**Lab File**

**Analysis and Design of Algorithms**

**(CSE 303)**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

A blue and yellow logo

Description automatically generated with low confidence

Submitted to: Submitted by:

Dr Dolly Sharma Shaina Mehta

Associate Professor A2305219268

CSE Department, ASET B.tech. C.S.E.

5CSE-4Y

Amity School Of Engineering and technology

Amity University Uttar Pradesh

Noida -201301

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Exp No | Assignment  Category | Code | Name of Experiment | Date of Allotment | Date of Evaluation | Max  Marks | Marks  Obtained | Faculty  Sign |
| 1 | Mandatory  Experiment |  | To implement Bubble Sort and analyse its complexity. | 21-07-2021 | 24-10-2021 |  |  |  |
| 2 |  | To implement Insertion Sort and analyse its complexity. | 28-07-2021 | 24-10-2021 |  |  |  |
| 3 |  | To implement Recursive Binary search and determine the time taken to search an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n. | 4-08-2021 | 24-10-2021 |  |  |  |
| 4 |  | To sort a given set of elements using Quick Sort method and determine the time taken to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. | 07-09-2021 | 24-10-2021 |  |  |  |
| 5 |  | Implement Knapsack Problem using Greedy Approach. | 21-09-2021 | 24-10-2021 |  |  |  |
| 6 |  | To Implement 0/1 Knapsack Problem using Dynamic Programming method. | 28-09-2021 | 24-10-2021 |  |  |  |
| 7 |  | To implement BFS traversal on a graph. | 05-10-2021 | 24-10-2021 |  |  |  |
| 8 |  |  | To implement DFS traversal on a graph. | 12-10-2021 | 24-10-2021 |  |  |  |
| 9 |  |  | To implement Dijkstra’s Algorithm. | 19-10-2021 | 24-10-2021 |  |  |  |
| 10 |  |  | To implement N – Queen Problem using Backtracking Approach. | 26-10-2021 | 9-11-2021 |  |  |  |
| 11 |  |  | To implement Knapsack Problem based on Backtracking algorithm. | 26-10-2021 | 9-11-2021 |  |  |  |
| 12 |  |  | To implement Traveling Salesman problem based on Branch and Bound technique. | 3-10-2021 | 9-11-2021 |  |  |  |
|  | Viva | Viva |  |  |  |  |  |  |

**Experiment 1**

**Aim:** To implement Bubble Sort and analyse its complexity.

**Code:**

1. **Worst Case**

#include <iostream>

using namespace std;

int pass=0, comp=0, swaps=0;

void Bubble\_Sort(int \*arr, int n){

for(int i=0;i<n-1;i++){

pass++;

for(int j=0;j<n-1-i;j++){

comp++;

if(arr[j]>arr[j+1]){

swaps++;

int temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

cout<<"The array after pass "<<i+1<<" is: ";

for(int k=0;k<n;k++){

cout<<arr[k]<<" ";

}

cout<<endl;

}

}

int main(){

int n;

cout<<"Bubble Sort Algorithm"<<endl;

cout<<"Best Case Time Complexity is: O(n)"<<endl;

cout<<"Worst Case Time Complexity is: O(n^2)"<<endl;

cout<<"Space Complexity is: O(n)"<<endl;

cout<<"Enter the size of an array (Total entries should not be more than 100): ";

cin>>n;

int \*arr = new int [100];

cout<<"Enter the array elements: ";

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

cin>>arr[i];

}

cout<<"The original array is: ";

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

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

}

cout<<endl;

Bubble\_Sort(arr,n);

cout<<"The sorted array in ascending order is: ";

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

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

}

cout<<endl;

cout<<"No of passes is: "<<pass<<endl;

cout<<"No of comparisons is: "<<comp<<endl;

cout<<"No of swaps is: "<<swaps<<endl;

delete arr;

}

1. **Best Case**

#include <iostream>

using namespace std;

int pass=0, comp=0, swaps=0;

void Bubble\_Sort(int \*arr, int n){

for(int i=0;i<n-1;i++){

pass++;

cout<<"The array after pass "<<i+1<<" is: ";

for(int k=0;k<n;k++){

cout<<arr[k]<<" ";

}

cout<<endl;

}

}

int main(){

int n;

cout<<"Bubble Sort Algorithm"<<endl;

cout<<"Best Case Time Complexity is: O(n)"<<endl;

cout<<"Worst Case Time Complexity is: O(n^2)"<<endl;

cout<<"Space Complexity is: O(n)"<<endl;

cout<<"Enter the size of an array (Total entries should not be more than 100): ";

cin>>n;

int \*arr = new int [100];

cout<<"Enter the array elements: ";

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

cin>>arr[i];

}

cout<<"The original array is: ";

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

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

}

cout<<endl;

Bubble\_Sort(arr,n);

cout<<"The sorted array in ascending order is: ";

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

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

}

cout<<endl;

cout<<"No of passes is: "<<pass<<endl;

cout<<"No of comparisons is: "<<comp<<endl;

cout<<"No of swaps is: "<<swaps<<endl;

delete arr;

}

**Output:**

1. **Best Case**

Text

Description automatically generated

1. **Worst Case**

Text

Description automatically generated

**Conclusion:** The bubble sort sorting algorithm has been implemented successfully.

**Experiment 2**

**Aim:** To implement Insertion Sort and analyse its complexity.

**Code:**

#include <iostream>

#include <ctime>

using namespace std;

int pass=0,comp=0,swaps=0;

void Insertion\_Sort(int \*a, int n){

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

pass++;

int temp = a[i];

int j = i-1;

while(j>=0){

comp++;

if(temp<a[j]){

a[j+1]=a[j];

swaps++;

}

else{

break;

}

j--;

}

a[j+1]=temp;

}

}

int main(){

int n;

clock\_t s,f;

cout<<"Insertion Sort Algorithm"<<endl;

cout<<"Best Case Time Complexity is: O(n)"<<endl;

cout<<"Worst Case Time Complexity is: O(n^2)"<<endl;

cout<<"Space Complexity is: O(n)"<<endl;

cout<<"Enter the size of an array: ";

cin>>n;

int \*arr = new int [100];

cout<<"Enter the array elements: ";

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

cin>>arr[i];//=rand()%100;

}

cout<<"The original array is: ";

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

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

}

cout<<endl;

s=clock();

//cout<<s;

Insertion\_Sort(arr,n);

f=clock();

//cout<<" "<<f<<endl;

cout<<"The sorted array in ascending order is: ";

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

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

}

cout<<endl;

cout<<"No of passes is: "<<pass<<endl;

cout<<"No of comparisons is: "<<comp<<endl;

cout<<"No of swaps is: "<<swaps<<endl;

//cout<<"The time required for insertion sort is: "<< ((double)(f-s))/CLOCKS\_PER\_SEC <<" seconds."<<endl;

delete arr;

}

**Output:**

1. **Best Case:**

**Text

Description automatically generated**

1. **Worst Case:**

**Text

Description automatically generated**

**Analysis:**

|  |  |
| --- | --- |
| Entries | Time Taken (seconds) |
| 10 | 0 |
| 50 | 0 |
| 100 | 0 |
| 500 | 0 |
| 1000 | 0 |
| 5000 | 0.015 |
| 10000 | 0.1 |
| 50000 | 2.21 |
| 100000 | 8.134 |
| 500000 | 233.036 |

**Chart, line chart

Description automatically generated**

**Conclusion:** The insertion sort sorting algorithm has been implemented and analysed successfully.

**Experiment 3**

**Aim:** To implement Recursive Binary search and determine the time taken to search an element. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.

**Code:**

#include <iostream>

#include <algorithm>

#include <chrono>

using namespace std;

using namespace std::chrono;

int helper(int a[],int si,int ei,int item){

int ind = -1;

if(si > ei){

return ind;

}

int mid = (si+ei)/2;

if(a[mid] == item){

return mid;

}

else if(a[mid] > item){

ind = helper(a,si,mid-1,item);

}

else{

ind = helper(a,mid+1,ei,item);

}

return ind;

}

int recursiveBinarySearch(int a[],int n,int item){

int si = 0;

int ei = n-1;

int ind = helper(a,si,ei,item);

return ind;

}

int main(){

int n;

cout<<"Recursive Binary Search Algorithm"<<endl;

cout<<"Worst Case Time Complexity is O(lgn)"<<endl;

cout<<"Best Case Time Complexity is O(1)"<<endl;

cout<<"Space complexity is O(1)"<<endl;

cout<<"Enter the array limit: ";

cin>>n;

cout<<"Enter the array elements: ";

int \*a = new int[n];

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

cin>>a[i];

}

int item;

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

cout<<a[i]<<" ";

}\*/

cout<<"The sorted array is: ";

sort(a,a+n);

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

cout<<a[i]<<" ";

}

cout<<endl;

//int m = n/2;

cout<<"Enter the item to be searched: ";

cin>>item;

//cout<<item<<endl;

auto s = high\_resolution\_clock::now();

int index = recursiveBinarySearch(a,n,item);

auto f = high\_resolution\_clock::now();

if(index != -1){

cout<<"The element is found in the array at: "<<index<<".";

}

else{

cout<<"The element is not found in the array";

}

auto duration = duration\_cast<nanoseconds>(f-s);

//cout<<"The time required : "<< duration.count()<<" nanosecondsseconds."<<endl;

delete a;

}

**Output:**

**Text

Description automatically generated**

**Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| N | Best Case | Worst Case | Random Case |
| 10 | 346 | 478 | 494 |
| 50 | 387 | 607 | 442 |
| 100 | 360 | 512 | 557 |
| 500 | 318 | 596 | 449 |
| 1000 | 200 | 643 | 570 |
| 5000 | 307 | 605 | 549 |
| 10000 | 257 | 614 | 332 |
| 50000 | 400 | 408 | 595 |
| 100000 | 229 | 688 | 1545 |
| 500000 | 257 | 462 | 777 |

**Chart, line chart

Description automatically generated**

**Results and Conclusion:** Implementation and analysis of Recursive Binary Search has been done successfully.

**Experiment 4**

**Aim:** To sort a given set of elements using Quick Sort method and determine the time taken to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

**Code:**

#include <iostream>

using namespace std;

int Pivot(int \*arr, int si, int ei){

int x = arr[si];

int c = 0;

for(int i=si+1; i<=ei; i++){

if(x>=arr[i]){

c++;

}

}

int p = si+c;

int temp = arr[si];

arr[si] = arr[p];

arr[p] = temp;

int i = si, j = ei;

while(i<p && j>p){

if(arr[i] <= x){

i++;

}

else if(arr[j] > x){

j--;

}

else{

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

i++;

j--;

}

}

return p;

}

void Quick(int \*arr, int si, int ei){

if(si >= ei){

return;

}

int c = Pivot(arr,si,ei);

Quick(arr,si,c-1);

Quick(arr,c+1,ei);

}

void Quick\_Sort(int \* arr, int n){

int si = 0, ei = n-1;

Quick(arr,si,ei);

}

int main(){

int n;

cout<<"Quick Sort Algorithm"<<endl;

cout<<"Best Case Time Complexity is: O(n\*logn)"<<endl;

cout<<"Worst Case Time Complexity is: O(n^2)"<<endl;

cout<<"Space Complexity is: O(n)"<<endl;

cout<<"Enter the size of an array (Total entries should not be more than 100): ";

cin>>n;

int \*arr = new int [100];

cout<<"Enter the array elements: ";

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

cin>>arr[i];

}

cout<<"The original array is: ";

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

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

}

cout<<endl;

Quick\_Sort(arr,n);

cout<<"The sorted array in ascending order is: ";

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

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

}

cout<<endl;

delete arr;

}

**Output:**

1. **Best Case:**

A picture containing text, screenshot, computer, monitor

Description automatically generated

1. **Worst Case:**

A picture containing text, screenshot, computer, monitor

Description automatically generated

­­­­­­­

**Analysis:**

|  |  |
| --- | --- |
| Entries | Quick Sort |
| 10 | 0.001 |
| 50 | 0.001 |
| 100 | 0.001 |
| 500 | 0.001 |
| 1000 | 0.001 |
| 5000 | 0.001 |
| 10000 | 0.003 |
| 50000 | 0.036 |
| 100000 | 0.112 |

Chart, line chart

Description automatically generated

**Conclusion:** The quick sort sorting algorithm has been implemented and analysed successfully.

**Experiment 5**

**Aim:** Implement Knapsack Problem using Greedy Approach.

**Code:**

#include <iostream>

using namespace std;

float FractionalKnapsack(float profit[],float weight[],float fraction[],float selObj[],float maxWeight,int obj){

int i;

float totalProfit = 0;

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

if(weight[i]<maxWeight){

totalProfit += profit[i];

maxWeight -= weight[i];

selObj[i] = 1;

}

else{

break;

}

}

if(i<obj){

float fract = maxWeight/weight[i];

totalProfit += (profit[i]\*fract);

maxWeight = 0;

selObj[i] = fract;

}

return totalProfit;

}

int main(){

int obj;

float maxWeight, totalProfit;

cout<<"Enter the no of objects to be put into the knapsack: ";

cin>>obj;

float \*profit = new float[obj];

float \*weight = new float[obj];

float \*fraction = new float[obj];

cout<<"Enter the weight associated with the objects: ";

for(int i=0;i<obj;i++){

cin>>weight[i];

}

cout<<"Enter the profit associated with the objects: ";

for(int i=0;i<obj;i++){

cin>>profit[i];

}

cout<<"Enter the maximum capacity of the knapsack: ";

cin>>maxWeight;

for(int i=0;i<obj;i++){

fraction[i] = profit[i]/weight[i];

}

float \*selObj = new float[obj];

for(int i=0;i<obj;i++){

selObj[i] = 0;

}

for(int i=0;i<obj-1;i++){

for(int j=0;j<obj-1-i;j++){

if(fraction[j]<fraction[j+1]){

float t1 = fraction[j];

fraction[j] = fraction[j+1];

fraction[j+1] = t1;

float t2 = weight[j];

weight[j] = weight[j+1];

weight[j+1] = t2;

float t3 = profit[j];

profit[j] = profit[j+1];

profit[j+1] = t3;

}

}

}

totalProfit = FractionalKnapsack(profit,weight,fraction,selObj,maxWeight,obj);

cout<<"Total profit in the knapsack is: "<<totalProfit<<endl;

cout<<"The weight of the objects: ";

for(int i=0;i<obj;i++){

cout<<weight[i]<<" ";

}

cout<<endl;

cout<<"The profit associated with the object: ";

for(int i=0;i<obj;i++){

cout<<profit[i]<<" ";

}

cout<<endl;

cout<<"The objects selected for knapsack is: ";

for(int i=0;i<obj;i++){

cout<<selObj[i]<<" ";

}

delete profit;

delete weight;

delete fraction;

delete selObj;

}

**Output:**

**Text

Description automatically generated**

**Conclusion:** Fractional Knapsack problem has been implemented successfully.

**Experiment 6**

**Aim:** To Implement 0/1 Knapsack Problem using Dynamic Programming method.

**Code:**

//To implement 0/1 Knapsack problem

#include<iostream>

#include<algorithm>

using namespace std;

class Items{

public:

int weight, profit;

Items(){

this->weight = 0;

this->profit = 0;

}

};

bool cmp(Items a, Items b){

return a.weight<b.weight;

}

int Knapsack(Items I[],int maxWeight,int obj){

//int totalProfit = 0;

int m = obj, n = maxWeight;

int \*\*ans = new int \*[m+1];

for(int i=0;i<=m;i++){

ans[i] = new int [n+1];

}

for(int i=0;i<=m;i++){

ans[i][0] = 0;

}

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

ans[0][j] = 0;

}

for(int i=0;i<=m;i++){

for(int j=0;j<=n;j++){

if (i==0 || j==0){

ans[i][j] = 0;

}

else if (I[i-1].weight<=j){

ans[i][j] = max(I[i-1].profit+ans[i-1][j-I[i-1].weight],ans[i-1][j]);

}

else{

ans[i][j] = ans[i-1][j];

}

}

}

return ans[m][n];

}

int main(){

int obj;

int maxWeight, totalProfit;

cout<<"Enter the no of objects to be put into the knapsack: ";

cin>>obj;

//int \*profit = new int[obj];

//int \*weight = new int[obj];

Items \*I= new Items [obj];

cout<<"Enter the weight associated with the objects: ";

for(int i=0;i<obj;i++){

cin>>I[i].weight;

}

cout<<"Enter the profit associated with the objects: ";

for(int i=0;i<obj;i++){

cin>>I[i].profit;

}

//int \*buf = new int buf[obj];

cout<<"Enter the maximum capacity of the knapsack: ";

cin>>maxWeight;

//int \*selObj = new int[obj];

//for(int i=0;i<obj;i++){

// selObj[i] = 0;

//}

sort(I,I+obj,cmp);

totalProfit = Knapsack(I,maxWeight,obj);

cout<<"Total profit in the knapsack is: "<<totalProfit<<endl;

//cout<<"The objects selected for knapsack is: ";

//for(int i=0;i<obj;i++){

// cout<<selObj[i]<<" ";

//}

delete I;

//delete profit;

//delete weight;

//delete selObj;

}

**Output:**

**Text

Description automatically generated**

**Conclusion:** 0/1 Knapsack problem has been implemented successfully.

**Experiment 7**

**Aim:** To implement BFS traversal on a graph.

**Code:**

//To implement BFS Traversal of a graph G

#include <iostream>

#include <list>

using namespace std;

class Graph

{

int numV;

list <int> \*adjL;

public:

Graph(int numV); // Constructor

// function to add an edge to graph

void addEdge(int u, int v);

// prints BFS traversal from a given source s

void BFS(int startV);

};

Graph::Graph(int numV) {

this->numV = numV;

adjL = new list<int>[numV];

}

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

adjL[u].push\_back(v);

}

void Graph::BFS(int startV) {

//Visited array is used to track whether the node is visited or not

bool \*visited = new bool [numV];

for(int i=0;i<numV;i++) {

visited[i] = false;

}

list <int> q;

//Pushing the starting vertex into the queue

visited[startV] = true;

q.push\_back(startV);

// Get all adjacent vertices of the dequeued

// vertex s. If a adjacent has not been visited,

// then mark it visited and enqueue it

list<int>::iterator i;

while(!q.empty()) {

startV = q.front();

cout<<startV<<" ";

q.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=adjL[startV].begin();i!=adjL[startV].end();++i) {

if(!visited[\*i]){

visited[\*i] = true;

q.push\_back(\*i);

}

}

}

delete visited;

}

int main()

{

int n;

cout<<"Enter the number of vertices of the graph: ";

cin>>n;

Graph g = Graph(n);

cout<<"Enter the edges of the graph: "<<endl;

//int c = 1;

while(true)

{

int u,v;

cout<<"Enter the source vertex: ";

cin>>u;

cout<<"Enter the destination vertex: ";

cin>>v;

g.addEdge(u,v);

cout<<"Do you want to add new edge (1-yes/0-no)?";

int c;

cin>>c;

if(c==0){

break;

}

}

int sVertex;

cout<<"Enter the starting vertex: ";

cin>>sVertex;

cout<<"BFS Traversal of a Graph G is: ";

g.BFS(sVertex);

return 0;

}

**Output:**

**Text

Description automatically generated**

**Conclusion:** BFS traversal has been implemented successfully.

**Experiment 8**

**Aim:** To implement DFS traversal on a graph.

**Code:**

#include <iostream>

#include <list>

#include <map>

using namespace std;

class Graph

{

map <int,bool> visited;

map <int,list <int>> adjL;

public:

// function to add an edge to graph

void addEdge(int u, int v);

// prints DFS traversal from a given source s

void DFS(int startV);

};

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

adjL[u].push\_back(v);

}

void Graph::DFS(int startV) {

//Marking the vertex as visited

visited[startV] = true;

//Printing of the vertex

cout<<startV<<" ";

// Recur for all the vertices adjacent

// to this vertex

list <int>::iterator i;

for(i=adjL[startV].begin();i!=adjL[startV].end();++i) {

if(!visited[\*i]){

DFS(\*i);

}

}

}

int main()

{

Graph g;

cout<<"Enter the edges of the graph: "<<endl;

//int c = 1;

while(true)

{

int u,v;

cout<<"Enter the source vertex: ";

cin>>u;

cout<<"Enter the destination vertex: ";

cin>>v;

g.addEdge(u,v);

cout<<"Do you want to add new edge (1-yes/0-no)?";

int c;

cin>>c;

if(c==0){

break;

}

}

int sVertex;

cout<<"Enter the starting vertex: ";

cin>>sVertex;

cout<<"DFS Traversal of a Graph G is: ";

g.DFS(sVertex);

return 0;

}

**Output:**

**Text

Description automatically generated**

**Conclusion:** DFS traversal has been implemented successfully.

**Experiment 9**

**Aim:** To implement Dijkstra’s Algorithm.

**Code:**

#include<iostream>

#include<climits>

using namespace std;

// A utility function to find the vertex with minimum distance value, from the set of vertices not yet included in shortest path tree

int minDistance(int \*dist, bool \*sptSet, int n){

//Initialize the minimum value

int minVal = INT\_MAX, minIndex;

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

if(sptSet[i] == false && dist[i] <= minVal){

minVal = dist[i];

minIndex = i;

}

}

return minIndex;

}

// A utility function to print the constructed distance array

void printSolution(int \*dist, int n){

cout<<"Vertex \t Distance from Source: "<<endl;

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

cout << i << " \t\t"<<dist[i]<< endl;

}

}

void dijkstras(int \*\*adjM,int n,int source){

// The output array. dist[i] will hold the shortest distance from source vertex to ith vertex

int \*dist = new int [n];

// path tree or shortest distance from source vertex to ith vertex is finalized

//sptSet[i] will be true if vertex i is included in shortest

bool \*sptSet = new bool[n];

//Initialize all the distance to INFINITE and stpSet[i] to false

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

dist[i] = INT\_MAX;

sptSet[i] = false;

}

//initializing the source vertex as 0

dist[source] = 0;

//finding the shortest path of all the vertices

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

// Pick the minimum distance vertex from the set of vertices not yet processed. u is always equal to source vertex in the first iteration.

int u = minDistance(dist, sptSet, n);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.

for(int v=0;v<n;v++){

//Update dist[v] only if it is not in the sptSet, there is an edge from u to v, and total weight of path from src to v through u is smaller than current value of dist[v]

if(!sptSet[v] && adjM[u][v] && dist[u] != INT\_MAX && dist[u]+adjM[u][v] < dist[v]){

dist[v] = dist[u] + adjM[u][v];

}

}

}

// print the constructed distance array

printSolution(dist,n);

}

int main(){

//No of Vertices

int n;

cout<<"Enter the number of vertices present in the graph: ";

cin>>n;

//Adjacency Matrix

int \*\*adjM = new int \*[n];

cout<<"Enter the graph: "<<endl;

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

adjM[i] = new int[n];

for(int j=0;j<n;j++){

cin>>adjM[i][j];

}

}

int source;

cout<<"Enter the source vertex of the graph: ";

cin>>source;

dijkstras(adjM,n,source);

return 0;

}

**Output:**

**Text

Description automatically generated**

**Conclusion:** Dijkstra’s Algorithm has been implemented successfully.

**Experiment 10**

**Aim:** To implement N – Queen Problem using Backtracking Approach.

**Code:**

#include <iostream>

using namespace std;

#define N 8

void output(int board[N][N]){

cout<<"Number of queens = "<<N<<endl;

cout<<"One of the possible solutions is"<<endl;

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

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

if(board[i][j] == 1)

cout<<'Q'<<" ";

else

cout<<"\_ ";

cout<<endl;

}

}

bool check(int board[N][N], int row, int col){

int i, j;

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

if (board[row][i])

return false;

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

if (board[i][j])

return false;

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

if (board[i][j])

return false;

return true;

}

bool nQueenAlgo(int board[N][N], int col){

if (col >= N)

return true;

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

if (check(board, i, col)) {

board[i][col] = 1;

if (nQueenAlgo(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

bool nQueenBoard(){

int board [N][N];

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

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

board[i][j] = 0;

if (nQueenAlgo(board, 0) == false) {

cout<<"Solution does not exist"<<endl;

return false;

}

output(board);

return true;

}

int main()

{

nQueenBoard();

return 0;

}

**Output:**

Text

Description automatically generated

**Conclusion:** Implementation of N – Queen problem has been done successfully.

**Experiment 11**

**Aim:** To implement Knapsack Problem based on Backtracking algorithm.

**Code:**

#include <iostream>

using namespace std;

#define max 100

int itemNo[max],w[max],i,j,p[max],n,m,y[max],x[max],fp=-1,fw;;

float unit[max];

void insert(){

cout<<"Enter total number of items: ";

cin>>n;

cout<<"Enter the Maximum capacity of the Sack: ";

cin>>m;

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

{

itemNo[i]=i+1;

cout<<"Enter the weight of the item "<<i+1<<": ";

cin>>w[i];

cout<<"Enter the profit of the item "<<i+1<<": ";

cin>>p[i];

}

}

void show(){

float s=0.0;

cout<<"\n\tItem\tWeight\tSelected\tProfit ";

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

cout<<"\n\t "<<itemNo[i]<<"\t\t "<<w[i]<<"\t\t "<<x[i]<<"\t\t\t "<<p[i];

cout<<"\n\nThe Sack now holds following items : ";

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

if(x[i]==1)

{

cout<<i+1<<"\t";

s += (float) p[i] \* (float) x[i];

}

cout<<"\nMaximum Profit: "<<s<<endl;

}

void sort(){

int t,t1;

float t2;

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

unit[i] = (float) p[i] / (float) w[i];

for(i=0;i<n-1;i++)

{

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

{

if(unit[i] < unit[j])

{

t = itemNo[i];

itemNo[i] = itemNo[j];

itemNo[j] = t;

t2 = unit[i];

unit[i] = unit[j];

unit[j] = t2;

t = p[i];

p[i] = p[j];

p[j] = t;

t1 = w[i];

w[i] = w[j];

w[j] =t1;

}

}

}

}

float bound(float cp,float cw,int k){

float b = cp;

float c = cw;

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

{

c = c+w[i];

if( c < m)

b = b +p[i];

else

return (b+(1-(c-m)/ (float)w[i])\*p[i]);

}

return b;

}

void knapsack(int k,float cp,float cw){

if(cw+w[k] <= m)

{

y[k] = 1;

if(k <= n)

knapsack(k+1,cp+p[k],cw+w[k]);

if(((cp+p[k]) > fp) && ( k == n))

{

fp = cp+p[k];

fw = cw+w[k];

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

x[j] = y[j];

}

}

if(bound(cp,cw,k) >= fp)

{

y[k] = 0;

if( k <= n)

knapsack(k+1,cp,cw);

if((cp > fp) && (k == n))

{

fp = cp;

fw = cw;

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

x[j] = y[j];

}

}

}

int main(){

insert();

cout<<"\nThe Sack is arranged in the order…\n";

sort();

knapsack(0,0.0,0.0);

show();

return 0;

}

**Output:**

Text

Description automatically generated

**Conclusion:** Implementation of Knapsack Problem has been done successfully.

**Experiment 12**

**Aim:** To implement Traveling Salesman problem based on Branch and Bound technique.

**Code:**

#include <climits>

using namespace std;

// `N` is the total number of total nodes on the graph or cities on the map

#define N 5

// Sentinel value for representing `INFINITY`

#define INF INT\_MAX

// State Space Tree nodes

struct Node

{

// stores edges of the state-space tree

// help in tracing the path when the answer is found

vector<pair<int, int>> path;

// stores the reduced matrix

int reducedMatrix[N][N];

// stores the lower bound

int cost;

// stores the current city number

int vertex;

// stores the total number of cities visited so far

int level;

};

// Function to allocate a new node `(i, j)` corresponds to visiting city `j`

// from city `i`

Node\* newNode(int parentMatrix[N][N], vector<pair<int, int>> const &path,

int level, int i, int j)

{

Node\* node = new Node;

// stores ancestors edges of the state-space tree

node->path = path;

// skip for the root node

if (level != 0)

{

// add a current edge to the path

node->path.push\_back(make\_pair(i, j));

}

// copy data from the parent node to the current node

memcpy(node->reducedMatrix, parentMatrix,

sizeof node->reducedMatrix);

// Change all entries of row `i` and column `j` to `INFINITY`

// skip for the root node

for (int k = 0; level != 0 && k < N; k++)

{

// set outgoing edges for the city `i` to `INFINITY`

node->reducedMatrix[i][k] = INF;

// set incoming edges to city `j` to `INFINITY`

node->reducedMatrix[k][j] = INF;

}

// Set `(j, 0)` to `INFINITY`

// here start node is 0

node->reducedMatrix[j][0] = INF;

// set number of cities visited so far

node->level = level;

// assign current city number

node->vertex = j;

// return node

return node;

}

// Function to reduce each row so that there must be at least one zero in each row

int rowReduction(int reducedMatrix[N][N], int row[N])

{

// initialize row array to `INFINITY`

fill\_n(row, N, INF);

// `row[i]` contains minimum in row `i`

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

{

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

{

if (reducedMatrix[i][j] < row[i]) {

row[i] = reducedMatrix[i][j];

}

}

}

// reduce the minimum value from each element in each row

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

{

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

{

if (reducedMatrix[i][j] != INF && row[i] != INF) {

reducedMatrix[i][j] -= row[i];

}

}

}

}

// Function to reduce each column so that there must be at least one zero

// in each column

int columnReduction(int reducedMatrix[N][N], int col[N])

{

// initialize all elements of array `col` with `INFINITY`

fill\_n(col, N, INF);

// `col[j]` contains minimum in col `j`

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

{

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

{

if (reducedMatrix[i][j] < col[j]) {

col[j] = reducedMatrix[i][j];

}

}

}

// reduce the minimum value from each element in each column

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

{

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

{

if (reducedMatrix[i][j] != INF && col[j] != INF) {

reducedMatrix[i][j] -= col[j];

}

}

}

}

// Function to get the lower bound on the path starting at the current minimum node

int calculateCost(int reducedMatrix[N][N])

{

// initialize cost to 0

int cost = 0;

// Row Reduction

int row[N];

rowReduction(reducedMatrix, row);

// Column Reduction

int col[N];

columnReduction(reducedMatrix, col);

// the total expected cost is the sum of all reductions

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

{

cost += (row[i] != INT\_MAX) ? row[i] : 0,

cost += (col[i] != INT\_MAX) ? col[i] : 0;

}

return cost;

}

// Function to print list of cities visited following least cost

void printPath(vector<pair<int, int>> const &list)

{

for (int i = 0; i < list.size(); i++) {

cout << list[i].first + 1 << " —> " << list[i].second + 1 << endl;

}

}

// Comparison object to be used to order the heap

struct comp

{

bool operator()(const Node\* lhs, const Node\* rhs) const {

return lhs->cost > rhs->cost;

}

};

// Function to solve the traveling salesman problem using Branch and Bound

int solve(int costMatrix[N][N])

{

// Create a priority queue to store live nodes of the search tree

priority\_queue<Node\*, vector<Node\*>, comp> pq;

vector<pair<int, int>> v;

// create a root node and calculate its cost.

// The TSP starts from the first city, i.e., node 0

Node\* root = newNode(costMatrix, v, 0, -1, 0);

// get the lower bound of the path starting at node 0

root->cost = calculateCost(root->reducedMatrix);

// Add root to the list of live nodes

pq.push(root);

// Finds a live node with the least cost, adds its children to the list of

// live nodes, and finally deletes it from the list

while (!pq.empty())

{

// Find a live node with the least estimated cost

Node\* min = pq.top();

// The found node is deleted from the list of live nodes

pq.pop();

// `i` stores the current city number

int i = min->vertex;

// if all cities are visited

if (min->level == N - 1)

{

// return to starting city

min->path.push\_back(make\_pair(i, 0));

// print list of cities visited

printPath(min->path);

// return optimal cost

return min->cost;

}

// do for each child of min

// `(i, j)` forms an edge in a space tree

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

{

if (min->reducedMatrix[i][j] != INF)

{

// create a child node and calculate its cost

Node\* child = newNode(min->reducedMatrix, min->path,

min->level + 1, i, j);

/\* Cost of the child =

cost of the parent node +

cost of the edge(i, j) +

lower bound of the path starting at node j

\*/

child->cost = min->cost + min->reducedMatrix[i][j]

+ calculateCost(child->reducedMatrix);

// Add a child to the list of live nodes

pq.push(child);

}

}

// free node as we have already stored edges `(i, j)` in vector.

// So no need for a parent node while printing the solution.

delete min;

}

}

int main()

{

// cost matrix for traveling salesman problem.

/\*

int costMatrix[N][N] =

{

{INF, 5, INF, 6, 5, 4},

{5, INF, 2, 4, 3, INF},

{INF, 2, INF, 1, INF, INF},

{6, 4, 1, INF, 7, INF},

{5, 3, INF, 7, INF, 3},

{4, INF, INF, INF, 3, INF}

};

\*/

// cost 34

int costMatrix[N][N] =

{

{ INF, 10, 8, 9, 7 },

{ 10, INF, 10, 5, 6 },

{ 8, 10, INF, 8, 9 },

{ 9, 5, 8, INF, 6 },

{ 7, 6, 9, 6, INF }

};

/\*

// cost 16

int costMatrix[N][N] =

{

{INF, 3, 1, 5, 8},

{3, INF, 6, 7, 9},

{1, 6, INF, 4, 2},

{5, 7, 4, INF, 3},

{8, 9, 2, 3, INF}

};

\*/

/\*

// cost 8

int costMatrix[N][N] =

{

{INF, 2, 1, INF},

{2, INF, 4, 3},

{1, 4, INF, 2},

{INF, 3, 2, INF}

};

\*/

/\*

// cost 12

int costMatrix[N][N] =

{

{INF, 5, 4, 3},

{3, INF, 8, 2},

{5, 3, INF, 9},

{6, 4, 3, INF}

};

\*/

cout << "\n\nTotal cost is " << solve(costMatrix);

return 0;

}

**Output:**

Text

Description automatically generated

**Conclusion:** Implementation of TSP problem has been done successfully.