

## **UNIT IV – DIVIDE & CONQUER, GREEDY AND DYNAMIC PROGRAMMING**

Divide and conquer: Merge sort - Quick sort - Binary search. Greedy method: Knapsack problem- Job sequencing with deadlines - Minimum cost spanning tree – Single source shortest path. dynamic programming: All pair shortest path - Knapsack problem – Traveling salesman problem - Flow shop scheduling.

---

### **DYNAMIC PROGRAMMING**

Dynamic Programming is an algorithmic technique used in computer science and mathematics to solve complex problems by breaking them down into smaller overlapping subproblems. The core idea behind DP is to store solutions to subproblems so that each is solved only once.

To solve DP problems, we first write a recursive solution in a way that there are overlapping subproblems in the recursion tree (the recursive function is called with the same parameters multiple times).

To make sure that a recursive value is computed only once (to improve time taken by algorithm), we store results of the recursive calls.

There are two ways to store the results, one is top down (or memoization) and other is bottom up (or tabulation).

#### **All pair shortest path**

The Floyd-Warshall algorithm, named after its creators Robert Floyd and Stephen Warshall, is a fundamental algorithm in computer science and graph theory. It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both positive and negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems.

The Floyd Warshall Algorithm is an all pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). It follows a Dynamic Programming approach to check every possible path going via every possible node in order to calculate shortest distance between every pair of nodes.

#### **Floyd Warshall Algorithm:**

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.

- The idea is to pick all vertices one by one and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number  $k$  as an intermediate vertex, we already have considered vertices  $\{0, 1, 2, \dots, k-1\}$  as intermediate vertices.
- For every pair  $(i, j)$  of the source and destination vertices respectively, there are two possible cases.
  - $k$  is not an intermediate vertex in the shortest path from  $i$  to  $j$ . We keep the value of  $\text{dist}[i][j]$  as it is.
  - $k$  is an intermediate vertex in shortest path from  $i$  to  $j$ . We update the value of  $\text{dist}[i][j]$  as  $\text{dist}[i][k] + \text{dist}[k][j]$ , if  $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$

### **Pseudo-Code of Floyd Warshall Algorithm :**

For  $k = 0$  to  $n - 1$

    For  $i = 0$  to  $n - 1$

        For  $j = 0$  to  $n - 1$

$\text{Distance}[i, j] = \min(\text{Distance}[i, j], \text{Distance}[i, k] + \text{Distance}[k, j])$

where  $i =$  source Node,  $j =$  Destination Node,  $k =$  Intermediate Node

**Example:** [Follow class notes]

**Time Complexity:**  $O(V^3)$ , where  $V$  is the number of vertices in the graph and we run three nested loops each of size  $V$

**Auxiliary Space:**  $O(V^2)$ , to create a 2-D matrix in order to store the shortest distance for each pair of nodes.

### **KNAPSACK PROBLEM**

Given  $N$  items where each item has some weight and profit associated with it and also given a bag with capacity  $W$ , [i.e., the bag can hold at most  $W$  weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible. The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Follow the below steps to solve the problem:

The maximum value obtained from ' $N$ ' items is the max of the following two values.

- Case 1 (include the  $N$ th item): Value of the  $N$ th item plus maximum value obtained by remaining  $N-1$  items and remaining weight i.e. ( $W$ -weight of the  $N$ th item).
- Case 2 (exclude the  $N$ th item): Maximum value obtained by  $N-1$  items and  $W$  weight.

- If the weight of the 'Nth' item is greater than 'W', then the Nth item cannot be included and Case 2 is the only possibility.

**Illustration:**

```
#include <stdio.h>
int max(int a, int b) { return (a > b) ? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
    if (n == 0 || W == 0)
        return 0;
    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);
    else
        return max(
            val[n - 1]
            + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}
int main()
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    printf("%d", knapSack(W, weight, profit, n));
    return 0;
}
```

**Example:** [Follow class notes]

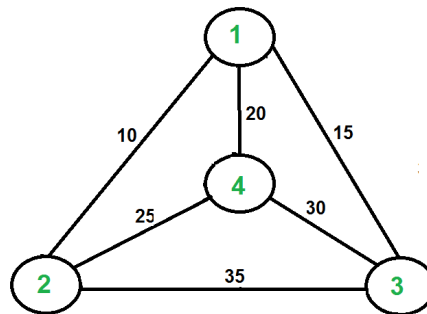
**Time Complexity:**  $O(N * W)$ . As redundant calculations of states are avoided.

**Auxiliary Space:**  $O(N * W) + O(N)$ . The use of a 2D array data structure for storing intermediate states and  $O(N)$  auxiliary stack space(ASS) has been used for recursion stack

**TRAVELING SALESMAN PROBLEM**

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we

know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is  $10+25+30+15$  which is 80. The problem is a famous NP hard problem. There is no polynomial-time known solution for this problem. The following are different solutions for the traveling salesman problem.

#### Naive Solution:

- 1) Consider city 1 as the starting and ending point.
- 2) Generate all  $(n-1)!$  Permutations of cities.
- 3) Calculate the cost of every permutation and keep track of the minimum cost permutation.
- 4) Return the permutation with minimum cost.

Let the given set of vertices be  $\{1, 2, 3, 4, \dots, n\}$ . Let us consider 1 as the starting and ending point of output. For every other vertex  $i$  (other than 1), we find the minimum cost path with 1 as the starting point,  $i$  as the ending point, and all vertices appearing exactly once. Let the cost of this path be  $\text{cost}(i)$ , and the cost of the corresponding Cycle would be  $\text{cost}(i) + \text{dist}(i, 1)$  where  $\text{dist}(i, 1)$  is the distance from  $i$  to 1. Finally, we return the minimum of all  $[\text{cost}(i) + \text{dist}(i, 1)]$  values.

**Time Complexity:**  $O(n^2 \cdot 2^n)$  where  $O(n^2 \cdot 2^n)$  are the maximum number of unique subproblems/states and  $O(n)$  for transition (through for loop as in code) in every state.

**Auxiliary Space:**  $O(n^2 \cdot 2^n)$ , where  $n$  is the number of Nodes/Cities here.

---

## FLOW-SHOP SCHEDULING

Flow-shop scheduling is an optimization problem in computer science and operations research. It is a variant of optimal job scheduling. In a general job-scheduling problem, we are given  $n$  jobs  $J_1, J_2, \dots, J_n$  of varying processing times, which need to be scheduled on  $m$  machines with varying processing power, while trying to minimize the makespan – the total length of the schedule (that is, when all the jobs have finished processing). In the specific variant known as flow-shop scheduling, each job contains exactly  $m$  operations. The  $i$ -th operation of the job must be executed

on the  $i$ -th machine. No machine can perform more than one operation simultaneously. For each operation of each job, execution time is specified.

There are  $m$  machines and  $n$  jobs. Each job contains exactly  $m$  operations. The  $i$ -th operation of the job must be executed on the  $i$ -th machine. No machine can perform more than one operation simultaneously. For each operation of each job, execution time is specified.

Operations within one job must be performed in the specified order. The first operation gets executed on the first machine, then (as the first operation is finished) the second operation on the second machine, and so on until the  $m$ -th operation. Jobs can be executed in any order, however. Problem definition implies that this job order is exactly the same for each machine. The problem is to determine the optimal such arrangement, i.e. the one with the shortest possible total job execution makespan.

**Algorithm:**

Algorithm JOHNSON\_FLOWSHOP( $T, Q$ )

$Q = \Phi$

for  $j = 1$  to  $n$  do

$t =$  minimum machine time scanning in booth columns

    if  $t$  occurs in column 1 then

        Add Job  $j$  to the first empty slot of  $Q$

    else

        Add Job  $j$  to last empty slot of  $Q$

    end

    Remove processed job from consideration

end

return  $Q$

**Example:** [Follow class notes]

Two-machine case: The two-machine permutation flow-shop case can be solved in  $O(n \log n)$  time

---