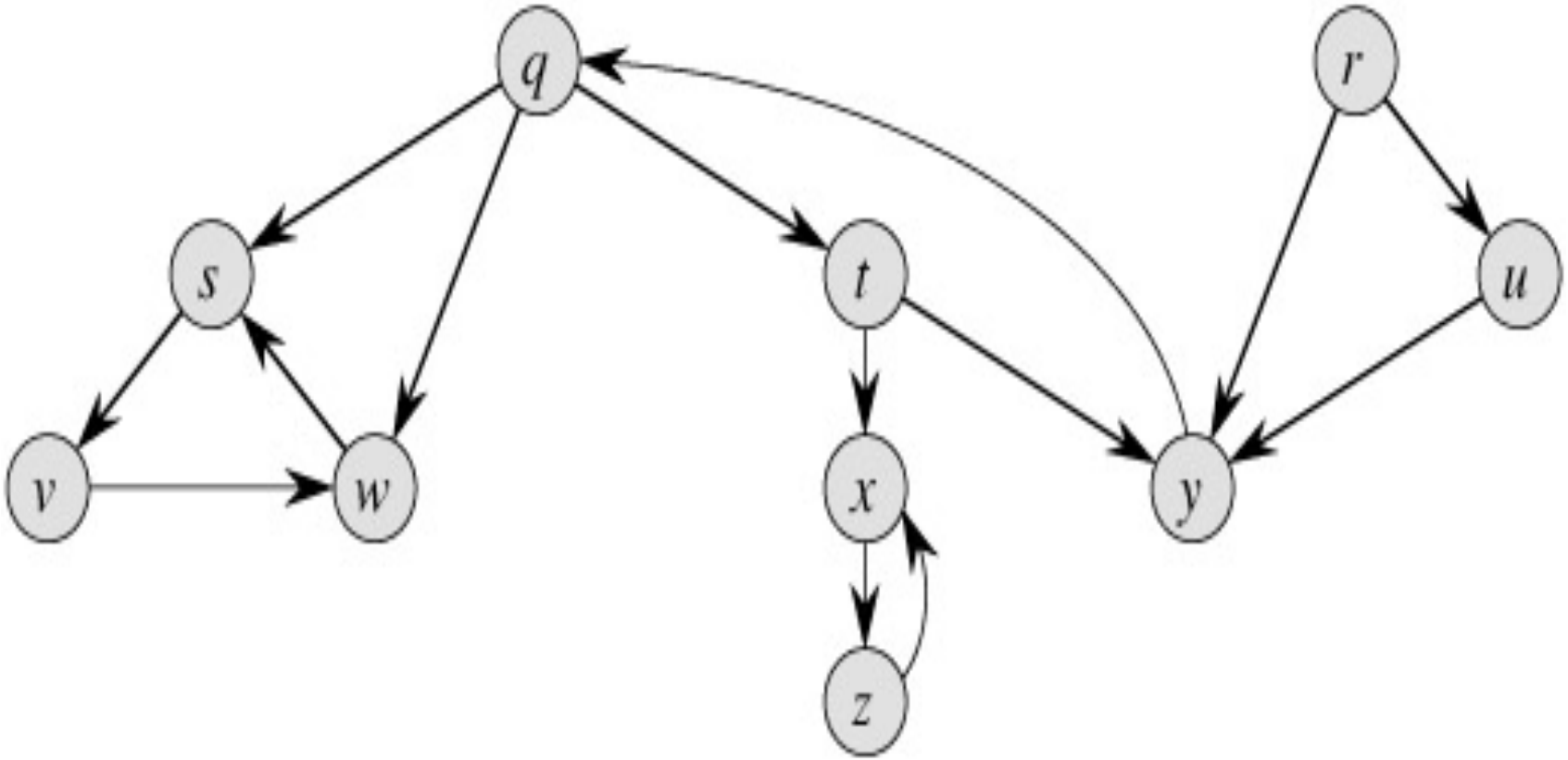
KosarajuSharir(graph):
1. Create an empty stack, 'stack', to hold vertices by finish time in decreasing order.
2. Mark all vertices as not visited.
3. For each vertex 'v' in the graph:
    if 'v' is not visited:
        fillOrder(v, visited, stack)
4. Generate the transpose of the graph, 'graphTranspose'.
5. Mark all vertices as not visited (for the second DFS).
6. While 'stack' is not empty:
    Pop a vertex 'v' from 'stack'.
    if 'v' is not visited in the transposed graph:
        DFS on 'graphTranspose' starting from 'v', and print/record the SCC.
        Print a new line or separate the SCCs distinctly.

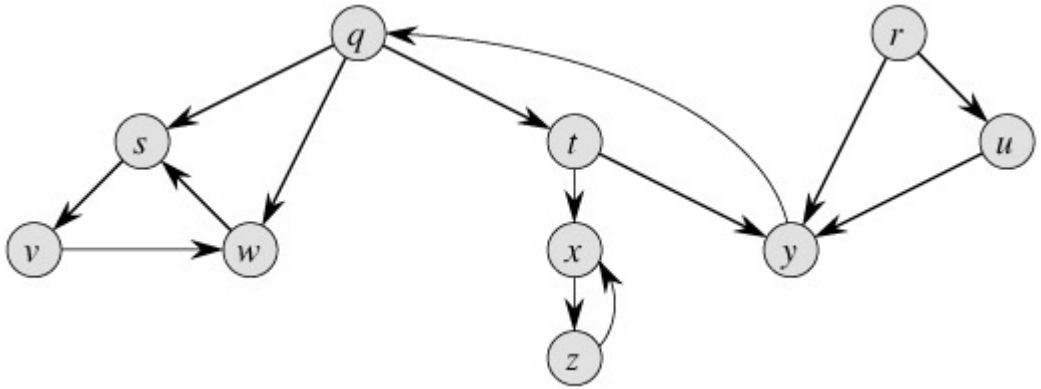
```
fillOrder(vertex, visited, stack):
    Mark 'vertex' as visited.
    For each neighbor 'n' of 'vertex':
        if 'n' is not visited:
            fillOrder(n, visited, stack)
    Push 'vertex' to 'stack'.

DFS(graph, vertex, visited):
    Mark 'vertex' as visited and print/record 'vertex' as part of the current SCC.
    For each neighbor 'n' of 'vertex' in 'graph':
        if 'n' is not visited:
            DFS(graph, n, visited)
```



Show the ordering of vertices produced by the topological sort algorithm when it is run on the DAG above. Assume that DFS considers vertices in alphabetical order, and that all adjacency lists are also given in alphabetical order.

Show how Kosaraju's algorithm works on the following graph:




q
r
s
t
u
v
w
x
y
z

GREEDY ALGORITHMS

- Greedy Algorithms make optimal local decisions, in the hope that it will lead to a global optimal solution
- Used when the problem can be divided in stages, where a decision has to be made at each stage
- They never undo their decisions
- They are not always correct

DIJKSTRA's ALGORITHM : MINIMUM WEIGHTED PATH


```
Dijkstra(Graph, source):
    dist[source] ← 0                          // Initialization
    For each vertex v in Graph:
        If v ≠ source, then dist[v] ← infinity // Unknown distance from source to v
        prev[v] ← undefined                    // Predecessor of v

    Q ← the set of all nodes in Graph         // All nodes in the graph are unoptimized - thus are in Q
    While Q is not empty:                     // The main loop
        u ← vertex in Q with min dist[u]      // Node with the smallest distance
        remove u from Q

        For each neighbor v of u:             // where v has not yet been removed from Q.
            alt ← dist[u] + length(u, v)
            If alt < dist[v]:                 // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

MST : Kruskal's Algorithm

```
Kruskal(Graph):
    A = ∅                                   // A will contain the resulting MST

    For each vertex v in Graph.V:
        MAKE-SET(v)

    sort the edges of Graph.E into non-decreasing order by weight w

    For each edge (u, v) in Graph.E, taken in non-decreasing order by weight:
        If FIND-SET(u) ≠ FIND-SET(v):
            A = A ∪ {(u, v)}
            UNION(u, v)

    return A
```

MST : PRIM's ALGORITHM

```
Prim(Graph, source):
    Initialize a priority queue Q
    For each vertex v in Graph.V:
        If v is source
            key[v] = 0            // Make the key of the source vertex as 0 to get picked first
        Else
            key[v] = ∞            // Initialize all other keys as infinite
        parent[v] = NIL           // No parent of any vertex yet
        Q.insert(v, key[v])       // Insert vertex v into the priority queue Q with its key

    While Q is not empty:
        u = Q.extractMin()        // Remove and return the vertex with the smallest key
        For each vertex v adjacent to u:
            If v is in Q and weight(u, v) < key[v]:
                parent[v] = u     // Update parent to the current vertex
                key[v] = weight(u, v)
                Q.decreaseKey(v, key[v])   // Update the new key of v in the priority queue

    // The MST is represented by the parent array
```