Dynamic Programming : 0/1 Knapsack

Problem definition:

Given a knapsack with capacity M and N items with weights w_i and benefit b_i for 1<=i<=N, the problem is to

Maximize $\sum P_i * X_i$,

Subject to the condition $\sum W_i * X^* <= M$ where Xi= 0 or 1.

Structure of Optimal solution :
(optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.)

Problem substructure- every element with its associated weight and profit.

For the 0-1 problem, consider that the most valuable collection weighs at most W units. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W-w_j$ that the thief can take from the n -1 original items excluding j .

In other words, at every step , the decision to add or remove any item is taken based on following conditions

a. If new element j can be added to gain more profit than existing knapsack profit
b. Simple addition of new element has more profit or adding it by removing some elements from knapsack gives more profit
   As every decision depends on both of above conditions, final profit is the best of all of the elements' configuration within the given constraints.

Recursive formula:

$$\mathrm{B(i,w)} = \begin{cases} 0, if \;\; i = 0 \; or \; j = 0 \\ B[i-1, j], if \;\; wi < w \\ \max\{B[i-1, j], bi + B[i-1, w-wi\} \; Otherwise \end{cases}$$

Complexity :

T(n)= O(N*M) where ,

N= number of elements

M= Knapsack Capacity

Dynamic Programming: Multistage Graphs

Problem definition:

A multistage graph G=(V,E) is a directed graph in which the vertices are partitioned into k ≥ 2 disjoint sets V_i, 1 ≤ i ≤ k . In addition if (u, v) is an edge in E, then u ε V_i and v ε V_{i+1}, for some i, 1 ≤ i ≤ k. The sets V_1 and V_k are such that $|V_1|=|V_k|=1$. Let s and t, respectively be the vertex in V_1 and V_k. The vertex s is the source, and t the sink.

Structure of Optimal solution :

A dynamic programming formulation for k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of (k-2) decisions. The i-th decision involves determining which vertex in V_{i+1}, 1< i < k-2, is to be on the path.

For every k= 1,2…. K-2 decision, a path with minimum value is chosen, which assure that principal of optimality holds and the user gets the path of minimum length from S to T.

Recursive Formula:

Forward computation

$$\text{cost(i,j)} = \begin{cases} C(j,T), \quad if \;\; i == k-1 \; (second\ last\ stage\ in\ the\ graph) \\ \min \quad \{\text{c(j ,l)} + \text{ cost(i + 1,l)}\} \\ where\ l \in Vi-1 \ and < l,j > \in E \ , for\ all\ such\ values\ of\ l \end{cases}$$

Forward computation

$$\text{bcost(i,j)} = \begin{cases} \quad C(S,j), \quad if \;\; i == 2 \; (second\ stage\ in\ the\ graph) \\ \min \quad \{\text{bcost(l ,j)} + \text{ c(i − 1,l)}\} \\ where\ l \in Vi+1 \ and < j,l > \in E \ , for\ all\ such\ values\ of\ l \end{cases}$$

Complexity :

N= total number of vertices

K-2= number of stages excluding source and destination stages

So, T(n)=O(N*(k-2))=O(NK)

Dynamic Programming: Single Source Shortest Path

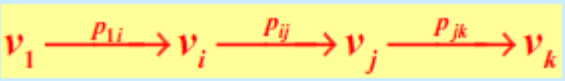
Problem definition: Let G=<V,E> be a weighted graph and source Є V, then single source shortest path is the problem of computing shortest path from source to every other vertex in V. The weights could be negative.

Optimal Substructure:

Suproblem structure:- path length

Given a weighted directed graph (G=(V,E),w). If d=<v1,v2,v3…vk> is the shortest path from v1 to vk, then for every i,j where 1<=i<=j<=k, the subpath dij=<vi,vi+1,…vj> is also a shortest path from Vi to vj.

As the shortest path is chosen for path length k=2,3,..n-1, the final path is optimal and principal of optimality holds.



Recursive Formula:

$Dist^{k}_{[u]}=min\{ dist^{k-1}_{[u]}, min\ I\ \{dist^{k}_{[i]}+cost[i,u]\}\}$

for k=2,3,….n-1

Complexity:

N=total number of vertices,

n-1= Max Path length

N*(N-1)*N= for N vertices, N-1 possible iterations , calculated N times.

So, $T(n)=O(n^3)$

Dynamic Programming: All Sources Shortest Path

Problem definition:

Given graph G=(V,E) with positive weights W(u,v) on the edges (u, v) for all values of u & v, then all sources shortest path is the problem of computing shortest path from every vertex to every other vertex in V. The weights cannot be negative.

Optimal Substructure:

Let $D_{ij}^{(m)}$ be the length of shortest path from i to j that contains at most m edges and $D^{(m)}$ be the nXn matrix with all $[d_{ij}^{(m)}]$ values. Then $d_{ij}^{(n-1)}$ gives the shortest path from every vertex i to every vertex j .

This way, computing $D^{(m)}$ for m=1,…n-1 while choosing the minimum value at every stage assures optimal solution and so principal of optimality holds.

Recursive Formula

$A^k[i,j]=$ O IF (i==j)

$A^k[i,j]=$ min{ $A^{k-1}[i,k]+A^{k-1}[k,j]$}, $A^{k-1}[i,j]$}, otherwise.

Complexity:

N= number of vertices

N-1= Number of iterations for N source vertices to N destination vertices

So, $T(N)= O(N^3)$

Dynamic Programming: Longest Common subsequence

Problem definition: Given two strings X and Y such that, $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$, the problem is to find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

Optimal Substructure:

Let X = <x_1; x_2,…, x_m> and Y = <y_1; y_2, ….., y_n> be sequences, and let Z=<z_1;z_2, …Z_k> be any LCS of X and Y .
1. If x_m= y_n, then z_k =x_m = y_n and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If x_m != y_n, then z_k != x_m implies that Z is an LCS of X_{m-1} and Y .
3. If x_m !=y_n, then z_k!= y_n implies that Z is an LCS of X and Y_{n-1}.

Recursive Formula:

$$C[i,j] = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ C[i-1, j-1] + 1 & if\ i,j > 0\ and\ Xi == Yj \\ max\{\ C[i-1,j], C[i,j-1]\} & otherwise \end{cases}$$

Complexity:

M= length of string 1

N= length of string 2

T(N)= O(N*M)

Dynamic Programming: Travelling Salesperson Problem

Problem definition:

Given a complete graph $G=(V, E)$ that has nonnegative integer cost $c(u, v)$ associated with each edge (u, v) in E, the problem is to find a minimum cost tour of G which starts and ends at the same source vertex.

Optimal Substructure:

Let g(i,S) be the length of minimum cost tour which starts at i, goes through all vertices of S and ends at chosen source vertex.

Computing this g(i,S) for all i=2….n gives the minimum cost tours by gradually increasing path length.

All these g(i,S) values contribute to computation of g(1,{S}) where S= {V}-1 to yield an optimal tour for the given graph.

Recursive Formula:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{C1k + g(k, V - \{1, k\})\} \qquad (1)$$

Generalizing (1), we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{ c_{ij} + g(j, S - \{j\})\} \qquad (2)$$

Complexity:

N= Number of cities

2^N= number of subsets of set S; each subset takes linear time to solve

$T(N)= O(N*2^N)$

Algorithm DP_TSP(V,E,C)

//V is set of vertices, E-set of edges and Cis the cost matrix.

{

g(1,{1}) = 0

for j= 2 to n

for s = 2 to n:
    for all subsets S ⊆ {1,2,...,n} of size s and containing 1:
        g(1,S) = ∞

                for all j∈S,j≠1:
            g(j,S) = min{Cij+ g(j, S−{j}):i∈S,i≠j}
return C1j+minj g(j,{1,...,n})

}

Dynamic Programming :Matrix Chain Multiplication

Problem definition:

Given a sequence of N matrices, the matrix chain multiplication problem is to find the most efficient way to multiply these matrices by minimizing the number of computations involved during multiplications.

Optimal Substructure: parenthesization/ select the subgroup of matrices that will result in least number of computations.

For multiplication of matrix series Ai to A_j, choose A_k such that multiplication of matrices through A_i..k and A_{k+1}…j will incur least number of computations for any k such that i<=k<j.

Recursive Formula:

Step 2: Recursively define the value of an optimal solution.

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) & i < j \end{cases}$$

Complexity: for every Ai to Aj, compute values for all k such that i<=k<j, resulting in three for loops. i.e. for i,j and k.

So for N matrices-

Complexity: The loops are nested three deep.

Each loop index takes on $\le n$ values.

Hence the time complexity is $O(n^3)$. Space complexity $\Theta(n^2)$.