COMP20007 Assignment 2
Task 1 Written Tasks

Name: Quoc Khang Do

Student ID: 1375531

Task 1, Part B

Given two sequences of data measuring the same type of data (e.g., stock prices of two different companies, or the frequency response of two different audio signals), Dynamic Time Warping (DTW) is used to match/align those two sequences of data in the most similar way possible.

DTW differs from simple sequence matching in that it can still find similarities in the trends of the two sequences even if there were differences in the timing and speed of these trends. For example, if values of sequence A increases by 1 for every unit of time t, and sequence B increases by 2 at that same rate, simple sequence matching would consider those two sequences to not be similar, i.e., simple sequence matching considers similarity in absolute terms. However, the DTW algorithm could stretch sequence B by a factor of 2 to align with sequence A, making them similar.

The DTW cost matrix is $n + 1$ rows by $m + 1$ columns, where n and m are the number of data points in sequences A and B respectively. The extra +1 row and column (specifically the 0th row and 0th column) is used to store sentinel values to make implementing the algorithm simpler. Each cell in the DTW cost matrix represents the cumulative cost of choosing the minimal-cost path from the starting point (1, 1) to that cell. The cost at a matrix cell (i, j) is defined as the absolute difference between the ith data point in Sequence A and the jth data point in Sequence B. Initially, all values within the matrix are set to infinity to ensure that no valid minimum path has been found yet.

Now, to populate the matrix, starting from cell (1, 1), its cumulative cost is just equal to its cost since it is the starting point. For the remaining cells, its cumulative cost = its cost + the minimum cumulative cost out of the three previous cells it could have departed from – top, left, or top left:

- Downwards movement (departing from the top cell) corresponds to sequence B being stretched relative to sequence A.
- Rightwards movement (departing from the left cell) corresponds to sequence A being stretched relative to sequence B.
- Diagonal movement (departing from the top left cell) corresponds to a match.

Therefore, the value of the very bottom right cell of the matrix (n, m) is the cumulative cost of choosing the minimal-cost path from the starting point (1, 1), i.e., the optimum cost of aligning the two sequences.

Task 1, Part C

The recurrence relation of the DTW algorithm:

$$dtwMatrix[i][j] = cost[i][j] + \min(dtwMatrix[i-1][j], dtwMatrix[i][j-1], dtwMatrix[i-1][j-1])$$

This algorithm makes use of dynamic programming by storing known cumulative costs (sub-problems) in the entries of $dtwMatrix$, meaning we can access them without having to recompute these values.

For example, if we were to find the cumulative cost at cell (i, j) without using dynamic programming, we would have to calculate the cumulative costs of the preceding cells, which requires having to calculate the cumulative costs of the cells before that, and so on…, inevitably leading to overlapping problems as we would have to recalculate them again in later iterations. However, we can instead store the answers to these subproblems in a matrix for direct access as seen above in the recurrence relation: $dtwMatrix[i-1][j]$, $dtwMatrix[i][j-1]$, and $dtwMatrix[i-1][j-1]$.

Essentially, the algorithm provides a bottom-up dynamic programming approach by iteratively tabulating all the possible sub-problems in a matrix, which can be used to find the optimal path.

Task 1, Part E

Let n and m be the lengths of sequences A and B respectively, then both algorithms in parts A and D require a matrix of $(n+1)\times(m+1)$ in memory to store cumulative costs. Therefore, both algorithms' space complexities are $O(nm)$.

As we are discussing the computational complexity, consider a basic operation to be computing the cumulative cost for a cell in the matrix.

The algorithm in part A has nested 'for' loops, where the outer loop runs n times and the inner loop runs m times. Therefore, the algorithm does $n\times m$ basic operations in all cases (for all input). So, its complexity is $\Theta(\mathrm{nm})$.

The algorithm in part D also has nested 'for' loops, but with boundary constraints, meaning not all cells in the matrix have to be computed. Therefore, performing fewer basic operations as seen in its recurrence relation:

$$dtwMatrix[i][j] = \begin{cases} cost[i][j] + \min(dtwMatrix[i-1][j], dtwMatrix[i][j-1], dtwMatrix[i-1][j-1]), & |i-j| \le windowSize \\ \text{infinity}, & \text{otherwise} \end{cases}$$

$|i-j| \le windowSize$ ensures that we only compute cells within the given window size. Otherwise, do nothing because all cells in the matrix were already initialized to infinity.

Let the window size be w, and there are $\min(n,m)$ cells in the diagonal of the matrix. So, for every diagonal cell, we only compute at most w cells to either side of that diagonal cell, including itself – so, $2w+1$ cells per diagonal element. Therefore, the complexity of the algorithm in part D is $O\big(\min(n,m)\times(2w+1)\big) \in O(\min(n,m)\times w)$.

Task 1, Part G

Like Part E, a matrix takes up $O(nm)$ space. However, the algorithm in part F introduces a 'layer' dimension (indexed by k below in the recurrence relation), where there are now $maxPathLength$ number of matrices. Let $maxPathLength = d$, therefore the algorithm in part F has a space complexity of $O(dnm)$.

Again, consider the basic operation to be computing the cumulative cost for a cell in a matrix. Now, we have to compute cell values for d matrices. But for each matrix, there are boundary constraints, so not every cell of every matrix needs to be computed. This is shown in its recurrence relation:

$$dtwMatrix[k][i][j] = \begin{cases} cost[i][j] + \min(dtwMatrix[k-1][i-1][j], dtwMatrix[k-1][i][j-1], dtwMatrix[k-1][i-1][j-1]), & i,j \le k \text{ } and \text{ } i+j > k \\ \text{infinity}, & \text{otherwise} \end{cases}$$

$i,j \le k \text{ } and \text{ } i+j > k$ ensures that we only compute the cells that are only reachable from k steps for each matrix. Otherwise, do nothing because all cells in the matrix were already initialized to infinity.

As seen in an example below (Figure 1) given in the assignment specification, we need to compute 1 cell for a path length of 1, then 3 cells for a path length of 2, then 6 cells for a path length of 3, and so on… So, the total number of cells required to compute across all the layers is $\sum_{k=1}^{d}\sum_{h=1}^{k} h$.

Evaluating this:

$$\sum_{k=1}^{d}\sum_{h=1}^{k} h$$

$$= \sum_{k=1}^{d} \frac{k(k+1)}{2}$$

$$= \frac{1}{2}\sum_{k=1}^{d} k^2 + \frac{1}{2}\sum_{k=1}^{d} k$$

$$= \frac{1}{2}\left(\frac{d(d+1)}{2}\right) + \frac{1}{2}\left(\frac{d(d+1)(2d+1)}{6}\right)$$ known formulas for sum the of first k integers, and the sum of the squares of the first k integers.

$$= \frac{d(d+1)}{4} + \frac{d(d+1)(2d+1)}{12}$$

$$= \frac{3d(d+1)+d(d+1)(2d+1)}{12}$$

$$= \frac{2d^3+6d^2+4d}{12}$$

$$= \frac{d^3+3d^2+2d}{6}$$

Therefore, the computational complexity of the algorithm in part F is $O\left(\frac{d^3+3d^2+2d}{6}\right) \epsilon\ O(d^3)$, where $d = maxPathLength$.

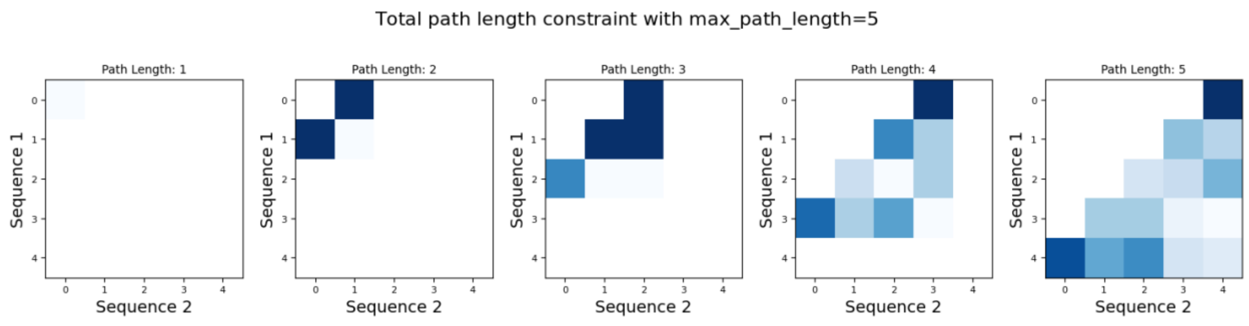

Figure 1: 5 layers of DTW matrices