**Capstone Project – Digital Humanities**

Marlene Williams, Greg Dawkins, & Kevin Woods

INTRODUCTION

Over the years students have found various ways of transforming text from the original authors into something that they could possibly use in their research. Many times it becomes very hard for a student to transform this published text into a usable, trustworthy solution that they might use in a formal paper. It is often the challenge of the student to know which publication is the original transcript of the text. Calculating the number of modifications between similar text is one way of determining sequence analysis for identification of the most likely original text. This comparison of texts is known as string editing metrics.

The technique of computing the distances between texts is very complicated and can be achieved using many different methods. The testing base mark for this Fall, 2015, Capstone Project will use Levenshtein’s Distance method. This distance algorithm will be put in place to determine a distance measurement between two text inputs. This text input is to be convert to a string during the input process. The coding for this method has been done through the input of group members Greg Dawkins, Marlene Williams, and Kevin Woods.

Once the Levenstein Distance method is analyzed by the group the input string will then be analyzed by three different distance algorithm methods. The methods that have been chosen are Jaro-Winkler method, Needleman-Wunsch method, and the Hunt-McIlroy method. We opted to create our own input files for comparison so that we would know the correct sequence in which they were created, and to be able to determine if the algorithms we chose would produce the expected results.

LEVENSHTEIN’S DISTANCE METHOD

Levenshtein’s distance algorithm is a method that was developed by Vladimir Levenshtein in 1965. In the *Dictionary of Algorithms and Data Structures,* Levenshtein’s method is described as an algorithm that uses the smallest number of insertions, deletions, and substitutions that would be needed to change one string into a similar string sequence. He published this as a tool to measure the distance between two sequences. This tool is still used widely today in various computational sequences like DNA analysis, spell checker software, or even fuzzy string matching.

JARO-WINKLER ALGORITHM

In the late 1980’s and early 1990’s, Matt Jaro and Bill Winkler worked on string comparison methods which would later become the standard comparator used in the C language, strcmp.h was the result of their hard work and years of research. The Jaro-Winkler Distance measures similarity between two strings. This specific algorithm was developed and best suited for shorter strings, such as names or single words. Although at the time of development, addressable memory was much less abundant than what we find in computers of today, it can be used for much longer strings now. This is because a string’s capacity is determined by the size of any given hardware’s maximum integer.

The premise of this algorithm is that by counting the matching characters in two different strings, while taking into account possible typos like transpositions within a calculated range, one can determine the similarity between the two in a value between 0 and 1. That alone is the Jaro Distance. This is in contrast to Levenshtein’s algorithm, in which the number of insertion, deletions, or substitutions is merely counted. With Jaro-Winkler, the greater the value the more matches exist, so a value of 1 is an exact match. The portion of Jaro-Winkler Distance that separates it from the simpler Jaro Distance is a common prefix scale is also taken into account. Givin an arbitrary boost threshold (in their case, 0.7 was used), if the Jaro distance is less than the boost threshold, Jaro-Winkler Distance is the Jaro Distance. However, if it is greater than the threshold, the prefix scale is applied and Jaro-Winkler Distance becomes the Jaro Distance plus a scaling factor times the length of the common prefix times the difference between 1 and the Jaro Distance. So, longer common prefixes of strings results in a slightly higher Jaro-Winkler Distance versus Jaro Distance alone.

The complexity of this algorithm is rather disappointing in terms of efficiency, with an O() solution. As it turns out, Levenshtein’s Distance was developed later with intentions of being a more time-efficient algorithm to do the same kind of comparison.

NEEDLEMAN-WUNSCH ALGORITHM

Another string comparison method was developed by Saul B Needleman and Christian D Wunsch around 1970. This method is long been used in biometrics to find the best alignment of string sequences. In the article written by Authur M. Lesk for Encyclopeadia Britannica, he describes the Needleman-Wunsch as an algorithm that divides the larger string into smaller subsets which in turn constructs the solution to larger problem.

Their algorithm is implemented by creating a grid based on the length of the strings. This grid of strings is then sequenced through each letter or symbol while making comparisons of the ASCII representation. These computations are the identified in this grid as match or mismatch. Also, there is a function to allow identification of gaps in sequences known as a gap penalty. Finally, after the grid is computed, the alignment is traced back through the grid by way of recursion for the max separation distance of the set of strings.

The pseudo-code for Needleman-Wunsch is basically generated in two parts: construction of an alignment matrix and then the code that looks at the recursion logic of the alignment. First, we create the number of rows and columns of the matrix using the value of the possible decimal values of ASCII table. Then fill the matrix with by comparing the row number and column numbers for match/mismatch. An example is that row 1 and column 1 are matches in the numeric value identifying the headers so this would be a match. Whereas, row 1 and column 2 have different numeric values identifying the headers so this would be a mismatch. The pseudo code for this first part is :

for i = 0 thru the dec ASCII value (127)

for j=0 thru the dec ASCII value (127)

if i = j --> matrix [i][j] = match value

else matrix [i][j] = mismatch value

For the purposes of smaller input data, the match and mismatch have been set a 1 and -2 respectively. There is a value that measure a penalty for the amount of nonmatched values that is known as the gap penalty. Again, this value can be adjusted according to the type and amount of input data received and the desired output values. To compute the matrix, all possible alignments are recorded and then the max value is recorded into this grid.

for i=0 to the length(string A)

Array[0][i] = gapPenalty \* i;

for j=0 to the length(string B)

Array[j][0] = gapPenalty \* j;

for i=1 to the length(string A)

for j=1 to length (string B)

{

Match = Array[i-1][ j-1] + matrix (int)(string A, string B);

Delete = Array[i-1][ j] + gapPenalty;

Insert = Array[i][ j-1] + gapPenalty;

Array [i][j] = max (Match, Delete, Insert);

}

Once this grid is filled then recursion is done to the optimal alignment sequence to the input data. This can be demonstrated by the following pseudo coding:

Set Alignment A and B = ""

i = length(string A);

j = length(string B);

for (i>=1 && j>=1; --i)

{

if ( Array[i][j] = Array [i-1][j-1] + matrix (int)(string A, string B);

{

Alignment A = (int) string A[i-1] + Alignment A;

Alignment B = (int) string B[i-1] + Alignment B;

}

else if ( Array[i][j] = Array [i-1][j] + gapPenalty);

{

Alignment A = (int) string A[i-1] + Alignment A;

Alignment B = (int) '-' + Alignment B;

}

else

{

Alignment A = (int) '-' + Alignment A;

Alignment B = (int) string B[i-1] + Alignment B;

}

}

while (i >= 1 && j < 1)

{

Alignment A = string A[i-1] + Alignment A;

Alignment B = '-' + Alignment B;

}

while (j >= 1 && i < 1)

{

Alignment A = '-' + Alignment A;

Alignment B = string B[j-1] + Alignment B;

}

return matrix [ length (string A)][length (string B)];

The return value represents the scoring integer in the bottom, right corner of the matrix. This would be the max score returned by using values associated with the match, mismatch, and gap penalty value assignments.

These algorithms are widely used for DNA sequencing and even voice/face recognition. The complexities of the algorithm on small scale strings make this algorithm one of the widely used in the industry for many of these type sequencing analysis tasks. However, because of the grid allotment of the string comparisons this algorithm is not optimal for larger data sets. This would not be suitable when making larger string comparisons. Because of the way this algorithm aligns the input, the complexity of this algorithm can be defined as O(mn) due to needed resources to store the input.

The Levenstein’s Distance method and the Needleman-Wunsch are similar in several ways. Both traverse through the string to identify like characters. There are a few dissimilarities. One major difference is the global and local alignment sequences. Levenshtein’s is more efficient but lacks the reliability of the Needleman-Wunsch when measuring smaller data sets. In the results from testing given data set (Emily Dickenson Poems), the average run time in three controlled executions of the program returned values of .045532 (NW) and .037693 (L). This shows that the Levenschtein's measurement ran about .007839 seconds faster than the Needleman-Wunsch. However, when a much smaller data input is measured, the gap between the run times is minimized. This is the result of the complexity differences in the two algorithms. The data shows similar results in the distance measurements but greater efficiency of the Levenshtein algorithm run time. The results comparing Needleman-Wunsch and Levenshtein's algorithm is that when larger data input is needed for comparison, Levenshteins distance measurement is more efficient. But the role of the Needleman-Wunsch is better utilized when there is smaller inputs because the adjustments for the scoring mechanism (match, mismatch, and gap penalty) can potentially create a more visible separation between the different inputs being compared.

\*\* Need a Pie Chart to compared Average times between LEV and NW.

HUNT-MCILROY

The Hunt-McIlroy algorithm was discovered by J. W. Hunt from the Department of Electrical Engineering at Stanford University in Stanford, California and M. D. McIlroy from Bell Laboratories in Murray Hill, New Jersey. The algorithm is also called the diff program. Hunt and McIlroy published their findings in the document, An Algorithm for Differential File Comparison, which was first published in Bell Laboratories Computing Science Technical Report #41 in 1976. Diff is utilized as the central procedure and used to solve the longest common subsequence problem. The main difference between the Hunt-McIlroy algorithm and Levenshtein Distance is that Hunt-McIlroy determines the string subsequences in which two files are similar. It uses that information and attempts to construct a new file using the similarities. However, Levenshtein Distance returns the amount of changes between the two files. Another difference is that Hunt-McIlroy analyzes the files line-by-line while Levenshtein compares the entire file.

To use the algorithm, one must first read in the files line-by-line, where file one would be called and file two called . The *i*th line of file one would then be compared to the *j*th line of file two. An integer will be returned which tells how many characters are common in the two lines, *Pij*. For instance, would be satisfied by

This integer represents the longest common subsequence between the two lines of the two files. For large files, meaning thousands of lines, one would create a hash value. For instance, one possible hash value might be to remove all white spaces within each linein order to create one long computer word, which would then be compared. For our purposes, the files were read-in as one line since the poem used was four lines long.

The next step in the algorithm would be to determine a *k*-candidate, a set of coordinate values used to represent matches between the two files. This step is considered the *diff* algorithm. Three criteria must be met before a *k*-candidate can be considered:

1. A longest common subsequence of length *k* exists between the first *i* elements of file one and the first *j* elements of file two.
2. No common subsequence of length *k* exists when either *i* or *j* is reduced.

If one could imagine a two-dimensional array of size , where each line is represented as an element, matching coordinates would become *k*-candidates. For instance, if file one has lines and file two has lines, the set of *k*-candidates might be

One would then use *k*-candidates to construct a new file using only the shortest distance of the matches. The time complexity of this algorithm for a worse-case is generated by the merging of the two files, .

However, for the purposes of our experiment, the entire Hunt-McIlroy algorithm was not used. It was modified to only return the value since the files did not require merging into a third file. The Hunt-McIlroy algorithm has been used to determine plagiarism, as well as create patches for older programs. While the algorithm works well for these purposes, Levenshtein’s Distance is best for the purpose of determining the order in which files have been created.

CONCLUSION

In conclusion, the Levenshtein’s Distance method may be an optimal choice for larger data. It would be determined by the size of the data, storage of the string, complexity of the algorithms and capacity of the hardware. As for using any of these algorithms to determine the sequence in which the drafts were created, we have not found any evidence to support the ordering assumed by our results. To continue this project, we would recommend that further research should be done in regards to other more time-efficient algorithms and the selection of larger input data. Those developed in more recent years, such as Minkowski Distance or trigrams would likely return results in less time and handle more data, as those we tested are older algorithms that simply aren’t optimized for handling large data sets, with the exception of Hunt-McIlroy, which was designed for comparing larger files line by line.

\*\* Results attached in Excel workbook.