**Assignment 2. Implementation of DP Matching**

**Student ID:** m5281022 **Name:** LAI HUI SHAN

**Theoretical Background**

Dynamic Programming Matching (DPM) is an algorithmic technique used to find the best possible alignment between two sequences. This method calculates the minimum cumulative cost to align every point in one sequence with every point in the other sequence using a cost matrix.

* **Definition and Initialization:** The DP matrix is a two-dimensional array where each cell (i, j) represents the minimum cost of aligning the first i elements of sequence A with the first j elements of sequence B. The matrix is initialized such that the first row and the first column reflect the base cases of aligning the sequences starting from the beginning.
* **Recurrence Relation:** The value of each cell in the DP matrix is determined by the formula: dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + cost(i, j), where cost(i, j) measures the dissimilarity between elements i of sequence A and j of sequence B. This recursive approach ensures that each step considers the optimal alignment of the previous elements, building up to a solution that covers the entire sequences.
* **Computational Complexity:** The time complexity of the DP matching algorithm is O(n\*m), where n and m are the lengths of the two sequences. This is due to the necessity of filling an n-by-m matrix where the computation of each element requires constant time. The space complexity is also O(n\*m), reflecting the storage required for the matrix. This could be a limitation for very long sequences.

**Algorithm Implementation**

The implementation of the DP matching algorithm involves two main steps: filling the DP matrix and backtracking to determine the optimal alignment path.

* **DP Matrix Calculation:** Starting from the initialized values in the first row and column, the matrix is filled out according to the defined recurrence relation. Each cell accumulates the minimum alignment cost from the start to the positions (i, j) based on the previously computed values.
* **Backtracking:** Once the DP matrix is fully populated, the optimal path is traced back from the bottom right corner to the top left. This path indicates the sequence of steps that results in the minimum alignment cost. During backtracking, decisions are made at each step to move to the cell that contributed the minimum cost to the current cell, reflecting the choices of aligning or misaligning elements.
* **Signal Adjustment:** Based on the optimal path identified through backtracking, adjustments are made to the positions of elements in sequence B to align it more closely with sequence A. This adjusted sequence is then used for further analysis or visualization, demonstrating the practical application of the DP matching algorithm to real-world data alignment problems.

**Programming Language Used:** Python

**Required Libraries:**

* numPy
* matplotlib

**How to Run the Program:**

1. **Environment Setup:**

* Ensure Python 3.x is installed on your system.
* Install the required libraries using the commands:

pip install numpy matplotlib

1. **Executing the Script:**

* Save the code in a file named **dp\_matching.py**.
* Open a command line or terminal window.
* Navigate to the directory where the script is saved.
* Run the script using the command:

python dp\_matching.py

**Code Explanation:**

* **load\_data:** This function reads the data from a text file and returns it as a NumPy array.
* **dp\_matching:** Implements the dynamic programming algorithm to find the minimum cost matching between two data series.
* **backtracking:** Identifies the optimal path through the DP matrix and adjusts the data points of series B accordingly.
* **plot\_data:** Plots the original and adjusted data series on the same plot for comparison.

**Example Output:**

The output includes plots of the original and matched signals. The adjusted signal B is aligned with signal A to demonstrate the matching accuracy.

Figure 1 shows the initial unadjusted signals A and B as loaded from the data files. The signals are plotted to visualize their initial similarity and difference.

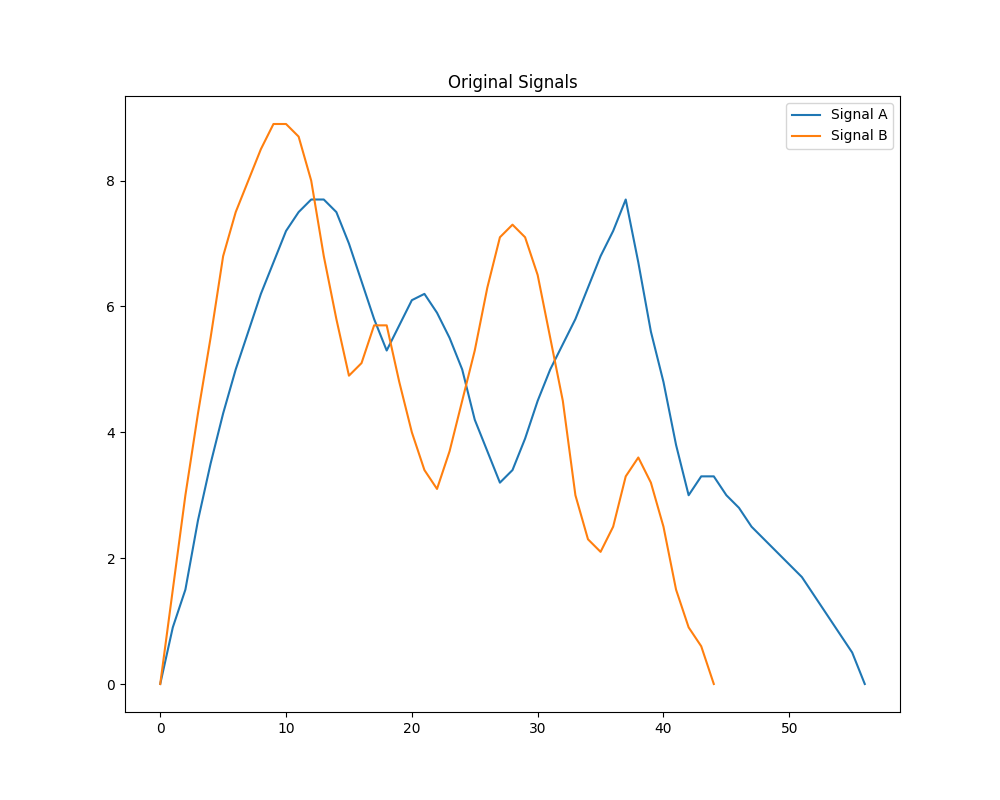


Figure 1: Original Signals

Figure 2 displays the results of applying the DP matching algorithm. Signal B has been adjusted to align with Signal A based on the calculated optimal path, demonstrating the effectiveness of the alignment.

A graph of a signal

Description automatically generated

Figure 2: DP Matching Adjusted Signals

As shown in Figure 1, the original signals start with different alignments, indicating the need for adjustment.

Following the application of the DP matching algorithm, as illustrated in Figure 2, we can observe that Signal B has been adjusted to closely match the profile of Signal A, demonstrating the algorithm's capability to effectively align disparate data sets.

|  |
| --- |
| **Source Code:** |
| import numpy as np import matplotlib.pyplot as plt  # Function to load data from a file and convert it to a NumPy array. # Each line of the file is assumed to contain one float number. def load\_data(filename):  with open(filename, 'r') as file:  data = np.array([float(line.strip()) for line in file])  return data  # Load data for Signal A and Signal B from text files. data\_a = load\_data('data\_a.txt') data\_b = load\_data('data\_b.txt')  # Plot and save the original signals A and B for initial visualization. plt.figure(figsize=(10, 8)) plt.plot(data\_a, label='Signal A') plt.plot(data\_b, label='Signal B') plt.title('Original Signals') plt.legend() plt.savefig('original\_signals.png') # Save the figure to a file plt.show() # Display the figure  # Initialize the DP matrix with zeros and set the first element based on the first data points of A and B. n, m = len(data\_a), len(data\_b) dp = np.zeros((n, m)) dp[0, 0] = np.abs(data\_a[0] - data\_b[0])  # Fill the DP matrix with the minimum cost paths based on the recurrence relation defined. for i in range(1, n):  dp[i, 0] = dp[i-1, 0] + np.abs(data\_a[i] - data\_b[0]) for j in range(1, m):  dp[0, j] = dp[0, j-1] + np.abs(data\_a[0] - data\_b[j]) for i in range(1, n):  for j in range(1, m):  cost = np.abs(data\_a[i] - data\_b[j])  dp[i, j] = min(dp[i-1, j-1], dp[i, j-1], dp[i-1, j]) + cost  # Backtracking from the bottom-right corner of the matrix to find the optimal matching path. i, j = n - 1, m - 1 path = [] while i > 0 and j > 0:  path.append((i, j))  if dp[i, j] == dp[i-1, j-1] + np.abs(data\_a[i] - data\_b[j]):  i, j = i-1, j-1  elif dp[i, j] == dp[i, j-1] + np.abs(data\_a[i] - data\_b[j]):  j -= 1  else:  i -= 1 path.append((i, j)) path.reverse() # Reverse the path to start from the beginning  # Adjust the time coordinate of Signal B based on the optimal path found during backtracking. adjusted\_b = np.empty(n) for index, (i, j) in enumerate(path):  adjusted\_b[i] = data\_b[j]  # Plot and save the adjusted signals where Signal B has been aligned to Signal A. plt.figure(figsize=(10, 8)) plt.plot(data\_a, label='Signal A') plt.plot(adjusted\_b, label='Adjusted Signal B') plt.title('DP Matching Adjusted Signals') plt.legend() plt.savefig('adjusted\_signals.png') # Save the adjusted figure to a file plt.show() # Display the figure |