BM40A1500 DATA STRUCTURES AND ALGORITHMS

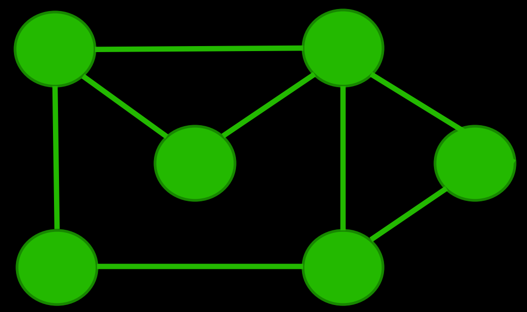
GRAPHS 1

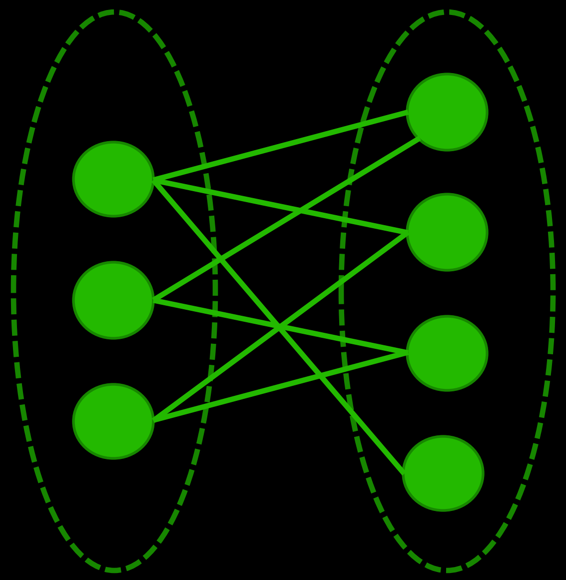
2024

WHY GRAPHS ARE IMPORTANT?

- Many problems can be formulated as graphs:
  - Finding shortest routes through road network
  - Computer network traffic routing
  - Electrical grids optimization
  - Matching and allocation (e.g., job tasks, restaurant tables)
    - Bipartite graph
  - Artificial intelligence in computer games

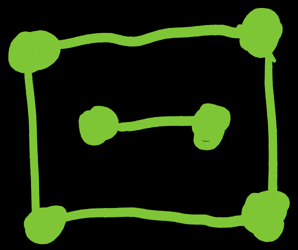
- Graph theory has been studied for decades resulting in a large number of algorithms.
  - These can be used for a large pool of problems.

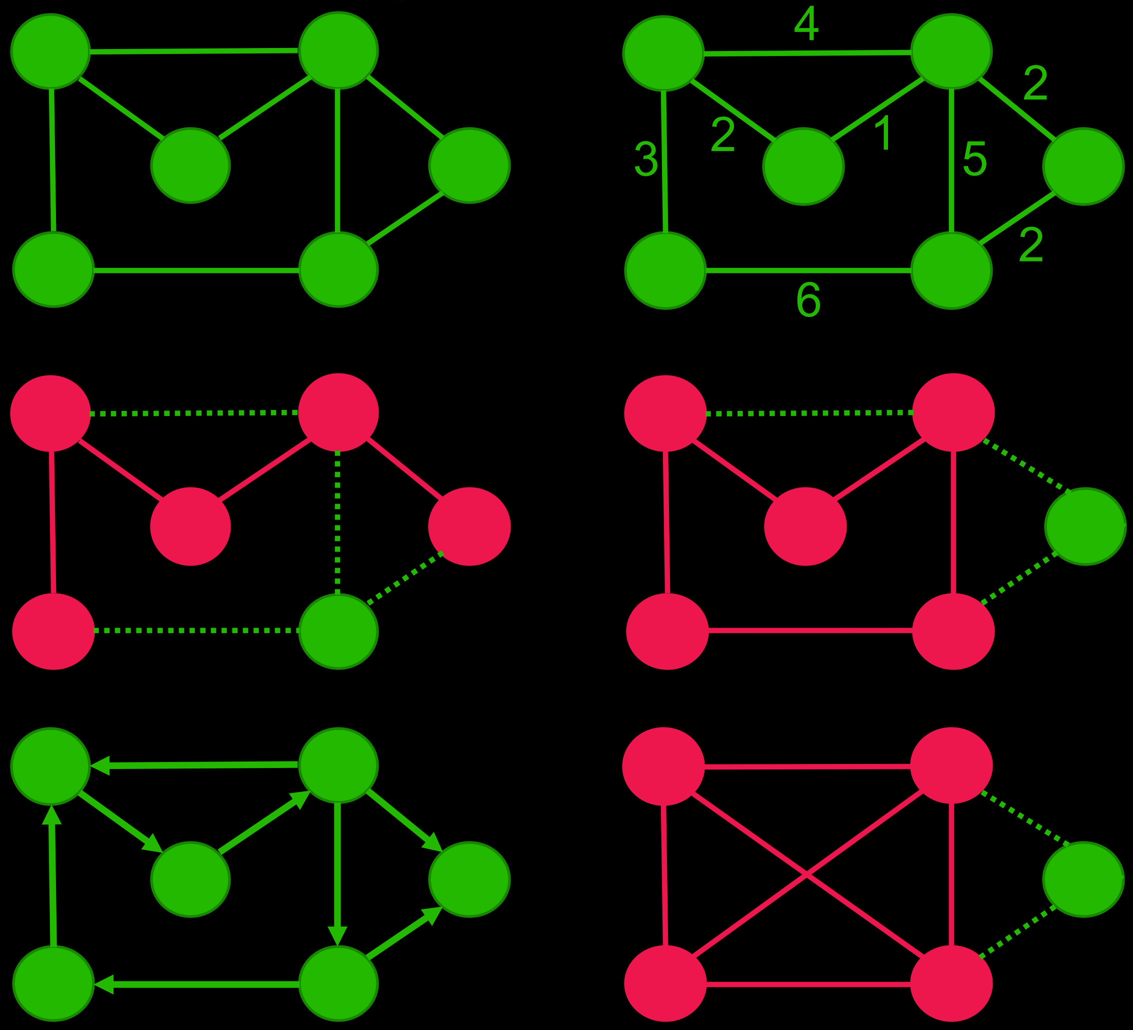




TERMINOLOGY

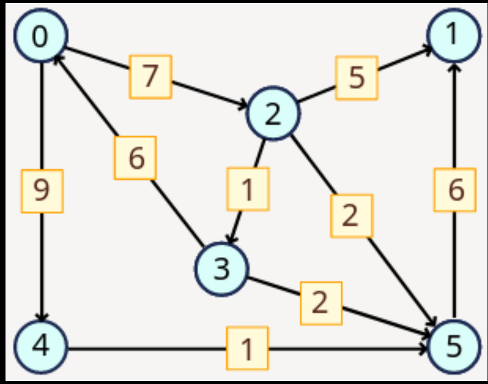
- Vertices (or nodes) and edges
- Weighted graph
  - Each edge has a weight or distance
- Path and cycle
- Directed and undirected graph
- Acyclic graph and directed acyclic graph
- Complete graph
  - A graph containing all possible edges
- Clique
  - Subgraph that is complete
- Connected component





IMPLEMENTING GRAPH

- Adjancency matrix
  - A 2-dimensional array where each row and each column corresponds to a vertex in the graph.
  - A given row and column in the matrix corresponds to an edge from the vertex corresponding to the row to the vertex corresponding to the column.
- Adjancency list
  - An (array-based) list to represent the vertices of the graph.
  - Each vertex is in turn represented by a list of the vertices that are neighbors.

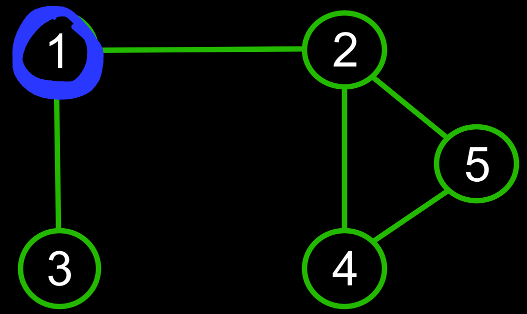


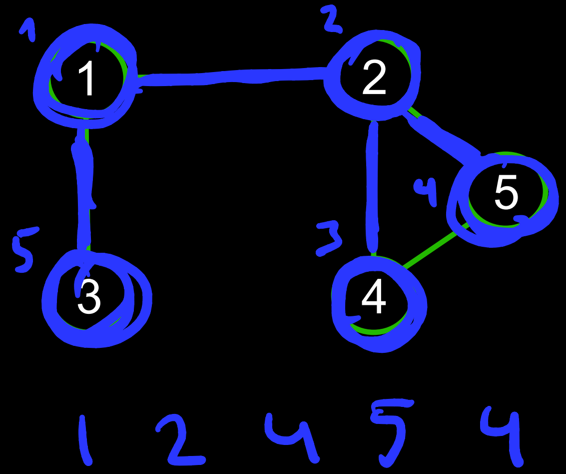

```
adj_matrix = [
#    0  1  2  3  4  5
    [0, 0, 7, 0, 9, 0],   # 0
    [0, 0, 0, 0, 0, 0],   # 1
    [0, 5, 0, 1, 0, 2],   # 2
    [6, 0, 0, 0, 0, 2],   # 3
    [0, 0, 0, 0, 0, 1],   # 4
    [0, 6, 0, 0, 0, 0]    # 5
]

# each pair = (neighbour, weight)
adj_list = [
    [(2, 7), (4, 9)],           # 0
    [],                         # 1
    [(1, 5), (3, 1), (5, 2)],   # 2
    [(0, 6), (5, 2)],           # 3
    [(5, 1)],                   # 4
    [(1, 6)]                    # 5
]
```

GRAPH TRAVERSALS

- Depth-First Search
  - Follow one branch through the graph to its conclusion, then back up and follow another branch, and so on.
  - Flag vertices as visited to avoid visiting the same vertex twice (graph might contain loops).
  - Can be implemented recursively.
  - Alternative approach is to use stack

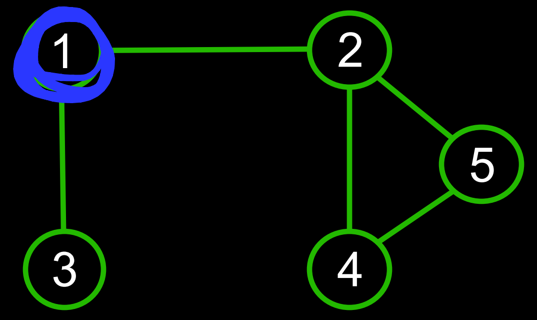
```
procedure DFS(start)
    S.push(start)
    while S is not empty
        V = S.pop
        if not visited[V]
            print(V)
            visited[V] = True
        for all neighbours of V
            if not visited(neighbuor)
                S.push(neighbour)
```



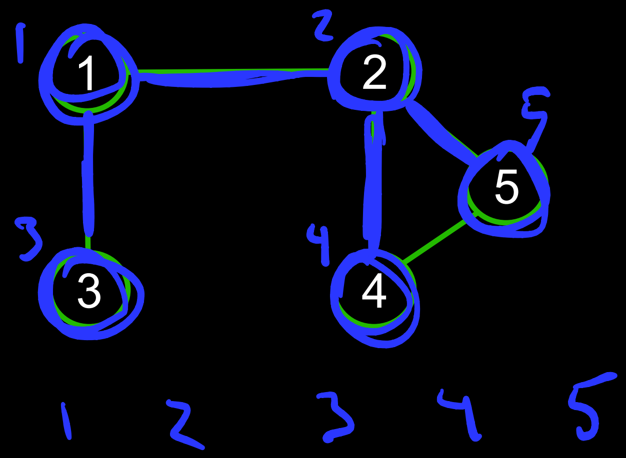


GRAPH TRAVERSALS

- Breadth-First Search
  - Examine all vertices connected to the start vertex before visiting vertices further away.
  - Flag vertices as visited to avoid visiting the same vertex twice.
  - An alternative approach is to use queue.

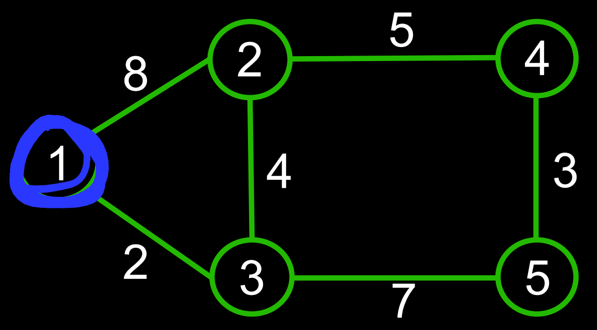


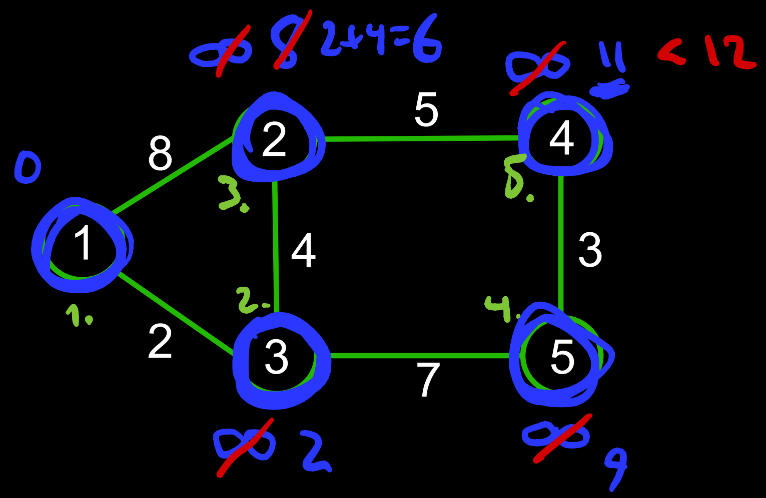

```
procedure BFS(start)
    Q.enqueue(start)
    while Q is not empty
        V = Q.dequeue
        if not visited[V]
            print(V)
            visited[V] = True
        for all neighbours of V
            if not visited(neighbour)
                Q.enqueue(neighbour)
```



SHORTEST-PATHS PROBLEM: DIJKSTRA ALGORITHM

- Single-source shortest paths
  - The shortest distances from the starting vertex to all other nodes.
- Dijkstra algorithm
  1. Process the vertices one by one.
  2. For each vertex, go trough its neighbors and all update the distances to the starting vertex.
     - Is the distance via the current node shorter the shortest distance found earlier?
  3. Mark the current vertex as visited.
     - Once a vertex is marked as visited, its distance is no longer updated.
  4. Move to the unvisited vertex with the shortest distance from the starting vertex.
  5. Repeat the process until all vertices have been visited.






LUT
University