Experiment No. 3: Dynamic Programming

Date:

Aim: Write a C program to implement the following program using Dynamic programming strategy

a. Multistage Graph
b. All Pair Shortest Paths
c. Single Source Shortest Path : General Weights
d. Optimal Binary Search Trees
e. 0/1 Knapsack

THEORY:

- Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.
- One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try all possible decision sequences. We could enumerate all decision sequences and then pick out the best. But the time and space requirements may be prohibitive.
- Dynamic programming often drastically reducesthe amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal. In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the principle of optimality
- The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- Thus, the essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing suboptimal subsequences cannot be optimal (if the principleof optimality holds) and so will not (as far as possible) be generated.
- Advantages: Efficient solution construction through optimal substructure and memoization.
- Disadvantages: Requires careful identification of subproblems and may require significant memory space.

a. Multistage graph

Date:

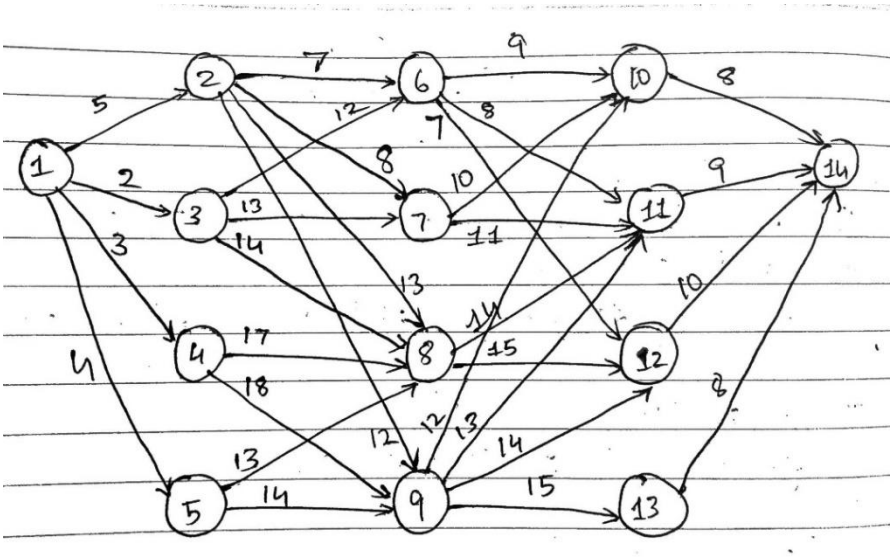
In a **multistage graph**, dynamic programming can be used to find the shortest path or minimum cost from the source vertex (stage 1) to the destination vertex (last stage).

The approach involves breaking down the problem into smaller subproblems, solving them recursively, and storing the optimal solutions in a table to avoid redundant computations. The final result is obtained by combining the optimal solutions of the subproblems.

- **Forward Graph**

Problem Statement:

Write a c program to implement forward graph on following graph:



Algorithm

```
1   Algorithm FGraph(G, k, n, p)
2   // The input is a k-stage graph G = (V, E) with n vertices
3   // indexed in order of stages. E is a set of edges and c[i, j]
4   // is the cost of ⟨i, j⟩. p[1 : k] is a minimum-cost path.
5   {
6       cost[n] := 0.0;
7       for j := n − 1 to 1 step −1 do
8       { // Compute cost[j].
9           Let r be a vertex such that ⟨j, r⟩ is an edge
10          of G and c[j, r] + cost[r] is minimum;
11          cost[j] := c[j, r] + cost[r];
12          d[j] := r;
13      }
14      // Find a minimum-cost path.
15      p[1] := 1; p[k] := n;
16      for j := 2 to k − 1 do p[j] := d[p[j − 1]];
17  }
```

Code

```
#include<stdio.h>
#include<stdlib.h>
#include<limits.h>
#define MAX 100

int g[MAX][MAX];
float cost[MAX];
int p[MAX];
int d[MAX];

void fgraph(int n,int e, int k){
   int j,r,i;
   cost[n] = 0.0;

   for(j = n -1; j >= 1; j--){
      r = -1;

      for (i = 1; i <= n; i++) {
         if (g[j][i] != INT_MAX) {
            if (r == -1 || g[j][i] + cost[i] < g[j][r] + cost[r]) {
               r = i;
            }
         }
      }
      cost[j] = g[j][r] + cost[r];
      d[j] = r;
   }
   p[1] = 1;
   p[k] = n;
   for(j = 2; j<= k-1; j++){
      p[j] = d[p[j-1]];
   }
}
int main(){
   int n,k,e,s,d,cost1,i;
   printf("Enter the no. of vertics : ");
   scanf("%d",&n);
   printf("Enter the no. of stages : ");
   scanf("%d",&k);

   for (int i = 1; i <= n; i++)
   {
      for (int j = 1; j <= n; j++)
      {
         g[i][j] = g[j][i] = INT_MAX;
      }
   }

   printf("Enter the number of edges : ");
   scanf("%d", &e);
   for (int i = 0; i < e; i++)
   {
      printf("Source : ");
      scanf("%d", &s);
      printf("Destination : ");
      scanf("%d", &d);
      printf("Cost : ");
      scanf("%d", &cost1);
      g[s][d] = cost1;
   }
   fgraph(n,e,k);

   printf("The Path is : with Cost = %.2f\n",cost[1]);
   for(i = 1; i <= k; i++){
      printf("-> %d ",p[i]);
   }
   return 0;
}
```

Output

```
Enter the no. of vertics : 14
Enter the no. of stages : 5
Enter the number of edges : 30
Source : 1
Destination : 2
Cost : 5
Source : 1
Destination : 3
Cost : 2
Source : 1
Destination : 4
Cost : 3
Source : 1
Destination : 5
Cost : 4
Source : 2
Destination : 6
Cost : 7
Source : 2
Destination : 7
Cost : 8
Source : 2
Destination : 8
Cost : 13
Source : 2
Destination : 9
Cost : 12
Source : 3
Destination : 6
Cost : 12
Source : 3
Destination : 7
Cost : 13
Source : 3
Destination : 8
Cost : 14
Source : 4
Destination : 8
Cost : 17
Source : 4
Destination : 9
Cost : 18
Source : 5
Destination : 8
Cost : 13
Source : 5
Destination : 9
Cost : 14
```

Source : 6

Destination : 10

Cost : 9

Source : 6

Destination : 11

Cost : 8

Source : 6

Destination : 12

Cost : 7

Source : 7

Destination : 10

Cost : 10

Source : 7

Destination : 11

Cost : 11

Source : 8

Destination : 11

Cost : 14

Source : 8

Destination : 12

Cost : 15

Source : 9

Destination : 10

Cost : 12

Source : 9

Destination : 11

Cost : 13

Source : 9

Destination : 12

Cost : 14

Source : 9

Destination : 13

Cost : 15

Source : 10

Destination : 14

Cost : 8

Source : 11

Destination : 14

Cost : 9

Source : 12

Destination : 14

Cost : 10

Source : 13

Destination : 14

Cost : 8

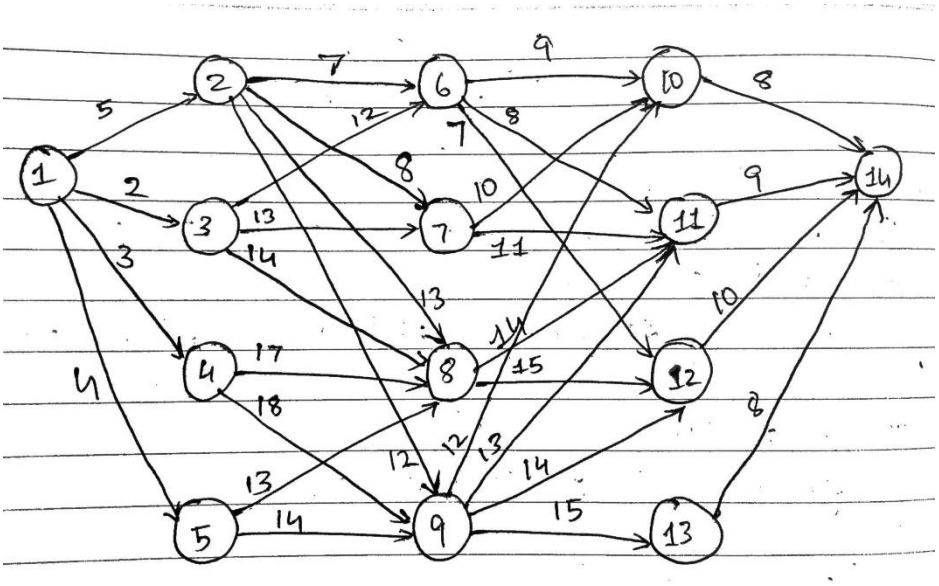
The Path is : with Cost = 29.00

-> 1 -> 2 -> 6 -> 10 -> 14

➢ Backward Graph

Problem Statement:

Write a c program to implement backward graph on following graph:



Algorithm

```
1   Algorithm BGraph(G, k, n, p)
2   // Same function as FGraph
3   {
4       bcost[1] := 0.0;
5       for j := 2 to n do
6       { // Compute bcost[j].
7           Let r be such that ⟨r, j⟩ is an edge of
8           G and bcost[r] + c[r, j] is minimum;
9           bcost[j] := bcost[r] + c[r, j];
10          d[j] := r;
11      }
12      // Find a minimum-cost path.
13      p[1] := 1; p[k] := n;
14      for j := k − 1 to 2 do p[j] := d[p[j + 1]];
15  }
```

Code

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <limits.h>
int i, j, k, a, b, u, v, n, ne = 1;
int min, mincost = 0, cost[9][9], parent[9];
int find(int i){
while (parent[i]){
i = parent[i];
}
return i;
}
int uni(int i, int j){
if (i != j){
parent[j] = i;
return 1;
}
return 0;
}
int main(){
int c, origin, dest;
printf("\nEnter the no. of vertices:");
scanf("%d", &n);
for (i = 1; i <= n; i++){
for (j = 1; j <= n; j++){
cost[i][j] = INT_MAX;
}}
int max_edges=(n*(n-1))/2;
for (i=1; i<=max_edges; i++){
printf("Enter origin and destination(-1 -1): ");
scanf("%d %d", &origin, &dest);
if (origin==-1 && dest==-1)
break;
printf("Enter cost: ");
scanf("%d", &c);
cost[origin][dest]=c;
cost[dest][origin]=c;
}
printf("\n");
while (ne < n){
min = INT_MAX;
for (i = 1; i <= n; i++){
for (j = 1; j <= n; j++){
if (cost[i][j] < min){
min = cost[i][j];
a = u = i;
b = v = j;
}}}
u = find(u);
v = find(v);
if (uni(u, v)){
mincost += min;
ne++;
printf("Step cost : %d\n",mincost);
}
cost[a][b] = cost[b][a] = INT_MAX;
```

```
}
printf("\n=> Minimum cost = %d\n",mincost);
return 0;
}
```

Output:

```
Enter the no. of vertices:8
Enter origin and destination(-1 -1): 1 2
Enter cost: 5
Enter origin and destination(-1 -1): 1 3
Enter cost: 6
Enter origin and destination(-1 -1): 2 4
Enter cost: 8
Enter cost: 4
Enter origin and destination(-1 -1): 6 8
Enter cost: 7
Enter origin and destination(-1 -1): 4 7
Enter cost: 2
Enter origin and destination(-1 -1): 7 8
Enter cost: 6
Enter origin and destination(-1 -1): 3 5
Enter cost: 9
Enter origin and destination(-1 -1): 5 7
Enter cost: 10
Enter origin and destination(-1 -1): -1 -1

Step cost : 2
Step cost : 6
Step cost : 11
Step cost : 17
Step cost : 23
Step cost : 31
Step cost : 40
=> Minimum cost = 40
```

Time Complexity & Space Complexity

Time Complexity: The time complexity of solving a multistage graph using dynamic programming is O(n * m * d), where n is the number of stages, m is the maximum number of vertices in any stage, and d is the maximum outdegree of any vertex.

Space Complexity: The space complexity is O(n * m) for the dynamic programming table, as we need to store optimal values for each vertex in each stage. Additional space may be required for graph representation, typically O(m + e), where e is the number of edges in the graph.

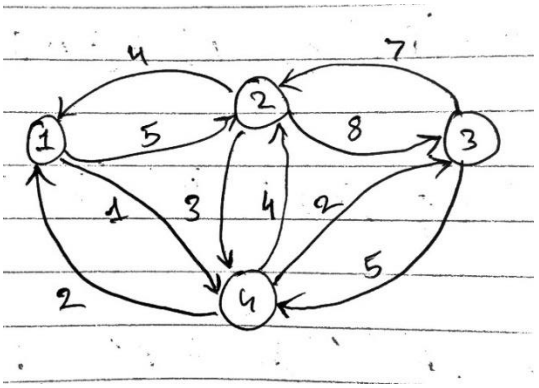
b. All Pair Shortest Paths

Date:

The **all-pairs shortest path** problem using dynamic programming aims to find the shortest path between every pair of vertices in a graph. It involves constructing a distance matrix iteratively, considering intermediate vertices in each step. By comparing and updating distances, the optimal shortest paths are obtained.

Problem Statement:

Write a c program to implement all pair shortest paths on following graph:



Algorithm

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

Time Complexity and Space Complexity

Time Complexity: The time complexity of the Floyd-Warshall algorithm, which is commonly used for the all-pairs shortest path problem, is O(V^3), where V is the number of vertices in the graph. This is because we need to consider all possible pairs of vertices and update the distances for each intermediate vertex.

Space Complexity: The space complexity is O(V^2) as we need to store a V x V distance matrix to keep track of the shortest distances between all pairs of vertices. Additionally, if we also want to reconstruct the shortest paths, an additional V x V matrix may be required to store the next vertex in the path.

Code

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX 100

int cost[MAX][MAX];

int A[MAX][MAX];

int min(int a, int b){

if(a > b){

return b;

}}

void allpairshort(int n, int e){

int i,j,k,count = 0;

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

cost[i][i] = 0;

}

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

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

A[i][j] = cost[i][j];

}}

//display A0 Matrix

printf("\nA0 = \n");

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

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

if (A[i][j] == INT_MAX) {

printf("INF ");

} else {

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

}}

printf("\n");

}

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

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

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

if (A[i][k] != INT_MAX && A[k][j] !=
INT_MAX) {

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

}}}

//display all the A1 to An matrix

printf("\nA%d = \n",k);

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

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

if (A[i][j] == INT_MAX) {

printf("INF ");

} else {
```

```
printf("%d ", A[i][j]);
}}
printf("\n");
}}}
int main()
{
int n, e, s, d, cost1,i,j;
printf("Enter the number of vertex : ");
scanf("%d", &n);
for (int i = 1; i <= n; i++){
for (int j = 1; j <= n; j++){
cost[i][j] = cost[j][i] = INT_MAX;
}
}
printf("Enter the number of edges : ");
scanf("%d", &e);
for (int i = 0; i < e; i++)
{
printf("Source : ");
scanf("%d", &s);
printf("Destination : ");
scanf("%d", &d);
printf("Cost : ");
scanf("%d", &cost1);
cost[s][d] = cost1;
}
allpairshort(n, e);
return 0;
}
```

Output:

```
Enter the number of vertex : 4
Enter the number of edges : 10
Source : 1
Destination : 2
Cost : 5
Source : 2
Destination : 1
Cost : 4
Source : 1
Destination : 4
Cost : 1
Source : 4
Destination : 1
Cost : 2
```

Source : 2

Destination : 4

Cost : 3

Source : 4

Destination : 2

Cost : 4

Source : 2

Destination : 3

Cost : 8

Source : 3

Destination : 2

Cost : 7

Source : 4

Destination : 3

Cost : 2

Source : 3

Destination : 4

Cost : 5

A0 =

0 5 INF 1

4 0 8 3

INF 7 0 5

2 4 2 0

A1 =

0 5 INF 1

4 0 8 3

INF 7 0 5

2 4 2 0

A2 =

0 5 13 1

4 0 8 3

11 7 0 5

2 4 2 0

A3 =

0 5 13 1

4 0 8 3

11 7 0 5

2 4 2 0

A4 =

0 5 3 1

4 0 5 3

7 7 0 5

2 4 2 0

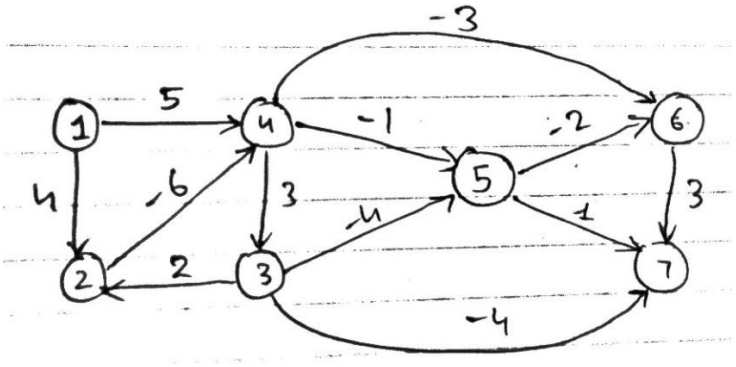
c. Single Source Shortest Paths : General Weights

Date:

The **single-source shortest paths** problem with general weights using dynamic programming aims to find the shortest path from a single source vertex to all other vertices in a graph, considering arbitrary edge weights. It is also called **bellman ford** algorithm

Problem Statement:

Write a c program to implement single souce shortest paths on following graph:



Algorithm

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n − 1 do
8           for each u such that u ≠ v and u has
9                   at least one incoming edge do
10              for each ⟨i, u⟩ in the graph do
11                  if dist[u] > dist[i] + cost[i, u] then
12                      dist[u] := dist[i] + cost[i, u];
13  }
```

Time Complexity and Space Complexity

Time Complexity: The time complexity of solving the single-source shortest paths problem with general weights using dynamic programming is typically O(V * E), where V is the number of vertices and E is the number of edges in the graph.

Space Complexity: The space complexity is O(V), where V is the number of vertices, as we need to store the shortest distance values for each vertex.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAX 100
int cost[MAX][MAX];
int dist[MAX];
void bellmanford(int v, int n, int e){
int i,k,j;
for(i = 1; i <= n; i++){
dist[i] = cost[v][i];
}
dist[v] = 0;
for(k = 2; k <= n-1; k++){
for (int u = 1; u <= n; u++) {
for (int i = 1; i <= n; i++) {
if(i != v) {
for(j = 1; j <= n; j++) {
if(cost[i][j] != INT_MAX){
if(dist[j] > dist[i] + cost[i][j]) {
dist[j] = dist[i] + cost[i][j];
}}}}}}}
int main(){
int n, e, s, d, cost1,i,j,v = 1;
printf("Enter the number of vertex : ");
scanf("%d", &n);
for (int i = 1; i <= n; i++){
for (int j = 1; j <= n; j++){
cost[i][j] = cost[j][i] = INT_MAX;
}}
printf("Enter the number of edges : ");
scanf("%d", &e);
for (int i = 0; i < e; i++)
{
printf("Source : ");
scanf("%d", &s);
printf("Destination : ");
scanf("%d", &d);
printf("Cost : ");
scanf("%d", &cost1);
cost[s][d] = cost1;
}
bellmanford(v, n, e);
printf("Shortest distance from source
vertex %d :\n", v);
for (int i = 1; i <= n; i++) {
printf("Vertex %d : %d\n",i,dist[i]);
}
return 0;
}
```

Output:

Enter the number of vertex : 7

Enter the number of edges : 12

Source : 1

Destination : 2

Cost : 4

Source : 1

Destination : 4

Cost : 5

Source : 2

Destination : 4

Cost : -6

Source : 3

Destination : 2

Cost : 2

Source : 4

Destination : 3

Cost : 3

Source : 4

Destination : 6

Cost : -3

Source : 4

Destination : 5

Cost : -1

Source : 3

Destination : 5

Cost : -4

Source : 3

Destination : 7

Cost : -4

Source : 5

Destination : 6

Cost : -2

Source : 5

Destination : 7

Cost : 1

Source : 6

Destination : 7

Cost : 3

Shortest distance from source vertex 1 :

Vertex 1 : 0

Vertex 2 : -2147483647

Vertex 3 : 1

Vertex 4 : -2

Vertex 5 : -3

Vertex 6 : -5

Vertex 7 : -3

d. Optimal Binary Search Trees

Date:

Optimal Binary Search Trees is a dynamic programming algorithm that finds the most efficient binary search tree for a given set of keys with associated probabilities or frequencies. It utilizes bottom-up computation and memoization to find the optimal structure, minimizing the expected search cost.

Problem Statement:

To implement Optimal binary search tree :

$(a_{1.....}a_5) = \{ Apr,\ Mar,\ May,\ Oct,\ Sept\}$ n = 5

$p_{1.....}p_5 = (3, 4, 3, 2, 4)$

$q_{0.....}q_5 = (4, 4, 5, 4, 5, 4)$

Algorithm

```
1   Algorithm OBST(p, q, n)
2   // Given n distinct identifiers a_1 < a_2 < ... < a_n and probabilities
3   // p[i], 1 ≤ i ≤ n, and q[i], 0 ≤ i ≤ n, this algorithm computes
4   // the cost c[i, j] of optimal binary search trees t_ij for identifiers
5   // a_{i+1}, ..., a_j. It also computes r[i, j], the root of t_ij.
6   // w[i, j] is the weight of t_ij.
7   {
8       for i := 0 to n − 1 do
9       {
10          // Initialize.
11          w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;
12          // Optimal trees with one node
13          w[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
14          r[i, i + 1] := i + 1;
15          c[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
16      }
17      w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;
18      for m := 2 to n do  // Find optimal trees with m nodes.
19          for i := 0 to n − m do
20          {
21              j := i + m;
22              w[i, j] := w[i, j − 1] + p[j] + q[j];
23              // Solve 5.12 using Knuth's result.
24              k := Find(c, r, i, j);
25                  // A value of l in the range r[i, j − 1] ≤ l
26                  // ≤ r[i + 1, j] that minimizes c[i, l − 1] + c[l, j];
27              c[i, j] := w[i, j] + c[i, k − 1] + c[k, j];
28              r[i, j] := k;
29          }
30      write (c[0, n], w[0, n], r[0, n]);
31  }
```

```
1   Algorithm Find(c, r, i, j)
2   {
3       min := ∞;
4       for m := r[i, j − 1] to r[i + 1, j] do
5           if (c[i, m − 1] + c[m, j]) < min then
6           {
7               min := c[i, m − 1] + c[m, j]; l := m;
8           }
9       return l;
10  }
```

Time complexity and Space complexity

Time Complexity: The time complexity of the Optimal Binary Search Trees algorithm using dynamic programming is O(n^3), where n is the number of keys in the input set.

Space Complexity: The space complexity is O(n^2), where n is the number of keys. This is because we need to store a table of size n x n to store the computed values for the optimal subproblems.

Code

```
#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

#define MAX 100

int p[MAX]={999,3,4,3,2,4};

int q[MAX]={4,4,5,4,5,4};

int w[MAX][MAX];

int c[MAX][MAX];

int r[MAX][MAX];

void display();

int find(int i,int j){

   int min=INT_MAX;

   int l;

   for(int m=i+1;m<=j;m++)

   {

      if(c[i][m-1]+c[m][j]<min)

      {

         min=c[i][m-1]+c[m][j];

         l=m;

      }}

   return l;

}

int main(){

   int k;

   for(int i=0;i<=5;i++){

      w[i][i]=q[i];

      r[i][i]=0;

      c[i][i]=0;

   }

   for(int i=0;i<=3;i++){
```

```
        w[i][i+1]=q[i]+q[i+1]+p[i+1];
        r[i][i+1]=i+1;
        c[i][i+1]=q[i]+q[i+1]+p[i+1];
    }
    for(int m=1;m<=6;m++){
        for(int i=0;i<=6-m;i++){
            int j=i+m;
            w[i][j]=w[i][j-1]+p[j]+q[j];
            k=find(i,j);
            c[i][j]=w[i][j]+c[i][k-1]+c[k][j];
            r[i][j]=k;
        }}
display();
}
void display(){
   int a=0;
   int i=0;
   int j=0;
   while(a<6)
   {
    i=0;
    j=a;
   while(i<6-a&&j<6){
    printf("w[%d][%d]=%d\t",i,j,w[i][j]);
    i++;
    j++;
    }
    i=0;
    j=a;
    printf("\n");
    while(i<6-a&&j<6){
        printf("c[%d][%d]=%d\t",i,j,c[i][j]);
        i++;
        j++;
    }
    i=0;
    j=a;
    printf("\n");
    while(i<6-a&&j<6){
    printf("r[%d][%d]=%d\t",i,j,r[i][j]);
    i++;
    j++;
    }
    printf("\n------------------------------\n");
    a++;
  }
  }
```

Output

w[0][0]=4 w[1][1]=4 w[2][2]=5 w[3][3]=4 w[4][4]=5 w[5][5]=4

c[0][0]=0 c[1][1]=0 c[2][2]=0 c[3][3]=0 c[4][4]=0 c[5][5]=0

r[0][0]=0 r[1][1]=0 r[2][2]=0 r[3][3]=0 r[4][4]=0 r[5][5]=0

---

w[0][1]=11 w[1][2]=13 w[2][3]=12 w[3][4]=11 w[4][5]=13

c[0][1]=11 c[1][2]=13 c[2][3]=12 c[3][4]=11 c[4][5]=13

r[0][1]=1 r[1][2]=2 r[2][3]=3 r[3][4]=4 r[4][5]=5

---

w[0][2]=20 w[1][3]=20 w[2][4]=19 w[3][5]=19

c[0][2]=31 c[1][3]=32 c[2][4]=30 c[3][5]=30

r[0][2]=2 r[1][3]=2 r[2][4]=3 r[3][5]=5

---

w[0][3]=27 w[1][4]=27 w[2][5]=27

c[0][3]=50 c[1][4]=51 c[2][5]=52

r[0][3]=2 r[1][4]=3 r[2][5]=4

---

w[0][4]=34 w[1][5]=35

c[0][4]=75 c[1][5]=78

r[0][4]=2 r[1][5]=3

---

w[0][5]=42

c[0][5]=103

r[0][5]=3

---

e. 0/1 Knapsack

Date:

The **0/1 knapsack** problem is a dynamic programming algorithm that solves the optimization problem of selecting items with maximum total value within a given weight capacity. It utilizes a dynamic programming table to store and compute the optimal solutions for subproblems, considering the constraints of item weights and values.

Problem Statement:

To implement 0/1 Knapsack Using Dynamic Programming :

p = (18, 4, 6, 21, 7, 14, 13) m = 15

w = (3, 1, 3, 4, 1, 1, 2) n = 7

Algorithm

```
1   Algorithm DKnap(p, w, x, n, m)
2   {
3       // pair[ ] is an array of PW's.
4       b[0] := 1; pair[1].p := pair[1].w := 0.0; // S^0
5       t := 1; h := 1; // Start and end of S^0
6       b[1] := next := 2; // Next free spot in pair[ ]
7       for i := 1 to n − 1 do
8       { // Generate S^i.
9           k := t;
10          u := Largest(pair, w, t, h, i, m);
11          for j := t to u do
12          { // Generate S_1^{i−1} and merge.
13              pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
14                  // (pp, ww) is the next element in S_1^{i−1}.
15              while ((k ≤ h) and (pair[k].w ≤ ww)) do
16              {
17                  pair[next].p := pair[k].p;
18                  pair[next].w := pair[k].w;
19                  next := next + 1; k := k + 1;
20              }
21              if ((k ≤ h) and (pair[k].w = ww)) then
22              {
23                  if pp < pair[k].p then pp := pair[k].p;
24                  k := k + 1;
25              }
26              if pp > pair[next − 1].p then
27              {
28                  pair[next].p := pp; pair[next].w := ww;
29                  next := next + 1 ;
30              }
31              while ((k ≤ h) and (pair[k].p ≤ pair[next − 1].p))
32                  do k := k + 1;
33          }
34          // Merge in remaining terms from S^{i−1}.
35          while (k ≤ h) do
36          {
37              pair[next].p := pair[k].p; pair[next].w := pair[k].w;
38              next := next + 1; k := k + 1;
39          }
40          // Initialize for S^{i+1}.
41          t := h + 1; h := next − 1; b[i + 1] := next;
42      }
43      TraceBack(p, w, pair, x, m, n);
44  }
```

Time complexity and Space complexity

Time Complexity: The time complexity of the 0/1 knapsack problem using dynamic programming is O(nW), where n is the number of items and W is the maximum capacity of the knapsack.

Space Complexity: The space complexity is O(nW), where n is the number of items and W is the maximum capacity of the knapsack. This is because we need to store a table of size (n+1) x (W+1) to store the computed values for the optimal subproblems.

Code

```
#include<stdio.h>

#define MAX_ITEMS 100

void knapsack(int n, int weight[MAX_ITEMS], int profit[MAX_ITEMS], int maxWeight, int knapsackTable[MAX_ITEMS][MAX_ITEMS], int selected[MAX_ITEMS]){

int i,j,k;

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

knapsackTable[i][0]=0;

}

for(j=0;j<=maxWeight;j++){

knapsackTable[0][j]=0;

}

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

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

if(weight[i]>j){

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

}

else{

if(knapsackTable[i-1][j]>profit[i]+knapsackTable[i-1][j-weight[i]]){

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

}else{

knapsackTable[i][j]=profit[i]+knapsackTable[i-1][j-weight[i]];
```

```
}}}}
i=n;
k=maxWeight;
while(i>0 && k>0){
if(knapsackTable[i][k]!=knapsackTable[i-1][k]){
selected[i]=1;
k=k-weight[i];
}
i=i-1;
}}
int main(){
int i,j,n,maxWeight,profitSum,weightSum;
int weight[MAX_ITEMS], profit[MAX_ITEMS],
knapsackTable[MAX_ITEMS][MAX_ITEMS], selected[MAX_ITEMS];
printf("Enter the number of items: ");
scanf("%d",&n);
printf("Enter the maximum knapsack weight: ");
scanf("%d",&maxWeight);
printf("\nWeight of items\n");
for(i=1;i<=n;i++){
printf("Enter weight and profit of item %d: ",i);
scanf("%d",&weight[i]);
scanf("%d",&profit[i]);
}
knapsack(n,weight,profit,maxWeight,knapsackTable,selected);
profitSum=0;
weightSum=0;
for(i=1;i<=n;i++){
```

```
profitSum+=selected[i]*profit[i];
weightSum+=selected[i]*weight[i];
}
printf("Max Profit: %d\n",profitSum);
printf("Optimal weight: %d\n",weightSum);
return 0;
}
```

Output

```
Enter the number of items: 7
Enter the maximum knapsack weight: 15
Weight of items
Enter weight and profit of item 1: 3 18
Enter weight and profit of item 2: 1 4
Enter weight and profit of item 3: 3 6
Enter weight and profit of item 4: 4 21
Enter weight and profit of item 5: 1 7
Enter weight and profit of item 6: 1 14
Enter weight and profit of item 7: 2 13
Max Profit: 83
Optimal weight: 15
```

CONCLUSION:

Dynamic programming was studied. The programs for (a) Multistage graph, (b) All pair shortest paths (c) Single Source Shortest Path : General Weights (d) Optimal Binary Search Trees, and (e) 0/1 Knapsack using dynamic programming were studied and implemented successfully.