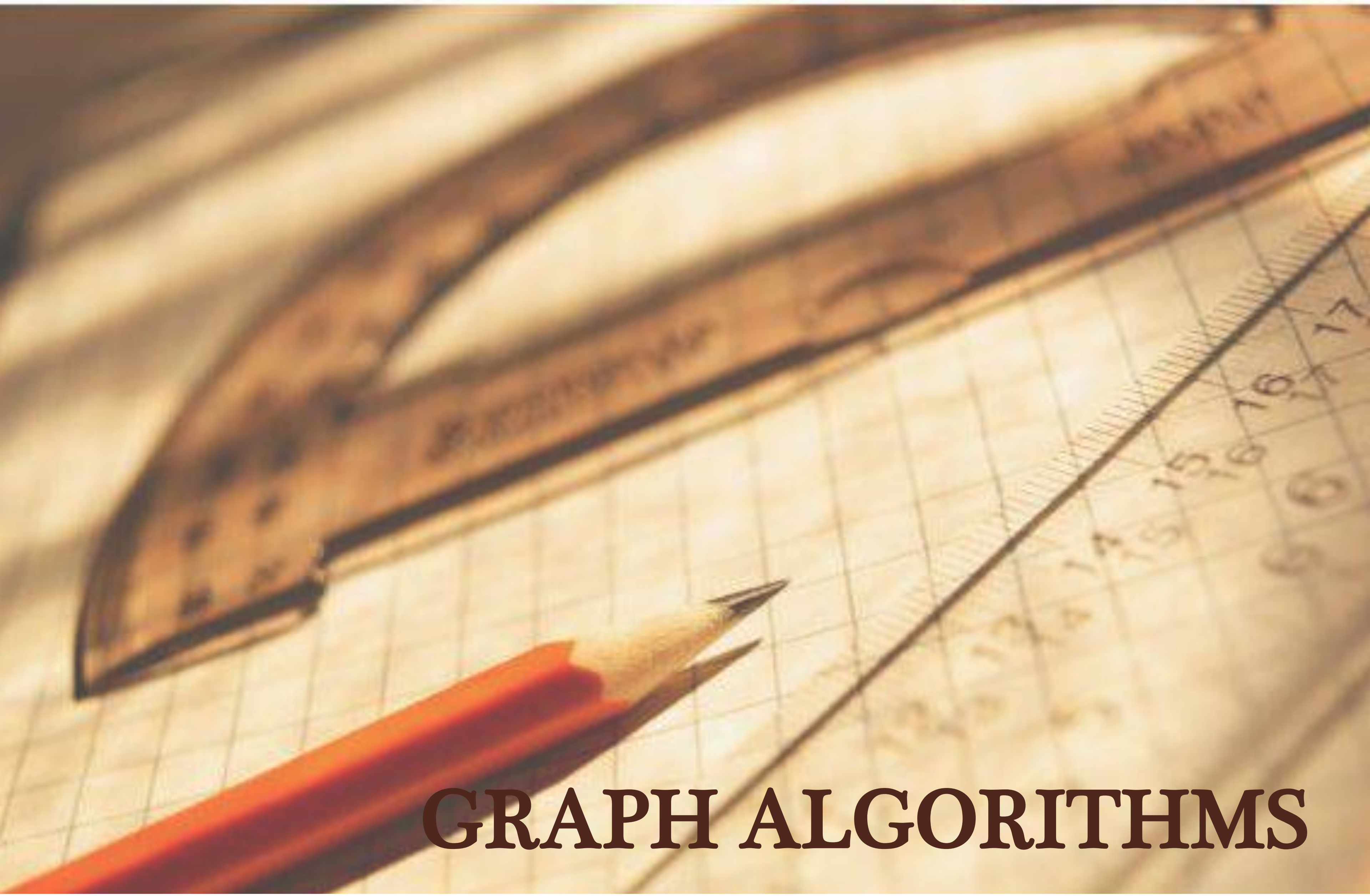
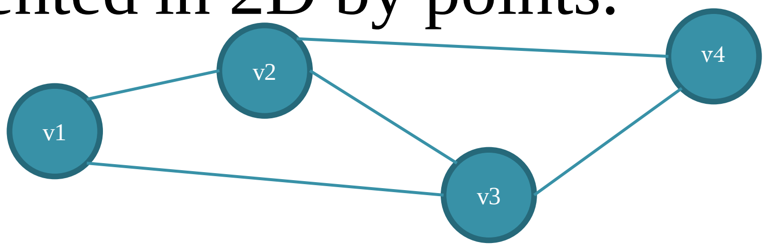


GRAPH ALGORITHMS

By: Tamal Chakraborty

Definitions & Terminology

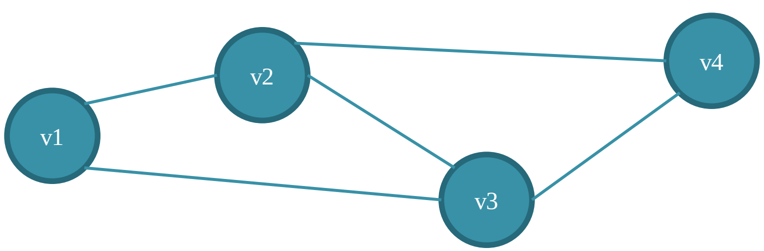
- A graph is a practical representation of the relationship between some objects, represented in a 2D diagram.
- The set of objects [V = {v1, v2, …}] are called vertices and are represented in 2D by points.



- The set of relationships [E = {e1, e2, …}] between the objects are called edges and are represented on 2D by lines.
- Every edge e_k, is associated with a pair of vertices v_i & v_j, which are called the end-vertices of e_k.

Adjacency Matrix representation of a Graph

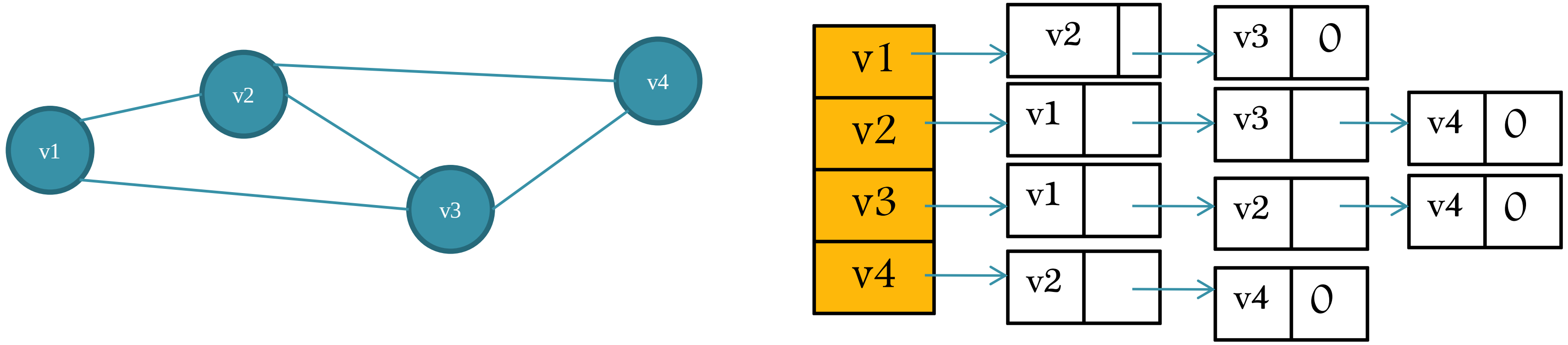
- A graph G = (V, E) Can be represented by an Adjacency matrix A = [A_{ij}] according to the following rules:
- A_{ij} = 1, if there is an edge between vertex i, j
- A_{ij} = 0, if there is no edge between vertex i, j
- For example the graph below can be represented by its adjacency matrix:



	v1	v2	v3	v4
v1	0	1	1	0
v2	1	0	1	1
v3	1	1	0	1
v4	0	1	1	0

Adjacency List representation of a Graph

- In this representation, the n rows of the adjacency matrix are represented as n linked lists.
- There is one list for each vertex in G.
- The nodes in list i represent the vertices that are adjacent to vertex i.
- For example the graph below can be represented by its adjacency list:



- The number of edges incident on a vertex (self loops counted twice) is called the degree of that vertex. For a graph with no self loops the degree of a vertex can be obtained by counting the number of nodes in the adjacency list.

Traversing a Graph

- One of the most fundamental graph problem is to traverse a graph.
- We have to start from one of the vertices, and then mark each vertex when we visit it. For each vertex we maintain three flags:
  1. Unvisited
  2. Visited but unexplored
  3. Visited and completely explored
- The order in which vertices are explored depends upon the kind of data structure used to store intermediate vertices.
  1. Queue (FIFO)
  2. Stack (LIFO)

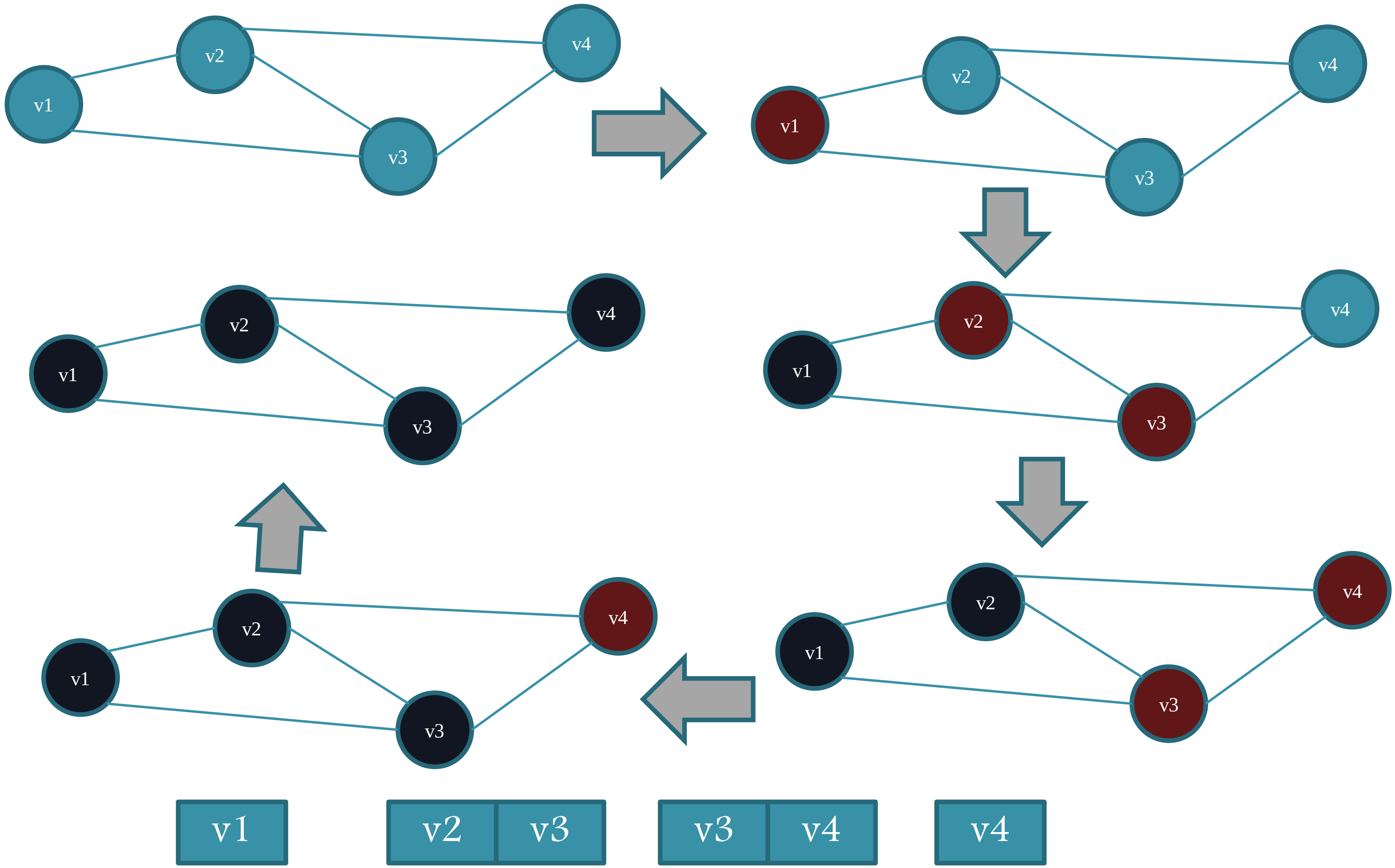
Breadth First Search (BFS)

- In this technique we start from a given vertex v and then mark it as visited (but not completely explored).
- A vertex is said to be explored when all the vertices adjacent to it are visited.
- All vertices adjacent to v are visited next, these are new unexplored vertices.
- The vertex v is now completely explored.
- The newly visited vertices which are not completely explored are put at the end of a queue.
- The first vertex of this queue is explored next.
- Exploration continues until no unexplored vertices are left.

Breadth First Search (BFS) Algorithm

```
BFS(v)
{
        u = v;
        visited[v] = 1;
        do {
                for all vertices w adjacent to u do
                {
                        if (visited[w] == 0) {
                                add w to q;  // q is the queue of unexplored vertices
                                visited[w] = 1;
                        }
                }
                if q is empty then return; // no unexplored vertices
                u = front element of q; // get first unexplored vertex
                delete front eleent of q;
        } while (true);
}
```

Breadth First Search (BFS)



Breadth First Search (BFS)

- If BFS is used on a connected graph then all vertices in G get visited and the graph is traversed.
- However if G is not connected, a complete traversal can be made by repeatedly calling BFS for every vertex of the graph.
- If adjacency matrix is used BFS takes $O(n^2)$ time.
- If adjacency list is used then BFS takes $O(n + e)$ time.
- where n is the number of vertices in the graph, e is the number of edges in the graph.

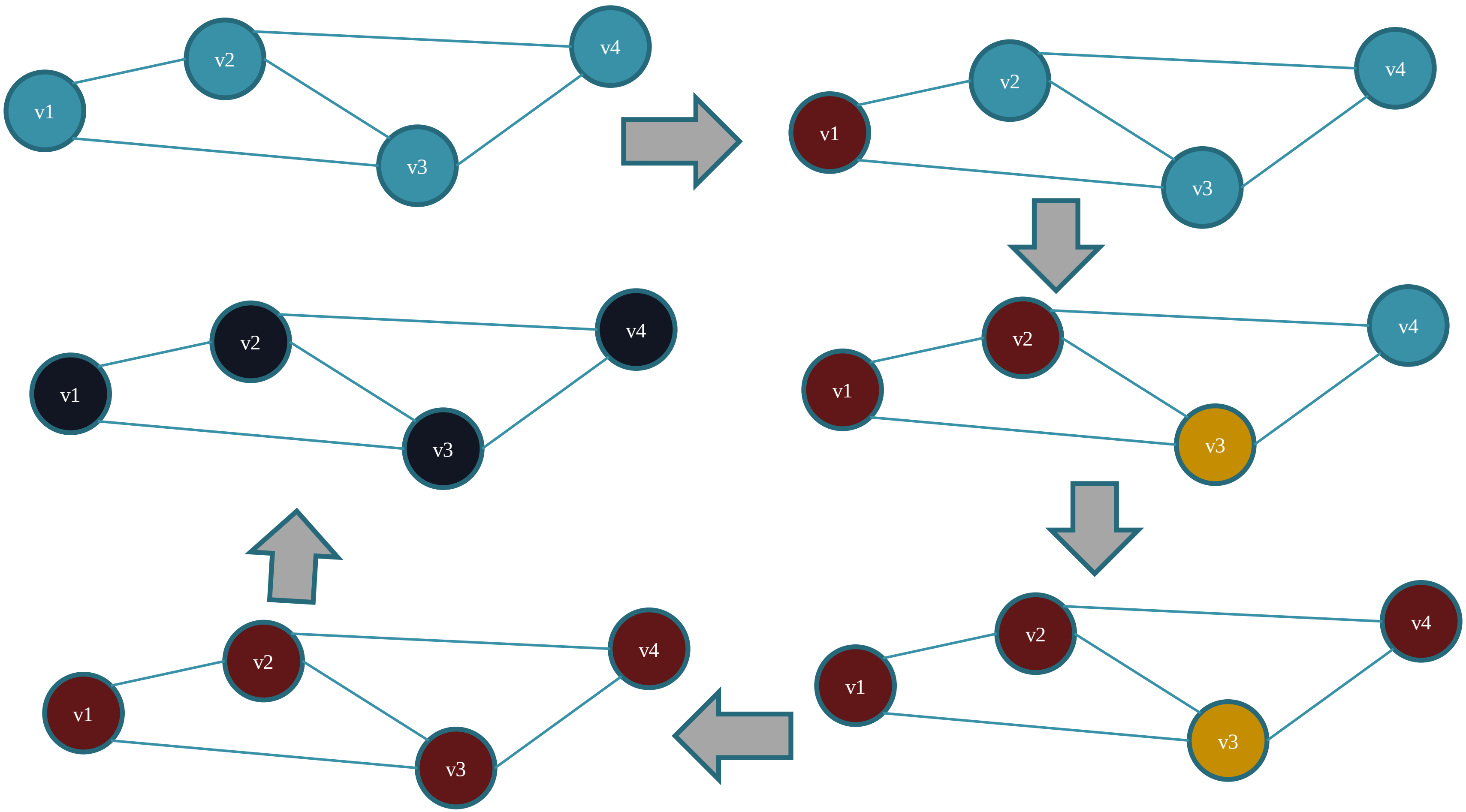
Depth First Search (DFS)

- In this technique we start from a given vertex v and then mark it as visited.
- A vertex is said to be explored when all the vertices adjacent to it are visited.
- An vertex adjacent to v are put at the top of a stack next.
- The top vertex of this stack is explored next.
- Exploration continues until no unexplored vertices are left.
- The search process can be described recursively.

Depth First Search (DFS) Algorithm

```
DFS(v)
{
        visited[v] = 1;
        for all vertices w adjacent to v do
        {
                if (visited[w] == 0) then DFS(w);
        }
  }
```

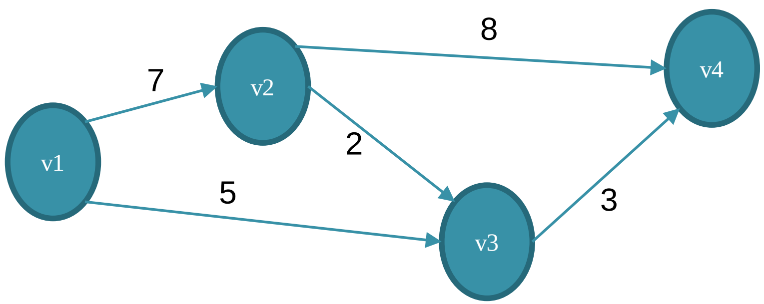
Depth First Search (DFS)



Depth First Search (DFS)

- In BFS a node is fully explored before exploration of a new node begins. Whereas in DFS exploration of a node is suspended as soon as a new unexplored node is reached, and exploration of this new node begins.
- If adjacency matrix is used DFS takes $O(n^2)$ time.
- If adjacency list is used then DFS takes $O(n + e)$ time.
- where n is the number of vertices in the graph, e is the number of edges in the graph.

Shortest Path Problem



$$D = \begin{pmatrix} 0 & 7 & 5 & \infty \\ \infty & 0 & 2 & 8 \\ \infty & \infty & 0 & 3 \\ \infty & \infty & \infty & 0 \end{pmatrix}$$

- A simple weighted directed graph G can be represented by an n x n matrix D = [d_{ij}] where:
  - d_{ij} = weight of the directed edge from i o j
  - = 0, if i = j
  - = ∞, if there is no edge between i and j
- We have to find out the shortest path from any given vertex to all other vertices

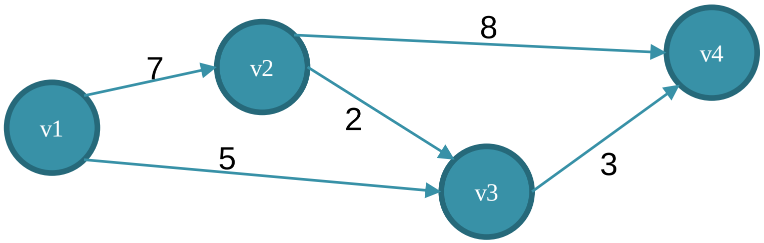
Dijkstra's Algorithm

- Begin
  1. Assign a permanent label 0 to the start vertex and a temporary label ∞ to all other vertices
  2. Update label of each vertex j with temporary label using the following rule:

     $$Label_j = \min[Label_j, Label_i + d_{ij}]$$

     Where i is the latest vertex permanently labeled and d_{ij} is the direct distance between i and j.
  3. Choose the smallest value among all the temporary labels as the new permanent label. In case of a tie select any one of the candidates.
  4. Repeat steps 2 and 3 until all the vertices are permanently labeled
- End

Dijkstra's Algorithm



v1	v2	v3	v4
0	∞	∞	∞
0	7	5	∞
0	7	5	8
0	7	5	8

Dijkstra's Algorithm

- Dijkstra's algorithm uses a similar approach as Breadth First Search
- Instead of pushing the visited vertices in a queue we use a priority queue
- The vertex with the max priority (minimum temporary label) is selected at each step for expansion
- For a matrix representation complexity of BFS is $O(n^2)$
- Hence Dijkstra's algorithm runs in $O(n^2)$ time

All pairs of shortest paths

- Given a vertex of a graph, Dijkstra's algorithm enables us to find the shortest path from that vertex to all other vertices
- The next problem is to find out the shortest path between any given pair of vertices of a graph
- The restriction is that G have no cycles with negative length
- If we allow G to contain cycles with negative length then the shortest path between any two vertices on this cycle is $-\infty$
- The all pairs of shortest path problem is to determine a matrix A such that A(i, j) is the length of the shortest path from i to j.

All pairs of shortest paths

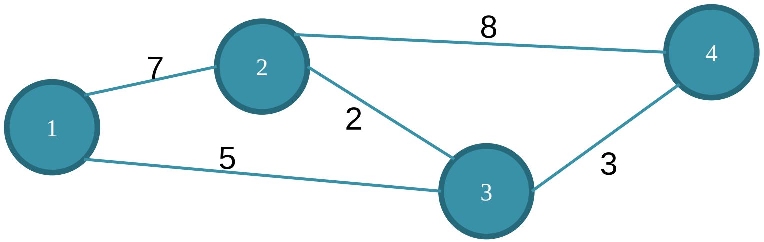
- We assume all the vertices of the graph are numbered from 1 to n
- Let $A^k(i, j)$ be the length of the shortest path from i to j going through no intermediate vertex greater than k
- Then there are two possibilities…
  1. The path from i to j goes through k: In which case we can split the path in two parts, one from i to k and the other from k to j. Note that neither of these two paths can go through any intermediate vertex greater than k – 1. Length of such a path is: $A^{k-1}(i, k) + A^{k-1}(k, j)$
  2. The path from i to j does not go through k: Which means that this path goes through no intermediate vertex greater than k–1. Its length would be: $A^{k-1}(i, j)$
- Clearly $A^k(i, j)$ is the minimum of these two choices
- Hence $A^k(i, j) = \min\{ A^{k-1}(i, j)\ ,\ A^{k-1}(i, k) + A^{k-1}(k, j) \}$

All pairs of shortest paths

```
AllPaths(cost, A, n)
{
      for i = 1 to n do
         for j = 1 to n do
                  A(i, j) = D(i, j);
      for k = 1 to n do
         for i = 1 to n do
                  for j = 1 to n do
                           A(i, j) = min{ A[i j], A[i, k] + A[k, j] }
}
```

Evidently the algorithm runs in $O(n^3)$ time

All pairs of shortest paths



$$D = \begin{pmatrix} 0 & 7 & 5 & \infty \\ 7 & 0 & 2 & 8 \\ 5 & 2 & 0 & 3 \\ \infty & 8 & 3 & 0 \end{pmatrix}$$

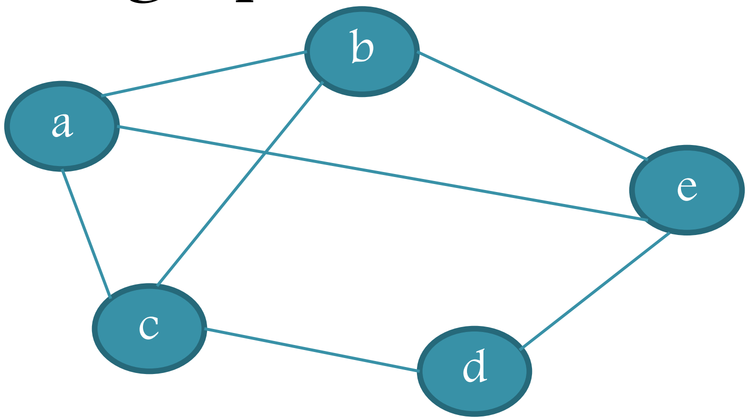
$$A = \begin{pmatrix} 0 & 7 & 5 & 8 \\ 7 & 0 & 2 & 5 \\ 5 & 2 & 0 & 3 \\ 8 & 5 & 3 & 0 \end{pmatrix}$$

Graph Coloring Problem

- Painting all the vertices of a graph with colors such that no two adjacent vertices have the same color is called the proper coloring of a graph.
- A graph in which every vertex has been assigned a color according to proper coloring is called a properly colored graph.
- A graph G that requires k different colors for its proper coloring, and no less, is called a k-chromatic graph. The number k is called the chromatic number of G.

Graph Coloring Problem

- Let G be a given graph, as shown below:



- G can be represented by the following adjacency matrix

$$\begin{vmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{vmatrix}$$

Graph Coloring Problem

- Let m be a given positive integer. In our example, say m = 3.
- We want to find whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used.
- We design a backtracking algorithm such that given the adjacency matrix of a graph G and a positive integer n, we can find all possible ways to properly color the graph.

Graph Coloring Problem

```
void next(int k) // find a legal color for x[k], k is the index of next vertex to color
{
        do
        {
                x[k] = (x[k] + 1) % (m + 1); // next highest color
                if (x[k] == 0) return; // all colors exhausted
                for (int j = 0; j < n; j++)
                {
                        if ((G[k][j] != 0) && (x[k] == x[j]))
                        // if (k,j) is an edge and if adjacent
                        // vertices have the same color
                        {
                                break;
                        }
                        if (j == (n - 1)) return; // new color found
                }
        } while (1); // otherwise try to find another color
}
```

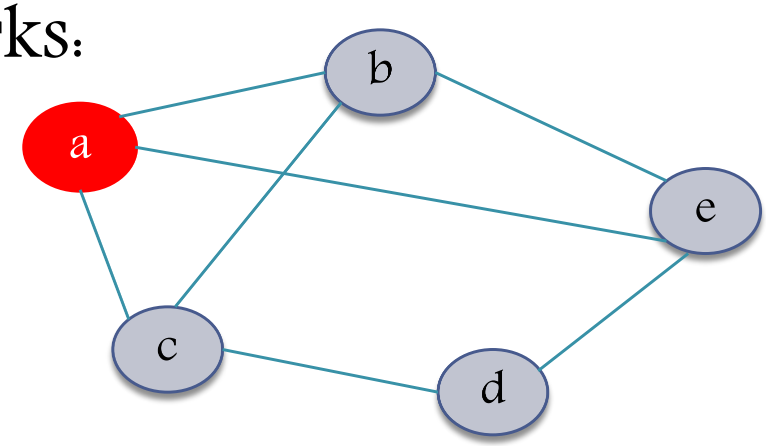
Graph Coloring Problem

```
void mColors(int k) // assign m colors to n vertices recursively
{
        do
        {
                next(k); // assign x[k] a legal color
                if (x[k] == 0) return; // no new colors available
                if (k == n) // at most m colors have been used
                {
                        for (int j = 0; j < n; j++)
                        {
                                cout<<x[j]<<" ";
                        }
                        cout<<endl;
                        break;
                }
                else
                {
                        mColors(k+1);
                }
        }
        while(1);
}
```

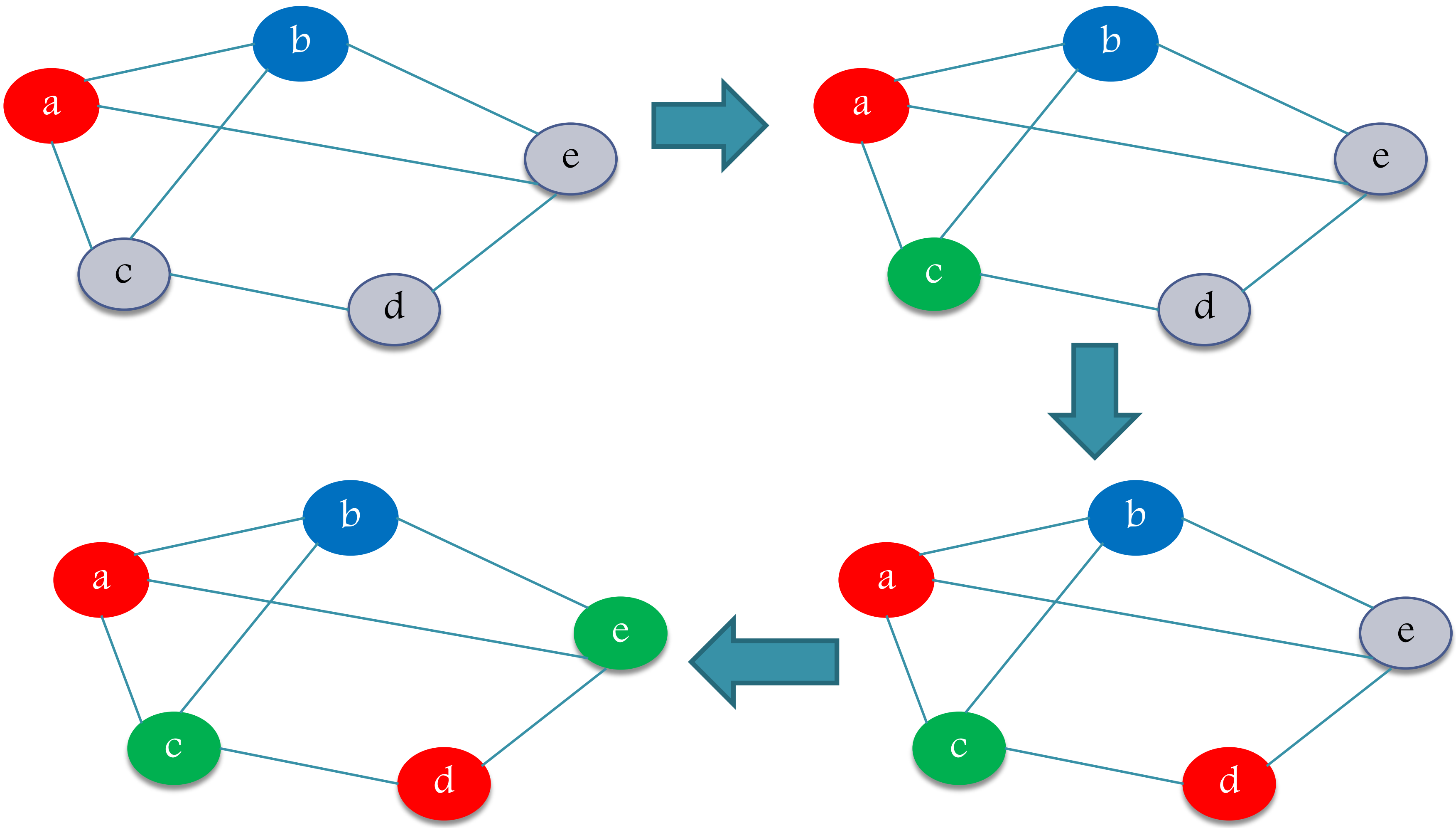
Graph Coloring Problem: How it works

- Let Red be color 1
- Let Green be color 2
- Let Blue be color 3

- Let us examine how the backtracking algorithm for coloring works:



Graph Coloring Problem

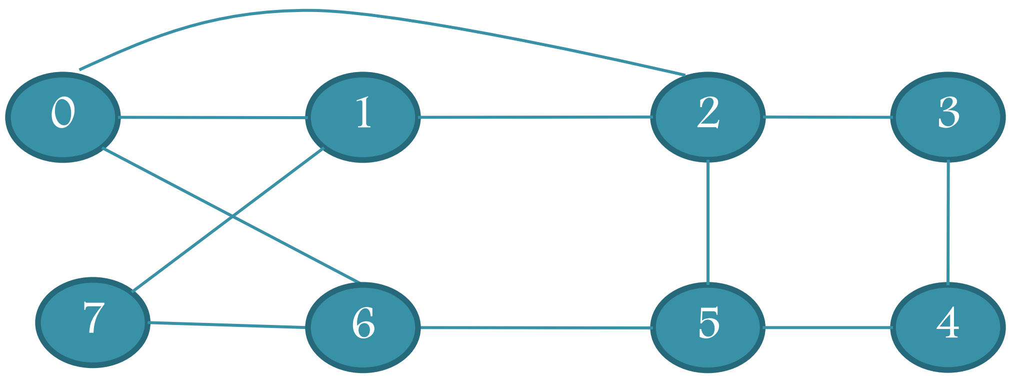


Hamiltonian Cycles

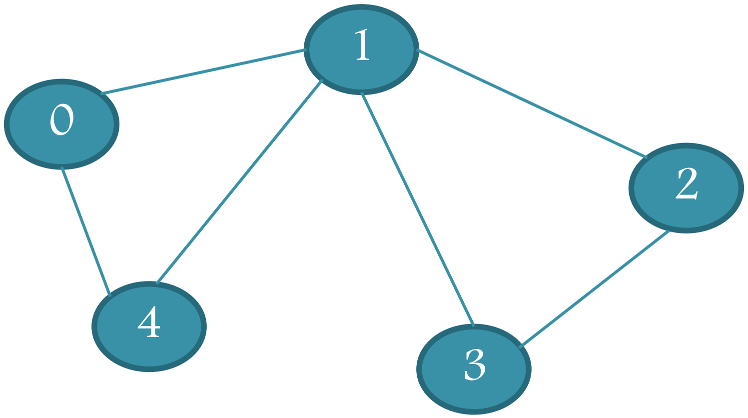
- Let G(V, E) be a connected graph with n vertices. A Hamiltonian cycle is a round trip path in G that visits every vertex once and returns to the starting position.

Hamiltonian Cycles

- The graph G1 below has Hamiltonian cycles.



- Whereas G2 has no Hamiltonian cycles.

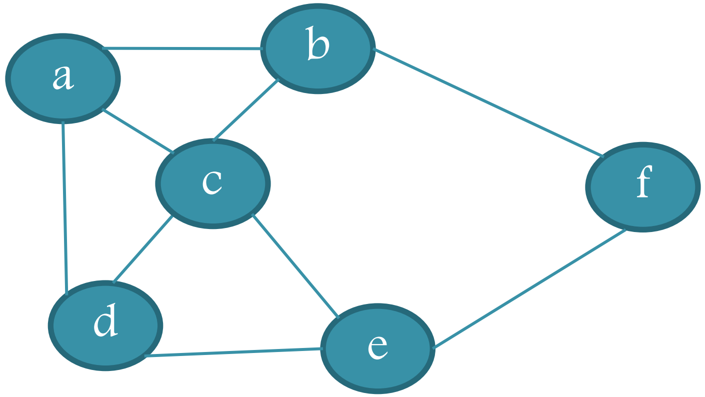


Hamiltonian Cycles

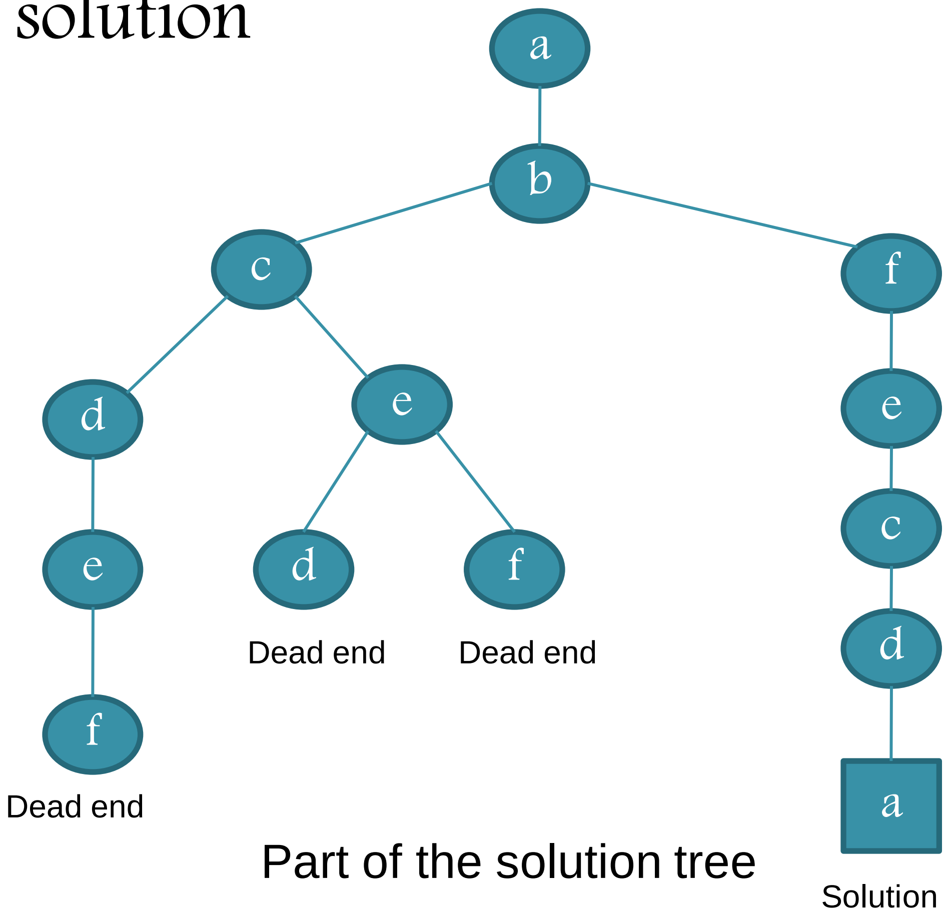
- The Hamiltonian cycle problem is defined as: "Does a graph G have a Hamiltonian Cycle?"
- We want to find all the Hamiltonian cycles in a graph

Hamiltonian Cycles

- The following figure illustrates the backtracking approach for a possible solution



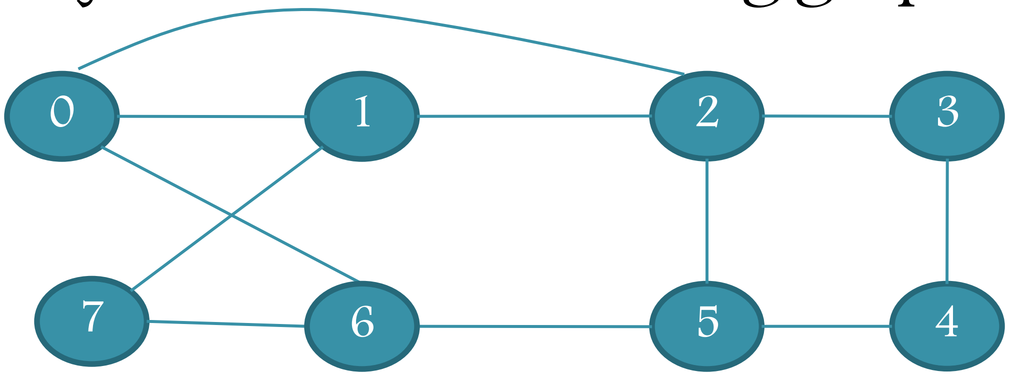
Given graph



Part of the solution tree

Hamiltonian Cycles

- Let us say we have the following graph as input:



- The adjacency matrix representation is:

0	1	1	0	0	0	1	0
1	0	1	0	0	0	0	1
1	1	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	1	0
1	0	0	0	0	1	0	1
0	1	0	0	0	0	1	0

Hamiltonian Cycles

- Let G(0 … n, 0 … n) be the adjacency matrix of the graph.
- Let (x1, x2, …, xn) be a solution such that xi represents the ith visited vertex of the proposed cycle.
- We design a backtracking algorithm to find the possible solutions of the Hamiltonian cycle problem.

Hamiltonian Cycle Problem

```
void next(int k)
{
        do
        {
                x[k] = (x[k] + 1) % (n + 1); // next vertex
                if (x[k] == 0) return;
                if (G[x[k-1]][x[k]] != 0)
                { // is there an edge
                        int j;
                        for (j = 0; j < k; j++)
                        {
                                if (x[j] == x[k]) break;
                        }
                        if (j == k) // check for distinctness, if true vertex is distinct
                        {
                                if ((k<n) || ((k==n) && (G[x[n]][x[0]] != 0)))
                                {
                                        return;
                                }
                        }

                }
        } while (1);
}
```

Hamiltonian Cycle Problem

```
void hamiltonian(int k)
{
        do
        {
                next(k); // generate values for x[k], next legal value
                if (x[k] == 0) return;
                if (k == n)
                {
                        for (int j = 0; j <= n; j++)
                        {
                                cout<<x[j]<<" ";
                        }
                        cout<<endl;
                        break;
                }
                else
                {
                        hamiltonian(k+1);
                }
        }
        while(1);
}
```

THANK YOU!