|  |  |  |  |
| --- | --- | --- | --- |
| **Academic Year** | **2025 - 26** | **Experiment No.** | **6 & 7** |
| **Course & Semester** | **S.E. – Sem. III** | **Subject Name** | **Analysis of Algorithm** |
| **Experiment Type** | **Software Performance** | **Subject Code** | **25PCC12CS05** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name of Student:** | Atharva Dharmendra Jagtap | **Roll No.:** | 10937 |
| **Date of Performance:** | 05-10-2025 | **Date of Submission:** | 05-10-2025 |
| **LO Mapping** | 25PCC12CS05.1: Analyze the time and space complexity of algorithms.  25PCC12CS05.4: Apply dynamic programming strategy to solve optimization problem. | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Indicator** | **Poor** | **Average** | **Good** | | Timeline Maintains submission deadline (3) | Submission not done (0) | One or More than One week late (1-2) | Maintains deadline (3) | | Completion and Organization (3) | N/A | Document is just acceptable (1-2) | Completed whole document and neatly organized (3) | | Program Performance (2) | Could not perform at all (0) | Implemented few parts (1) | Full implementation (2) | | Knowledge In depth knowledge of the Experiment (2) | Unable to answer questions (0) | Unable to answer few questions (1) | Able to answer all questions (2) | |
| **Assessment Marks:**   |  |  | | --- | --- | | Timeline |  | | Completion and Organization |  | | Program Performance |  | | Knowledge |  | |
| Total: (Out of 10) |
| Teacher’s Sign: Student Sign: |

**Experiment No. 6 & 7**

**AIM:** Identify and implement an algorithm to be used in disaster management and emergency response systems to find the shortest path for emergency vehicles, such as ambulances or fire trucks, to reach affected areas or victims.

Identify and implement an algorithm to be used to compare DNA/RNA sequences to identify similarities and evolutionary relationships between organisms.

**THEORY:**

Dynamic programming (DP) is a powerful algorithmic technique used to solve complex optimization problems by breaking them down into simpler subproblems. It is particularly useful for problems that involve making decisions in stages, where the optimal solution to the overall problem can be constructed from the optimal solutions to smaller subproblems.

The core idea of dynamic programming is to store the solutions to subproblems in a table (memoization), avoiding redundant calculations and improving efficiency. This approach transforms problems with overlapping subproblems and optimal substructure into solvable instances with reduced time complexity.

In real-world applications, dynamic programming can be applied to various optimization problems, such as finding the shortest path in transportation systems (e.g., for emergency vehicles or urban planning), aligning DNA/RNA sequences in bioinformatics, and optimizing resource allocation for more efficient operations.

DP algorithms generally have polynomial time complexity, making them suitable for solving large-scale problems where brute-force methods would be inefficient. Through the systematic approach of DP, we can find optimal solutions to problems involving decision-making, pathfinding, and resource optimization.

**ALGORITHM:**

**Bellman-Ford Algorithm**

**Input:**

* Read the number of vertices (V) and edges (E) in the road network graph.
* Read the edge list: each edge contains (source, destination, weight), where weight represents time or distance.
* Read the source node (e.g., ambulance station).

**Initialize:**

* Set distance[] array of size V with all values as ∞ (infinity).
* Set distance[source] = 0.

**Relax Edges:**

* Repeat V−1 times:
* For each edge (u, v, w):
* If distance[u] + w < distance[v], then update distance[v] = distance[u] + w.

**Check for Negative Cycles:**

* For each edge (u, v, w):
* If distance[u] + w < distance[v], report a negative weight cycle (not applicable in real-world road networks, but useful for validation).

**Output:**

* Print the shortest distance from source to all other vertices.
* Optionally, reconstruct and print the shortest path to a specific destination

**Longest Common Subsequence (LCS) for DNA/RNA Comparison**

**Input:**

* Read two sequences: seq1 and seq2 (DNA or RNA strings).

**Initialize:**

* Create a 2D matrix dp[][] of size (len(seq1)+1) × (len(seq2)+1).
* Initialize all cells to 0.

**Fill the Matrix:**

For i from 1 to len(seq1):

For j from 1 to len(seq2):

If seq1[i−1] == seq2[j−1]:

dpi][j] = dpi−1][j−1] + 1

Else:

dpi][j] = max(dpi−1][j], dpi][j−1])

**Traceback (Optional):**

* Start from dplen(seq1)][len(seq2)] and trace back to reconstruct the actual LCS string.

**Output:**

* Print the length of the longest common subsequence.
* Optionally, print the LCS string itself.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#define INF 99999

// ---------- Bellman-Ford Algorithm ----------

struct Edge {

    int u, v, w;

};

void bellmanFord(int V, int E, struct Edge edges[], int src) {

    int distance[V];

    // Initialize distances

    for (int i = 0; i < V; i++)

        distance[i] = INF;

    distance[src] = 0;

    // Relax edges V-1 times

    for (int i = 1; i <= V - 1; i++) {

        for (int j = 0; j < E; j++) {

            int u = edges[j].u;

            int v = edges[j].v;

            int w = edges[j].w;

            if (distance[u] != INF && distance[u] + w < distance[v])

                distance[v] = distance[u] + w;

        }

    }

    // Check for negative cycles

    for (int j = 0; j < E; j++) {

        int u = edges[j].u;

        int v = edges[j].v;

        int w = edges[j].w;

        if (distance[u] != INF && distance[u] + w < distance[v]) {

            printf("Graph contains a negative weight cycle!\n");

            return;

        }

    }

    // Output shortest distances

    printf("\nShortest distances from source %d:\n", src);

    for (int i = 0; i < V; i++)

        printf("Vertex %d: %d\n", i, distance[i]);

}

// ---------- Longest Common Subsequence (LCS) ----------

int max(int a, int b) {

    return (a > b) ? a : b;

}

void LCS(char seq1[], char seq2[]) {

    int m = strlen(seq1);

    int n = strlen(seq2);

    int dp[m + 1][n + 1];

    // Initialize dp table

    for (int i = 0; i <= m; i++)

        for (int j = 0; j <= n; j++)

            dp[i][j] = 0;

    // Fill dp matrix

    for (int i = 1; i <= m; i++) {

        for (int j = 1; j <= n; j++) {

            if (seq1[i - 1] == seq2[j - 1])

                dp[i][j] = dp[i - 1][j - 1] + 1;

            else

                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);

        }

    }

    // Output LCS length

    printf("\nLength of Longest Common Subsequence: %d\n", dp[m][n]);

    // Traceback to reconstruct LCS

    int index = dp[m][n];

    char lcs[index + 1];

    lcs[index] = '\0';

    int i = m, j = n;

    while (i > 0 && j > 0) {

        if (seq1[i - 1] == seq2[j - 1]) {

            lcs[index - 1] = seq1[i - 1];

            i--;

            j--;

            index--;

        } else if (dp[i - 1][j] > dp[i][j - 1])

            i--;

        else

            j--;

    }

    printf("LCS: %s\n", lcs);

}

// ---------- Main Function ----------

int main() {

    int V, E, src;

    printf("Enter number of vertices and edges: ");

    scanf("%d %d", &V, &E);

    struct Edge edges[E];

    printf("Enter edges (source destination weight):\n");

    for (int i = 0; i < E; i++)

        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].w);

    printf("Enter source vertex: ");

    scanf("%d", &src);

    bellmanFord(V, E, edges, src);

    char seq1[100], seq2[100];

    printf("\nEnter first DNA/RNA sequence: ");

    scanf("%s", seq1);

    printf("Enter second DNA/RNA sequence: ");

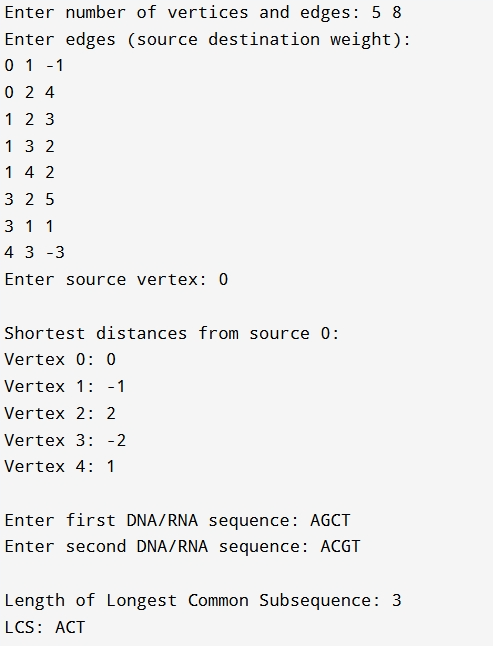
    scanf("%s", seq2);

    LCS(seq1, seq2);

    return 0;

}

**OUTPUT:**

****

**POST LAB QUESTIONS**

1. Why do we use a table (or matrix) in dynamic programming algorithms like LCS and Bellman-Ford? What advantage does it give compared to solving the problem without it?

**Ans.** The table stores intermediate results to avoid recalculating subproblems.

* Advantages:
  + Eliminates redundant computations.
  + Reduces time complexity from exponential to polynomial.
  + Enables reconstruction of optimal paths or subsequences.

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Table Purpose** | **Benefit** |
| Bellman-Ford | Stores shortest distances | Detects shorter paths iteratively |
| LCS | Stores length of common subsequences | Builds longest subsequence efficiently |

1. What does the Bellman-Ford algorithm do when it finds a shorter path to a node? Why does it repeat the process multiple times?

**Ans.** Each iteration propagates shorter paths one edge further. Repeating **V−1 times** ensures all shortest paths are discovered (since the longest possible simple path has V−1 edges). If another update occurs after V−1 iterations → **negative cycle exists**.

1. If two DNA sequences are "AGCT" and "ACGT", what is their longest common subsequence? How does the LCS algorithm help us find it

**Ans.** For sequences "AGCT" and "ACGT", the LCS is:

* Comparing character by character:

A G C T

A C G T

* Matching sequence: A–G–T
* Hence, LCS = "AGT", Length = 3

How LCS helps:

* Quantifies genetic similarity between DNA/RNA sequences.
* Used in bioinformatics for **mutation detection**, **evolutionary studies**, and **gene alignment**.

**CONCLUSION:**

The Bellman-Ford algorithm effectively computes shortest paths in weighted graphs, even handling negative weights, through systematic edge relaxation.

The LCS algorithm leverages dynamic programming to efficiently identify genetic similarity between sequences. Both methods demonstrate the power of tabulation in breaking complex problems into manageable subproblems, ensuring optimal and reliable solutions in real-world applications like routing and bioinformatics.