**Experiment-10**

**N-Queen Problems**

#include<stdio.h>

#include<math.h>

int board[20],count;

int main()

{

int n,i,j;

void queen(int row,int n);

printf(" - N Queens Problem Using Backtracking -");

printf("\n\nEnter number of Queens:");

scanf("%d",&n);

queen(1,n);

return 0;

}

//function for printing the solution

void print(int n)

{

int i,j;

printf("\n\nSolution %d:\n\n",++count);

for(i=1;i<=n;++i)

printf("\t%d",i);

for(i=1;i<=n;++i)

{

printf("\n\n%d",i);

for(j=1;j<=n;++j) //for nxn board

{

if(board[i]==j)

printf("\tQ"); //queen at i,j position

else

printf("\t-"); //empty slot

}

}

}

/\*funtion to check conflicts

If no conflict for desired postion returns 1 otherwise returns 0\*/

int place(int row,int column)

{

int i;

for(i=1;i<=row-1;++i)

{

//checking column and digonal conflicts

if(board[i]==column)

return 0;

else

if(abs(board[i]-column)==abs(i-row))

return 0;

}

return 1; //no conflicts

}

//function to check for proper positioning of queen

void queen(int row,int n)

{

int column;

for(column=1;column<=n;++column)

{

if(place(row,column))

{

board[row]=column; //no conflicts so place queen

if(row==n) //dead end

print(n); //printing the board configuration

else //try queen with next position

queen(row+1,n);

}

}

}

Output:



**Flyod Warshal**

#include<stdio.h>

#include<conio.h>

int min(int,int);

void floyds(int p[10][10],int n) {

int i,j,k;

for (k=1;k<=n;k++)

for (i=1;i<=n;i++)

for (j=1;j<=n;j++)

if(i==j)

p[i][j]=0; else

p[i][j]=min(p[i][j],p[i][k]+p[k][j]);

}

int min(int a,int b) {

if(a<b)

return(a); else

return(b);

}

void main() {

int p[10][10],w,n,e,u,v,i,j;

;

clrscr();

printf("\n Enter the number of vertices:");

scanf("%d",&n);

printf("\n Enter the number of edges:\n");

scanf("%d",&e);

for (i=1;i<=n;i++) {

for (j=1;j<=n;j++)

p[i][j]=999;

}

for (i=1;i<=e;i++) {

printf("\n Enter the end vertices of edge%d with its weight \n",i);

scanf("%d%d%d",&u,&v,&w);

p[u][v]=w;

}

printf("\n Matrix of input data:\n");

for (i=1;i<=n;i++) {

for (j=1;j<=n;j++)

printf("%d \t",p[i][j]);

printf("\n");

}

floyds(p,n);

printf("\n Transitive closure:\n");

for (i=1;i<=n;i++) {

for (j=1;j<=n;j++)

printf("%d \t",p[i][j]);

printf("\n");

}

printf("\n The shortest paths are:\n");

for (i=1;i<=n;i++)

for (j=1;j<=n;j++) {

if(i!=j)

printf("\n <%d,%d>=%d",i,j,p[i][j]);

}

getch();

}

**Topological sorting**

#include<stdio.h>

#define MAX 200

int n,adj[MAX][MAX];

int front = -1,rear = -1,queue[MAX];

void main() {

int i,j = 0,k;

int topsort[MAX],indeg[MAX];

create\_graph();

printf(“The adjacency matrix is:\n”);

display();

for (i=1;i<+n;i++) {

indeg[i]=indegree(i);

if(indeg[i]==0)

insert\_queue(i);

}

while(front<=rear) {

k=delete\_queue();

topsort[j++]=k;

for (i=1;i<=n;i++) {

if(adj[k][i]==1) {

adj[k][i]=0;

indeg[i]=indeg[i]-1;

if(indeg[i]==0)

insert\_queue(i);

}

}

}

printf("Nodes after topological sorting are:\n");

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

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

printf("\n");

}

create\_graph() {

int i,max\_edges,origin,destin;

printf("\n Enter number of vertices:");

scamf("%d",&n);

max\_edges = n \* (n - 1);

for (i = 1;i <= max\_edges;i++) {

printf("\n Enter edge %d (00 to quit):",i);

scanf("%d%d",&origin,&destin);

if((origin == 0) && (destin == 0)) {

printf("Invalid edge!!\n");

i–;

} else

adj[origin][destin] = 1;

}

return;

}

display() {

int i,j;

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

for (j = 1;jrear) {

printf(“Queue Underflow”);

return;

} else {

del\_item = queue[front];

front = front + 1;

return del\_item;

}

}

int indegree(int node) {

int i,in\_deg = 0;

for (i = 1;i <= n;i++)

if(adj[i][node] == 1)

in\_deg++;

returnin\_deg;

}

**Transitive closure of a graph**

#include <iostream>

#include <vector>

#include <cstring>

#include <iomanip>

using namespace std;

// Number of vertices in the graph

#define N 4

// data structure to store graph edges

struct Edge {

int src, dest;

};

// class to represent a graph object

class Graph

{

public:

// An array of vectors to represent adjacency list

vector<int> adjList[N];

// Constructor

Graph(vector<Edge> edges)

{

// add edges to the directed graph

for (Edge edge: edges)

{

int src = edge.src;

int dest = edge.dest;

adjList[src].push\_back(dest);

}

}

};

// C is connectivity matrix and stores transitive closure of graph

// root is the topmost node in DFS tree(it is starting vertex of DFS)

// descendent is current vertex to be explored in DFS

// Invariant: A path already exists from root -> descendent in graph

void DFS(Graph const& graph, bool C[N][N], int root, int descendent)

{

for (int child : graph.adjList[descendent])

{

// if child is an adjacent vertex of descendent, we have

// found a path from root->child

if (!C[root][child])

{

C[root][child] = true;

DFS(graph, C, root, child);

}

}

}

int main()

{

// array of graph edges as per above diagram

vector<Edge> edges = {

{ 0, 2 }, { 1, 0 }, { 3, 1 }

};

// create a graph from edges

Graph graph(edges);

// C is connectivity matrix and stores the transitive closure

// of the graph. The value of C[i][j] is 1 only if a directed

// path exists from vertex i to vertex j.

bool C[N][N];

memset(C, false, sizeof C);

// consider each vertex and start DFS from it

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

{

C[v][v] = true;

DFS(graph, C, v, v);

// print path info for vertex v

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

cout << left << setw(4) << C[v][u];

cout << endl;

}

return 0;

}

OUTPUT:

