Sequence Alignment Problem - Brute Force vs Dynamic Optimization
COT 6405 Project, Spring 2020
Adam Cankaya, acankaya2017@fau.edu

1. Problem definition.

The Sequence Alignment Problem is a fundamental string comparison computing problem. Given two or more strings consisting of a sequence of characters the problem asks to "line up" the strings and to produce a composite string that maximizes similarity through deletion of characters and insertion of gaps. We define similarity based on a scoring matrix that compares an index position across strings. If characters are the same we skip over them, but if the characters differ we deduct points ("mismatch cost"). To maximize our point score we also consider adding gaps to the composite string - although every gap also has a deduction of points associated with it ("gap penalty"). This gives us a "a concrete numerical definition for the similarity between strings X and Y: it is the minimum cost of an alignment between X and Y...the lower this cost, the more similar we declare the strings to be" [1].

The input size is dependent upon the application. A simple real world application are online dictionaries that allow users to enter a word with slight misspellings and get suggestion dictionary entries returned back. If a user searches the dictionary for *occurance* they can quickly be offered the entry for the word *occurence*. Given the small number of characters in most English words this string comparison between user input and dictionary keys is not necessarily computationally costly. However when the number of characters increases beyond normal language, such as the case in genetic sequences, the computational cost can become enormous. The human genome can be written in a language of four characters representing four nucleobases - guanine (G), cytosine (C), adenine (A), and thymine (T). These bases are written in a string of characters separated into 23 chromosomes and for an average human consists of over 3 billion characters ("base pairs") [2].

2. Algorithms and RT analysis.

We are considering two algorithms - a simple brute force approach and a more efficient dynamic programming optimization algorithm similar to what is known as Smith-Waterman [3]. For both algorithms we will use a simple system where the gap penalty is 1 point and a mismatch is 3 points. If two characters match then no points are assigned. The solution with the lowest cost score is declared the winner.

a. Brute Force

```
Input: two strings of length m, list of four nucleotide and one gap character G,C,A,T,-

For every permutation of the nucleotides & gap characters creating a string of length m:
        Calculate score when comparing the permutation to string one and to string two
                For each index in the permutation string:
                        Compare the permutation character value to the character at the same
                        index in string one and string two:
```

```
                    If at least one of the characters is a gap:
                        add the gap penalty to the score
                    If the characters match:
                        do nothing.
                    Else if the characters are not gaps and do not match:
                        Add the match penalty to the score
                    If the characters do not match
            Keep track of the best (lowest scoring) sequence and its score
        Return the best (lowest scoring) sequence and its score
```

The runtime for each permutation is O(m*m) because for each character in one string of length m we are comparing it to every character in another string of length m so we have m*m operations. For four nucleobases and one gap character we have 5^m permutations so the overall runtime is $O(5^m m^2)$

b. <u>Dynamic Optimization</u>

```
Input: two strings of length m, list of four nucleotide and one gap character G,C,A,T,-

Create the scoring matrix A of size m x m and fill it with zeros

Create the directional matrix arrows of size m x m

Fill the left most column of A so that A[i,] = i * gap_penalty

Fill the bottom most row of A so that A[,j] = j * gap_penalty

# Fill up the scoring matrix and arrows matrix
For i between 1 and m+1:
        For j between 1 and m+1:
                # calculate the three values associated with diagonal, left, and up directions
                If string_one[i-1] is not same as string_two[i-1]: # see if the characters match
                        value_one = A[i-1,j-1] + mismatch_penalty
                else:
                        value_one = A[i-1,j-1]
                value_two = A[i-1, j] + gap_penalty
                value_three = A[i, j-1] + gap_penalty

                #  find minimum value and track the direction it came from
                A[i,j] = minimum(value_one, value_two, value_three)
                arrows[i,j] = “diagonal” for value_one OR “left” for value_two
                        OR “down” for value_three
```

```
# Traceback the solution starting from the uppermost right hand corner of the scoring matrix
x_index = y_index = m
best_score = 0
while true:
        best_score += A[i,j]
        if arrows[x_index, y_index] == “diagonal”:     # values match
                solution_one.prepend(string_one value at x_index) # last value of string one
                solution_two.prepend(string_two value at y_index)  # last value of string two
                x_index = x_index + 1
                y_index = y_index - 1
        else if arrows[x_index, y_index] == “left”:      # first string has a gap
                solution_one.prepend(“-”)
                solution_two.prepend(string_two value at y_index)
                y_index = y_index - 1
        else if arrows[x_index, y_index] == “down”:  # second string has a gap
                x_index = x_index + 1
                solution_one.prepend(string_one value at x_index)
                solution_two.prepend(“-”)
        else:    # no more directions to follow so we’re done
                break

return best_score, solution_one, solution_two
```

For each character in one string of length m we are comparing it to every character in another string of length m so we have m*m operations or a runtime of $O(m^2)$.

3. Experimental Results

Both algorithms are implemented using Python 2.73 and the Numpy library. All of our strings are stored in standard Python one dimensional list objects and our two dimensional matrices are stored in Numpy list objects. Before running an algorithm we generate a random character string for a given size using the list of nucleotide characters. To create a second input string we randomly mutate 50% of the characters in string one to another choice from the list of nucleotides. We experimented with different mutation rates between 10% to 90%, but found that 50% created a new string that is both different enough and similar enough for the algorithms to find intuitive results. We utilize the standard Python random() function to create and mutate the string characters.

Due to the drastic difference in runtimes between the two algorithms we utilize different sized inputs. The brute force algorithm is so inefficient that running it on strings of 15 characters took over two days per run to perform. With this in mind we limited our brute force input strings to sizes 2,3,...15. The dynamic optimization algorithm can handle larger strings much more efficiently so we have expanded our input strings to have sizes of 100, 1,000, 2,000, ... 9,000 characters. We originally wanted to test even larger size inputs, but have memory error issues

for inputs of 10,000 characters or more. For both algorithms we perform three runs for each input size and keep track of how long each run takes to complete not including any time needed to generate the random input strings

a. **Brute Force Results**

Input length	Average runtime (seconds)		Input length	Average runtime (seconds)
2	0.00		9	8.78
3	0.0033		10	54.067
4	0.01		11	303.97
5	0.017		12	1403.20
6	0.12		13	7115.72
7	0.35		14	38234.45
8	1.64		15	195247.58

Table 1 - Brute force sequence alignment results

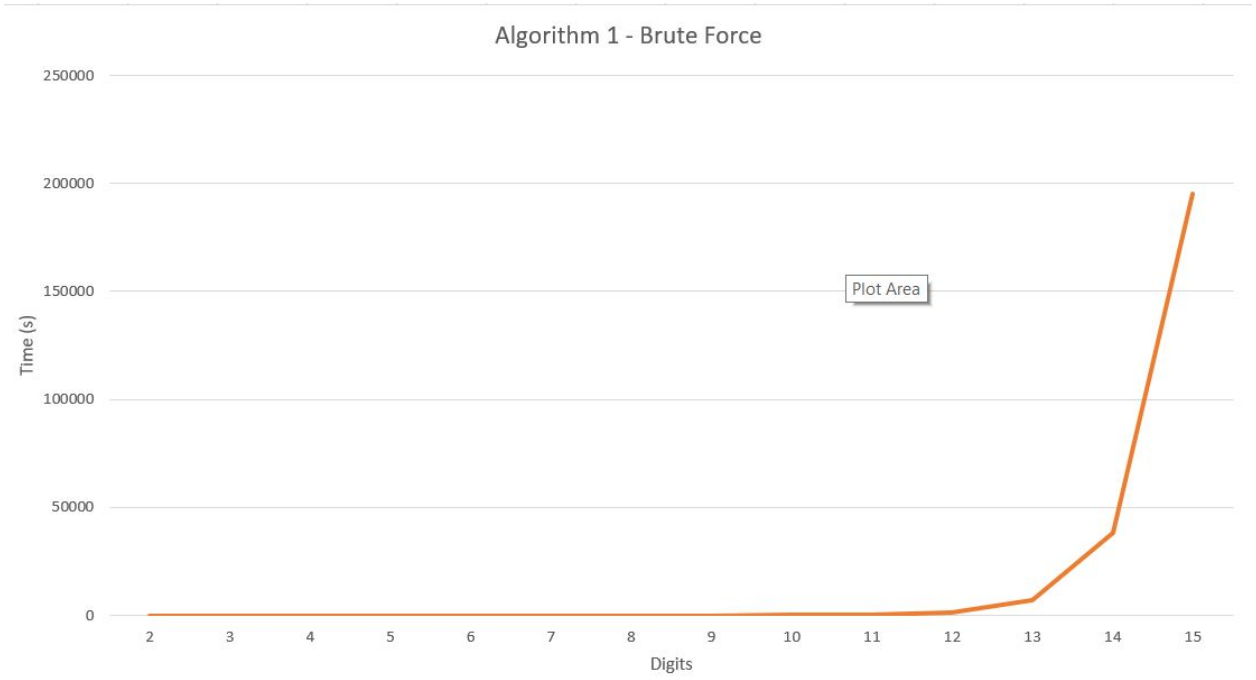

Figure 1 - Brute Force sequence alignment results - number of input string digits vs runtime.

For an example, using the 14 digit input strings “CGTGTCCGGTTGCA” and “TCTTTGCCATTGGT” gives us an optimal result of “ --T-T-C--TTG--” with a cost of 16. The

algorithm correctly found all the matching characters and then replaced the mismatches with a gap. Our results correlate with the exponential runtime we expected. Input strings of 7 characters or less finish in less than a second. Every increase in input length by one character results in an approximately 5 times increase in runtime. Strings of 10 digits take one minute to analyze, 11 digits take five minutes, 12 digits takes twenty-five minutes, 13 digits takes almost two hours, 14 digits takes almost ten hours, and 15 digits takes almost two days. This factor of five correlates with the five characters of permutation (four nucleobases plus one gap character).

b. <u>Dynamic Optimization Results</u>

Input length	Average runtime (seconds)		Input length	Average runtime (seconds)
100	0.11		5000	238.41
1000	10.82		6000	427.78
2000	43.07		7000	549.33
3000	97.11		8000	631.32
4000	168.22		9000	797.13

Table 2 - Dynamic optimization sequence alignment results

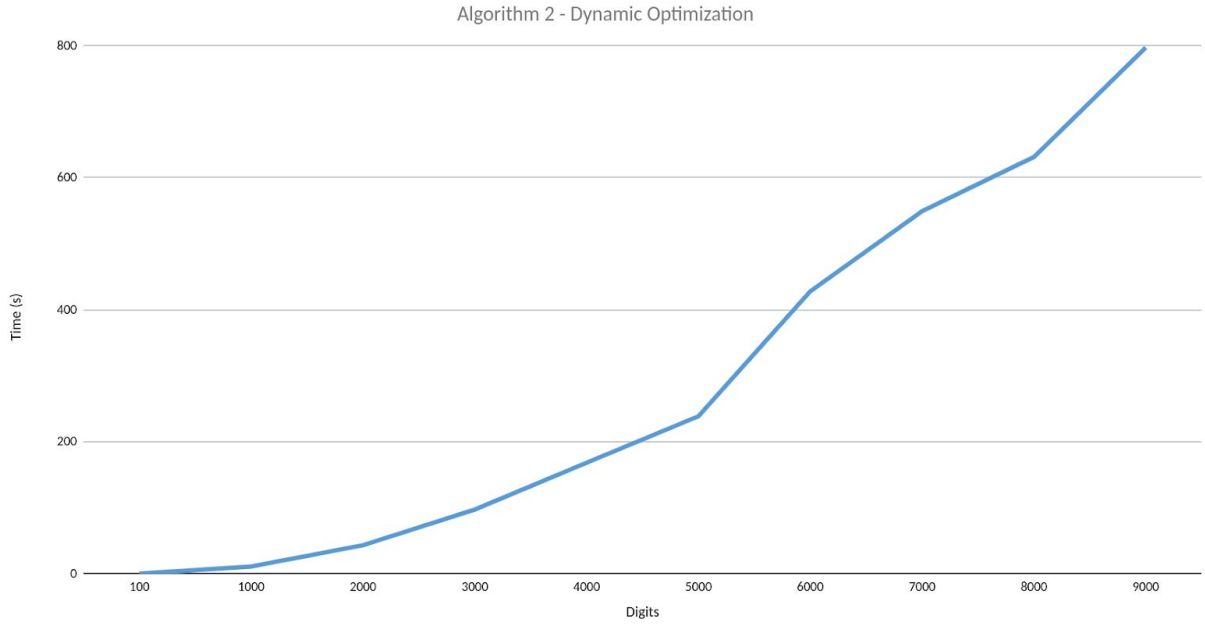

Figure 2 - Dynamic optimization alignment results - number of input string digits versus runtime.

For example, the 10 digit input strings of *CGTTCTCGGT* and *TGCTTCCGAT* have mismatching characters at index 0, 2, & 7. For 3 mismatched characters with a mismatch penalty of 3 each this gives us a base score of 9. When we run these two input strings through the dynamic optimization algorithm we get a new alignment that gives an optimum score of 6 for the results *C-G-TTCTCGG-'T* and *-TGCTTC-C-GAT*. The algorithm has correctly inserted 6 gaps for a penalty of only 1 point each. Our results correlate with the polynomial runtime we expected. Every doubling of input size results in a four times increase in runtime. Input strings of 1000 characters finish in about 10 seconds, strings of 2000 characters finish in about 40 seconds, strings of 4000 characters finish in about 160 seconds, and strings of about 8000 characters finish in about 640 seconds.

Since we used mostly different sized inputs between the two algorithms we did not do test runs using the same input strings. We cannot guarantee that both algorithms will give the same results for the same input because Algorithm 2 is an optimization algorithm and does not claim to get the absolute best result like the brute force algorithm does. Also the two algorithms give slightly different types of results. Algorithm 1 returns a single string that represents the best overall average between the two input strings, while Algorithm 2 returns a subset of the characters found conserved in both strings.

To verify both algorithms give similar results using the same input strings we did a separate run of Algorithm 2 using the 14 digit input example presented early. Running Algorithm 2 using inputs of *CGTGTCCGGTTGCA* and *TCTTTGCCATTGGT*, which has six mismatched characters and hence a base score of 18, gives us an optimum score of 12 for the results *-CGTG-T-CCGG-TTGCA--* and *TC-T-TTGCC--ATTG--GT*. Taking the pairs gives us (1,2), (3,3), (5,5), (6,7), (7,8), (10,10), (11,11), & (12,12) or *CTTCCTTG*. This compares to the result from the brute force algorithm 1 of *--T-T-C--TTG--* or with the gaps removed *TTCTTG*. Both algorithms found the directly matching pairs, but the dynamic optimization went one step further and used gaps to line up previously mismatched pairs such as the first character of string one ‘C’ and the second character of string two ‘C’.

4. Conclusions

Our experimental results match very nicely with the expected theoretical results. We expected the brute force algorithm to be much more computationally costly than the dynamic optimization algorithm due to its increased runtime. We theoretically predicted the brute force algorithm would have exponential results with an increasing number of digits. We found this to be true with every input string increase in length by a single digit resulting in a runtime five times longer. Similarly we predicted the dynamic optimization algorithm to not only be able to handle longer input strings, but return results in a polynomial time. We found this to be confirmed in our empirical results where a doubling of the input length caused the runtime to increase by four times.

Overall we can conclude that the dynamic optimization algorithm is obviously much more efficient than a brute force approach. The brute force algorithm can barely handle input strings of an average word’s length and results take minutes to return, while the dynamic programming algorithm can handle input lengths in the thousands and take only seconds to get results. Such

a drastic increase in ability to handle longer input strings enables the dynamic optimization algorithm to be useful for comparing modern genetic data involving billions of characters.

5. References

[1] J. Kleinberg and E. Tardos, *Algorithm Design: Pearson New International Edition*. Pearson Higher Ed, 2013.

[2] J. Pevsner, *Bioinformatics and Functional Genomics*. John Wiley & Sons, 2005.

[3] M. Cardei, “Dynamic Programming,” COT 6405 Analysis of Algorithms, Florida Atlantic University, Spring 2020.