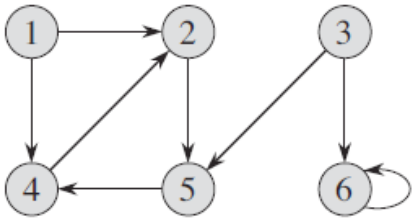
**CSCI205 Computer Science III**
**Lab Assignment**
**Graphs**

For the exercises that require a Graph, you should use the provided ***Graph.h*** and ***Vertex.h*** implementations. You may have to make some modifications when extra information needs tracking.

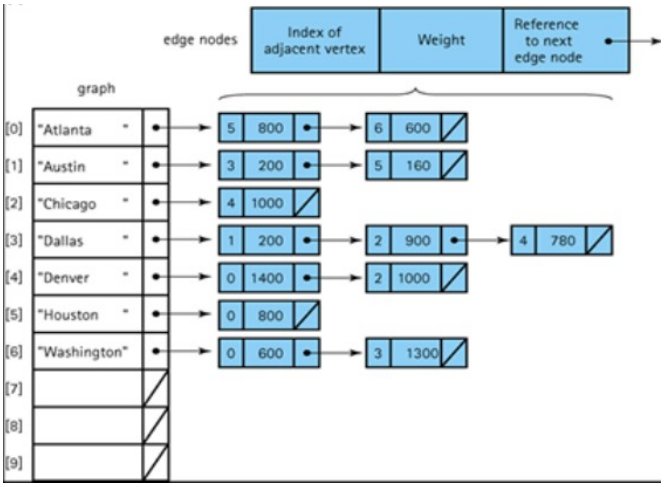
1) Sketch the unweighted graph and adjacency list represented by the following adjacency matrix

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

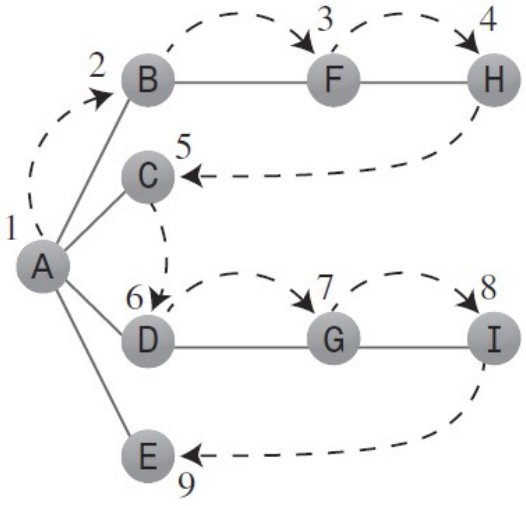
2) Sketch the adjacency list and adjacency matrix that represents the following unweighted graph



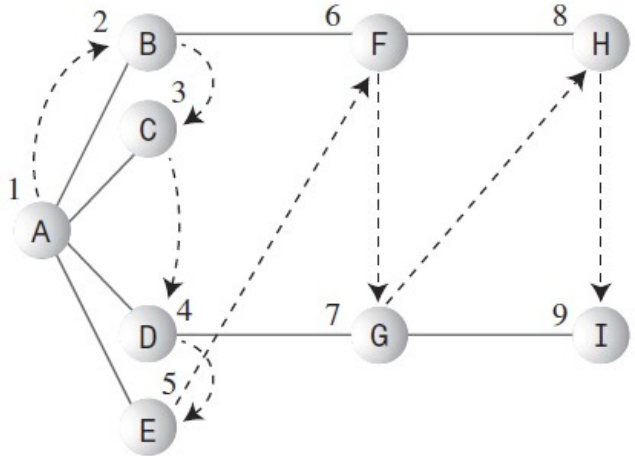

3) Sketch the graph and adjacency matrix represented by the following adjacency list



**Depth-First Search** The depth-first search follows a path from a source vertex to its culmination before returning to visit the source's other neighbors. It uses a **stack** to remember where it should go when it reaches a dead end.




**Breadth-First Search**
BFS is implemented using a **queue** instead of a stack and visits all of the source vertex's neighbors before going deeper into the field.




The breadth-first search has an interesting property: It first finds all the vertices that are one edge away from the starting point, then all the vertices that are two edges away, and so on. This is useful if you are trying to find the shortest path from the starting vertex to a given vertex. You start a BFS, and when you find the specified vertex, you know the path you have traced so far is the shortest path to the node. If there were a shorter path, the BFS would have found it already.

**You have been provided with two classic implementations of these algorithms.**

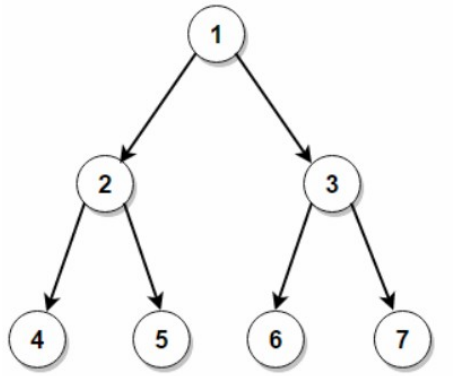
1. Runestone provides the implementation that utilizes vertex "coloring" to communicate visited status
2. The class repository includes an implementation that uses an array of booleans to communicate visited status

**Exercises:** For each of the following exercises create a C++ program to implement the algorithms. These are classic CS problems and as such you can find examples online. Use these examples to study and understand the algorithms. I'd like you to implement the algorithms using the ***Graph.h and Vertex.h*** classes that have been provided to you. You are free to modify the Vertex class if you feel that additional information is necessary to complete a problem. If you do modify the class include ample comments describing why you made this choice. Include multiple cases in your output. Name each "main" file according to the exercise.

**Exercise: Level order traversal of a binary tree**

Given a binary tree, print its nodes level by level, i.e., print all nodes of level 1 first, followed by nodes of level 2 and so on… Print nodes for any level from left to right.

For example, the level order traversal for the following tree is `1, 2, 3, 4, 5, 6, 7`:




We have already discussed pre, post and in-order traversals of the binary tree. These are variations of depth-first traversal of a Tree. Level order traversals are also possible, where every node on a level is visited before descending to a lower level. This type of search is called a level order traversal or ***breadth-first,*** as the search tree is broadened as much as possible on each depth before going to the next depth.

Print all nodes of level 1 first, followed by level 2, until level h, where h is the tree's height. We can print all nodes present in a level by modifying the preorder traversal on the tree.

**Exercise: Shortest Path through a maze**

This is a common problem in Computer Science. Given a maze in the form of binary rectangular matrix, find the shortest path's length from a given source to a given destination. A path consists of cells with 1, and at any given moment, we can only move one cell in one of the four directions. The valid moves are:

**Go Top:** (x, y) ——> (x – 1, y)
**Go Left:** (x, y) ——> (x, y – 1)
**Go Down:** (x, y) ——> (x + 1, y)
**Go Right:** (x, y) ——> (x, y + 1)

For example, consider the following binary matrix. If `source = (0, 0)` and `destination = (7, 5)`, the shortest path from source to destination has length 12.

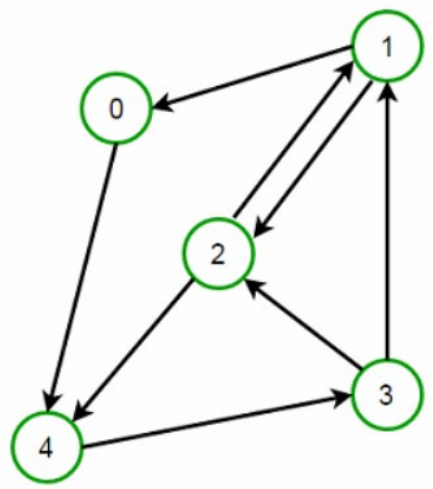
[ **1** **1** **1** 1 1 0 0 1 1 1 ]
[ 0 1 **1** 1 1 1 0 1 0 1 ]
[ 0 0 **1** 0 1 1 1 0 0 1 ]
[ 1 0 **1** **1** 1 0 1 1 0 1 ]
[ 0 0 0 **1** 0 0 0 1 0 1 ]
[ 1 0 1 **1** **1** 0 0 1 1 0 ]
[ 0 0 0 0 **1** 0 0 1 0 1 ]
[ 0 1 1 1 **1** **1** 1 1 0 0 ]
[ 1 1 1 1 1 0 0 1 1 1 ]
[ 0 0 1 0 0 1 1 0 0 1 ]

1. Create an empty queue and enqueue the source vertex having a distance 0 from the source (itself) and mark it as visited.
2. Continue until the queue is exhausted.
    - Dequeue the front vertex.
    - If the vertex is the destination node, then return its distance.
    - Otherwise, for each of four adjacent cells, enqueue each vertex with `+1` distance and mark them as visited.
3. If the destination has not been reached after visiting all vertices, return false.
4. It may be helpful to modify the Vertex class to have a property to store the distance from the source.
5. It may be helpful to modify the Vertex class to have an `x and y` property to store its coordinates in the matrix.
6. Also, be sure to stay in bounds. You will need to check to see if you are on a matrix edge.

**Exercise: Strongly Connected Graph**

Given a directed graph, check if it is strongly connected or not. A directed graph is considered “strongly connected” if every vertex is reachable from every other vertex.

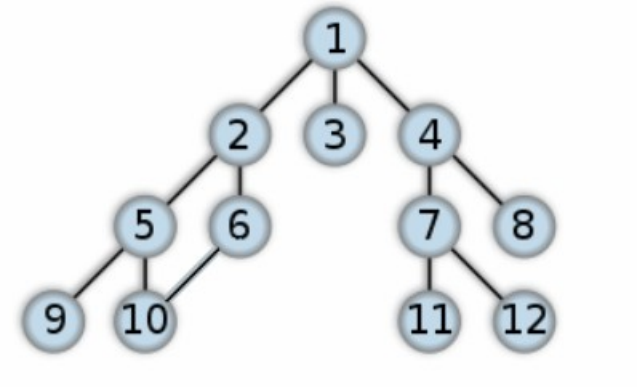
Examine the following strongly connected graph. Notice that a path exists between all pairs of vertices:




A simple solution is to perform a depth or breadth first traversal starting from every vertex in the graph. If each DFS/BFS call visits every other vertex in the graph, then the graph is strongly connected. Use a vector to track the “visited” status of the vertices.

**Exercise: Cycle Detection:** Given a connected, un-directed graph, determine if there are any cycles.

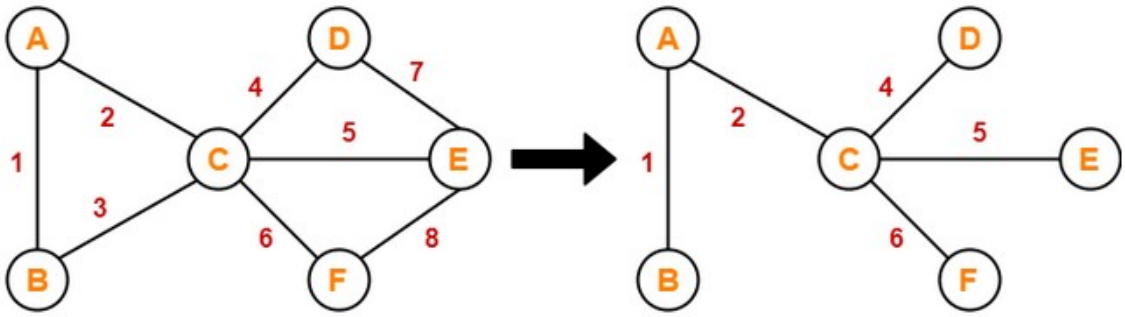
For example, the following graph contains a cycle `2–5–10–6–2`:




When performing a breadth first search from any vertex v in an un-directed graph, you may encounter **a cross-edge** that points to a previously discovered vertex that is neither an ancestor nor a descendant of the current vertex. Each “cross edge” defines a cycle in an un-directed graph. If the cross edge is x —> y, then since y is already discovered, we have a path from v to y (or from y to v since the graph is un-directed), where v is the starting vertex of BFS. So, we can say that we have a path v ~~ x ~ y ~~ v that forms a cycle. (Here, ~~ represents one more edge in the path, and ~ represents a direct edge).

**Exercise: Minimum Spanning Trees**

Often it would be nice to have an algorithm that, for any connected vertices and edges, would remove any extra edges. The result would be a graph with the minimum number of edges necessary to connect the vertices. For example, Figure a) shows five vertices with an excessive number of edges, while Figure b) shows the same vertices with the minimum number of edges necessary to connect them. This constitutes a ***minimum spanning tree (MST)***.

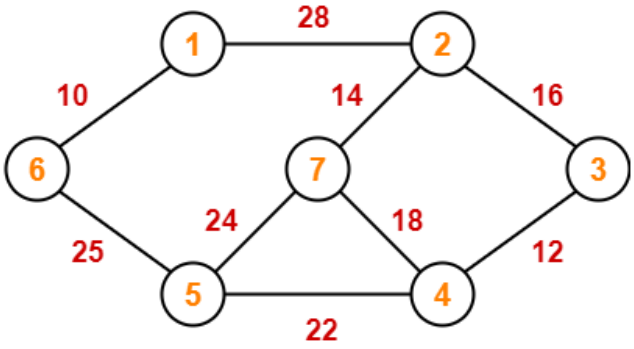



Given Graph

Minimum Spanning Tree (MST)

(Cost = 18 units)

Sketch the MST from the following graph using Prim's algorithm and determine the "cost"




Implement Prim's algorithm using the Graph.h and Vertex.h classes and demonstrate that your solution is correct.