Module 2: Graphs

Module Overview
Graphs are a powerful means to represent, model, and analyze any engineering system that can be represented as a set of vertices (or nodes) connected with edges. Common problems formulated and solved using graphs including shortest path problems, network flow problems, matching problems, and graph coloring problems, among others. This module provides you with foundational knowledge of graphs and graph algorithms, which is necessary for solving a variety of engineering problems using graph data structures, as discussed in this module and subsequent modules of this course. Specifically, in this module, you will learn the fundamentals of graphs, their applications, basic definitions and principles, and representations using adjacency matrices and lists. Further, you will learn two basic algorithms, Breadth-First Search (BFS) and Depth-First Search (DFS), for studying graph connectivity, traversal, and bipartiteness, as well as an algorithm for topological ordering in Directed Acyclic Graphs (DAGs). Several illustrative examples are provided along with Python codes for implementation of the algorithms.

Learning Objectives
• Explain the definitions, principles, and engineering applications of graph algorithms. • Compute and analyze graph connectivity algorithms: Breadth First Search (BFS) and Depth First Search (DFS). • Implement the BFS and DFS algorithms using stack and queue data structures. • Implement the BFS algorithm for testing the bipartiteness of graphs. • Compute and analyze topological ordering in directed acyclic graphs.

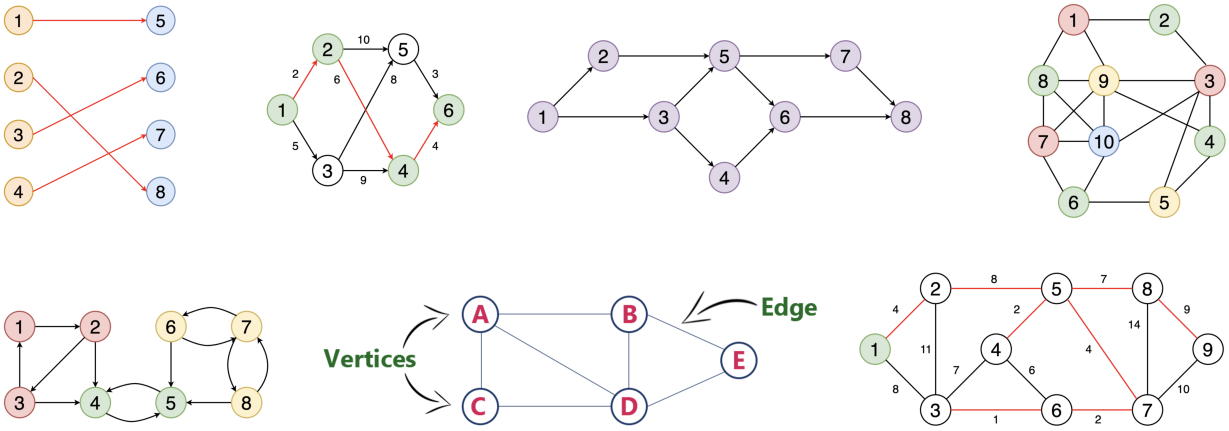
Course Outcomes
• Reading(s) • Lesson(s) • Discussion • Assignment • [Add_activity]

Reading and Resources	
Optional Resources	**Description**
Jon Kleinberg and Eva Tardos. Algorithm Design. USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN: 0321295358.	Chapter 3
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press, 2009. ISBN: 0262033844.	Chapter 22

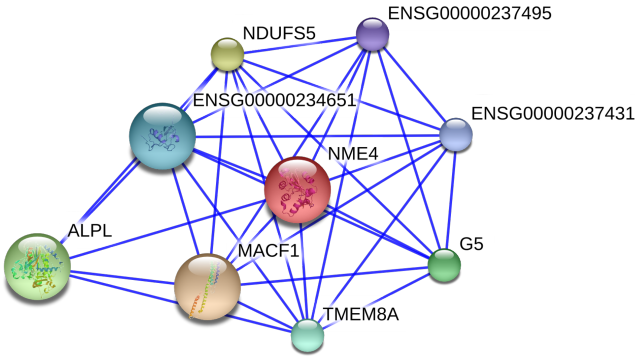
Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. Data Structures and Algorithms in Python, First Edition. Wiley Publishing, 2013. ISBN: 1118290275.	Chapter 14

Lesson 2.1: Fundamentals Of Graphs

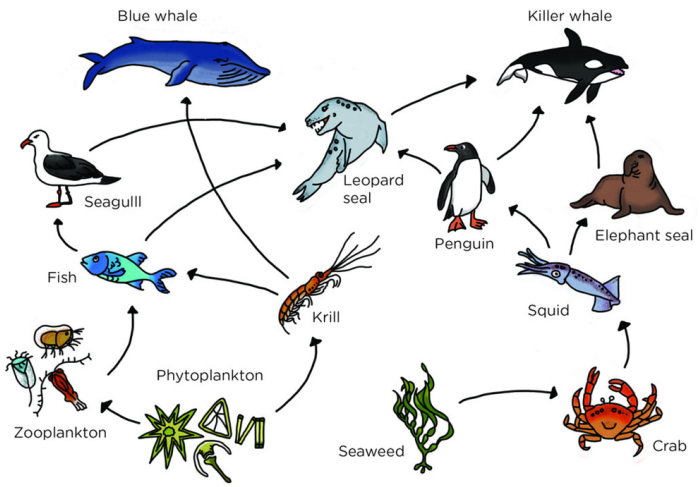
Graphs can be used to model a variety of natural and artificial objects—from characterizing atomic structures or metabolic and protein interaction networks to finding shortest path in a road, representing communication networks, designing electric circuits, managing transportation or supply networks, or forming relationships between people. A graph is simply a collection of objects or *vertices* linked together using *edges*. The abundance and diversity of these linked objects in the world is a motivation to study important algorithmic problems in graphs. Graphs can be used to model, solve, and analyze any engineering problems that can be represented as a set of vertices connected via a set of edges.



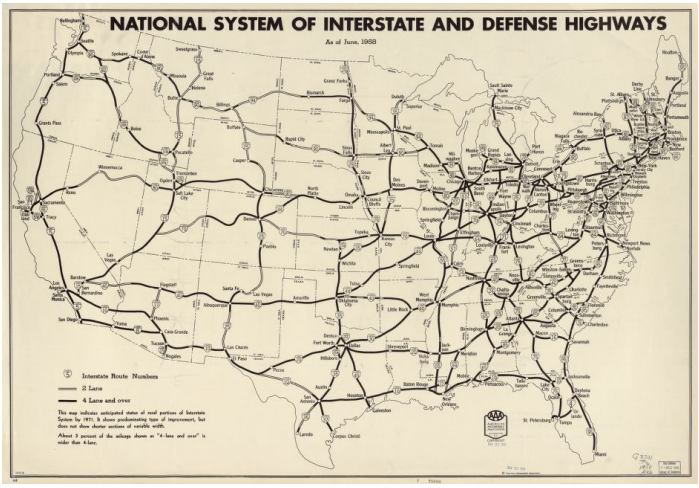
Protein Interaction Network. *Vertices*: Proteins. *Edges*: Protein modification, metabolism, binding, or transport.



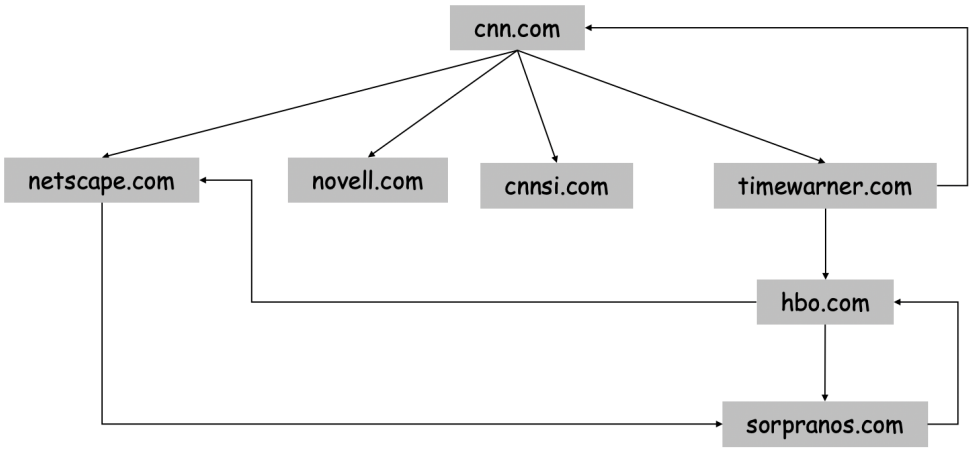
Food Web. *Vertices*: Species. *Edges*: Predator-prey relationship.



U.S. Highway Network. *Vertices*: Hubs/intersections. *Edges*: Highways.

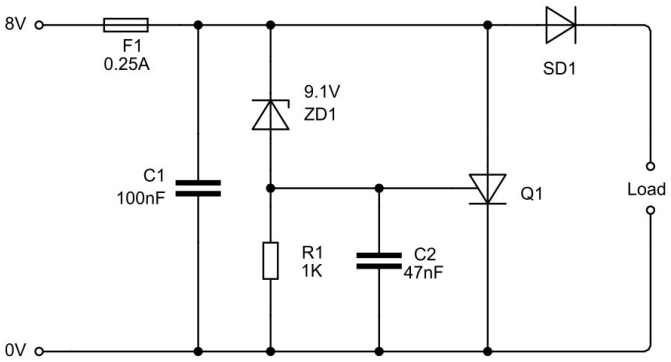


World Wide Web. *Vertices*: Web pages. *Edges*: Hyperlinks between web pages.

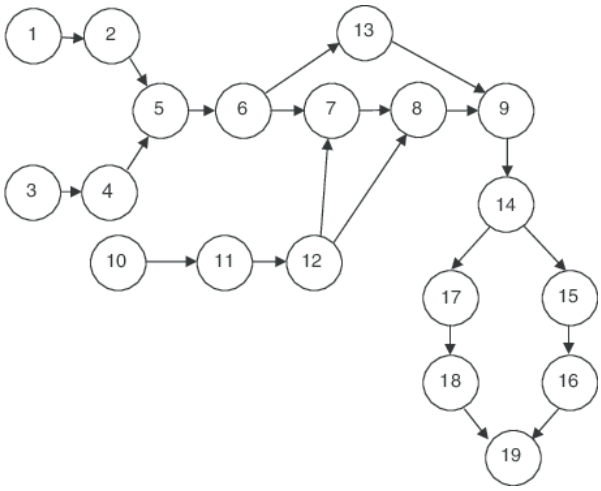


Social Network. *Vertices*: People. *Edges*: Connections and relationships.

Electric Circuits. *Vertices*: Gates, capacitors, resistors, etc. *Edges*: Wires.



Scheduling. *Vertices*: Task. *Edges*: Precedence constraints.



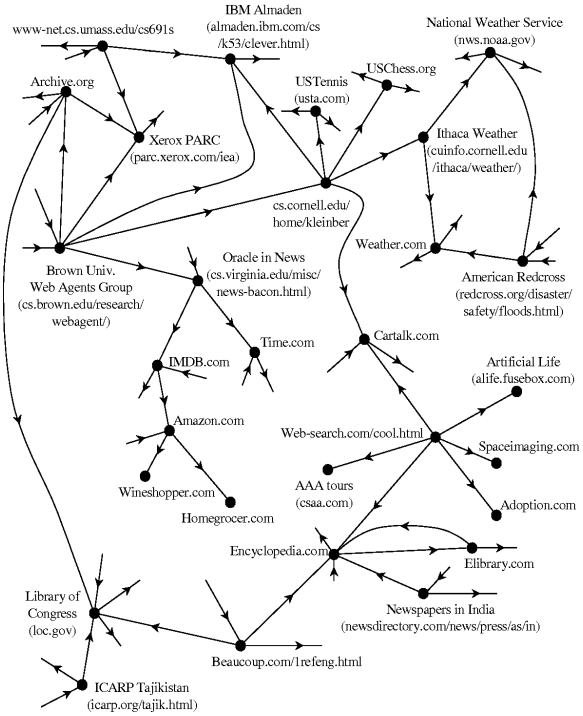
2.1.1 Basic Graph Definitions

Graphs are simply a way of encoding pairwise relationships among a set of objects, and are used to solve a plethora of practical problems when the complete problem space can be represented as a network. A graph G consists of a set V of vertices and a set E of edges.

Edge. Edges establish relationships between the vertices in a graph. Each edge links two vertices in a **directed** or **undirected** fashion. An edge is thus represented as $e \in E$, as a two-element subset of $V : e = u, v$ for some $u, v \in V$, where u and v are called the **ends** of e. The total number of edges $|E|$ is typically referred to as the **size** of the graph.

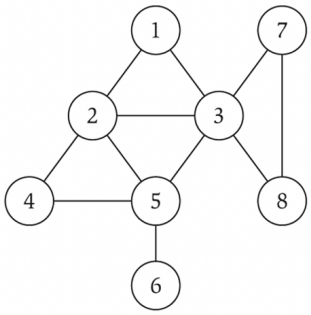
Vertex. Vertices are the most important components in the graph where data is stored and the relationships between different vertices are expressed using edges. A vertex in a graph may also be referred to as a node. The total number of vertices $|V|$ is typically referred to as the **order** of the graph.

Directed Graph. A directed graph G consists of a set of vertices V and a set of directed edges E. Each $e \in E$ is an ordered pair (u, v), where u is the tail and v the head of the edge, and their roles are not interchangeable. That is, edge e leaves vertex u and enters vertex v [2]. The World Wide Web is a great example of a directed graph, where there is an edge from webpage A to webpage B, if there is a link of webpage B on webpage A.



Undirected Graph. An undirected graph G consists of a set of vertices V and a set of edges E. Each $e \in E$ is an ordered pair (u, v), where the roles of u and v are interchangeable. The term graph typically means an undirected graph unless otherwise specified [2]. Facebook is an example of an undirected graph, where there is an edge between user A and user B, if A and B are friends on Facebook.

Example 1: An undirected graph $G = (V, E)$ is depicted below [2], where V is the sets of vertices, E is the set of edges, $n = |V|$ is the graph order and $m = |E|$ is the graph size.



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }

n = 8

m = 11

Add Interactive Media Here

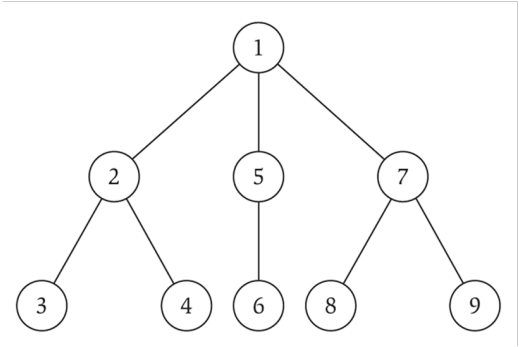
Sample interactive media for graph basics, definition, and representation: https://d3gt.com/index.html

Path. A path in an undirected graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, ..., v_k$ with the property that each consecutive pair v_i, v_{i+1} is linked by an edge. In Example 1, vertices 1, 2, 3, 5, 6 form a path. A path $v_1, v_2, ..., v_k$ may be **simple** where all vertices are distinct, or **cyclic** where $v_1 = v_k$, $k > 2$, and the first $k - 1$ vertices are all distinct. Further, the **shortest path** between vertices u and v is the minimum number of edges in the u-v path [2]. In Example 1, the shortest path between vertices 4 and 7 is 3.

Connectivity. A graph G is connected if for each pair of vertices $u, v \in G$, there is a path from u to v. A graph is **strongly connected** if for each pair of vertices $u, v \in G$, there is a path from u to v and a path from v to u. The graph depicted in Example 1 is strongly connected [2]. An interesting property of any n-vertex undirected graph G is that any two of the following statements implies the third: (1) G is connected; (2) G does not contains a cycle; (3) G has $n - 1$ edges.

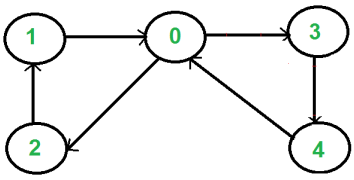
Tree. An undirected graph is a tree if it is connected and does not contain a cycle. A tree T has a **root** vertex r from which all edges of T are oriented away. For each pair of vertices u, v other than r, u is referred to as the **parent** of v if u directly precedes v on its path from r. In that case, v is referred to as the **child** of u. In more general terms, vertex w is considered a **descendant** of vertex v or v is considered an **ancestor** of w, if vertex v is on the path from the root to w. A vertex with no descendant is typically referred to as a **leaf**. [2]. An interesting property of trees is that any n-vertex tree has exactly $n - 1$ edges.

Example 2: An undirected graph $G = (V, E)$ is depicted below [2], vertex 1 is the root and, for example, vertex 7 is the parent of vertices 8 and 9, and vertices 3, 4, 6, 8, and 9 are all the leaves of the tree. Vertices 3, 4, 6, 8, and 9 are descendants of vertex 1, and vertex 1 is considered their ancestor.



CYK 1. What type of graph is displayed in the figure below?

a Undirected graph
b Directed graph
c Weighted graph
d Tree graph



Correct answer: b

CYK 2. How many edges does a complete graph with n vertices contain?

a $n(n-1)/2$
b n^2
c $n(n+1)/2$
d None of the above

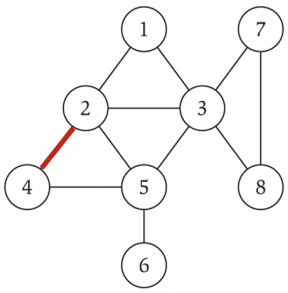
Correct answer: a

2.1.2 Graph Representation

Graph representation is a technique to store the graph information (i.e., vertices and edges) in a format readable by a computer, using standard data structures such as stacks or queues. The most common ways to represent a graph are adjacency matrices and adjacency lists.

Adjacency Matrix. An adjacency matrix is a 2D array of size $|V|x|V|$, where V is the set of vertices in a graph. Let the 2D array be $adj[][]$. If $adj[i][j] = 1$, there is an edge from vertex i to vertex j; otherwise, $adj[i][j] = 1$. The adjacency matrix for an undirected graph is always symmetric. Adjacency matrices are also used to represent weighted graphs. In that case, $adj[i][j] = w$ implies that there is an edge from vertex i to vertex j with weight w. An adjacency matrix provides two representations of each edge, and the required space is proportional to n^2, where $n = |V|$.

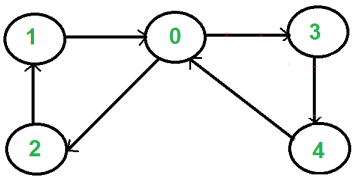
Example 3: The adjacency matrix of an undirected graph is depicted below [2].



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

CYK 3. What is the dimension of the adjacency matrix representation of the graph depicted below? How many of the elements in the matrix are non-zero?

a 4×5, 6
b 5×5, 6
c 5×5, 12
d 4×4, 6



Correct answer: b

CYK 4. The number of edges for the following adjacency matrix of an undirected graph G is:

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

Adjacency Matrix

a 6
b 12
c 10
d 5

Correct answer: a

CYK 5. For building an directed graph from an adjacency matrix, we just need to iterate through either the upper triangular matrix or the lower triangular matrix.

a True
b False

Correct answer: b

My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a simple representation of graph using an adjacency matrix.

```
# Graph representation using adjacency matrix
class Graph:
    def __init__(self, numvertex):
        # Create an empty n * n size matrix filled with zeros, where n is the
        ↪  number of vertices
        self.adjMatrix = [[0] * numvertex for x in range(numvertex)]
        self.numvertex = numvertex
        # Create a dictionary to match the vertices' names to a number e.g.,
        ↪  0 - 'a'
        self.vertices = {}
        self.verticeslist = [0] * numvertex

    def set_vertex(self, vtx, id):
        if 0 <= vtx <= self.numvertex:
            self.vertices[id] = vtx
            self.verticeslist[vtx] = id

    def set_edge(self, frm, to, cost=0):
        frm = self.vertices[frm]
        to = self.vertices[to]
        self.adjMatrix[frm][to] = 1 if cost == 0 else cost
        # for directed graph do not add this
        self.adjMatrix[to][frm] = 1 if cost == 0 else cost

    def get_vertex(self):
```

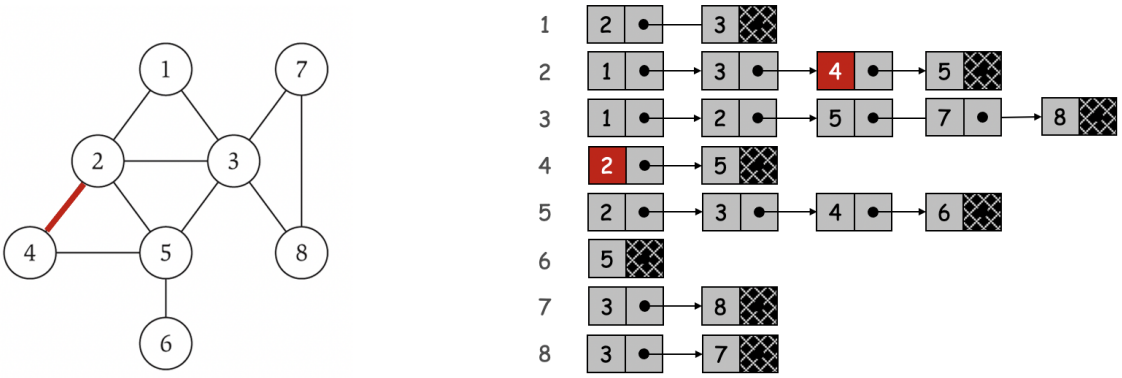
```
        return self.verticeslist

    def get_edges(self):
        edges = []
        for i in range(self.numvertex):
            for j in range(self.numvertex):
                if (self.adjMatrix[i][j] != 0):
                    edges.append((self.verticeslist[i],
                    ↪  self.verticeslist[j]))
        return edges

    def get_matrix(self):
        return self.adjMatrix
```

Use the above Python class along with the script below to represent the graph from Example 3 using an adjacency matrix.

```
# Driver code
G = Graph(8)
G.set_vertex(0, '1')
G.set_vertex(1, '2')
G.set_vertex(2, '3')
G.set_vertex(3, '4')
G.set_vertex(4, '5')
G.set_vertex(5, '6')
G.set_vertex(6, '7')
G.set_vertex(7, '8')
G.set_edge('1', '2')
G.set_edge('2', '3')
G.set_edge('1', '3')
G.set_edge('2', '4')
G.set_edge('2', '5')
G.set_edge('4', '5')
G.set_edge('5', '6')
G.set_edge('3', '5')
G.set_edge('3', '6')
G.set_edge('7', '8')

print("Vertices of the Graph")
print(G.get_vertex())
print("Edges of the Graph")
print(G.get_edges())
print("Adjacency Matrix of the Graph")
print(G.get_matrix())
```

Adjacency List. An alternative way to represent a graph is an array of lists. The size of the array is equal to the number of vertices. Let $array[]$ denote the array. An entry $array[i]$ represents the list of vertices adjacent to the ith vertex of the graph. This representation can also be used to represent a weighted graph, where the weights of the edges can be represented as lists of pairs. An adjacency matrix provides two representations of each edge, and the required space is proportional to $m + n$, where $n = |V|$ and $m = |E|$.

Example 3 (revisited): The adjacency list of the undirected graph in Example 3 is depicted below [2].



Adjacency Matrix or Adjacency List? That's the Question. Consider a graph $G = (V, E)$ and let $n = |V|$ and $m = |E|$ denote the number of edges and the number of vertices, respectively. Representing graph G using an $n \times n$ adjacency matrix A, $A[u, v]$ would be 1 if the graph contains edge (u, v), and 0 otherwise. This method of representation takes $O(n^2)$ space for a graph algorithm to examine all edges incident to a given vertex, even if the graph is sparse. Now if an adjacency list is chosen to represent graph G, we have an array Adj, where $Adj[v]$ is a record containing a list of all vertices adjacent to vertex v. For an undirected graph $G = (V, E)$, each edge $(v, w) \in E$ occurs on two adjacency lists—vertex w appears on the list for vertex v, and vertex v appears on the list for vertex w. Thus, checking if (u, v) is an edge takes $O(deg(u))$ time where $deg(u)$ is the number of neighbors of u, and identifying all edges takes $O(m + n)$ space.

Example 3 (revisited): Representing the undirected graph in Example 3 using an adjacency matrix consumes $O(64)$ space while using an adjacency only takes $O(19)$ space.

CYK 6. Which of the following statements are true?

a Adjacency list is better than adjacency matrix for representation of sparse graphs
b Finding whether there is an edge between any two nodes in a graph is faster using adjacency list than adjacency matrix
c Adding a vertex in adjacency list representation is faster than in adjacency matrix representation

Correct answers: a and c

My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a simple representation of graph using an adjacency list.

```
# Graph representation using adjacency list
from collections import defaultdict
class Graph:
    def __init__(self):
        # default dictionary to store graph
        self.graph = defaultdict(list)
        self.vertices = set()

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

        # uncomment the next line for undirected graph
        self.graph[v].append(u)

        # Creating a set of all the vertices for future use
        self.vertices.add(u)
        self.vertices.add(v)

    def printGraph(self):
        for i in self.vertices:
            print("Adjacency list of vertex {}\n head".format(i), end="")
            temp = self.graph[i]
            for v in temp:
                print(" -> {}".format(v), end="")
```

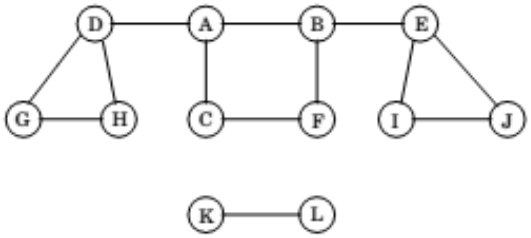
```
            print("\n")
```

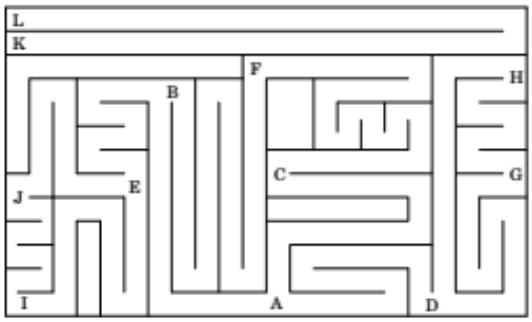
Use the above Python class along with the script below to represent the graph from Example 3 using an adjacency list.

```
# Driver code
g = Graph()
g.addEdge('1', '2')
g.addEdge('2', '4')
g.addEdge('1', '3')
g.addEdge('2', '5')
g.addEdge('2', '3')
g.addEdge('4', '5')
g.addEdge('5', '6')
g.addEdge('3', '7')
g.addEdge('3', '8')
g.addEdge('3', '5')
g.addEdge('7', '8')

g.printGraph()
```

Lesson 2.2: Graph Connectivity

Consider a graph $G = (V, E)$ and two particular vertices $u, v \in G$. Graph connectivity is concerned with finding an efficient algorithm that answers the question: **Is there a path from u to v in G?** For small graphs, this question can often be answered by visual inspection. For relatively larger graphs, however, this may not be a trivial question to answer through visual inspection. The graph connectivity problem is also referred to as the **maze-solving problem**: Considering G as a maze with a room corresponding to each vertex and a hallway corresponding to each edge that connects the rooms (i.e., the vertices) together, then the problem would be to start from room u and find the way to room v [2]. In this lesson, you will learn two fundamental algorithms for solving the graph connectivity problem: Breadth-First Search (BFS) and Depth-First Search (DFS). You will learn how to efficiently implement each algorithm using graph data structures. **Practical applications:** BFS and DFS have numerous practical applications such finding the shortest paths in a transportation network, finding the minimum spanning trees to optimize the design of a telecommunication network, building indices of web pages for search engine crawlers, optimizing machine schedules, or studying connections on social media.





2.2.1 Breadth First Search (BSF) Algorithm

Explore outward from vertex u in all possible directions, adding vertices one "layer" at a time:

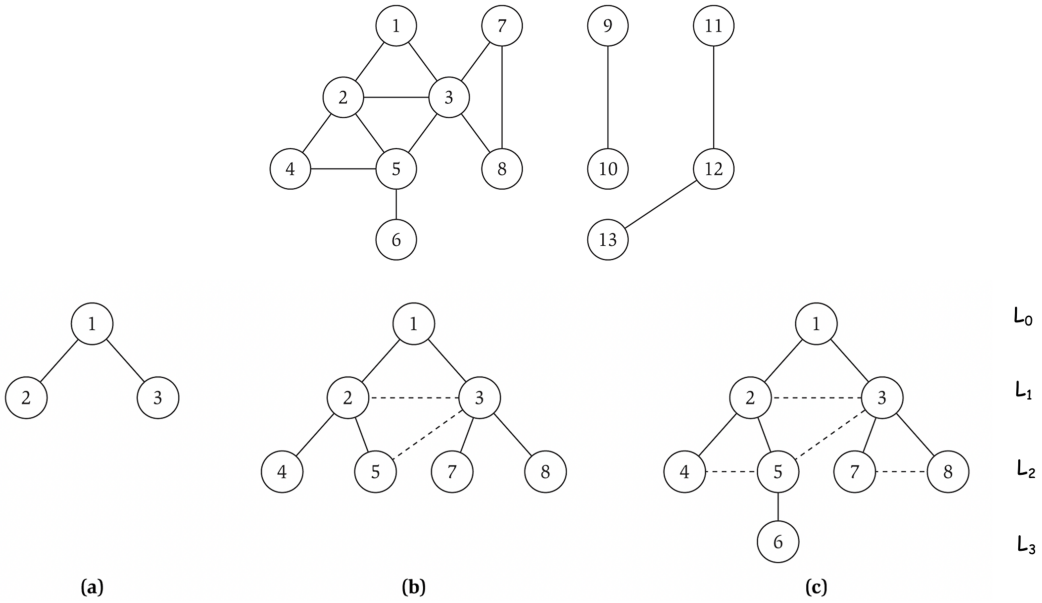
- Start with u as Layer 0: $L_0 = \{u\}$.
- Include all vertices that are immediately connected to it by an edges—this is the first layer: $L_1 = \{\text{all neighbors of } L_0\}$.
- include all additional vertices that are connected by an edge to any vertex in the first layer—this is the second layer: $L_2 = \{\text{all vertices that do not belong to } L_0 \text{ or } L_1, \text{ and that have an edge to a vertex in } L_1\}$.
- Continue until no new vertices are left.

All vertices in layer L_i are at the exact distance i from u, and there is a path from u to v if v appears in some layer. A vertex fails to appear in any of the layers if and only if there is no path to it. Thus, BFS not only determines the vertices that u can reach; it also computes the shortest paths to them [2].

Theorem 1. Let T be a BFS tree of $G = (V, E)$, (u, v) be an edge of G, and let u and v belong to layers L_i and L_j of T, respectively. The maximum difference between the levels of u and v in T is 1 [2].

Proof. Suppose $i < j - 1$, and consider the point in the BFS algorithm where the edges incident to vertex u are being examined. Since u belongs to layer L_i, the only vertices discovered from u belong to layers L_{i+1} and earlier. Hence, if v is a neighbor of u, it should have been discovered by this point at the latest, and therefore hence should belong to layer L_{i+1} or earlier [2]. □

Example 4: Consider vertex 1 as the starting point in the following graph. The first layer of the search would consist of vertices 2 and 3 (a). Further search from these vertices would yield vertices 4, 5, 7, and 8 as the second layer (b). The third layer would just consist of vertex 6 (c). The algorithm would now stop as there are no more vertices to be added. An important point to note is that vertices 9 through 13 are never reached in the search.



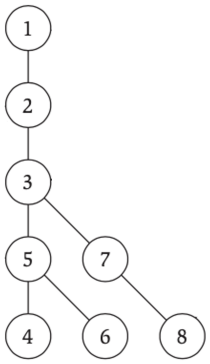
Add Interactive Media Here

Sample interactive media for the BFS algorithm:
https://www.cs.usfca.edu/galles/visualization/BFS.html

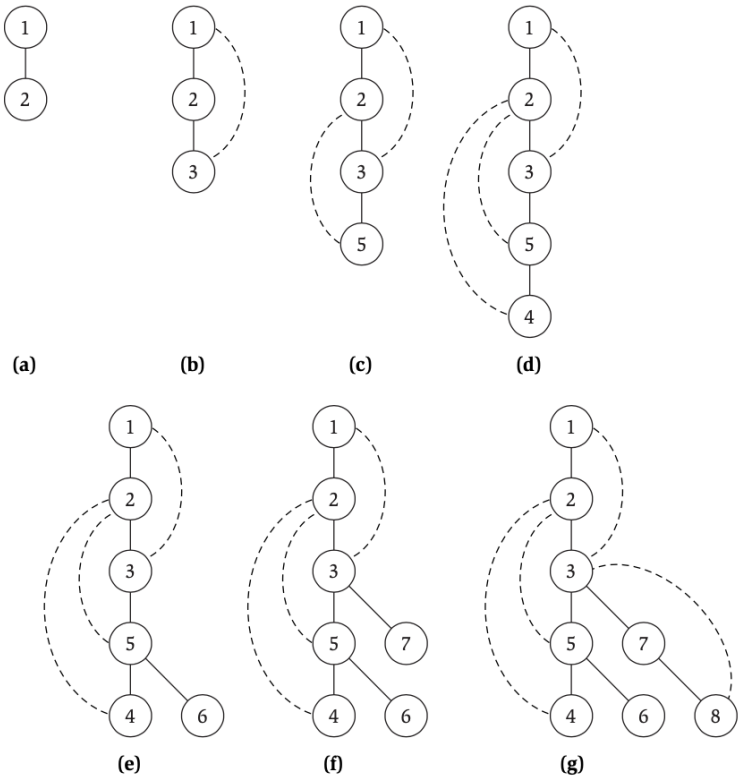
2.2.2 Depth First Search (DFS) Algorithm

Starting from vertex u, try the first edge leading out of it to vertex v, then follow the first edge leading out of v, and continue in this way until a "dead end" is reached—a vertex for which all neighbor vertices have already been explored. Then, backtrack until a vertex with an unexplored neighbor is reached, and resume from there. Although DFS ultimately visits exactly the same set of vertices as BFS, it typically does so in a different order—it probes its way down long paths, getting far away from u, before backing up to try nearer unexplored vertices. In the tree T generated using the DFS algorithm, whenever DFS(v) is invoked directly during the call to DFS(u), the edge (u, v) is added to T [2].

Example 5: The DFS algorithm is implemented on the following graph as demonstrated below.



a Start with vertex 1 as root of the tree and search for the first edge, which is vertex 2.
b Move from vertex 2 to find its first outward edge which is vertex 3.
c Move from vertex 3 to find its first outward edge which is vertex 5.
d Continuing the search would lead to a dead end at vertex 4.
e Backtrack to vertex 5 and find another edge which is vertex 6 and also another dead end.
f Backtrack to vertex 3 and find one more edge which results in finding new vertex 7.
g Move from vertex 7 to find its first outward edge which is vertex 8.



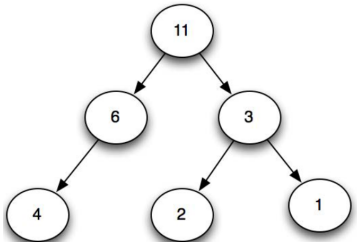
Theorem 2. Let T be a DFS tree of graph G, u and v be two vertices in T, and (u, v) be an edge of G that is not an edge of T. Then, one of u and v is descendant of the other.

Proof. Let (u, v) be an edge of G that is not an edge of T. Suppose, without loss of generality, that u is reached first by the DFS algorithm, and when the edge (u, v) is examined during the execution of DFS(u), it is not added to T because v is marked "explored". Since v was not marked "explored" when DFS(u) was first invoked, it is a vertex that was discovered between the invocation and end of the recursive call DFS(u). Thus, v is a descendant of u [2]. □

Add Interactive Media Here

Sample interactive media for the DFS algorithm:
https://www.cs.usfca.edu/galles/visualization/DFS.html

CYK 7. Which item correctly represents the path taken by the BFS algorithm from the root to the leaves of the tree below?



a 4, 2, 1, 6, 3, 11
b 4, 6, 11, 2, 3, 1
c 11, 6, 4, 6, 11, 3, 2, 3, 1
d 11, 6, 4, 3, 2, 1
e 11, 6, 3, 4, 2, 1

Correct answer: e

CYK 8. Which item correctly represents the path taken by the DFS algorithm from the root to the leaves of the tree above?

a 4, 2, 1, 6, 3, 11
b 4, 6, 11, 2, 3, 1
c 11, 6, 4, 6, 11, 3, 2, 3, 1
d 11, 6, 4, 3, 2, 1
e 11, 3, 4, 6, 2, 1

Correct answer: d

Comparison Between the BFS and DFS Algorithms

BFS	**DFS**
BFS starts traversal from the root vertex and visits vertices in a level-by-level manner (i.e., visiting the ones closest to the root first)	DFS starts the traversal from the root vertex and visits vertices by getting as far from the root vertex as possible (i.e., depth wise)
BFS uses queue data structure for exploring connectivity and shortest path	DFS uses stack data structure exploring connectivity and shortest path
BFS is more suitable for searching vertices which are closer to the root	DFS is more suitable when there are solutions away from root

BFS considers all neighbors first and therefore is not suitable for decision-making trees used in games or puzzles	DFS is more suitable for game or puzzle problems—players make a decision, then explore all paths stemming from that decision
The time complexity of BFS is $O(\|V\| + \|E\|)$ when adjacency list is used, and $O(\|V\|^2)$ when adjacency matrix is used, where V is the set of vertices and E is the set of edges	The time complexity of DFS is also $O(\|V\|+\|E\|)$ when adjacency list is used, and $O(\|V\|^2)$ when adjacency matrix is used, where V is the set of vertices and E is the set of edges

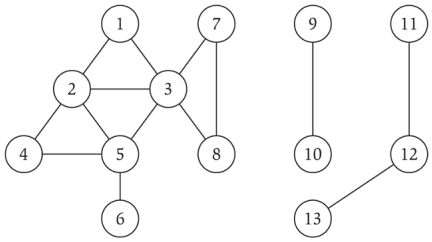
2.2.3 Connected Components

The vertices discovered by the BFS or DFS algorithm is precisely those reachable from the starting vertex u. This set of vertices R is called the connected components of graph G containing u. This set helps us solve the u-v connectivity problem by simply checking whether v belongs to R or not. Set R can be built by exploring G in any order starting from u. Initially, $R = \{r\}$ where and whenever a new edge (v, w) is discovered where $v \in R$ and $w \notin R$, w is added to R. This process is continued to until there are no more edges leading out of any vertices in R.

Algorithm 1: Connected Components Set

R will consist of vertices to which u has a path
Initially $R = \{u\}$;
while *there is an edge* (v, w) *where* $v \in R$ *and* $w \notin R$ **do**
  add w to R
end

Example 6: The connected component set of the following graph containing vertex 1 is built by traversing all the edges through either BFS or DFS.



Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

2.2.4 Connectivity In Directed Graphs

So far in this lesson, connectivity has been considered for undirected graphs. If all vertices of an undirected graph are visited, then it is considered connected. However, this approach does not work for a directed graphs, because in a directed graph an edge (u, v) has a direction—it goes from

u to v. BFS and DFS function almost the same in directed graphs as they do in undirected graphs. The difference, however, is that it is possible for a vertex u to have a path to a vertex v even though there is no path from v to u. Now suppose that for a given vertex u, we wanted the set of vertices with paths to u rather than the set of vertices to which u has paths. This could be done by defining a new directed graph G^{rev} obtained from the directed graph G by simply reversing the direction of every edge. The BFS or DFS algorithm could then be applied to G^{rev}: a vertex has a path from u in G^{rev} if and only if it has a path to u in G [2].

Strong Connectivity. A directed graph is strongly connected if for every two vertices u and v, there is a path from u to v and a path from v to u. This implies that every pair of vertices is mutually reachable [2].

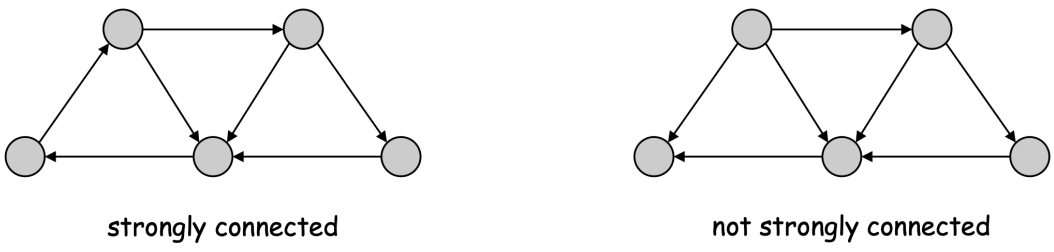
Theorem 3. If u and v are mutually reachable, and v and w are mutually reachable as well, then u and w are mutually reachable.

Proof. To construct a path from u to w, we must first go from u to v along the path guaranteed by the mutual reachability of u and v u and v, and then from v to w along the path guaranteed by the mutual reachability of v to w. To construct a path from w to v, we just reverse this reasoning: we first go from w to v along the path guaranteed by the mutual reachability of w and v, and then from v to u along the path guaranteed by the mutual reachability of v to u [2]. □

The following algorithm determines if a directed graph is strongly connected or not in liner time $O(|V| + |E|)$:

Algorithm 2: Strong Connectivity in Directed Graphs
Pick any vertex u;
Run BFS from u in G;
Run BFS from u in G^{rev};
Return true if and only if all vertices reached in both BFS executions

Example 7: Below are examples of strongly connected and not strongly connected directed graphs.



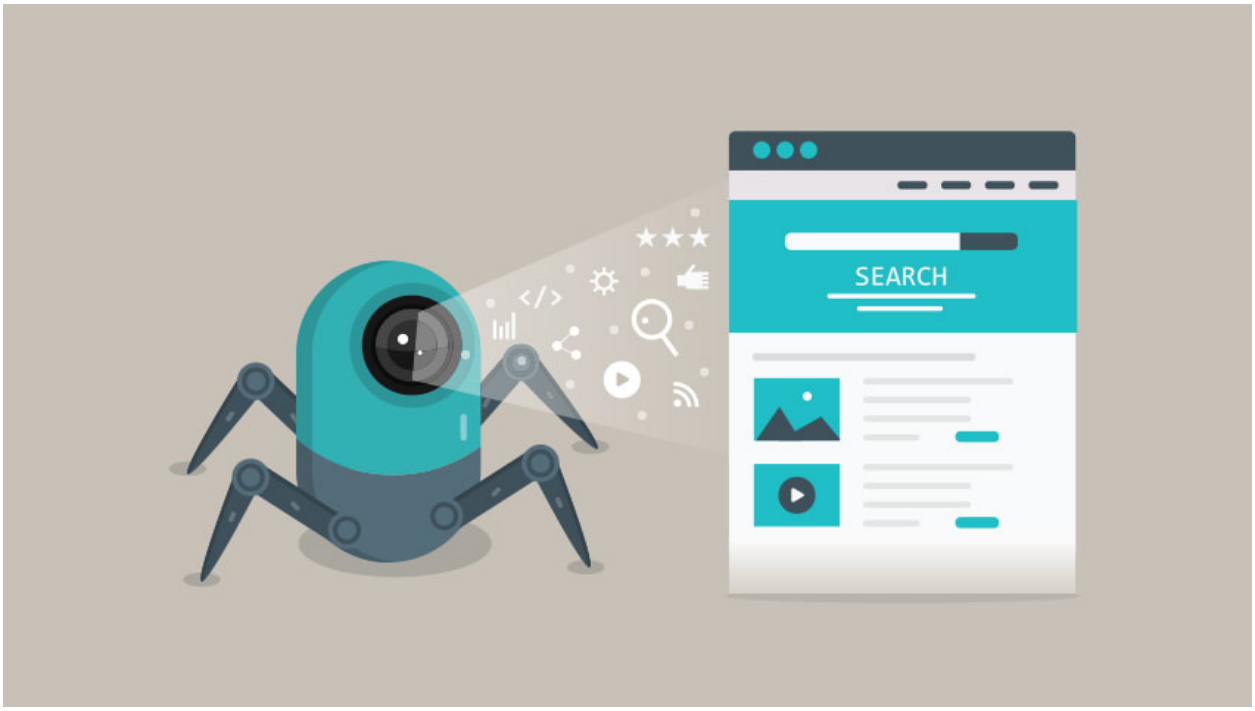
Strong Components in Directed Graphs. The strong components containing a vertex u in a directed graph are the set of all vertices v such that u and v are mutually reachable.

Theorem 4. For any two vertices u and v in a directed graph, their strong components are either identical or disjoint.

Proof. Consider two mutually reachable vertices u and v. For any vertex w, if u and w are mutually reachable, then by Theorem 2, v and w are mutually reachable as well. Similarly, if v and w are mutually reachable, then again by Theorem 2, u and w are mutually reachable [2]. □

Lesson 2.3: Graph Traversal

Graph traversal is the process of visiting, checking, or updating each vertex in graph G. The key consideration is the order in which the vertices are visited. The BFS and DFS algorithms discussed earlier can guide us through the traversal process. **Practical Applications:** Graph traversal using BFS and DFS has several important applications in practice. A good example is *web crawlers in search engines*, which start from an initial source webpage and progressively browse all the links from the visited pages to the next. Other common application examples include finding neighbor nodes in peer-to-peer networks, finding the k-closest contacts of a person on social media, GPS navigation, garbage collection systems, design of broadcasting networks, path finding, and finding the shortest path in a transportation network, factory, warehouse, or telecommunication network, among others.



2.3.1 BFS Implementation

BFS traverses graph G layer by layer until all vertices are visited. BFS runs in $O(m+n)$ time. We can implement the BFS algorithm using a list L maintained as a queue. The algorithm thus visits vertices in the order they are first discovered: Each time a vertex is discovered, it is added to the end of the queue. BFS always processes the edges out of the vertex that is first in the queue. The BFS pseudocode is presented below. Note that BFS sets the Discovered[v] array to true as soon as vertex v is first discovered.

Algorithm 3: Breadth First Search Algorithm

Set Discovered[s] = true and Discovered[v] = false for all v.
Initialize $L[0] = \{s\}$.
Set layer counter $i = 0$.
Set current BFS tree $T = \emptyset$.
while $L[i]$ *is not empty* **do**
  Initialize empty list $L[i+1]$
  for *each vertex* $u \in L[i]$ **do**
    Consider each edge (u, v) incident to u
    if *Discovered*[v] = *false* **then**
      Set Discovered[v] = true
      Add edge (u, v) to tree T
      Add vertex v to list $L[i+1]$
    end
  end
  Increment layer counter i by 1
end

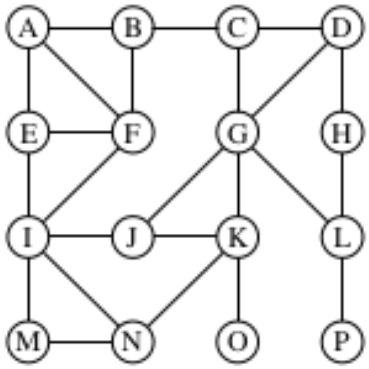
My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a Python script for BFS implementation.

```
# BFS implementation for graph traversal
from collections import defaultdict
class Graph:
    # Constructor
    def __init__(self):
        # Default dictionary to store graph
        self.graph = defaultdict(list)
        self.vertices = set()

    # Function for adding an edge to graph
```

```
    def addEdge(self, u, v):
        self.graph[u].append(v)
        # NOTE: uncomment the next line for undirected graph
        # self.graph[v].append(u)

        # Creating a set of all vertices for future use
        self.vertices.add(u)
        self.vertices.add(v)

    # Function for printing a BFS of graph
    def BFS(self, s):
        # Mark all vertices as not visited
        visited = {}
        for v in list(self.vertices):
            visited[v] = False
        # Create a queue for BFS
        queue = []
        # Mark the source vertex as visited and enqueue it
        queue.append(s)
        visited[s] = True
        while queue:
            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
            # Get all adjacent vertices of the dequeued vertex s. If an
            ↪  adjacent vertex has not been visited, mark it visited and
            ↪  enqueue it.
            for i in self.graph[s]:
                if not visited[i]:
                    queue.append(i)
                    visited[i] = True
                    print("from", s, "to", i)
```

Example 8: For the undirected graph depicted below [1], run the above Python script for BFS implementation along with the following script for graph traversal using the BSF algorithm.



```
# Driver code
g = Graph()
g.addEdge('A', 'B')
g.addEdge('A', 'E')
g.addEdge('A', 'F')
g.addEdge('B', 'C')
g.addEdge('B', 'F')
g.addEdge('C', 'D')
g.addEdge('C', 'G')
g.addEdge('D', 'G')
g.addEdge('D', 'H')
g.addEdge('E', 'F')
g.addEdge('E', 'I')
g.addEdge('G', 'J')
g.addEdge('G', 'K')
g.addEdge('G', 'L')
g.addEdge('H', 'L')
g.addEdge('I', 'J')
g.addEdge('I', 'M')
g.addEdge('I', 'N')
g.addEdge('J', 'K')
g.addEdge('K', 'N')
g.addEdge('K', 'O')
g.addEdge('L', 'P')
g.addEdge('M', 'N')

print("BFS traversal starting from vertex A")
g.BFS('A')
```

2.3.2 DFS Implementation

DFS traverses graph G from vertex u by trying the first edge leading out of it to vertex v, then following the first edge leading out of v, and continuing in this way until until a "dead end" is reached. DFS adds the identified to a stack S, so that these neighbors will be explored before we return to explore the other neighbors of u. Similar to BFS, DFS runs in $O(m+n)$ time. The DFS pseudocode is presented below. Note that DFS sets the Explored[v] array to true when all incident edges of vertex v are explored.

Algorithm 4: Depth First Search Algorithm

Initialize S to be a stack with one element s.
while S *is not empty* **do**
  Take a vertex u from S
  if *Explored*$[u] = false$ **then**
    Set Explored$[u] = true$
    for *each edge* (u, v) *incident to* u **do**
      Add v to the stack S
    end
  end
end

My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a Python script for DFS implementation.

```
# DFS implementation for graph traversal
from collections import defaultdict
class Graph:
    # Constructor
    def __init__(self):
        # default dictionary to store graph
        self.graph = defaultdict(list)
        self.vertices = set()

    # Function for adding an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)
        # NOTE: uncomment the next line for undirected graph
        # self.graph[v].append(u)

        # Creating a set of all the vertices for future use
```

```
        self.vertices.add(u)
        self.vertices.add(v)

    # Function for printing a DFS of graph
    def DFSUtil(self, v, visited):
        # Mark the current vertex as visited and print it
        visited[v] = True

        # Recur for all the vertices adjacent to this vertex
        for neighbour in self.graph[v]:
            if not visited[neighbour]:
                self.DFSUtil(neighbour, visited)
                print("from", v, "to", neighbour)

    def DFS(self, v):
        # Mark all the vertices as not visited
        visited = {}
        for i in list(self.vertices):
            visited[i] = False
        self.DFSUtil(v, visited)
```

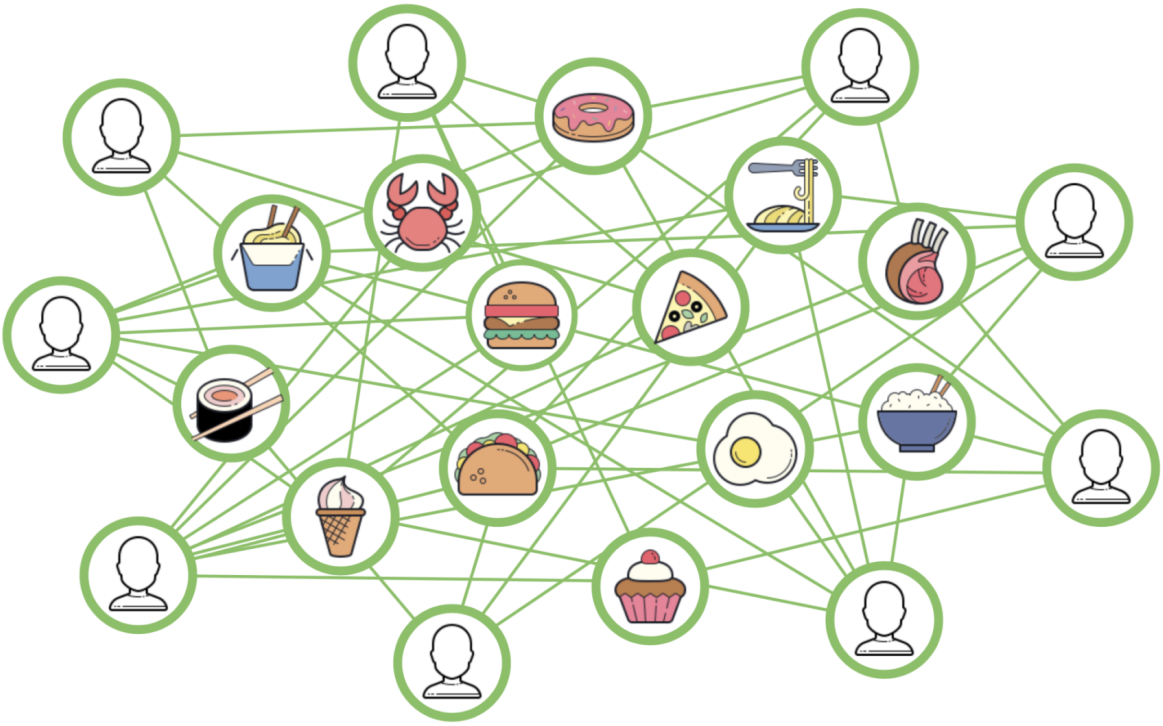
Example 8 (revisited): For the undirected graph in Example 8, run the above Python script for DFS implementation along with the following script for graph traversal using the DSF algorithm.

```
# Driver code
g = Graph()
g.addEdge('A', 'B')
g.addEdge('A', 'E')
g.addEdge('A', 'F')
g.addEdge('B', 'C')
g.addEdge('B', 'F')
g.addEdge('C', 'D')
g.addEdge('C', 'G')
g.addEdge('D', 'G')
g.addEdge('D', 'H')
g.addEdge('E', 'F')
g.addEdge('E', 'I')
g.addEdge('G', 'J')
g.addEdge('G', 'K')
g.addEdge('G', 'L')
g.addEdge('H', 'L')
g.addEdge('I', 'J')
g.addEdge('I', 'M')
g.addEdge('I', 'N')
g.addEdge('J', 'K')
```

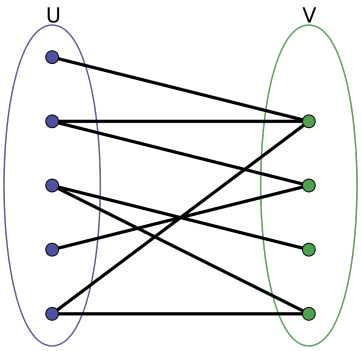
```
g.addEdge('K', 'N')
g.addEdge('K', 'O')
g.addEdge('L', 'P')
g.addEdge('M', 'N')

print("DFS traversal starting from vertex A")
g.DFS('A')
```

Lesson 2.4: Testing Graph Bipartiteness With The BFS Algorithm

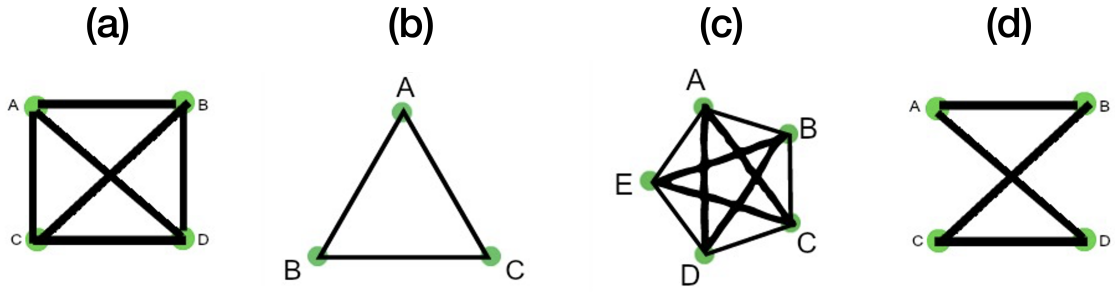
Bipartite graphs are a powerful tools for representing, modeling, and analyzing the relationships between two sets of entities; e.g., jobs and production machines, players and football clubs, students and schools, residents and hospitals, suppliers and customers, passenger and rides, travelers and hotels. A recent and popular example of a bipartite graph is Uber Eats—customers who would like to order food online can be considered as one set C and the restaurants can be considered as the other set R. If the order of customer $i \in C$ is matched to restaurant $j \in R$, that means there is an edge between vertices i and j. A bipartite graph like this could be used to match the best restaurant in R to a customer order in C.



Bipartite Graph. A bipartite graph, or a **bi-graph**, is a graph whose vertices can be divided into two disjoint and independent sets U and V, where each edge connects a pair of vertices in U and V. In a bi-graph, sets U and V are usually referred to as the parts of the graph. A bipartite graph does not contain any odd-length cycles [2]. Special graphs that are naturally bipartite include trees, cycle graphs with an even number of vertices, and planar graph whose faces all have even length. A **complete bipartite graph** on m and n vertices is usually denoted by $G = (U, V, E)$, where U and V are disjoint sets of size m and n, respectively, and E connects every vertex in U with all vertices in V.



CYK 9. Which of the following graphs is bipartite?



Correct answer: d

BFS Test for Bipartiteness. The BFS algorithm can be used to test the bipartiteness of a given graph G. First, we compute the connected components of the graph to ensure its connectivity. Next, we pick any vertex $s \in V$ and color it in red. It follows that all neighbors of vertex s must be colored in blue. It then follows that all neighbors of those vertices must be colored in red, their neighbors must be colored in blue, and so on. Once the entire graph is colored, all edges should be either colored red or blue; otherwise, the graph is considered not bipartite.

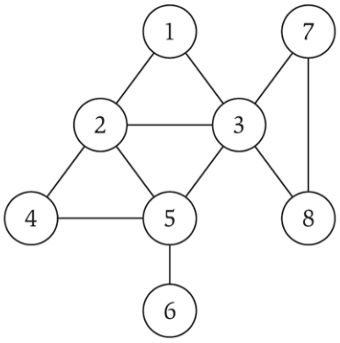
My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a Python script for testing graph bipartiteness using the BFS algorithm.

```
# Testing bipartiteness using BFS
from collections import defaultdict
class Graph():
    def __init__(self, V):
        self.V = V
        self.graph = [[0 for column in range(V)] for row in range(V)]

    # This function returns true if graph G[V][V] is Bipartite, else false
    def isBipartite(self, src):
        # Create a color array to store colors assigned to all veritces.
        ↪  Vertex number is used as index in this array.
        # The value '-1' of  colorArr[i] is used to indicate that no color is
        ↪  assigned to vertex 'i'.
        # The value 1 is used to indicate first color is assigned and value 0
        ↪  indicates second color is assigned.
        colorArr = [-1]*self.V
        # Assign first color to source
        colorArr[src] = 1
        queue = []
        queue.append(src)
        while queue:
            u = queue.pop()
            # Return false if there is a self-loop
            if self.graph[u][u] == 1:
                return False;
            for v in range(self.V):
                # An edge from u to v exists and destination v is not colored
                if self.graph[u][v] == 1 and colorArr[v] == -1:
                    # Assign alternate color to this adjacent v of u
```

```
                    colorArr[v] = 1 - colorArr[u]
                    queue.append(v)
                # An edge from u to v exists and destination v is colored
                ↪  with same color as u
                elif self.graph[u][v] == 1 and colorArr[v] == colorArr[u]:
                    return False
        return True
```

Example 9: Test the bipartiteness of the following undirected graph from Example 6 by running the above Python script along with the following script.



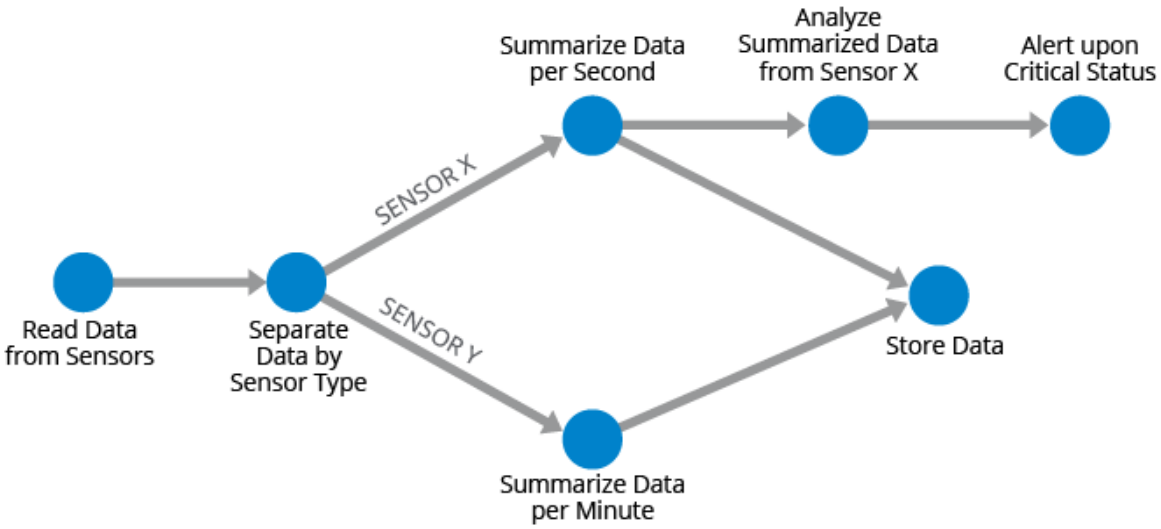

```
# Driver code
g = Graph(8)
g.graph = [[0, 1, 1, 0, 0, 0, 0, 0],
           [1, 0, 1, 1, 1, 0, 0, 0],
           [1, 1, 0, 0, 1, 0, 1, 1],
           [0, 1, 0, 0, 1, 0, 0, 0],
           [0, 1, 1, 1, 0, 1, 0, 0],
           [0, 0, 0, 0, 1, 0, 0, 0],
           [0, 0, 1, 0, 0, 0, 0, 1],
           [0, 0, 1, 0, 0, 0, 1, 0],
          ]

if g.isBipartite(0):
  print("The graph is bipartite")
else:
  print("The graph is not bipartite")
```

Theorem 5. Let G be a connected graph, and $L_1, L_2, ...$ be the layers produced by BFS starting from vertex s. Exactly one of the following conditions is true [2]: (1) There is no edge of G joining two vertices of the same layer. In this case G is a bipartite graph. (2) There is an edge of G joining two vertices of the same layer. In this case, G contains an odd-length cycle, and so it cannot be bipartite.

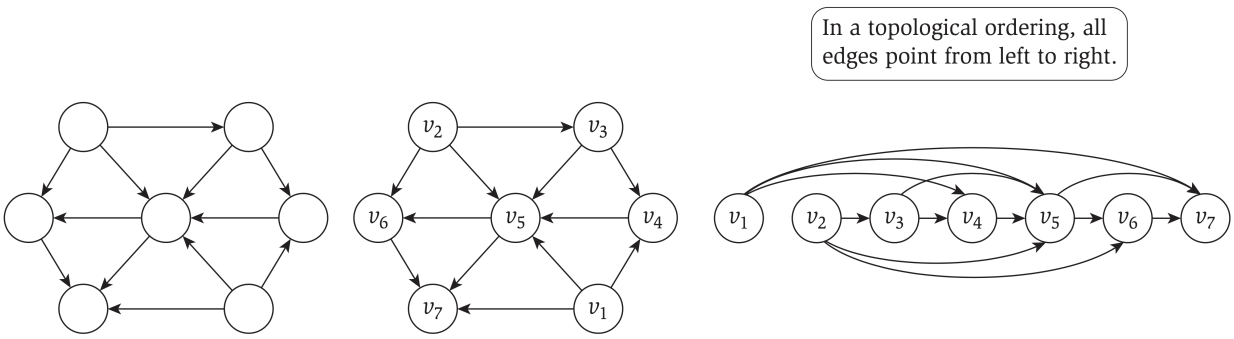
Proof. By Theorem 1, we know that every edge of G joins vertices either in the same layer or in adjacent layers. The assumption for case (1) is that the first of these two alternatives never happens. This means that every edge joins two vertices in adjacent layers. Thus, the coloring procedure gives vertices in adjacent layers and on two ends of every edge the opposite colors. This coloring establishes that G is bipartite [2]. □

Lesson 2.5: Topological Ordering In Directed Acyclic Graphs

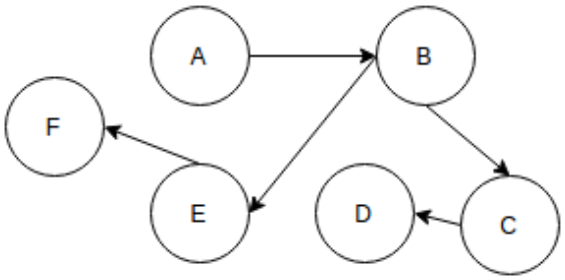
Topological ordering (aka, topological sorting) in a Directed Acyclic Graph (DAG) is a linear ordering of the vertices in which each vertex u comes before vertex v, if there us a directed edge (u, v). A **DAG** is a directed graph that flows in one direction such that no vertex can be a child of itself. DAGs are a very important type of data structure and can be used to encode **precedence or dependency diagrams**. Consider a set of tasks for processing sensor signals, where, where a given task cannot start until its predecessors are finished. Such an interdependent set of tasks can be represented using a directed graph G, where each vertex u represents a particular task and each directed edge (u, v) implies that u must be done before v. In order for the precedence relation to be meaningful, the resulting directed graph G must be a DAG; otherwise, task u may be both the predecessor and the successor of task v if G contains a cycle. Practical applications of topological ordering include scheduling jobs with dependencies, data serialization, and instruction scheduling, among others.



Topological Ordering. Given a DAG (e.g., a task precedence diagram), topological ordering seeks a valid order of of the vertices (e.g., the order to perform the tasks). Specifically, a topological ordering of G is an ordering of its vertices as $v_1, v_2, ..., v_n$ so that for every edge (v_i, v_j), we have $i < j$. That is, all edges must point forward in the topological order [2]. The key consideration in the topological ordering of a DAG is finding the starting vertex on the topological order.



CYK 10. Which of the following is not a topological ordering of the graph below?



a A B C D E F
b A B E C F D
c A B C D F E
d A B F E D C

Correct answer: c

Theorem 6. If a directed graph G has a topological order, then G is a DAG [2].

Proof. By contradiction: Suppose G has both a topological order $v_1, v_2, ..., v_n$ and a cycle C. Let v_i be the lowest indexed vertex on C and v_j be the vertex on C just before v_i. However, the choice of i suggests that $i < j$, which contradicts the assumption that $v_1, v_2, ..., v_n$ is a topological order of G [2]. □

Theorem 7. In every DAG G, there is a vertex v with no incoming edges [2].

Proof. By contradiction: Let G be a directed graph in which each vertex has at least one incoming edge. Pick node v and follow the edges backward from v. We can continue this process indefinitely, because every vertex we encounter has at least one incoming edge. After $n + 1$ iterations, we will have visited at least one vertex twice, which contradicts the assumption that G is a DAG [2]. □

CYK 11. Which statement is true?

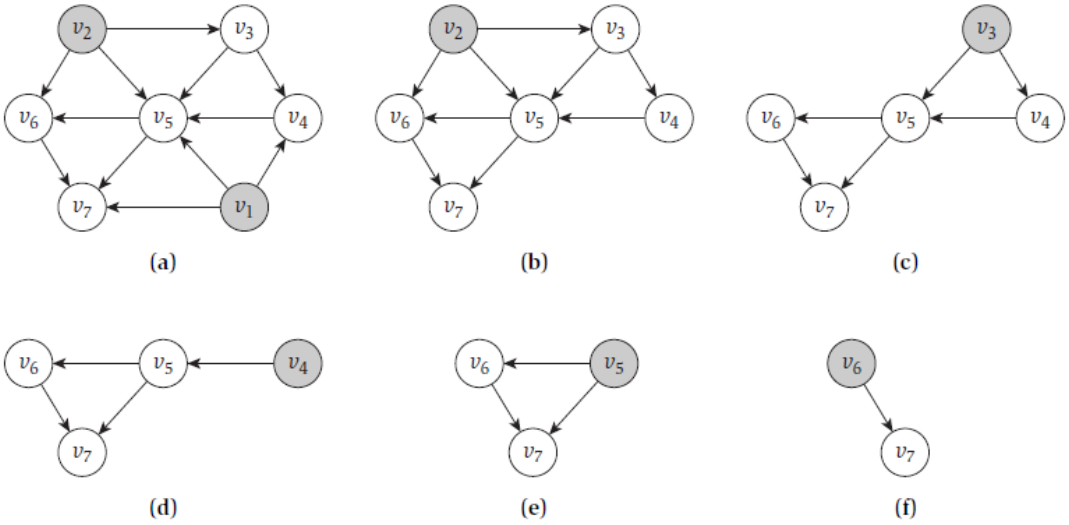
a All directed cyclic graphs have topological orderings
b All directed acyclic graphs have topological orderings
c All directed graphs must have a cycle
d All directed graphs have topological orderings

Correct answer: b

Algorithm 5: Topological Ordering of a DAG

Result: To compute a topological ordering of G:
Find a vertex v with no incoming edges and order it first
for *vertex v with no incoming edges* **do**
  Delete v from G
  Recursively compute a topological ordering of $G - \{v\}$, and append this order after v
end

Example 10: The topological ordering process for a DAG with seven vertices is shown below [2].



Add Instructor's Video Here

My screen recording programming and explaining the following Python script on Google Colab.

Python Script. Here is a Python script for topological order of a DAG.

```
# Topological ordering of a DAG
from collections import defaultdict
#Class to represent a graph
class Graph:
        def __init__(self,vertices):
                self.graph = defaultdict(list) # Dictionary containing
                ↪  adjacency list
                self.V = vertices # Number of vertices

        # Function for adding an edge to graph
        def addEdge(self,u,v):
                self.graph[u].append(v)

        # A recursive function used by topologicalSort
        def topologicalSortUtil(self,v,visited,stack):
                # Mark the current node as visited
                visited[v] = True
                # Recur for all the vertices adjacent to this vertex
                for i in self.graph[v]:
                        if visited[i] == False:
                                self.topologicalSortUtil(i,visited,stack)
                # Push current vertex to stack which stores result
                stack.insert(0,v)

        # Function for Topological Sort, using uses recursive
        ↪  topologicalSortUtil()
        def topologicalSort(self):
                # Mark all vertices as not visited
                visited = [False]*self.V
                stack =[]
                # Call the recursive helper function to store Topological
                ↪  Sort starting from all vertices one by one
                for i in range(self.V):
                        if visited[i] == False:
                                self.topologicalSortUtil(i,visited,stack)
                # Print contents of stack
                print(stack)
```

Example 10 (revisited): Compute the topological order of the DAG in Example 10 by running the above Python script along with the following script.

```
# Driver code
g = Graph(8)
g.addEdge(1,4)
```

```
g.addEdge(1,5)
g.addEdge(1,7)
g.addEdge(2,3)
g.addEdge(2,5)
g.addEdge(2,6)
g.addEdge(3,4)
g.addEdge(3,5)
g.addEdge(4,5)
g.addEdge(5,6)
g.addEdge(5,7)
g.addEdge(6,7)

print("The topological order of the DAG is:")
g.topologicalSort()
```

Quiz/Check Your Knowledge	
Title	**Prompt**
CYK1–CYK11	[Add_prompt]

Discussion	
Title	**Prompt**
Applying Concepts	In this module we looked at X, how does that relate to ...? Refer to the ..... rubric for additional information.

Assignment	
Title	**Prompt**
Application Memo	Write a memo that outlines... Refer to the ..... rubric for additional information.

Module Summary
TBD.

Facilitator Guide

Facilitator Introduction

Provide a short 2-3 sentence overview of the module, what students should do, and how it fits into the course as a whole

Required Elements

Identify the required elements and why they are required.

Instructor Flexibility

Identify areas of the module that are flexible - such as discussion questions. Include options if available or requirements of the changed elements

Possible Challenges For Students

Identify areas of the module that may be especially challenging for students, and include suggestions on how to support them.