Lab 4: Implementation of Dynamic Programming using C++

Objective:
To implement and analyze two classic greedy algorithms, 0/1 Knapsack Algorithm and Single Source Shortest Path Problem. These algorithm are chosen to demonstrate the efficiency and practical applications of dynamic programming approach for solving problems.

Theory:
Dynamic Programming (DP) is an algorithmic technique that solves complex problems by breaking them down into simpler subproblems. Its particularly useful for optimization problems.

1. <u>0/1 Knapsack Problem:</u> The 0/1 Knapsack Problem is a classic optimization problem where we need to select items to maximize value while staying within a weight limit. Unlike the Fractional Knapsack Problem, items cannot be divided, they are either excluded or included.
   <u>Algorithm :</u>

   Input : array of items with value and weight
   Output: Maximum value
   a. N := Length of items.
   b. Create a 2d array dp[n+1][max_weight+1] initialized with all 0.
   c. For I from 1 to n
      i. For w from 0 to max_weight
         1. If items[i-1].weight <= w then
         2. Dp[i][w] := max(dp[i-1][w],dp[i-1][w-items[i-1].weight] + items[i-1].value)
         3. Else
         4. dp[i][w] := dp[i-1][w]
   d. return dp[n][W
2. <u>Single Source Shortest path Problem:</u> This problem involves finding the shortest path from a source node to every other node in a weighted graph, where the weight can also be negative. It detects negative cycles and stops the algorithm.
   <u>Algorithm:</u>

   Input : number of edges (E), number of vertices (V), set of edges where edge has source, destination and weight.

   Output: vertices and minimum distance to each vertex

a. Initialize dist[V] with dist[src] = 0 and rest Infinity
b. For I in 0 to v-1
    i. For j in 0 to E
        1. u = edges[j].src
        2. v = edges[j].dest
        3. weight = g.edges[j].weight;
        4. if dist[u] != Infinity and dist[u] + weight < dist[v]
        5. dist[v] = dist[u] + weight b.
c. For i in 0 to e i. u = edges[i].src
    i. v = edges[i].dest
    ii. weight = edges[i].weight
    iii. if desit[u] != Infinity and dist[u] + weight < dist[v]
    iv. print “Negative weight cycle”
    v. exit c. for i in 0 to
    vi. print “vertex” i “distance = “ dist[i]
    vii. exit
d. for i in 0 to v
    i. print “vertex” i “distance = “ dist[i]
e. exit

Observation:

1. 0/1 Knapsack Problem:

```
#include <iostream>
#include <functional>

using namespace std;

class KnapSack{
  public:

  int knapSackRec(int W, int wt[], int val[], int index, int** dp){
    if(index < 0 || W == 0){
      return 0;
    }

    if(dp[index][W] != -1){
      return dp[index][W];
    }

    if(wt[index] > W){
      return dp[index][W] = knapSackRec(W, wt, val, index - 1, dp);
    }else{
      dp[index][W] = max(val[index] + knapSackRec(W - wt[index], wt, val, index - 1,
dp), knapSackRec(W, wt, val, index - 1, dp));
      return dp[index][W];
    }
  }

  int knapSack(int W, int wt[], int val[], int n){
    int** dp;
    dp = new int*[n];

    for (int i = 0; i < n; i++){
      dp[i] = new int[W + 1];
    }

    for (int i = 0; i < n; i++){
      for (int j = 0; j < W + 1; j++){
        dp[i][j] = -1;
      }
    }
    return knapSackRec(W, wt, val, n - 1, dp);
  }
};
```

```cpp
int main(){
    int size = 1000;
    for (int i = 0; i < 5; i++){
        int profit[size];
        int weight[size];
        for (int i = 0; i < size; i++){
            auto v = rand() % 10;
            auto w = rand() % 10;
            profit[i] = v;
            weight[i] = w;
        }
        int n = sizeof(profit) / sizeof(profit[0]);
        int W = 10;
        KnapSack knapSack;
        auto knap = [&](){
            knapSack.knapSack(W, weight, profit, n);
        };
        cout << "Time taken to process " << size << " elements: " << getTime(knap) << "ns" << endl;
        size += 500;
    }
}
```

Output:

```
● ➜ lab4 git:(main) ✗ g++ Knapsack.cpp -o Knapsack.out
● ➜ lab4 git:(main) ✗ ./Knapsack.out
  Time taken to process 1000 elements: 261ns
  Time taken to process 1500 elements: 344ns
  Time taken to process 2000 elements: 488ns
  Time taken to process 2500 elements: 611ns
  Time taken to process 3000 elements: 697ns
```

2. Single source shortest problem:

```
#include <iostream>
#include <bits/stdc++.h>

using namespace std;


struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

void printArr(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
```

```
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX
                && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX
            && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle");
            return;
        }
    }

    printArr(dist, V);

    return;
}
int main()
{
    int V = 5;
    int E = 8;
    struct Graph* graph = createGraph(V, E);

    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    graph->edge[4].src = 1;
```

```
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;

    auto bellman = [&](){
        BellmanFord(graph, 0);
    };

    cout << getTime(bellman) << "ns time taken" << endl;
    return 0;
}
```

Output:

```
● ➜ lab4 git:(main) ✗ g++ Bellman_ford.cpp -o Bellman_ford.out
● ➜ lab4 git:(main) ✗ ./Bellman_ford.out
  Vertex   Distance from Source
  0                0
  1                -1
  2                2
  3                -2
  4                1
  32ns time taken
```

Conclusion:

We solved fractional knapsack problem and single source shortest algorithm by applying dynamic programming in C++.