Ai Ling Chiam (Irene) 1384412 Design of Algorithms Assignment 2

Task 1:

Part B:

Dynamic Time Warping is an algorithm to measure how similar 2 sequences are and calculate the optimal alignment between them via stretching and compressing parts of the sequence. The optimal alignment is the minimal alignment cost between all points in the sequences.

Dynamic Time Warping is flexible, as each element in a sequence is matched with every element in the other sequence, hence it could align elements of different sequences at different times. Unlike simple sequencing matching, Dynamic Time Warping uses dynamic programming, which allows us to explore all possible alignments and calculate the minimal alignment cost. In addition, Dynamic Time Warping handles variations in speed during the cost calculation, which is the absolute difference between the elements. This further shows the algorithm's flexible matching between values.

In the initial set-up of the cost matrix, each cell represents the cumulative alignment cost of 2 sequences up to that point. The cells are initialized as the greatest long double value, which is DBL_MAX. I didn't initialize the cells as 0 because 0 is the smallest cost value, so the value won't change as I iterate the points of the sequence. Using the largest possible long double value allows me to replace it if I find a smaller alignment cost for that cell.

I start by iterating from the first point of the first sequence and comparing it with every point of the second sequence. The alignment cost of the current points is the absolute difference between them. This will be added to the minimum cost of the 3 operations: Insertion, Deletion, and Match. Using the minimal costs allows us to achieve minimal total cost for each cell continuously. The optimal alignment cost is the minimal cost from the start (costs[0][0]) to the end, which would be costs[length of sequence A][length of sequence B].

Part C:

The recurrence relation used in Dynamic Time Warping is

```
(s->matrix)[i][j] = cost + min((s->matrix)[i-1][j], (s->matrix)[i][j-1], (s->matrix)[i-1][j-1]);
```

Which is the addition of the alignment cost of the current points and the minimal cost between Insertion, Deletion, and Match. This facilitates the dynamic approach to finding the minimal distance. As we iterate through the cost matrix, we are always trying to find the smallest alignment costs for each point.

Part E:

The computational complexity of the algorithm implemented in Part D is O(m * n), where m is the length of the first sequence and n is the length of the second sequence. The computational complexity of the standard DTW algorithm is also O(m * n), however, due to Part D's addition of

window size, it reduces the number of cells to remove unnecessary computations. Hence, it would be faster than Part A. Also, the accuracy of the optimal alignment cost would be affected depending on the window size. The recurrence relation is fairly the same, except it just performs in on points that are within the window size.

Part G:

The computational complexity of the algorithm implemented in Part F is O(maximum length * length of first sequence * length of second sequence), it is comparably larger than the computational complexity of the standard DTW algorithm. This came from the initialization of the 3D matrix, as well as the iterating through it to find the cumulative costs for every possible path length. Lastly, the 3D matrix is iterated again to find the optimal cost from the start of the sequences to the end of them regardless of path length. The recurrence relation compared to Part A is also similar, but since I added an extra dimension to the cost matrix, I compared the Insertion, Deletion, and Match costs from the previous length.