**DFS**

#include <stdio.h>

#include <stdlib.h>

int s,v,e,time,visited[20],G[20][20];

void DFS(int i)

{

int j;

visited[i]=1;

printf("%d\t",i+1);

for(j=0;j<v;j++)

{

if(G[i][j]==1 && visited[j]==0)

DFS(j);

}

}

int main()

{

int i,j,v1,v2;

printf("\t\t\tGraphs\n");

printf("Enter the no of Edges:");

scanf("%d",&e);

printf("Enter the no of Vertices:");

scanf("%d",&v);

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

{

for(j=0;j<v;j++)

G[i][j]=0;

}

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

{

printf("Enter the edges (v1 v2): ");

scanf("%d %d",&v1,&v2);

G[v1-1][v2-1]=1;

}

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

{

for(j=0;j<v;j++)

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

printf("\n");

}

printf("Enter the start position: ");

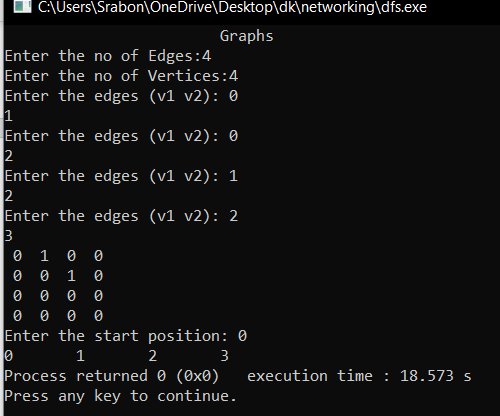
scanf("%d",&s);

DFS(s-1);

return 0;

}

**OUTPUT:**



**BFS**

#include <stdio.h>

#include <stdlib.h>

#define SIZE 40

struct queue {

int items[SIZE];

int front;

int rear;

};

struct queue\* createQueue();

void enqueue(struct queue\* q, int);

int dequeue(struct queue\* q);

void display(struct queue\* q);

int isEmpty(struct queue\* q);

void printQueue(struct queue\* q);

struct node {

int vertex;

struct node\* next;

};

struct node\* createNode(int);

struct Graph {

int numVertices;

struct node\*\* adjLists;

int\* visited;

};

// BFS algorithm

void bfs(struct Graph\* graph, int startVertex) {

struct queue\* q = createQueue();

graph->visited[startVertex] = 1;

enqueue(q, startVertex);

while (!isEmpty(q)) {

printQueue(q);

int currentVertex = dequeue(q);

printf("Visited %d\n", currentVertex);

struct node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->vertex;

if (graph->visited[adjVertex] == 0) {

graph->visited[adjVertex] = 1;

enqueue(q, adjVertex);

}

temp = temp->next;

}

}

}

// Creating a node

struct node\* createNode(int v) {

struct node\* newNode = malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Creating a graph

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

graph->visited = malloc(vertices \* sizeof(int));

int i;

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

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

// Add edge

void addEdge(struct Graph\* graph, int src, int dest) {

// Add edge from src to dest

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

// Add edge from dest to src

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

// Create a queue

struct queue\* createQueue() {

struct queue\* q = malloc(sizeof(struct queue));

q->front = -1;

q->rear = -1;

return q;

}

// Check if the queue is empty

int isEmpty(struct queue\* q) {

if (q->rear == -1)

return 1;

else

return 0;

}

// Adding elements into queue

void enqueue(struct queue\* q, int value) {

if (q->rear == SIZE - 1)

printf("\nQueue is Full!!");

else {

if (q->front == -1)

q->front = 0;

q->rear++;

q->items[q->rear] = value;

}

}

// Removing elements from queue

int dequeue(struct queue\* q) {

int item;

if (isEmpty(q)) {

printf("Queue is empty");

item = -1;

} else {

item = q->items[q->front];

q->front++;

if (q->front > q->rear) {

printf("Resetting queue ");

q->front = q->rear = -1;

}

}

return item;

}

// Print the queue

void printQueue(struct queue\* q) {

int i = q->front;

if (isEmpty(q)) {

printf("Queue is empty");

} else {

printf("\nQueue contains \n");

for (i = q->front; i < q->rear + 1; i++) {

printf("%d ", q->items[i]);

}

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 1, 4);

addEdge(graph, 1, 3);

addEdge(graph, 2, 4);

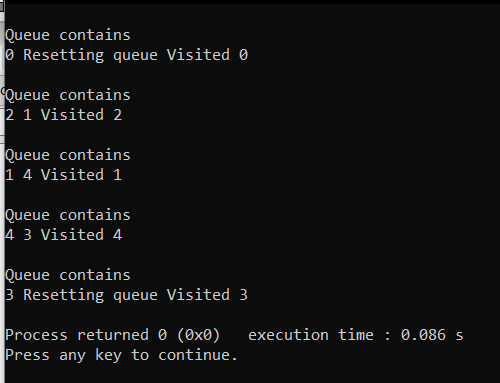
addEdge(graph, 3, 4);

bfs(graph, 0);

return 0;

}

**OUTPUT:**



**N-QUEEN**

#include<stdio.h>

//#include<conio.h>

#include<math.h>

int a[30],count=0;

int place(int pos)

{

int i;

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

{

if((a[i]==a[pos])||((abs(a[i]-a[pos])==abs(i-pos))))

return 0;

}

return 1;

}

void print\_sol(int n)

{

int i,j;

count++;

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

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

{

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

{

if(a[i]==j)

printf("Q\t");

else

printf("\*\t");

}

printf("\n");

}

}

void queen(int n)

{

int k=1;

a[k]=0;

while(k!=0)

{

a[k]=a[k]+1;

while((a[k]<=n)&&!place(k))

a[k]++;

if(a[k]<=n)

{

if(k==n)

print\_sol(n);

else

{

k++;

a[k]=0;

}

}

else

k--;

}

}

void main()

{

int i,n;

// clrscr();

printf("Enter the number of Queens\n");

scanf("%d",&n);

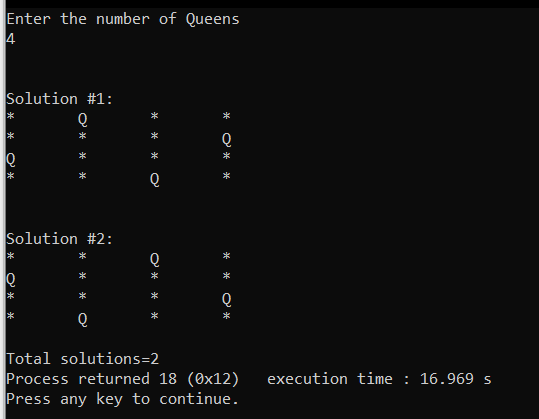
queen(n);

printf("\nTotal solutions=%d",count);

// getch();

}

OUTPUT:



**HILL CLIMBING**

import random

def randomSolution(tsp):

cities = list(range(len(tsp)))

solution = []

for i in range(len(tsp)):

randomCity = cities[random.randint(0, len(cities) - 1)]

solution.append(randomCity)

cities.remove(randomCity)

return solution

def routeLength(tsp, solution):

routeLength = 0

for i in range(len(solution)):

routeLength += tsp[solution[i - 1]][solution[i]]

return routeLength

def getNeighbours(solution):

neighbours = []

for i in range(len(solution)):

for j in range(i + 1, len(solution)):

neighbour = solution.copy()

neighbour[i] = solution[j]

neighbour[j] = solution[i]

neighbours.append(neighbour)

return neighbours

def getBestNeighbour(tsp, neighbours):

bestRouteLength = routeLength(tsp, neighbours[0])

bestNeighbour = neighbours[0]

for neighbour in neighbours:

currentRouteLength = routeLength(tsp, neighbour)

if currentRouteLength < bestRouteLength:

bestRouteLength = currentRouteLength

bestNeighbour = neighbour

return bestNeighbour, bestRouteLength

def hillClimbing(tsp):

currentSolution = randomSolution(tsp)

currentRouteLength = routeLength(tsp, currentSolution)

neighbours = getNeighbours(currentSolution)

bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

while bestNeighbourRouteLength < currentRouteLength:

currentSolution = bestNeighbour

currentRouteLength = bestNeighbourRouteLength

neighbours = getNeighbours(currentSolution)

bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

return currentSolution, currentRouteLength

def main():

tsp = [

[0, 400, 500, 300],

[400, 0, 300, 500],

[500, 300, 0, 400],

[300, 500, 400, 0]

]

print(hillClimbing(tsp))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT:**

