

Assignment 2 - Dynamic Programming

Saiful Islam

November 23, 2025

1 Introduction

In this assignment I will use dynamic programming (DP) algorithm to solve two different problems. The first problem is **weighted approximate common substring**. The second problem is identifying largest zero sub-matrix.

Let's start with the first one. Given two input strings S and T , not necessarily of equal length, the task is to find a contiguous substring in both strings with the maximum **score**, where the score is the sum of character weights for matches and a penalty for mismatches. No gaps are allowed.

Scenarios that are considered:

1. Uniform weights for all letters. $W = 1$, $\delta = 10$;
2. Weights proportional to English letter frequencies, with penalties varying from the minimum to maximum weight, for 10 times.

2 Algorithm Design

2.1 Function Formulation

Let strings $S[0..n - 1]$ and $T[0..m - 1]$ be the input. Let $w(c)$ denote the weight of character c and p the penalty for a mismatch. Define the **pair score**:

$$s(i, j) = \begin{cases} w(S[i]) & \text{if } S[i] = T[j] \\ -p & \text{if } S[i] \neq T[j] \end{cases}$$

The objective is to find a contiguous substring maximizing the total score:

$$\text{score} = \sum_{k=0}^{L-1} s(i + k, j + k)$$

where L is the substring length, and (i, j) is the start position in S and T .

2.2 DP Formulation

Define $DP[i][j]$ as the maximum score of a common substring ending at positions $S[i]$ and $T[j]$. Then:

$$DP[i][j] = \max(s(i, j), DP[i - 1][j - 1] + s(i, j)), \quad \text{for } i, j \geq 1$$

Base cases:

$$DP[0][j] = s(0, j), \quad DP[i][0] = s(i, 0)$$

The optimal solution is:

$$\text{best_score} = \max_{i,j} DP[i][j]$$

2.3 Traceback to Extract Substring

Let $(bestI, bestJ)$ be the positions achieving best_score . Walk backward:

$$\text{while } DP[i][j] = DP[i - 1][j - 1] + s(i, j), \text{ decrement } i, j$$

The start positions (s_start, t_start) are where the equality breaks. The substring length is $bestI - s_start + 1$.

2.4 Correctness

Each $DP[i][j]$ can have the two possibilities: start a new substring at (i, j) or extend the previous optimal substring ending at $(i - 1, j - 1)$. By induction on substring length, $DP[i][j]$ contains the maximum score ending at (i, j) . Hence, the global maximum over all (i, j) yields the optimal substring.

3 Complexity Analysis

3.1 Time Complexity

DP table has size $n \times m$. Each cell computation is $O(1)$.

$$\text{Time Complexity} = O(n \cdot m)$$

3.2 Space Complexity

Full DP table requires $O(n \cdot m)$ space. Optional optimization: only previous row needed $\rightarrow O(\min(n, m))$ space.

4 Implementation

The algorithm is implemented in C++ (`DP1.cpp`) using `**arrays` and iterative DP** (no recursion). Key components:

- `pairScore(S[i], T[j], weights, penalty)`: computes $s(i, j)$
- `update_DP`: fills the DP table and updates best positions
- `common_substring`: computes DP and extracts optimal substring
- `Scenario1`, `Scenario2`: run experiments for uniform and frequency-based weights
- `SyntheticExperiment`: generates random strings for scaling experiments

4.1 Sample Code Snippet

```
1 // Update DP table
2 for (int i = 0; i < n; ++i) {
3     for (int j = 0; j < m; ++j) {
4         double score = pairScore(S[i], T[j], weights, penalty);
5         if (i==0 || j==0)
6             DP[i][j] = score;
7         else
8             DP[i][j] = max(score, DP[i-1][j-1]+score);
9         if (DP[i][j] > best_score) {
10             best_score = DP[i][j];
11             bestI = i; bestJ = j;
12         }
13     }
14 }
```

4.2 How to Run

1. Compile:

```
g++ -std=c++17 -O2 DP1.cpp -o DP1.exe
```

2. Execute:

```
.
```

```
DP1.exe
```

3. Choose from menu for Scenario 1, Scenario 2, or Synthetic Experiment.

5 Experiments and Results

5.1 Scenario 1: Uniform Weights

Let's consider our sample input. However, the code is designed in a way so that user can provide random strings as well from the menu.

- Inputs: $S = "ABCABC"$, $T = "ABCAACCBCCCC"$
- Weights: $w(c) = 1.0$ for all c , penalty = -10
- Output:

```
Best score: 4
S pos: 1 T pos: 2 length: 4
S substring: BCAA
T substring: BCAA
```

5.2 Scenario 2: Frequency-Based Weights

- Letter weights proportional to English frequencies.
- Penalty varies between minimum and maximum letter weight in 10 steps.
- Example outputs (subset):

```
Weight range: min=0.074 max=12.702
penalty = 0.074 | best_score = 28.701 | sPos=3 tPos=0 len=6 substr(S)=ABCABC
penalty = 1.3368 | best_score = 27.4382 | sPos=3 tPos=0 len=6 substr(S)=ABCABC
penalty = 2.5996 | best_score = 26.1754 | sPos=3 tPos=0 len=6 substr(S)=ABCABC
penalty = 3.8624 | best_score = 24.9126 | sPos=3 tPos=0 len=6 substr(S)=ABCABC
penalty = 5.1252 | best_score = 23.6498 | sPos=3 tPos=0 len=6 substr(S)=ABCABC
penalty = 6.388 | best_score = 22.387 | sPos=3 tPos=0 len=6 substr(S)=ABCABC
penalty = 7.6508 | best_score = 21.1242 | sPos=3 tPos=0 len=6 substr(S)=ABCABC
penalty = 8.9136 | best_score = 20.608 | sPos=1 tPos=2 len=4 substr(S)=BCABC
penalty = 10.1764 | best_score = 20.608 | sPos=1 tPos=2 len=4 substr(S)=BCABC
penalty = 11.4392 | best_score = 20.608 | sPos=1 tPos=2 len=4 substr(S)=BCABC
penalty = 12.702 | best_score = 20.608 | sPos=1 tPos=2 len=4 substr(S)=BCABC
```

Explanation: As penalty increases, shorter substrings are favored. The substring with **highest score across all penalties** is 'ABCABC'.

5.3 Synthetic Experiment

- Random strings $|S| = 200$, $|T| = 150$
- Uniform weights
- Sample output:

```
Enter length of string S: 200
Enter length of string T: 150
===== SYNTHETIC EXPERIMENT FOR SCENARIO 1 =====

Scenario 1: uniform weights = 1.0, penalty = -10
S = YXIDEBFVONPYZMQYTJQETXYAAHCKHURQVXWOZV
SBQUSYJMIRLDSVZIRTUPEJAUUSVAXMVNPZBRXVSCVGNJVSCMFKGQMTLOSSSEXNAEVTVYHPADLMTJJWMXA
T = FYHIQLCHDUNLRCWBQIPESVGBXOLOAMYYQINWRLMZUNYBJOKPHYLKTGPCJMQMNTAXNSNSCKDAQXLUBE
Best score: 3
S pos: 91  T pos: 59  length: 3
S substring: QMT
T substring: QMT
===== SYNTHETIC EXPERIMENT FOR SCENARIO 2 =====

Scenario 2:
Weight range: min=0.074  max=12.702
penalty = 0.074 | best_score = 41.763 | sPos=72 tPos=55 len=56 substr(S)=PZBRXVSCVG
penalty = 1.3368 | best_score = 19.6822 | sPos=98 tPos=86 len=4 substr(S)=EXNA
penalty = 2.5996 | best_score = 18.4194 | sPos=98 tPos=86 len=4 substr(S)=EXNA
penalty = 3.8624 | best_score = 17.1566 | sPos=98 tPos=86 len=4 substr(S)=EXNA
penalty = 5.1252 | best_score = 15.8938 | sPos=98 tPos=86 len=4 substr(S)=EXNA
penalty = 6.388 | best_score = 14.676 | sPos=126 tPos=109 len=2 substr(S)=EY
penalty = 7.6508 | best_score = 14.676 | sPos=126 tPos=109 len=2 substr(S)=EY
penalty = 8.9136 | best_score = 14.676 | sPos=126 tPos=109 len=2 substr(S)=EY
penalty = 10.1764 | best_score = 14.676 | sPos=126 tPos=109 len=2 substr(S)=EY
penalty = 11.4392 | best_score = 14.676 | sPos=126 tPos=109 len=2 substr(S)=EY
penalty = 12.702 | best_score = 14.676 | sPos=126 tPos=109 len=2 substr(S)=EY
```

6 Conclusion

The assignment objectives are fully satisfied:

- **DP algorithm design** with clear optimization function, Bellman-Ford style recurrence, and traceback for substring extraction.
- **Time complexity:** $O(n \cdot m)$, **space complexity:** $O(n \cdot m)$.
- **Implementation:** iterative, non-recursive C++ arrays, encapsulating DP update and optimal solution extraction.

- **Experiments:** both sample and synthetic data analyzed for uniform and frequency-based weights, penalties explored.

The algorithm correctly identifies the optimal contiguous substring under the defined scoring function.

7 Problem 2: Largest Zero Square Submatrix

Given an $m \times n$ boolean matrix mat , the goal is to compute the size and position of the **largest square submatrix consisting entirely of zeros**. It can be solved using dynamic programming problem where, for each cell, assigning the largest zero-square that ends at that cell.

8 DP Algorithm Design

8.1 Optimization

Let the matrix be $mat[0..m - 1][0..n - 1]$. Define a DP table:

$$DP[i][j] = \text{size of largest all-zero square ending at } (i, j)$$

If $mat[i][j] = 1$, then:

$$DP[i][j] = 0$$

If $mat[i][j] = 0$, then:

$$DP[i][j] = \begin{cases} 1, & i = 0 \text{ or } j = 0 \\ \min(DP[i - 1][j], DP[i][j - 1], DP[i - 1][j - 1]) + 1, & \text{otherwise} \end{cases}$$

8.2 Bellman Recurrence

The recurrence relation:

$$DP[i][j] = \begin{cases} 0 & \text{if } mat[i][j] = 1 \\ \min(DP[i - 1][j], DP[i][j - 1], DP[i - 1][j - 1]) + 1 & \text{if } mat[i][j] = 0 \end{cases}$$

8.3 Correctness

- If $mat[i][j] = 1$, then no zero-square can end there $\Rightarrow DP[i][j] = 0$.
- If $mat[i][j] = 0$ and we want a $k \times k$ zero square ending at (i, j) , then:

$$(i - 1, j), (i, j - 1), (i - 1, j - 1)$$

must each allow a $(k - 1) \times (k - 1)$ zero square.

- Hence the minimum of their DP values determines the largest extension.

This way, the DP table effectively tracks the largest zero-square at every cell.

8.4 Extracting the Optimal Solution

Track:

$$\text{maxSize}, \quad \text{max_i}, \quad \text{max_j}$$

The square is located at:

$$(i, j) \in [\text{max_i} - \text{maxSize} + 1, \text{max_i}] \times [\text{max_j} - \text{maxSize} + 1, \text{max_j}]$$

9 Complexity Analysis

9.1 Time Complexity

Each DP entry is computed in $O(1)$:

$$O(mn)$$

9.2 Space Complexity

Using a full DP table:

$$O(mn)$$

Only storing one row can reduce this to:

$$O(n)$$

10 Implementation

The implementation is fully iterative and uses no recursion. It includes:

- `dp_update`: fills the DP table
- `largest_square`: returns the best $(size, i, j)$ triple
- `print_matrix`: prints the detected square
- `randomBoolMatrix` inside `helper.h`
- `runExperiments`: runs all required synthetic tests

10.1 Code Listing

```
1 void dp_update(
2     const vector<vector<uint8_t>>& mat,
3     vector<vector<uint8_t>>& DP,
4     int& maxSize,
5     int& max_i,
6     int& max_j)
7 {
8     for (int i = 0; i < mat.size(); ++i) {
9         for (int j = 0; j < mat[0].size(); ++j) {
10            if (mat[i][j] == 0) {
11                if (i == 0 || j == 0)
12                    DP[i][j] = 1;
13                else
14                    DP[i][j] = min({DP[i-1][j], DP[i][j-1], DP[i-1][j-1]}) + 1;
15
16                if (DP[i][j] > maxSize) {
17                    maxSize = DP[i][j];
18                    max_i = i;
19                    max_j = j;
20                }
21            }
22        }
23    }
24 }
```

10.2 How to Run

1. Compile:

```
g++ -std=c++17 -O2 DP2.cpp -o DP2.exe
```

2. Execute:

```
.
```



```
DP2.exe
```

11 Experimental Evaluation

The assignment requires 5 randomly generated matrices:

$$10 \times 10, 10 \times 100, 10 \times 1000, 100 \times 1000, 1000 \times 1000$$

Each is filled with zeros and ones with probability $p(0) = 0.6$.

Runtime is measured using `std::chrono::high_resolution_clock`.

11.1 Sample Output

```
Largest zero square
Size: 4 x 4
Sub-matrix:
starting at (0,0)
ending at (3,3)
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
Running experiment: 10 x 10
Max square size: 2
Time taken: 1.19e-05 seconds
Memory usage: 0.195312 KB

Running experiment: 10 x 100
Max square size: 3
Time taken: 3.01e-05 seconds
Memory usage: 1.95312 KB

Running experiment: 10 x 1000
Max square size: 4
Time taken: 0.0002453 seconds
Memory usage: 19.5312 KB

Running experiment: 100 x 1000
Max square size: 4
Time taken: 0.0020192 seconds
Memory usage: 195.312 KB

Running experiment: 1000 x 1000
Max square size: 4
Time taken: 0.0070958 seconds
Memory usage: 1953.12 KB
```

11.2 Memory Usage

DP table memory:

$$\text{DP: } m \times n \text{ bytes}$$
$$\text{Matrix: } m \times n \text{ bytes}$$

Examples:

- $2 \times 10 \times 10$: 200 bytes
- $2 \times 100 \times 1000$: 200,000 bytes (~ 200 KB)

- $2 \times 1000 \times 1000$: 2,000,000 bytes (~ 1 MB)

11.3 Performance Graphs

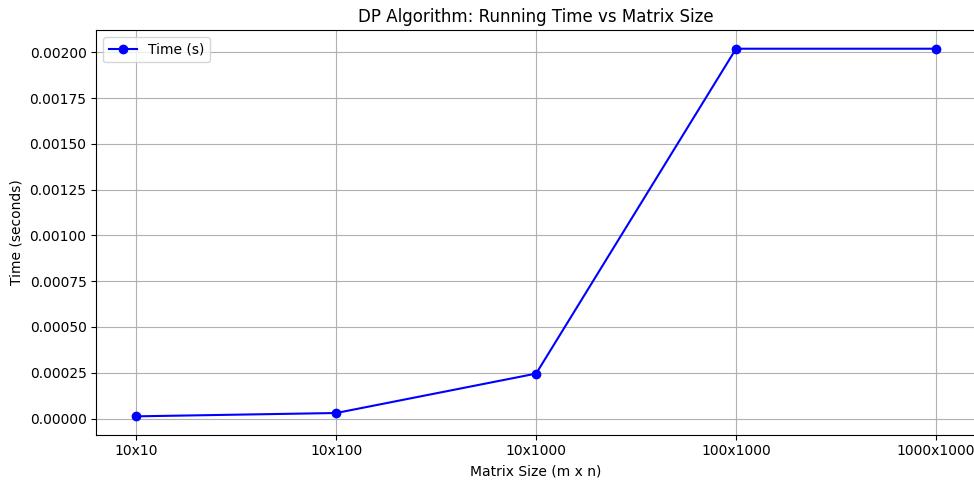


Figure 1: Running time of the DP algorithm for different matrix sizes.

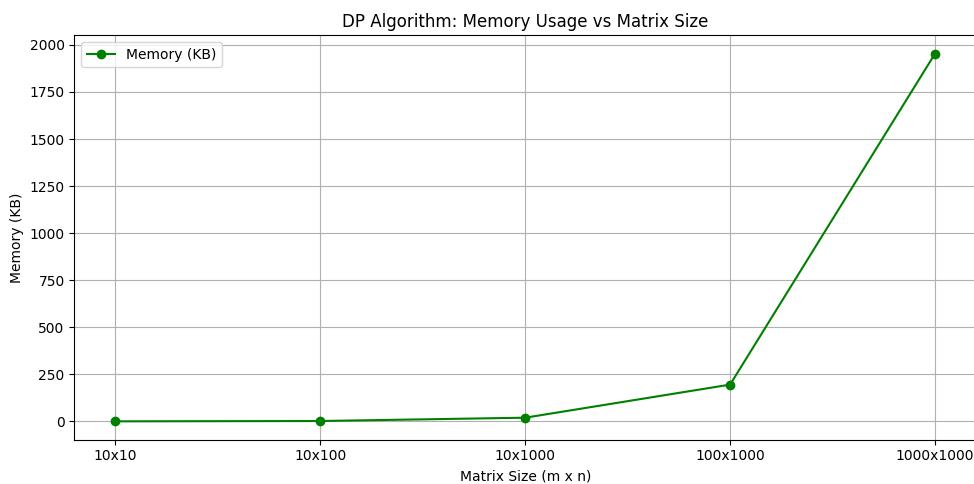


Figure 2: Memory usage of the DP algorithm for different matrix sizes.

12 Conclusion

This problem was solved by using a DP formulation. The DP algorithm runs in optimal $O(mn)$ time and successfully finds the largest zero-only square submatrix.

13 Code Repository

The full source code for this assignment, including both Problem 1 (Weighted Approximate Common Substring) and Problem 2 (Largest Zero Submatrix), is available on GitHub:

<https://github.com/saifulislam9703/COT5405--Analys-of-Algorithms--Assignment2>

Clone the repository and run the provided C++ programs using the instructions in the README.