Quasilinear-Time Search and Comparison for Sequential Data

ILHAN KARIĆ, ZANIN VEJZOVIĆ
Faculty of Information Technologies
University of "Džemal Bijedić"
Sjeverni logor bb 88104 Mostar
Bosnia and Herzegovina
ilhan.karic@edu.fit.ba, zanin@edu.fit.ba
http://www.fit.ba/

Abstract: - This paper proposes a new algorithm for the evaluation of similarity between two sequences in quasilinear time. It describes the theoretical, practical and implementational aspects of the algorithm. The proposed method is a new approach dedicated to the computation of sequential similarity in contrast to other methods like the Jaccard Index which although designed for the computation of similarity of sets have been frequently used on sequences. The method is generalizable and applicable to any form of sequential data of a finite alphabet (binary files, DNA sequences, natural language etc.)

Key-Words: - Sequence Similarity, Comparison, Contextual Similarity, Quasilinear-Time Complexity

1 Introduction

Evaluating the similarity of two sequences is a standard computer science problem and has a wide area of use. A great deal of applications, from search engines to document ranking, from gene finding to prediction of protein functions, from network surveillance tools to anti-virus programs critically depend on analysis of sequential data. [1]

We will describe the matching of two sequences as well as the search process for the closest matching sequence from a pool of sequences which appeared to be the bottleneck of other methods like the Jaccard Index. We are also going to explain the naïve implementation which would directly follow from the mathematical model and optimizations, for which we have provided proof in the same paper, which will reduce the time complexity for both sequence comparison and sequence matching down to quasilinear time. In the end we will provide our benchmarks with data collected during the research.

2 The Problem

The main performance issue with sequence matching proved to be the fact that known methods didn't perform well on measurements made on the union of two or more sequences which are not directly correlated. This is mainly because those methods were designed to evaluate the similarity of sets, not sequences, meaning that the data must be either sorted in some way or split up into smaller logical chunks to reduce the time required to find the closest match. Even if the initial similarity function was linear, the search for the closest match would have to be quadratic because the function would have to be performed for each set individually. The **contextual similarity** function proved to be of quasilinear-time complexity for both comparison and matching of sequences.

3 The Function

The basic idea behind the concept lies in one of the most important properties of a sequence, the order of items within it. The second property, the content, will be awarded in a way where it won't matter as much as the order or position where it is located. The best idea is to imagine the initial sequence as a set of characters (sentence) and the second as a text. The goal is to score how relevant the text is given the first sequence of characters. From this point on, the first sequence (sentence) will be denoted as N and the second (the text) as M.

The probability of N being found as a subsequence of M due to sheer coincidence (without having any contextual relation with it) decreases exponentially as the length of N increases. We will apply this as a heuristic although it has some deep roots in linguistics due to the power law and the Zipfian distribution. [2] This means that we're going to award the appearance of a subsequence of N within M based on the length of the subsequence, naming it **shared context**:

$$\delta(N,M) = \sum_{i=1}^{|N|}\sum_{j=i}^{|N|}\left|N_{ij} \cap M\right| \qquad (1)$$

We are summing up the lengths (cardinalities) of all possible subsequences of N found in M thus rewarding the position and order rather than the actual number of matches. We don't pay attention to how many times a certain subset occurs to avoid "is", "the", "a" and similar words to add up on quantity rather than the way they create contextual meaning.

The problem with this measurement is that it has no upper bound thus grows infinitely. This would result in sequences which are generally larger to also be "more similar" than sequences with a smaller length which is not true. To solve this problem we must normalize our function by dividing with the coefficient T_σ which represents the total score that can be achieved under the assumption that the first sequence is a proper subsequence of the second, meaning that $N \subseteq M$. In this case we can compute T_σ directly as shown in (2) below:

$$T_\sigma = \sum_{k=1}^{n}\frac{k(k+1)}{2} = \frac{n(n+1)(n+2)}{3!} \qquad (2)$$

Assuming that n is replaced by the length of our first sequence N we can express our normalization coefficient as the following binomial:

$$T_\sigma = \binom{|N|+2}{3} \qquad (3)$$

Now, once we have normalized our initial function (1) we arrive at the final expression for the **contextual similarity**:

$$\bar{\delta}(N,M) = \frac{1}{T_\sigma}\sum_{i=1}^{|N|}\sum_{j=i}^{|N|}\left|N_{ij} \cap M\right| \qquad (4)$$

This is the normalized function and measures the contextual relation between two sequences. The domain of the function (4) is $0 \le \bar{\delta} \le 1$ where the similarity (and probability of non-random relationship between the two sequences) is increasing with $\bar{\delta}$ respectively. When the entire first sequence is a subsequence of M, the contextual similarity $\bar{\delta} = 1$. In the case where the two squences are disjunctive, $\bar{\delta} = 0$.

4 Optimizations

We will stop to review the naïve implementation which would follow directly from the mathematical model. The **first** thing to notice is that if our second sequence M does not contain the subsequence N_{ij} there is no need to check for $N_{i(j+1)}$ For instance, if N = "ABC" and M = "AXBC" we will find that M does not contain "AB" thus it is redundant to check for "ABC" and we can stop further computation on the given subsequence of N.

The **second** optimization addresses the problem where $N = M$ or in general when N contains long subsequences of M resulting in many nested iterations. If we look closely we will notice that our normalization coefficient can be used in a way which will lead to us avoiding re-doing done work. Consider the following example where N = "AAAAxA" and M = "AAAAAA". Once we've evaluated the sum of all lengths of all subsequences of N up to "x" ("AAAA") we can agree that repeating the same process for ("AAA") is a waste because we already know that they exist within M so we can skip that part and add $\binom{|N|-1+2}{3}$ directly. This will further reduce the gaps between the **worst**, **average** and **best** cases.

The **third** optimization would be to use the unique property of the **contextual similarity** function that allows us to review two or more sequences at once by concatenating them. These sequences can be picked at random and don't have to be correlated, meaning that we could merge four sequences into two super-sequences and review two sequences at once. Note that we should add one character as a separator to avoid creating subsequences which weren't there initially. For this we should use a character which is not part of the alphabet. Now, we can compare our initial sequence N with, for example, two super-sequences $M_{p1} \cup M_{q1}$ and $M_{p2} \cup M_{q2}$ in two computations rather than four. The super-sequence which scores more has an increased probability of one of its subsequences to be related to N. Once we decide which super-sequence probably contains our information, we can slice it in two halves and repeat the process until only one item is left. To keep the information about the initial groups we will use a list where each item is a known sequence M and only do grouping logically, based on indexes, performing a context-driven dichotomic search. This way, instead of doing 1024 computations for 1024 sequences we only have to do $2\log_2(1024) = 20$

5 Theoretical Complexity

In this section we're going to discuss the **worst**, **best** and **average** time complexity of the **optimized** algorithm.

The **worst case** occurs when the two strings we are comparing are equal. In this case the outer loop would only iterate once (due to the **second** optimization) and the inner loop would iterate **N** times. For this calculation, we're assuming the Boyer-Moore string search algorithm which is known to have a linear average time complexity. This would give us the **worst case** time complexity of $O(1 * n * n) = O(n^2)$ so we can say that our algorithm, in the worst case, will execute in quadratic time.

The **best case** occurs when the two strings that we are comparing are disjunctive thus the outer loop would iterate **N** times while the inner loop would iterate once per outer iteration and the string search operation would be performed on a single character every time giving us the **best case** time complexity of: $O(n * 1 * 1) = O(n)$

The **average case**, unlike the previous two, is more difficult to evaluate. Since we don't know how the input will look like, we're assuming two random sequences each of length **N**. The actual character at any position is picked uniformly at random from an alphabet of size **q**. The probability of picking any specific character is $p = \frac{1}{q}$. By picking two fixed positions, we can say that the probability for **m** consecutive matches, starting at position **i** in the first sequence and starting at position **j** in the second sequence is p^m.

Let $x_{ij}(m)$ be a random variable with the value 1 if starting at position **i** and position **j** there is a match of length **m** and 0 otherwise. It takes on 1 with the probability p^m and 0 with the probability $1 - p^m$. Counting all matches we can say that:

$$C_m \equiv \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}(m) \quad (5)$$

Remember that C_m represents the total number of pairs with starting positions **i**, **j** that have a match of length **m**. We're interested in the expected average value $E(C_m)$:

$$E\left(\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}(m)\right) \quad (6)$$

Due to the linearity of expectation we can take the sum of the expected values instead of the expectation of the sum so we can define $E(C_m)$ as:

$$E(C_m) = \sum_{i=1}^{n} \sum_{j=1}^{n} E\left(x_{ij}(m)\right) \quad (7)$$

If we look at the expectation of $x_{ij}(m)$ we can see that: $E\left(x_{ij}(m)\right) = 1 * p^m + 0 * (1 - p^m) = p^m$. We can finally express the expected number of matches in two random sequences as shown in (8):

$$E(C_m) = \sum_{i=1}^{n} \sum_{j=1}^{n} p^m = n^2 p^m \quad (8)$$

The assumption is that $n \gg m$ which means that we don't have to account for different upper limits of the two sums. This is how the algorithm was designed to work so it is a good approximation for the average case. We're interested in how $E(C_m)$ behaves asymptotically as **m** increases.

Let **r** be the largest value of **m** such that $E(C_m) \geq 1$. Knowing what **r** is will give us a good heuristic insight into knowing what the expected longest common substring is. We arrive at the following inequality:

$$n^2 p^r \geq 1 \quad (9)$$

Since **r** is the largest value of **m** such that this holds, we must also say that $n^2 p^{(r+1)} < 1$. Once rearranged and further simplified, we express the probability **p** in function of the alphabet size **q** as initially and arrive at the following approximation:

$$log_q(n-1) < r \leq log_q(n) \quad (10)$$

From this approximation we can learn about the asymptotic behaviour of the expected length of the longest common substring as the length of the string **n** increases. A simulation has been created to help visualise our mathematical predictions which can be seen in Fig. 1. We have created pairs of randomly generated sequences of the same alphabet size and measured the longest common substring length in function of the lengths of the two sequences. Each data point on the scatterplot in Fig. 1. was computed as the arithmetic mean over 1000 iterations in order to rule out random occurances as much as possible. We can see the results of this simulation in the scatterplot below:

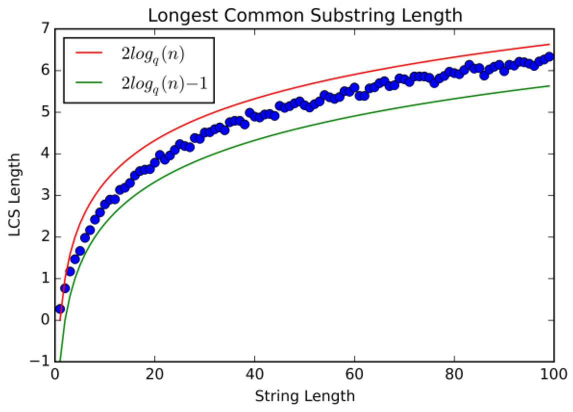

Fig. 1. The asymptotic behavior of the growth of the longest common substrings in function of string size.

We can see our upper and lower bounds which give us an insight into where we expect to find our random values.

With this approximation proven, we can finally compute the **average time complexity** of our algorithm. To simplify and leave some room for error, we're going to take out the **second** optimization from our evaluation. This means that the outer loop structure will always iterate **N** times while the inner loop, due to the **first** optimization, only iterates in function of the longest common substring which was just proven to grow logarithmically. Since the string matching algorithm inside the second loop only ever operates on the longest common substrings of length $\log(N)$ we can say that the **average time complexity** of our algorithm is: $O(n * 2\log n * \log n) = O(n\log^2 n)$

It is **important** to note that our method allows for measurments on the union of two or more sequences. Since we know that binary search has a known average time complexity of $\log(n)$ the complexity of comparing our sequence to all existing sequences is: $O(n\log^2 n\log n) = O(n\log^3 n)$ which is still quasilinear. In contrast, other methods such as the **Jaccard Index** would perform search operations in **quadratic** time thus the proposed method is faster.

Another thing to note is that the reason why our assumption of random strings would be a good representative for the average case is due to the fact that both randomly generated and coherent text follow a Zipfian distribution. [2]

6 Benchmarks

We have measured only the **first** and **second** optimization impact on the general performance of the algorithm. The **third** optimization was always enabled to speed things up. The first three tests have been ran for all three cases (with no optimization, first optimization and second optimization enabled). The first sequence N was always populated with 33 characters and each list item M was exactly 33 characters long. The list was filled with 2, 4, 8, 16.. 32768 items where every item was 33 characters long. The time was measured on each iteration.

The **first test** populated the list with items which would yield a very high similarity when evaluated. The **second test** generated a list of random items each 33 characters long, to test the performance against random similarity. The **third test** populated the list with items that would yield a low similarity on average.

Our theory predicted that the first optimization would reduce the execution time when the similarity was high (worst case) and that our second optimization would further reduce the speed gap between the worst and best case (high and low similarity). The results only show how each optimization changed the algorithm performance in the case of high, random and low similarity in data. In the end the goal was to decrease the speed on data sets where the similarity is high or low at the cost of increasing the time required to process purely random data. The algorithm was designed to perform on data sets where we expect to have some sort of similarity either low or high. Purely random data, by definition, does not contain any useful information meaning that we give our algorithm the best chances of performing in real life cases. The figures 2, 3 and 4 show the results of the tests.

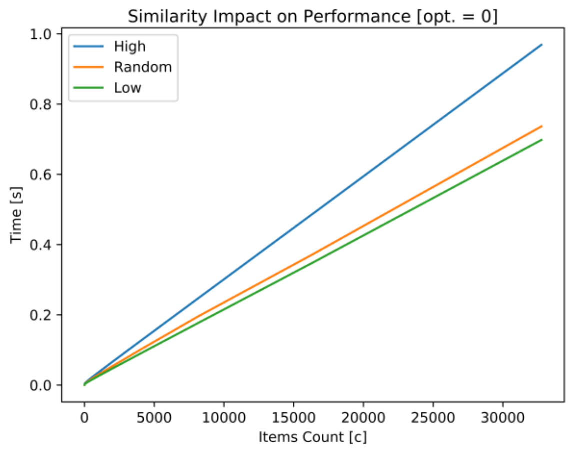

Fig. 2. Algorithm search performance in the case of low, random and high similarity data, without optimizations.

Remember that Fig. 2. shows time required to find the best matching sequence and not only compare two sequences which still appears to be linear. This is due to the large scope of the x axis. If we zoom in to the beginning we can see a logarithmic curve.

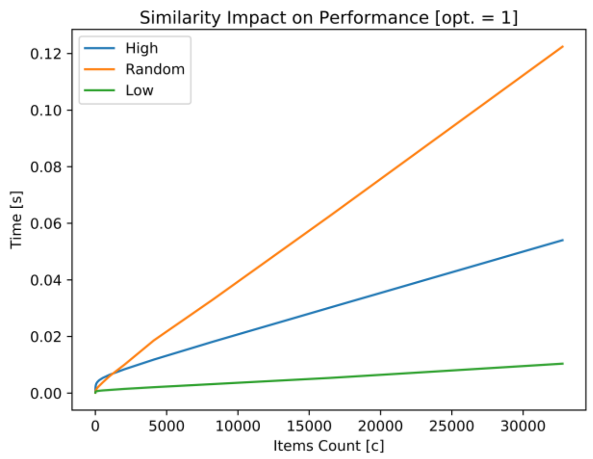

Fig. 3. Algorithm search performance in the case of low, random and high similarity data, with only the first optimization enabled.

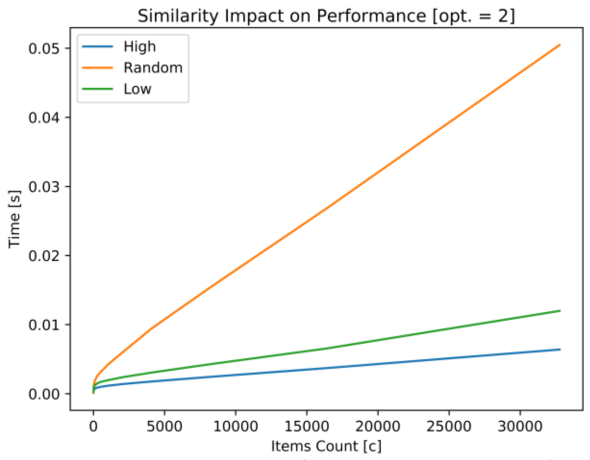

Fig. 4. Algorithm search performance in the case of low, random and high similarity data, with only the second optimization enabled.

The last two tests have been designed to test the asymptotic behavior of the contextual similarity function in function of the input size and search area. Fig.5 and Fig.6 show the results of the named tests.

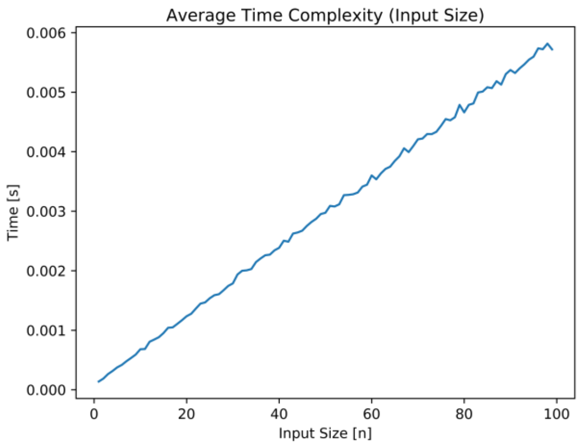

Fig. 5. Asymptotic behavior of the average time execution in function of the length of the first sequence.

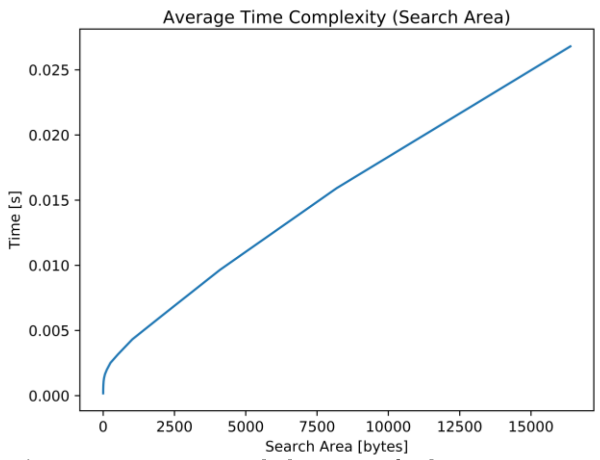

Fig. 6. Asymptotic behavior of the average time execution in function of the number of sequences to search through. All sequences have been generated randomly.

7 Conclusion

We conclude that our algorithm both from the theoretical and practical perspective is indeed quasilinear in nature. Notice how other similarity functions such as the **Jaccard Index**, **Cosine Similarity** or **Sørensen–Dice** are different. The **Jaccard Index**, for instance, can only operate on two sets at a time meaning that to calculate the similarity for **n** sets one would have to perform n^2 Jaccard calculations, which would result in **quadratic time** complexity. The proposed method provides a unique scoring which awards position, order and structure of the sequence. The algorithm allows for a unique way of performing a **dichotomic search** over inherently random and unsorted arrays of data which allows us to quickly search, find and match sequences based on their similarity.

References:

[1] X1. Konrad Rieck, Pavel Laskov, “Linear-Time Computation of Similarity Measures for Sequential Data”, Journal of Machine Learning Research 9 (2008) 23-48 pp. 1

[2] S. T. Piantadosi, “Zipf’s word frequency law in natural language: A critical review and future directions”, Psychonomic Bulletin & Review, vol. 21, 2014, pp. 1112-1130