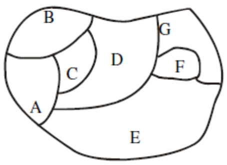
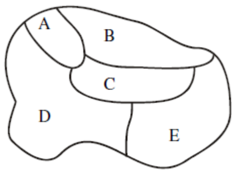
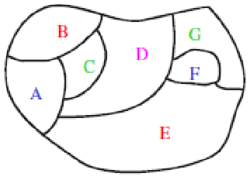
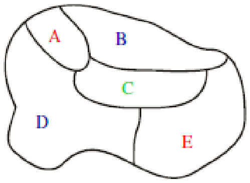
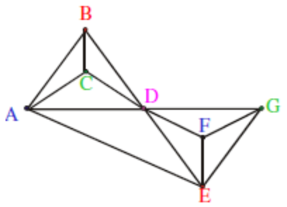
Graphs II:

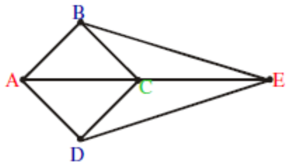






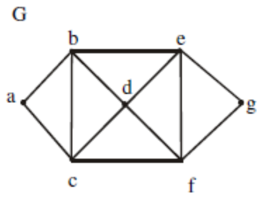


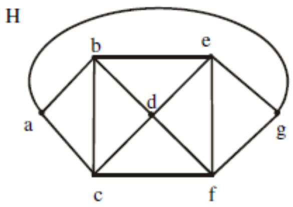




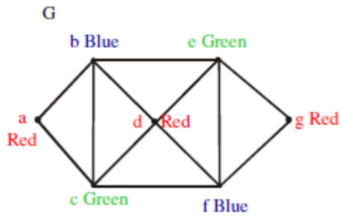
Note:

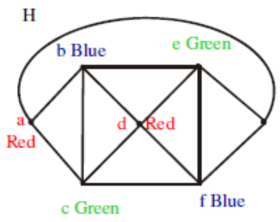
- Coloring of a simple graph is the assignment of a color to each vertex of the graph so that no two adjacent vertices are assigned the same color.
- The chromatic number of a graph is the least (minimum) number of colors for coloring of this graph.





Chormatic Number for the Graph G: 3
Chormatic Number for the Graph H: 4

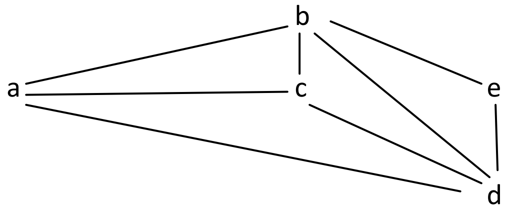




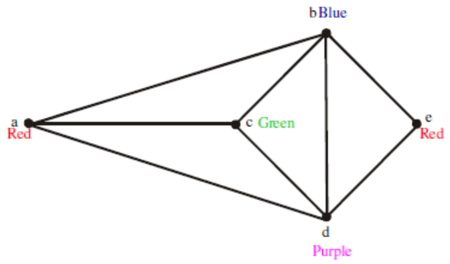
PROBLEM: Suppose that a chemist wishes to store five chemicals a, b, c, d and e in various areas of a warehouse. Some of these chemicals react violently when in contact, and so must be kept in separate areas. How many areas are needed?

In the following table, an asterisk indicates those pairs of chemicals that must be separated.

	a	b	c	d	e
a	-	*	*	*	-
b	*	-	*	*	*
c	*	*	-	*	-
d	*	*	-	-	-
e	-	*	-	*	-



Areas Required = Chromatic Number = 4



Paths and Circuits:

- Walk: A walk from v to w is a finite alternating sequence of adjacent vertices and edges of G.
- Closed Walk: A closed walk is a walk that starts and ends at the same vertex.
- Circuit: A circuit is a closed walk that does not contain a repeated edge.
- Simple Circuit: A simple circuit is a circuit that does not have any other repeated vertex except the first and last.
- Path: A path from v to w is a walk from v to w that does not contain a repeated edge.
- Simple Path: A simple path from v to w is a path that does not contain a repeated vertex

	Repeat Edge	**Repeat Vertex**	**Start and End at the same vertex**
Walk	Allowed	Allowed	Allowed
Closed Walk	Allowed	Allowed	YES (Mandatory)
Circuit	No	Allowed	YES (Mandatory)
Simple Circuit	No		YES (Mandatory)
Path	No	Allowed	Allowed
Simple Path	No	No	No

Connectedness:

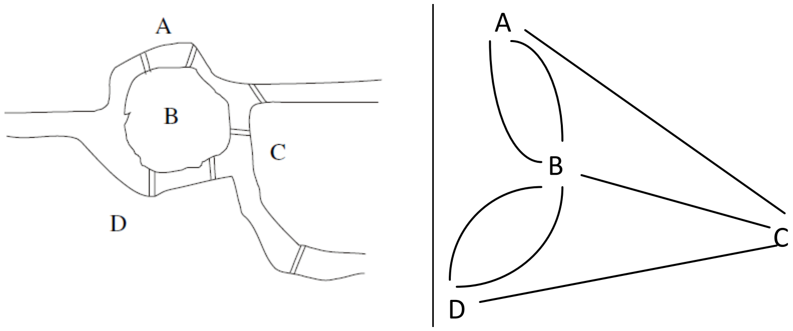
- Let G be a graph. Two vertices v and w of G are connected if, and only if, there is a walk from v to w.
- The graph G is connected if, and only if, given any two vertices v and w in G, there is a walk from v to w.

Euler Circuit:

- Let G be a graph. An Euler circuit for G is a circuit that contains every vertex and every edge of G.
- An Euler circuit for G is sequence of adjacent vertices and edges in G that starts and ends at the same vertex uses every vertex of G **at least once**, and used every edge of G **exactly** once.

Theorem: A graph G has an Euler circuit if, and only if, G is connected and every vertex of G has an even degree.

KONIGSBERG Bridges Problem: Is it possible for a person to take a walk around town, starting and ending at the same location and crossing each of the seven bridges exactly once? Answer: **NO**, because all vertices do **NOT** have even degree.



Hamiltonian Circuit:

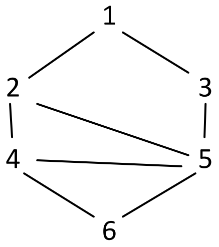
- Given a graph G, a Hamiltonian circuit for G is a simple circuit that includes every vertex of G.
- A Hamiltonian circuit for G is a sequence of adjacent vertices and distinct edges in which every vertex of G appears exactly once.

Traversals:

- Breadth First Search
- Depth First Search

Breadth First Search: O(V+E)

- Traverse in Layers
- Queue is used
- Boolean Array: Visited or Not
- Starts at some arbitrary node of a graph and explores the neighbor node first before moving to the next level neighbor.



1	2	3	4	5	6
N	N	N	N	N	N

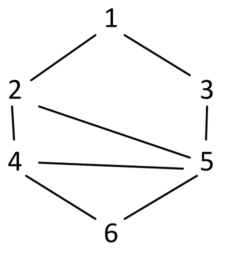
Queue: 1		1	2	3	4	5	6
		Y	N	N	N	N	N
Queue: 2,3	Output: 1	1	2	3	4	5	6
		Y	Y	Y	N	N	N
Queue: 3,4,5	Output: 1,2	1	2	3	4	5	6
		Y	Y	Y	Y	Y	N
Queue: 4,5	Output: 1,2,3	1	2	3	4	5	6
		Y	Y	Y	Y	Y	N
Queue: 5,6	Output: 1,2,3,4	1	2	3	4	5	6
		Y	Y	Y	Y	Y	Y
Queue: 6	Output: 1,2,3,4,5	1	2	3	4	5	6
		Y	Y	Y	Y	Y	Y
	Output: 1,2,3,4,5,6	1	2	3	4	5	6
		Y	Y	Y	Y	Y	Y

```
// Program to print BFS traversal from a given
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include <iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;
public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}
```

```
void Graph::addEdge(int v, int w)
{
   adj[v].push_back(w); // Add w to v’s list.
}

void Graph::BFS(int s)
{
   // Mark all the vertices as not visited
   bool *visited = new bool[V];
   for(int i = 0; i < V; i++)
      visited[i] = false;

   // Create a queue for BFS
   list<int> queue;

   // Mark the current node as visited and enqueue it
   visited[s] = true;
   queue.push_back(s);

   // 'i' will be used to get all adjacent
   // vertices of a vertex
   list<int>::iterator i;

   while(!queue.empty())
   {
      // Dequeue a vertex from queue and print it
      s = queue.front();
      cout << s << " ";
      queue.pop_front();

      // Get all adjacent vertices of the dequeued
      // vertex s. If a adjacent has not been visited,
      // then mark it visited and enqueue it
      for (i = adj[s].begin(); i != adj[s].end(); ++i)
      {
         if (!visited[*i])
         {
            visited[*i] = true;
            queue.push_back(*i);
         }
      }
   }
}

// Driver program to test methods of graph class
int main()
{
   // Create a graph given in the above diagram
   Graph g(4);
   g.addEdge(0, 1);
   g.addEdge(0, 2);
   g.addEdge(1, 2);
   g.addEdge(2, 0);
   g.addEdge(2, 3);
   g.addEdge(3, 3);

   cout << "Following is Breadth First Traversal "
       << "(starting from vertex 2) \n";
   g.BFS(2);

   return 0;
}
```

Depth First Search: O(V+E)

- Traverse in Depth
- Stack is used
- Boolean Array: Visited or Not
- Go forward in depth while there is any such possibility, if not then backtrack.

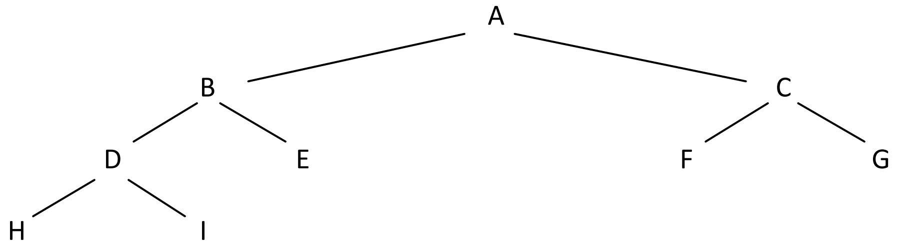


1	2	3	4	5	6
N	N	N	N	N	N

Stack: 1		1	2	3	4	5	6	
		Y	N	N	N	N	N	
Stack: 2,3	Output: 1	1	2	3	4	5	6	(Popped 1)
		Y	Y	Y	N	N	N	
Stack: 3 ,4,5	Output: 1,2	1	2	3	4	5	6	(Popped 2)
		Y	Y	Y	Y	Y	N	
Stack: 4,5	Output: 1,2,3	1	2	3	4	5	6	(Popped 3)
		Y	Y	Y	Y	Y	N	
Stack: 5,6	Output: 1,2,3,4	1	2	3	4	5	6	(Popped 4)
		Y	Y	Y	Y	Y	Y	
Stack: 6	Output: 1,2,3,4,5	1	2	3	4	5	6	(Popped 5)
		Y	Y	Y	Y	Y	Y	
	Output: 1,2,3,4,5,6	1	2	3	4	5	6	(Popped 6)
		Y	Y	Y	Y	Y	Y	

```
// C++ program to print DFS traversal from
// a given vertex in a  given graph
#include<bits/stdc++.h>
using namespace std;

// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}

// Driver code
int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
            " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}
```

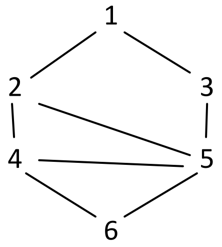
Note: Tree is also a Graph



Starting Node: A

BFS: A,B,C,D,E,F,G,H,I Level-Ordering

DFS: A,B,D,H,I,E,C,F,G Pre-Order Traversal of a binary tree



Adjacency List:

Node	**Neighbors**
1	2,3
2	1,4,5
3	1,5
4	2,5,6
5	2,3,4,6
6	4,5