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. 0/1 Knapsack Problem: 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. Single Source Shortest path Problem: 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. For i in 0 to e
    i. u = edges[i].src
    ii. v = edges[i].dest
    iii. weight = edges[i].weight
    iv. if desit[u] != Infinity and dist[u] + weight < dist[v]
    v. print “Negative weight cycle”
    vi. exit
  c. for i in 0 to v
    i. print “vertex” i “distance = “ dist[i]
  d. exit

Observation

1. 0/1 Knapsack Problem:

```
#include <iostream>
#include "gettime.h"
#include <vector>
#include <algorithm>
using namespace std;

int knapsack(int max_w, int values[], int weights[], int n)
{
    vector<vector<int>> matrix(n + 1, vector<int>(max_w + 1, 0));
    for (int i = 1; i <= n; i++)
    {
        for (int w = 0; w <= max_w; w++)
        {
            int current_value = values[i - 1];
            int current_weight = weights[i - 1];
            if (current_weight <= w)
            {
                matrix[i][w] = max(current_value + matrix[i - 1][w -
current_weight], matrix[i - 1][w]);
            }
            else
            {
                matrix[i][w] = matrix[i - 1][w];
            }
        }
    }
    return matrix[n][max_w];
}

int main()
{
    vector<int> sizes;
    int start_size = 1000;
    int increment = 500;
    sizes.push_back(start_size);
    for (size_t i = 0; i < 4; i++)
    {
        sizes.push_back(sizes.back() + increment);
    }
    srand(time(0));
    for (int i = 0; i < sizes.size(); i++)
    {
```

```
        int size = sizes[i];
        int *values = new int[size];
        int *weights = new int[size];
        for (int i = 0; i < size; i++)
        {
            values[i] = rand() % 1000;
            weights[i] = rand() % 1000;
        }
        int max_w = 100;
        long long time = getTime([&]()
                                 { knapsack(max_w, values, weights, size); });
        cout << "Size: " << size << " Time: " << time << endl;
    }
    return 0;
}
```

Output:

```
Size: 1000 Time: 35054700
Size: 1500 Time: 138440900
Size: 2000 Time: 254552900
Size: 2500 Time: 340822000
Size: 3000 Time: 397872500
```

2. Single source shortest problem:

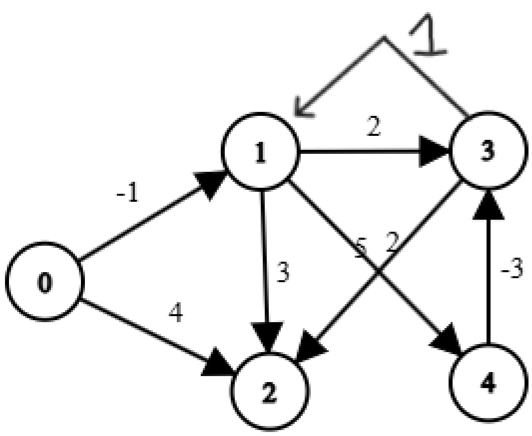
```
#include <iostream>
#include <vector>
#include <algorithm>
#include "gettime.h"
using namespace std;

class Edge
{
public:
    int src, dest, weight;
    Edge()
    {
    }
    Edge(int src, int dest, int weight)
    {
        this->src = src;
        this->dest = dest;
```

```cpp
        this->weight = weight;
    }
};

class Graph
{
public:
    int V, E;
    Edge *edges;
    Graph(int V, int E)
    {
        this->V = V;
        this->E = E;
        this->edges = new Edge[E];
    }
};

void bellman_ford(Graph g, int src)
{
    int v = g.V;
    int e = g.E;
    int *dist = new int[v];
    for (int i = 0; i < v; i++)
    {
        dist[i] = INT_MAX;
    }
    dist[src] = 0;
    for (int i = 0; i < v - 1; i++)
    {
        for (int j = 0; j < e; j++)
        {
            int u = g.edges[j].src;
            int v = g.edges[j].dest;
            int weight = g.edges[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 = g.edges[i].src;
        int v = g.edges[i].dest;
        int weight = g.edges[i].weight;
```

```
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
        {
            cout << "Negative weight cycle " << endl;
            return;
        }
    }

    for (int i = 0; i < v; i++)
    {
        cout << i << "\t\t " << dist[i] << endl;
    }
}

int main()
{
    int V = 5;
    int E = 8;
    Graph graph(V, E);

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

    graph.edges[1].src = 0;
    graph.edges[1].dest = 2;
    graph.edges[1].weight = 4;

    graph.edges[2].src = 1;
    graph.edges[2].dest = 2;
    graph.edges[2].weight = 3;

    graph.edges[3].src = 1;
    graph.edges[3].dest = 3;
    graph.edges[3].weight = 2;

    graph.edges[4].src = 1;
    graph.edges[4].dest = 4;
    graph.edges[4].weight = 2;

    graph.edges[5].src = 3;
    graph.edges[5].dest = 2;
    graph.edges[5].weight = 5;

    graph.edges[6].src = 3;
    graph.edges[6].dest = 1;
```

```
    graph.edges[6].weight = 1;

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

    long long time = getTime([&]()
                             { bellman_ford(graph, 0); });
    cout << "time = " << time << " ns";
}
```

Input:



Output:

```
0                0
1                -1
2                2
3                -2
4                1
time = 3988800 ns
```

Conclusion

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