1. **Find if there is a path of more than k length from a source:**

Given a graph, a source vertex in the graph and a number k, find if there is a simple path (without any cycle) starting from given source and ending at any other vertex.



**Input :** Source s = 0, k = 58

**Output :** True

There exists a simple path 0 -> 7 -> 1-> 2 -> 8 -> 6 -> 5 -> 3 -> 4. Which has a total distance of 60 km which is more than 58.

**Input :** Source s = 0, k = 62

**Output :** False

In the above graph, the longest simple path has distance 61 (0 -> 7 -> 1-> 2 -> 3 -> 4 -> 5-> 6 -> 8, so output should be false for any input greater than 61.

**Approach:**

One important thing to note is, simply doing BFS or DFS and picking the longest edge at every step would not work. The reason is, a shorter edge can produce longer path due to higher weight edges connected through it.

The idea is to use Backtracking. We start from given source, explore all paths from current vertex. We keep track of current distance from source. If distance becomes more than k, we return true. If a path doesn’t produces more than k distance, we backtrack.

How do we make sure that the path is simple and we don’t loop in a cycle? The idea is to keep track of current path vertices in an array. Whenever we add a vertex to path, we check if it already exists or not in current path. If it exists, we ignore the edge.

**// Program to find if there is a simple path with**

**// weight more than k**

#include<bits/stdc++.h>

using namespace std;

**// iPair ==> Integer Pair**

typedef pair<int, int> iPair;

**// This class represents a graph using**

**// adjacency list representation**

class Graph

{

int V; **// No. of vertices**

**// In a weighted graph, we need to store vertex**

**// and weight pair for every edge**

list< pair<int, int> > \*adj;

**//we will create an array of Adjacency lists dynamically**

**//the size of the adjacency list will be V   
 //as for every vertex we need an adjacency list**

**//Now, in the pair, first argument will be the adjacent vertex, second argument the weight associated to it**

bool pathMoreThanKUtil(int src, int k, vector<bool> &path);

public:

Graph(int V); **// Constructor**

**// function to add an edge to graph**

void addEdge(int u, int v, int w);

bool pathMoreThanK(int src, int k);

};

**// Returns true if graph has path more than k length**

**bool Graph::pathMoreThanK(int src, int k)**

**{**

**//Here, src is given source**

**//K is the given length**

**//Length is attached to an edge as a value**

**// Create a path array with nothing included**

**// in path**

vector<bool> path(V, false);

**// Add source vertex to path**

path[src] = 1;

return pathMoreThanKUtil(src, k, path);

**}**

**// Prints shortest paths from src to all other vertices**

bool Graph::pathMoreThanKUtil(int src, int k, vector<bool> &path)

{

**// If k is 0 or negative, return true;**

**//The initial distance is sent as k, after that if we choose an edge in the final solution, the length associated with the edge is deducted from k**

if (k <= 0)

return true;

**// Get all adjacent vertices of source vertex src and**

**// recursively explore all paths from src.**

list<iPair>::iterator i;

for (i = adj[src].begin(); i != adj[src].end(); ++i)

{

**// Get adjacent vertex and weight of edge**

int v = (\*i).first;

int w = (\*i).second;

**// If vertex v is already there in path, then**

**// there is a cycle (we ignore this edge)**

//We should always avoid choosing cycle. Because, then k’s value will not matter. Result will be always true

if (path[v] == true)

continue;

/**/If w(length associated with the edge source-v) of is more than k, return true**

**//Because, deducting it from k and finally generating the result as true will take one more step. Which is unnecessary**

if (w >= k)

return true;

**//add this vertex to path**

path[v] = true;

//If current length of the edge is not greater than k, but, choosing current vertex as true leads to a path which finally generates the result as true

if (pathMoreThanKUtil(v, k-w, path))

return true;

**// Backtrack**

path[v] = false;

**//For backtrack, we omit for choosing v (an adjacent vertex of src)**

}

**// If no adjacent could produce longer path, return false**

**//If one of the adjacent vertex produces the desired path, then we aleady return true**

return false;

}

**// Allocates memory for adjacency list**

**//constructor**

**//Now, we create an array of adjacent list of size V dynamically**

Graph::Graph(int V)

{

this->V = V;

adj = new list<iPair> [V];

}

**// Utility function to an edge (u, v) of weight w**

void Graph::addEdge(int u, int v, int w)

{

adj[u].push\_back(make\_pair(v, w));

adj[v].push\_back(make\_pair(u, w));

**//as you can understand, this is an undirected graph**

}

**// Driver program to test methods of graph class**

int main()

{

**// create the graph given in above fugure**

int V = 9;

Graph g(V);

**// making above shown graph**

g.addEdge(0, 1, 4);

g.addEdge(0, 7, 8);

g.addEdge(1, 2, 8);

g.addEdge(1, 7, 11);

g.addEdge(2, 3, 7);

g.addEdge(2, 8, 2);

g.addEdge(2, 5, 4);

g.addEdge(3, 4, 9);

g.addEdge(3, 5, 14);

g.addEdge(4, 5, 10);

g.addEdge(5, 6, 2);

g.addEdge(6, 7, 1);

g.addEdge(6, 8, 6);

g.addEdge(7, 8, 7);

int src = 0;

int k = 62;

g.pathMoreThanK(src, k)? cout << "Yes\n" :

cout << "No\n";

k = 60;

g.pathMoreThanK(src, k)? cout << "Yes\n" :

cout << "No\n";

return 0;

}

1. **Tug of War:**

Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly n/2 and if n is odd, then size of one subset must be (n-1)/2 and size of other subset must be (n+1)/2.

For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10. Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).

Let us consider another example where n is odd. Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}. The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}. The sums of elements in two subsets are 120 and 121 respectively.

The following solution tries every possible subset of half size. If one subset of half size is formed, the remaining elements form the other subset. We initialize current set as empty and one by one build it. There are two possibilities for every element, either it is part of current set, or it is part of the remaining elements (other subset). We consider both possibilities for every element. When the size of current set becomes n/2, we check whether this solutions is better than the best solution available so far. If it is, then we update the best solution.

Following is the implementation for Tug of War problem. It prints the required arrays.

#include <iostream>

#include <stdlib.h>

#include <limits.h>

using namespace std;

**// function that tries every possible solution by calling itself recursively**

void TOWUtil(int\* arr, int n, bool\* curr\_elements, int no\_of\_selected\_elements,

bool\* soln, int\* min\_diff, int sum, int curr\_sum, int curr\_position)

{

**// checks whether the it is going out of bound**

if (curr\_position == n)

return;

**//no longer there’s a list from where we can choose element if the chosen subset is still not of size n/2**

**// checks that the numbers of elements left are not less than the**

**// number of elements required to form the solution**

if ((n/2 - no\_of\_selected\_elements) > (n - curr\_position))

return;

**//Because, if n-curr\_position< (n/2-no\_of\_selected\_elements)**

**//There is not enough element left to form the solution**

**//we cannot make the the chosen subset of size n/2**

**// consider the cases when current element is not included in the solution**

TOWUtil(arr, n, curr\_elements, no\_of\_selected\_elements,

soln, min\_diff, sum, curr\_sum, curr\_position+1);  
 **//Now, as you can see, the upper case is when the current element is not included**

**//Because, no\_of\_selected\_elements is not incremented**

**//min\_diff, sum, curr\_sum are not updated**

**//Now, construct a solution which chooses the current element in the list**

**// add the current element to the solution**

no\_of\_selected\_elements++;

curr\_sum = curr\_sum + arr[curr\_position];

curr\_elements[curr\_position] = true;

**// checks if a solution is formed**

if (no\_of\_selected\_elements == n/2)

{

**// checks if the solution formed is better than the best solution so far**

if (abs(sum/2 - curr\_sum) < \*min\_diff)

{

\*min\_diff = abs(sum/2 - curr\_sum);

for (int i = 0; i<n; i++)

soln[i] = curr\_elements[i];

}

**//In that case, we found a more optimized full solution**

**//because, in the chosen set, there are n/2 elements**

**//and, abs(sum/2-curr\_sum)<\*min\_diff**

}

else

{

**// consider the cases where current element is included in the solution**

TOWUtil(arr, n, curr\_elements, no\_of\_selected\_elements, soln,

min\_diff, sum, curr\_sum, curr\_position+1);

}

**// removes current element before returning to the caller of this function**

curr\_elements[curr\_position] = false;

}

**// main function that generate an arr**

void tugOfWar(int \*arr, int n)

{

**// the boolean array that contains the inclusion and exclusion of an element**

**// in current set. The number excluded automatically form the other set**

bool\* curr\_elements = new bool[n];

**// The inclusion/exclusion array for final solution**

bool\* soln = new bool[n];

int min\_diff = INT\_MAX;

**//Current min\_diff is set as INT\_MAX so, we can keep minimizing it**

**int sum = 0;**

**for (int i=0; i<n; i++)**

**{**

**sum += arr[i];**

**curr\_elements[i] = soln[i] = false;**

**}**

//Finding total sum was important

//Soln will contain final set of solution

//Curr\_elements will contain the current set of solutions

**// Find the solution using recursive function TOWUtil()**

TOWUtil(arr, n, curr\_elements, 0, soln, &min\_diff, sum, 0, 0);

**// Print the solution**

cout << "The first subset is: ";

for (int i=0; i<n; i++)

{

if (soln[i] == true)

cout << arr[i] << " ";

}

cout << "\nThe second subset is: ";

for (int i=0; i<n; i++)

{

if (soln[i] == false)

cout << arr[i] << " ";

}

}

**// Driver program to test above functions**

int main()

{

int arr[] = {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4};

int n = sizeof(arr)/sizeof(arr[0]);

tugOfWar(arr, n);

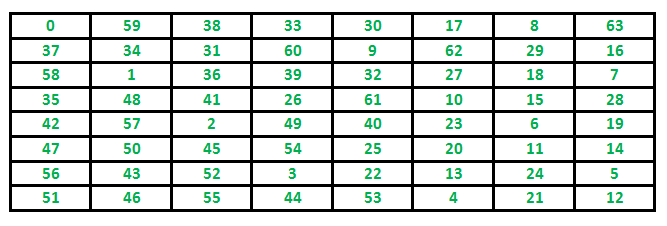
return 0;

}

1. **The Knight’s tour problem**

The knight is placed on the first block of an empty board and, moving according to the rules of chess, **must visit each square exactly once. (we need to ensure that)**

**Path followed by Knight to cover all the cells**



**Backtracking Algorithm for Knight’s tour:**

If all squares are visited

print the solution

Else

a) Add one of the next moves to solution vector and recursively

check if this move leads to a solution. (A Knight can make maximum

eight moves. We choose one of the 8 moves in this step).

b) If the move chosen in the above step doesn't lead to a solution

then remove this move from the solution vector and try other

alternative moves.

c) If none of the alternatives work then return false (Returning false

will remove the previously added item in recursion and if false is

returned by the initial call of recursion then "no solution exists" )

**// C program for Knight Tour problem**

#include<stdio.h>

#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N],

int xMove[], int yMove[]);

**/\* A utility function to check if i,j are valid indexes**

**for N\*N chessboard \*/**

**//Now, with the bound checking, we are also checking if a node is previously visited**

bool isSafe(int x, int y, int sol[N][N])

{

return ( x >= 0 && x < N && y >= 0 &&

y < N && sol[x][y] == -1);

}

**/\* A utility function to print solution matrix sol[N][N] \*/**

void printSolution(int sol[N][N])

{

for (int x = 0; x < N; x++)

{

for (int y = 0; y < N; y++)

printf(" %2d ", sol[x][y]);

printf("\n");

}

}

**/\* This function solves the Knight Tour problem using**

**Backtracking. This function mainly uses solveKTUtil()**

**to solve the problem. It returns false if no complete**

**tour is possible, otherwise return true and prints the**

**tour.**

**Please note that there may be more than one solutions,**

**this function prints one of the feasible solutions. \*/**

bool solveKT()

{

int sol[N][N];

**/\* Initialization of solution matrix \*/**

for (int x = 0; x < N; x++)

for (int y = 0; y < N; y++)

sol[x][y] = -1;

**//Initially all nodes are unvisited**

**/\* xMove[] and yMove[] define next move of Knight.**

**xMove[] is for next value of x coordinate**

**yMove[] is for next value of y coordinate \*/**

int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };

int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

**// Since the Knight is initially at the first block**

sol[0][0] = 0;

**//Now, it will contain the visiting sequence**

**//(0,0) will be the first one to become visited**

**/\* Start from 0,0 and explore all tours using**

**solveKTUtil() \*/**

if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == false)

{

**//(0,0) is the current move’s position**

**//1 is the sequence number of steps we will be looking at next step**

**//xMove and yMove are the arrays**

printf("Solution does not exist");

return false;

}

else

printSolution(sol);

return true;

}

**/\* A recursive utility function to solve Knight Tour**

**problem \*/**

int solveKTUtil(int x, int y, int movei, int sol[N][N],

int xMove[N], int yMove[N])

{

int k, next\_x, next\_y;

**//In the valid way, the move count cannot be >=64 since, move starts form 0**

if (movei == N\*N)

return true;

**/\* Try all next moves from the current coordinate x, y \*/**

for (k = 0; k < 8; k++)

{

next\_x = x + xMove[k];

next\_y = y + yMove[k];

if (isSafe(next\_x, next\_y, sol))

{

sol[next\_x][next\_y] = movei;

**//A node can be added**

**//Now, we will check if adding this node will lead to the final solution**

if (solveKTUtil(next\_x, next\_y, movei+1, sol,

xMove, yMove) == true)

return true;

else

sol[next\_x][next\_y] = -1;**// backtracking**

}

}

return false;

}

**/\* Driver program to test above functions \*/**

int main()

{

solveKT();

return 0;

}

1. **Rat in a Maze**

Maze is given as N\*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with limited number of moves.

**Backtracking Algorithm:**

If destination is reached

print the solution matrix

Else

a) Mark current cell in solution matrix as 1.

b) Move forward in horizontal direction and recursively check if this

move leads to a solution.

c) If the move chosen in the above step doesn't lead to a solution

then move down and check if this move leads to a solution.

d) If none of the above solutions work then unmark this cell as 0

(BACKTRACK) and return false.

**/\* C/C++ program to solve Rat in a Maze problem using**

**backtracking \*/**

#include<stdio.h>

**// Maze size**

#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N]);

**/\* A utility function to print solution matrix sol[N][N] \*/**

void printSolution(int sol[N][N])

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

printf(" %d ", sol[i][j]);

printf("\n");

}

}

**/\* A utility function to check if x,y is valid index for N\*N maze \*/**

bool isSafe(int maze[N][N], int x, int y)

{

**// if (x,y outside maze) return false**

**//Also, maze[x][y]=0 means it is a dead end  
 //so, in the path we cannot choose a node which is dead end**

if(x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)

return true;

return false;

}

**/\* This function solves the Maze problem using Backtracking. It mainly**

**uses solveMazeUtil() to solve the problem. It returns false if no**

**path is possible, otherwise return true and prints the path in the**

**form of 1s. Please note that there may be more than one solutions,**

**this function prints one of the feasible solutions.\*/**

bool solveMaze(int maze[N][N])

{

int sol[N][N] = { {0, 0, 0, 0},

{0, 0, 0, 0},

{0, 0, 0, 0},

{0, 0, 0, 0}

};

if(solveMazeUtil(maze, 0, 0, sol) == false)

{

printf("Solution doesn't exist");

return false;

}

printSolution(sol);

return true;

}

**/\* A recursive utility function to solve Maze problem \*/**

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])

{

**// if (x,y is goal) return true**

if(x == N-1 && y == N-1)

{

sol[x][y] = 1;

return true;

}

**// Check if maze[x][y] is valid**

if(isSafe(maze, x, y) == true)

{

**// mark x,y as part of solution path**

sol[x][y] = 1;

**/\* Move forward in x direction \*/**

if (solveMazeUtil(maze, x+1, y, sol) == true)

return true;

**/\* If moving in x direction doesn't give solution then**

**Move down in y direction \*/**

if (solveMazeUtil(maze, x, y+1, sol) == true)

return true;

**/\* If none of the above movements work then BACKTRACK:**

**unmark x,y as part of solution path \*/**

sol[x][y] = 0;

return false;

}

return false;

}

**// driver program to test above function**

int main()

{

int maze[N][N] = { {1, 0, 0, 0},

{1, 1, 0, 1},

{0, 1, 0, 0},

{1, 1, 1, 1}};

solveMaze(maze);

return 0;

}

The major thing is isSafe’s definition.

1. **N Queen’s Problem:**

We have discussed Knight’s tour and Rat in a Maze problems in Set 1 and Set 2 respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.

**Backtracking Algorithm**

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a

row due to clashes then we backtrack and return false.

1) Start in the leftmost column

2) If all queens are placed

return true

3) Try all rows in the current column. Do following for every tried row.

a) If the queen can be placed safely in this row then mark this [row,

column] as part of the solution and recursively check if placing

queen here leads to a solution.

b) If placing queen in [row, column] leads to a solution then return

true.

c) If placing queen doesn't lead to a solution then umark this [row,

column] (Backtrack) and go to step (a) to try other rows.

3) If all rows have been tried and nothing worked, return false to trigger

backtracking.

(column is fixed, row is varying. For instance, when we start with leftmost column, Now, suppose, we choose (0,0) for the first queen. Now, it might not lead us to the current solution. We will try (1,0), (2,0)….(n-1,0)

**/\* C/C++ program to solve N Queen Problem using**

**backtracking \*/**

#define N 4

#include<stdio.h>

**/\* A utility function to print solution \*/**

void printSolution(int board[N][N])

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

printf(" %d ", board[i][j]);

printf("n");

}

}

**/\* A utility function to check if a queen can**

**be placed on board[row][col]. Note that this**

**function is called when "col" queens are**

**already placed in columns from 0 to col -1.**

**So we need to check only left side for**

**attacking queens \*/**

bool isSafe(int board[N][N], int row, int col)

{

int i, j;

**/\* Check this row on left side \*/**

for (i = 0; i < col; i++)

if (board[row][i])

return false;

**/\* Check upper diagonal on left side \*/**

for (i=row, j=col; i>=0 && j>=0; i--, j--)

if (board[i][j])

return false;

**/\* Check lower diagonal on left side \*/**

for (i=row, j=col; j>=0 && i<N; i++, j--)

if (board[i][j])

return false;

return true;

}

**/\* A recursive utility function to solve N**

**Queen problem \*/**

bool solveNQUtil(int board[N][N], int col)

{

**/\* base case: If all queens are placed**

**then return true \*/**

if (col >= N)

return true;

**/\* Consider this column and try placing**

**this queen in all rows one by one \*/**

for (int i = 0; i < N; i++)

{

**/\* Check if queen can be placed on**

**board[i][col] \*/**

if ( isSafe(board, i, col) )

{

**/\* Place this queen in board[i][col] \*/**

board[i][col] = 1;

**/\* recur to place rest of the queens \*/**

if ( solveNQUtil(board, col + 1) )

return true;

**/\* If placing queen in board[i][col]**

**doesn't lead to a solution, then**

**remove queen from board[i][col] \*/**

board[i][col] = 0; **// BACKTRACK**

}

}

**/\* If queen can not be place in any row in**

**this column col then return false \*/**

**return false;**

**}**

**/\* This function solves the N Queen problem using**

**Backtracking. It mainly uses solveNQUtil() to**

**solve the problem. It returns false if queens**

**cannot be placed, otherwise return true and**

**prints placement of queens in the form of 1s.**

**Please note that there may be more than one**

**solutions, this function prints one of the**

**feasible solutions.\*/**

bool solveNQ()

{

int board[N][N] = { {0, 0, 0, 0},

{0, 0, 0, 0},

{0, 0, 0, 0},

{0, 0, 0, 0}

};

if ( solveNQUtil(board, 0) == false )

{

printf("Solution does not exist");

return false;

}

printSolution(board);

return true;

}

**// driver program to test above function**

int main()

{

solveNQ();

return 0;

}

# m Coloring Problem:

Given an **undirected graph** and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

**Input:**

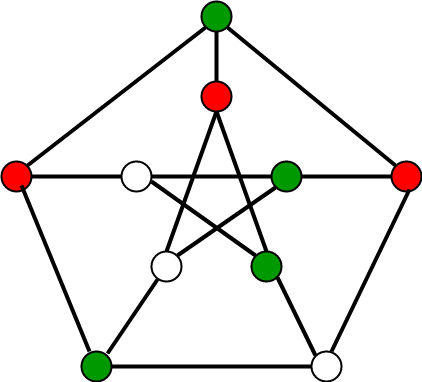
1) A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

2) An integer m which is maximum number of colors that can be used.

**Output:**

An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Following is an example of graph that can be colored with 3 different colors.



**Backtracking Algorithm**

The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, we check for safety by considering already assigned colors to the adjacent vertices. If we find a color assignment which is safe, we mark the color assignment as part of solution. If we do not a find color due to clashes then we backtrack and return false.

#include<stdio.h>

**// Number of vertices in the graph**

#define V 4

void printSolution(int color[]);

**/\* A utility function to check if the current color assignment**

**is safe for vertex v \*/**

bool isSafe (int v, bool graph[V][V], int color[], int c)

{

for (int i = 0; i < V; i++)

if (graph[v][i] && c == color[i])

**//If one of the adjacent vertices in already visited and has the color which we are trying to assign to assign to the current vertex v**

return false;

return true;

}

**/\* A recursive utility function to solve m coloring problem \*/**

bool graphColoringUtil(bool graph[V][V], int m, int color[], int v)

{

**/\* base case: If all vertices are assigned a color then**

**return true \*/**

if (v == V)

return true;

**/\* Consider this vertex v and try different colors \*/**

for (int c = 1; c <= m; c++)

{

**/\*check color one by one and try to assign it to current vertex\*/**

**/\* Check if assignment of color c to v is fine\*/**

if (isSafe(v, graph, color, c))

{

color[v] = c;

**/\* recur to assign colors to rest of the vertices \*/**

if (graphColoringUtil (graph, m, color, v+1) == true)

**//Instead of visiting adjacent vertices (which can lead to a cycle if we don’t visit them carefully) we visit vertices and check if it can be colored**

return true;

**/\* If assigning color c doesn't lead to a solution**

**then remove it \*/**

color[v] = 0;

**//If assigning color c doesn't lead to a solution then remove it**

**//Now, another thing, We need not to print all possible solutions**

**//Hence, if we succeed with color[v]=c we will not backtrack**

**//As, you can notice color[v]=0 this statement will only be executed if**

**(graphColoringUtil(graph,m,color,v+1)==true) does not work**

**//However, the key point is, instead of doing dfs or bfs, we just traverse vertices**

}

}

**/\* If no color can be assigned to this vertex then return false \*/**

return false;

}

**/\* This function solves the m Coloring problem using Backtracking.**

**It mainly uses graphColoringUtil() to solve the problem. It returns**

**false if the m colors cannot be assigned, otherwise return true and**

**prints assignments of colors to all vertices. Please note that there**

**may be more than one solutions, this function prints one of the**

**feasible solutions.\*/**

bool graphColoring(bool graph[V][V], int m)

{

**// Initialize all color values as 0. This initialization is needed**

**// correct functioning of isSafe()**

int \*color = new int[V];

**//Now, color array will be of size v. Since, finally, it will contain the color chosen for all vertices**

for (int i = 0; i < V; i++)

color[i] = 0;

**//Initially non color is chosen for any of the vertices. As, color is from 1 to m.**

**// Call graphColoringUtil() for vertex 0**

0: vertex 0

if (graphColoringUtil(graph, m, color, 0) == false)

{

printf("Solution does not exist");

return false;

}

**// Print the solution**

printSolution(color);

return true;

}

**/\* A utility function to print solution \*/**

void printSolution(int color[])

{

printf("Solution Exists:"

" Following are the assigned colors \n");

for (int i = 0; i < V; i++)

printf(" %d ", color[i]);

printf("\n");

}

**// driver program to test above function**

int main()

{

**/\* Create following graph and test whether it is 3 colorable**

**(3)------(2)**

**| / |**

**| / |**

**| / |**

**(0)-------(1)**

**\*/**

bool graph[V][V] = {{0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0},

};

int m = 3; **// Number of colors**

graphColoring (graph, m);

return 0;

}

**7.Hamiltonian Cycle:**

**Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path.** Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

**Input:**

A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

**Output:**

An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}

(0)--(1)----(2)

| / \ |

| / \ |

| / \ |

1. ------------(4)

In this undirected graph, we can find Hamiltonian cycle.

And the following graph doesn’t contain any Hamiltonian Cycle.

(0)-----(1)---(2)

| / \ |

| / \ |

| / \ |

(3) (4)

**Note:** that in the definition of Hamiltonian cycle, the presence of the word **undirected graph.**

**/\* C/C++ program for solution of Hamiltonian Cycle problem**

**using backtracking \*/**

#include<stdio.h>

**// Number of vertices in the graph**

#define V 5

void printSolution(int path[]);

**/\* A utility function to check if the vertex v can be added at**

**index 'pos' in the Hamiltonian Cycle constructed so far (stored**

**in 'path[]') \*/**

bool isSafe(int v, bool graph[V][V], int path[], int pos)

{

**/\* Check if this vertex is an adjacent vertex of the previously**

**added vertex. \*/**

**//it must be an adjacent vertex**

if (graph [ path[pos-1] ][ v ] == 0)

return false;

**/\* Check if the vertex has already been included.**

**This step can be optimized by creating an array of size V \*/**

for (int i = 0; i < pos; i++)

**//It will check if the vertex is previously chosen**

if (path[i] == v)

return false;

return true;

}

**/\* A recursive utility function to solve hamiltonian cycle problem \*/**

bool hamCycleUtil(bool graph[V][V], int path[], int pos)

{

**/\* base case: If all vertices are included in Hamiltonian Cycle \*/**

if (pos == V)

{

**//If all vertices are added in the Hamiltonian cycle. Because, if vertex from 0 to v-1 are added in the list**

**// And if there is an edge from the last included vertex to the**

**// first vertex**

**//Only at that point we will check if the last vertex with is pos-1 has an indirect edge to the first vertex**

**//Not in any other point**

if ( graph[ path[pos-1] ][ path[0] ] == 1 )

return true;

else

return false;

}

**// Try different vertices as a next candidate in Hamiltonian Cycle.**

**// We don't try for 0 as we included 0 as starting point in in hamCycle()**

**//Vertex 0 is added in the path.**

for (int v = 1; v < V; v++)

{

**/\* Check if this vertex can be added to Hamiltonian Cycle \*/**

if (isSafe(v, graph, path, pos))

{

path[pos] = v;

**//It is added to to the current position of of path**

**//Again, we do not go for dfs or bfs**  
 **//At every step, we try to add a vertex which is not added before that is the way**

**/\* recur to construct rest of the path \*/**

if (hamCycleUtil (graph, path, pos+1) == true)

return true;

**/\* If adding vertex v doesn't lead to a solution,**

**then remove it \*/**

path[pos] = -1;

}

}

**/\* If no vertex can be added to Hamiltonian Cycle constructed so far,**

**then return false \*/**

return false;

}

**/\* This function solves the Hamiltonian Cycle problem using Backtracking.**

**It mainly uses hamCycleUtil() to solve the problem. It returns false**

**if there is no Hamiltonian Cycle possible, otherwise return true and**

**prints the path. Please note that there may be more than one solutions,**

**this function prints one of the feasible solutions. \*/**

bool hamCycle(bool graph[V][V])

{

int \*path = new int[V];

for (int i = 0; i < V; i++)

path[i] = -1;

**/\* Let us put vertex 0 as the first vertex in the path. If there is**

**a Hamiltonian Cycle, then the path can be started from any point**

**of the cycle as the graph is undirected \*/**

path[0] = 0;

if ( hamCycleUtil(graph, path, 1) == false )

{

printf("\nSolution does not exist");

return false;

}

printSolution(path);

return true;

}

**/\* A utility function to print solution \*/**

void printSolution(int path[])

{

printf ("Solution Exists:"

" Following is one Hamiltonian Cycle \n");

for (int i = 0; i < V; i++)

printf(" %d ", path[i]);

**// Let us print the first vertex again to show the complete cycle**

printf(" %d ", path[0]);

printf("\n");

}

**// driver program to test above function**

int main()

{

**/\* Let us create the following graph**

**(0)-----(1)--(2)**

**| / \ |**

**| / \ |**

**| / \ |**

**(3)------------(4) \*/**

bool graph1[V][V] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0},

};

**// Print the solution**

hamCycle(graph1);

**/\* Let us create the following graph**

**(0)----(1)-----(2)**

**| / \ |**

**| / \ |**

**| / \ |**

**(3) (4) \*/**

bool graph2[V][V] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 0},

{0, 1, 1, 0, 0},

};

**// Print the solution**

hamCycle(graph2);

return 0;

}