[[1]](#footnote-1)

A Study of Distance Editing Algorithms

Kevin T. Woods, Marlene M. Williams, and William G. Dawkins

*Abstract*—In determining the order of multiple drafts of text documents and manuscripts, long has it been a tedious task to study the changes from one draft to the others. Over time, as computing technology has progressed, so has the development of algorithms that compare edit distances, search, sort, find, replace and merge. Unfortunately, all such algorithms must contain logic that touches every character within a set of inputs. Levenshtein’s Distance algorithm has been proven to be very helpful in the determination of draft sequences, though it has a high order of complexity that can quickly consume the available resources within a given computer system. In hopes to find a better solution, our task herein is to compare alternate string editing metric algorithms to Levenshtein’s for the purpose of measuring their efficacy in terms of accuracy and performance.

*Index Terms*—draft sequencing, edit distance, Hunt-McIlroy, Jaro-Winkler, Levenshtein, Needleman-Wunsch, string editing metric, textual critic.

# INTRODUCTION

O

VER the years, authors have written multiple drafts of a document before reaching a final version that gets published. When available, textual critics seek to analyze these drafts in search of some insight into the mind and style of the author. It is most often a challenge to know in which order these drafts were created, as many older manuscripts were not dated or sequentially noted. To this effect, much time and research have been put into the finding of efficient ways to determine such information.

The technique of computing the distances between texts is complicated and can be achieved using many different methods. Our testing benchmark for this project will be the Levenshtein Distance method. This algorithm is used to determine a difference measurement between two character-based inputs.

Identifying the modifications between drafts of a manuscript or other document has been shown, with some degree of confidence, to assist in determining the sequence of a given set of drafts. For this purpose, we will be using a handful of known string editing metric algorithms (Hunt-McIlroy, Jaro-Winkler, and Needleman-Wunsch), or slightly modified versions thereof, as tools to compare multiple short drafts of sample texts.

In this study, we seek to confirm the validity of using a distance editing algorithms to determine the sequential order of text revisions, and to compare the complexity and efficiency of the other algorithms in relation to Levenshtein’s method. The only caveat herein is that the original transcript of the text must be known. We were provided with three slightly differing versions of an Emily Dickinson poem to start with, but also opted to create our own input files for comparison. This was done so we would know, with certainty, the correct sequence in which they were drafted and as such, be able to determine if the algorithms we chose would produce the expected results.

The C++ code for these methods has been developed by the authors of this paper, through research and re-implementation of work done by the original algorithm authors, who typically used the C programming language in their original implementations.

# Levenshtein’s Distance: The Base Case

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

The difficulty of using the Levenshtein distance method to compute edit distances between strings is that it is costly in regards to efficiency. The algorithm, when used with strings of equal length, would have a complexity of roughly O(), which, by current standards of computer capabilities, limits the comparable strings to short sequences to maintain a reasonable calculation time.

Unfortunately to date, character by character comparisons have been the only way to perform the type of analysis required to calculate such distance measurements. Those measurements can then be used to help determine the likely sequence in which drafts were created. In an effort to find a valid solution with less than O(), we have researched, implemented, and tested other algorithms in terms of their effectiveness and performance.

# Alternate Algorithms

## Jaro-Winkler Distance

In the late 1980’s and early 1990’s, Matt Jaro and Bill Winkler worked on string comparison methods, one of which would later become the standard comparator used in the C language, *strcmp.c.* 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 the Jaro Distance 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. With Jaro and Jaro-Winkler, the greater the value the closer the strings are to each other, so a value of 1 is an exact match. This is in stark contrast to Levenshtein’s algorithm, in which the number of insertions, deletions, and/or substitutions is merely counted, so an exact match would result in a distance value of 0 using Levenshtein’s method.

In the first equation for Jaro distance score, m is the number of matches within range, t is half the number of transpositions, s1 is string 1 and s2 is string 2, where their lengths are used in absolute value notation.

The range in which to search for a match in the Jaro distance is defined by the following equation.

The portion of Jaro-Winkler Distance that separates it from the simpler Jaro Distance is that a common prefix scale and scaling factor are also taken into account. Given 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 or equal to 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 the Jaro-Winkler algorithm is rather disappointing as it also has an O() worst case solution. This would result in the same set of difficulties and marginally reduced computation time as is found in Levenshtein’s Distance was developed with roughly the same overall efficiency, and to do the same kind of comparison. There has been some work done toward improving the efficiency of the Jaro-Winkler algorithm by attempting to minimize the reduction ratio involved in the algorithm by using complicated filters to discard similarity computations. This could be researched further but overall, the algorithm is still suited mainly for short inputs.

## Hunt-McIlroy Distance

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.

## Needleman-Wunsch Distance

Another string comparison method was developed by Saul B Needleman and Christian D Wunsch around 1970. This method has long been used in biometrics to find the best alignment of string sequences. In the article written by Arthur M. Lesk for Encyclopedia Britannica, he describes the Needleman-Wunsch method 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 either a match or a mismatch. Also, there is a way to allow identification of gaps in sequences, known as a gap penalty, which can vary depending on the length of inputs and expected differences. 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 generated in two parts: construction of the alignment matrix and the recursive logic to look at the alignment. First, we create the number of rows and columns of the matrix using the value of the possible decimal values of the ASCII table. Then we fill the matrix with by comparing the row number and column numbers for match/mismatch. An example could be where row 1 and column 1 are equal 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 measures a penalty for the amount of unmatched characters 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);

}

With a filled-in grid, the recursion is done to find the optimal alignment sequence to the inputs, as demonstrated by the following pseudo code:

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 maximum score returned by using values associated with the match, mismatch, and gap penalty values.

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(), given equal length inputs..

The Levenshtein’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 Levenshtein'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, Levenshtein’s distance measurement is more efficient. But the role of the Needleman-Wunsch is better utilized when working with smaller inputs, as the adjustments for the scoring mechanism (match, mismatch, and gap penalty) can potentially create a more visible separation between the different inputs being compared.

# Conclusion

As shown herein, we believe that the Levenshtein’s Distance method may still be the optimal choice for comparing larger data such as manuscript drafts. All of the algorithms have high order of complexity, where each character is compared with other nearby characters if not all others in the converse string. As for using any of these alternate algorithms to determine the sequence in which the drafts were created, we have not found evidence to support any claims that the ordering assumed by our raw results could be of value. Taking it to that level would require further assumptions in the interpretation and likely larger inputs as to avoid confusing results. The Needleman-Wunsch results proved to be the closest match to Levenshtein’s, and at times provided a better distance value, but did not perform better overall. Of all the algorithms we tested, only the Hunt-McIlroy method was intended to be used with large inputs, and though it was designed for comparing larger files line by line, it was intended to solve a different kind of problem.

To continue this project, we would recommend that further research should be done in regard to other more time-efficient algorithms, and the selection of larger input data. In any case, algorithms that are easily parallelizable would be preferable. After a short look into other algorithms based on these findings, algorithms developed in more recent years, such as Minkowski Distance or trigrams would likely return results in less time and be better equipped to handle larger data sets.

References

1. P. Black, *Dictionary of Algorithms and Data Structures.* Nat. Inst. of Stds. and Technol., [Online]. Available: https://xlinux.nist.gov/dads/
2. K. Dreßler and A. Ngonga, “On the Efficient Execution of Bounded Jaro-Winkler Distances,” presented at Proc. of Ont. Matching Workshop. [Online]. Available: http://www.semantic-web-journal.net/system/files/swj944.pdf
3. J. W. Hunt and M. D. McIlroy, “An Algorithm for Differential File Comparison,” *Comput. Sci. Tech. Rep.,* vol. 41, Bell Laboratories, June 1976. [Online]. Available: http://www.cs.dartmouth.edu/~doug/diff.pdf
4. S. B. Needleman and C. D Wunsch, “A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins.” *J. of Molecular Biol.*, vol. 43, no. 8,pp. 443-453, 1970. [doi:10.1016/0022-2836(70)90057-4](http://dx.doi.org/10.1016/0022-2836(70)90057-4)
5. K. Mayfield and C. Alberts, “GRALT: Graphical Analyzer of Literary Text Versions,” *N. Dakota Quart.,* vol. 79, no. 1, pp. 71-78. Winter 2012.

1. This work was sponsored by Katia Mayfield, Athens State University, Athens, AL. It was submitted for review by her and Dr. Adam W. Lewis, Professors of Computer Science, on November 19, 2015.

   K. T. Woods is a member of Phi Theta Kappa and graduating senior in Computer Science – Networking at Athens State University, Athens, AL 35611 USA. (e-mail: kwoods3@my.athens.edu).

   M. M. Williams is a member of Kappa Mu Epsilon, Math and Science Club and graduating senior in Computer Science and Mathematics at Athens State University, Athens, AL 35611 USA. (e-mail: mwilli29@my.athens.edu).

   W. G. Dawkins is a graduating senior in Computer Science – Computer Information Systems at Athens State University, Athens, AL 35611 USA.

   (e-mail: wdawkins@my.athens.edu). [↑](#footnote-ref-1)