Dept. Of Computer Science and Engineering
Dibrugarh University Institute of Engineering and Technology
Dibrugarh University
Dibrugarh-786004, Assam, India

ASSIGNMENT COVER SHEET
BACHELOR OF TECHNOLOGY

Please complete all areas of this form, sign, and attach to each submitted assignment. Submit each assignment according to the instructions provided in your Course Outline.

Assignment (Theory/Practical): Practical

Submitted By:

Name	Rathadeep Baruah
Roll No:	CSE 33/19
Semester:	5th sem

Subject Name: AAA LAB

Subject code: CSE-501

Declaration:

I declare that this assessment item is my own work, except where acknowledged, and has not been submitted for academic credit elsewhere, and acknowledge that the assessor of this item may, for the purpose of assessing this item:

- Reproduce this assessment item and provide a copy to another member of the University and/or;
- Communicate a copy of this assessment item to a plagiarism checking service (which may then retain a copy of the assessment item on its database for the purpose of future plagiarism checking).

I certify that I have read and understood the University Rules in respect of Student Academic Misconduct.

Student Signature: Rathadeep Baruah

Date: 1.02.2022

Submitted To (Faculty Name): Dr. Nomi Baruah

For Faculty:

Marks/ Grade:	

Remarks:

Dr. NOMI BARUAH
HOD (I/C)
Department of Computer Science & Engineering
DUIET, Dibrugarh University

Dept. I/C, C.S.E

Signature of Faculty

Dept. Of Computer Science and Engineering
Dibrugarh University Institute of Engineering and Technology
Dibrugarh University
Dibrugarh-786004, Assam, India

INDEX

Serial No.	Date of Experiment	Name/Subject of Experiment	Page No.	Signature	Remarks
01	04/01/22	Depth first Search	1-3		
02	05/01/22	Breadth first Search	4-6		
03	30/01/22	Dikstra's Algorithm	7-10		
04	31/01/22	Travelling Salesman Algorithm	11-13		

worst case complexity

N → no of adjacent edges

V → no of vertex

E → no of edges

Inserting all vertex in stack → O(V)

Mark all vertex as visited → O(V)

complexity of over each adjacent vertex → O(N)

for V no of vertex = $O(V * N) = O(E)$

$\therefore \quad 2V + E = O(V+E)$

$\therefore$ worst case complexity = $O(V+E)$

Algorithm: DFS(G, s)

where G is the graph & s is the source vertex.

step 1: START

step 2: Declare stack st[] & visited[]

step 3: st.push(s)

step 4: Mark s as visited in visited[]

step 5: while (st is not empty) repeat steps 6 to 10

step 6: Pop a vertex from stack st[] to visit next node

```
v = st.top()
st.pop().
```

step 7: Push all neighbours of v in stack st[] that are not visited.

step 8: for all neighbours w of v in graph G:
follow steps 9 & 10

step 9: If w is not visited do step 10

step 10: st.push(w)
mark w as visited in visited[].

step 11: End.

output -

Enter no of vertices : 4

Enter edges : 1

2

Enter edge : 2

1

Enter edge : 3

4

Enter edge : 4

3

Enter edge : -1

the graph is not connected.

Code:

```
#include <bits/stdc++.h>
using namespace std;
#define N 10000
vector<int> gr1[N], gr2[N];
bool vis1[N], vis2[N];
void add_edge(int u, int v)
{  gr1[u].push_back(u);
   gr2[v].push_back(u);
}
void dfs1(int u)
{ vis1[u] = true;
  for(auto i : gr1[u])
    if(!vis1[i])
      dfs1(i); }
void dfs2(int u)
{  vis2[u] = true;
  for(auto i : gr2[u])
  if(!vis2[i])
      dfs2(i); }
bool Is_connected(int n){
  memset(vis1, false, sizeof vis1);
  dfs1(1);
  memset(vis2, false, sizeof vis2);
  dfs2(1);
  for(int i = 1; i<=n; i++){
    if(!vis1[i] and !vis2[i])
      return false; }
  return true; }
```

```
int main(){
    int n, j, i, v;
    cout <<" Enter Number of vertices";
    cin >>n;
    while(1){
        cout <<" Enter edge";
        cin >> i;
        if(i==-1)
          break;
        cin>>j;
        add_edge(i,j);}
    if(Is_connected(n)){
        cout <<" Yes, the graph is connected";}
    else
        cout <<" The graph is not connected";
    return 0;}
```

worst case complexity:

N → No of adjacent edges

V → No of vertices

E → No of edges

Identifying all vertix in queue = O(V)

complexity to go over each adjacent vertex → O(N)

for V no of vertex = O(V * N) = O(E)

Mark all vertex as visited = O(V)

∴ T(N) = 2V + E = O(V * E)

∴ worst case complexity = O(V + E)

Algorithm : DFS (G, s)

where G is the graph & s is the source vertex.

step 1: START

step 2: Declare queue qt[] & visited[]

step 3: Inserting s in qt[] until all its neighbours vertices are marked

qt. enqueue (s)

step 4: Mark s as visited in visited[].

step 5: while (qt is not empty) repeat steps 6 to 9

step 6: Removing that vertex from queue, whose neighbour will be visited next.

v = qt. dequeue().

step 7: for all neighbours w of v in graph G repeat steps 8 to 9.

step 8: if w is not visited, do step 9

step 9: store w in qt[] to further visit its neighbours

qt. enqueue(w)

step 10: End.

output:

```
Enter number of vertices : 4

Enter edge : 0
1
Enter edge : 0
2
Enter edge : 2
0
Enter edge : 1
2
Enter edge : 2
3
Enter edge : -1

Enter source vertex : 0

Graph traversed
0 1 2 3
```

code:

```
#include<bits/stdc++.h>
using namespace std;
class graph{
    int v;
    list<int>* adj;
public:
    graph(int v);
    void addEdge(int v, int w);
    void BFS(int s);};
graph::graph(int v)
{   this->v = v;
    adj = new list<int>[v];}
void graph::addEdge(int v, int w)
{ adj[v].push_back(w);}
void graph::BFS(int s)
{ bool *visited = new bool[v];
for(int i=0; i<v; i++){
        visited[i] = false;}
list<int> queue;
visited[s] = true;
queue.push_back(s);
list<int>::iterator i;
while(!queue.empty()){
    s = queue.front();
    cout<<s<<" ";
    queue.pop_front();
```

```
for(! visited [* i]){
        visited [* i] = true;
        queue . push_back (* i);
    }
  }
 }
}
int main(){
    int n, i, j, v;
    cout << "Enter number of vertices";
    cin >> n;
     Graph g(n);
    while (1){
      cout << "Enter edge ";
      cin >> i;
    if (i == -1){
          break; }

     cin>>j;
     g. addEdge (i,j); }
    cout << " Enter source vertex ";
     cin >> v;
     g. DFS (s);
     return 0;

}
```

worst case complexity:

setting distance both v no of vertex → $O(V)$

updating each adjacent vertex weight using → $O(\log V)$
binary heap

The relaxation function traverse all edges → $O(E \log V)$

$$\therefore T(n) = O(V) + O(\log V) + O(E \log V)$$
$$= O(E \log V)$$

∴ worst case complexity = $O(E \log V)$

Algorithm : Dijkstra(G, s)

where G is the graph & s is the source vertex.

step 1 : START

step 2 : Declare array distance [], which will store distance. Declare array previous [], which will store previous node. Declare priority Q[]

step 3 : for each vertex v in graph G, repeat 4 & 5

step 4 : distance [v] ← infinity
previous [v] ← NULL

step 5 : If v not equal to s
add v to priority queue Q[]
distance [s] ← 0

step 6 : Mark selected source node with current distance 0 & rest with infinity.

step 7 : while Q[] is not empty, repeat steps 8 to 12

step 8 : u ← construct Min from Q

step 9 : Set the non-visited node with smallest current distance as the current node.

step 10 : for each unvisited neighbour v of u repeat 11 & 12

step 11 : Add current distance of the node v with the weight of the edge connecting node v with its neighbour node.

step 12 : If it is smaller than current distance of node v, set it as the current distance of v

distance [v] ← temp distance
previous [v] ← u

step 13 : END.

Output :

```
Enter number of vertices & edges : 4 6
Enter src, dist and weight : 0 1 5
Enter src, dist and weight : 1 2 2
Enter src, dist and weight : 2 3 4
Enter src, dist and weight : 3 0 7
Enter src, dist and weight : 0 2 3
Enter src, dist and weight : 1 3 1
0 to 1 , cost : 5      Previous : 0
0 to 2 , cost : 3      Previous : 0
0 to 3 , cost : 6      Previous : 1
```

Code:

```
#include <bits/stdc++.h>
using namespace std;
typedef struct nodes {
        int dest;
        int cost; } node;
class graph {
        int n;
        list <node> * adjList;
    Private:
        void showList (int src, list <node> lt) {
    list <node> :: iterator i;
    node tempNode;
for (i = lt.begin(); i != lt.end(); i++) {
    tempNode = *i;
    cout << "(" << src << ")... (" <<
tempNode.dest << "!" << tempNode.cost << ")";
}
    cout << endl;
    }
Public:
    graph () { n = 0; }
    graph (int nodeCount) {
        n = nodeCount;
        adjList = new list <node> [n]; }
void addEdge (int source, int dest, int cost) {
    node newNode;
    newNode.dest = dest;
    newNode.cost = cost;
    adjList [source].push_back (newNode); }
```

```
void displayEdges(){
    for(int i=0; i<n; i++){
        list<node> templist = adjList[i];
        showList(i, templist); } }

friend void dijkstra(graph g, int * dist,
                     int * prev, int start); };

void dijkstra(graph g, int * dist, int * prev, int start){
    int n = g.n;
    for(int u = 0; u<n; u++){
        dist[u] = 9999;
        prev[u] = -1; }

    dist[start] = 0;
    set<int> s;
    list<int> q;
    for(int u=0; u<n; u++){
        q.push_back(u); }

    while(!q.empty()){
        list<int>::iterator i;
        i = min_element(q.begin(), q.end());
        int u = *i;
        q.remove(u);
        s.insert(u);
        list<node>::iterator it;
    for(it = q.adjList[u].begin(); it!=q.adjList[u].end()
                                   it++){
        if((dist[u] + (it->cost)) < dist[it->dest]){
            dist[it->dest] = (dist[u] + (it->cost));
            prev[it->dest] = u; } } }
```

```
int main(){
    int n, e;
    int src_v, dest_v, weight;
    cout << "Enter number of vertex and edge"
    cin >> n >> e;
    graph g(n);
    int dist[n], prev[n];
    int start = 0;
    for(int i=0; i<e; i++){
      cout << "Enter source and destination with
                                    weight";
      cin >> src_v >> dest_v >> weight;
      g.addEdge(src_v, dest_v, weight);}
    dijkstra(g.dist, g.prev, start);
    for(int i=0; i<n; i++){
      if(i != start){
        cout << start << "to" << i << "cost:" << dist[i] <<
          "Previous:" << prev[i] << endl;
      }
    }
}
```

worst case complexity:

No of vertex = n

from source vertex the program has to travel $(n-1)$ vertices

so, worst case no of possible routes = $(n-1)!$

$\therefore$ Time complexity = $O(n!)$

Algorithm: TSP_dp(s, N, dist())
where s is the source vertex, N is the subset of all nodes & dist() is the function to calculate cost.

step 1: START
step 2: Declare int cost, int K for next node, visited[K]
step 3: If N = 2 i.e no of nodes & K not equal to s, do step 4. else go to step 5.
step 4: The distance bet^n the two points is the cost
cost = dist(s,K)
step 5: for j in N, do steps 6 to 8
step 6: for i in N & visited[i] = 0, do steps 7 & 8
step 7: if j not equal to i & j not equal to s, do step 8
step 8: cost = min(TSP_dp(N-{i,j}, j) + dist(j-i)
visited[j] = i
step 9: END.

output -

Travelling Salesman Distance is 85

code-

```
#include<bits/stdc++.h>
using namespace std;
#define INT_MAX 99999
int n = 4;
int dist[10][10] = { {0,20,42,25},
                     {20,0,30,34},
                     {42,30,0,10},
                     {25,34,10,0} };
int visited_ALL = (1<<n)-1;
int dp[16][4];
int tsp(int mask, int pos){
    if(mask == visited_ALL){
        return dist[pos][0];}
    if(dp[mask][pos] != -1){
        return dp[mask][pos];}
    int ans = INT_MAX;
    for(int city = 0; city<n; city++){
        if((mask & (1<<city)) == 0){
            int newAns = dist[pos][city] + tsp(mask|
                (1<<city), city);
            ans = min(ans, newAns);}}
    return dp[mask][pos] = ans;}
int main(){ for(int i=0; i<(1<<n); i++){
    for(int j=0; j<n; j++){ dp[i][j] = -1;}}
    cout<<"Travelling Salesman Distance is"<<tsp(1,0);
    return 0;
}
```