EXPERIMENT 3 **DATE –**

DYNAMIC PROGRAMMING

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 decision.

An optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision. This is true for all problems solvable by the greedy method. For many other problems, it is not possible to make stepwise decisions (based only on local information) in such a manner that the sequence of decisions made is optimal.

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 reduces the 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 principle of optimality holds) and so will not (as far as possible) be generated.

While the principle of optimality has been stated only with respect to the initial state and decision, it can be applied equally well to intermediate states and decisions.

Because of the use of the principle of optimality, decision sequences containing subsequences that are suboptimal are not considered. Although the total number of different decision sequences is exponential in the number of decisions (if there are d choices for each of the n decisions to be made then there are d^n possible decision sequences), dynamic programming algorithms often have a polynomial complexity. Another important feature of the dynamic programming approach is that optimal solutions to subproblems are retained so as to avoid recomputing their values. The use of these tabulated values makes it natural to recast the recursive equations into an iterative algorithm

Date -

MULTISTAGE GRAPHS (FORWARD AND BACKWARD APPROACH)

AIM – Write a C program to calculate shortest path for traversing multistage stage graph using forward and backward approach

Statement – Given a multistage graph with k stages ,n vertices and given edges ,implement multistage graph approach to determine shortest path for traversing the graph .

Input - Number of stages = 5 , number of vertices = 14 and all the edges of graph are inputed

Output - i} Shortest path for traversing the graph from 1st stage to 5th stage

ii} Shortest path for traversing the graph from 5th stage to 1st stage

ALGORITHM –

I] Algorithm FGraph(G,k,n,p)

```
// The input is a k-stage graph G=(V,E) with n vertices
// indexed in order of stages. E is a set of edges and c[i,j]
// is the cost of <i,j>. p[1:k] is a minimum-cost path.
{
  cost[n] := 0.0;
  for j := n-1 to 1 step -1 do
  { // Compute cost[j].
    Let r be a vertex such that <j,r> is an edge
    of G and c[j,r] + cost[r] is minimum;
    cost[j] := c[j,r] + cost[r];
    d[j] := r;
  }
  // Find a minimum-cost path.
  p[1] := 1; p[k] := n;
  for j := 2 to k-1 do p[j] := d[p[j-1]];
}
```

Time Complexity:

The current algorithm has a time complexity of **O(n^2)**, but this can potentially be improved to **O(V + E)** (where V is the number of vertices and E is the number of edges) using a priority queue (min-heap) and an adjacency list.

Space Complexity:

The current algorithm has a space complexity of **O(1)**, since the graph size and the arrays used are all bounded by the constant MAX. The ideal space complexity, when using an adjacency list, would be **O(V + E)**, which is much more efficient for sparse graphs where the number of edges E is much smaller than V^2.

II]Algorithm BGraph(G,k,n,p)

// Same function as FGraph

```
bcost[1] := 0.0;
for j := 2 to n do
{ // Compute bcost[j].
  Let r be such that <r,j> is an edge of
  G and bcost[r] + c[r,j] is minimum;
  bcost[j] := bcost[r] + c[r,j];
  d[j] := r;
}
// Find a minimum-cost path.
p[1] := 1; p[k] := n;
for j := k - 1 to 2 do p[j] := d[p[j+1]];
}
```

Time Complexity:

The current algorithm has a time complexity of $O(n^2)$, but this can potentially be improved to O(V + E) (where V is the number of vertices and E is the number of edges) using a priority queue (min-heap) and an adjacency list.

Space Complexity:

The current algorithm has a space complexity of O(1), since the graph size and the arrays used are all bounded by the constant MAX. The ideal space complexity, when using an adjacency list, would be O(V + E), which is much more efficient for sparse graphs where the number of edges E is much smaller than V^2.

Date –

BELLMAN FORD ALGORITHM

AIM – Write a C program to calculate shortest path from a source vertex to all the vertexes of a given graph using bellman ford algorithm

Problem statement – Given a cost adjacency matrix of a graph apply bellman ford algorithm to calculate shortest path from a given vertex to all other vertex in the graph

Input – The cost adjacency matrix A is inputed

Output - Shortest path from a given vertex to all vertexes of the graph .

ALGORITHM

Algorithm BellmanFord(v, cost, dist, n)

```
// Single-source/all-destinations shortest paths with negative edge costs
{
   for i := 1 to n do            // Initialize dist.
      dist[i] := cost[v, i];

   for k := 2 to n − 1 do         // Relax edges repeatedly
      for each u such that u ≠ v and u has at least one incoming edge do
         for each (i, u) in the graph do
            if dist[u] > dist[i] + cost[i, u] then
               dist[u] := dist[i] + cost[i, u];
}
```

Time Complexity:

Each iteration of the for loop takes $O(n^2)$ time if adjacency matrices are used and O(E) time if adjacency lists are used, where E is the number of edges in the graph, resulting in an overall complexity of **$O(n^3)$** when using adjacency matrices and **O(nE)** when using adjacency lists.

Space Complexity:

The space complexity of the Bellman-Ford algorithm is O(V) (for distance array) + O(E) (graph storage) = **O(V+ E).** In practice, the distance array is the critical storage requirement. The edge list is part of the input and not always counted in auxiliary space. Thus, the standard space complexity is considered **O(V).**

Time Complexity:

The OBST algorithm computes the weight (w), cost (c), and root (r) matrices in $\mathbf{O(n^2)}$ time using Knuth's optimization (restricting root searches to $r(i,j-1) \leq k \leq r(i+1,j)$), then constructs the optimal tree from r in **O(n)** time.

Space Complexity:

The space complexity is $\mathbf{O(n^2)}$ due to the three n×n matrices (w, c, r).

II]Algorithm mColoring(k)

```
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1:n,1:n]. All assignments of 1,2,...,m to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index
// of the next vertex to color.
{
   repeat
   {// Generate all legal assignments for x[k].
      NextValue(k); // Assign to x[k] a legal color.
      if (x[k] = 0) then return; // No new color possible
      if (k = n) then    // At most m colors have been
      // used to color the n vertices.
         write (x[1:n]);
      else mColoring(k + 1);
   } until (false);
}
```

Time complexity: O(nm^n)

n : number of vertices

m: number of colors

This is because each vertex can be assigned one of m colors, and we explore all possibilities recursively.

Space complexity: O(n^2)

until (false); }

II]Algorithm Hamiltonian(k)

```
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1:n,1:n]. All cycles begin at node 1.
{
   repeat
   { // Generate values for x[k].
      NextValue(k); // Assign a legal next value to x[k].
      if (x[k] = 0) then return;
      if (k = n) then write (x[1:n]);
      else Hamiltonian(k + 1);
   } until (false);
}
```

Time complexity: O(N!)

N is the number of vertices in the graph.

Space complexity: O(N)

N is the number of vertices in the graph.

DATE –

KNAPSACK PROBLEM

AIM- Write a C program to implement 0/1 Knapsack problem using backtracking.

Problem statement – Consider we are given n objects and knapsack capacity of 'm' ,each object i has weight w_i and profit p_i .The objective is to fill the knapsack that maximizes the profits and since the capacity is m ,the toal weight must be less than or equal to m .

Input – $[p_1.....p_7]$ = { 17,14,20,18,22}

$[w_1....w_7]$ = 6,5,10,11,14} m = 21

Output - Generate the list of items that give maximum profit and has the total weight within the knapsack capacity .

ALGORITHM

I]Algorithm Bound(cp,cw,k)

```
// cp is the current profit total, cw is the current
// weight total; k is the index of the last removed
// item; and m is the knapsack size.
{
   b := cp; c := cw;
   for i := k + 1 to n do
   {
      c := c + w[i];
      if (c < m) then b := b + p[i];
      else return b + (1 - (c - m)/w[i]) * p[i];
   }
   return b;
}
```

II]Algorithm BKnap(k,cp,cw)

```
// m is the size of the knapsack; n is the number of weights
// and profits. w[] and p[] are the weights and profits.
// p[i]/w[i]≥p[i+1]/w[i+1]. fw is the final weight of
// knapsack; fp is the final maximum profit. x[k]=0 if w[k]
// is not in the knapsack; else x[k]=1.
{
   // Generate left child.
   if (cw + w[k] ≤ m) then
   {
      y[k] := 1;
      if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
      if ((cp + p[k] > fp) and (k = n)) then
      {
         fp := cp + p[k]; fw := cw + w[k];
         for j := 1 to k do x[j] := y[j];
      }
   }

   // Generate right child.
   if (Bound(cp,cw,k) ≥ fp) then
   {
      y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
      if ((cp > fp) and (k = n)) then
      {
         fp := cp; fw := cw;
         for j := 1 to k do x[j] := y[j];
      }
   }
}
```

Time complexity: $O(2^n)$.

Space complexity: O(n).