Experiment No: 03

Aim: Implementation of Single source shortest path using Dijkstra algorithm and its analysis.

Theory:

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

DIJKSTRA (G, w, s)

```
1.  INITIALIZE SINGLE-SOURCE (G, s)
2.  S ← { }     // S will ultimately contains vertices of final shortest-
     path weights from s
3.  Initialize priority queue Q i.e., Q ←  V[G]
4.  while priority queue Q is not empty do
5.      u  ←  EXTRACT_MIN(Q)    // Pull out new  vertex
6.      S  ←  S È {u}                    // Perform relaxation for each vertex  v adjacent to u
7.      for each vertex v in Adj[u] do
8.          Relax (u, v, w)
```

ANALYSIS

Like Prim's algorithm, Dijkstra's algorithm runs in O(|E|lg|V|) time.

Conclusion: Thus we implemented the Single source shortest path using Dijkstra algorithm.

Experiment No: 04

Aim: To implement Fractional knapsack algorithm using dynamic programming.

Theory: Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.
Problem Scenario
A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of ith item is wi and its profit is pi. What items should the thief take?
In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.
Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack
In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.
According to the problem statement,

- There are n items in the store
- Weight of $\mathbf{i^{th}}$ item wi>0wi>0
- Profit for $\mathbf{i^{th}}$ item pi>0pi>0 and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction xi of $\mathbf{i^{th}}$ item.

$$0 \leqslant xi \leqslant 10 \leqslant xi \leqslant 1$$

The $\mathbf{i^{th}}$ item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit.
Hence, the objective of this algorithm is to subject to constraint,

$$\sum n=1n(xi.wi) \leqslant W \sum n=1n(xi.wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.
Thus, an optimal solution can be obtained by

$$\sum n=1n(xi.wi)=W \sum n=1n(xi.wi)=W$$

In this context, first we need to sort those items according to the value of piwipiwi, so that pi+1wi+1pi+1wi+1 ≤ piwipiwi . Here, x is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
   do x[i] = 0
weight = 0
for i = 1 to n
   if weight + w[i] ≤ W then
      x[i] = 1
      weight = weight + w[i]
  else
       x[i] = (W - weight) / w[i]
       weight = W
       break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn).***

Conclusion: The fractional knapsack problem is implemented using dynamic programming, which is used to fill the knapsack with items to get maximum benefit (value or profit) without crossing the weight capacity of the knapsack. And we are also allowed to take an item in fractional part.